AN INTRODUCTION TO CP/M FEATURES AND FACILITIES
Copyright (c) 1979 by Exidy Inc. All Rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronics, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Exidy Inc., 390 Java Drive, Sunnyvale, California, 94086.

Disclaimer

Exidy Inc. makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Exidy Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Exidy Inc. to notify any person of such revision or changes.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. FUNCTIONAL DESCRIPTION OF CP/M</td>
<td>3</td>
</tr>
<tr>
<td>2.1. General Command Structure</td>
<td>3</td>
</tr>
<tr>
<td>2.2. File References</td>
<td>3</td>
</tr>
<tr>
<td>3. SWITCHING DISKS</td>
<td>6</td>
</tr>
<tr>
<td>4. THE FORM OF BUILT-IN COMMANDS</td>
<td>7</td>
</tr>
<tr>
<td>4.1. ERA afn cr</td>
<td>7</td>
</tr>
<tr>
<td>4.2. DIR afn cr</td>
<td>8</td>
</tr>
<tr>
<td>4.3. REN ufn1=ufn2 cr</td>
<td>8</td>
</tr>
<tr>
<td>4.4. SAVE n ufn cr</td>
<td>9</td>
</tr>
<tr>
<td>4.5. TYPE ufn cr</td>
<td>9</td>
</tr>
<tr>
<td>5. LINE EDITING AND OUTPUT CONTROL</td>
<td>11</td>
</tr>
<tr>
<td>6. TRANSIENT COMMANDS</td>
<td>12</td>
</tr>
<tr>
<td>6.1. STAT cr</td>
<td>13</td>
</tr>
<tr>
<td>6.2. ASM ufn cr</td>
<td>16</td>
</tr>
<tr>
<td>6.3. LOAD ufn cr</td>
<td>17</td>
</tr>
<tr>
<td>6.4. PIP cr</td>
<td>18</td>
</tr>
<tr>
<td>6.5. ED ufn cr</td>
<td>25</td>
</tr>
<tr>
<td>6.6. SYSGEN cr</td>
<td>27</td>
</tr>
<tr>
<td>6.7. SUBMIT ufn parm#1 ... parm#n cr</td>
<td>28</td>
</tr>
<tr>
<td>6.8. DUMP ufn cr</td>
<td>30</td>
</tr>
<tr>
<td>6.9. MOVCPM cr</td>
<td>30</td>
</tr>
<tr>
<td>7. BDOS ERROR MESSAGES</td>
<td>33</td>
</tr>
<tr>
<td>8. OPERATION OF CP/M ON THE MDS</td>
<td>34</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

CP/M is a monitor control program for microcomputer system development which uses IBM-compatible flexible disks for backup storage. Using a computer mainframe based upon Intel's 8080 microcomputer, CP/M provides a general environment for program construction, storage, and editing, along with assembly and program check-out facilities. An important feature of CP/M is that it can be easily altered to execute with any computer configuration which uses an Intel 8080 (or Zilog Z-80) Central Processing Unit, and has at least 16K bytes of main memory with up to four IBM-compatible diskette drives. A detailed discussion of the modifications required for any particular hardware environment is given in the Exidy Inc. document entitled "CP/M System Alteration Guide". Although the standard Exidy version operates on a single-density Intel MDS 800, several different hardware manufacturers support their own input-output drivers for CP/M.

The CP/M monitor provides rapid access to programs through a comprehensive file management package. The file subsystem supports a named file structure, allowing dynamic allocation of file space as well as sequential and random file access. Using this file system, a large number of distinct programs can be stored in both source and machine executable form.

CP/M also supports a powerful context editor.

CP/M is logically divided into several distinct parts:

- **BIOS** Basic I/O System (hardware dependent)
- **BDOS** Basic Disk Operating System
- **CCP** Console Command Processor
- **TPA** Transient Program Area

The BIOS provides the primitive operations necessary to access the diskette drives and to interface standard peripherals (TTY, CRT, paper tape reader/punch, and user-defined peripherals), and can be tailored by the user to any particular hardware environment by "patching" this portion of CP/M. The BDOS provides disk management by controlling one or more disk drives containing independent file directories. The BDOS implements disk allocation strategies which provide fully dynamic file construction while minimizing head movement across the disk during access. Any particular file may contain any number of records, not exceeding the size of any single disk. In a standard CP/M system, each disk can contain up to 64 distinct files. The
BDOS has entry points which include the following primitive operations which can be programmatically accessed:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEARCH</td>
<td>Look for a particular disk file by name.</td>
</tr>
<tr>
<td>OPEN</td>
<td>Open a file for further operations.</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Close a file after processing.</td>
</tr>
<tr>
<td>RENAME</td>
<td>Change the name of a particular file.</td>
</tr>
<tr>
<td>READ</td>
<td>Read a record from a particular file.</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write a record onto the disk.</td>
</tr>
<tr>
<td>SELECT</td>
<td>Select a particular disk drive for further operations.</td>
</tr>
</tbody>
</table>

The CCP provides symbolic interface between the user’s console and the remainder of the CP/M system. The CCP reads the console device and processes commands which include listing the file directory, printing the contents of files, and controlling the operation of transient programs, such as assemblers, editors, and debuggers. The standard commands which are available in the CCP are listed in a following section.

The last segment of CP/M is the area called the Transient Program Area (TPA). The TPA holds programs which are loaded from the disk under command of the CCP. During program editing, for example, the TPA holds the CP/M text editor machine code and data areas. Similarly, programs created under CP/M can be checked out by loading and executing these programs in the TPA.

It should be mentioned that any or all of the CP/M component subsystems can be "overlayed" by an executing program. That is, once a user's program is loaded into the TPA, the CCP, BDOS, and BIOS areas can be used as the program's data area. A "bootstrap" loader is programmatically accessible whenever the BIOS portion is not overlayed; thus, the user program need only branch to the bootstrap loader at the end of execution, and the complete CP/M monitor is reloaded from disk.

It should be reiterated that the CP/M operating system is partitioned into distinct modules, including the BIOS portion which defines the hardware environment in which CP/M is executing. Thus, the standard system can be easily modified to any non-standard environment by changing the peripheral drivers to handle the custom system.
2. FUNCTIONAL DESCRIPTION OF CP/M.

The user interacts with CP/M primarily through the CCP, which reads and interprets commands entered through the console. In general, the CCP addresses one of several disks which are online (the standard system addresses up to four different disk drives). These disk drives are labelled A, B, C, and D. A disk is "logged in" if the CCP is currently addressing the disk. In order to clearly indicate which disk is the currently logged disk, the CCP always prompts the operator with the disk name followed by the symbol ">" indicating that the CCP is ready for another command. Upon initial start up, the CP/M system is brought in from disk A, and the CCP displays the message

```
xxK CP/M VER m.m
```

where xx is the memory size (in kilobytes) which this CP/M system manages, and m.m is the CP/M version number. All CP/M systems are initially set to operate in a 16K memory space, but can be easily reconfigured to fit any memory size on the host system (see the MOVCPM transient command). Following system signon, CP/M automatically logs in disk A, prompts the user with the symbol "A>" (indicating that CP/M is currently addressing disk "A"), and waits for a command. The commands are implemented at two levels: built-in commands and transient commands.

2.1. GENERAL COMMAND STRUCTURE.

Built-in commands are a part of the CCP program itself, while transient commands are loaded into the TPA from disk and executed. The built-in commands are

- **ERA**: Erase specified files.
- **DIR**: List file names in the directory.
- **REN**: Rename the specified file.
- **SAVE**: Save memory contents in a file.
- **TYPE**: Type the contents of a file on the logged disk.

Nearly all of the commands reference a particular file or group of files. The form of a file reference is specified below.

2.2. FILE REFERENCES.

A file reference identifies a particular file or group of files on a particular disk attached to CP/M. These file references can be either "unambiguous" (ufn) or "ambiguous" (afn). An unambiguous file reference uniquely identifies a single file, while an ambiguous file reference may be
satisfied by a number of different files.

File references consist of two parts: the primary name and the secondary name. Although the secondary name is optional, it usually is generic; that is, the secondary name "ASM," for example, is used to denote that the file is an assembly language source file, while the primary name distinguishes each particular source file. The two names are separated by a "." as shown below:

```
ppppppppp.sss
```

where ppppppppp represents the primary name of eight characters or less, and sss is the secondary name of no more than three characters. As mentioned above, the name

```
ppppppppp
```

is also allowed and is equivalent to a secondary name consisting of three blanks. The characters used in specifying an unambiguous file reference cannot contain any of the special characters

```
< > . , ; : = ? * [ ]
```

while all alphanumerics and remaining special characters are allowed.

An ambiguous file reference is used for directory search and pattern matching. The form of an ambiguous file reference is similar to an unambiguous reference, except the symbol "?" may be interspersed throughout the primary and secondary names. In various commands throughout CP/M, the "?" symbol matches any character of a file name in the "?" position. Thus, the ambiguous reference

```
x?Z.COM
```

is satisfied by the unambiguous file names

```
XYZ.COM
```

and

```
xZ,CAM
```

Note that the ambiguous reference

```
*.
```

is equivalent to the ambiguous file reference

```
?????????.???
```

while
and

*,sss

are abbreviations for

pppppppp.???
and

?????????.sss

respectively. As an example,

DIR *.*

is interpreted by the CCP as a command to list the names of all disk files in the directory, while

DIR X.Y

searches only for a file by the name X.Y Similarly, the command

DIR X?Y.C?M

causes a search for all (unambiguous) file names on the disk which satisfy this ambiguous reference.

The following file names are valid unambiguous file references:

X
XYZ
GAMMA
X.Y
XYZ.COM
GAMMA.1

As an added convenience, the programmer can generally specify the disk drive name along with the file name. In this case, the drive name is given as a letter A through Z followed by a colon (:). The specified drive is then "logged in" before the file operation occurs. Thus, the following are valid file names with disk name prefixes:

A:X.Y
B:XYZ
C:GAMMA
Z:XYZ.COM
B:X.A?M
C:*..ASM

It should also be noted that all alphabetic lower case letters in file and drive names are always translated to upper case when they are processed by the CCP.
3. SWITCHING DISKS.

The operator can switch the currently logged disk by typing the disk drive name (A, B, C, or D) followed by a colon (:) when the CCP is waiting for console input. Thus, the sequence of prompts and commands shown below might occur after the CP/M system is loaded from disk A:

16K CP/M VER 1.4

A>DIR List all files on disk A.
SAMPLE ASM
SAMPLE PRN
A>B: Switch to disk B.
B>DIR *.ASM List all "ASM" files on B.
DUMP ASM
FILES ASM
B>A: Switch back to A.
4. THE FORM OF BUILT-IN COMMANDS.

The file and device reference forms described above can now be used to fully specify the structure of the built-in commands. In the description below, assume the following abbreviations:

- ufn - unambiguous file reference
- afn - ambiguous file reference
- cr - carriage return

Further, recall that the CCP always translates lower case characters to upper case characters internally. Thus, lower case alphabetics are treated as if they are upper case in command names and file references.

4.1 ERA afn cr

The ERA (erase) command removes files from the currently logged-in disk (i.e., the disk name currently prompted by CP/M preceding the ">"). The files which are erased are those which satisfy the ambiguous file reference afn. The following examples illustrate the use of ERA:

- **ERA X.Y**
  The file named X,Y on the currently logged disk is removed from the disk directory, and the space is returned.

- **ERA X.***
  All files with primary name X are removed from the current disk.

- **ERA *.ASM**
  All files with secondary name ASM are removed from the current disk.

- **ERA X?Y.C?M**
  All files on the current disk which satisfy the ambiguous reference X?Y,C?M are deleted.

- **ERA **
  Erase all files on the current disk (in this case the CCP prompts the console with the message "ALL FILES (Y/N)?" which requires a Y response before files are actually removed).

- **ERA B:*.*PRN**
  All files on drive B which satisfy the ambiguous reference ???????????.*,PRN are deleted, independently of the currently logged disk.
4.2. DIR afn cr

The DIR (directory) command causes the names of all files which satisfy
the ambiguous file name afn to be listed at the console device. As a special
case, the command

    DIR

lists the files on the currently logged disk (the command "DIR" is equivalent
to the command "DIR *.*"). Valid DIR commands are shown below.

    DIR X,Y
    DIR X?Z.C?M
    DIR ??Y

Similar to other CCP commands, the afn can be preceded by a drive name.
The following DIR commands cause the selected drive to be addressed before the
directory search takes place.

    DIR B:
    DIR B:X,Y
    DIR B:*.*,A?M

If no files can be found on the selected diskette which satisfy the
directory request, then the message "NOT FOUND" is typed at the console.

4.3. REN ufn1=ufn2 cr

The REN (rename) command allows the user to change the names of files on
disk. The file satisfying ufn2 is changed to ufn1. The currently logged disk
is assumed to contain the file to rename (ufn1). The CCP also allows the user
to type a left-directed arrow instead of the equal sign, if the user's console
supports this graphic character. Examples of the REN command are

    REN X,Y=Q,R       The file Q,R is changed to X,Y.
    REN XYZ.COM=XYZ.XXX The file XYZ.XXX is changed to XYZ.COM.

The operator can precede either ufn1 or ufn2 (or both) by an optional
drive address. Given that ufn1 is preceded by a drive name, then ufn2 is
assumed to exist on the same drive as ufn1. Similarly, if ufn2 is preceded by
a drive name, then ufn1 is assumed to reside on that drive as well. If both
ufn1 and ufn2 are preceded by drive names, then the same drive must be
specified in both cases. The following REN commands illustrate this format.

REN A:X.ASM = Y.ASM    The file Y.ASM is changed to X.ASM on drive A.
REN B:ZAP.BAS = ZOT.BAS The file ZOT.BAS is changed to ZAP.BAS on drive B.
REN B:A.ASM = A.BAK    The file A.BAK is renamed to A.ASM on drive B.

If the file ufn1 is already present, the REN command will respond with the error "FILE EXISTS" and not perform the change. If ufn2 does not exist on the specified diskette, then the message "NOT FOUND" is printed at the console.

4.4. SAVE n ufn cr

The SAVE command places n pages (256-byte blocks) onto disk from the TPA and names this file ufn. In the CP/M distribution system, the TPA starts at 100H (hexadecimal), which is the second page of memory. Thus, if the user’s program occupies the area from 100H through 2FFH, the SAVE command must specify 2 pages of memory. The machine code file can be subsequently loaded and executed. Examples are:

SAVE 3 X.COM          Copies 100H through 3FFH to X.COM.
SAVE 40 Q             Copies 100H through 28FFH to Q (note that 28 is the page count in 28FFH, and that 28H = 2*16 + 8 = 40 decimal).
SAVE 4 X.Y            Copies 100H through 4FFH to X.Y.

The SAVE command can also specify a disk drive in the afd portion of the command, as shown below.

SAVE 10 B:ZOT.COM     Copies 10 pages (100H through 8AFFH) to the file ZOT.COM on drive B.

4.5. TYPE ufn cr

The TYPE command displays the contents of the ASCII source file ufn on the currently logged disk at the console device. Valid TYPE commands are

TYPE X,Y
TYPE X.PLN

TYPE XXX

The TYPE command expands tabs (ctl-I characters), assuming tab positions are set at every eighth column. The ufn can also reference a drive name as shown below.

TYPE B:X.PRN                   The file X.PRN from drive B is displayed.
5. LINE EDITING AND OUTPUT CONTROL.

The CCP allows certain line editing functions while typing command lines.

rubout     Delete and echo the last character typed at the console.
ctl-U      Delete the entire line typed at the console.
ctl-X      (Same as ctl-U)
ctl-R      Retype current command line: types a "clean line" following character deletion with rubouts.
ctl-E      Physical end of line: carriage is returned, but line is not sent until the carriage return key is depressed.
ctl-C      CP/M system reboot (warm start)
ctl-Z      End input from the console (used in PIP and ED).

The control functions ctl-P and ctl-S affect console output as shown below.

ctl-P      Copy all subsequent console output to the currently assigned list device (see the STAT command). Output is sent to both the list device and the console device until the next ctl-P is typed.

ctl-S      Stop the console output temporarily. Program execution and output continue when the next character is typed at the console (e.g., another ctl-S). This feature is used to stop output on high speed consoles, such as CRT's, in order to view a segment of output before continuing.

Note that the ctl-key sequences shown above are obtained by depressing the control and letter keys simultaneously. Further, CCP command lines can generally be up to 255 characters in length; they are not acted upon until the carriage return key is typed.
6. TRANSIENT COMMANDS.

Transient commands are loaded from the currently logged disk and executed in the TPA. The transient commands defined for execution under the CCP are shown below. Additional functions can easily be defined by the user (see the LOAD command definition).

STAT       List the number of bytes of storage remaining on the currently logged disk, provide statistical information about particular files, and display or alter device assignment.

ASM        Load the CP/M assembler and assemble the specified program from disk.

LOAD       Load the file in Intel "hex" machine code format and produce a file in machine executable form which can be loaded into the TPA (this loaded program becomes a new command under the CCP).

DDT        Load the CP/M debugger into TPA and start execution.

PIP        Load the Peripheral Interchange Program for subsequent disk file and peripheral transfer operations.

ED         Load and execute the CP/M text editor program.

SYSGEN     Create a new CP/M system diskette.

SUBMIT     Submit a file of commands for batch processing.

DUMP       Dump the contents of a file in hex.

MOVCPM     Regenerate the CP/M system for a particular memory size.

Transient commands are specified in the same manner as built-in commands, and additional commands can be easily defined by the user. As an added convenience, the transient command can be preceded by a drive name, which causes the transient to be loaded from the specified drive into the TPA for execution. Thus, the command

B:STAT

causes CP/M to temporarily "log in" drive B for the source of the STAT transient, and then return to the original logged disk for subsequent processing.
The basic transient commands are listed in detail below.

6.1. STAT cr

The STAT command provides general statistical information about file storage and device assignment. It is initiated by typing one of the following forms:

\[
\begin{align*}
\text{STAT cr} \\
\text{STAT "command line" cr}
\end{align*}
\]

Special forms of the "command line" allow the current device assignment to be examined and altered as well. The various command lines which can be specified are shown below, with an explanation of each form shown to the right.

\[
\begin{align*}
\text{STAT cr} & \quad \text{If the user types an empty command line, the STAT transient calculates the storage remaining on all active drives, and prints a message} \\
& \quad \text{x: R/W, SPACE: nnnK} \\
& \quad \text{or} \\
& \quad \text{x: R/O, SPACE: nnnK}
\end{align*}
\]

for each active drive x, where R/W indicates the drive may be read or written, and R/O indicates the drive is read only (a drive becomes R/O by explicitly setting it to read only, as shown below, or by inadvertently changing diskettes without performing a warm start). The space remaining on the diskette in drive x is given in kilobytes by nnn.

\[
\begin{align*}
\text{STAT x: cr} & \quad \text{If a drive name is given, then the drive is selected before the storage is computed. Thus, the command "STAT B:" could be issued while logged into drive A, resulting in the message} \\
& \quad \text{BYTES REMAINING ON B: nnnK}
\end{align*}
\]

\[
\begin{align*}
\text{STAT afn cr} & \quad \text{The command line can also specify a set of files to be scanned by STAT. The files which satisfy afn are listed in alphabetical order, with storage requirements for each file under the heading} \\
& \quad \text{RECS BYTS EX D:FILENAME,TYP} \\
& \quad \text{rrrr bbbK ee d:ppppppppp.sss}
\end{align*}
\]

where rrrr is the number of 128-byte records
allocated to the file, bbb is the number of kilo-
bytes allocated to the file (bbb=rerr*128/1024),
ee is the number of 16K extensions (ee=bbb/16),
d is the drive name containing the file (A,...?),
ppppppppp is the (up to) eight-character primary
file name, and sss is the (up to) three-character
secondary name. After listing the individual
files, the storage usage is summarized.

STAT x:afn cr

As a convenience, the drive name can be given
ahead of the afn. In this case, the specified
drive is first selected, and the form "STAT afn"
is executed.

STAT x:=R/O cr

This form sets the drive given by x to read-only,
which remains in effect until the next warm or
cold start takes place. When a disk is read-only,
the message

BDOS ERR ON x: READ ONLY

will appear if there is an attempt to write to
the read-only disk x. CP/M waits until a key
is depressed before performing an automatic warm
start (at which time the disk becomes R/W).

The STAT command also allows control over the physical to logical device
assignment (see the IOBYTE function described in the manuals "CP/M Interface
Guide" and "CP/M System Alteration Guide"). In general, there are four
logical peripheral devices which are, at any particular instant, each assigned
to one of several physical peripheral devices. The four logical devices are
named:

CON: The system console device (used by CCP
for communication with the operator)
RDR: The paper tape reader device
PUN: The paper tape punch device
LST: The output list device

The actual devices attached to any particular computer system are driven
by subroutines in the BIOS portion of CP/M. Thus, the logical RDR: device,
for example, could actually be a high speed reader, Teletype reader, or
cassette tape. In order to allow some flexibility in device naming and
assignment, several physical devices are defined, as shown below:
TTY: Teletype device (slow speed console)
CRT: Cathode ray tube device (high speed console)
BAT: Batch processing (console is current RDR:, output goes to current LST: device)
UC1: User-defined console
PTR: Paper tape reader (high speed reader)
URL: User-defined reader #1
UR2: User-defined reader #2
PTP: Paper tape punch (high speed punch)
UP1: User-defined punch #1
UP2: User-defined punch #2
LPT: Line printer
ULL: User-defined list device #1

It must be emphasized that the physical device names may or may not actually correspond to devices which the names imply. That is, the PTP: device may be implemented as a cassette write operation, if the user wishes. The exact correspondence and driving subroutine is defined in the BIOS portion of CP/M. In the standard distribution version of CP/M, these devices correspond to their names on the MDG 800 development system.

The possible logical to physical device assignments can be displayed by typing

STAT VAL: cr

The STAT prints the possible values which can be taken on for each logical device:

CON. = TTY:  CRT:  BAT:  UC1:
RDR. = TTY:  PTR:  URL:  UR2:
PUN. = TTY:  PTP:  UP1:  UP2:
LST. = TTY:  CRT:  LPT:  ULL:

In each case, the logical device shown to the left can take any of the four physical assignments shown to the right on each line. The current logical to physical mapping is displayed by typing the command

STAT DEV: cr
which produces a listing of each logical device to the left, and the current corresponding physical device to the right. For example, the list might appear as follows:

CON: = CRT:
RDR: = URF:
PUN: = PTP:
LST: = TTY:

The current logical to physical device assignment can be changed by typing a STAT command of the form

STAT ld1 = pd1, ld2 = pd2, ..., ldn = pdn cr

where ld1 through ldn are logical device names, and pd1 through pdn are compatible physical device names (i.e., ld1 and pd1 appear on the same line in the "VAL:" command shown above). The following are valid STAT commands which change the current logical to physical device assignments:

STAT CON:=CRT: cr
STAT PUN: = TTY:, LST:=LPT:, RDR:=TTY: cr

6.2. ASM ufn cr

The ASM command loads and executes the CP/M 8080 assembler. The ufn specifies a source file containing assembly language statements where the secondary name is assumed to be ASM, and thus is not specified. The following ASM commands are valid:

ASM X

ASM GAMMA

The two-pass assembler is automatically executed. If assembly errors occur during the second pass, the errors are printed at the console.

The assembler produces a file

x.PRN

where x is the primary name specified in the ASM command. The PRN file contains a listing of the source program (with imbedded tab characters if present in the source program), along with the machine code generated for each statement and diagnostic error messages, if any. The PRN file can be listed
at the console using the TYPE command, or sent to a peripheral device using PIP (see the PIP command structure below). Note also that the PRN file contains the original source program, augmented by miscellaneous assembly information in the leftmost 16 columns (program addresses and hexadecimal machine code, for example). Thus, the PRN file can serve as a backup for the original source file: if the source file is accidently removed or destroyed, the PRN file can be edited (see the ED operator's guide) by removing the leftmost 16 characters of each line (this can be done by issuing a single editor "macro" command). The resulting file is identical to the original source file and can be renamed (REN) from PRN to ASM for subsequent editing and assembly. The file

x.HEX

is also produced which contains 8080 machine language in Intel "hex" format suitable for subsequent loading and execution (see the LOAD command). For complete details of CP/M's assembly language program, see the "CP/M Assembler Language (ASM) User's Guide."

Similar to other transient commands, the source file for assembly can be taken from an alternate disk by prefixing the assembly language file name by a disk drive name. Thus, the command

ASM B:ALPHA cr

loads the assembler from the currently logged drive and operates upon the source program ALPHA.ASM on drive B. The HEX and PRN files are also placed on drive B in this case.

6.3, LOAD ufn cr

The LOAD command reads the file ufn, which is assumed to contain "hex" format machine code, and produces a memory image file which can be subsequently executed. The file name ufn is assumed to be of the form

x.HEX

and thus only the name x need be specified in the command. The LOAD command creates a file named

x.COM

which marks it as containing machine executable code. The file is actually loaded into memory and executed when the user types the file name x immediately after the prompting character ">" printed by the CCP.

In general, the CCP reads the name x following the prompting character and looks for a built-in function name. If no function name is found, the CCP searches the system disk directory for a file by the name
If found, the machine code is loaded into the TPA, and the program executes. Thus, the user need only LOAD a hex file once; it can be subsequently executed any number of times by simply typing the primary name. In this way, the user can "invent" new commands in the CCP. (Initialized disks contain the transient commands as COM files, which can be deleted at the user's option.) The operation can take place on an alternate drive if the file name is prefixed by a drive name. Thus,

LOAD B:BETA

brings the LOAD program into the TPA from the currently logged disk and operates upon drive B after execution begins.

It must be noted that the BETA.HEX file must contain valid Intel format hexadecimal machine code records (as produced by the ASM program, for example) which begin at 100H, the beginning of the TPA. Further, the addresses in the hex records must be in ascending order; gaps in unfilled memory regions are filled with zeroes by the LOAD command as the hex records are read. Thus, LOAD must be used only for creating CP/M standard "COM" files which operate in the TPA. Programs which occupy regions of memory other than the TPA can be loaded under DDT.

6.4. PIP cr

PIP is the CP/M Peripheral Interchange Program which implements the basic media conversion operations necessary to load, print, punch, copy, and combine disk files. The PIP program is initiated by typing one of the following forms

(1) PIP cr
(2) PIP "command line" cr

In both cases, PIP is loaded into the TPA and executed. In case (1), PIP reads command lines directly from the console, prompted with the "*" character, until an empty command line is typed (i.e., a single carriage return is issued by the operator). Each successive command line causes some media conversion to take place according to the rules shown below. Form (2) of the PIP command is equivalent to the first, except that the single command line given with the PIP command is automatically executed, and PIP terminates immediately with no further prompting of the console for input command lines. The form of each command line is

destination = source#1, source#2, ..., source#n cr

where "destination" is the file or peripheral device to receive the data, and
"source#1, ..., source#n" represents a series of one or more files or devices which are copied from left to right to the destination.

When multiple files are given in the command line (i.e., n > 1), the individual files are assumed to contain ASCII characters, with an assumed CP/M end-of-file character (ctl-Z) at the end of each file (see the 0 parameter to override this assumption). The equal symbol (=) can be replaced by a left-oriented arrow, if your console supports this ASCII character, to improve readability. Lower case ASCII alphabets are internally translated to upper case to be consistent with CP/M file and device name conventions. Finally, the total command line length cannot exceed 255 characters (ctl-E can be used to force a physical carriage return for lines which exceed the console width).

The destination and source elements can be unambiguous references to CP/M source files, with or without a preceding drive name. That is, any file can be referenced with a preceding drive name (A:, B:, C:, or D:) which defines the particular drive where the file may be obtained or stored. When the drive name is not included, the currently logged disk is assumed. Further, the destination file can also appear as one or more of the source files, in which case the source file is not altered until the entire concatenation is complete. If the destination file already exists, it is removed if the command line is properly formed (it is not removed if an error condition arises). The following command lines (with explanations to the right) are valid as input to PIP:

```
X = Y cr  
Copy to file X from file Y, where X and Y are unambiguous file names; Y remains unchanged.

X = Y,Z cr  
Concatenate files Y and Z and copy to file X, with Y and Z unchanged.

X,ASM=Y,ASM,Z,ASM,FIN,ASM cr  
Create the file X,ASM from the concatenation of the Y, Z, and FIN files with type ASM.

NEW,ZOT = B:OLD,ZAP cr  
Move a copy of OLD,ZAP from drive B to the currently logged disk; name the file NEW,ZOT.

B:A,U = B:B,V,A:C,W,D,X cr  
Concatenate file B,V from drive B with C,W from drive A and D,X, from the logged disk; create the file A,U on drive B.
```

For more convenient use, PIP allows abbreviated commands for transferring files between disk drives. The abbreviated forms are
PIP x:=afn cr

PIP x:=y:afn cr

PIP ufn = y: cr

PIP x:ufn = y: cr

The first form copies all files from the currently logged disk which satisfy the afn to the same file names on drive x (x = A..Z). The second form is equivalent to the first, where the source for the copy is drive y (y = A..Z). The third form is equivalent to the command "PIP ufn=y:ufn cr" which copies the file given by ufn from drive y to the file ufn on drive x. The fourth form is equivalent to the third, where the source disk is explicitly given by y.

Note that the source and destination disks must be different in all of these cases. If an afn is specified, PIP lists each ufn which satisfies the afn as it is being copied. If a file exists by the same name as the destination file, it is removed upon successful completion of the copy, and replaced by the copied file.

The following PIP commands give examples of valid disk-to-disk copy operations:

B:=*,COM cr          Copy all files which have the secondary name "COM" to drive B from the current drive.

A:=B:ZAP,* cr        Copy all files which have the primary name "ZAP" to drive A from drive B.

ZAP,ASM=B: cr        Equivalent to ZAP,ASM=B:ZAP,ASM

B:ZOT,COM=A: cr      Equivalent to B:ZOT,COM=A:ZOT,COM

B:=GAMMA,BAS cr      Same as B:GAMMA,BAS=GAMMA,BAS

B:=A:GAMMA,BAS cr    Same as B:GAMMA,BAS=A:GAMMA,BAS

PIP also allows reference to physical and logical devices which are attached to the CP/M system. The device names are the same as given under the STAT command, along with a number of specially named devices. The logical devices given in the STAT command are

CON: (console), RDR: (reader), PUN: (punch), and LST: (list)

while the physical devices are
TTY: (console, reader, punch, or list)
CRT: (console, or list), UCL: (console)
PTR: (reader), URL: (reader), UR2: (reader)
PTP: (punch), UPL: (punch), UP2: (punch)
LPT: (list), ULL: (list)

(Note that the "BAT:" physical device is not included, since this assignment is used only to indicate that the RDR: and LST: devices are to be used for console input/output.)

The RDR, LST, PUN, and CON devices are all defined within the BIOS portion of CP/M, and thus are easily altered for any particular I/O system. (The current physical device mapping is defined by IOBYTE; see the "CP/M Interface Guide" for a discussion of this function). The destination device must be capable of receiving data (i.e., data cannot be sent to the punch), and the source devices must be capable of generating data (i.e., the LST: device cannot be read).

The additional device names which can be used in PIP commands are

NUL: Send 40 "nulls" (ASCII 0's) to the device (this can be issued at the end of punched output).

EOF: Send a CP/M end-of-file (ASCII ctl-Z) to the destination device (sent automatically at the end of all ASCII data transfers through PIP).

INP: Special PIP input source which can be "patched" into the PIP program itself: PIP gets the input data character-by-character by CALLing location 103H, with data returned in location 109H (parity bit must be zero).

OUT: Special PIP output destination which can be patched into the PIP program: PIP CALLs location 106H with data in register C for each character to transmit. Note that locations 109H through 1FFH of the PIP memory image are not used and can be replaced by special purpose drivers using DDT (see the DDT operator's manual).

PRN: Same as LST:, except that tabs are expanded at every eighth character position, lines are numbered, and page ejects are inserted every 60 lines, with an initial eject (same as [t8np]).

File and device names can be interspersed in the PIP commands. In each case, the specific device is read until end-of-file (ctl-Z for ASCII files, and a real end of file for non-ASCII disk files). Data from each device or file is concatenated from left to right until the last data source has been
read. The destination device or file is written using the data from the
source files, and an end-of-file character (ctl-Z) is appended to the result
for ASCII files. Note if the destination is a disk file, then a temporary
file is created ($$$ secondary name) which is changed to the actual file name
only upon successful completion of the copy. Files with the extension "COM"
are always assumed to be non-ASCII.

The copy operation can be aborted at any time by depressing any key on
the keyboard (a rubout suffices). PIP will respond with the message "ABORTED"
to indicate that the operation was not completed. Note that if any operation
is aborted, or if an error occurs during processing, PIP removes any pending
commands which were set up while using the SUBMIT command.

It should also be noted that PIP performs a special function if the
destination is a disk file with type "HEX" (an Intel hex formatted machine
code file), and the source is an external peripheral device, such as a paper
tape reader. In this case, the PIP program checks to ensure that the source
file contains a properly formed hex file, with legal hexadecimal values and
checksum records. When an invalid input record is found, PIP reports an error
message at the console and waits for corrective action. It is usually
sufficient to open the reader and rerun a section of the tape (pull the tape
back about 20 inches). When the tape is ready for the re-read, type a single
carriage return at the console, and PIP will attempt another read. If the
tape position cannot be properly read, simply continue the read (by typing a
return following the error message), and enter the record manually with the ED
program after the disk file is constructed. For convenience, PIP allows the
end-of-file to be entered from the console if the source file is a PDR:
device. In this case, the PIP program reads the device and monitors the
keyboard. If ctl-Z is typed at the keyboard, then the read operation is
terminated normally.

Valid PIP commands are shown below.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIP LST: = X,PRN cr</td>
<td>Copy X,PRN to the LST device and terminate the PIP program.</td>
</tr>
<tr>
<td>PIP cr</td>
<td>Start PIP for a sequence of commands (PIP prompts with &quot;.&quot;).</td>
</tr>
<tr>
<td>*CON:=X,ASM,Y,ASM,Z,ASM cr</td>
<td>Concatenate three ASM files and copy to the CON device.</td>
</tr>
<tr>
<td>*X,HEX=CON:,Y,HEX,PTR: cr</td>
<td>Create a HEX file by reading the CON (until a ctl-Z is typed), followed by data from Y,HEX, followed by data from PTR until a ctl-Z is encountered.</td>
</tr>
<tr>
<td>*cr</td>
<td>Single carriage return stops PIP.</td>
</tr>
</tbody>
</table>
PIP PUN: = NUL:, X, ASM, EOF:, NUL: cr

Send 40 nulls to the punch device; then copy the X,ASM file to the punch, followed by an end-of-file (ctl-Z) and 40 more null characters.

The user can also specify one or more PIP parameters, enclosed in left and right square brackets, separated by zero or more blanks. Each parameter affects the copy operation, and the enclosed list of parameters must immediately follow the affected file or device. Generally, each parameter can be followed by an optional decimal integer value (the S and Q parameters are exceptions). The valid PIP parameters are listed below.

B Block mode transfer: data is buffered by PIP until an ASCII x-off character (ctl-S) is received from the source device. This allows transfer of data to a disk file from a continuous reading device, such as a cassette reader. Upon receipt of the x-off, PIP clears the disk buffers and returns for more input data. The amount of data which can be buffered is dependent upon the memory size of the host system (PIP will issue an error message if the buffers overflow).

Dn Delete characters which extend past column n in the transfer of data to the destination from the character source. This parameter is used most often to truncate long lines which are sent to a (narrow) printer or console device.

E Echo all transfer operations to the console as they are being performed.

F Filter form feeds from the file. All imbedded form feeds are removed. The P parameter can be used simultaneously to insert new form feeds.

H Hex data transfer: all data is checked for proper Intel hex file format. Non-essential characters between hex records are removed during the copy operation. The console will be prompted for corrective action in case errors occur.

I Ignore "$00" records in the transfer of Intel hex format file (the I parameter automatically sets the H parameter).

L Translate upper case alphabetics to lower case.

N Add line numbers to each line transferred to the destination starting at one, and incrementing by 1. Leading zeroes are suppressed, and the number is followed by a colon. If N2 is specified, then leading zeroes are included, and a tab is inserted following the number. The tab is expanded if T is
set.

O  Object file (non-ASCII) transfer: the normal CP/M end of file is ignored.

Pn Include page ejects at every n lines (with an initial page eject). If n = 1 or is excluded altogether, page ejects occur every 60 lines. If the F parameter is used, form feed suppression takes place before the new page ejects are inserted.

Qs^z Quit copying from the source device or file when the string s (terminated by ctl-Z) is encountered.

Ss^z Start copying from the source device when the string s is encountered (terminated by ctl-Z). The S and Q parameters can be used to "abstract" a particular section of a file (such as a subroutine). The start and quit strings are always included in the copy operation.

NOTE - the strings following the s and q parameters are translated to upper case by the CCP if form (2) of the PIP command is used. Form (1) of the PIP invocation, however, does not perform the automatic upper case translation.

(1) PIP cr
(2) PIP "command line" cr

Tn Expand tabs (ctl-I characters) to every nth column during the transfer of characters to the destination from the source.

U Translate lower case alphabetics to upper case during the copy operation.

V Verify that data has been copied correctly by rereading after the write operation (the destination must be a disk file).

Z Zero the parity bit on input for each ASCII character.

The following are valid PIP commands which specify parameters in the file transfer:

PIP X,ASM=B:[v] cr  Copy X,ASM from drive B to the current drive and verify that the data was properly copied.

PIP LPT=:=X,ASM[nt8u] cr  Copy X,ASM to the LPT: device; number each line, expand tabs to every eighth column, and translate lower case alphabetics to upper case.
PIP PUN: = X,HEX[i], Y,ZOT[h] cr First copy X,HEX to the PUN: device and ignore the trailing ":00" record in X,HEX; then continue the transfer of data by reading Y,ZOT, which contains hex records, including any ":00" records which it contains.

PIP X,LIB = Y,ASM [ sSUBR1; 'z qJMP L3'z ] cr Copy from the file Y,ASM into the file X,LIB. Start the copy when the string "SUBR1:" has been found, and quit copying after the string "JMP L3" is encountered.

PIP PRN: = X,ASM[p50] Send X,ASM to the LST: device, with line numbers, tabs expanded to every eighth column, and page ejects at every 50th line. Note that nt8p60 is the assumed parameter list for a PRN file; p50 overrides the default value.

6.5. ED ufn cr

The ED program is the CP/M system context editor, which allows creation and alteration of ASCII files in the CP/M environment. Complete details of operation are given the ED user's manual, "ED: a Context Editor for the CP/M Disk System." In general, ED allows the operator to create and operate upon source files which are organized as a sequence of ASCII characters, separated by end-of-line characters (a carriage-return line-feed sequence). There is no practical restriction on line length (no single line can exceed the size of the working memory), which is instead defined by the number of characters typed between cr's. The ED program has a number of commands for character string searching, replacement, and insertion, which are useful in the creation and correction of programs or text files under CP/M. Although the CP/M has a limited memory work space area (approximately 5000 characters in a 16K CP/M system), the file size which can be edited is not limited, since data is easily "paged" through this work area.

Upon initiation, ED creates the specified source file, if it does not exist, and opens the file for access. The programmer then "appends" data from the source file into the work area, if the source file already exists (see the A command), for editing. The appended data can then be displayed, altered, and written from the work area back to the disk (see the W command). Particular points in the program can be automatically paged and located by context (see the N command), allowing easy access to particular portions of a large file.

Given that the operator has typed

ED X,ASM cr
the ED program creates an intermediate work file with the name

X.$$$

to hold the edited data during the ED run. Upon completion of ED, the X,ASM file (original file) is renamed to X,BAK, and the edited work file is renamed to X,ASM. Thus, the X,BAK file contains the original (unedited) file, and the X,ASM file contains the newly edited file. The operator can always return to the previous version of a file by removing the most recent version, and renaming the previous version. Suppose, for example, that the current X,ASM file was improperly edited; the sequence of CCP command shown below would reclaim the backup file.

```
DIR X,*        Check to see that BAK file is available.
ERA X,ASM      Erase most recent version.
REN X,ASM=X,BAK Rename the BAK file to ASM.
```

Note that the operator can abort the edit at any point (reboot, power failure, ctrl-C, or Q command) without destroying the original file. In this case, the BAK file is not created, and the original file is always intact.

The ED program also allows the user to "ping-pong" the source and create backup files between two disks. The form of the ED command in this case is

```
ED ufn d:
```

where ufn is the name of a file to edit on the currently logged disk, and d is the name of an alternate drive. The ED program reads and processes the source file, and writes the new file to drive d, using the name ufn. Upon completion of processing, the original file becomes the backup file. Thus, if the operator is addressing disk A, the following command is valid:

```
ED X,ASM B:
```

which edits the file X,ASM on drive A, creating the new file X,$$$ on drive B. Upon completion of a successful edit, A:X,ASM is renamed to A:X,BAK, and B:X,$$$ is renamed to B:X,ASM. For user convenience, the currently logged disk becomes drive B at the end of the edit. Note that if a file by the name B:X,ASM exists before the editing begins, the message

```
FILE EXISTS
```

is printed at the console as a precaution against accidently destroying a source file. In this case, the operator must first ERAs the existing file and then restart the edit operation.
Similar to other transient commands, editing can take place on a drive different from the currently logged disk by preceding the source file name by a drive name. Examples of valid edit requests are shown below.

ED A:X.ASM  Edit the file X.ASM on drive A, with new file and backup on drive A.

ED B:X.ASM A: Edit the file X.ASM on drive B to the temporary file X.SSS on drive A. On termination of editing, change X.ASM on drive B to X.BAK, and change X.SSS on drive A to X.ASM.

6.6. SYSGEN cr

The SYSGEN transient command allows generation of an initialized diskette containing the CP/M operating system. The SYSGEN program prompts the console for commands, with interaction as shown below.

SYSGEN cr  Initiate the SYSGEN program.
SYSGEN VERSION m.m  SYSGEN sign-on message.
SOURCE DRIVE NAME (OR RETURN TO SKIP)  Respond with the drive name (one of the letters A, B, C, or D) of the disk containing a CP/M system; usually A. If a copy of CP/M already exists in memory, due to a MOVCPM command, type a cr only. Typing a drive name x will cause the response:

SOURCE ON x THEN TYPE RETURN  Place a diskette containing the CP/M operating system on drive x (x is one of A, B, C, or D). Answer with cr when ready.

FUNCTION COMPLETE  System is copied to memory. SYSGEN will then prompt with:

DESTINATION DRIVE NAME (OR RETURN TO REBOOT)  If a diskette is being initialized, place the new disk into a drive and answer with the drive name. Otherwise, type a cr and the system will reboot from drive A. Typing drive name x will cause SYSGEN to prompt.
with:

**DESTINATION** on x then type return

Place new diskette into drive
x; type return when ready.

**FUNCTION COMPLETE**

New diskette is initialized
in drive x.

The "DESTINATION" prompt will be repeated until a single carriage return is
typed at the console, so that more than one disk can be initialized.

Upon completion of a successful system generation, the new diskette
contains the operating system, and only the built-in commands are available.
A factory-fresh IBM-compatible diskette appears to CP/M as a diskette with an
empty directory; therefore, the operator must copy the appropriate COM files
from an existing CP/M diskette to the newly constructed diskette using the PIP
transient.

The user can copy all files from an existing diskette by typing the PIP
command

\[ \text{PIP B: = A: *.*[v] cr} \]

which copies all files from disk drive A to disk drive B, and verifies that
each file has been copied correctly. The name of each file is displayed at
the console as the copy operation proceeds.

It should be noted that a SYSGEN does not destroy the files which already
exist on a diskette; it results only in construction of a new operating
system. Further, if a diskette is being used only on drives B through D, and
will never be the source of a bootstrap operation on drive A, the SYSGEN need
not take place. In fact, a new diskette needs absolutely no initialization to
be used with CP/M.

6.7. **SUBMIT ufn parm#1 ... parm#n cr**

The **SUBMIT** command allows CP/M commands to be batched together for
automatic processing. The ufn given in the **SUBMIT** command must be the
filename of a file which exists on the currently logged disk, with an assumed
file type of "SUB." The SUB file contains CP/M prototype commands, with
possible parameter substitution. The actual parameters parm#1 ... parm#n are
substituted into the prototype commands, and, if no errors occur, the file of
substituted commands are processed sequentially by CP/M.
The prototype command file is created using the ED program, with interspersed "$" parameters of the form

$$\text{S}_1\text{ S}_2\text{ S}_3 \ldots \text{ S}_n$$

corresponding to the number of actual parameters which will be included when the file is submitted for execution. When the SUBMIT transient is executed, the actual parameters $\text{parm}_1 \ldots \text{parm}_n$ are paired with the formal parameters $\text{S}_1 \ldots \text{ S}_n$ in the prototype commands. If the number of formal and actual parameters does not correspond, then the submit function is aborted with an error message at the console. The SUBMIT function creates a file of substituted commands with the name

$$\text{ $$$.SUB}$$
on the logged disk. When the system reboots (at the termination of the SUBMIT), this command file is read by the CCP as a source of input, rather than the console. If the SUBMIT function is performed on any disk other than drive A, the commands are not processed until the disk is inserted into drive A and the system reboots. Further, the user can abort command processing at any time by typing a rubout when the command is read and echoed. In this case, the $$$.SUB file is removed, and the subsequent commands come from the console. Command processing is also aborted if the CCP detects an error in any of the commands. Programs which execute under CP/M can abort processing of command files when error conditions occur by simply erasing any existing $$$.SUB file.

In order to introduce dollar signs into a SUBMIT file, the user may type a "$" which reduces to a single "$" within the command file. Further, an up-arrow symbol "↑" may precede an alphabetic character x, which produces a single ctrl-x character within the file.

The last command in a SUB file can initiate another SUB file, thus allowing chained batch commands.

Suppose the file ASMBL$.SUB exists on disk and contains the prototype commands

```
AS$ $1
DIR $1.*
ERA *.BAK
PI$ $2:=$1.PR$N
ERA $1.PR$N
```

and the command

SUBMIT ASMBL X PRN cr

is issued by the operator. The SUBMIT program reads the ASMBL$.SUB file, substituting "x" for all occurrences of $1 and "PRN" for all occurrences of $2, resulting in a $$$.SUB file containing the commands.
which are executed in sequence by the CCP.

The SUBMIT function can access a SUB file which is on an alternate drive by preceding the file name by a drive name. Submitted files are only acted upon, however, when they appear on drive A. Thus, it is possible to create a submitted file on drive B which is executed at a later time when it is inserted in drive A.

6.8. DUMP ufn cr

The DUMP program types the contents of the disk file (ufn) at the console in hexadecimal form. The file contents are listed sixteen bytes at a time, with the absolute byte address listed to the left of each line in hexadecimal. Long typeouts can be aborted by pushing the rubout key during printout. (The source listing of the DUMP program is given in the "CP/M Interface Guide" as an example of a program written for the CP/M environment.)

6.9. MOVCPM cr

The MOVCPM program allows the user to reconfigure the CP/M system for any particular memory size. Two optional parameters may be used to indicate (1) the desired size of the new system and (2) the disposition of the new system at program termination. If the first parameter is omitted or a "**" is given, the MOVCPM program will reconfigure the system to its maximum size, based upon the kilobytes of contiguous RAM in the host system (starting at 0000H). If the second parameter is omitted, the system is executed, but not permanently recorded; if "**" is given, the system is left in memory, ready for a SYSGEN operation. The MOVCPM program relocates a memory image of CP/M and places this image in memory in preparation for a system generation operation. The command forms are:

MOVCPM cr  Relocate and execute CP/M for management of the current memory configuration (memory is examined for contiguous RAM, starting at 100H). Upon completion of the relocation, the new system is executed but not permanently recorded on the diskette. The system which is constructed contains a BIOS for the Intel MDS 800.
MOVCPM n cr
Create a relocated CP/M system for management of an n kilobyte system (n must be in the range 16 to 64), and execute the system, as described above.

MOVCPM ** cr
Construct a relocated memory image for the current memory configuration, but leave the memory image in memory, in preparation for a SYSGEN operation.

MOVCPM n * cr
Construct a relocated memory image for an n kilobyte memory system, and leave the memory image in preparation for a SYSGEN operation.

The command

MOVCPM **
for example, constructs a new version of the CP/M system and leaves it in memory, ready for a SYSGEN operation. The message

READY FOR "SYSGEN" OR "SAVE 32 CPMJxx.COM"

is printed at the console upon completion, where xx is the current memory size in kilobytes. The operator can then type

SYSGEN cr
Start the system generation.

SOURCE DRIVE NAME (OR RETURN TO SKIP)
Respond with a cr to skip the CP/M read operation since the system is already in memory as a result of the previous MOVCPM operation.

DESTINATION DRIVE NAME (OR RETURN TO REBOOT)
Respond with B to write new system to the diskette in drive B. SYSGEN will prompt with:

DESTINATION ON B, THEN TYPE RETURN
Ready the fresh diskette on drive B and type a return when ready.

Note that if you respond with "A" rather than "B" above, the system will be written to drive A rather than B. SYSGEN will continue to type the prompt:

DESTINATION DRIVE NAME (OR RETURN TO REBOOT)

until the operator responds with a single carriage return, which stops the
SYSGEN program with a system reboot.

The user can then go through the reboot process with the old or new diskette. Instead of performing the SYSGEN operation, the user could have typed

```
SAVE 32 CPMxx.COM
```

at the completion of the MOVCP function, which would place the CP/M memory image on the currently logged disk in a form which can be "patched". This is necessary when operating in a non-standard environment where the BIOS must be altered for a particular peripheral device configuration, as described in the "CP/M System Alteration Guide".

Valid MOVCP commands are given below:

- **MOVCPM 48 cr** Construct a 48K version of CP/M and start execution
- **MOVCPM 48 * cr** Construct a 48K version of CP/M in preparation for permanent recording; response is READY FOR "SYSGEN" OR "SAVE 32 CPM48.COM"
- **MOVCPM * * CR** Construct a maximum memory version of CP/M and start execution.

It is important to note that the newly created system is serialized with the number attached to the original diskette and is subject to the conditions of the Exidy Inc. Software Licensing Agreement.
7. BDOS ERROR MESSAGES.

There are three error situations which the Basic Disk Operating System intercepts during file processing. When one of these conditions is detected, the BDOS prints the message:

BDOS ERR ON x: error

where x is the drive name, and "error" is one of the three error messages:

BAD SECTOR
SELECT
READ ONLY

The "BAD SECTOR" message indicates that the disk controller electronics has detected an error condition in reading or writing the diskette. This condition is generally due to a malfunctioning disk controller, or an extremely worn diskette. If you find that your system reports this error more than once a month, you should check the state of your controller electronics, and the condition of your media. You may also encounter this condition in reading files generated by a controller produced by a different manufacturer. Even though controllers are claimed to be IBM-compatible, one often finds small differences in recording formats. The MDS-800 controller, for example, requires two bytes of one's following the data CRC byte, which is not required in the IBM format. As a result, diskettes generated by the Intel MDS can be read by almost all other IBM-compatible systems, while disk files generated on other manufacturer's equipment will produce the "BAD SECTOR" message when read by the MDS. In any case, recovery from this condition is accomplished by typing a ctl-C to reboot (this is the safest!), or a return, which simply ignores the bad sector in the file operation. Note, however, that typing a return may destroy your diskette integrity if the operation is a directory write, so make sure you have adequate backups in this case.

The "SELECT" error occurs when there is an attempt to address a drive beyond the A through D range. In this case, the value of x in the error message gives the selected drive. The system reboots following any input from the console.

The "READ ONLY" message occurs when there is an attempt to write to a diskette which has been designated as read-only in a STAT command, or has been set to read-only by the BDOS. In general, the operator should reboot CP/M either by using the warm start procedure (ctl-C) or by performing a cold start whenever the diskettes are changed. If a changed diskette is to be read but not written, BDOS allows the diskette to be changed without the warm or cold start, but internally marks the drive as read-only. The status of the drive is subsequently changed to read/write if a warm or cold start occurs. Upon issuing this message, CP/M waits for input from the console. An automatic warm start takes place following any input.
8. OPERATION OF CP/M ON THE MDS.

This section gives operating procedures for using CP/M on the Intel MDS microcomputer development system. A basic knowledge of the MDS hardware and software systems is assumed.

CP/M is initiated in essentially the same manner as Intel's ISIS operating system. The disk drives are labelled 0 through 3 on the MDS, corresponding to CP/M drives A through D, respectively. The CP/M system diskette is inserted into drive 0, and the BOOT and RESET switches are depressed in sequence. The interrupt 2 light should go on at this point. The space bar is then depressed on the device which is to be taken as the system console, and the light should go out (if it does not, then check connections and baud rates). The BOOT switch is then turned off, and the CP/M signon message should appear at the selected console device, followed by the "A>" system prompt. The user can then issue the various resident and transient commands.

The CP/M system can be restarted (warm start) at any time by pushing the INT 0 switch on the front panel. The built-in Intel ROM monitor can be initiated by pushing the INT 7 switch (which generates a RST 7), except when operating under DDT, in which case the DDT program gets control instead.

Diskettes can be removed from the drives at any time, and the system can be shut down during operation without affecting data integrity. Note, however, that the user must not remove a diskette and replace it with another without rebooting the system (cold or warm start), unless the inserted diskette is "read only."

Due to hardware hang-ups or malfunctions, CP/M may type the message

```
BDOS ERR ON x: BAD SECTOR
```

where x is the drive which has a permanent error. This error may occur when drive doors are opened and closed randomly, followed by disk operations, or may be due to a diskette, drive, or controller failure. The user can optionally elect to ignore the error by typing a single return at the console. The error may produce a bad data record, requiring re-initialization of up to 128 bytes of data. The operator can reboot the CP/M system and try the operation again.

Termination of a CP/M session requires no special action, except that it is necessary to remove the diskettes before turning the power off, to avoid random transients which often make their way to the drive electronics.

It should be noted that factory-fresh IBM-compatible diskettes should be used rather than diskettes which have previously been used with any ISIS version. In particular, the ISIS "FORMAT" operation produces non-standard sector numbering throughout the diskette. This non-standard numbering seriously degrades the performance of CP/M, and will operate noticeably slower.
than the distribution version. If it becomes necessary to reformat a diskette (which should not be the case for standard diskettes), a program can be written under CP/M which causes the MDS 800 controller to reformat with sequential sector numbering (1-26) on each track.

Note: "MDS 800" and "ISIS" are registered trademarks of Intel Corporation.
CP/M INTERFACE GUIDE
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>CP/M Organization</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Operation of Transient Programs</td>
<td>1</td>
</tr>
<tr>
<td>1.3</td>
<td>Operating System Facilities</td>
<td>3</td>
</tr>
<tr>
<td>2.</td>
<td>BASIC I/O FACILITIES</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Direct and Buffered I/O</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>A Simple Example</td>
<td>5</td>
</tr>
<tr>
<td>3.</td>
<td>DISK I/O FACILITIES</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>File System Organization</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>File Control Block Format</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>Disk Access Primitives</td>
<td>12</td>
</tr>
<tr>
<td>3.4</td>
<td>Random Access</td>
<td>18</td>
</tr>
<tr>
<td>4.</td>
<td>SYSTEM GENERATION</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Initializing CP/M from an Existing Diskette</td>
<td>19</td>
</tr>
<tr>
<td>5.</td>
<td>CP/M ENTRY POINT SUMMARY</td>
<td>20</td>
</tr>
<tr>
<td>6.</td>
<td>ADDRESS ASSIGNMENTS</td>
<td>22</td>
</tr>
<tr>
<td>7.</td>
<td>SAMPLE PROGRAMS</td>
<td>23</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

This manual describes the CP/M system organization including the structure of memory, as well as system entry points. The intention here is to provide the necessary information required to write programs which operate under CP/M, and which use the peripheral and disk I/O facilities of the system.

1.1 CP/M Organization

CP/M is logically divided into four parts:

- BIOS - the basic I/O system for serial peripheral control
- BDOS - the basic disk operating system primitives
- CCP - the console command processor
- TPA - the transient program area

The BIOS and BDOS are combined into a single program with a common entry point and referred to as the FDOS. The CCP is a distinct program which uses the FDOS to provide a human-oriented interface to the information which is cataloged on the diskette. The TPA is an area of memory (i.e., the portion which is not used by the FDOS and CCP) where various non-resident operating system commands are executed. User programs also execute in the TPA. The organization of memory in a standard CP/M system is shown in Figure 1.

The lower portion of memory is reserved for system information (which is detailed in later sections), including user defined interrupt locations. The portion between tbase and cbase is reserved for the transient operating system commands, while the portion above cbase contains the resident CCP and FDOS. The last three locations of memory contain a jump instruction to the FDOS entry point which provides access to system functions.

1.2 Operation of Transient Programs

Transient programs (system functions and user-defined programs) are loaded into the TPA and executed as follows. The operator communicates with the CCP by typing command lines following each prompt character. Each command line takes one of the forms:

\[
\begin{align*}
\text{<command>} \\
\text{<command> <filename>} \\
\text{<command> <filename>.<filetype>}
\end{align*}
\]
Figure 1. CP/M Memory Organization

- fbase: FDOS
- cbase: CCP
- tbase: System Parameters
- boot: Address field of jump is fbase

Note: The exact addresses for boot, tbase, cbase, fbase, and entry vary with the CP/M version (see Section 6. for version correspondence).

entry: The principal entry point to FDOS is at location 0005 which contains a JMP to fbase. The address field at location 0006 can be used to determine the size of available memory, assuming the CCP is being overlayed.
Where <command> is either a built-in command (e.g., DIR or TYPE), or the name of a transient command or program. If the <command> is a built-in function of CP/M, it is executed immediately; otherwise the CCP searches the currently addressed disk for a file by the name

<command>.COM

If the file is found, it is assumed to be a memory image of a program which executes in the TPA, and thus implicitly originates at tbase in memory (see the CP/M LOAD command). The CCP loads the COM file from the diskette into memory starting at tbase, and extending up to address cbase.

If the <command> is followed by either a <filename> or <filename>.<filetype>, then the CCP prepares a file control-block (FCB) in the system information area of memory. This FCB is in the form required to access the file through the FDOS, and is given in detail in Section 3.2.

The program then executes, perhaps using the I/O facilities of the FDOS. If the program uses no FDOS facilities, then the entire remaining memory area is available for data used by the program. If the FDOS is to remain in memory, then the transient program can use only up to location fbase as data.* In any case, if the CCP area is used by the transient, the entire CP/M system must be reloaded upon the transient's completion. This system reload is accomplished by a direct branch to location "boot" in memory.

The transient uses the CP/M I/O facilities to communicate with the operator's console and peripheral devices, including the floppy disk subsystem. The I/O system is accessed by passing a "function number" and an "information address" to CP/M through the address marked "entry" in Figure 1. In the case of a disk read, for example, the transient program sends the number corresponding to a disk read, along with the address of an FCB, and CP/M performs the operation, returning with either a disk read complete indication or an error number indicating that the disk operation was unsuccessful. The function numbers and error indicators are given in detail in Section 3.3.

1.3 Operating System Facilities

CP/M facilities which are available to transients are divided into two categories: BIOS operations, and BDOS primitives. The BIOS operations are listed first:**

* Address "entry" contains a jump to the lowest address in the FDOS, and thus "entry+1" contains the first FDOS address which cannot be overlayed.

**The device support (exclusive of the disk subsystem) corresponds exactly to Intel's peripheral definition, including I/O port assignment and status byte format (see the Intel manual which discusses the Intellec MDS hardware environment).
Read Console Character
Write Console Character
Read Reader Character
Write Punch Character
Write List Device Character
Set I/O Status
Interrogate Device Status
Print Console Buffer
Read Console Buffer
Interrogate Console Status

The exact details of BIOS access are given in Section 2. The BDOS primitives include the following operations:

Disk System Reset
Drive Select
File Creation
File Open
File Close
Directory Search
File Delete
File Rename
Read Record
Write Record
Interrogate Available Disks
Interrogate Selected Disk
Set DMA Address

The details of BDOS access are given in Section 3.

2. BASIC I/O FACILITIES

Access to common peripherals is accomplished by passing a function number and information address to the BIOS. In general, the function number is passed in Register C, while the information address is passed in Register pair D,E. Note that this conforms to the PL/M Conventions for parameter passing, and thus the following PL/M procedure is sufficient to link to the BIOS when a value is returned:

DECLARE ENTRY LITERALLY '0005H'; /* MONITOR ENTRY */

MON2: PROCEDURE (FUNC, INFO) BYTE;
DECLARE FUNC BYTE, INFO ADDRESS;
GO TO ENTRY;

END MON2;
or

MON1: PROCEDURE (FUNC,INFO);
    DECLARE FUNC BYTE, INFO ADDRESS;
    GO TO ENTRY;
    END MON1

if no returned value is expected.

2.1 Direct and Buffered I/O.

The BIOS entry points are given in Table I. In the case of simple character I/O to the console, the BIOS reads the console device, and removes the parity bit. The character is echoed back to the console, and tab characters (control-I) are expanded to tab positions starting at column one and separated by eight character positions. The I/O status byte takes the form shown in Table I, and can be programmatically interrogated or changed. The buffered read operation takes advantage of the CP/M line editing facilities. That is, the program sends the address of a read buffer whose first byte is the length of the buffer. The second byte is initially empty, but is filled-in by CP/M to the number of characters read from the console after the operation (not including the terminating carriage-return). The remaining positions are used to hold the characters read from the console. The BIOS line editing functions which are performed during this operation are given below:

break  - line delete and transmit
rubout  - delete last character typed, and echo
ccontrol-C - system reboot
control-U - delete entire line
control-E - return carriage, but do not transmit buffer (physical carriage return)
<cr>   - transmit buffer

The read routine also detects control character sequences other than those shown above, and echos them with a preceding "!" symbol. The print entry point allows an entire string of symbols to be printed before returning from the BIOS. The string is terminated by a "$" symbol.

2.2 A Simple Example

As an example, consider the following PL/M procedures and procedure calls which print a heading, and successively read the console buffer. Each console buffer is then echoed back in reverse order:
PRINTCHAR:  PROCEDURE (B);
  /* SEND THE ASCII CHARACTER B TO THE CONSOLE */
  DECLARE B BYTE;
  CALL M0N1(2,B);
  END PRINTCHAR;

CRLF:  PROCEDURE;
  /* SEND CARRIAGE-RETURN-LINE-FEED CHARACTERS */
  CALL PRINTCHAR (0DH); CALL PRINTCHAR (0AH);
  END CRLF;

PRINT: PROCEDURE (A);
  /* PRINT THE BUFFER STARTING AT ADDRESS A */
  DECLARE A ADDRESS;
  CALL M0N1(9,A);
  END PRINT;

DECLARE RDBUFF (130) BYTE;

READ:  PROCEDURE;
  /* READ CONSOLE CHARACTERS INTO 'RDBUFF' */
  RDBUFF=128; /* FIRST BYTE SET TO BUFFER LENGTH */
  CALL M0N1(10,.RDBUFF);
  END READ;

DECLARE I BYTE;
CALL CRLF;
  CALL PRINT (.'TYPE INPUT LINES $');
  DO WHILE 1; /* INFINITE LOOP-UNTIL CONTROL-C */
    CALL CRLF; CALL PRINTCHAR ("*"); /* PROMPT WITH ' * ' */
    CALL READ; I = RDBUFF(1);
    DO WHILE (I:= I -1) <> 255;
      CALL PRINTCHAR (RDBUFF(I+2));
    END;
  END;

The execution of this program might proceed as follows:

TYPE INPUT LINES
*HELLO,
OLLEH
*WALL WALLA WASH;
HSAW ALLAW ALLAW
*MOM WOW,
WOW MOM
*↑C             (system reboot)
<table>
<thead>
<tr>
<th>FUNCTION/NUMBER</th>
<th>ENTRY PARAMETERS</th>
<th>RETURNED VALUE</th>
<th>TYPICAL CALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Console 1</td>
<td>None</td>
<td>ASCII Character</td>
<td>I = MON2(1,0)</td>
</tr>
<tr>
<td>Write Console 2</td>
<td>ASCII Character</td>
<td>None</td>
<td>CALL MON1(2,'A')</td>
</tr>
<tr>
<td>Read Reader 3</td>
<td>None</td>
<td>ASCII Character</td>
<td>I = MON2(3,0)</td>
</tr>
<tr>
<td>Write Punch 4</td>
<td>ASCII Character</td>
<td>None</td>
<td>CALL MON1(4,'B')</td>
</tr>
<tr>
<td>Write List 5</td>
<td>ASCII Character</td>
<td>None</td>
<td>CALL MON1(5,'C')</td>
</tr>
<tr>
<td>Get I/O Status 7</td>
<td>None</td>
<td>I/O Status Byte</td>
<td>IOSTAT=MON2(7,0)</td>
</tr>
<tr>
<td>Set I/O Status 8</td>
<td>I/O Status Byte</td>
<td>None</td>
<td>CALL MON1(8,IOSTAT)</td>
</tr>
<tr>
<td>Print Buffer 9</td>
<td>Address of string terminated by '§'</td>
<td>None</td>
<td>CALL MON1(9, . 'PRINT THIS §')</td>
</tr>
</tbody>
</table>
TABLE I (continued)

<table>
<thead>
<tr>
<th>FUNCTION/NUMBER</th>
<th>ENTRY PARAMETERS</th>
<th>RETURNED VALUE</th>
<th>TYPICAL CALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Buffer 10</td>
<td>Address of Read Buffer* (See Note₁)</td>
<td>Read buffer is filled to maximum length with console characters</td>
<td>CALL MON1(10, .RDBUFF);</td>
</tr>
<tr>
<td>Interrogate Console Ready 11</td>
<td>None</td>
<td>Byte value with least significant bit = 1 (true) if console character is ready</td>
<td>I = MON2(11, 0)</td>
</tr>
</tbody>
</table>

Note₁: Read buffer is a sequence of memory locations of the form:

\[
\begin{array}{cccccccc}
  m & k & c_1 & c_2 & c_3 & \ldots & c_k & \ldots \\
\end{array}
\]

- current buffer length
- Maximum buffer length

Note₂: The I/O status byte is defined as three fields A, B, C, and D requiring two bits each, listed from most significant to least significant bit, which define the current device assignment as follows:

<table>
<thead>
<tr>
<th>D</th>
<th>C</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Console</td>
<td>TTY</td>
<td>FAST READER</td>
<td>FAST PUNCH</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c|c|c|c|c}
  & A & B & C & D \\
\hline
\text{MSB} & 2b & 2b & 2b & 2b \\
\text{LSB} & & & & & \\
\end{array}
\]
3. DISK I/O FACILITIES

The BDOS section of CP/M provides access to files stored on diskettes. The discussion which follows gives the overall file organization, along with file access mechanisms.

3.1 File Organization

CP/M implements a named file structure on each diskette, providing a logical organization which allows any particular file to contain any number of records, from completely empty, to the full capacity of a diskette. Each diskette is logically distinct, with a complete operating system, disk directory, and file data area. The disk file names are in two parts: the <filename> which can be from one to eight alphanumeric characters, and the <filetype> which consists of zero through three alphanumeric characters. The <filetype> names the generic category of a particular file, while the <filename> distinguishes a particular file within the category. The <filetype>’s listed below give some generic categories which have been established, although they are generally arbitrary:

| ASM | assembler source file         |
| PRN | assembler listing file        |
| HEX | assembler or PL/M machine code |
|     | in "hex" format               |
| BAS | BASIC Source file             |
| INT | BASIC Intermediate file       |
| COM | Memory image file (i.e., "Command" file for transients, produced by LOAD) |
| BAK | Backup file produced by editor |
|     | (see ED manual)               |
| $$$ | Temporary files created and normally erased by editor and utilities |

Thus, the name

X.ASM

is interpreted as an assembly language source file by the CCP with <filename> X.

The files in CP/M are organized as a logically contiguous sequence of 128 byte records (although the records may not be physically contiguous on the diskette), which are normally read or written in sequential order. Random access is allowed under CP/M however, as described in Section 3.4. No particular format within records is assumed by CP/M, although some transients expect particular formats:
(1) Source files are considered a sequence of ASCII characters, where each "line" of the source file is followed by carriage-return-line-feed characters. Thus, one 128 byte CP/M record could contain several logical lines of source text. Machine code "hex" tapes are also assumed to be in this format, although the loader does not require the carriage-return-line-feed characters. End of text is given by the character control-z, or real end-of-file returned by CP/M.

and

(2) COM files are assumed to be absolute machine code in memory image form, starting at tbase in memory. In this case, control-z is not considered an end of file, but instead is determined by the actual space allocated to the file being accessed.

3.2 File Control Block Format

Each file being accessed through CP/M has a corresponding file control block (FCB) which provides name and allocation information for all file operations. The FCB is a 33-byte area in the transient program's memory space which is set up for each file. The FCB format is given in Figure 2. When accessing CP/M files, it is the programmer's responsibility to fill the lower 16 bytes of the FCB, along with the CR field. Normally, the FN and FT fields are set to the ASCII <filename> and <filetype>, while all other fields are set to zero. Each FCB describes up to 16K bytes of a particular file (0 to 128 records of 128 bytes each), and, using automatic mechanisms of CP/M, up to 15 additional extensions of the file can be addressed. Thus, each FCB can potentially describe files up to 256K bytes (which is slightly larger than the diskette capacity).

FCB's are stored in a directory area of the diskette, and are brought into central memory before file operations (see the OPEN and MAKE commands) then updated in memory as file operations proceed, and finally recorded on the diskette at the termination of the file operation (see the CLOSE command). This organization makes CP/M file organization highly reliable, since diskette file integrity can only be disrupted in the unlikely case of hardware failure during update of a single directory entry.

It should be noted that the CCP constructs an FCB for all transients by scanning the remainder of the line following the transient name for a <filename> or <filename>.<filetype> combination. Any field not specified is assumed to be all blanks. A properly formed FCB is set up at location tfcb (see Section 6), with an assumed I/O buffer at tbuff. The transient can use tfcb as an address in subsequent input or output operations on this file.
In addition to the default fcb which is set-up at address tfcb, the CCP also constructs a second default fcb at address tfcb+16 (i.e., the disk map field of the fcb at tbase). Thus, if the user types

```plaintext
PROGNAME X.ZOT Y.ZAP
```

the file PROGNAME.COM is loaded to the TPA, and the default fcb at tfcb is initialized to the filename X with filetype ZOT. Since the user typed a second file name, the 16 byte area beginning at tfcb + 16 is also initialized with the filename Y and filetype ZAP. It is the responsibility of the program to move this second filename and filetype to another area (usually a separate file control block) before opening the file which begins at tbase, since the open operation will fill the disk map portion, thus overwriting the second name and type.

If no file names were specified in the original command, then the fields beginning at tfcb and tfcb + 16 both contain blanks (20H). If one file name was specified, then the field at tfcb + 16 contains blanks. If the filetype is omitted, then the field is assumed to contain blanks. In all cases, the CCP translates lower case alphabets to upper case to be consistent with the CP/M file naming conventions.

As an added programming convenience, the default buffer at tbuff is initialized to hold the entire command line past the program name. Address tbuff contains the number of characters, and tbuff+1, tbuff+2, ..., contain the remaining characters up to, but not including, the carriage return. Given that the above command has been typed at the console, the area beginning at tbuff is set up as follows:

```plaintext
tbuff:
+0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11 +12 +13 +14 +15
```

where 12 is the number of valid characters (in binary), and \( X \) represents an ASCII blank. Characters are given in ASCII upper case, with uninitialized memory following the last valid character.

Again, it is the responsibility of the program to extract the information from this buffer before any file operations are performed since the FDOS uses the tbuff area to perform directory functions.

In a standard CP/M system, the following values are assumed:

```plaintext
boot: 0000H  bootstrap load (warm start)
entry: 0005H  entry point to FDOS
tfcb: 0015H  first default file control block
tfcb+16 006CH second file name
tbuff 0080H  default buffer address
tbase: 0100H  base of transient area
```
Figure 2. File Control Block Format

<table>
<thead>
<tr>
<th>FIELD</th>
<th>FCB POSITIONS</th>
<th>PURPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ET</td>
<td>0</td>
<td>Entry type (currently not used, but assumed zero)</td>
</tr>
<tr>
<td>FN</td>
<td>1-8</td>
<td>File name, padded with ASCII blanks</td>
</tr>
<tr>
<td>FT</td>
<td>9-11</td>
<td>File type, padded with ASCII blanks</td>
</tr>
<tr>
<td>EX</td>
<td>12</td>
<td>File extent, normally set to zero</td>
</tr>
<tr>
<td></td>
<td>13-14</td>
<td>Not used, but assumed zero</td>
</tr>
<tr>
<td>RC</td>
<td>15</td>
<td>Record count is current extent Size (0 to 128 records)</td>
</tr>
<tr>
<td>DM</td>
<td>16-31</td>
<td>Disk allocation map, filled-in and used by CP/M</td>
</tr>
<tr>
<td>NR</td>
<td>32</td>
<td>Next record number to read or write</td>
</tr>
</tbody>
</table>
3.3 Disk Access Primitives

Given that a program has properly initialized the FCB's for each of its files, there are several operations which can be performed, as shown in Table II. In each case, the operation is applied to the currently selected disk (see the disk select operation in Table II), using the file information in a specific FCB. The following PL/M program segment, for example, copies the contents of the file X.Y to the (new) file NEW.FIL:

DECLARE RET BYTE;

OPEN:  PROCUREMENT (A)
    DECLARE A ADDRESS;
    RET=MON2(15,A);
    END OPEN;

CLOSE: PROCUREMENT (A);
    DECLARE A ADDRESS;
    RET=MON2(16,A);
    END;

MAKE:  PROCUREMENT (A);
    DECLARE A ADDRESS;
    RET=MON2(22,A);
    END MAKE;

DELETE: PROCUREMENT (A);
    DECLARE A ADDRESS;
    /\* IGNORE RETURNED VALUE */
    CALL MON1(19,A);
    END DELETE;

READBF: PROCUREMENT (A);
    DECLARE A ADDRESS;
    RET=MON2(20,A);
    END READBF;

WRITEBF: PROCUREMENT (A);
    DECLARE A ADDRESS;
    RET=MON2(21,A);
    END WRITEBF;

INIT:  PROCUREMENT;
    CALL MON1(13,0);
    END INIT;

/\* SET UP FILE CONTROL BLOCKS */
DECLARE FCB1 (33) BYTE
    INITIAL (0,'X' ',Y',0,0,0,0),
FCB2 (33) BYTE
    INITIAL (0,'NEW ', 'FIL',0,0,0,0);
CALL INIT;
/* ERASE 'NEW.FIL' IF IT EXISTS */
CALL DELETE (.FCB2);
/* CREATE 'NEW.FIL' AND CHECK SUCCESS */
CALL MAKE (.FCB2);
IF RET = 255 THEN CALL PRINT (.NO DIRECTORY SPACE $);
ELSE
   DO; /* FILE SUCCESSFULLY CREATED, NOW OPEN 'X.Y' */
   CALL OPEN (.FCB1);
   IF RET = 255 THEN CALL PRINT (.FILE NOT PRESENT $);
   ELSE
      DO; /* FILE X.Y FOUND AND OPENED, SET
      NEXT RECORD TO ZERO FOR BOTH FILES */
      FCB1(32), FCB2(32) = 0;
      /* READ FILE X.Y UNTIL EOF OR ERROR */
      CALL READBF (.FCB1); /*READ TO 80H*/
      DO WHILE RET = 0;
         CALL WRITEBF (.FCB2) /*WRITE FROM 80H*/
         IF RET = 0 THEN /*GET ANOTHER RECORD*/
            CALL READBF (.FCB1); ELSE
            CALL PRINT (.DISK WRITE ERROR $);
      END;
      IF RET <> 1 THEN CALL PRINT (.TRANSFER ERROR $);
      ELSE
         DO; CALL CLOSE (.FCB2);
         IF RET = 255 THEN CALL PRINT (.CLOSE ERROR$);
      END;
   END;
END;
EOF

This program consists of a number of utility procedures for opening, closing, creating, and deleting files, as well as two procedures for reading and writing data. These utility procedures are followed by two FCB's for the input and output files. In both cases, the first 16 bytes are initialized to the <filename> and <filetype> of the input and output files. The main program first initializes the disk system, then deletes any existing copy of "NEW.FIL" before starting. The next step is to create a new directory entry (and empty file) for "NEW.FIL". If file creation is successful, the input file "X.Y" is opened. If this second operation is also successful, then the disk to disk copy can proceed. The NR fields are set to zero so that the first record of each file is accessed on subsequent disk I/O operations. The first call to READBF fills the (implied) DMA buffer at 80H with the first record from X.Y. The loop which follows copies the record at 80H to "NEW.FIL" and then reports any errors, or reads another 128 bytes from X.Y. This transfer operation continues until either all data has been transferred, or an error condition arises. If an error occurs, it is reported; otherwise the new file is closed and the program halts.
<table>
<thead>
<tr>
<th>FUNCTION/NUMBER</th>
<th>ENTRY PARAMETERS</th>
<th>TYPICAL CALL</th>
<th>RETURNED VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lift Head</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Initialize BDOS and select disk &quot;A&quot;</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Set DMA address to 80H</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Log-in and select disk X</td>
<td>An integer value corresponding to the log-in: A=0, B=1, C=2, etc.</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Open file</td>
<td>Address of the FCB for the file to be accessed</td>
<td>Byte address of the FCB in the directory, if found, or 255 if not present. The DM bytes are set by the BDOS.</td>
<td>File opened</td>
</tr>
<tr>
<td>Close file</td>
<td>Address of an FCB which has been previously created or opened</td>
<td>File closed</td>
<td>File closed</td>
</tr>
<tr>
<td>FUNCTION/NUMBER</td>
<td>ENTRY PARAMETERS</td>
<td>RETURNED VALUE</td>
<td>TYPICAL CALL</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------</td>
<td>----------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Search for file 17</td>
<td>Address of FCB containing &lt;filename&gt; and &lt;filetype&gt; to match. ASCII &quot;?&quot; in FCB matches any character.</td>
<td>Byte address of first FCB in directory that matches input FCB, if any; otherwise 255 indicates no match.</td>
<td>I = MON2(17,.FCB)</td>
</tr>
<tr>
<td>Search for next occurrence 18</td>
<td>Same as above, but called after function 17 (no other intermediate BDOS calls allowed)</td>
<td>Byte address of next</td>
<td>I = MON2(18,.FCB)</td>
</tr>
<tr>
<td>Delete File 19</td>
<td>Address of FCB containing &lt;filename&gt; and &lt;filetype&gt; of file to delete from diskette</td>
<td>None</td>
<td>I = MON2(19,.FCB)</td>
</tr>
<tr>
<td>Read Next Record 20</td>
<td>Address of FCB of a successfully OPENed file, with NR set to the next record to read (see note1)</td>
<td>0 = successful read 1 = read past end of file 2 = reading unwritten data in random access</td>
<td>I = MON2(20,.FCB)</td>
</tr>
</tbody>
</table>

Note1: The I/O operations transfer data to/from address 80H for the next 128 bytes unless the DMA address has been altered (see function 26). Further, the NR field of the FCB is automatically incremented after the operation. If the NR field exceeds 128, the next extent is opened automatically and the NR field is reset to zero.
<table>
<thead>
<tr>
<th>FUNCTION/NUMBER</th>
<th>ENTRY PARAMETERS</th>
<th>RETURNED VALUE</th>
<th>TYPICAL CALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Next Record 21</td>
<td>Same as above, except NR is set to the next record to write</td>
<td>0 = successful write 1 = error in extending file 2 = end of disk data 255 = no more directory space</td>
<td>I = MON2(21,.FCB)</td>
</tr>
<tr>
<td>Make File 22</td>
<td>Address of FCB with &lt;filename&gt; and &lt;file-type&gt; set. Directory entry is created, the file is initialized to empty.</td>
<td>Byte address of directory entry allocated to the FCB, or 255 if no directory space is available</td>
<td>I = MON2(22,.FCB)</td>
</tr>
<tr>
<td>Rename FCB 23</td>
<td>Address of FCB with old FN and FT in first 16 bytes, and new FN and FT in second 16 bytes</td>
<td>None</td>
<td>I = MON2(23,.FCB)</td>
</tr>
<tr>
<td>FUNCTION/NUMBER</td>
<td>ENTRY PARAMETERS</td>
<td>RETURNED VALUE</td>
<td>TYPICAL CALL</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------</td>
<td>---------------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Interrogate login vector</td>
<td>None</td>
<td>Byte value with &quot;1&quot; in bit positions of &quot;on line&quot; disks, with least significant bit corresponding to disk &quot;A&quot;</td>
<td>I = MON2(24,0)</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set DMA address</td>
<td>Address of 128 byte DMA buffer</td>
<td>None Subsequent disk I/O takes place at specified address in memory</td>
<td>CALL MON1(26,2000H)</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrogate Allocation</td>
<td>None</td>
<td>Address of the allocation vector for the current disk (used by STATUS command)</td>
<td>MON3: PROCEDURE(...) ADDRESS;</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td>A = MON3(27,0);</td>
</tr>
<tr>
<td>Interrogate Drive number</td>
<td>None</td>
<td>Disk number of currently logged disk (i.e., the drive which will be used for the next disk operation</td>
<td>I = MON2(25,0);</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.4 Random Access

Recall that a single FCB describes up to a 16K segment of a (possibly) larger file. Random access within the first 16K segment is accomplished by setting the NR field to the record number of the record to be accessed before the disk I/O takes place. Note, however, that if the 128th record is written, then the BDOS automatically increments the extent field (EX), and opens the next extent, if possible. In this case, the program must explicitly decrement the EX field and re-open the previous extent. If random access outside the first 16K segment is necessary, then the extent number e be explicitly computed, given an absolute record number r as

\[ e = \left\lfloor \frac{r}{128} \right\rfloor \]

or equivalently,

\[ e = \text{SHR}(r, 7) \]

this extent number is then placed in the EX field before the segment is opened. The NR value n is then computed as

\[ n = r \mod 128 \]

or

\[ n = r \text{ AND } 7FH. \]

When the programmer expects considerable cross-segment accesses, it may save time to create an FCB for each of the 16K segments, open all segments for access, and compute the relevant FCB from the absolute record number r.

4. SYSTEM GENERATION

As mentioned previously, every diskette used under CP/M is assumed to contain the entire system (excluding transient commands) on the first two tracks. The operating system need not be present, however, if the diskette is only used as secondary disk storage on drives B, C, ..., since the CP/M system is loaded only from drive A.

The CP/M file system is organized so that an IBM-compatible diskette from the factory (or from a vendor which claims IBM compatibility) looks like a diskette with an empty directory. Thus, the user must first copy a version of the CP/M system from an existing diskette to the first two tracks of the new diskette, followed by a sequence of copy operations, using PIP, which transfer the transient command files from the original diskette to the new diskette.
NOTE: before you begin the CP/M copy operation, read your Licensing Agreement. It gives your exact legal obligations when making reproductions of CP/M in whole or in part, and specifically requires that you place the copyright notice on each diskette which results from the copy operation.

4.1. Initializing CP/M from an Existing Diskette

The first two tracks are placed on a new diskette by running the transient command SYSGEN, as described in the document "An Introduction to CP/M Features and Facilities." The SYSGEN operation brings the CP/M system from an initialized diskette into memory, and then takes the memory image and places it on the new diskette.

Upon completion of the SYSGEN operation, place the original diskette on drive A, and the initialized diskette on drive B. Reboot the system; the response should be

A> indicating that drive A is active. Log into drive B by typing

B:

and CP/M should respond with

B>

indicating that drive B is active. If the diskette in drive B is factory fresh, it will contain an empty directory. Non-standard diskettes may, however, appear as full directories to CP/M, which can be emptied by typing

ERA *. *

when the diskette to be initialized is active. Do not give the ERA command if you wish to preserve files on the new diskette since all files will be erased with this command.

After examining disk B, reboot the CP/M system and return to drive A for further operations.

The transient commands are then copied from drive A to drive B using the PIP program. The sequence of commands shown below, for example, copy the principal programs from a standard CP/M diskette to the new diskette:

A> PIP
*B:STAT.COM=STAT.COM
*B:PIP.COM=PIP.COM
*B:LOAD.COM=LOAD.COM
*B:ED.COM=ED.COM
*B:ASM.COM=ASM.COM
*B:SYSGEN.COM=SYSGEN.COM
*B:DDT.COM=DDT.COM
*
A

The user should then log in disk B, and type the command

DIR *.*

to ensure that the files were transferred to drive B from drive A. The various programs can then be tested on drive B to check that they were transferred properly.

Note that the copy operation can be simplified somewhat by creating a "submit" file which contains the copy commands. The file could be named GEN.SUB, for example, and might contain

SYSGEN
PIP B:STAT.COM=STAT.COM,
PIP B:PIP.COM=PIP.COM,
PIP B:LOAD.COM=LOAD.COM,
PIP B:ED.COM=ED.COM,
PIP B:ASM.COM=ASM.COM,
PIP B:SYSGEN.COM=SYSGEN.COM,
PIP B:DDT.COM=DDT.COM,

The generation of a new diskette from the standard diskette is then done by typing simply

SUBMIT GEN,

5. CP/M ENTRY POINT SUMMARY

The functions shown below summarize the functions of the FDOS. The function number is passed in Register C (first parameter in PL/M), and the information is passed in Registers D,E (second PL/M parameter). Single byte results are returned in Register A. If a double byte result is returned, then the high-order byte comes back in Register B (normal PL/M return). The transient program enters the FDOS through location "entry" (see Section 7.) as shown in Section 2. for PL/M, or

CALL entry

in assembly language. All registers are altered in the FDOS.
<table>
<thead>
<tr>
<th>Function</th>
<th>Number</th>
<th>Information</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>System Reset</td>
<td></td>
<td>ASCII character</td>
</tr>
<tr>
<td>1</td>
<td>Read Console</td>
<td></td>
<td>ASCII character</td>
</tr>
<tr>
<td>2</td>
<td>Write Console</td>
<td>ASCII character</td>
<td>ASCII character</td>
</tr>
<tr>
<td>3</td>
<td>Read Reader</td>
<td></td>
<td>ASCII character</td>
</tr>
<tr>
<td>4</td>
<td>Write Punch</td>
<td></td>
<td>ASCII character</td>
</tr>
<tr>
<td>5</td>
<td>Write List</td>
<td></td>
<td>ASCII character</td>
</tr>
<tr>
<td>6</td>
<td>(not used)</td>
<td></td>
<td>I/O Status Byte</td>
</tr>
<tr>
<td>7</td>
<td>Interrogate I/O Status</td>
<td>I/O Status Byte</td>
<td>ASCII character</td>
</tr>
<tr>
<td>8</td>
<td>Alter I/O Status</td>
<td>Buffer Address</td>
<td>Buffer Address</td>
</tr>
<tr>
<td>9</td>
<td>Print Console Buffer</td>
<td>Buffer Address</td>
<td>Buffer Address</td>
</tr>
<tr>
<td>10</td>
<td>Read Console Buffer</td>
<td>Buffer Address</td>
<td>Buffer Address</td>
</tr>
<tr>
<td>11</td>
<td>Check Console Status</td>
<td>True if character Ready</td>
<td>True if character Ready</td>
</tr>
<tr>
<td>12</td>
<td>Lift Disk Head</td>
<td>Disk number</td>
<td>Completion Code</td>
</tr>
<tr>
<td>13</td>
<td>Reset Disk System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Select Disk</td>
<td>Disk number</td>
<td>Completion Code</td>
</tr>
<tr>
<td>15</td>
<td>Open File</td>
<td>FCB Address</td>
<td>Completion Code</td>
</tr>
<tr>
<td>16</td>
<td>Close File</td>
<td>&quot; &quot;</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>17</td>
<td>Search First</td>
<td>&quot; &quot;</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>18</td>
<td>Search Next</td>
<td>&quot; &quot;</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>19</td>
<td>Delete File</td>
<td>&quot; &quot;</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>20</td>
<td>Read Record</td>
<td>&quot; &quot;</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>21</td>
<td>Write Record</td>
<td>&quot; &quot;</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>22</td>
<td>Create File</td>
<td>&quot; &quot;</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>23</td>
<td>Rename File</td>
<td>&quot; &quot;</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>24</td>
<td>Interrogate Login</td>
<td>Login Vector</td>
<td>Login Vector</td>
</tr>
<tr>
<td>25</td>
<td>Interrogate Disk</td>
<td>Selected Disk Number</td>
<td>Selected Disk Number</td>
</tr>
<tr>
<td>26</td>
<td>Set DMA Address</td>
<td>DMA Address</td>
<td>Address of Allocation Vector</td>
</tr>
<tr>
<td>27</td>
<td>Interrogate Allocation</td>
<td>DMA Address</td>
<td>Address of Allocation Vector</td>
</tr>
</tbody>
</table>
6. ADDRESS ASSIGNMENTS

The standard distribution version of CP/M is organized for an Intel MDS microcomputer developmental system with 16K of main memory, and two diskette drives. Larger systems are available in 16K increments, providing management of 32K, 48K, and 64K systems (the largest MDS system is 62K since the ROM monitor provided with the MDS resides in the top 2K of the memory space). For each additional 16K increment, add 4000H to the values of cbase and fbase.

The address assignments are

<table>
<thead>
<tr>
<th>Address</th>
<th>Value (H)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boot</td>
<td>0000H</td>
<td>warm start operation</td>
</tr>
<tr>
<td>tfcb</td>
<td>005CH</td>
<td>default file control block location</td>
</tr>
<tr>
<td>buff</td>
<td>0080H</td>
<td>default buffer location</td>
</tr>
<tr>
<td>base</td>
<td>0100H</td>
<td>base of transient program area</td>
</tr>
<tr>
<td>cbase</td>
<td>2900H</td>
<td>base of console command processor</td>
</tr>
<tr>
<td>fbase</td>
<td>3200H</td>
<td>base of disk operating system</td>
</tr>
<tr>
<td>entry</td>
<td>0005H</td>
<td>entry point to disk system from user programs</td>
</tr>
</tbody>
</table>
7. SAMPLE PROGRAMS

This section contains two sample programs which interface with the CP/M operating system. The first program is written in assembly language, and is the source program for the DUMP utility. The second program is the CP/M LOAD utility, written in PL/M.

The assembly language program begins with a number of "equates" for system entry points and program constants. The equate

\[
\text{BDOS} \quad \text{EQU} \quad 0005H
\]

for example, gives the CP/M entry point for peripheral I/O functions. The default file control block address is also defined (FCB), along with the default buffer address (BUFF). Note that the program is set up to run at location 100H, which is the base of the transient program area. The stack is first set-up by saving the entry stack pointer into OLDSP, and resetting SP to the local stack. The stack pointer upon entry belongs to the console command processor, and need not be saved unless control is to return to the CCP upon exit. That is, if the program terminates with a reboot (branch to location 0000H) then the entry stack pointer need not be saved.

The program then jumps to MAIN, past a number of subroutines which are listed below:

- **BREAK** - when called, checks to see if there is a console character ready. BREAK is used to stop the listing at the console.

- **PCHAR** - print the character which is in register A at the console.

- **CRLF** - send carriage return and line feed to the console.

- **PNIB** - print the hexadecimal value in register A in ASCII at the console.

- **PHEX** - print the byte value (two ASCII characters) in register A at the console.

- **ERR** - print error flag #n at the console, where n is
  
  1 if file cannot be opened
  2 if disk read error occurred

- **GNB** - get next byte of data from the input file. If the IBP (input buffer pointer) exceeds the size of the input buffer, then another disk record of 128 bytes is read. Otherwise, the next character in the buffer is returned. IBP is updated to point to the next character.
The MAIN program then appears, which begins by calling SETUP. The SETUP subroutine, discussed below, opens the input file and checks for errors. If the file is opened properly, the GLOOP (get loop) label gets control.

On each successive pass through the GLOOP label, the next data byte is fetched using GNB and save in register B. The line addresses are listed every sixteen bytes, so there must be a check to see if the least significant 4 bits is zero on each output. If so, the line address is taken from registers h and l, and typed at the left of the line. In all cases, the byte which was previously saved in register B is brought back to register A, following label NONUM, and printed in the output line. The cycle through GLOOP continues until an end of file condition is detected in DISKR, as described below. Thus, the output lines appear as

```
0000  bb bb bb bb bb bb bb bb bb bb bb bb
0010  bb bb bb bb bb bb bb bb bb bb bb bb
...
```

until the end of file.

The label FINIS gets control upon end of file. CRLF is called first to return the carriage from the last line output. The CCP stack pointer is then reclaimed from OLDSP, followed by a RET to return to the console command processor. Note that a JMP 0000H could be used following the FINIS label, which would cause the CP/M system to be brought in again from the diskette (this operation is necessary only if the CCP has been overlayed by data areas).

The file control block format is then listed (FCBDN ... FCBLN) which overlays the fcb at location 05CH which is setup by the CCP when the DUMP program is initiated. That is, if the user types

```
DUMP X.Y
```

then the CCP sets up a properly formed fcb at location 05CH for the DUMP (or any other) program when it goes into execution. Thus, the SETUP subroutine simply addresses this default fcb, and calls the disk system to open it. The DISKR (disk read) routine is called whenever GNB needs another buffer full of data. The default buffer at location 80H is used, along with a pointer (IBP) which counts bytes as they are processed. Normally, an end of file condition is taken as either an ASCII 1AH (control-z), or an end of file detection by the DOS. The file dump program, however, stops only on a DOS end of file.
FILE DUMP PROGRAM, READS AN INPUT FILE AND PRINTS IN

COPYRIGHT (C), DIGITAL RESEARCH, 1975, 1976

ORG 100H
BDOS EQU 0005H ;DOS ENTRY POINT
OPENF EQU 15 ;FILE OPEN
READF EQU 20 ;READ FUNCTION
TYPEF EQU 2 ;TYPE FUNCTION
CONSF EQU 1 ;READ CONSOLE
BRKF EQU 11 ;BREAK KEY FUNCTION (TRUE IF CHAR READY)
FCB EQU 5CH ;FILE CONTROL BLOCK ADDRESS
BUFF EQU 80H ;INPUT DISK BUFFER ADDRESS

SET UP STACK
LXI H,0
DAD SP
SHLD OLDSP
LXI SP,STKTOP
JMP MAIN

IBF: DS 2 ;INPUT BUFFER POINTER

STACK AREA
OLDSP: DS 2
STACK: DS 64
STKTOP EQU $5000H

SUBROUTINES
BREAK: ;CHECK BREAK KEY (ACTUALLY ANY KEY WILL DO)
PUSH H! PUSH D! PUSH B; ENVIRONMENT SAVED
MVI C,BRKF
CALL BDOS
POP B! POP D! POP H; ENVIRONMENT RESTORED
RET

PCHAR: ;PRINT A CHARACTER
PUSH H! PUSH D! PUSH B; SAVED
MVI C,TYPENF
MOV E,A
CALL BDOS
POP B! POP D! POP H; RESTORED
RET

CRLF:
MVI A,0DH
CALL PCHAR
MVI A,0AH
CALL PCHAR
RET

PNIB: ;PRINT NIBBLE IN REG A
ANI 0FH ;LOW 4 BITS
CPI 10
JNC P10
; LESS THAN OR EQUAL TO 9
017C C630  ADI '0'
017E C38301  JMP PRN

; GREATER OR EQUAL TO 10
0181 C637  P10: ADI 'A' - 10
0183 CD5D01  PRN: CALL PCHAR
0186 C9  RET

; PHEX: ;PRINT HEX CHAR IN REG A
0187 F5  PUSH PSW
0188 0F  RRC
0189 0F  RRC
018A 0F  RRC
018B 0F  RRC
018C CD7501  CALL PNIB ;PRINT NIBBLE
018F F1  POP PSW
0190 CD7501  CALL PNIB
0193 C9  RET

; ERR: ;PRINT ERROR MESSAGE
0194 CD6A01  CALL CRLF
0197 3E23  MVI A, '#'
0199 CD5D01  CALL PCHAR
019C 78  MOV A,B
019D C630  ADI '0'
019F CD5D01  CALL PCHAR
01A2 CD6A01  CALL CRLF
01A5 C3F701  JMP FINIS

; GNB: ;GET NEXT BYTE
01A8 3A0D01  LDA IBP
01AB FE80  CPI 80H
01AD C2B401  JNZ G0
; READ ANOTHER BUFFER

; 01B0 CD1602  CALL DISKR
01B3 AF  XRA A

; G0: ;READ THE BYTE AT BUFF+REG A
01B4 5F  MOV E,A
01B5 1600  MVI D,0
01B7 3C  INR A
01B8 320D01  STA IBP
; POINTER IS INCREMENTED
; SAVE THE CURRENT FILE ADDRESS
01BB E5  PUSH H
01BC 218000  LXI H,BUFF
01BF 19  DAD D
01C0 7E  MOV A,M
; BYTE IS IN THE ACCUMULATOR

; RESTORE FILE ADDRESS AND INCREMENT
01C1 E1  POP H
01C2 23  INX H
01C3 C9  RET

; MAIN: ; READ AND PRINT SUCCESSIVE BUFFERS
01C4 CDFF01  CALL SETUP ;SET UP INPUT FILE
3E80 MVI A, 80H
32D01 STA IBP ;SET BUFFER POINTER TO 80H
21FF LXI H,0FFFH ;SET TO -1 TO START

CDA801 CALL GNB
D2 47 MOV B,A
; PRINT HEX VALUES

7D MOV A,L
E60F ANI OFH ;CHECK LOW 4 BITS
2EB01 JNZ NONUM

CD6A01 CALL CRLF

DC5101 CALL BREAK
D0F 0F RRC
DAF701 JC FINIS ;DON'T PRINT ANY MORE

7C MOV A,H
4D8701 CALL PHEX
7D MOV A,L
8D8701 CALL PHEX

3E20 MVI A, '
5D01 CALL PCHAR
78 MOV A, B
8701 CALL PHEX

C3CF01 JMP GLOOP

36A01 CALL CRLF
2A0F01 LHLDD OLDSP
F9 SPHL
C9 RET

FILE CONTROL BLOCK DEFINITIONS

FCBDN EQU FCB+0 ;DISK NAME
FCBFN EQU FCB+1 ;FILE NAME
FCBPT EQU FCB+9 ;DISK FILE TYPE (3 CHARACTERS)
FCBRL EQU FCB+12 ;FILE'S CURRENT REEL NUMBER
FCBRC EQU FCB+15 ;FILE'S RECORD COUNT (0 TO 128)
FCBCR EQU FCB+32 ;CURRENT (NEXT) RECORD NUMBER (0 TO 127)
FCBLN EQU FCB+33 ;FCB LENGTH

SETUP: ;SET UP FILE
; OPEN THE FILE FOR INPUT
115C00 LXI D, FCB
020F MVI C, OPENF
D0500 CALL BDOS
; CHECK FOR ERRORS
FEFF CPI 255
C21102 JNZ OPNOK
; BAD OPEN
MVI B,1 ;OPEN ERROR
CALL ERR

; OPNOK: ;OPEN IS OK.
XRA A
STA FCBCR
RET

; DISKR: ;READ DISK FILE RECORD
PUSH H! PUSH D! PUSH B
LXI D,FCB
MVI C,READF
CALL BDOS
POP B! POP D! POP H
CPI 0 ;CHECK FOR ERRS
R2
MAY BE EOF
CPI 1
J2 FINIS

; MVI B,2 ;DISK READ ERROR
CALL ERR

; END
The PL/M program which follows implements the CP/M LOAD utility. The function is as follows. The user types

```
LOAD filename
```

If filename.HEX exists on the diskette, then the LOAD utility reads the "hex" formatted machine code file and produces the file

```
filename.COM
```

where the COM file contains an absolute memory image of the machine code, ready for load and execution in the TPA. If the file does not appear on the diskette, the LOAD program types

```
SOURCE IS READER
```

and reads an Addmaster paper tape reader which contains the hex file.

The LOAD program is set up to load and run in the TPA, and, upon completion, return to the CCP without rebooting the system. Thus, the program is constructed as a single procedure called LOADCOM which takes the form

```
OFAH:
LOADCOM: PROCEDURE;
/* LIBRARY PROCEDURES */
MON1: ...
/* END LIBRARY PROCEDURES */
MOVE: ...
GETCHAR: ...
PRINTNIB: ...
PRINTHEX: ...
PRINTADDR: ...
RELOC: ...
    SETMEM:
    READHEX:
    READBYTE:
    READCS:
    MAKEDOUBLE:
    DIAGNOSE:
END RELOC;
DECLARE STACK(16) ADDRESS, SP ADDRESS;
SP = STACKPTR; STACKPTR = .STACK(LENGTH(STACK));
...
CALL RELOC;
...
STACKPTR = SP;
RETURN 0;
END LOADCOM;
```

EOF
The label OFAH at the beginning sets the origin of the compilation to OFAH, which causes the first 6 bytes of the compilation to be ignored when loaded (i.e., the TPA starts at location 100H and thus OFAH,...,OFFH are deleted from the COM file). In a PL/M compilation, these 6 bytes are used to set up the stack pointer and branch around the subroutines in the program. In this case, there is only one subroutine, called LOADCOM, which results in the following machine memory image for LOAD

```
OFAH: LXI SP,plmstack ;SET SP TO DEFAULT STACK
OFFH: JMP pastsubr ;JUMP AROUND LOADCOM
100H: beginning of LOADCOM procedure
      ....
      end of LOADCOM procedure
      RET
      pastsubr:
      EI
      HLT
```

Since the machine code between OFAH and OFFH is deleted in the load, execution actually begins at the top of LOADCOM. Note, however, that the initialization of the SP to the default stack has also been deleted; thus, there is a declaration and initialization of an explicit stack and stack pointer before the call to RELOC at the end of LOADCOM. This is necessary only if we wish to return to the CCP without a reboot operation: otherwise the origin of the program is set to 100H, the declaration of LOADCOM as a procedure is not necessary, and termination is accomplished by simply executing a

```
GO TO 0000H;
```

at the end of the program. Note also that the overhead for a system reboot is not great (approximately 2 seconds), but can be bothersome for system utilities which are used quite often, and do not need the extra space.

The procedures listed in LOADCOM as "library procedures" are a standard set of PL/M subroutines which are useful for CP/M interface. The RELOC procedure contains several nested subroutines for local functions, and actually performs the load operation when called from LOADCOM. Control initially starts on line 327 where the stack pointer is saved and re-initialized to the local stack. The default file control block name is copied to another file control block (SFCB) since two files may be open at the same time. The program then calls SEARCH to see if the HEX file exists; if not, then the high speed reader is used. If the file does exist, it is opened for input (if possible). The filetype COM is moved to the default file control block area, and any existing copies of filename.COM files are removed from the diskette before creating a new file. The MAKE operation creates a new file, and, if successful, RELOC is called to read the HEX file and produce the COM file. At the end of processing by RELOC, the COM file is closed (line 350). Note that the HEX file does not need to be closed since it was opened for input only. The data written to a file is not permanently recorded until the file is successfully closed.
Disk input characters are read through the procedure GETCHAR on line 137. Although the DMA facilities of CP/M could be used here, the GETCHAR procedure instead uses the default buffer at location 80H and moves each buffer into a vector called SBUFF (source buffer) as it is read. On exit, the GETCHAR procedure returns the next input character and updates the source buffer pointer (SBP).

The SETMEM procedure on line 191 performs the opposite function from GETCHAR. The SETMEM procedure maintains a buffer of loaded machine code in pure binary form which acts as a "window" on the loaded code. If there is an attempt by RELOC to write below this window, then the data is ignored. If the data is within the window, then it is placed into MBUFF (memory buffer). If the data is to be placed above this window, then the window is moved up to the point where it would include the data address by writing the memory image successively (by 128 byte buffers), and moving the base address of the window. Using this technique, the programmer can recover from checksum errors on the high-speed reader by stopping the reader, rewinding the tape for some distance, then restarting LOAD (in this case, LOADING is resumed by interrupting with a NOP instruction). Again, the SETMEM procedure uses the default buffer at location 80H to perform the disk output by moving 128 byte segments to 80H through OFFH before each write.
00001 1 0FAH: DECLARE BDOS LITERALLY '005H';
00002 1 /* TRANSIENT COMMAND LOADER PROGRAM
00003 1
00004 1
00005 1
00006 1
00007 1 */
00008 1
00009 1 LOADCOM: PROCEDURE BYTE;
00010 2 DECLARE FCBA ADDRESS INITIAL(5CH);
00011 2 DECLARE FCB BASED FCBA (33) BYTE;
00012 2
00013 2 DECLARE BUFFA ADDRESS INITIAL(80H), /* I/O BUFFER ADDR
00014 2 BUFFER BASED BUFFA (128) BYTE;
00015 2
00016 2 DECLARE SFCB(33) BYTE, /* SOURCE FILE CONTROL BLOCK */
00017 2
00018 2 BSIZE LITERALLY '1024',
00019 2 EOFILE LITERALLY '1AH',
00020 2 SBUFF(BSIZE) BYTE /* SOURCE FILE BUFFER */
00021 2 INITIAL(EOFILE),
00022 2 RFLAG BYTE, /* READER FLAG */
00023 2 SBP ADDRESS; /* SOURCE FILE BUFFER POINTER
00024 2
00025 2 /* LOADCOM LOADS TRANSIENT COMMAND FILES TO THE DISK F
00026 2 ROM THE CURRENTLY DEFINED READER PERIPHERAL. THE LOADER PLACE
00027 2 THE MACH CODE INTO A FILE WHICH APPEARS IN THE LOADCOM COMMAND
00028 2 */
00029 2 */ *************** LIBRARY PROCEDURES FOR DISKIO **********
00030 2
00031 2 MON1: PROCEDURE(F,A);
00032 3 DECLARE F BYTE,
00033 3 A ADDRESS;
00034 3 GO TO BDOS;
00035 3 END MON1;
00036 2 MON2: PROCEDURE(F,A) BYTE;
00037 3 DECLARE F BYTE,
00038 3 A ADDRESS;
00039 3 GO TO BDOS;
00040 3 END MON2;
00041 2 READRDR: PROCEDURE BYTE;
00042 3 /* READ CURRENT READER DEVICE */
00043 3 RETURN MON2(3,0);
00044 3 END READRDR;
00045 2
00046 2 DECLARE
00047 2 TRUE LITERALLY '1',
00048 2 FALSE LITERALLY '0',
00049 2 FOREVER LITERALLY 'WHILE TRUE',
00050 2 CR LITERALLY '13',
LF LITERALLY '10',
WHAT LITERALLY '63';
PRINTCHAR: PROCEDURE(CHAR);
DECLARE CHAR BYTE;
CALL MON1(2,CHAR);
END PRINTCHAR;
CRLF: PROCEDURE;
CALL PRINTCHAR(CR);
CALL PRINTCHAR(LF);
END CRLF;
PRINT: PROCEDURE(A);
DECLARE A ADDRESS;
/* PRINT THE STRING STARTING AT ADDRESS A UNTIL THE 
NEXT DOLLAR SIGN IS ENCLOSED */
CALL CRLF;
CALL MON1(9,A);
END PRINT;
DECLARE DCNT BYTE;
INITIALIZE: PROCEDURE;
CALL MON1(13,0);
END INITIALIZE;
SELECT: PROCEDURE(D);
DECLARE D BYTE;
CALL MON1(14,D);
END SELECT;
OPEN: PPROCEDURE(FCB);
DECLARE FCB ADDRESS;
DCNT = MON2(15,FCB);
END OPEN;
CLOSE: PROCEDURE(FCB);
DECLARE FCB ADDRESS;
DCNT = MON2(16,FCB);
END CLOSE;
SEARCH: PROCEDURE(FCB);
DECLARE FCB ADDRESS;
DCNT = MON2(17,FCB);
END SEARCH;
SEARCHN: PROCEDURE;
DCNT = MON2(18,0);
END SEARCHN;
DELETE: PROCEDURE(FCB);
DECLARE FCB ADDRESS;
CALL MON1(19,FCB);
END DELETE;
DISKREAD: PROCEDURE(FCB) BYTE;
DECLARE FCB ADDRESS;
RETURN MON2(20,FCB);
END DISKREAD;
DISKWRITE: PROCEDURE(FCB) BYTE;
DECLARE FCB ADDRESS;
RETURN MON2(21, FCB);
END DISKWRITE;

MAKE: PROCEDURE(FCB);
DECLARE FCB ADDRESS;
DCNT = MON2(22, FCB);
END MAKE;

RENAME: PROCEDURE(FCB);
DECLARE FCB ADDRESS;
CALL MON1(23, FCB);
END RENAME;

00127 2

/* *************** END OF LIBRARY PROCEDURES ***********/

MOVE: PROCEDURE(S, D, N);
DECLARE (S, D) ADDRESS, N BYTE,
A BASED S BYTE, B BASED D BYTE;
DO WHILE (N := N-1) <> 255;
B = A; S=S+1; D=D+1;
END;
END MOVE;

GETCHAR: PROCEDURE BYTE;
/* GET NEXT CHARACTER */
DECLARE I BYTE;
IF RFLAG THEN RETURN READRDR;
IF (SBP := SBP+1) <= LAST(SBUFF) THEN
RETURN SBUF(SBP);
/* OTHERWISE READ ANOTHER BUFFER FULL */
DO SBP = 0 TO LAST(SBUFF) BY 128;
IF (I := DISKREAD(.SPCB)) = @ THEN
CALL MOVE(80H, SBUFF(SBP), 80H); ELSE
DO; IF I<1 THEN CALL PRINT(. 'DISK READ ERROR');

SBUFF(SBP) = EOFILE;
SBP = LAST(SBUFF);
END;
END GETCHAR;

DECLARE
STACKPOINTER LITERALLY 'STACKPTR';

PRINTNIB: PROCEDURE(N);
DECLARE N BYTE;
IF N > 9 THEN CALL PRINTCHAR(N+"A"-10); ELSE
CALL PRINTCHAR(N+"0");
END PRINTNIB;

PRINTHEX: PROCEDURE(B);
DECLARE B BYTE;
CALL PRINTNIB(SHR(B, 4)); CALL PRINTNIB(B AND 0FH);
END PRINTHEX;
00169 2 PRINTADDR: PROCEDURE(A);
00170 3 DECLARE A ADDRESS;
00171 3 CALL PRINTHEX(HIGH(A)); CALL PRINTHEX(LOW(A));
00172 3 END PRINTADDR;
00173 2
00174 2
00175 2 /* INTEL HEX FORMAI LOADER */
00176 2
00177 2 RELOC: PROCEDURE;
00178 3 DECLARE (RL, CS, RT) BYTE;
00179 3 DECLARE
00180 3 LA ADDRESS, /* LOAD ADDRESS */
00181 3 TA ADDRESS, /* TEMP ADDRESS */
00182 3 SA ADDRESS, /* START ADDRESS */
00183 3 FA ADDRESS, /* FINAL ADDRESS */
00184 3 NB ADDRESS, /* NUMBER OF BYTES LOADED */
00185 3 SP ADDRESS, /* STACK POINTER UPON ENTRY TO REL
00186 3 OC */
00187 3
00188 3 MBUFF(256) BYTE;
00189 3 P BYTE;
00190 3 L ADDRESS;
00191 3
00192 4 SETMEM: PROCEDURE(B);
00193 4 /* SET MBUFF TO B AT LOCATION LA MOD LENGTH(MBUFF) */
00194 4 DECLARE (B,I) BYTE;
00195 4 IF LA < L THEN /* MAY BE A RETRY */ RETURN;
00196 4 DO WHILE LA > L + LAST(MBUFF); /* WRITE A PARA
00197 4 GRAPH */
00198 4 DO I = 0 TO 127; /* COPY INTO BUFFER */
00199 5 BUFFER(I) = MBUFF(LOW(L)); L = L + 1;
00200 5 END;
00201 5 /* WRITE BUFFER ONTO DISK */
00202 5 P = P + 1;
00203 5 IF DISKWRITE(FCBA) <> 0 THEN
00204 6 DO; CALL PRINT(´DISK WRITE ERRORS´);
00205 6 HALT;
00206 6 /* RETRY AFTER INTERRUPT NOP */
00207 6 L = L - 128;
00208 6 END;
00209 5 END;
00210 5 MBUFF(LOW(LA)) = B;
00211 5 END SETMEM;
00212 3
00213 3 READHEX: PROCEDURE BYTE;
00214 4 /* READ ONE HEX CHARACTER FROM THE INPUT */
00215 4 DECLARE H BYTE;
00216 4 IF (H := GETCHAR) - ´0´ <= 9 THEN RETURN H - ´0´;
00217 4 IF H - ´A´ > 5 THEN GO TO CHARERR;
00218 4 RETURN H - ´A´ + 10;
00219 4 END READHEX;
00220 5
00221 3 READBYTE: PROCEDURE BYTE;
00222 4 /* READ TWO HEX DIGITS */
00223 4 RETURN SHL(READHEX,4) OR READHEX;
00224 4 END READBYTE;
00225 4
00226 3 READCS: PROCEDURE BYTE;
00227 4 /* READ BYTE WHILE COMPUTING CHECKSUM */
DECLARE B BYTE;
CS = CS + (B := READBYTE);
RETURN B;
END READCS;

MAKESDOUBLE: PROCEDURE(H,L) ADDRESS;
/∗ CREATE A DOUBLE BYTE VALUE FROM TWO SINGLE BYTE S ∗/
DECLARE (H,L) BYTE;
RETURN SHL(DOUBLE(H),8) OR L;
END MAKESDOUBLE;

DIAGNOSE: PROCEDURE;
DECLARE M BASED TA BYTE;
NEWLINE: PROCEDURE;
CALL CRLF; CALL PRINTADDR(TA); CALL PRINTCHAR(’:)’;
CALL PRINTCHAR(’.’);
END NEWLINE;
/* PRINT DIAGNOSTIC INFORMATION AT THE CONSOLE */
CALL PRINT(.’LOAD ADDRESS $’); CALL PRINTADDR(TA);
CALL PRINT(.’ERROR ADDRESS $’); CALL PRINTADDR(LA);
CALL PRINT(.’BYTES READ:$’); CALL NEWLINE;
DO WHILE TA < LA;
IF (LOW(TA) AND 0FH) = 0 THEN CALL NEWLINE;
CALL PRINTHEX(MBUFF(TA-L)); TA=TA+1;
CALL PRINTCHAR(’.’);
END;
CALL CRLF;
HALT;
END DIAGNOSE;

/∗ INITIALIZE ∗/
SA, PA, NB = 0;
SP = STACKPOINTER;
P = 0; /∗ PARAGRAPH COUNT ∗/
TA, LA, L = 100H; /∗ BASE ADDRESS OF TRANSIENT ROUTINES ∗/
IF FALSE THEN
CHARR: /∗ ARRIVE HERE IF NON-HEX DIGIT IS ENCOUN-
TERED ∗/
DO; /∗ RESTORE STACKPOINTER ∗/ STACKPOINTER = SP;
CALL PRINT(.’NON-HEXADECIMAL DIGIT ENCLOSED $’)
CALL DIAGNOSE;
END;

/∗ READ RECORDS UNTIL :00XXXX IS ENCOUNTERED ∗/
DO FOREVER;
/∗ SCAN THE ∗/
DO WHILE GETCHAR <> ’:’;
END;

/* PRINT DIAGNOSTIC INFORMATION AT THE CONSOLE */
CALL PRINT(.’LOAD ADDRESS $’); CALL PRINTADDR(TA);
CALL PRINT(.’ERROR ADDRESS $’); CALL PRINTADDR(LA);
CALL PRINT(.’BYTES READ:$’); CALL NEWLINE;
DO WHILE TA < LA;
IF (LOW(TA) AND 0FH) = 0 THEN CALL NEWLINE;
CALL PRINTHEX(MBUFF(TA-L)); TA=TA+1;
CALL PRINTCHAR(’.’);
END;
CALL CRLF;
HALT;
END DIAGNOSE;

/∗ INITIALIZE ∗/
SA, PA, NB = 0;
SP = STACKPOINTER;
P = 0; /∗ PARAGRAPH COUNT ∗/
TA, LA, L = 100H; /∗ BASE ADDRESS OF TRANSIENT ROUTINES ∗/
IF FALSE THEN
CHARR: /∗ ARRIVE HERE IF NON-HEX DIGIT IS ENCOUN-
TERED ∗/
DO; /∗ RESTORE STACKPOINTER ∗/ STACKPOINTER = SP;
CALL PRINT(.’NON-HEXADECIMAL DIGIT ENCLOSED $’)
CALL DIAGNOSE;
END;

/∗ READ RECORDS UNTIL :00XXXX IS ENCOUNTERED ∗/
DO FOREVER;
/∗ SCAN THE ∗/
DO WHILE GETCHAR <> ’:’;
END;
/* SET CHECK SUM TO ZERO, AND SAVE THE RECORD LENGTH */
CS = 0;
/* MAY BE THE END OF TAPE */
IF (RL := READCS) = 0 THEN
  GO TO FIN;
NB = NB + RL;
TA, LA = MAKEDOUBLE(READCS, READCS);
IF SA = 0 THEN SA = LA;

/* READ THE RECORD TYPE (NOT CURRENTLY USED) */
RT = READCS;

/* PROCESS EACH BYTE */
DO WHILE (RL := RL - 1) <> 255;
  CALL SETMEM(READCS); LA = LA+1;
END;
IF LA > FA THEN FA = LA - 1;

/* NOW READ CHECKSUM AND COMPARE */
IF CS + READBYTE <> 0 THEN
  DO; CALL PRINT(.'CHECK SUM ERROR $');
    CALL DIAGNOSE;
    END;
END;

FIN:
/* EMPTY THE BUFFERS */
TA = LA;
DO WHILE L < TA;
  CALL SETMEM(0); LA = LA+1;
END;

/* PRINT FINAL STATISTICS */
CALL PRINT('.FIRST ADDRESS $'); CALL PRINTADDR(SA);
CALL PRINT('.LAST ADDRESS $'); CALL PRINTADDR(FA);
CALL PRINT('.BYTES READ $'); CALL PRINTADDR(NB);
CALL PRINT('.RECORDS WRITTEN $'); CALL PRINTHEX(P);
CALL CRLF;

END RELOC;

/* ARRIVE HERE FROM THE SYSTEM MONITOR, READY TO READ THE HEX TAPE */

/* SET UP STACKPOINTER IN THE LOCAL AREA */
DECLARE STACK(16) ADDRESS, SP ADDRESS;
SP = STACKPOINTER; STACKPOINTER = .STACK(LENGTH(STACK));

SBP = LENGTH(SBUFF);

/* SET UP THE SOURCE FILE */
CALL MOVE(FCA, SFCB, 33);
CALL MOVE(.'HEX', 0, SFCB(9), 4);
CALL SEARCH(SFCB);
IF (RFLAG := DCTRL = 255) THEN
  CALL PRINT(.'SOURCE IS READER$'); ELSE
  DO; CALL PRINT(.'SOURCE IS DISK$');
CALL OPEN(, .SFCB);
IF DCNT = 255 THEN CALL PRINT(, "CANNOT OPEN SOURC E$")
END;
CALL CRLF;
CALL MOVE(, "COM", FCBA+9, 3);
/* REMOVE ANY EXISTING FILE BY THIS NAME */
CALL DELETE(FCBA);
/* THEN OPEN A NEW FILE */
CALL MAKE(FCBA);  FCB(32) = 0; /* CREATE AND SET NEXT RECORD */
IF DCNT = 255 THEN CALL PRINT(, "NO MORE DIRECTORY SPACES",
) ELSE
DO; CALL RELOC;
CALL CLOSE(FCBA);
IF DCNT = 255 THEN CALL PRINT(, "CANNOT CLOSE FILES",
);
END;
CALL CRLF;
/* RESTORE STACKPOINTER FOR RETURN */
STACKPOINTER = SP;
RETURN 0;
END LOADCOM;
EOF
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. FIRST LEVEL SYSTEM REGENERATION</td>
<td>2</td>
</tr>
<tr>
<td>3. SECOND LEVEL SYSTEM REGENERATION</td>
<td>6</td>
</tr>
<tr>
<td>4. SAMPLE GETSYS AND PUTSYS PROGRAMS</td>
<td>10</td>
</tr>
<tr>
<td>5. DISKETTE ORGANIZATION</td>
<td>12</td>
</tr>
<tr>
<td>6. THE BIOS ENTRY POINTS</td>
<td>14</td>
</tr>
<tr>
<td>7. A SAMPLE BIOS</td>
<td>20</td>
</tr>
<tr>
<td>8. A SAMPLE COLD START LOADER</td>
<td>20</td>
</tr>
<tr>
<td>9. RESERVED LOCATIONS IN PAGE ZERO</td>
<td>21</td>
</tr>
<tr>
<td>10. NOTES FOR USERS OF CP/M VERSION 1.3</td>
<td>23</td>
</tr>
</tbody>
</table>

## Appendix

A. THE MDS LOADER MOVE PROGRAM
B. THE MDS COLD START LOADER
C. THE MDS BASIC I/O SYSTEM (BIOS)
D. A SKELETAL CBIOS
E. A SKELETAL GETSYS/PUTSYS PROGRAM
F. A SKELETAL COLD START LOADER
CP/M System Alteration Guide

1. INTRODUCTION

The standard CP/M system assumes operation on an Intel MDS microcomputer development system, but is designed so that the user can alter a specific set of subroutines which define the hardware operating environment. In this way, the user can produce a diskette which operates with a non-standard (but IBM-compatible format) drive controller and/or peripheral devices.

In order to achieve device independence, CP/M is separated into three distinct modules:

- BIOS - Basic I/O System which is environment dependent
- BDOS - Basic Disk Operating System which is not dependent upon the hardware configuration
- CCP - the Console Command Processor which uses the BDOS

Of these modules, only the BIOS is dependent upon the particular hardware. That is, the user can "patch" the distribution version of CP/M to provide a new BIOS which provides a customized interface between the remaining CP/M modules and the user's own hardware system. The purpose of this document is to provide a step-by-step procedure for patching the new BIOS into CP/M.

The new BIOS requires some relatively simple software development and testing; the current BIOS, however, is listed in Appendix C, and can be used as a model for the customized package. A skeletal version of the BIOS is given in Appendix D which can form the base for a modified BIOS. In addition to the BIOS, the user must write a simple memory loader, called GETSYS, which brings the operating system into memory. In order to patch the new BIOS into CP/M, the user must write the reverse of GETSYS, called PUTSYS, which places an altered version of CP/M back onto the diskette. PUTSYS is usually derived from GETSYS by changing the disk read commands into disk write commands. Sample skeletal GETSYS and PUTSYS programs are described in Section 3, and listed in Appendix E. In order to make the CP/M system work automatically, the user must also supply a cold start loader, similar to the one provided with CP/M (listed in Appendices A and B). A skeletal form of a cold start loader is given in Appendix F which can serve as a model for your loader.
2. FIRST LEVEL SYSTEM REGENERATION

The procedure to follow to patch the CP/M system is given below in several steps. Address references in each step are followed by an "H" to denote the hexadecimal radix, and are given for a 16K CP/M system. For larger CP/M systems, add a "bias" to each address which is shown with a "+b" following it, where b is equal to the memory size minus 16K. Values for b in various standard memory sizes are

\[
\begin{align*}
24K: & \quad b = 24K - 16K = 8K = 02000H \\
32K: & \quad b = 32K - 16K = 16K = 04000H \\
48K: & \quad b = 48K - 16K = 32K = 08000H \\
56K: & \quad b = 56K - 16K = 40K = 0A000H \\
62K: & \quad b = 62K - 16K = 46K = 0B800H \\
64K: & \quad b = 64K - 16K = 48K = 0C000H
\end{align*}
\]

Note: The standard distribution version of CP/M is configured as a 16K system. Therefore, you must first bring up the 16K CP/M system, and then configure it for your actual memory size (see Second Level System Generation).

1. Review Section 4 and write a GETSYS program which reads the first two tracks of a diskette into memory. The data from the diskette must begin at location 2880H. Code GETSYS so that it starts at location 100H (base of the TPA), as shown in the first part of Appendix E.

2. Test the GETSYS program by reading a blank diskette into memory, and check to see that the data has been read properly, and that the diskette has not been altered in any way by the GETSYS program.

3. Run the GETSYS program using an initialized CP/M diskette to see if GETSYS loads CP/M starting at 2880H (the operating system actually starts 128 bytes later at 2900H).

4. Review Section 4 and write the PUTSYS program which writes memory starting at 2880H back onto the first two tracks of the diskette. The PUTSYS program should be located at 200H, as shown in the second part of Appendix E.

5. Test the PUTSYS program using a blank uninitialized diskette by writing a portion of memory to the first two tracks; clear memory and read it back using GETSYS. Test PUTSYS completely, since this program will be used to alter CP/M on disk.

6. Study Sections 5, 6, and 7, along with the distribution version of the BIOS given in Appendix C, and write a simple version which performs a similar function for the customized environment. Use the program given in Appendix D as a model. Call this new BIOS by the name CB IOS (customized BIOS). Implement only the primitive disk operations on a single drive, and
simple console input/output functions in this phase.

(7) Test BIOS completely to ensure that it properly performs console
coloration I/O and disk reads and writes. Be especially careful to ensure that
no disk write operations occur accidentally during read operations, and check
that the proper track and sectors are addressed on all reads and writes.
Failure to make these checks may cause destruction of the initialized CP/M
system after it is patched.

(8) Referring to figure 1 in Section 5, note that the BIOS is located
between locations 3E00H and 3FFFH. Read the CP/M system using GETSYS, and
replace the BIOS segment by the new BIOS developed in step (6) and tested in
step (7). This replacement is done in the memory of the machine and will be
placed on the diskette in the next step.

(9) Use PUTSYS to place the patched memory image of CP/M onto the first
two tracks of a blank diskette for testing.

(10) Use GETSYS to bring the copied memory image from the test diskette
back into memory at 2880H, and check to ensure that it has loaded back
properly (clear memory, if possible, before the load). Upon successful load,
branch to the cold start code at location 3E00H. The cold start routine will
initialize page zero, then jump to the CCP (location 2900H) which will call
the BDOS, which will call the BIOS. The BIOS will be asked to read several
sectors on track 2 twice in succession, and, if successful, CP/M will type
"A>".

When you make it this far, you are almost on the air. If you have trouble,
use whatever debug facilities you have available to trace and breakpoint your
BIOS.

(11) Upon completion of step (10), CP/M has prompted the console for a
command input. Test the disk write operation by typing

    SAVE 1 X,COM

(recall that all commands must be followed by a carriage return). CP/M should
respond with another prompt (after several disk accesses):

    A>

If it does not, debug your disk write functions and try again.

(12) Test the directory command by typing

    DIR

CP/M should respond with

    A: X       COM
(13) Test the erase command by typing

    ERA X.COM

CP/M should respond with the A prompt. When you make it this far, you should
have an operational system which will only require a bootstrap loader to
function completely.

(14) Write a bootstrap loader which is similar to GETSYS, and place it
    on track 0, sector 1 using PUTSYS (again using the test diskette, not the
distribution diskette). See Sections 5 and 8 for more information on the
bootstrap operation.

(15) Retest the new test diskette with the bootstrap loader installed by
    executing steps (11), (12), and (13). Upon completion of these tests, type a
control-C (control and C keys simultaneously). The system should then execute
a "warm start" which reboots the system and types the A> prompt.

(16) At this point, you probably have a good version of your customized
CP/M system on your test diskette. Use GETSYS to load CP/M from your test
diskette. Remove the test diskette, place the distribution diskette (or a
legal copy) into the drive, and use PUTSYS to replace the distribution version
by your customized version. Do not make this replacement if you are unsure of
your patch since this step destroys the system which was sent to you from
Digital Research.

(17) Load your modified CP/M system, and test it by typing

    DIR

CP/M should respond with a list of files which are provided on the initialized
diskette. One such file should be the memory image for the debugger, called
DDT.COM.

    NOTE: from now on, it is important that you always reboot
the CP/M system if a diskette is removed and replaced
by another diskette, unless the new diskette is to be
read only.

(18) Load and test the debugger by typing

    DDT

(see the document "CP/M Dynamic Debugging Tool (DDT)" for operating
information and examples). Take time to familiarize yourself with DDT; it
will be your best friend in later steps.

(19) Before making further CBIOs modifications, practice using the editor
(see the ED user's guide), and assembler (see the ASM user's guide). Then
recode and test the GETSYS, PUTSYS, and CB IOS programs using
ED, ASM, and DDT. Code and test a COPY program which does a
sector-to-sector copy from one diskette to another to obtain
back-up copies of the original diskette (NOTE: read your CP/M
Licensing Agreement; it specifies your legal responsibilities
when copying the CP/M system). Place the copyright notice

Copyright (c) 1979
Exidy Inc.

on each copy which is made with your COPY program.

(20) Modify your CB IOS to include the extra functions for
punches, readers, signon messages, and so-forth, and add the
facilities for additional drives, if they exist on your system.
You can make these changes with the GETSYS and PUTSYS programs
which you have developed, or you can refer to the following
section, which outlines CP/M facilities which will aid you in
the regeneration process.

You now have a good copy of the customized CP/M system.
Note that although the CB IOS portion of CP/M which you have
developed belongs to you, the modified version of CP/M which
you have created can be copied for your use only (again, read
your Licensing Agreement) and cannot be legally copied for
anyone else's use.

It should be noted that your system remains file-
compatible with all other CP/M systems, which allows transfer
of non-proprietary software between users of CP/M.
3. SECOND LEVEL SYSTEM GENERATION

Now that you have the CP/M system running, you will want to configure CP/M for your memory size. In general, you will first get a memory image of CP/M with the "MOVCPM" program (system relocator) and place this memory image onto a named disk file. The disk file can then be loaded, examined, patched, and replaced using the editor, assembler, debugger, and system generation program. For further details on the operation of these programs, see the "Guide to CP/M Features and Facilities" manual.

To get the memory image of CP/M into the TPA configured for the desired memory size, give the command:

```
MOVCPM xx *
```

where "xx" is the memory size in decimal K bytes (e.g., 32 for 32K). The response will be:

```
CONSTRUCTING xxK CP/M VERS 1.4
READY FOR "SYSGEN" OR
"SAVE 32 CPMxx.COM"
```

At this point, the image of CP/M in the TPA is configured for the desired memory size. The memory image is at location 0000H through 207FH (i.e., the BOOT is at 0000H, the CCP is at 9800H, and the BIOS is at 1E80H). Note that the memory image has the standard MDS-800 BIOS and BOOT on it. It is now necessary to save the memory image in a file so that you can patch your CBIOS and CBOOT into it:

```
SAVE 32 CPMxx.COM
```

Save 20H = 32 pages of memory

The memory image created by the "MOVCPM" program is offset by a negative bias so that it loads into the free area of the TPA, and thus does not interfere with the operation of CP/M in higher memory. This memory image can be subsequently loaded under DDT and examined or changed in preparation for a new generation of the system. DDT is loaded with the memory image by typing:

```
DDT CPMxx.COM
```

Load DDT, then read the CPM image

DDT should respond with

```
NEXT PC
2100 0100
```

You can then use the display (D) and disassembly (L) commands to examine portions of the memory image between 9000H and 207FH. Note, however, that to find any particular address within the memory image, you must apply the negative bias to the CP/M address to find the actual address. Track 00, sector 01 is loaded to location 9000H (you should find the cold start loader at
900H to 97FH, track 00, sector 02 is loaded into 980H (this is the base of the CCP), and so-forth through the entire CP/M system load. In a 16K system, for example, the CCP resides at the CP/M address 2900H, but is placed into memory at 980H by the SYSGEN program. Thus, the negative bias, denoted by n, satisfies

\[ 2900H + n = 980H, \text{ or } n = 980H - 2900H \]

Assuming two's complement arithmetic, \( n = 0E080H \), which can be checked by

\[ 2900H + 0E080H = 10980H = 0980H \] (ignoring high-order overflow).

Note that for larger systems, \( n \) satisfies

\[
(2900H+b) + n = 980H, \text{ or} \\
\text{n = 980H - (2900H + b), or} \\
\text{n = 0E080H - b.}
\]

The value of \( n \) for common CP/M systems is given below

<table>
<thead>
<tr>
<th>Memory size</th>
<th>Bias b</th>
<th>Negative offset n</th>
</tr>
</thead>
<tbody>
<tr>
<td>16K</td>
<td>0000H</td>
<td>0E080H - 0000H = 0E080H</td>
</tr>
<tr>
<td>24K</td>
<td>2000H</td>
<td>0E080H - 2000H = 0C080H</td>
</tr>
<tr>
<td>32K</td>
<td>4000H</td>
<td>0E080H - 4000H = 0A080H</td>
</tr>
<tr>
<td>40K</td>
<td>6000H</td>
<td>0E080H - 6000H = 08080H</td>
</tr>
<tr>
<td>48K</td>
<td>8000H</td>
<td>0E080H - 8000H = 06080H</td>
</tr>
<tr>
<td>56K</td>
<td>0A000H</td>
<td>0E080H - 0A000H = 4080H</td>
</tr>
<tr>
<td>62K</td>
<td>0B800H</td>
<td>0E080H - 0B800H = 2880H</td>
</tr>
<tr>
<td>64K</td>
<td>0C000H</td>
<td>0E080H - 0C000H = 2080H</td>
</tr>
</tbody>
</table>

Assume, for example, that you want to locate the address \( x \) within the memory image loaded under DDT in a 16K system. First type

\[ Hx,n \]

Hexadecimal sum and difference

and DDT will respond with the value of \( x+n \) (sum) and \( x-n \) (difference). The first number printed by DDT will be the actual memory address in the image where the data or code will be found. The input

\[ H2900,E080 \]

for example, will produce 980H as the sum, which is where the CCP is located in the memory image under DDT.

Use the L command to disassemble portions of your CBIOS located at \( (3E00H+b)+n \) which, when you use the H command, produces an actual address of 1E80H. The disassembly command would thus be

\[ L1E80 \]
Terminate DDT by typing a control-C or "G0" in order to prepare the patch program. Your CBIOS and BOOT can be modified using the editor and assembled using ASM, producing files called CBIOS.HEX and BOOT.HEX which contain the machine code for CBIOS and BOOT in Intel hex format. In order to integrate your new modules, return to DDT by typing

DDT CP100.COM

Start DDT and load the CP100 image

It is now necessary to patch in your CBOOT and CBIOS routines. The BOOT resides at location 0000H in the memory image. If the actual load address is X, then to calculate the bias (m) use the command:

H900,X

Subtract load address from target address.

The second number typed in response to the command is the desired bias (m). For example, if your BOOT executes at 0000H, the command:

H900,80

will reply

0900 0800

Sum and difference in hex.

Therefore, the bias "m" would be 0800H. To read the BOOT in, give the command:

ICBOOT.HEX

Input file CBOOT.HEX

Then:

Rm

Read CBOOT with a bias of m (=900H-x)

You may now examine your CBOOT with:

L900

We are now ready to replace the CBIOS. Examine the area at 1E80H where the previous version of the CBIOS resides. Then type

ICBIOS.HEX

Ready the hex file for loading

Assume that your CBIOS is being integrated into a 16K CP/M system, and thus is based at location 3E00H. In order to properly locate the CBIOS in the memory image under DDT, we must apply the negative bias n for a 16K system when loading the hex file. This is accomplished by typing

RE080

Read the file with bias 0E080H
Upon completion of the read, re-examine the area where the CB IOS has been loaded (use an LR80 command), to ensure that it was loaded properly. When you are satisfied that the patch has been made, return from DDT using a control-C or GO command.

Now use SYSGEN to place the patched memory image back onto a diskette (use a test diskette until you are sure of your patch), as shown in the following interaction:

SYSGEN
SYSGEN VERSION 1.4
SOURCE DRIVE NAME (OR RETURN TO SKIP)
Start the SYSGEN program
Sign-on message from SYSGEN
Respond with a carriage return to skip the CP/M read operation since the system is already in memory.

DESTINATION DRIVE NAME (OR RETURN TO REBOOT)
Respond with B to write the new system to the diskette in drive B.

DESTINATION ON B, THEN TYPE RETURN
Hit the return key to perform the actual write.

FUNCTION COMPLETE
DESTINATION DRIVE NAME (OR RETURN TO REBOOT)
Respond with a carriage return to reboot.

Place the test diskette on drive B (if you are operating with a single-drive system, answer "A" rather than "B" to the DESTINATION request; then remove your diskette, and replace it with the test diskette), and type a return. The system will be replaced on the test diskette. Test the new CP/M system by placing the test diskette in drive A and cold-starting.

Write the Exidy Inc. copyright notice on the diskette, as specified in your Licensing Agreement:

Copyright (c), 1979
Exidy Inc.
4. **SAMPLE GETSYS AND PUTSYS PROGRAMS**

The following program provides a framework for the GETSYS and PUTSYS programs referenced in Section 2. The READSEC and WRITSEC subroutines must be inserted by the user to read and write the specific sectors.

```
; GETSYS PROGRAM - READ TRACKS 0 AND 1 TO MEMORY AT 2880H
; REGISTER USE
; A (SCRATCH REGISTER)
; B TRACK COUNT (0, l)
; C SECTOR COUNT (1, 2, ..., 26)
; DE (SCRATCH REGISTER PAIR)
; HL LOAD ADDRESS
; SP SET TO STACK ADDRESS
;
START: LXI SP, 2880H ; SET STACK POINTER TO SCRATCH AREA
    LXI H, 2880H ; SET BASE LOAD ADDRESS
    MVI B, 0 ; START WITH TRACK 0
RDTTRK:
    MVI C, 1 ; READ NEXT TRACK (INITIALLY 0)
    CALL READSEC ; READ STARTING WITH SECTOR 1
    MVI C, 1 ; READ NEXT SECTOR
    CAL READSEC ; USER-SUPPLIED SUBROUTINE
    LDI D, 128 ; MOVE LOAD ADDRESS TO NEXT 1/2 PAGE
    ADD D, HL ; HL = HL + 128
    INR C ; SECTOR = SECTOR + 1
    MOV A, C ; CHECK FOR END OF TRACK
    CPI 27 ; CARRY GENERATED IF SECTOR < 27
;
    JC RDSEC ; ARRIVE HERE AT END OF TRACK, MOVE TO NEXT TRACK
    INR B
    MOV A, B ; TEST FOR LAST TRACK
    CPI 2
    JC RDTTRK ; CARRY GENERATED IF TRACK < 2
;
    ARRIVE HERE AT END OF LOAD, HALT FOR NOW
    HLT
;
    USER-SUPPLIED SUBROUTINE TO READ THE DISK
READSEC:
; ENTER WITH TRACK NUMBER IN REGISTER B,
; SECTOR NUMBER IN REGISTER C, AND
; ADDRESS TO FILL IN HL
;
    PUSH B ; SAVE B AND C REGISTERS
    PUSH H ; SAVE HL REGISTERS

; Perform disk read at this point, branch to
;
Perform disk read at this point, branch to
label START if an error occurs
```

10
POP  H ;RECOVER HL
POP  B ;RECOVER B AND C REGISTERS
RET    ;BACK TO MAIN PROGRAM

END    START

Note that this program is assembled with an assumed origin of 0100, and listed in Appendix D for reference purposes. The hexadecimal operation codes which are listed on the left may be useful if the program has to be entered through your machine's front panel switches.

The PUTFSY program can be constructed from GETSYS by changing only a few operations in the GETSYS program given above, as shown in Appendix E. The register pair HL becomes the dump address (next address to write), and operations upon these registers do not change within the program. The READSEC subroutine is replaced by a WRITESEC subroutine which performs the opposite function: data from address HL is written to the track given by register B and the sector given by register C. It is often useful to combine GETSYS and PUTFSY into a single program during the test and development phase, as shown in Appendix E.
5. DISKETTE ORGANIZATION

The sector allocation for the standard distribution version of CP/M is given here for reference purposes. The first sector (see Figure 1) contains an optional software boot section. Disk controllers are often set up to bring track 0, sector 1 into memory at a specific location (often location 0000H). The program in this sector, called LBOOT, has the responsibility of bringing the remaining sectors into memory starting at location 2900H+b. If your controller does not have a built-in sector load, you can ignore the program in track 0, sector 1 and begin the load from track 0 sector 2 to location 2900H+b.

As an example, the Intel MD8-800 hardware cold start loader brings track 0, sector 1 into absolute address 3000H. Thus, the distribution version contains two very small programs in track 0, sector 1:

MBOOT - a storage move program which moves LBOOT into place following the cold start (Appendix A)

LBOOT - the cold start boot loader (Appendix B)

Upon MD8 start-up, the 128 byte segment on track 0, sector 1 is brought into 3000H. The MBOOT program gets control, and moves the LBOOT program from location 301EH down to location 80H in memory, in order to get LBOOT out of the area where CP/M is loaded in a 16K system. Note that the MBOOT program would not be needed if the MD8 loaded directly to 80H. In general, the LBOOT program could be located anywhere outside the CP/M load area, but is most often located in the area between 000H and 0FFH (below the TPA).

After the move, MBOOT transfers to LBOOT at 80H. LBOOT, in turn, loads the remainder of track 0 and the initialized portion of track 1 to memory, starting at 2900H+b. The user should note that MBOOT and LBOOT are of little use in a non-MD8 environment, although it is useful to study them since some of their actions will have to be duplicated in your cold start loader.

Figure 1. Diskette Allocation

<table>
<thead>
<tr>
<th>Track#</th>
<th>Sector#</th>
<th>Page#</th>
<th>Memory Address</th>
<th>CP/M Module name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>00</td>
<td>2900H+b</td>
<td>CCP</td>
</tr>
<tr>
<td>&quot;</td>
<td>02</td>
<td>00</td>
<td>2980H+b</td>
<td>&quot;</td>
</tr>
<tr>
<td>&quot;</td>
<td>03</td>
<td>01</td>
<td>2A00H+b</td>
<td>&quot;</td>
</tr>
<tr>
<td>&quot;</td>
<td>04</td>
<td>02</td>
<td>2A80H+b</td>
<td>&quot;</td>
</tr>
<tr>
<td>&quot;</td>
<td>05</td>
<td>03</td>
<td>2B00H+b</td>
<td>&quot;</td>
</tr>
<tr>
<td>&quot;</td>
<td>06</td>
<td>04</td>
<td>2B80H+b</td>
<td>&quot;</td>
</tr>
<tr>
<td>&quot;</td>
<td>07</td>
<td>05</td>
<td>2C00H+b</td>
<td>&quot;</td>
</tr>
<tr>
<td>&quot;</td>
<td>08</td>
<td>06</td>
<td>2C80H+b</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

12
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 18 08</td>
<td>3100H+b</td>
<td>BDOS</td>
</tr>
<tr>
<td>00 19 09</td>
<td>3180H+b</td>
<td></td>
</tr>
<tr>
<td>00 20 10</td>
<td>3200H+b</td>
<td></td>
</tr>
<tr>
<td>00 21 11</td>
<td>3280H+b</td>
<td></td>
</tr>
<tr>
<td>00 22 12</td>
<td>3300H+b</td>
<td></td>
</tr>
<tr>
<td>00 23 13</td>
<td>3380H+b</td>
<td></td>
</tr>
<tr>
<td>00 24 14</td>
<td>3400H+b</td>
<td></td>
</tr>
<tr>
<td>00 25 15</td>
<td>3480H+b</td>
<td></td>
</tr>
<tr>
<td>00 26 16</td>
<td>3500H+b</td>
<td></td>
</tr>
<tr>
<td>01 01 01</td>
<td>3580H+b</td>
<td></td>
</tr>
<tr>
<td>01 02 02</td>
<td>3600H+b</td>
<td></td>
</tr>
<tr>
<td>01 03 03</td>
<td>3680H+b</td>
<td></td>
</tr>
<tr>
<td>01 04 04</td>
<td>3700H+b</td>
<td></td>
</tr>
<tr>
<td>01 05 05</td>
<td>3780H+b</td>
<td></td>
</tr>
<tr>
<td>01 06 06</td>
<td>3800H+b</td>
<td></td>
</tr>
<tr>
<td>01 07 07</td>
<td>3880H+b</td>
<td></td>
</tr>
<tr>
<td>01 08 08</td>
<td>3900H+b</td>
<td></td>
</tr>
<tr>
<td>01 09 09</td>
<td>3980H+b</td>
<td></td>
</tr>
<tr>
<td>01 10 10</td>
<td>3A00H+b</td>
<td></td>
</tr>
<tr>
<td>01 11 11</td>
<td>3A80H+b</td>
<td></td>
</tr>
<tr>
<td>01 12 12</td>
<td>3B00H+b</td>
<td></td>
</tr>
<tr>
<td>01 13 13</td>
<td>3B80H+b</td>
<td></td>
</tr>
<tr>
<td>01 14 14</td>
<td>3C00H+b</td>
<td></td>
</tr>
<tr>
<td>01 15 15</td>
<td>3C80H+b</td>
<td></td>
</tr>
<tr>
<td>01 16 16</td>
<td>3D00H+b</td>
<td></td>
</tr>
<tr>
<td>01 17 17</td>
<td>3D80H+b</td>
<td>BDOS</td>
</tr>
</tbody>
</table>

**Notes:**

- **Address ranges:**
  - 01 18-21: 3E00H+b - BIOS
  - 01 19-22: 3E80H+b - |
  - 01 20-21: 3F00H+b - |
  - 01 22-26: (not currently used)
  - 02-76 01-26: (directory and data)

13
6. THE BIOS ENTRY POINTS

The entry points into the BIOS from the cold start loader and BDOS are detailed below. Entry to the BIOS is through a "jump vector" between locations 3E00H+8 and 3E2CH+8, as shown below (see also Appendices, pages C-2 and D-1). The jump vector is a sequence of 15 jump instructions which send program control to the individual BIOS subroutines. The BIOS subroutines may be empty for certain functions (i.e., they may contain a single RET operation) during regeneration of CP/M, but the entries must be present in the jump vector.

It should be noted that there is a 16 byte area reserved in page zero (see Section 9) starting at location 40H, which is available as a "scratch" area in case the BIOS is implemented in ROM by the user. This scratch area is never accessed by any other CP/M subsystem during operation.

The jump vector at 3E00H+8 takes the form shown below, where the individual jump addresses are given to the left:

\[
\begin{align*}
3E00H+8 & \quad \text{JMP BOOT} \quad ;\text{ARRIVE HERE FROM COLD START LOAD} \\
3E03H+8 & \quad \text{JMP WBOOT} \quad ;\text{ARRIVE HERE FOR WARM START} \\
3E06H+8 & \quad \text{JMP CONST} \quad ;\text{CHECK FOR CONSOLE CHAR READY} \\
3E09H+8 & \quad \text{JMP CONIN} \quad ;\text{READ CONSOLE CHARACTER IN} \\
3E0CH+8 & \quad \text{JMP CONOUT} \quad ;\text{WRITE CONSOLE CHARACTER OUT} \\
3E0FH+8 & \quad \text{JMP LIST} \quad ;\text{WRITE LISTING CHARACTER OUT} \\
3E12H+8 & \quad \text{JMP PUNCH} \quad ;\text{WRITE CHARACTER TO PUNCH DEVICE} \\
3E15H+8 & \quad \text{JMP READER} \quad ;\text{READ READER DEVICE} \\
3E18H+8 & \quad \text{JMP HOME} \quad ;\text{MOVE TO TRACK 00 ON SELECTED DISK} \\
3E1BH+8 & \quad \text{JMP SELDISK} \quad ;\text{SELECT DISK DRIVE} \\
3E1EH+8 & \quad \text{JMP SETTRK} \quad ;\text{SET TRACK NUMBER} \\
3E21H+8 & \quad \text{JMP SETSEC} \quad ;\text{SET SECTOR NUMBER} \\
3E24H+8 & \quad \text{JMP SETDMA} \quad ;\text{SET DMA ADDRESS} \\
3E27H+8 & \quad \text{JMP READ} \quad ;\text{READ SELECTED SECTOR} \\
3E2AH+8 & \quad \text{JMP WRITE} \quad ;\text{WRITE SELECTED SECTOR}
\end{align*}
\]

Each jump address corresponds to a particular subroutine which performs the specific function, as outlined below. There are three major divisions in the jump table: (1) the system (re)initiation which results from calls on BOOT and WBOOT, (2) simple character I/O performed by calls on CONST, CONIN, CONOUT, LIST, PUNCH, and READER, and (3) diskette I/O performed by calls on HOME, SELDISK, SETTRK, SETSEC, SETDMA, READ, and WRITE.

All simple character I/O operations are assumed to be performed in ASCII, upper and lower case, with high order (parity bit) set to zero. An end-of-file condition is given by an ASCII control-z (IAH). Peripheral devices are seen by CP/M as "logical" devices, and are assigned to physical devices within the BIOS. In order to operate, the BDOS needs only the CONST, CONIN, and CONOUT subroutines (LIST, PUNCH, and READER are used by PIP, but not by the BDOS). Thus, the initial version of CBIOS may have empty
subroutines for the remaining ASCII devices. The characteristics of each device are

**CONSOLE**

The principal interactive console which communicates with the operator, accessed through CONST, CONIN, and CONOUT. Typically, the CONSOLE is a device such as a CRT or Teletype.

**LIST**

The principal listing device, if it exists on your system, which is usually a hard-copy device, such as a printer or Teletype.

**PUNCH**

The principal tape punching device, if it exists, which is normally a high-speed paper tape punch or Teletype.

**READER**

The principal tape reading device, such as a simple optical reader or Teletype.

Note that a single peripheral can be assigned as the LIST, PUNCH, and READER device simultaneously. If no peripheral device is assigned as the LIST, PUNCH, or READER device, the CBIOS created by the user should give an appropriate error message so that the system does not "hang" if the device is accessed by PIP or some other user program. Alternately, the PUNCH and LIST routines can simply return, and the READER routine can return with a LAH (ctl-Z) in reg A to indicate immediate end-of-file.

For added flexibility, the user can optionally implement the "IOBYTE" function which allows reassignment of physical and logical devices. The IOBYTE function creates a mapping of logical to physical devices which can be altered during CP/M processing (see the STAT command). The definition of the IOBYTE function corresponds to the Intel standard as follows: a single location in memory (currently location 0003H) is maintained, called IOBYTE, which defines the logical to physical device mapping which is in effect at a particular time. The mapping is performed by splitting the IOBYTE into four distinct fields of two bits each, called the CONSOLE, READER, PUNCH, and LIST fields, as shown below:

<table>
<thead>
<tr>
<th>IOBYTE AT 0003H</th>
<th>LIST</th>
<th>PUNCH</th>
<th>READER</th>
<th>CONSOLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 6,7</td>
<td>bits 4,5</td>
<td>bits 2,3</td>
<td>bits 0,1</td>
<td></td>
</tr>
</tbody>
</table>

The value in each field can be in the range 0-3, defining the assigned source or destination of each logical device. The values which can be assigned to each field are given below
CONSOLE field (bits 0,1)
0 - console is assigned to the console printer device (TTY:)
1 - console is assigned to the CRT device (CRT:)
2 - batch mode: use the READER as the CONSOLE input, and the LIST device as the CONSOLE output (BAT:)
3 - user-defined console device (UC1:)

READER field (bits 2,3)
0 - READER is the Teletype device (TTY:)
1 - READER is the high-speed reader device (PTR:)
2 - user-defined reader # 1 (URL1:)
3 - user-defined reader # 2 (UR2:)

PUNCH field (bits 4,5)
0 - PUNCH is the Teletype device (TTY:)
1 - PUNCH is the high speed punch device (FTP:)
2 - user-defined punch # 1 (UP1:)
3 - user-defined punch # 2 (UP2:)

LIST field (bits 6,7)
0 - LIST is the Teletype device (TTY:)
1 - LIST is the CRT device (CRT:)
2 - LIST is the line printer device (LPT:)
3 - user-defined list device (UL1:)

Note again that the implementation of the IOBYTE is optional, and affects only the organization of your CBIOS. No CP/M systems use the IOBYTE (although they tolerate the existence of the IOBYTE at location 0003H), except for PIP which allows access to the physical devices, and STAT which allows logical-physical assignments to be made and/or displayed (for more information, see the "CP/M Features and Facilities Guide"). In any case, the IOBYTE implementation should be omitted until your basic CBIOS is fully implemented and tested; then add the IOBYTE to increase your facilities.

Disk I/O is always performed through a sequence of calls on the various disk access subroutines. These set up the disk number to access, the track and sector on a particular disk, and the direct memory access (DMA) address involved in the I/O operation. After all these parameters have been set up, a call is made to the READ or WRITE function to perform the actual I/O operation. Note that there is often a single call to SELDSK to select a disk drive, followed by a number of read or write operations to the selected disk, before selecting another drive for subsequent operations. Similarly, there may be a single call to set the DMA address, followed by several calls which read or write from the selected DMA address, before the DMA address is changed. The track and sector subroutines are always called before the READ or WRITE operations are performed. Note that the READ and WRITE routines should perform several re-tries (10 is a good number) before reporting the error condition to the BDOS. If the error condition is returned to the BDOS, it will report the error to the user. The HOME subroutine may or may not actually perform the track 0 seek, depending upon your controller.
characteristics; the important point is that track 00 has been selected for
the next operation, and is often treated in exactly the same manner as SETTRK
with a parameter of 00.

The exact responsibilities of each entry point subroutine are given below:

**BOOT**
The BOOT entry point gets control from the cold start loader
and is responsible for basic system initialization, includ-
ing sending a signon message (which can be omitted in the
first version). If the IOBYTE function is implemented, it
must be set at this point. The various system parameters
which are set by the WBOOT entry point must be initialized,
and control is transferred to the CCP at 2900H+b for further
processing. Note that reg C must be set to zero to select
drive A.

**WBOOT**
The WBOOT entry point gets control when a warm start occurs.
A warm start is performed whenever a user program branches to
location 0000H, or when the CPU is reset from the front panel.
The CP/M system must be loaded from the first two tracks of
drive A up to, but not including, the BIOS (or CB IOS, if you
have completed your patch). System parameters must be ini-
tialized as shown below:

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1,2</td>
<td>Set to JMP WBOOT for warm starts (0000H: JMP 3E03H+b).</td>
</tr>
<tr>
<td>3</td>
<td>Set initial value of IOBYTE, if implemented in your CB IOS.</td>
</tr>
</tbody>
</table>
| 5,6,7    | Set to JMP BD OS, which is the primary entry point to CP/M for
transient programs (0005H: JMP 3106H+b). |

(See Section 9 for complete details of page zero use.)
Upon completion of the initialization, the WBOOT program
must branch to the CCP at 2900H+b to (re)start the system.
Upon entry to the CCP, register C is set to the drive to
select after system initialization.

**CONST**
Sample the status of the currently assigned console device;
return 0FFH in register A if a character is ready to read
and 00H in register A if no console characters are ready.

**CONIN**
Read the next console character into register A, and set the
high-order (parity bit). If no console character is ready,
wait until a character is typed before returning.

**CONOUT**
Send the character from register C to the console output de-
vice. The character is in ASCII, with high-order (parity) bit
set to zero. You may want to include a time-out on a line
feed or carriage return, if your console device requires some
time interval at the end of the line (such as a TI Silent 700
terminal). You can, if you wish, filter out control char-
acters which cause your console device to react in a strange
way (a control-z causes the Lear Seigler terminal to clear
the screen, for example).

**LIST**

Send the character from register C to the currently assigned
listing device. The character is in ASCII with zero parity.

**PUNCH**

Send the character from register C to the currently assigned
punch device. The character is in ASCII with zero parity.

**READER**

Read the next character from the currently assigned reader de-
vice into register A with zero parity (high-order bit must be
zero), an end-of-file condition is reported by returning an
ASCII control-z (1AH).

**HOME**

Return the disk head of the currently selected disk (initially
disk A) to the track 00 position. If your controller allows
access to the track 0 flag from the drive, step the head until
the track 0 flag is detected. If your controller does not
support this feature, you can translate the HOME call into a
call on SETTRK with a parameter of 0.

**SELDISK**

Select the disk drive given by register C for further opera-
tions, where register C contains 0 for drive A, 1 for drive B,
2 for drive C, and 3 for drive D. (The standard CP/M
distribution version supports a maximum of four drives). If
your system has less than 4 drives, you may wish to give an
error message at the console, and terminate execution. It is
advisable to postpone the actual disk select operation until
an I/O function (seek, read or write) is actually performed,
since disk selects often occur without ultimately perform-
ing any disk I/O, and many controllers will unload the head of the
current disk before selecting the new drive. This would
cause an excessive amount of noise and disk wear.

**SETTRK**

Register C contains the track number for subsequent disk
accesses on the currently selected drive. You can choose to
seek the selected track at this time, or delay the seek until
the next read or write actually occurs. Register C can take
on values in the range 0-76 corresponding to valid track
numbers.

**SETHSEC**

Register C contains the sector number (1 through 26) for sub-
sequent disk accesses on the currently selected drive. You
can choose to send this information to the controller at this
point, or instead delay sector selection until a read or
write operation occurs.
SETDMA

Registers B and C (high-order 8 bits in B, low-order 8 bits in C) contain the DMA (Direct Memory Access) address for subsequent read or write operations. For example, if B = 00H and C = 80H when SETDMA is called, then all subsequent read operations read their data into 80H through 0FFH, and all subsequent write operations get their data from 80H through 0FFH, until the next call to SETDMA occurs. The initial DMA address is assumed to be 80H. Note that the controller need not actually support direct memory access. If, for example, all data is received and sent through I/O ports, the BIOS which you construct will use the 128-byte area starting at the selected DMA address for the memory buffer during the following read or write operations.

READ

Assuming the drive has been selected, the track has been set, the sector has been set, and the DMA address has been specified, the READ subroutine attempts to read one sector based upon these parameters, and returns the following error codes in register A:

0 no errors occurred
1 non-recoverable error condition occurred

Currently, CP/M responds only to a zero or non-zero value as the return code. That is, if the value in register A is 0 then CP/M assumes that the disk operation completed properly. If an error occurs, however, the BIOS should attempt at least 10 re-tries to see if the error is recoverable. When an error is reported the BDOS will print the message "BDOS ERR ON x: BAD SECTOR." The operator then has the option of typing <cr> to ignore the error, or control-C to abort.

WRITE

Write the data from the currently selected DMA address to the currently selected drive, track, and sector. The data should be marked as "non deleted data" to maintain compatibility with other CP/M systems. The error codes given in the READ command are returned in register A, with error recovery attempts as described above.
7. A SAMPLE BIOS

The program shown in Appendix D can serve as a basis for your first BIOS. The simplest functions are assumed in this BIOS, so that you can enter it through the front panel, if absolutely necessary. Note that the user must alter and insert code into the subroutines for CONST, CONIN, CONOUT, READ, WRITE, and WAITIO. Storage is reserved for user-supplied code in these regions. The scratch area reserved in page zero (see Section 9) for the BIOS is used in this program, so that it could be implemented in ROM, if desired.

Once operational, this skeletal version can be enhanced to print the initial sign-on message and perform better error recovery. The subroutines for LIST, PUNCH, and READER can be filled-out, and the I ObYTE function can be implemented.

8. A SAMPLE COLD START LOADER

The program shown in Appendix E can serve as a basis for your cold start loader. The disk read function must be supplied by the user, and the program must be loaded somehow starting at location 0000. Note that space is reserved for your patch so that the total amount of storage required for the cold start loader is 128 bytes. Eventually, you will probably want to get this loader onto the first disk sector (track 0, sector 1) and cause your controller to load it into memory automatically upon system start-up. Alternatively, you may wish to place the cold start loader into ROM and place it above the CP/M system. In this case, it will be necessary to originate the program at a higher address and key-in a jump instruction at system start-up which branches to the loader. Subsequent warm starts will not require this key-in operation, since the entry point 'WBOOT' gets control, thus bringing the system in from disk automatically. Note also that the skeletal cold start loader has minimal error recovery, which may be enhanced on later versions.
9. RESERVED LOCATIONS IN PAGE ZERO

Main memory page zero, locations 00H through 0FFH, contains several segments of code and data which are used during CP/M processing. The code and data areas are given below for reference purposes.

<table>
<thead>
<tr>
<th>Locations from to</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000H - 0002H</td>
<td>Contains a jump instruction to the warm start entry point at location 3E03H+6. This allows a simple programmed restart (JMP 0000H) or manual restart from the front panel.</td>
</tr>
<tr>
<td>0003H - 0003H</td>
<td>Contains the Intel standard I/O BYTE, which is optionally included in the user's CB IOS, as described in Section 6.</td>
</tr>
<tr>
<td>0004H - 0004H</td>
<td>Current default drive number (0=A, 1=B, 2=C, 3=D).</td>
</tr>
<tr>
<td>0005H - 0007H</td>
<td>Contains a jump instruction to the BDOS, and serves two purposes: JMP 0005H provides the primary entry point to the BDOS, as described in the manual &quot;CP/M Interface Guide,&quot; and LHLD 0006H brings the address field of the instruction to the HL register pair. This value is the lowest address in memory used by CP/M (assuming the CCP is being overlayed). Note that the DDT program will change the address field to reflect the reduced memory size in debug mode.</td>
</tr>
<tr>
<td>0008H - 0027H</td>
<td>(interrupt locations 1 through 5 not used)</td>
</tr>
<tr>
<td>0030H - 0037H</td>
<td>(interrupt location 6, not currently used - reserved)</td>
</tr>
<tr>
<td>0038H - 003AH</td>
<td>Contains a jump instruction into the DDT program when running in debug mode for programmed breakpoints, but is not otherwise used by CP/M.</td>
</tr>
<tr>
<td>003BH - 003FH</td>
<td>(not currently used - reserved)</td>
</tr>
<tr>
<td>0040H - 004FH</td>
<td>16 byte area reserved for scratch by CB IOS, but is not used for any purpose in the distribution version of CP/M</td>
</tr>
<tr>
<td>0050H - 005DH</td>
<td>(not currently used - reserved)</td>
</tr>
<tr>
<td>005CH - 007CH</td>
<td>Default File Control Block produced for a transient program by the Console Command Processor.</td>
</tr>
<tr>
<td>007DH - 007FH</td>
<td>(not currently used - reserved)</td>
</tr>
</tbody>
</table>

21
$0000H - 00FFH  Default 128-byte disk buffer (also filled with the command line when a transient is loaded under the CCP).

Note that this information is setup for normal operation under the CP/M system, but can be overwritten by a transient program if the BIOS facilities are not required by the transient. If, for example, a particular program performs only simple I/O and must begin execution at location 0, it can be first loaded into the TPA, using normal CP/M facilities, with a small memory move program which gets control when loaded (the memory move program must get control from location 0100H, which is the assumed beginning of all transient programs). The move program can then proceed to move the entire memory image down to location 0, and pass control to the starting address of the memory load. Note that if the BIOS is overwritten, or if location 0 (containing the warm start entry point) is overwritten, then the programmer must bring the CP/M system back into memory with a cold start sequence.
10. NOTES FOR USERS OF CP/M VERSION 1.3

The only difference in memory layout between CP/M versions 1.3 and 1.4 is the location of the BDOS, which has been moved down one page (3100h+b instead of 3200h+b). Therefore, your present CB IOS must be changed to reflect this. Normally, the only change is found in the initialization of the jump instruction at location 5. This jump should now be JMP 3106H+b instead of JMP 3206H+b. Note that the CCP is one page shorter, offsetting the longer BDOS, so that the system load address (2900H+b) remains the same. CP/M 1.4 also supports four drives, and thus your CB IOS must account for a drive select value in the range 0-3. No other changes to CP/M affect the CB IOS organization.
APPENDIX A: THE MDS LOADER MOVE PROGRAM

; MDS LOADER MOVE PROGRAM, PLACES COLD START BOOT AT BOOTB
;
3000 ORG 3000H ;WE ARE LOADED HERE ON COLD START
0000 = BOOTB EQU 80H ;START OF COLD BOOT PROGRAM
0001 = BOOTL EQU 80H ;LENGTH OF BOOT
D900 = MBIAS EQU 900H-$ ;BIAS TO ADD DURING LOAD
0078 = BASE EQU 078H ;'BASE' USED By DISK CONTROLLER
0079 = RTYPE EQU BASE+1 ;RESULT TYPE
007B = RBYTE EQU BASE+3 ;RESULT TYPE
;
00FF = BSW EQU 0FFH ;BOOT SWITCH
;
; CLEAR DISK STATUS
3000 DB79 IN RTYPE
3002 DB7B IN RBYTE
;
COLDSTART:
3004 DBFF IN BSW
3006 E602 ANI 2H ;SWITCH ON?
3008 C20430 JNZ COLDSTART
;
300B 211E30 LXI H,BOOTV ;VIRTUAL BASE
300E 0680 MVI B,BOOTL ;LENGTH OF BOOT
3010 118000 LXI D,BOOTB ;DESTINATION OF BOOT
3013 7E MOVE: MOV A,M
3014 12 STAX D ;TRANSFERRED ONE BYTE
3015 23 INX H
3016 13 INX D
3017 05 DCR B
3018 C21330 JNZ MOVE
301B C38000 JMP BOOTB ;TO BOOT SYSTEM

; BOOTV: ;BOOT LOADER PLACE HERE AT SYSTEM GENERATION
008E = LBIAS EQU $-80H+MBIAS ;COLD START BOOT BEGINS AT 80H
301E END
APPENDIX B: THE MDS COLD START LOADER

; MDS COLD START LOADER FOR CP/M
; VERSION 1.4 JANUARY, 1978

0100 = BIAS EQU 100H ;BIAS FOR PLOLOCATION
0000 = FALSE EQU 0
FFFF = TRUE EQU NOT FALSE
0000 = TESTING EQU FALSE ;IF TRUE, THEN GO TO MON80 ON ERRORS

0100 = BDOSB EQU BIAS ;BASE OF DOS LOAD
0906 = BDOS EQU 806H+BIAS ;ENTRY TO DOS FOR CALLS
1800 = BDOSE EQU 1700H+BIAS ;END OF DOS LOAD
1600 = BOOT EQU 1500H+BIAS ;COLD START ENTRY POINT
1603 = RBOOT EQU BOOT+3 ;WARM START ENTRY POINT

0000 = ORG 80H ;LOADED DOWN FROM HARDWARE BOOT AT 3000H

1700 = BDOSL EQU BDOS-BDOSB
0002 = NTRKS EQU 2 ;NUMBER OF TRACKS TO READ
^92E = BDOS EQU BDOSL/128 ;NUMBER OF SECTORS IN DOS
.19 = BDOS0 EQU 25 ;NUMBER OF BDOS SECTORS ON TRACK 0
0015 = BDOS1 EQU BDOS-BDOS0 ;NUMBER OF SECTORS ON TRACK 1

F800 = MON80 EQU 0F800H ;INTEL MONITOR BASE
F0F = RMON80 EQU 0FF0FH ;RESTART LOCATION FOR MON80
0078 = BASE EQU 078H ;'BASE' USED BY CONTROLLER
0079 = RTYPE EQU BASE+1 ;RESULT TYPE
007B = RBYTE EQU BASE+3 ;RESULT BYTE
007F = RESET EQU BASE+7 ;RESET CONTROLLER

0078 = DSTAT EQU BASE ;DISK STATUS FORT
0079 = ILLOW EQU BASE+1 ;LOW IOPB ADDRESS
007A = IHIGH EQU BASE+2 ;HIGH IOPB ADDRESS
0003 = RECAL EQU 3H ;RECALIBRATE SELECTED DRIVE
0004 = READF EQU 4H ;DISK READ FUNCTION
0100 = STACK EQU 100H ;USE END OF BOOT FOR STACK

; RSTART:

0080 310001 LXI SP,STACK;IN CASE OF CALL TO MON80
0083 D37F CLEAR THE CONTROLLER

0085 0602 OUT RESET ;LOGIC CLEARED

0087 21B700 MVI B,NTRKS ;NUMBER OF TRACKS TO READ
0087 21B700 LXI H,IOPB0

; START:
READ FIRST/NEXT TRACK INTO BDOSB

MOV A.L
OUT ILOW
MOV A,H
OUT IHIGH

WAIT0: IN DSTAT
ANI 4
JZ WAIT0

CHECK DISK STATUS

IN RTYPE
ANI 11B
CPI 2

IF TESTING
CNC RMON80 ;GO TO MONITOR IF 11 OR 10
ENDIF

IF NOT TESTING
JNC RSTART ;RETRY THE LOAD
ENDIF

IN RBYTE ;I/O COMPLETE, CHECK STATUS

IF NOT READY, THEN GO TO MON80

RAL
CC RMON80 ;NOT READY BIT SET
RAR ;RESTORE
ANI 11110B ;OVERR/R ADDR ERR/SEEK/CRC/XXX

IF TESTING
QNZ RMON80 ;GO TO MONITOR
ENDIF

IF NOT TESTING
JNZ RSTART ;RETRY THE LOAD
ENDIF

LXI D,IOPBL ;LENGTH OF IOPB
DAD D ;ADDRESSING NEXT IOPB
DCR B ;COUNT DOWN TRACKS
JNZ START

JMP TO BOOT TO PRINT INITIAL MESSAGE, AND SET UP JMPS
JMP BOOT

IOPB0: DB 80H ;IOW, NO UPDATE
DB READF ;READ FUNCTION
DB BDOS0 ;# SECTORS TO READ ON TRACK 0
00BA 00    DB  0    ;TRACK 0
00BB 02    DB  2    ;START WITH SECTOR 2 ON TRACK 0
00BC 0001  DW  BDOSB  ;START AT BASE OF BDOS
0007 =     IOPBL  EQU $-IOPB0
            ;
00BE 80    IOPB1: DB  80H
00BF 04    DB  READF
00C0 15    DB  BDOS1  ;SECTORS TO READ ON TRACK 1
00C1 01    DB  1    ;TRACK 1
00C2 01    DB  1    ;SECTOR 1
00C3 800D  DW  BDOSB+BDOS0*128  ;BASE OF SECOND READ
00C5        END
APPENDIX C: THE MDS BASIC I/O SYSTEM (BIOS)

; MDS I/O DRIVERS FOR CP/M
; (FOUR DRIVE SINGLE DENSITY VERSION)
; VERSION 1.4 JANUARY, 1978

000E =
VERS EQU 14 ;VERSION 1.4

; COPYRIGHT (C) 1978
; DIGITAL RESEARCH
; BOX 579, PACIFIC GROVE
; CALIFORNIA, 93950

FFFF =
TRUE EQU 0FFFFH ;VALUE OF "TRUE"
0000 =
FALSE EQU NOT TRUE ;"FALSE"
FFFF =
SAMPLE EQU TRUE ;TRUE IF SAMPLE BIOS

; IF SAMPLE
2900 =
BIAS EQU 2900H ;SAMPLE PROGRAM IN 16K SYSTEM
ENDIF
IF NOT SAMPLE
BIAS EQU 0000H ;GENERATE RELOCATABLE CP/M SYSTEM
ENDIF

3E00 =
PATCH EQU 1500H+BIAS

; ORG PATCH
2900 =
CPMB EQU 0000H+BIAS ;BASE OF CPM CONSOLE PROCESSOR
3106 =
BDOS EQU 806H+BIAS ;BASIC DOS (RESIDENT FORTION)
1500 =
CPML EQU $-CPMB ;LENGTH (IN BYTES) OF CPM SYSTEM
002A =
NSECTS EQU CPML/128 ;NUMBER OF SECTORS TO LOAD
0002 =
OFFSET EQU 2 ;NUMBER OF DISK TRACKS USED BY CP/M
0004 =
CDISK EQU 0004H ;ADDRESS OF LAST LOGGED DISK ON WARM START
0080 =
BUFF EQU 0080H ;DEFAULT BUFFER ADDRESS
000A =
RETRY EQU 10 ;MAX RETRIES ON DISK I/O BEFORE ERROR

; PERFORM FOLLOWING FUNCTIONS
; BOOT COLD START
; WBOOT WARM START (SAVE I/O BYTE)
; (BOOT AND WBOOT ARE THE SAME FOR MDS)
; CONST CONSOLE STATUS
; REG-A = 00 IF NO CHARACTER READY
; REG-A = FF IF CHARACTER READY
; CONIN CONSOLE CHARACTER IN (RESULT IN REG-A)
; CONOUT CONSOLE CHARACTER OUT (CHAR IN REG-C)
; LIST LIST OUT (CHAR IN REG-C)
; PUNCH PUNCH OUT (CHAR IN REG-C)
; READER PAPER TAPE READER IN (RESULT TO REG-A)
; HOME MOVE TO TRACK 00

C-1
(THE FOLLOWING CALLS SET-UP THE IO PARAMETER BLOCK FOR THE
MDS, WHICH IS USED TO PERFORM SUBSEQUENT READS AND WRITES)
SELDISK SELECT DISK GIVEN BY REG-C (0,1,2,...)
SETTRK SET TRACK ADDRESS (0,...,76) FOR SUBSEQUENT READ/WRITE
SETSEC SET SECTOR ADDRESS (1,...,26) FOR SUBSEQUENT READ/WRITE
SETDMA SET SUBSEQUENT DMA ADDRESS (INITIALLY 80H)

(READ AND WRITE ASSUME PREVIOUS CALLS TO SET UP THE IO PARAMETERS)
READ READ TRACK/SECTOR & PRESET DMA ADDRESS
WRITE WRITE TRACK/SECTOR FROM PRESET DMA ADDRESS

JUMP VECTOR FOR INDIVIDUAL Routines

3E00 C343E JMP BOOT
3E03 C354E JMP WB0TE:
3E06 C3F23E JMP CONST
3E09 C3F53E JMP CONIN
3E0C C3FB3E JMP CONOUT
3E0F C3FE3E JMP LIST
3E12 C3013F JMP PUNCH
3E15 C3043F JMP READER
3E18 C3073F JMP HOME
3E1B C30C3F JMP SELD1SK
3E1E C32A3F JMP SETTRK
3E21 C32F3F JMP SETSEC
3E24 C3343F JMP SETDMA
3E27 C33A3F JMP READ
3E2A C3433F JMP WRITE

END OF CONTROLLER - INDEPENDENT CODE, THE REMAINING SUBROUTINES
ARE TAILORED TO THE PARTICULAR OPERATING ENVIRONMENT, AND MUST
BE ALTERED FOR ANY SYSTEM WHICH DIFFERS FROM THE INTEL MDS.

THE FOLLOWING CODE ASSUMES THE MDS MONITOR EXISTS AT 0F800H
AND USES THE I/O SUBROUTINES WITHIN THE MONITOR

0004 = NDISKS EQU 4 ;NUMBER OF DRIVES AVAILABLE
00FD = REVRT EQU 0FDH ;INTERRUPT REVERT PORT
00FC = INTC EQU 0FCH ;INTERRUPT MASK PORT
00F3 = ICON EQU 0F3H ;INTERRUPT CONTROL PORT
007E = INTE EQU 0111$1110B ;ENABLE RST 0(WARM BOOT), RST 7 (MONITOR)

MDS MONITOR EQUATES

P800 = MON80 EQU 0F800H ;MDS MONITOR
PF80F = RMON80 EQU 0FF0FH ;RESTART MON80 (BOOT ERROR)
P803 = CI EQU 0F803H ;CONSOLE CHARACTER TO REG-A
P806 = RI EQU 0F806H ;READER IN TO REG-A
P809 = CO EQU 0F809H ;CONSOLE CHAR FROM C TO CONSOLE OUT

C-2
F80C = PO EQU 0F80CH ;PUNCH CHAR FROM C TO PUNCH DEVICE
F80F = LO EQU 0F80FH ;LIST FROM C TO LIST DEVICE
F812 = CSTS EQU 0F812H ;CONSOLE STATUS 00/FF TO REGISTER A

; DISK FORTS AND COMMANDS
0078 = BASE EQU 78H ;BASE OF DISK COMMAND IO FORTS
0078 = DSTAT EQU BASE ;DISK STATUS (INPUT)
0079 = RTYPE EQU BASE+1 ;RESULT TYPE (INPUT)
007B = RBYTE EQU BASE+3 ;RESULT BYTE (INPUT)

0079 = ILOW EQU BASE+1 ;IOPB LOW ADDRESS (OUTPUT)
007A = IHIGH EQU BASE+2 ;IOPB HIGH ADDRESS (OUTPUT)

0004 = READF EQU 4H ;READ FUNCTION
0006 = WRITF EQU 6H ;WRITE FUNCTION
0003 = RECAL EQU 3H ;RECALIBRATE DRIVE
0004 = IORDY EQU 4H ;I/O FINISHED MASK
000D = CR EQU 0DH ;CARRIAGE RETURN
000A = LF EQU 0AH ;LINE FEED

;SIGNON: ;SIGNON MESSAGE: XXK CP/M VERS Y.Y
3E2D 0D0A0A DB CR,LF,LF
IF SAMPLE
3E30 3136 DB '16' ;16K EXAMPLE BIOS
ENDIF
IF NOT SAMPLE
DB '00' ;MEMORY SIZE FILLED BY RELOCATOR
ENDIF
3E32 4B2043502F DB 'K CP/M VERS '.
3E33 312E34 DB VERS/10+'0',',',VERS MOD 10+'0'
3E41 0D0A00 DB CR,LF,0

;BOOT: ;PRINT SIGNON MESSAGE AND GO TO CCP
; (NOTE: MDS BOOT INITIALIZED I/OBYTE AT 0003H)
3E44 310001 LXI SP,BUFF+80H
3E47 212D3E LXI H,SIGNON
3E4A 0D4C3F CALL PRMSG ;PRINT MESSAGE
3E4D AF XRA A ;CLEAR ACCUMULATOR
3E4E 320400 STA CDISK ;SET INITIALLY TO DISK A
3E51 C3A03E JMP GOCMP ;GO TO CP/M

; WBOOT: ; LOADER ON TRACK 0, SECTOR 1, WHICH WILL BE SKIPPED FOR WARM
; READ CP/M FROM DISK - ASSUMING THERE IS A 128 BYTE COLD START
; START.
3E54 310000 LXI SP,BUFF ;USING DMA - THUS 80 THRU FF AVAILABLE FOR STACK

3E57 0E0A MVI C,RETRY ;MAX RETRIES
3E59 C5 PUSH B

C-3
WBOOT0: ;ENTER HERE ON ERROR RETRIES
3E5A 010029 LXI B,CPMB ;SET DMA ADDRESS TO START OF DISK SYSTEM
3E5D CD343F CALL SETDMA
3E60 0000 MVI C,0 ;BOOT FROM DRIVE 0
3E62 CD033F CALL SELSDK
3E65 0000 MVI C,0
3E67 CD2A3F CALL SETTRK ;START WITH TRACK 0
3E6A 0002 MVI C,2 ;START READING SECTOR 2
3E6C CD2F3F CALL SETSEC

;READ SECTORS, COUNT NSECTS TO ZERO
3E6F C1 POP B ;10-ERROR COUNT
3E70 062A MVI B,NSECTS
3E72 C5 PUSH B ;SAVE SECTOR COUNT
3E73 CD3A3F CALL READ
3E76 C2DA3E JNZ BOOTERR ;RETRY IF ERRORS OCCUR
3E79 2AE53F LHLD IOD ;INCREMENT DMA ADDRESS
3E7C 118000 LXI D,128 ;SECTOR SIZE
3E7F 19 DAD D ;INCREMENTED DMA ADDRESS IN HL
3E80 44 MOV B,H
3E81 4D MOV C,L ;READY FOR CALL TO SET DMA
3E82 CD343F CALL SETDMA
3E85 3AE43F LDA IOT ;SECTOR NUMBER JUST READ
3E88 FE1A CPI 26 ;READ LAST SECTOR?
3E8A DA963E JC RD1 ;MUST BE SECTOR 26, ZERO AND GO TO NEXT TRACK
3E8D 3AE33F LDA IOT ;GET TRACK TO REGISTER A
3E90 3C INR A
3E91 4F MOV C,A ;READY FOR CALL
3E92 CD2A3F CALL SETTRK
3E95 AF XRA A ;CLEAR SECTOR NUMBER
3E96 3C RD1: INR A ;TO NEXT SECTOR
3E97 4F MOV C,A ;READY FOR CALL
3E98 CD2F3F CALL SETSEC
3E9B C1 POP B ;RECALL SECTOR COUNT
3E9C 05 DCR B ;DONE?
3E9D C2723E JNZ RDSEC

;DONE WITH THE LOAD, RESET DEFAULT BUFFER ADDRESS
GOCPM: ;(ENTER HERE FROM COLD START BOOT)
;ENABLE RST0 AND RST7
3E9E P3 DI
3EA0 3E12 MVI A,12H ;INITIALIZE COMMAND
3EA3 D3FD OUT REVRT
3EA5 AF XRA A
3EA6 D3FC OUT INTC ;CLEARED
3EA8 3E7E MVI A,INTE ;RST0 AND RST7 BITS ON
3EAA D3FC OUT INTC
3EAC AF XRA A
3EAD D3F3          OUT ICON ;INTERRUPT CONTROL

3EAF 018000        MOV A,80H
3EB2 CD343F         CALL SETDMA

3EB5 3EC3           MOV A,JMP
3EB7 320000         STA 0
3EBA 21033E         LXI H,WBOOT
3EBD 202100         SHLD 1 ;JMP WBOOT AT LOCATION 00
3EC0 320500         STA 5
3EC3 210631         LXI H,BDOS
3EC6 220600         SHLD 6 ;JMP BDOS AT LOCATION 5
3EC9 323000         STA 7*8 ;JMP TO MON80 (MAY HAVE BEEN CHANGED BY DDT)
3ECC 21080F         LXI H,MON80
3ECF 223000         SHLD 7*8+1

; LEAVE I/O BYTE SET
; PREVIOUSLY SELECTED DISK WAS B, SEND PARAMETER TO CPM
3ED2 3A0400         LDA CDISK ;LAST LOGGED DISK NUMBER
3ED5 4F             MOV C,A ;SEND TO CCP TO LOG IT IN
3ED6 FB             EI
3ED7 C30029         JMP CPMB

; ERROR CONDITION OCCURRED, PRINT MESSAGE AND RETRY
BOOTERR:
3EDA C1             POP B ;RECALL COUNTS
3EDB 0D             DCR C
3EDC CAE33E         JZ BOOTERR
3EDF C5             PUSH B
3EE0 C35A3E         JMP WBOOT

BOOTERR:
; OTHERWISE TOO MANY RETRIES
3EE3 21EC3E         LXI H,BOOTMSG
3EE6 CD43CF         CALL PRMSG
3EE9 C30FFF         JMP RMON80 ;MDS HARDWARE MONITOR

BOOTMSG:
3EEC 3F424F4F54 DB '?'BOOT',0

; CONST: ;CONSOLE STATUS TO REG-A
; (EXACTLY THE SAME AS MDS CALL)
3EF2 C312F8         JMP CSTS

; CONIN: ;CONSOLE CHARACTER TO REG-A
3EF5 CD03F8         CALL CI
3EF8 B67F           ANI 7FH ;REMOVE PARITY BIT

C-5
3EFA C9     RET

; CONOUT:  ; CONSOLE CHARACTER FROM C TO CONSOLE OUT
3EFB C309F8 JMP CO

; LIST:    ; LIST DEVICE OUT
; (EXACTLY THE SAME AS MDS CALL)
3EFE C30FF8 JMP LO

; PUNCH:   ; PUNCH DEVICE OUT
; (EXACTLY THE SAME AS MDS CALL)
3F01 C30CF8 JMP PO

; READER:  ; READER CHARACTER IN TO REG-A
; (EXACTLY THE SAME AS MDS CALL)
3F04 C306F8 JMP RI

; HOME:    ; MOVE TO HOME POSITION
; TREAT AS TRACK 00 SEEK
3F07 0E00  MOV C,0
3F09 C32A3F JMP SETTRK

; SELDISK: ; SELECT DISK GIVEN BY REGISTER C
; CP/M HAS CHECKED FOR DISK SELECT 0 - 3, BUT WE MAY HAVE
; A SMALLER MDS SYSTEM, SO CHECK AGAIN AND GIVE ERROR
; BY CALLING MON80
3F0C 79     MOV A,C
3F0D FE04   CPI NDISKS ; TOO LARGE?
3F0F D40FFF  CNC RMON80 ; GIVES #ADDR MESSAGE AT CONSOLE

3F12 E602   ANI 10B ; 00 00 FOR DRIVE 0, 1 AND 10 10 FOR DRIVE 2, 3
3F14 32DF3F  STA DBANK ; TO SELECT DRIVE BANK
3F17 79     MOV A,C ; 00, 01, 10, 11
3F18 E601   ANI 1B ; MDS HAS 0, 1 AT 78, 2, 3 AT 88
3F1A B7     ORA A ; RESULT 00?
3F1B CA203F  JZ SETDRIVE
3F1E 3E30   MOV A,00110000B ; SELECTS DRIVE 1 IN BANK

SETDRIVE:
3F20 4F     MOV C,A ; SAVE THE FUNCTION
3F21 21E13F  LXI H,IOF ; IO FUNCTION
3F24 7E     MOV A,M
3F25 E6CF   ANI 11001111B ; MASK OUT DISK NUMBER
3F27 B1     ORA C ; MASK IN NEW DISK NUMBER
3F28 77     MOV M,A ; SAVE IT IN IOPB
3F29 C9     RET

; SETTRK:  ; SET TRACK ADDRESS GIVEN BY C
3F2A 21E33F  LXI H,IOT
3F2D 71     MOV M,C

C-6
3F2E C9  RET
;
  SETSEC: ;SET SECTOR NUMBER GIVEN BY C
3F2F 79  MOV  A,C   ;SECTOR NUMBER TO ACCUM
3F30 32E4F  STA  IOS   ;STORE SECTOR NUMBER TO IOPB
3F33 C9  RET
;
  SETDMA: ;SET DMA ADDRESS GIVEN BY REGS B,C
3F34 69  MOV  L,C
3F35 60  MOV  H,B
3F36 22E53F  SHLD  IOD
3F39 C9  RET
;
  READ: ;READ NEXT DISK RECORD (ASSUMING DISK/TRK/SEC/DMA SET)
3F3A 0E04  MVI  C,READF ;SET TO READ FUNCTION
3F3C CD593F  CALL  SETFUNC
3F3F CD693F  CALL  WAITIO   ;PERFORM READ FUNCTION
3F42 C9  RET   ;MAY HAVE ERROR SET IN REG-A
;
  WRITE: ;DISK WRITE FUNCTION
3F43 0E06  MVI  C,WRITF
3F45 CD593F  CALL  SETFUNC ;SET TO WRITE FUNCTION
3F48 CD693F  CALL  WAITIO
3F4B C9  RET   ;MAY HAVE ERROR SET
;
  UTILITY SUBROUTINES
PRMSG: ;PRINT MESSAGE AT H,L TO 0
3F4C 7E  MOV  A,M
3F4D B7  ORA  A   ;ZERO?
3F4E C8  RZ
;
  MORE TO PRINT
3F4F E5  PUSH  H
3F50 4F  MOV  C,A
3F51 CDFF3E  CALL  CONOUT
3F54 E1  POP  H
3F55 23  INX  H
3F56 C34C3F  JMP  PRMSG
;
  SETFUNC:
  SET FUNCTION FOR NEXT I/O (COMMAND IN REG-C)
3F59 21E13F  LXI  H,IOF   ;IO FUNCTION ADDRESS
3F5C 7E  MOV  A,M   ;GET IT TO ACCUMULATOR FOR MASKING
3F5D E6F8  ANI  11111000B   ;REMOVE PREVIOUS COMMAND
3F5F B1  ORA  C   ;SET TO NEW COMMAND
3F60 77  MOV  M,A   ;REPLACED IN IOPB
;
  THE MDS-800 CONTROLLER REQUIRES DISK BANK BIT IN SECTOR BYTE
;
  THE MDS-800 CONTROLLER REQUIRES DISK SELECT BIT
3F61 E620  ANI  00100000B   ;MASK THE DISK SELECT BIT

C-7
3F63 21E43F LXI H,IOS ;ADDRESS THE SECTOR SELECT BYTE
3F66 B6 ORA M SELECT PROPER DISK BANK
3F67 77 MOV M,A ;SET DISK SELECT BIT ON/OFF
3F68 C9 RET

; WAITIO:
3F69 0E0A MVI C,RETRY ;MAX RETRIES BEFORE PERM ERROR

; REWAIT:
; START THE I/O FUNCTION AND WAIT FOR COMPLETION
3F6B CDB83F CALL INTYPE ;IN RTYPE
3F6E CDC53F CALL INBYTE ;CLEAR THE CONTROLER

; 3F71 3ADF3F LDA DBANK ;SET BANK FLAGS
3F74 B7 ORA A ;ZERO IF DRIVE 0,1 AND NZ IF 2,3
3F75 3EE0 MVI A,IOPB AND 0FFH ;LOW ADDRESS FOR IOPB
3F77 063F MVI B,IOPB SHR 8 ;HIGH ADDRESS FOR IOPB
3F79 C2843F JNZ IODR1 ;DRIVE BANK 1?
3F7C D379 OUT ILOW ;LOW ADDRESS TO CONTROLLER
3F7E 78 MOV A,B
3F7F D37A OUT IHIGH ;HIGH ADDRESS
3F80 C3893F JMP WAIT0 ;TO WAIT FOR COMPLETE

; IODR1: ;DRIVE BANK 1
3F81 D389 OUT ILOW+10H ;88 FOR DRIVE BANK 10
3F84 78 MOV A,B
3F85 D38A OUT IHIGH+10H

; 3F89 CDD23F WAIT0: CALL INSTAT ;WAIT FOR COMPLETION
3F8C E604 ANI IORDY ;READY?
3F8E CA893F JZ WAIT0

; CHECK IO COMPLETION OK
3F91 CDB83F CALL INTYPE ;MUST BE IO COMPLETE (80) UNLINKED
; 00 UNLINKED I/O COMPLETE, 01 LINKED I/O COMPLETE (NOT USED)
; 10 DISK STATUS CHANGED 11 (NOT USED)
3F94 FE02 CPI 10B ;READY STATUS CHANGE?
3F96 CAAB3F JZ WREADY

; MUST BE 00 IN THE ACCUMULATOR
3F99 B7 ORA A
3F9A C2B13F JNZ WERROR ;SOME OTHER CONDITION, RETRY

; CHECK I/O ERROR BITS
3F9D CDC53F CALL INBYTE
3FA0 17 RAL
3FA1 DAAAB3F JC WREADY ;UNIT NOT READY
3FA4 1F RAR
3FA5 E6FE ANI 11111110B ;ANY OTHER ERRORS? (DELETED DATA OK)
3FA7 C2B13F JNZ WERROR
3FAA C9
; READ OR WRITE IS OK, ACCUMULATOR CONTAINS ZERO
RET
;
WREADY: ;NOT READY, TREAT AS ERROR FOR NOW
3FAB CDC53F
CALL INBYTE ;CLEAR RESULT BYTE
3FAE C3B13F
JMP TRYCOUNT
;
WERROR: ;RETURN HARDWARE MALFUNCTION (CRC, TRACK, SEEK, ETC.)
; THE MDS CONTROLLER HAS RETURNED A BIT IN EACH POSITION
; OF THE ACCUMULATOR, CORRESPONDING TO THE CONDITIONS:
; 0  - DELETED DATA (ACCEPTED AS OK ABOVE)
; 1  - CRC ERROR
; 2  - SEEK ERROR
; 3  - ADDRESS ERROR (HARDWARE MALFUNCTION)
; 4  - DATA OVER/UNDER FLOW (HARDWARE MALFUNCTION)
; 5  - WRITE PROTECT (TREATED AS NOT READY)
; 6  - WRITE ERROR (HARDWARE MALFUNCTION)
; 7  - NOT READY
; (ACCUMULATOR BITS ARE NUMBERED 7 6 5 4 3 2 1 0)
;
; IT MAY BE USEFUL TO FILTER OUT THE VARIOUS CONDITIONS,
; BUT WE WILL GET A PERMANENT ERROR MESSAGE IF IT IS NOT
; RECOVERABLE. IN ANY CASE, THE NOT READY CONDITION IS
; TREATED AS A SEPARATE CONDITION FOR LATER IMPROVEMENT
TRYCOUNT:
; REGISTER C CONTAINS RETRY COUNT, DECREMENT 'TIL ZERO
3FB1 0D
3FB2 C26B3F
DCR C
3FB5 3E01
3FB7 C9
JNZ RWAIT ;FOR ANOTHER TRY
;
; CANNOT RECOVER FROM ERROR
3FB8 3ADF3F
MVI A,1 ;ERROR CODE
3FB7 C9
RET
;
; INTYPE, INBYTE, INSTAT READ DRIVE BANK 00 OR 10
3FB8 3ADF3F
INTYPE: LDA DBANK
3FBB B7
ORA A
3FBC C2C23F
JNZ INTYPE1 ;SKIP TO BANK 10
3FBB DB79
IN RTYPE
3FC1 C9
RET
3FC2 DB89
INTYPE1: IN RTYPE+10H ;78 FOR 0,1 88 FOR 2,3
3FC4 C9
RET
;
3FC5 3ADF3F
INBYTE: LDA DBANK
3FC8 B7
ORA A
3FC9 C2CF3F
JNZ INBYTE1
3FCC DB7B
IN RBYTE
3FCE C9
RET
3FCF DB88
INBYTE1: IN RBYTE+10H
3FD1 C9
RET
3FD2 3ADF3F  INSTAT: LDA DBANK
3FD5 B7    ORA A
3FD6 C2DC3F JNZ INSTAL
3FD9 DB78   IN DSTAT
3FDB C9    RET
3FDC DB88  INSTAL: IN DSTAT+10H
3FDE C9    RET

; ; ;
; DATA AREAS (MUST BE IN RAM)
3FDF 00  DBANK: DB 0 ;DISK BANK 00 IF DRIVE 0,1
            DB 10 IF DRIVE 2,3
            IOPB: ;IO PARAMETER BLOCK
3FE0 80    DB 80H ;NORMAL I/O OPERATION
3FE1 04    IOF: DB READF ;IO FUNCTION, INITIAL READ
3FE2 01    ION: DB 1 ;NUMBER OF SECTORS TO READ
3FE3 02    IOT: DB OFFSET ;TRACK NUMBER
3FE4 01    IOS: DB 1 ;SECTOR NUMBER
3FE5 8000  IOD: DW BUFF ;IO ADDRESS
            ;
3FE7  END
APPENDIX D: A SKELETAL CB IOS

; SKELETAL CB IOS FOR FIRST LEVEL OF CP/M ALTERATION

; NOTE: MSIZE DETERMINES WHERE THIS CB IOS IS LOCATED

0010 = MSIZE EQU 16 ;CP/M VERSION MEMORY SIZE IN KILOBYTES
3E00 = PATCH EQU MSIZE*1024-2*256 ;START OF THE CB IOS PATCH

; WE WILL USE THE AREA RESERVED STARTING AT LOCATION
; 40H IN PAGE 0 FOR HOLDING THE VALUES OF:
; TRACK = LAST SELECTED TRACK
; SECTOR = LAST SELECTED SECTOR
; DMAAD = LAST SELECTED DMA ADDRESS
; DISKNO = LAST SELECTED DISK NUMBER
; (NOTE THAT ALL ARE BYTE VALUES EXCEPT FOR DMAAD)

0040 = SCRAT EQU 40H ;BASE OF SCRATCH AREA (FROM 40H TO 4FH)
0040 = TRACK EQU SCRAT ;CURRENTLY SELECTED TRACK
0041 = SECTOR EQU SCRAT+1 ;CURRENTLY SELECTED SECTOR
0042 = DMAAD EQU SCRAT+2 ;CURRENT DMA ADDRESS
0044 = DISKNO EQU SCRAT+4 ;CURRENT DISK NUMBER

3E00 = ORG PATCH ;ORIGIN OF THIS PROGRAM
0000 = CBASE EQU (MSIZE-16)*1024 ;BIAS FOR SYSTEMS LARGER THAN 16K
2900 = CPMB EQU CBASE+2900H ;BASE OF CP/M (= BASE OF CCP)
3106 = BDOS EQU CBASE+3106H ;BASE OF RESIDENT PORTION OF CP/M
1500 = CPML EQU $-CPMB ;LENGTH OF THE CP/M SYSTEM IN BYTES
002A = NSECTS EQU CPML/128 ;NUMBER OF SECTORS TO LOAD ON WARM START

3E00 C32D3E JMP BOOT ;COLD START

WBOOT:
3E03 C3303E JMP WBOOT ;WARM START
3E06 C3993E JMP CONST ;CONSOLE STATUS
3E09 C3AC3E JMP CONIN ;CONSOLE CHARACTER IN
3E0C C3BF3E JMP CONOUT ;CONSOLE CHARACTER OUT
3E0F C3D13E JMP LIST ;LIST CHARACTER OUT
3E12 C3D33E JMP PUNCH ;PUNCH CHARACTER OUT
3E15 C3D53E JMP READER ;READER CHARACTER OUT
3E18 C3D93E JMP HOME ;MOVE HEAD TO HOME POSITION
3E1B C3E03E JMP SELDISK ;SELECT DISK
3E1E C3F53E JMP SETRTRK ;SET TRACK NUMBER
3E21 C30A3F JMP SETSEC ;SET SECTOR NUMBER
3E24 C31F3F JMP SETDMA ;SET DMA ADDRESS
3E27 C3353F JMP READ ;READ DISK
3E2A C3483F JMP WRITE ;WRITE DISK

D-1
INDIVIDUAL SUBROUTINES TO PERFORM EACH FUNCTION

BOOT: ;SIMPLEST CASE IS TO JUST PERFORM PARAMETER INITIALIZATION
JMP GOCPM ;INITIALIZE AND GO TO CP/M

WBOOT: ;SIMPLEST CASE IS TO READ THE DISK UNTIL ALL SECTORS LOADED
LXI SP, 80h ;USE SPACE BELOW BUFFER FOR STACK
MVI C, 0 ;SELECT DISK 0
CALL SELSK
CALL HOME ;GO TO TRACK 00

MVI B, NSECTS ;B COUNTS THE NUMBER OF SECTORS TO LOAD
MVI C, 0 ;C HAS THE CURRENT TRACK NUMBER
MVI D, 2 ;D HAS THE NEXT SECTOR TO READ
NOTE THAT WE BEGIN BY READING TRACK 0, SECTOR 2 SINCE SECTOR 1
CONTAINS THE COLD START LOADER, WHICH IS SKIPPED IN A WARM START
LXI H, CPMB ;BASE OF CP/M (INITIAL LOAD POINT)

LOAD1: ;LOAD ONE MORE SECTOR
PUSH B ;SAVE SECTOR COUNT, CURRENT TRACK
PUSH D ;SAVE NEXT SECTOR TO READ
PUSH H ;SAVE DMA ADDRESS
MOV C, D ;GET SECTOR ADDRESS TO REGISTER C
CALL SETSEC ;SET SECTOR ADDRESS FROM REGISTER C
POP B ;RECALL DMA ADDRESS TO B, C
POP B ;REPLACE ON STACK FOR LATER RECALL
CALL SETDMA ;SET DMA ADDRESS FROM B, C

DRIVE SET TO 0, TRACK SET, SECTOR SET, DMA ADDRESS SET
CALL READ
CPI 00h ;ANY ERRORS?
JNZ WBOOT ;RETRY THE ENTIRE BOOT IF AN ERROR OCCURS

NO ERROR, MOVE TO NEXT SECTOR
POP H ;RECALL DMA ADDRESS
LXI D, 128 ;DMA=DMA+128
DAD D ;NEW DMA ADDRESS IN H, L
POP D ;RECALL SECTOR ADDRESS
POP B ;RECALL NUMBER OF SECTORS REMAINING, AND CURRENT TRK
DCR B ;SECTORS=SECTORS-1
JZ GOCPM ;TRANSFER TO CP/M IF ALL HAVE BEEN LOADED

MORE SECTORS REMAIN TO LOAD, CHECK FOR TRACK CHANGE

INR D
MOV A, D ;SECTOR=27?, IF SO, CHANGE TRACKS
CPI 27
JC LOAD1 ;CARRY GENERATED IF SECTOR<27

END OF CURRENT TRACK, GO TO NEXT TRACK
MVI D, 1 ;BEGIN WITH FIRST SECTOR OF NEXT TRACK
INC C ;TRACK=TRACK+1
; SAVE REGISTER STATE, AND CHANGE TRACKS
3E6D C5
3E6E D5
3E6F E5
3E70 CDF53E
3E73 E1
3E74 D1
3E75 C1
3E76 C3443E
; CALL SETTRK ;TRACK ADDRESS SET FROM REGISTER C
; POP H
; POP B
; JMP LOAD1 ;FOR ANOTHER SCTOR
; END OF LOAD OPERATION, SET PARAMETERS AND GO TO CP/M
; GOCPM:
3E79 3EC3
3E7B 320000
3E7E 21033E
3E81 220100
; MOV A,\0C3H ;C3 IS A JMP INSTRUCTION
; STA 0 ;FOR JMP TO WBOOT
; LXI H,WBOOTE ;WBOOT ENTRY POINT
; SHLD 1 ;SET ADDRESS FIELD FOR JMP AT 0
3E84 320500
3E87 210631
3E8A 220600
; STA 5 ;FOR JMP TO BDOS
; LXI H,BDOS ;BDOS ENTRY POINT
; SHLD 6 ;ADDRESS FIELD OF JUMP AT 5 TO BDOS
3E8D 018000
3E90 CD1F3F
; CALL SETDMA
3E93 FB
; EI ;ENABLE THE INTERRUPT SYSTEM
; FUTURE VERSIONS OF CCP WILL SELECT THE DISK GIVEN BY REGISTER
; C UPON ENTRY, HENCE ZERO IT IN THIS VERSION OF THE BIOS FOR
; FUTURE COMPATIBILITY.
3E94 0E00
3E96 C30029
; MVI C,0 ;SELECT DISK ZERO AFTER INITIALIZATION
; JMP CPMB ;GO TO CP/M FOR FURTHER PROCESSING
; SIMPLE I/O HANDLERS (MUST BE FILLED IN BY USER)
; IN EACH CASE, THE ENTRY POINT IS PROVIDED, WITH SPACE RESERVED
; TO INSERT YOUR OWN CODE
; CONST: ;CONSOLE STATUS, RETURN 0FFH IF CHARACTER READY, 00H IF NOT
3E99
3EA9 3E00
3EAB C9
; DS 10H ;SPACE FOR STATUS SUBROUTINE
; MVI A,00H
; RET
; CONIN: ;CONSOLE CHARACTER INTO REGISTER A
3EAC
3EBC E67F
3EBE C9
; DS 10H ;SPACE FOR INPUT ROUTINE
; ANI 7FH ;STRIP PARITY BIT
; RET
; CONOUT: ;CONSOLE CHARACTER OUTPUT FROM REGISTER C
3EBF 79
3EC0
3ED0 C9
; MOV A,C ;GET TO ACCUMULATOR
; DS 10H ;SPACE FOR OUTPUT ROUTINE
; RET
LIST:  ;LIST CHARACTER FROM REGISTER C
3ED1 79  MOV  A,C  ;CHARACTER TO REGISTER A
3ED2 C9  RET  ;NULL SUBROUTINE

PUNCH:  ;PUNCH CHARACTER FROM REGISTER C
3ED3 79  MOV  A,C  ;CHARACTER TO REGISTER A
3ED4 C9  RET  ;NULL SUBROUTINE

READER:  ;READ CHARACTER INTO REGISTER A FROM READER DEVICE
3ED5 3E1A  MOV  A,1AH  ;ENTER END OF FILE FOR NOW (REPLACE LATER)
3ED7 E67F  ANI  7FH  ;REMEMBER TO STRIP PARITY BIT
3ED9 C9  RET

I/O DRIVERS FOR THE DISK FOLLOW
; FOR NOW, WE WILL SIMPLY STORE THE PARAMETERS AWAY FOR USE
; IN THE READ AND WRITE SUBROUTINES

HOME:  ;MOVE TO THE TRACK 00 POSITION OF CURRENT DRIVE
; TRANSLATE THIS CALL INTO A SETTRK CALL WITH PARAMETER 00
3EDE 0E00  MVI  C,0  ;SELECT TRACK 0
3EDC CDF53E  CALL  SETTRK
3EDF C9  RET  ;WE WILL MOVE TO 00 ON FIRST READ/WRITE

SELSDK:  ;SELECT DISK GIVEN BY REGISTER C
3EE0 79  MOV  A,C
3EE1 324000  STA  DISKNO
3EE4  10H  DS  SPACE FOR DISK SELECTION ROUTINE
3EF4 C9  RET

SETTRK:  ;SET TRACK GIVEN BY REGISTER C
3EF5 79  MOV  A,C
3EF6 324000  STA  TRACK
3EF9  10H  DS  SPACE FOR TRACK SELECT
3F09 C9  RET

SETEC:  ;SET SECTOR GIVEN BY REGISTER C
3F0A 79  MOV  A,C
3F0B 324100  STA  SECTOR
3F0E  10H  DS  SPACE FOR SECTOR SELECT
3F1E C9  RET

SETDMA:  ;SET DMA ADDRESS GIVEN BY REGISTERS B AND C
3F1F 69  MOV  L,C  ;LOW ORDER ADDRESS
3F20 60  MOV  H,B  ;HIGH ORDER ADDRESS
3F21 224200  SHLD  DMAAD  ;SAVE THE ADDRESS
3F24  10H  DS  SPACE FOR SETTING THE DMA ADDRESS
3F34 C9  RET
; READ: ;PERFORM READ OPERATION (USUALLY THIS IS SIMILAR TO WRITE
; SO WE WILL ALLOW SPACE TO SET UP READ COMMAND, THEN USE
; COMMON CODE IN WRITE)
3F35 DS 10H ;SET UP READ COMMAND
3F45 C3583F JMP WAITIO ;TO PERFORM THE ACTUAL I/O

; WRITE: ;PERFORM A WRITE OPERATION
3F48 DS 10H ;SET UP WRITE COMMAND

; WAITIO: ;ENTER HERE FROM READ AND WRITE TO PERFORM THE ACTUAL I/O
; OPERATION. RETURN A 00H IN REGISTER A IF THE OPERATION COMPLETES
; PROPERLY, AND 01H IF AN ERROR OCCURS DURING THE READ OR WRITE
;
; IN THIS CASE, WE HAVE SAVED THE DISK NUMBER IN 'DISKNO' (0,1)
; THE TRACK NUMBER IN 'TRACK' (0-76)
; THE SECTOR NUMBER IN 'SECTOR' (1-26)
; THE DMA ADDRESS IN 'DMAAD' (0-65535)
;
00A7 = ALL REMAINING SPACE FROM $ THROUGH MSIZE*1024-1 IS AVAILABLE:
LEFT EQU (MSIZE*1024-1)-$ ;SPACE REMAINING IN CBIOŚ

3F58 3E01 MVI A,1 ;ERROR CONDITION
3F5A C9 RET ;REPLACED WHEN FILLED-IN
3F5B END
APPENDIX E: A SKELETAL GETSYS/PUTSYS PROGRAM

COMBINED GETSYS AND PUTSYS PROGRAMS FROM SECTION 4

START THE PROGRAMS AT THE BASE OF THE TRANSIENT PROGRAM AREA
ORG 100H

0100

MSIZE EQU 16 ; SIZE OF MEMORY IN KILOBYTES
BIAS IS THE AMOUNT TO ADD TO ADDRESSES FOR SYSTEMS LARGER THAN 16K
(REFERRED TO AS 'B' THROUGHOUT THE TEXT)

0000 =

BIAS EQU (MSIZE-16)*1024

GETSYS PROGRAM - READ TRACKS 0 AND 1 TO MEMORY AT 2880H+BIAS
REGISTER USE
A (SCRATCH REGISTER)
B TRACK COUNT (0...76)
C SECTOR COUNT (1...26)
D,E (SCRATCH REGISTER PAIR)
H,L LOAD ADDRESS
SP SET TO STACK ADDRESS

GSTART:

0100 318028
0103 218028
0106 0600

RDTRK:

0108 0E01

RDSEC:

0110 0D0003
0110 D118000
0111 19
0112 0C
0112 79
0113 F11B
0115 DA0A01

0118 04
0119 78
011A FE02
011C DA0801

011F FB
0120 76

0200

GETSYS PROGRAM - READ TRACKS 0 AND 1 TO MEMORY AT 2880H+BIAS

; START OF THE GETSYS PROGRAM
; SET STACK POINTER TO SCRATCH AREA
; SET BASE LOAD ADDRESS
; START WITH TRACK 00
; READ FIRST (NEXT) TRACK
; READ STARTING WITH SECTOR 1

CALL READSEC
;

; READ NEXT SECTOR
; CHANGE LOAD ADDRESS TO NEXT 1/2 PAGE
; HL=HL+128 TO NEXT ADDRESS
; SECTOR=SECTOR+1
; CHECK FOR END OF TRACK

; CARRY GENERATED IF C<27

; ARRIVE HERE AT END OF TRACK, MOVE TO NEXT TRACK
; TRACK=TRACK+1
; CHECK FOR LAST TRACK
; TRACK=27
; CARRY GENERATED IF TRACK < 2

; ARRIVE HERE AT END OF LOAD, HALT FOR NOW
EI

Halt

; PUTSYS PROGRAM - PLACE MEMORY STARTING AT 2880H+BIAS BACK TO TRACKS
0 AND 1. START THIS PROGRAM ON THE NEXT PAGE
ORG ($+100H) AND 0FF00H
; REGISTER USE
; A (SCRATCH REGISTER)
; B TRACK COUNT (0,1)
; C SECTOR COUNT (1...26)
; D,E (SCRATCH REGISTER PAIR)
; H,L DUMP ADDRESS
; SP SET TO STACK ADDRESS

PSTART: ;START OF THE PUTFYS PROGRAM
0200 310828 LXI SP, 2000H+BIAS ;SET STACK POINTER TO SCRATCH AREA
0203 210828 LXI H, 28000H+BIAS ;SET BASE DUMP ADDRESS
0206 0600 MVI B, 0 ;START WITH TRACK 0
WRTRK:
0208 0E01 MVI C, 1 ;WRITE FIRST (NEXT) TRACK
;START WRITING AT SECTOR 1
WRSEC:
020A CD0003 CALL WRITSEC ;WRITE FIRST (NEXT) SECTOR
020D 180000 LXI D, 128 ;PERFORM THE WRITE
0210 0F DD19 DAD D ;MOVE DUMP ADDRESS TO NEXT 1/2 PAGE
0211 0C INR C ;HL=HL+128
0212 75 MOV A, C ;SECTOR=SECTOR+1
0213 FE1B CPI 27 ;CHECK FOR END OF TRACK
0215 DA0A02 JC WRSEC ;SECTOR=27?
; CARRY GENERATED IF SECTOR < 27
;
; ARRIVE HERE AT END OF TRACK, MOVE TO NEXT TRACK
0218 04 INR B ;TRACK=TRACK+1
0219 78 MOV A, B ;TEST FOR LAST TRACK
021A FE02 CPI 2 ;TRACK=2?
021C DA0802 JC WRTRK ;CARRY GENERATED IF TRACK < 2
;
; ARRIVE HERE AT END OF DUMP, HALT FOR NOW
021F FB EI
0220 76 HLT
;
USER-SUPPLIED SUBROUTINES FOR SECTOR READ AND SECTOR WRITE
;
MOVE TO NEXT PAGE FOR READSEC AND WRITSEC
0300 ORG ($+100H) AND 0FF00H
Readsec: ;READ THE NEXT SECTOR
; TRACK TO READ IS IN REGISTER B
; SECTOR TO READ IS IN REGISTER C
; BRANCH TO LABEL PSTART IF ERROR OCCURS
; READ 128 BYTES OF DATA TO ADDRESS GIVEN BY H,L
0300 C5 PUSH B
0301 E5 PUSH H
; ** PLACE READ OPERATION HERE **
0302 E1 POP H
0303 C1 POP B
0304 C9 RET

E-2
; MOVE TO NEXT 1/2 PAGE FOR WRITESEC SUBROUTINE
0380 ORG ($ AND 0FF00H) + 80H

WRITESEC: ; WRITE THE NEXT SECTOR
; TRACK TO WRITE IS IN REGISTER B
; SECTOR TO WRITE IS IN REGISTER C
; BRANCH TO LABEL PSTART IF ERROR OCCURS
; WRITE 128 BYTES OF DATA FROM ADDRESS GIVEN BY H,L
0380 C5
0381 E5
; ** PLACE WRITE OPERATION HERE **
PUSH B
PUSH H
0382 E1
0383 C1
0384 C9
POP H
POP B
RET

0385 ; END OF GETSYS/PUTSYS PROGRAM
END
APPENDIX F: A SKELETAL COLD START LOADER

THIS IS A SAMPLE COLD START LOADER WHICH, WHEN MODIFIED, RESIDES ON TRACK 00, SECTOR 01 (THE FIRST SECTOR ON THE DISKETTE). WE ASSUME THAT THE CONTROLLER HAS LOADED THIS SECTOR INTO MEMORY UFON SYSTEM STARTUP (THIS PROGRAM CAN BE KEYED-IN, OR EXIST IN A PAGE OF READ-ONLY MEMORY BEYOND THE ADDRESS SPACE OF THE CP/M VERSION YOU ARE RUNNING). THE COLD START LOADER BRINGS THE CP/M SYSTEM INTO MEMORY AT 'LOADP' (NORMALLY 2900H) + 'BIAS' WHERE THE BIAS VALUE ACCOUNTS FOR MEMORY SYSTEMS LARGER THAN 16K, AND CP/M VERSIONS WHICH HANDLE THE LARGER MEMORY SPACE. IN A 16K SYSTEM, THE VALUE OF BIAS IS 0000H. AFTER LOADING THE CP/M SYSTEM, THE COLD START LOADER BRANCHES TO THE 'BOOT' ENTRY POINT OF THE BIOS, WHICH BEGINS AT 'BIOS' + 'BIAS'. THE COLD START LOADER IS NOT USED AGAIN UNTIL THE SYSTEM IS POWERED UP AGAIN, AS LONG AS THE BIOS IS NOT OVERWRITTEN.

THE ORIGIN IS 0, ASSUMING THE CONTROLLER LOADS THE COLD START PROGRAM AT THE BASE OF MEMORY. THIS ORIGIN MUST BE IN HIGH MEMORY (BEYOND THE END OF THE BIOS) IF THE COLD START LOADER IS IMPLEMENTED IN READ-ONLY-MEMORY.

```
0000 ORG 0000H ;BASE OF MEMORY
0010 = MSIZE EQU 16 ;MEMORY SIZE IN KILOBYTES
0000 = BIAS EQU (MSIZE-16)*1024 ;BIAS TO ADD TO LOAD ADDRESSES
2900 = LOADP EQU 2900H ;LOAD POINT FOR CP/M SYSTEM
3E00 = BIOS EQU 3E00H ;BASIC I/O SYSTEM (2 PAGES = 512 BYTES)
3E00 = BOOT EQU BIOS ;COLD START ENTRY POINT IN BIOS
1700 = SIZE EQU BIOS+512-LOADP ;SIZE OF THE CP/M SYSTEM TO LOAD
002E = SECTS EQU SIZE/128 ;NUMBER OF SECTORS TO LOAD

BEGIN THE LOAD OPERATION

0000 010200 COLD: LXI B,2 ;CLEAR B TO 0, SET C TO SECTOR 2
0003 162E MVI D,SECTS ;NUMBER OF SECTORS TO LOAD IS IN D
0005 210029 LXI H,LOADP+BIAS ;LOAD POINT IN H,L

LSECT: ;LOAD NEXT SECTOR

; INSERT IN-LINE CODE AT THIS POINT TO READ ONE 128 BYTE SECTOR FROM TRACK GIVEN BY REGISTER B,
; SECTOR GIVEN BY REGISTER C,
; INTO ADDRESS GIVEN BY REGISTER PAIR H,L
; BRANCH TO LOCATION 'COLD' IF A READ ERROR OCCURS

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

; USER SUPPLIED READ OPERATION GOES HERE
*************************************************************************

(SPACE IS RESERVED FOR YOUR PATCH)

0008 C36B00 JMP PASTPATCH ;REMOVE THIS JUMP WHEN PATCHED
000B DB 60H
```
; PASTPATCH:
; GO TO NEXT SECTOR IF LOAD IS INCOMPLETE
006B 15 DCR D ;SECTS=SECTS-1
006C CA003E JZ BOOT+BIAS ;GO TO BOOT LOADER AT 3E00H+BIAS

; MORE SECTORS TO LOAD
; USE SP FOR SCRATCH REGISTER TO HOLD LOAD ADDRESS INCREMENT
006F 318000 LXI SP,128
0072 39 DAD SP ;HL=HL+128 TO NEXT LOAD ADDRESS

0073 0C INR C ;SECTOR=SECTOR+1
0074 79 MOV A,C ;MOVE SECTOR COUNT TO A FOR COMPARE
0075 FE1B CPI 27 ;END OF CURRENT TRACK?
0077 DA0800 JC LSECT ;CARRY GENERATED IF SECTOR < 27

; END OF TRACK, MOVE TO NEXT TRACK
007A 0E01 MVI C,1 ;SECTOR=1
007C 04 INR B ;TRACK=TRACK+1
007D C30800 JMP LSECT ;FOR ANOTHER SECTOR

0080 END
ED: A CONTEXT EDITOR FOR THE CP/M DISK SYSTEM

USER'S MANUAL
# Table of Contents

1. ED TUTORIAL ............................................ 1
   1.1 Introduction to ED ................................. 1
   1.2 ED Operation ...................................... 1
   1.3 Text Transfer Functions ............................. 1
   1.4 Memory Buffer Organization .......................... 5
   1.5 Memory Buffer Operation ............................. 5
   1.6 Command Strings .................................... 7
   1.7 Text Search and Alteration .......................... 8
   1.8 Source Libraries .................................... 11
   1.9 Repetitive Command Execution ........................ 12

2. ED ERROR CONDITIONS ................................. 13

3. CONTROL CHARACTERS AND COMMANDS ............... 14
1. ED TUTORIAL

1.1. Introduction to ED.

ED is the context editor for CP/M, and is used to create and alter CP/M source files. ED is initiated in CP/M by typing

```
ED { <filename> }
 { <filename>.<filetype> }
```

In general, ED reads segments of the source file given by <filename> or <filename>.<filetype> into central memory, where the file is manipulated by the operator, and subsequently written back to disk after alterations. If the source file does not exist before editing, it is created by ED and initialized to empty. The overall operation of ED is shown in Figure 1.

1.2. ED Operation

ED operates upon the source file, denoted in Figure 1 by x.y, and passes all text through a memory buffer where the text can be viewed or altered (the number of lines which can be maintained in the memory buffer varies with the line length, but has a total capacity of about 6000 characters in a 16K CP/M system). Text material which has been edited is written onto a temporary work file under command of the operator. Upon termination of the edit, the memory buffer is written to the temporary file, followed by any remaining (unread) text in the source file. The name of the original file is changed from x.y to x.BAK so that the most recent previously edited source file can be reclaimed if necessary (see the CP/M commands ERASE and RENAME). The temporary file is then changed from x.$$ to x.y which becomes the resulting edited file.

The memory buffer is logically between the source file and working file as shown in Figure 2.

1.3. Text Transfer Functions

Given that n is an integer value in the range 0 through 65535, the following ED commands transfer lines of text from the source file through the memory buffer to the temporary (and eventually final) file:
Figure 1. Overall ED Operation

Note: the ED program accepts both lower and upper case ASCII characters as input from the console. Single letter commands can be typed in either case. The U command can be issued to cause ED to translate lower case alphabatics to upper case as characters are filled to the memory buffer from the console. Characters are echoed as typed without translation, however. The -U command causes ED to revert to "no translation" mode. ED starts with an assumed -U in effect.
Figure 2. Memory Buffer Organization

Figure 3. Logical Organization of Memory Buffer

Memory Buffer

first line --------<cr><lf>

----------<cr><lf>

current line CL --<cr><lf>

last line --------<cr><lf>
nA<cr>* - append the next n unprocessed source lines from the source file at SP to the end of the memory buffer at MP. Increment SP and MP by n.

nW<cr> - write the first n lines of the memory buffer to the temporary file free space. Shift the remaining lines n+1 through MP to the top of the memory buffer. Increment TP by n.

E<cr> - end the edit. Copy all buffered text to temporary file, and copy all unprocessed source lines to the temporary file. Rename files as described previously.

H<cr> - move to head of new file by performing automatic E command. Temporary file becomes the new source file, the memory buffer is emptied, and a new temporary file is created (equivalent to issuing an E command, followed by a reinvocation of ED using x,y as the file to edit).

O<cr> - return to original file. The memory buffer is emptied, the temporary file is deleted, and the SP is returned to position 1 of the source file. The effects of the previous editing commands are thus nullified.

Q<cr> - quit edit with no file alterations, return to CP/M.

There are a number of special cases to consider. If the integer n is omitted in any ED command where an integer is allowed, then 1 is assumed. Thus, the commands A and W append one line and write 1 line, respectively. In addition, if a pound sign (#) is given in the place of n, then the integer 65535 is assumed (the largest value for n which is allowed). Since most reasonably sized source files can be contained entirely in the memory buffer, the command #A is often issued at the beginning of the edit to read the entire source file to memory. Similarly, the command #W writes the entire buffer to the temporary file. Two special forms of the A and W

<cr> represents the carriage-return key
commands are provided as a convenience. The command OA fills the current memory buffer to at least half-full, while OW writes lines until the buffer is at least half empty. It should also be noted that an error is issued if the memory buffer size is exceeded. The operator may then enter any command (such as W) which does not increase memory requirements. The remainder of any partial line read during the overflow will be brought into memory on the next successful append.

1.4. Memory Buffer Organization

The memory buffer can be considered a sequence of source lines brought in with the A command from a source file. The memory buffer has an associated (imaginary) character pointer CP which moves throughout the memory buffer under command of the operator. The memory buffer appears logically as shown in Figure 3 where the dashes represent characters of the source line of indefinite length, terminated by carriage-return (<cr>) and line-feed (<lf>) characters, and [CP] represents the imaginary character pointer. Note that the CP is always located ahead of the first character of the first line, behind the last character of the last line, or between two characters. The current line CL is the source line which contains the CP.

1.5. Memory Buffer Operation

Upon initiation of ED, the memory buffer is empty (i.e., CP is both ahead and behind the first and last character). The operator may either append lines (A command) from the source file, or enter the lines directly from the console with the insert command

I<cr>

ED then accepts any number of input lines, where each line terminates with a <cr> (the <lf> is supplied automatically), until a control-z (denoted by +z is typed by the operator. The CP is positioned after the last character entered. The sequence

I<cr>
NOW IS THE<cr>
TIME FOR<cr>
ALL GOOD MEN<cr>
+z

leaves the memory buffer as shown below
Various commands can then be issued which manipulate the CP or display source text in the vicinity of the CP. The commands shown below with a preceding n indicate that an optional unsigned value can be specified. When preceded by the command can be unsigned, or have an optional preceding plus or minus sign. As before, the pound sign (#) is replaced by 65535. If an integer n is optional, but not supplied, then n=1 is assumed. Finally, if a plus sign is optional, but none is specified, then + is assumed.

±B<cr> - move CP to beginning of memory buffer if +, and to bottom if -.

±nC<cr> - move CP by ±n characters (toward front of buffer if +), counting the <cr><lf> as two distinct characters

±nD<cr> - delete n characters ahead of CP if plus and behind CP if minus.

±nK<cr> - kill (ie remove) ±n lines of source text using CP as the current reference. If CP is not at the beginning of the current line when K is issued, then the characters before CP remain if + is specified, while the characters after CP remain if - is given in the command.

±nL<cr> - if n=0 then move CP to the beginning of the current line (if it is not already there) if n≠0 then first move the CP to the beginning of the current line, and then move it to the beginning of the line which is n lines down (if +) or up (if -). The CP will stop at the top or bottom of the memory buffer if too large a value of n is specified.
±nT<cr> - If n=0 then type the contents of the current line up to CP. If n=1 then type the contents of the current line from CP to the end of the line. If n>1 then type the current line along with n-1 lines which follow, if + is specified. Similarly, if n>1 and - is given, type the previous n lines, up to the CP. The break key can be depressed to abort long type-outs.

±n<cr> - equivalent to ±nLT, which moves up or down and types a single line

1.6. Command Strings

Any number of commands can be typed contiguously (up to the capacity of the CP/M console buffer), and are executed only after the <cr> is typed. Thus, the operator may use the CP/M console command functions to manipulate the input command:

- Rubout remove the last character
- Control-U delete the entire line
- Control-C re-initialize the CP/M System
- Control-E return carriage for long lines without transmitting buffer (max 128 chars)

Suppose the memory buffer contains the characters shown in the previous section, with the CP following the last character of the buffer. The command strings shown below produce the results shown to the right:

<table>
<thead>
<tr>
<th>Command String</th>
<th>Effect</th>
<th>Resulting Memory Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. B2T&lt;cr&gt;</td>
<td>move to beginning of buffer and type 2 lines: &quot;NOW IS THE TIME FOR&quot;</td>
<td>CP NOW IS THE&lt;cr&gt;&lt;lf&gt; TIME FOR&lt;cr&gt;&lt;lf&gt;</td>
</tr>
<tr>
<td>2. 5C0T&lt;cr&gt;</td>
<td>move CP 5 characters and type the beginning of the line &quot;NOW I&quot;</td>
<td>NOW I CPU S THE&lt;cr&gt;&lt;lf&gt;</td>
</tr>
</tbody>
</table>
3. 2L-T<cr> move two lines down and type previous line "TIME FOR" NOW IS THE<cr><lf> TIME FOR<cr><lf> ALL GOOD MEN<cr><lf>

4. -L#K<cr> move up one line, delete 65535 lines which follow NOW IS THE<cr><lf>

5. I<cr> insert two lines of text NOW IS THE<cr><lf> TIME TO<cr><lf> INSERT<cr><lf> 
+z

6. -2L#T<cr> move up two lines, and type 65535 lines ahead of CP "NOW IS THE" NOW IS THE<cr><lf>

7. <cr> move down one line and type one line "INSERT" NOW IS THE<cr><lf> TIME TO<cr><lf> INSERT<cr><lf>

1.7. Text Search and Alteration

ED also has a command which locates strings within the memory buffer. The command takes the form

\[ nF \ C_1 C_2 \ldots C_k \{<cr>\} \plus\]

where \( C_1 \) through \( C_k \) represent the characters to match followed by either a \(<cr>\) or control \(-z\)\. ED starts at the current position of CP and attempts to match all \( k \) characters. The match is attempted \( n \) times, and if successful, the CP is moved directly after the character \( C_k \). If the \( n \) matches are not successful, the CP is not moved from its initial position. Search strings can include \(+Z\) (control-1), which is replaced by the pair of symbols \(<cr><lf>\).

*The control-z is used if additional commands will be typed following the \(+z\).*
The following commands illustrate the use of the F command:

<table>
<thead>
<tr>
<th>Command String</th>
<th>Effect</th>
<th>Resulting Memory Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. B#T&lt;cr&gt;</td>
<td>move to beginning and type entire buffer</td>
<td>NOW IS THE&lt;cr&gt;&lt;lf&gt;&lt;br/&gt;TIME FOR&lt;cr&gt;&lt;lf&gt; ALL GOOD MEN&lt;cr&gt;&lt;lf&gt;</td>
</tr>
<tr>
<td>2. FS T&lt;cr&gt;</td>
<td>find the end of the string &quot;S T&quot;</td>
<td>NOW IS T CP HE&lt;cr&gt;&lt;lf&gt;</td>
</tr>
<tr>
<td>3. FI+z0TT</td>
<td>find the next &quot;I&quot; and type to the CP then type the remainder of the current line: &quot;TIME FOR&quot;</td>
<td>NOW IS THE&lt;cr&gt;&lt;lf&gt; TI CP ME FOR&lt;cr&gt;&lt;lf&gt; ALL GOOD MEN&lt;cr&gt;&lt;lf&gt;</td>
</tr>
</tbody>
</table>

An abbreviated form of the insert command is also allowed, which is often used in conjunction with the F command to make simple textual changes. The form is:

\[ I \, c_1 \, c_2 \ldots \, c_n \, z \, \text{ or } \]

\[ I \, c_1 \, c_2 \ldots \, c_n \, <cr> \]

where \( c_1 \) through \( c_n \) are characters to insert. If the insertion string is terminated by a \( z \), the characters \( c_1 \) through \( c_n \) are inserted directly following the CP, and the CP is moved directly after character \( c_n \). The action is the same if the command is followed by a \( <cr> \) except that a \( <cr><lf> \) is automatically inserted into the text following character \( c_n \). Consider the following command sequences as examples of the F and I commands:

<table>
<thead>
<tr>
<th>Command String</th>
<th>Effect</th>
<th>Resulting Memory Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTHIS IS +z&lt;cr&gt;</td>
<td>Insert &quot;THIS IS &quot; at the beginning of the text</td>
<td>THIS IS NOW THE &lt;cr&gt;&lt;lf&gt;&lt;br/&gt;TIME FOR&lt;cr&gt;&lt;lf&gt; ALL GOOD MEN&lt;cr&gt;&lt;lf&gt;</td>
</tr>
</tbody>
</table>
n command is precisely the same as F except in the case that the string cannot be found within the current memory buffer. In this case, the entire memory contents is written (ie, an automatic #W is issued). Input lines are then read until the buffer is at least half full, or the entire source file is exhausted. The search continues in this manner until the string has been found n times, or until the source file has been completely transferred to the temporary file.

A final line editing function, called the juxtaposition command takes the form

\[ n \ J \ c_1 c_2 \ldots c_k \ z \ d_1 d_2 \ldots d_m \ z \ e_1 e_2 \ldots e_q (\text{<cr>})^z \]

with the following action applied n times to the memory buffer: search from the current CP for the next occurrence of the string \( c_1 c_2 \ldots c_k \). If found, insert the string \( d_1 d_2 \ldots d_m \), and move CP to follow \( d_m \). Then delete all characters following CP up to (but not including) the string \( e_1 e_2 \ldots e_q \), leaving CP directly after \( d_m \). If \( e_1 e_2 \ldots e_q \) cannot be found, then no deletion is made. If the current line is

\[ \hat{\text{CP}} \text{ NOW IS THE TIME<cr><lf> } \]

Then the command

\[ \text{JW} \ uparrow z \text{WHAT} \ uparrow z \ uparrow 1 \text{<cr>} \]

Results in

\[ \text{NOW WHAT} \hat{\text{CP}} \text{<cr><lf>} \]

(Recall that \( \uparrow 1 \) represents the pair \text{<cr><lf>} in search and substitute strings).

It should be noted that the number of characters allowed by ED in the F,S,N, and J commands is limited to 100 symbols.

1.8. Source Libraries

ED also allows the inclusion of source libraries during the editing process with the R command. The form of this command is
R \{f_1 f_2 \ldots f_n\} \text{ or }
R \{f_1 f_2 \ldots f_n\}<cr>

where \{f_1 f_2 \ldots f_n\} is the name of a source file on the disk with
as assumed filetype of 'LIB'. ED reads the specified file,
and places the characters into the memory buffer after CP,
in a manner similar to the I command. Thus, if the command

RMACRO<cr>

is issued by the operator, ED reads from the file MACRO.LIB
until the end-of-file, and automatically inserts the charac-
ters into the memory buffer.

1.9. Repetitive Command Execution

The macro command M allows the ED user to group ED com-
mands together for repeated evaluation. The M command takes
the form:

n M c_1 c_2 \ldots c_k \{<cr>\}

where \{c_1 c_2 \ldots c_k\} represent a string of ED commands, not inclu-
ding another M command. ED executes the command string n
times if n>1. If n=0 or 1, the command string is executed
repetitively until an error condition is encountered (e.g.,
the end of the memory buffer is reached with an F command).
As an example, the following macro changes all occur-
rences of GAMMA to DELTA within the current buffer, and
types each line which is changed:

MFGAMMA+z-5DIDELTA+z0TT<cr>

or equivalently

MSGAMMA+zDELTA+z0TT<cr>
2. ED ERROR CONDITIONS

On error conditions, ED prints the last character read before the error, along with an error indicator:

?     unrecognized command
>
memory buffer full (use one of the commands D,K,N,S, or W to remove characters), F,N, or S strings too long.

#     cannot apply command the number of times specified (e.g., in F command)

0     cannot open LIB file in R command

Cyclic redundancy check (CRC) information is written with each output record under CP/M in order to detect errors on subsequent read operations. If a CRC error is detected, CP/M will type

PERM ERR DISK d

where d is the currently selected drive (A,B,...). The operator can choose to ignore the error by typing any character at the console (in this case, the memory buffer data should be examined to see if it was incorrectly read), or the user can reset the system and reclaim the backup file, if it exists. The file can be reclaimed by first typing the contents of the BAK file to ensure that it contains the proper information:

TYPE x.BAK<cr>

where x is the file being edited. Then remove the primary file:

ERA x.y<cr>

and rename the BAK file:

REN x.y=x.BAK<cr>

The file can then be re-edited, starting with the previous version.
<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>nA</td>
<td>append lines</td>
</tr>
<tr>
<td>±B</td>
<td>begin bottom of buffer</td>
</tr>
<tr>
<td>±nC</td>
<td>move character positions</td>
</tr>
<tr>
<td>±nD</td>
<td>delete characters</td>
</tr>
<tr>
<td>E</td>
<td>end edit and close files (normal end)</td>
</tr>
<tr>
<td>nF</td>
<td>find string</td>
</tr>
<tr>
<td>H</td>
<td>end edit, close and reopen files</td>
</tr>
<tr>
<td>I</td>
<td>insert characters</td>
</tr>
<tr>
<td>nJ</td>
<td>place strings in juxtaposition</td>
</tr>
<tr>
<td>±nK</td>
<td>kill lines</td>
</tr>
<tr>
<td>±nL</td>
<td>move down/up lines</td>
</tr>
<tr>
<td>nM</td>
<td>macro definition</td>
</tr>
<tr>
<td>nN</td>
<td>find next occurrence with autoscan</td>
</tr>
<tr>
<td>O</td>
<td>return to original file</td>
</tr>
<tr>
<td>±nP</td>
<td>move and print pages</td>
</tr>
<tr>
<td>Q</td>
<td>quit with no file changes</td>
</tr>
<tr>
<td>R</td>
<td>read library file</td>
</tr>
<tr>
<td>nS</td>
<td>substitute strings</td>
</tr>
<tr>
<td>±nT</td>
<td>type lines</td>
</tr>
<tr>
<td>± U</td>
<td>translate lower to upper case if U, no translation if -U</td>
</tr>
<tr>
<td>nW</td>
<td>write lines</td>
</tr>
<tr>
<td>nZ</td>
<td>sleep</td>
</tr>
<tr>
<td>±n&lt;cr&gt;</td>
<td>move and type (±nLT)</td>
</tr>
</tbody>
</table>
Appendix A: ED 1.4 Enhancements

The ED context editor contains a number of commands which enhance its usefulness in text editing. The improvements are found in the addition of line numbers, free space interrogation, and improved error reporting.

The context editor issued with CP/M 1.4 produces absolute line number prefixes when the "V" (Verify Line Numbers) command is issued. Following the V command, the line number is displayed ahead of each line in the format:

nnnnn:

where nnnnn is an absolute line number in the range 1 to 65535. If the memory buffer is empty, or if the current line is at the end of the memory buffer, then nnnnn appears as 5 blanks.

The user may reference an absolute line number by preceding any command by a number followed by a colon, in the same format as the line number display. In this case, the ED program moves the current line reference to the absolute line number, if the line exists in the current memory buffer. Thus, the command

345:T

is interpreted as "move to absolute line 345, and type the line." Note that absolute line numbers are produced only during the editing process, and are not recorded with the file. In particular, the line numbers will change following a deleted or expanded section of text.

The user may also reference an absolute line number as a backward or forward distance from the current line by preceding the absolute line number by a colon. Thus, the command

:400T

is interpreted as "type from the current line number through the line whose absolute number is 400." Combining the two line reference forms, the command

345::400T

for example, is interpreted as "move to absolute line 345, then type through absolute line 400." Note that absolute line references of this sort can precede any of the standard ED commands.

A special case of the V command, "@V", prints the memory buffer statistics in the form:

free/total

where "free" is the number of free bytes in the memory buffer (in decimal), and "total" is the size of the memory buffer.
ED 1.4 also includes a "block move" facility implemented through the "X" (Xfer) command. The form

\[ nX \]

transfers the next n lines from the current line to a temporary file called

\[ X$$$$$.LIB \]

which is active only during the editing process. In general, the user can reposition the current line reference to any portion of the source file and transfer lines to the temporary file. The transferred line accumulate one after another in this file, and can be retrieved by simply typing:

\[ R \]

which is the trivial case of the library read command. In this case, the entire transferred set of lines is read into the memory buffer. Note that the X command does not remove the transferred lines from the memory buffer, although a K command can be used directly after the X, and the R command does not empty the transferred line file. That is, given that a set of lines has been transferred with the X command, they can be re-read any number of times back into the source file. The command

\[ nX \]

is provided, however, to empty the transferred line file.

Note that upon normal completion of the ED program through Q or E, the temporary LIB file is removed. If ED is aborted through ctl-C, the LIB file will exist if lines have been transferred, but will generally be empty (a subsequent ED invocation will erase the temporary file).

Due to common typographical errors, ED 1.4 requires several potentially disastrous commands to be typed as single letters, rather than in composite commands. The commands

\[ E \text{ (end), } H \text{ (head), } O \text{ (original), } Q \text{ (quit)} \]

must be typed as single letter commands.

ED 1.4 also prints error messages in the form

\[ \text{BREAK "x" AT c} \]

where x is the error character, and c is the command where the error occurred.
CP/M DYNAMIC DEBUGGING TOOL (DDT)

USER'S GUIDE
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. DDT COMMANDS</td>
<td>3</td>
</tr>
<tr>
<td>1. The A (Assemble) Command</td>
<td>3</td>
</tr>
<tr>
<td>2. The D (Display) Command</td>
<td>4</td>
</tr>
<tr>
<td>3. The F (Fill) Command</td>
<td>4</td>
</tr>
<tr>
<td>4. The G (Go) Command</td>
<td>4</td>
</tr>
<tr>
<td>5. The I (Input) Command</td>
<td>5</td>
</tr>
<tr>
<td>6. The L (List) Command</td>
<td>6</td>
</tr>
<tr>
<td>7. The M (Move) Command</td>
<td>6</td>
</tr>
<tr>
<td>8. The R (Read) Command</td>
<td>6</td>
</tr>
<tr>
<td>9. The S (Set) Command</td>
<td>7</td>
</tr>
<tr>
<td>10. The T (Trace) Command</td>
<td>7</td>
</tr>
<tr>
<td>11. The U (Untrace) Command</td>
<td>8</td>
</tr>
<tr>
<td>12. The X (Examine) Command</td>
<td>8</td>
</tr>
<tr>
<td>III. IMPLEMENTATION NOTES</td>
<td>9</td>
</tr>
<tr>
<td>IV. AN EXAMPLE</td>
<td>10</td>
</tr>
</tbody>
</table>
CP/M Dynamic Debugging Tool (DDT)

User's Guide

I. Introduction

The DDT program allows dynamic interactive testing and debugging of programs generated in the CP/M environment. The debugger is initiated by typing one of the following commands at the CP/M Console Command level:

DDT
DDT filename.HEX
DDT filename.COM

where "filename" is the name of the program to be loaded and tested. In both cases, the DDT program is brought into main memory in the place of the Console Command Processor (refer to the CP/M Interface Guide for standard memory organization), and thus resides directly below the Basic Disk Operating System portion of CP/M. The BDOS starting address, which is located in the address field of the JMP instruction at location 5H, is altered to reflect the reduced Transient Program Area size.

The second and third forms of the DDT command shown above perform the same actions as the first, except there is a subsequent automatic load of the specified HEX or COM file. The action is identical to the sequence of commands

DDT
Ifilename.HEX or Ifilename.COM
R

where the I and R commands set up and read the specified program to test (see the explanation of the I and R commands below for exact details).

Upon initiation, DDT prints the following sign-on message.

DDT VERS n.n

where nn represents the version number.
p2:  A, B, ..., Y  designates the disk name which will receive the hex file
     Z  skips the generation of the hex file

p3:  A, B, ..., Y  designates the disk name which will receive the print file
     X  places the listing at the console
     Z  skips generation of the print file

Thus, the command

     ASM  X.AAA

indicates that the source file (X.ASM) is to be taken from disk A, and that the hex (X.HEX) and print (X.PRN) files are to be created also on disk A. This form of the command is implied if the assembler is run from disk A. That is, given that the operator is currently addressing disk A, the above command is equivalent to

     ASM  X

The command

     ASM  X.ABX

indicates that the source file is to be taken from disk A, the hex file is placed on disk B, and the listing file is to be sent to the console. The command

     ASM  X.BZZ

takes the source file from disk B, and skips the generation of the hex and print files (this command is useful for fast execution of the assembler to check program syntax).

The source program format is compatible with both the Intel 8080 assembler (macros are not currently implemented in the CP/M assembler, however), as well as the Processor Technology Software Package #1 assembler. That is, the CP/M assembler accepts source programs written in either format. There are certain extensions in the CP/M assembler which make it somewhat easier to use. These extensions are described below.

2. PROGRAM FORMAT.

An assembly language program acceptable as input to the assembler consists of a sequence of statements of the form

     line#   label   operation   operand   ; comment

where any or all of the fields may be present in a particular instance. Each
DDT X.COM

which reloads previously saved program from loaction 100H through page 18 (12FFH). The machine state is not a part of the COM file, and thus the program must be restarted from the beginning in order to properly test it.

II. DDT COMMANDS.

The individual commands are given below in some detail. In each case, the operator must wait for the prompt character (-) before entering the command. If control is passed to a program under test, and the program has not reached a breakpoint, control can be returned to DDT by executing a RST 7 from the front panel (note that the runout key should be used instead if the program is executing a T or U command). In the explanation of each command, the command letter is shown in some cases with numbers separated by commas, where the numbers are represented by lower case letters. These numbers are always assumed to be in a hexadecimal radix, and from one to four digits in length (longer numbers will be automatically truncated on the right).

Many of the commands operate upon a "CPU state" which corresponds to the program under test. The CPU state holds the registers of the program being debugged, and initially contains zeroes for all registers and flags except for the program counter (P) and stack pointer (S), which default to 100H. The program counter is subsequently set to the starting address given in the last record of a HEX file if a file of this form is loaded (see the I and R commands).

1. The A (Assemble) Command. DDT allows inline assembly language to be inserted into the current memory image using the A command which takes the form

   As

   where s is the hexadecimal starting address for the inline assembly. DDT prompts the console with the address of the next instruction to fill, and reads the console, looking for assembly language mnemonics (see the Intel 8080 Assembly Language Reference Card for a list of mnemonics), followed by register references and operands in absolute hexadecimal form. Each successive load address is printed before reading the console. The A command terminates when the first empty line is input from the console.

   Upon completion of assembly language input, the operator can review the memory segment using the DDT disassembler (see the L command).

   Note that the assembler/disassembler portion of DDT can be overlayed by the transient program being tested, in which case the DDT program responds with an error condition when the A and L commands are used (refer to Section IV).
Technology assembler has the side effect in its operation of ignoring the characters after the operand field has been scanned. This causes an ambiguous situation when attempting to be compatible with Intel's language, since arbitrary expressions are allowed in this case. Hence, programs which use this side effect to introduce comments, must be edited to place a ";:" before these fields in order to assemble correctly.

The assembly language program is formulated as a sequence of statements of the above form, terminated optionally by an END statement. All statements following the END are ignored by the assembler.

3. FORMING THE OPERAND.

In order to completely describe the operation codes and pseudo operations, it is necessary to first present the form of the operand field, since it is used in nearly all statements. Expressions in the operand field consist of simple operands (labels, constants, and reserved words), combined in properly formed subexpressions by arithmetic and logical operators. The expression computation is carried out by the assembler as the assembly proceeds. Each expression must produce a 16-bit value during the assembly. Further, the number of significant digits in the result must not exceed the intended use. That is, if an expression is to be used in a byte move immediate instruction, then the most significant 8 bits of the expression must be zero. The restrictions on the expression significance is given with the individual instructions.

3.1. Labels.

As discussed above, a label is an identifier which occurs on a particular statement. In general, the label is given a value determined by the type of statement which it precedes. If the label occurs on a statement which generates machine code or reserves memory space (e.g., a MDV instruction, or a DS pseudo operation), then the label is given the value of the program address which it labels. If the label precedes an EQU or SET, then the label is given the value which results from evaluating the operand field. Except for the SET statement, an identifier can label only one statement.

When a label appears in the operand field, its value is substituted by the assembler. This value can then be combined with other operands and operators to form the operand field for a particular instruction.

3.2. Numeric Constants.

A numeric constant is a 16-bit value in one of several bases. The base, called the radix of the constant, is denoted by a trailing radix indicator. The radix indicators are

| B | binary constant (base 2) |
| O | octal constant (base 8) |
Gs,b,c
G,b
G,b,c

The first form starts execution of the program under test at the current value of the program counter in the current machine state, with no breakpoints set (the only way to regain control in DDT is through a RST 7 execution). The current program counter can be viewed by typing an X or XP command. The second form is similar to the first except that the program counter in the current machine state is set to address s before execution begins. The third form is the same as the second, except that program execution stops when address b is encountered (b must be in the area of the program under test). The instruction at location b is not executed when the breakpoint is encountered. The fourth form is identical to the third, except that two breakpoints are specified, one at b and the other at c. Encountering either breakpoint causes execution to stop, and both breakpoints are subsequently cleared. The last two forms take the program counter from the current machine state, and set one and two breakpoints, respectively.

Execution continues from the starting address in real-time to the next breakpoint. That is, there is no intervention between the starting address and the break address by DDT. Thus, if the program under test does not reach a breakpoint, control cannot return to DDT without executing a RST 7 instruction. Upon encountering a breakpoint, DDT stops execution and types

*d

where d is the stop address. The machine state can be examined at this point using the X (Examine) command. The operator must specify breakpoints which differ from the program counter address at the beginning of the G command. Thus, if the current program counter is 1234H, then the commands

G,1234

and

G400,400

both produce an immediate breakpoint, without executing any instructions whatsoever.

5. The I (Input) Command. The I command allows the operator to insert a file name into the default file control block at 5CH (the file control block created by CP/M for transient programs is placed at this location; see the CP/M Interface Guide). The default PCB can be used by the program under test as if it had been passed by the CP/M Console Processor. Note that this file name is also used by DDT for reading additional HEX and COM files. The form of the I command is

Ifilename

or
of the instruction (e.g., MOV A,B), the value of the instruction (in this case MOV) is the bit pattern of the instruction with zeroes in the optional fields (e.g., MOV produces 40H).

When the symbol "$" occurs in the operand field (not imbedded within identifiers and numeric constants) its value becomes the address of the next instruction to generate, not including the instruction contained within the current logical line.

3.4. String Constants.

String constants represent sequences of ASCII characters, and are represented by enclosing the characters within apostrophe symbols ('). All strings must be fully contained within the current physical line (thus allowing "!" symbols within strings), and must not exceed 64 characters in length. The apostrophe character itself can be included within a string by representing it as a double apostrophe (the two keystrokes ''), which becomes a single apostrophe when read by the assembler. In most cases, the string length is restricted to either one or two characters (the DB pseudo operation is an exception), in which case the string becomes an 8 or 16 bit value, respectively. Two character strings become a 16-bit constant, with the second character as the low order byte, and the first character as the high order byte.

The value of a character is its corresponding ASCII code. There is no case translation within strings, and thus both upper and lower case characters can be represented. Note however, that only graphic (printing) ASCII characters are allowed within strings. Valid strings are

'A'    'AB'    'ab'    'c'
'      'a'    '    '    '
"Walla Walla Wash."
"She said "Hello" to me."
"I said "Hello" to her."

3.5. Arithmetic and Logical Operators.

The operands described above can be combined in normal algebraic notation using any combination of properly formed operands, operators, and parenthesized expressions. The operators recognized in the operand field are

a + b  unsigned arithmetic sum of a and b
a - b  unsigned arithmetic difference between a and b
+ b    unary plus (produces b)
- b    unary minus (identical to 0 - b)
a * b  unsigned magnitude multiplication of a and b
a / b  unsigned magnitude division of a by b
a MOD b remainder after a / b
NOT b  logical inverse of b (all 0's become 1's, 1's become 0's), where b is considered a 16-bit value
assuming the tested program does not destroy the default area at 5CH. Further, any file specified with the filetype "COM" is assumed to contain machine code in pure binary form (created with the LOAD or SAVE command), and all others are assumed to contain machine code in Intel hex format (produced, for example, with the ASM command).

Recall that the command

```
DDT filename.filetype
```

which initiates the DDT program is equivalent to the commands

```
DDT
-Ifilename.filetype
-R
```

Whenever the R command is issued, DDT responds with either the error indicator "?" (file cannot be opened, or a checksum error occurred in a HEX file), or with a load message taking the form

```
NEXT PC
nnnn pppp
```

where nnnn is the next address following the loaded program, and pppp is the assumed program counter (100H for COM files, or taken from the last record if a HEX file is specified).

9. The S (Set) Command. The S command allows memory locations to be examined and optionally altered. The form of the command is

```
Ss
```

where s is the hexadecimal starting address for examination and alteration of memory. DDT responds with a numeric prompt, giving the memory location, along with the data currently held in the memory location. If the operator types a carriage return, then the data is not altered. If a byte value is typed, then the value is stored at the prompted address. In either case, DDT continues to prompt with successive addresses and values until either a period (.) is typed by the operator, or an invalid input value is detected.

10. The T (Trace) Command. The T command allows selective tracing of program execution for 1 to 65535 program steps. The forms are

```
T
Tn
```

In the first case, the CPU state is displayed, and the next program step is executed. The program terminates immediately, with the termination address
displayed as

*hhhh

where hhhh is the next address to execute. The display address (used in the D command) is set to the value of H and L, and the list address (used in the L command) is set to hhhh. The CPU state at program termination can then be examined using the X command.

The second form of the T command is similar to the first, except that execution is traced for n steps (n is a hexadecimal value) before a program breakpoint is occurs. A breakpoint can be forced in the trace mode by typing a rubout character. The CPU state is displayed before each program step is taken in trace mode. The format of the display is the same as described in the X command.

Note that program tracing is discontinued at the interface to CP/M, and resumes after return from CP/M to the program under test. Thus, CP/M functions which access I/O devices, such as the diskette drive, run in real-time, avoiding I/O timing problems. Programs running in trace mode execute approximately 500 times slower than real time since DDT gets control after each user instruction is executed. Interrupt processing routines can be traced, but it must be noted that commands which use the breakpoint facility (G, T, and U) accomplish the break using a RST 7 instruction, which means that the tested program cannot use this interrupt location. Further, the trace mode always runs the tested program with interrupts enabled, which may cause problems if asynchronous interrupts are received during tracing.

Note also that the operator should use the rubout key to get control back to DDT during trace, rather than executing a RST 7, in order to ensure that the trace for the current instruction is completed before interruption.

11. The U (Untrace) Command. The U command is identical to the T command except that intermediate program steps are not displayed. The untrace mode allows from 1 to 65535 (FFFFFH) steps to be executed in monitored mode, and is used principally to retain control of an executing program while it reaches steady state conditions. All conditions of the T command apply to the U command.

12. The X (Examine) Command. The X command allows selective display and alteration of the current CPU state for the program under test. The forms are

X
Xr

where r is one of the 8080 CPU registers

<table>
<thead>
<tr>
<th>C</th>
<th>Carry Flag</th>
<th>(0/1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>Zero Flag</td>
<td>(0/1)</td>
</tr>
</tbody>
</table>
M  Minus Flag  (0/1)
E  Even Parity Flag (0/1)
I  Interdigit Carry (0/1)
A  Accumulator (0-FF)
B  BC register pair (0-FFFF)
D  DE register pair (0-FFFF)
H  HL register pair (0-FFFF)
S  Stack Pointer (0-FFFF)
P  Program Counter (0-FFFF)

In the first case, the CPU register state is displayed in the format

CfZfMfEfIf A=bb B=dddd D=dddd H=dddd S=dddd P=dddd inst

where f is a 0 or 1 flag value, bb is a byte value, and dddd is a double byte
quantity corresponding to the register pair. The "inst" field contains the
disassembled instruction which occurs at the location addressed by the CPU
state's program counter.

The second form allows display and optional alteration of register values,
where r is one of the registers given above (C, Z, M, E, I, A, B, D, H, S, or
P). In each case, the flag or register value is first displayed at the
console. The DDT program then accepts input from the console. If a carriage
return is typed, then the flag or register value is not altered. If a value
in the proper range is typed, then the flag or register value is altered.
Note that BC, DE, and HL are displayed as register pairs. Thus, the operator
types the entire register pair when B, C, or the BC pair is altered.

III. IMPLEMENTATION NOTES.

The organization of DDT allows certain non-essential portions to be
overlaid in order to gain a larger transient program area for debugging large
programs. The DDT program consists of two parts: the DDT nucleus and the
assembler/disassembler module. The DDT nucleus is loaded over the Console
Command Processor, and, although loaded with the DDT nucleus, the
assembler/disassembler is overlayable unless used to assemble or disassemble.

In particular, the BDOS address at location 6H (address field of the JMP
instruction at location 5H) is modified by DDT to address the base location of
the DDT nucleus which, in turn, contains a JMP instruction to the BDOS. Thus,
programs which use this address field to size memory see the logical end of
memory at the base of the DDT nucleus rather than the base of the BDOS.

The assembler/disassembler module resides directly below the DDT nucleus
in the transient program area. If the A, L, T, or X commands are used during
the debugging process then the DDT program again alters the address field at
6H to include this module, thus further reducing the logical end of memory.
If a program loads beyond the beginning of the assembler/disassembler module,
the A and L commands are lost (their use produces a "?" in response), and the
trace and display (T and X) commands list the "inst" field of the display in hexadecimal, rather than as a decoded instruction.

IV. AN EXAMPLE.

The following example shows an edit, assemble, and debug for a simple program which reads a set of data values and determines the largest value in the set. The largest value is taken from the vector, and stored into "LARGE" at the termination of the program.

```
ED SCAN.ASM

* ORG 100H

MVI B, LEN
MVI C, 0
LOOP: P_D_O_L
LXI H, VECT
STX C
SUB C
JNC NFOUND
MVI A, H
MOV A, M
SUB C
JNC NFOUND
MVI A, H
MOV A, M
NFOUND: INX H
DCR B
JNZ LOOP
END OF SCAN, STORE C
MOV A, C
STA LARGE
JMP B

TEST DATA
DB 2, 0, 4, 3, 5, 6, 1, 5,
LEN EQU $-VECT
LARGE: DS 1

END

BOP

ORG 100H
MVI B, LEN
MVI C, 0
LXI H, VECT
LOOP: MOV A, M
SUB C
JNC NFOUND
NEW LARGEST VALUE, STORE IT TO C
MVI A, H
MOV A, M
NFOUND: INX H
DCR B
JNZ LOOP
```

Create Source Program: underlined characters typed by programmer. "_" represents carriage return.
; END OF SCAN, STORE C
MOV A,C ;GET LARGEST VALUE
STA LARGE
JMP 0 ;REBOOT

; TEST DATA
VECT: DB 2,0,4,3,5,6,1,5
LEN EQU $-VECT ;LENGTH
LARGE: DS 1 ;LARGEST VALUE ON EXIT
END

*£a ↓ End of Edit

ASM SCAN Start Assembler
CP/M ASSEMBLER - VER 1.0

$2H USE FACTOR Assembly Complete - Look at Program Listing
END OF ASSEMBLY

TYPE SCAN.PRN

Code Address Machine Code Source Program
0100 0608 ORG 100H ;START OF TRANSIENT AREA
0102 0E00 MV1 B, LEN ;LENGTH OF VECTOR TO SCAN
0104 211901 MV1 C, 0 ;LARGEST VALUE SO FAR
0106 7E LOOP: LKI H, VECT ;BASE OF VECTOR
0108 91 SUB C ;LARGER VALUE IN C?
010A D20D01 JNC NFOUND ;JUMP IF LARGER VALUE NOT FOUND
010C 4F ;NEW LARGEST VALUE, STORE IT TO C
010E 23 MV1 C, A
0110 05 INX H ;TO NEXT ELEMENT
0112 05 DCR B ;MORE TO SCAN?
0114 C20701 JNZ LOOP ;FOR ANOTHER

; END OF SCAN, STORE C
MOV A, C ;GET LARGEST VALUE
STA LARGE
JMP 0 ;REBOOT

TEST DATA
VECT: DB 2,0,4,3,5,6,1,5
LEN EQU $-VECT ;LENGTH
LARGE: DS 1 ;LARGEST VALUE ON EXIT
END
DDT SCAN.HEX

Start Debugger using hex format machine code

16K DDT VER 1.0
NEXT PC
0121 0000

-X, last load address +1

C620000010 A=00 B=0000 D=0000 H=0000 S=0100 P=0000 OUT 7F PC=0

-XP, examine registers before debug run

P=0000 100 change PC to 100

-X, look at registers again

C620000010 A=00 B=0000 D=0000 H=0000 S=0160 P=0100 MVI B, 08

-L100

0100 MVI B, 08
0102 MVI C, 00
0104 LXI H, 0119
0107 MOV A, M
0108 SUB C
0109 JNC 018D
010C MOV C, A
010D INX H
010E DCR B
010F JNZ 0107
0112 MOV A, C

Disassembled Machine
Code at 100H

(see source listing
for comparison)

-L,

0113 STA 0121
0116 JMP 0000
0119 STAX B
011A NOP
011B INR B
011C INX B
011D DCR B
011E MVI B, 01
0120 DCR B
0121 LXI D, 2200
0124 LXI H, 0200

A little more
machine code

(note that program
ends at location 116
with a JMP to 0000)

-A116 enter inline assembly mode to change the JMP to 0000 into a RST 7, which
will cause the program under test to return to DDT if 116H
is ever executed.

0116 RST 7

(Single carriage return stops assemble mode)

-L113 list code at 113H to check that RST 7 was properly inserted

0113 STA 0121
0116 RST 07

In place of JMP

12
Look at registers:

```
0000 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI B,08
```

Execute Program for one step. Initial CPU state before j is executed:

```
0000 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI B,08+0102
```

Trace one step again (note 08H in B)

```
0000 A=00 B=0000 D=0000 H=0000 C=0100 P=0102 MVI C,00+0104
```

Trace again (Register C is cleared)

```
0000 A=00 B=0000 D=0000 H=0000 S=0100 P=0104 LXI M,0119+0107
```

Trace three steps:

```
0000 A=00 B=0000 D=0000 H=0119 S=0100 P=0107 MOV A,M
```
```
0000 A=02 B=0000 D=0000 H=0119 S=0100 P=0108 SUB C
```
```
0000 A=02 B=0000 D=0000 H=0119 S=0100 P=0109 JNC 010D+010D
```

Display memory starting at 0119:

<table>
<thead>
<tr>
<th>Program data</th>
<th>Automatic breakpoint at 010D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0119</td>
<td>02 00 04 03 05 06 01</td>
</tr>
<tr>
<td>0120</td>
<td>05 11 00 22 21 00 02 7E EB 77 13 23 EB 0B 7B 01</td>
</tr>
<tr>
<td>0121</td>
<td>0B 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0150</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0160</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0170</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0180</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0190</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>01A0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>01B0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>01C0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>01D0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

Current CPU State:
```
0000 A=02 B=0000 D=0000 H=0119 S=0100 P=010D INX H
```

Trace 5 steps from current CPU State:
```
0000 A=02 B=0000 D=0000 H=0119 S=0100 P=010D INX H
```
```
0000 A=02 B=0000 D=0000 H=011A S=0100 P=010E DCR B
```
```
0000 A=02 B=0700 D=0000 H=011A S=0100 P=010F JNZ 0107 MOV A,M
```
```
0000 A=00 B=0700 D=0000 H=011A S=0100 P=0107 MOV A,M
```
```
0000 A=00 B=0700 D=0000 H=011A S=0100 P=0108 SUB C+0109
```

Trace without listing intermediate states:
```
0000 A=00 B=0700 D=0000 H=011A S=0100 P=0109 JNC 010D+0108
```

CPU State at end of US:
```
0000 A=04 B=0600 D=0000 H=011B S=0100 P=0108 SUB C
```

13
Run Program from current PC until completion (in real-time)

breakpoint at 116H caused by executing RST 7 in machine code

CPU state at end of Program

COZIME111 A=00 B=0000 D=0000 H=0121 S=0100 P=0116 RST 07

examine and change program counter

P=0116 100

COZIME111 A=00 B=0000 D=0000 H=0121 S=0100 P=0100 X05 B,GS

Trace 10 (hexadecimal) steps

COZIME111 A=00 B=0000 D=0000 H=0121 S=0100 P=0100 C05 M10 B,08

COZIME111 A=00 B=0000 D=0000 H=0121 S=0100 P=0102 X05 C,00

COZIME111 A=00 B=0000 D=0000 H=0121 S=0100 P=0104 X05 L,0119

COZIME111 A=00 B=0000 D=0000 H=0119 S=0100 P=0107 X05 A,M

COZIME111 A=02 B=0000 D=0000 H=0119 S=0100 P=0103 X05 C

COZIME111 A=02 B=0000 D=0000 H=0119 S=0100 P=0109 X05 J,010D

COZIME111 A=02 B=0000 D=0000 H=0119 S=0100 P=010D X05 I,010B

COZIME111 A=02 B=0000 D=0000 H=0114 S=0100 P=010E X05 D,0107

COZIME111 A=02 B=0700 D=0000 H=011A S=0100 P=010D X05 F,0107

COZIME111 A=02 B=0700 D=0000 H=011A S=0100 P=0107 X05 M,010B

COZIME111 A=00 B=0700 D=0000 H=011A S=0100 P=0108 X05 C

COZIME111 A=00 B=0700 D=0000 H=011A S=0100 P=010D X05 I,010D

COZIME111 A=00 B=0700 D=0000 H=011A S=0100 P=010D X05 I,010D

COZIME111 A=00 B=0600 D=0000 H=011B S=0100 P=010E X05 D,010B

COZIME111 A=00 B=0600 D=0000 H=011B S=0100 P=010D X05 J,010D

COZIME111 A=00 B=0600 D=0000 H=011B S=0100 P=010D X05 I,010D

Insert a "hot patch" into the machine code to change the JNC to JC

Stop DDT so that a version of the patched program can be saved

SAVE SCAN.COM

Program resides on first page, so save 1 page.

A>DDT SCAN.COM

Restart DDT with the saved memory image to continue testing

16K DDT VER 0.0
NEXT PC
0200 0100

List some code

0100 MVI B,08
0102 MVI C,00
0104 LXI H,0119
0107 MOV A,M
0108 SUB C
0109 JC 010D

Previous Patch is Present in X.COM
010C MOV C, A
010D INX H
010E DCR B
010F JNZ 0107
0112 MOV A, C

P=0100

-T-

--- Trace to see how patched version operates ---

C020B8E010 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI B, 00
C020B8E010 A=00 B=0000 D=0000 H=0000 S=0100 P=0102 MVI C, 00
C020B8E010 A=00 B=0000 D=0000 H=0000 S=0100 P=0104 LXI H, 0119
C020B8E010 A=00 B=0000 D=0000 H=0119 S=0100 P=0107 MOV A, M
C020B8E010 A=02 B=0000 D=0000 H=0119 S=0100 P=0108 SUB C
C020B8E011 A=02 B=0000 D=0000 H=0119 S=0100 P=0109 JC 010D
C020B8E011 A=02 B=0000 D=0000 H=0119 S=0100 P=010C MOV C, A
C020B8E011 A=02 B=0000 D=0000 H=0119 S=0100 P=010D INX H
C020B8E011 A=02 B=0000 D=0000 H=0119 S=0100 P=010E DCR B
C020B8E011 A=02 B=0000 D=0000 H=011A S=0100 P=010F JNZ 0107
C020B8E011 A=02 B=0000 D=0000 H=011A S=0100 P=0107 MOV A, M
C020B8E011 A=02 B=0000 D=0000 H=011A S=0100 P=0108 SUB C
C120B8E010 A=FE B=0702 D=0000 H=011A S=0100 P=0109 JC 010D
C120B8E010 A=FE B=0702 D=0000 H=011A S=0100 P=010D INX H
C120B8E010 A=FE B=0702 D=0000 H=011B S=0100 P=010E DCR B
C120B8E011 A=FE B=0602 D=0000 H=011B S=0100 P=010F JNZ 0107*0107

--- breakpoint after 16 steps ---

C120B8E011 A=FE B=0602 D=0000 H=011B S=0100 P=0107 MOV A, M

-G- 108H  Run from current PC and breakpoint at 108H

*0108

--- next data item ---

C120B8E011 A=04 B=0602 D=0000 H=011B S=0100 P=0108 SUB C

-T-

--- Single Step for a few cycles ---

C120B8E011 A=04 B=0602 D=0000 H=011B S=0100 P=0108 SUB C*0109

-T-

C020B8E011 A=02 B=0602 D=0000 H=011B S=0100 P=0109 JC 010D*010C

-T-

C020B8E011 A=02 B=0602 D=0000 H=011B S=0100 P=010C MOV C, A

-G-

--- Run to completion ---

*0116

-T-

C020B8E011 A=03 B=0000 D=0000 H=0121 S=0100 P=0116 RST 07

-S121- look at the value of "LARGE"

0121 03, Wrong Value!
0122 00,
0123 22,
0124 21,
0125 00,
0126 02, \textit{End of the S Command}
0127 7E,
\textendnote{L100}

\begin{align*}
0100 & \text{ MVI } B, 08 \\
0102 & \text{ MVI } C, 00 \\
0104 & \text{ LXI } H, 0119 \\
0107 & \text{ MOY } A, M \\
0108 & \text{ SUB } C \\
0109 & \text{ JC } 010D \\
010C & \text{ MOY } C, A \\
010D & \text{ INX } H \\
010E & \text{ DCR } B \\
010F & \text{ JNZ } 0107 \\
0112 & \text{ MOY } A, C \\
\textendnote{L100'}
0113 & \text{ STA } 0121 \\
0116 & \text{ RST } 07 \\
0117 & \text{ NOP } \\
0118 & \text{ NOP } \\
0119 & \text{ STAX } B \\
011A & \text{ NOP } \\
011B & \text{ INR } B \\
011C & \text{ INX } B \\
011D & \text{ DCR } B \\
011E & \text{ MVI } B, 01 \\
0120 & \text{ DCR } B \\
\textendnote{XP}
\end{align*}

\textendnote{P=0116 100, Reset the PC}

\textendnote{-T_1 Single step, and watch data values}

\textendnote{COZIMOE111} A=03 B=0003 D=0000 H=0121 S=0100 P=0100 MVI B, 03+0102

\textendnote{-T_2}

\textendnote{COZIMOE111} A=03 B=0003 D=0000 H=0121 S=0100 P=0102 MVI C, 00+0104

\textendnote{-T_3}

\textendnote{COZIMOE111} A=03 B=0000 D=0000 H=0121 S=0100 P=0104 LXI H, 0119*0107

\textendnote{-T_4}

\textendnote{COZIMOE111} A=03 B=0000 D=0000 H=0119 S=0100 P=0107 MOY A, M+0108
-I-

-\text{First data item brought to A}

021MBE01I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0108 SUB C=0109

-\text{First data item moved to C correctly}

020MBE01I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0109 JC 10D=010C

-\text{Second data item brought to A}

020MBE01I1 A=02 B=0802 D=0000 H=011A S=0100 P=010E DCR E=010F

-\text{Subtract destroys data value which was loaded!!!}

020MBE01I1 A=02 B=0702 D=0000 H=011A S=0100 P=010F JNZ 107=0107

020MBE01I1 A=02 B=0702 D=0000 H=011A S=0100 P=010F JNZ 107=0107

020MBE01I1 A=02 B=0702 D=0000 H=011A S=0100 P=010E DCR E=010F

-\text{This should have been a CMP so that register A would not be destroyed.}

0100 MVI B, 08
0102 MVI C, 00
0104 LXI H, 0119
0107 MOV A, M
0108 SUB C
0109 JC 010D
010C MOV C, A
010D INX H
010E DCR B
010F JNZ 0107
0112 MOV A, C

-\text{Not patch at 108H changes SUB to CMP}

0108 CMP C

0109

-\text{Stop DDT for SAVE}
SAVE 1 SCAN.COM

A>DDT SCAN.COM

16K DDT VER 1.0
NEXT PC
0200 0100
-XP

P=0100

-L116

0116 RST 07
0117 NOP
0118 NOP
0119 STAX B
011A NOP

Long typeout aborted with restart

-G.116 Run from LOOH to completion

*0116

-XC

Look at Carry (accidental type)

C12

Look at CPU state

C121 M=111 A=06 B=000 D=0000 H=0121 S=0100 P=0116 RST 07
-S121

Look at "Large" - it appears to be correct.

0121 06

0122 00

0123 22 *

-G0 Stop DDT

ED SCAN.COM*ASM

Re-edit the source program, and make both changes

*NSUB

*OLT

SUB C

LARGER VALUE IN C?

SSUB CMP Z OLT

LARGER VALUE IN C?

JNC NFOUND ;JUMP IF LARGER VALUE NOT FOUND

SHC CMPLT

NFOUND ;JUMP IF LARGER VALUE NOT FOUND
Re-assemble, selecting source from disk A
Hex-to-disk A
Print to Z (selects no print file)

.asm scan.aaz

CPM ASSEMBLER - VER 1.0

0122
002H USE FACTOR
END OF ASSEMBLY

.ddt scan.hex

Re-run debugger to check changes

16K DDT VER 1.0
NEXT PC
0121 0000
-L116-

0116 JMP 0000 check to ensure end is still at 116H
0119 STA % B
011A NOP
011B INR B
-(rubout)

-L100,116- Go from beginning with breakpoint at end

*0116 breakpoint reached
-D121-
Look at "LARGE" - correct value computed

0121 60 00 22 21 00 02 7E 88 77 13 23 8E 06 78 B1 .
0130 22 27 01 C3 03 29 00 00 00 00 00 00 00 00 00 00 00 00 00
0140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
-(rubout) abouts long typeout

-g2- Stop DDT, debug session complete
CP/M ASSEMBLER (ASM)

USER'S GUIDE
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. PROGRAM FORMAT</td>
<td>2</td>
</tr>
<tr>
<td>3. FORMING THE OPERAND</td>
<td>4</td>
</tr>
<tr>
<td>3.1. Labels</td>
<td>4</td>
</tr>
<tr>
<td>3.2. Numeric Constants</td>
<td>4</td>
</tr>
<tr>
<td>3.3. Reserved Words</td>
<td>5</td>
</tr>
<tr>
<td>3.4. String Constants</td>
<td>6</td>
</tr>
<tr>
<td>3.5. Arithmetic and Logical Operators</td>
<td>6</td>
</tr>
<tr>
<td>3.6. Precedence of Operators</td>
<td>7</td>
</tr>
<tr>
<td>4. ASSEMBLER DIRECTIVES</td>
<td>8</td>
</tr>
<tr>
<td>4.1. The ORG Directive</td>
<td>8</td>
</tr>
<tr>
<td>4.2. The END Directive</td>
<td>9</td>
</tr>
<tr>
<td>4.3. The EQU Directive</td>
<td>9</td>
</tr>
<tr>
<td>4.4. The SET Directive</td>
<td>10</td>
</tr>
<tr>
<td>4.5. The IF and ENDF Directives</td>
<td>10</td>
</tr>
<tr>
<td>4.6. The DB Directive</td>
<td>11</td>
</tr>
<tr>
<td>4.7. The DW Directive</td>
<td>12</td>
</tr>
<tr>
<td>5. OPERATION CODES</td>
<td>12</td>
</tr>
<tr>
<td>5.1. Jumps, Calls, and Returns</td>
<td>13</td>
</tr>
<tr>
<td>5.2. Immediate Operand Instructions</td>
<td>14</td>
</tr>
<tr>
<td>5.3. Increment and Decrement Instructions</td>
<td>14</td>
</tr>
<tr>
<td>5.4. Data Movement Instructions</td>
<td>14</td>
</tr>
<tr>
<td>5.5. Arithmetic Logic Unit Operations</td>
<td>15</td>
</tr>
<tr>
<td>5.6. Control Instructions</td>
<td>16</td>
</tr>
<tr>
<td>6. ERROR MESSAGES</td>
<td>16</td>
</tr>
<tr>
<td>7. A SAMPLE SESSION</td>
<td>17</td>
</tr>
</tbody>
</table>
CP/M Assembler User’s Guide

1. INTRODUCTION.

The CP/M assembler reads assembly language source files from the diskette, and produces 8080 machine language in Intel hex format. The CP/M assembler is initiated by typing

ASM filename

or

ASM filename.parms

In both cases, the assembler assumes there is a file on the diskette with the name

filename.ASM

which contains an 8080 assembly language source file. The first and second forms shown above differ only in that the second form allows parameters to be passed to the assembler to control source file access and hex and print file destinations.

In either case, the CP/M assembler loads, and prints the message

CP/M ASSEMBLER VER n.n

where n.n is the current version number. In the case of the first command, the assembler reads the source file with assumed file type “ASM” and creates two output files

filename.HEX

and

filename.PRN

the “HEX” file contains the machine code corresponding to the original program in Intel hex format, and the “PRN” file contains an annotated listing showing generated machine code, error flags, and source lines. If errors occur during translation, they will be listed in the PRN file as well as at the console.

The second command form can be used to redirect input and output files from their defaults. In this case, the “parms” portion of the command is a three letter group which specifies the origin of the source file, the destination of the hex file, and the destination of the print file. The form is

filename.plp2p3

where pl, p2, and p3 are single letters

pl: A,B,...,Y designates the disk name which contains
p2: A, B, ..., Y designate the disk name which will receive the hex file
    Z skips the generation of the hex file
p3: A, B, ..., Y designate the disk name which will receive the print file
    X places the listing at the console
    Z skips generation of the print file

Thus, the command

ASM X.AAA

indicates that the source file (X.ASM) is to be taken from disk A, and that the hex (X.HEX) and print (X.PRN) files are to be created also on disk A. This form of the command is implied if the assembler is run from disk A. That is, given that the operator is currently addressing disk A, the above command is equivalent to

ASM X

The command

ASM X.ABX

indicates that the source file is to be taken from disk A, the hex file is placed on disk B, and the listing file is to be sent to the console. The command

ASM X.BZZ

takes the source file from disk B, and skips the generation of the hex and print files (this command is useful for fast execution of the assembler to check program syntax).

The source program format is compatible with both the Intel 8080 assembler (macros are not currently implemented in the CP/M assembler, however), as well as the Processor Technology Software Package #1 assembler. That is, the CP/M assembler accepts source programs written in either format. There are certain extensions in the CP/M assembler which make it somewhat easier to use. These extensions are described below.

2. PROGRAM FORMAT.

An assembly language program acceptable as input to the assembler consists of a sequence of statements of the form

    line# label operation operand ;comment

where any or all of the fields may be present in a particular instance. Each
-embly language statement is terminated with a carriage return and line feed (the line feed is inserted automatically by the ED program), or with the character "!" which is a treated as an end-of-line by the assembler (thus, multiple assembly language statements can be written on the same physical line if separated by exclaim symbols).

The line# is an optional decimal integer value representing the source program line number, which is allowed on any source line to maintain compatibility with the Processor Technology format. In general, these line numbers will be inserted if a line-oriented editor is used to construct the original program, and thus ASM ignores this field if present.

The label field takes the form

    identifier
or

    identifier:

and is optional, except where noted in particular statement types. The identifier is a sequence of alphanumeric characters (alphabetics and numbers), where the first character is alphabetic. Identifiers can be freely used by the programmer to label elements such as program steps and assembler directives, but cannot exceed 16 characters in length. All characters are significant in an identifier, except for the embedded dollar symbol ($) which can be used to improve readability of the name. Further, all lower case alphabetics become are treated as if they were upper case. Note that the ":" following the identifier in a label is optional (to maintain compatibility between Intel and Processor Technology). Thus, the following are all valid instances of labels

    x   xy   long$Name
    x:  yx1:  longer$Named$data:
    xlY2  xlX2  x234$5678$9012$3456:

The operation field contains either an assembler directive, or pseudo operation, or an 8080 machine operation code. The pseudo operations and machine operation codes are described below.

The operand field of the statement, in general, contains an expression formed out of constants and labels, along with arithmetic and logical operations on these elements. Again, the complete details of properly formed expressions are given below.

The comment field contains arbitrary characters following the ";" symbol until the next real or logical end-of-line. These characters are read, listed, and otherwise ignored by the assembler. In order to maintain compatibility with the Processor Technology assembler, the CP/M assembler also treat statements which begin with a "*" in column one as comment statements, which are listed and ignored in the assembly process. Note that the Processor
Technology assembler has the side effect in its operation of ignoring the characters after the operand field has been scanned. This causes an ambiguous situation when attempting to be compatible with Intel's language, since arbitrary expressions are allowed in this case. Hence, programs which use this side effect to introduce comments, must be edited to place a ";" before these fields in order to assemble correctly.

The assembly language program is formulated as a sequence of statements of the above form, terminated optionally by an END statement. All statements following the END are ignored by the assembler.

3. FORMING THE OPERAND.

In order to completely describe the operation codes and pseudo operations, it is necessary to first present the form of the operand field, since it is used in nearly all statements. Expressions in the operand field consist of simple operands (labels, constants, and reserved words), combined in properly formed subexpressions by arithmetic and logical operators. The expression computation is carried out by the assembler as the assembly proceeds. Each expression must produce a 16-bit value during the assembly. Further, the number of significant digits in the result must not exceed the intended use. That is, if an expression is to be used in a byte move immediate instruction, then the most significant 8 bits of the expression must be zero. The restrictions on the expression significance is given with the individual instructions.

3.1. Labels.

As discussed above, a label is an identifier which occurs on a particular statement. In general, the label is given a value determined by the type of statement which it precedes. If the label occurs on a statement which generates machine code or reserves memory space (e.g., a MOV instruction, or a DS pseudo operation), then the label is given the value of the program address which it labels. If the label precedes an EQU or SET, then the label is given the value which results from evaluating the operand field. Except for the SET statement, an identifier can label only one statement.

When a label appears in the operand field, its value is substituted by the assembler. This value can then be combined with other operands and operators to form the operand field for a particular instruction.

3.2. Numeric Constants.

A numeric constant is a 16-bit value in one of several bases. The base, called the radix of the constant, is denoted by a trailing radix indicator. The radix indicators are

B binary constant (base 2)
O octal constant (base 8)
Q, octal constant (base 8)
D, decimal constant (base 10)
H, hexadecimal constant (base 16)

Q is an alternate radix indicator for octal numbers since the letter O is easily confused with the digit 0. Any numeric constant which does not terminate with a radix indicator is assumed to be a decimal constant.

A constant is thus composed as a sequence of digits, followed by an optional radix indicator, where the digits are in the appropriate range for the radix. That is binary constants must be composed of 0 and 1 digits, octal constants can contain digits in the range 0 - 7, while decimal constants contain decimal digits. Hexadecimal constants contain decimal digits as well as hexadecimal digits A (10D), B (11D), C (12D), D (13D), E (14D), and F (15D). Note that the leading digit of a hexadecimal constant must be a decimal digit in order to avoid confusing a hexadecimal constant with an identifier (a leading 0 will always suffice). A constant composed in this manner must evaluate to a binary number which can be contained within a 16-bit counter, otherwise it is truncated on the right by the assembler. Similar to identifiers, imbedded "$" are allowed within constants to improve their readability. Finally, the radix indicator is translated to upper case if a lower case letter is encountered. The following are all valid instances of numeric constants

1234   1234D   1100B   1111$0000$1111$0000B
1234H   0FFEH   33770   33$77$22Q
33770   0Fe3h   1234d   0xffffh

3.3. Reserved Words.

There are several reserved character sequences which have predefined meanings in the operand field of a statement. The names of 8080 registers are given below, which, when encountered, produce the value shown to the right

A    7
B    0
C    1
D    2
E    3
H    4
L    5
M    6
SP   6
PSW  6

(again, lower case names have the same values as their upper case equivalents). Machine instructions can also be used in the operand field, and evaluate to their internal codes. In the case of instructions which require operands, where the specific operand becomes a part of the binary bit pattern
of the instruction (e.g., MOV A, B), the value of the instruction (in this case MOV) is the bit pattern of the instruction with zeroes in the optional fields (e.g., MOV produces 40H).

When the symbol "$" occurs in the operand field (not imbedded within identifiers and numeric constants) its value becomes the address of the next instruction to generate, not including the instruction contained within the current logical line.

3.4. String Constants.

String constants represent sequences of ASCII characters, and are represented by enclosing the characters within apostrophe symbols ('). All strings must be fully contained within the current physical line (thus allowing "!" symbols within strings), and must not exceed 64 characters in length. The apostrophe character itself can be included within a string by representing it as a double apostrophe (the two keystrokes ""), which becomes a single apostrophe when read by the assembler. In most cases, the string length is restricted to either one or two characters (the DB pseudo operation is an exception), in which case the string becomes an 8 or 16 bit value, respectively. Two character strings become a 16-bit constant, with the second character as the low order byte, and the first character as the high order byte.

The value of a character is its corresponding ASCII code. There is no case translation within strings, and thus both upper and lower case characters can be represented. Note however, that only graphic (printing) ASCII characters are allowed within strings. Valid strings are

'A'  'AB'  'ab'  'C'
'a'  'a'

'Walla Walla Wash.'
'She said 'Hello' to me.'
'I said "Hello" to her.'

3.5. Arithmetic and Logical Operators.

The operands described above can be combined in normal algebraic notation using any combination of properly formed operands, operators, and parenthesized expressions. The operators recognized in the operand field are:

\[
\begin{align*}
\text{+ b} & \quad \text{unsigned arithmetic sum of a and b} \\
\text{- b} & \quad \text{unsigned arithmetic difference between a and b} \\
\text{+ b} & \quad \text{unary plus (produces b)} \\
\text{- b} & \quad \text{unary minus (identical to 0 - b)} \\
\text{a \times b} & \quad \text{unsigned magnitude multiplication of a and b} \\
\text{a \div b} & \quad \text{unsigned magnitude division of a by b} \\
\text{a MOD b} & \quad \text{remainder after a \div b} \\
\text{NOT b} & \quad \text{logical inverse of b (all 0's become 1's, 1's become 0's)}, \text{where b is considered a 16-bit value}
\end{align*}
\]
a AND b  bit-by-bit logical and of a and b
a OR b   bit-by-bit logical or of a and b
a XOR b  bit-by-bit logical exclusive or of a and b
a SHL b  the value which results from shifting a to the
         left by an amount b, with zero fill
a SHR b  the value which results from shifting a to the
         right by an amount b, with zero fill

In each case, a and b represent simple operands (labels, numeric
constants, reserved words, and one or two character strings), or fully
enclosed parenthesized subexpressions such as

10+20    10h+37Q    L1 /3 (L2+4) SHR 3
(¨a¨ and 5fh) + ¨0¨  (¨B¨+B) OR (PSW+M)
(1+(2+c)) SHR (A−(B+1))

Note that all computations are performed at assembly time as 16-bit unsigned
operations. Thus, −1 is computed as 0−1 which results in the value 0ffffh
(i.e., all 1’s). The resulting expression must fit the operation code in
which it is used. If, for example, the expression is used in a ADI (add
immediate) instruction, then the high order eight bits of the expression
must be zero. As a result, the operation “ADI −1” produces an error message (−1
becomes 0ffffh which cannot be represented as an 8 bit value), while “ADI (−1)
AND 0FFFFH” is accepted by the assembler since the “AND” operation zeroes the
high order bits of the expression.


As a convenience to the programmer, ASM assumes that operators have a
relative precedence of application which allows the programmer to write
expressions without nested levels of parentheses. The resulting expression
has assumed parentheses which are defined by the relative precedence.
The order of application of operators in unparenthesize expressions is listed
below. Operators listed first have highest precedence (they are applied first
in an unparenthesized expression), while operators listed last have lowest
precedence. Operators listed on the same line have equal precedence, and are
applied from left to right as they are encountered in an expression

*/MODSHLSHR
  −+
  NOT
  AND
  ORXOR

Thus, the expressions shown to the left below are interpreted by the assembler
as the fully parenthesize expressions shown to the right below

a * b + c          (a * b) + c
a + b * c          a + (b * c)
a MOD b * c SHR d  ((a MOD b) * c) SHR d
a OR b AND NOT c + d SHL e  a OR (b AND (NOT (c + (d SHL e))))

Balanced parenthesized subexpressions can always be used to override the
assumed parentheses, and thus the last expression above could be rewritten to
force application of operators in a different order as

(a OR b) AND (NOT c) + d SHL e

resulting in the assumed parentheses

(a OR b) AND ((NOT c) + (d SHL e))

Note that an unparenthesized expression is well-formed only if the expression
which results from inserting the assumed parentheses is well-formed.

4. ASSEMBLER DIRECTIVES.

Assembler directives are used to set labels to specific values during the
assembly, perform conditional assembly, define storage areas, and specify
starting addresses in the program. Each assembler directive is denoted by a
"pseudo operation" which appears in the operation field of the line. The
acceptable pseudo operations are

ORG  set the program or data origin
END  end program, optional start address
EQU  numeric "equate"
SET  numeric "set"
IF   begin conditional assembly
ENDIF end of conditional assembly
DB   define data bytes
DW   define data words
DS   define data storage area

The individual pseudo operations are detailed below

4.1. The ORG directive.

The ORG statement takes the form

label  ORG  expression

where "label" is an optional program label, and expression is a 16-bit
expression, consisting of operands which are defined previous to the ORG
statement. The assembler begins machine code generation at the location
specified in the expression. There can be any number of ORG statements within
a particular program, and there are no checks to ensure that the programmer is
not defining overlapping memory areas. Note that most programs written for
the CP/M system begin with an ORG statement of the form

ORG 100H
which causes machine code generation to begin at the base of the CP/M transient program area. If a label is specified in the ORG statement, then the label is given the value of the expression (this label can then be used in the operand field of other statements to represent this expression).

4.2. The END directive.

The END statement is optional in an assembly language program, but if it is present it must be the last statement (all subsequent statements are ignored in the assembly). The two forms of the END directive are

```
label END
label END expression
```

where the label is again optional. If the first form is used, the assembly process stops, and the default starting address of the program is taken as 0000. Otherwise, the expression is evaluated, and becomes the program starting address (this starting address is included in the last record of the Intel formatted machine code "hex" file which results from the assembly). Thus, most CP/M assembly language programs end with the statement

```
END 100H
```

resulting in the default starting address of 100H (beginning of the transient program area).

4.3. The EQU directive.

The EQU (equate) statement is used to set up synonyms for particular numeric values, the form is

```
label EQU expression
```

where the label must be present, and must not label any other statement. The assembler evaluates the expression, and assigns this value to the identifier given in the label field. The identifier is usually a name which describes the value in a more human-oriented manner. Further, this name is used throughout the program to "parameterize" certain functions. Suppose for example, that data received from a Teletype appears on a particular input port, and data is sent to the Teletype through the next output port in sequence. The series of equate statements could be used to define these ports for a particular hardware environment

```
TTYBASE EQU 10H ;BASE PORT NUMBER FOR TTY
TTYIN EQU TTYBASE ;TTY DATA IN
TTYOUT EQU TTYBASE+1;TTY DATA OUT
```

At a later point in the program, the statements which access the Teletype could appear as
IN TTYIN ;READ TTY DATA TO REG-A
***
OUT TTYOUT ;WRITE DATA TO TTY FROM REG-A

making the program more readable than if the absolute i/o ports had been used. Further, if the hardware environment is redefined to start the Teletype communications ports at 7FH instead of 10H, the first statement need only be changed to

TTYBASE EQU 7FH ;BASE PORT NUMBER FOR TTY

and the program can be reassembled without changing any other statements.

4.4. The SET Directive.

The SET statement is similar to the EQU, taking the form

label SET expression

except that the label can occur on other SET statements within the program. The expression is evaluated and becomes the current value associated with the label. Thus, the EQU statement defines a label with a single value, while the SET statement defines a value which is valid from the current SET statement to the point where the label occurs on the next SET statement. The use of the SET is similar to the EQU statement, but is used most often in controlling conditional assembly.

4.5. The IF and ENDIF directives.

The IF and ENDIF statements define a range of assembly language statements which are to be included or excluded during the assembly process. The form is

IF expression
statement#1
statement#2
***
statement#n
ENDIF

Upon encountering the IF statement, the assembler evaluates the expression following the IF (all operands in the expression must be defined ahead of the IF statement). If the expression evaluates to a non-zero value, then statement#1 through statement#n are assembled; if the expression evaluates to zero, then the statements are listed but not assembled. Conditional assembly is often used to write a single "generic" program which includes a number of possible run-time environments, with only a few specific portions of the program selected for any particular assembly. The following program segments for example, might be part of a program which communicates with either a Teletype or a CRT console (but not both) by selecting a particular value for TTY before the assembly begins.
TRUE  EQU  0FFFFH ;DEFINE VALUE OF TRUE
FALSE EQU NOT TRUE  ;DEFINE VALUE OF FALSE
TTY   EQU  TRUE    ;TRUE IF TTY, FALSE IF CRT
TTYBASE EQU  10H    ;BASE OF TTY I/O PORTS
CRTBASE EQU  20H    ;BASE OF CRT I/O PORTS
IF TTY  ASSEMBLE RELATIVE TO TTYBASE
CONIN  EQU  TTYBASE ;CONSOLE INPUT
CONOUT EQU  TTYBASE+1 ;CONSOLE OUTPUT
ENDIF

IF NOT TTY ASSEMBLE RELATIVE TO CRTBASE
CONIN  EQU  CRTBASE ;CONSOLE INPUT
CONOUT EQU  CRTBASE+1 ;CONSOLE OUTPUT
ENDIF

*** IN  CONIN  ;READ CONSOLE DATA
*** OUT  CONOUT  ;WRITE CONSOLE DATA

In this case, the program would assemble for an environment where a Teletype
is connected, based at port 10H. The statement defining TTY could be changed to

TTY    EQU    FALSE

and, in this case, the program would assemble for a CRT based at port 20H.


The DB directive allows the programmer to define initialize storage areas
in single precision (byte) format. The statement form is

label    DB    e#1, e#2, ..., e#n

where e#1 through e#n are either expressions which evaluate to 8-bit values
(the high order eight bits must be zero), or are ASCII strings of length no
greater than 64 characters. There is no practical restriction on the number
of expressions included on a single source line. The expressions are
evaluated and placed sequentially into the machine code file following the
last program address generated by the assembler. String characters are
similarly placed into memory starting with the first character and ending with
the last character. Strings of length greater than two characters cannot be
used as operands in more complicated expressions (i.e., they must stand alone
between the commas). Note that ASCII characters are always placed in memory
with the parity bit reset (0). Further, recall that there is no translation
from lower to upper case within strings. The optional label can be used to
reference the data area throughout the remainder of the program. Examples of
valid DB statements are

```
data: DB 0,1,2,3,4,5
DB data and 0ffh,5,3770,1+2+3+4
signon: DB 'please type your name\',cr,lf,0
DB 'AB' SHR 8, 'C', 'DE' AND 7FH
```

4.7. The DW Directive.

The DW statement is similar to the DB statement except double precision (two byte) words of storage are initialized. The form is

```
label DW e#1, e#2, ..., e#n
```

where e#1 through e#n are expressions which evaluate to 16-bit results. Note that ASCII strings of length one or two characters are allowed, but strings longer than two characters disallowed. In all cases, the data storage is consistent with the 8000 processor: the least significant byte of the expression is stored forst in memory, followed by the most significant byte. Examples are

```
doub: DW 0ffefh,doub+4,signon-$,255+255
DW 'a', 5, 'ab', 'CD', 6 shl 8 or 11b
```

4.8. The DS Directive.

The DS statement is used to reserve an area of uninitialized memory, and takes the form

```
label DS expression
```

where the label is optional. The assembler begins subsequent code generation after the area reserved by the DS. Thus, the DS statement given above has exactly the same effect as the statement

```
label: EQU $ ;LABEL VALUE IS CURRENT CODE LOCATION
ORG $+expression ;MOVE PAST RESERVED AREA
```

5. OPERATION CODES.

Assembly language operation codes form the principal part of assembly language programs, and form the operation field of the instruction. In general, ASM accepts all the standard mnemonics for the Intel 8080 microcomputer, which are given in detail in the Intel manual "8080 Assembly Language Programming Manual." Labels are optional on each input line and, if included, take the value of the instruction address immediately before the instruction is issued. The individual operators are listed breifly in the
following sections for completeness, although it is understood that the Intel manuals should be referenced for exact operator details. In each case,

e3 represents a 3-bit value in the range 0-7
which can be one of the predefined registers
A, B, C, D, E, H, L, M, SP, or PSW.

e8 represents an 8-bit value in the range 0-255

e16 represents a 16-bit value in the range 0-65535

which can themselves be formed from an arbitrary combination of operands and operators. In some cases, the operands are restricted to particular values within the allowable range, such as the PUSH instruction. These cases will be noted as they are encountered.

In the sections which follow, each operation codes is listed in its most general form, along with a specific example, with a short explanation and special restrictions.

5.1. Jumps, Calls, and Returns.

The Jump, Call, and Return instructions allow several different forms which test the condition flags set in the 8080 microcomputer CPU. The forms are

<table>
<thead>
<tr>
<th>JMP  e16</th>
<th>JMP L1</th>
<th>Jump unconditionally to label</th>
</tr>
</thead>
<tbody>
<tr>
<td>JNZ e16</td>
<td>JMP L2</td>
<td>Jump on non zero condition to label</td>
</tr>
<tr>
<td>JZ e16</td>
<td>JMP 100H</td>
<td>Jump on zero condition to label</td>
</tr>
<tr>
<td>JNC e16</td>
<td>JNC L1+4</td>
<td>Jump no carry to label</td>
</tr>
<tr>
<td>JC e16</td>
<td>JC L3</td>
<td>Jump on carry to label</td>
</tr>
<tr>
<td>JFO e16</td>
<td>JFO $+8</td>
<td>Jump on parity odd to label</td>
</tr>
<tr>
<td>JPE e16</td>
<td>JPE L4</td>
<td>Jump on even parity to label</td>
</tr>
<tr>
<td>JP e16</td>
<td>JP GAMMA</td>
<td>Jump on positive result to label</td>
</tr>
<tr>
<td>JM e16</td>
<td>JM al</td>
<td>Jump on minus to label</td>
</tr>
<tr>
<td>CALL e16</td>
<td>CALL S1</td>
<td>Call subroutine unconditionally</td>
</tr>
<tr>
<td>QNZ e16</td>
<td>QNZ S2</td>
<td>Call subroutine if non zero flag</td>
</tr>
<tr>
<td>CZ e16</td>
<td>CZ 100H</td>
<td>Call subroutine on zero flag</td>
</tr>
<tr>
<td>QNC e16</td>
<td>QNC S1+4</td>
<td>Call subroutine if no carry set</td>
</tr>
<tr>
<td>CC e16</td>
<td>CC S3</td>
<td>Call subroutine if carry set</td>
</tr>
<tr>
<td>CPO e16</td>
<td>CPO $+8</td>
<td>Call subroutine if parity odd</td>
</tr>
<tr>
<td>CPE e16</td>
<td>CPE S4</td>
<td>Call subroutine if parity even</td>
</tr>
<tr>
<td>CP e16</td>
<td>CP GAMMA</td>
<td>Call subroutine if positive result</td>
</tr>
<tr>
<td>CM e16</td>
<td>CM b1$c2</td>
<td>Call subroutine if minus flag</td>
</tr>
<tr>
<td>RST e3</td>
<td>RST 0</td>
<td>Programmed &quot;restart&quot;, equivalent to CALL 8*e3, except one byte call</td>
</tr>
</tbody>
</table>
5.2. Immediate Operand Instructions.

Several instructions are available which load single or double precision registers, or single precision memory cells, with constant values, along with instructions which perform immediate arithmetic or logical operations on the accumulator (register A).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVI e3,e8</td>
<td>Move immediate data to register A, B, C, D, E, H, L, or M (memory)</td>
</tr>
<tr>
<td>ADI e8</td>
<td>Add immediate operand to A without carry</td>
</tr>
<tr>
<td>ACI e8</td>
<td>Add immediate operand to A with carry</td>
</tr>
<tr>
<td>SUI e8</td>
<td>Subtract from A without borrow (carry)</td>
</tr>
<tr>
<td>SPI e8</td>
<td>Subtract from A with borrow (carry)</td>
</tr>
<tr>
<td>ANI e8</td>
<td>Logical &quot;and&quot; A with immediate data</td>
</tr>
<tr>
<td>XRI e8</td>
<td>&quot;Exclusive or&quot; A with immediate data</td>
</tr>
<tr>
<td>ORI e8</td>
<td>Logical &quot;or&quot; A with immediate data</td>
</tr>
<tr>
<td>CPI e8</td>
<td>Compare A with immediate data (same as SUI except register A not changed)</td>
</tr>
<tr>
<td>LXI e3,e16</td>
<td>Load extended immediate to register pair (e3 must be equivalent to B,D,H, or SP)</td>
</tr>
</tbody>
</table>

5.3. Increment and Decrement Instructions.

Instructions are provided in the 8080 repertoire for incrementing or decrementing single and double precision registers. The instructions are

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>INR e3</td>
<td>Single precision increment register (e3 produces one of A, B, C, D, E, H, L, M)</td>
</tr>
<tr>
<td>DCR e3</td>
<td>Single precision decrement register (e3 produces one of A, B, C, D, E, H, L, M)</td>
</tr>
<tr>
<td>INX e3</td>
<td>Double precision increment register pair (e3 must be equivalent to B,D,H, or SP)</td>
</tr>
<tr>
<td>DCX e3</td>
<td>Double precision decrement register pair (e3 must be equivalent to B,D,H, or SP)</td>
</tr>
</tbody>
</table>

5.4. Data Movement Instructions.
Instructions which move data from memory to the CPU and from CPU to memory are given below:

- **MOV e3, e3** Moves data to leftmost element from rightmost element (e3 produces one of A, B, C, D, E, H, L, or M). MOV M, M is disallowed.
- **LDA X e3** Loads register A from computed address (e3 must produce either B or D).
- **STA X e3** Stores register A to computed address (e3 must produce either B or D).
- **LHLD e16** Loads HL direct from location e16 (double precision load to H and L).
- **SHLD e16** Stores HL direct to location e16 (double precision store from H and L to memory).
- **LDA e16** Loads register A from address e16.
- **STA X e16** Stores register A into memory at e16.
- **POP e3** Loads register pair from stack, set SP (e3 must produce one of B, D, H, or PSW).
- **PUSH e3** Stores register pair into stack, set SP (e3 must produce one of B, D, H, or PSW).
- **IN e8** Loads register A with data from port e8.
- **OUT e8** Sends data from register A to port e8.
- **XTHI** Exchanges data from top of stack with HL.
- **PCHL** Fills program counter with data from HL.
- **XCHG** Exchanges DE pair with data from HL.

5.5. Arithmetic Logic Unit Operations.

Instructions which act upon the single precision accumulator to perform arithmetic and logic operations are:

- **ADD e3** Adds register given by e3 to accumulator without carry (e3 must produce one of A, B, C, D, E, H, or L).
- **ADC e3** Adds register to A with carry, e3 as above.
- **SUB e3** Subtracts register e3 from A without carry, e3 is defined as above.
- **SBB e3** Subtracts register e3 from A with carry, e3 is defined as above.
- **ANA e3** Logical "and" reg with A, e3 as above.
- **XRA e3** "Exclusive or" with A, e3 as above.
- **ORA e3** Logical "or" with A, e3 defined as above.
- **CMP e3** Compares register with A, e3 as above.
- **DAA** Decimal adjust register A based upon last arithmetic logic unit operation.
- **CMA** Complement the bits in register A.
- **STC** Set the carry flag to 1.
CMC
Complement the carry flag

RLC
Rotate bits left, (re)set carry as a side
effect (high order A bit becomes carry)

RRC
Rotate bits right, (re)set carry as side
effect (low order A bit becomes carry)

RAL
Rotate carry/A register to left (carry is
involved in the rotate)

RAR
Rotate carry/A register to right (carry
is involved in the rotate)

DAD e3  DAD B
Double precision add register pair e3 to
HL (e3 must produce B, D, H, or SP)

5.6. Control Instructions.

The four remaining instructions are categorized as control instructions,
and are listed below

HLT
Halt the 8080 processor

DI
Disable the interrupt system

EI
Enable the interrupt system

NOP
No operation

6. ERROR MESSAGES.

When errors occur within the assembly language program, they are listed as
single character flags in the leftmost position of the source listing. The
line in error is also echoed at the console so that the source listing need
not be examined to determine if errors are present. The error codes are

D  Data error: element in data statement cannot be
    placed in the specified data area

E  Expression error: expression is ill-formed and
cannot be computed at assembly time

L  Label error: label cannot appear in this context
    (may be duplicate label)

N  Not implemented: features which will appear in
    future ASM versions (e.g., macros) are recognized,
    but flagged in this version

O  Overflow: expression is too complicated (i.e., too
    many pending operators) to computed, simplify it

P  Phase error: label does not have the same value on
    two subsequent passes through the program

16
R Register error: the value specified as a register is not compatible with the operation code

V Value error: operand encountered in expression is improperly formed

Several error messages are printed which are due to terminal error conditions

NO SOURCE FILE PRESENT The file specified in the ASM command does not exist on disk

NO DIRECTORY SPACE The disk directory is full, erase files which are not needed, and retry

SOURCE FILE NAME ERROR Improperly formed ASM file name (e.g., it is specified with "?" fields)

SOURCE FILE READ ERROR Source file cannot be read properly by the assembler, execute a TYPE to determine the point of error

OUTPUT FILE WRITE ERROR Output files cannot be written properly, most likely cause is a full disk, erase and retry

CANNOT CLOSE FILE Output file cannot be closed, check to see if disk is write protected

7. A SAMPLE SESSION.

The following session shows interaction with the assembler and debugger in the development of a simple assembly language program.
ASMT SORT\[2\] assemble SORT.ASM

CP/M ASSEMBLER - VER 1.0

015C next free address
003H USE FACTOR  % of table used 00 TO FF (hexadecimal)

END OF ASSEMBLY

DIR SORT. *2

SORT ASM source file
SORT BAK backup from last edit
SORT PRN print file (contains tab characters)
SORT HEX machine code file
A)TYPE SORT.PRN

source line

machine code location

0100 100H ORG

0100 214601+ SORT

0103 3601 LXI M, 1 ; ADDRESS SWITCH TOGGLE

0105 214701 MYI M, 1 ; SET TO 1 FOR FIRST ITERATION

0108 3600 LXI H, I ; ADDRESS INDEX

MYI M, 0 ; I = 0

COMPARE I WITH ARRAY SIZE

010A 7E MOV A, M ; A REGISTER = I

010B FE09 CPI N-1 ; CY SET IF I < (N-1)

010D D21981 JNC CONT ; CONTINUE IF I <= (N-2)

END OF ONE PASS THROUGH DATA

0110 214601 LXI H, SW ; CHECK FOR ZERO SWITCHES

0113 7E87C20001 MOV A, M! ORA A! JNZ SORT ; END OF SORT IF SW=0

0118 FF RST 7 ; GO TO THE DEBUGGER INSTEAD OF RE.

CONTINUE THIS PASS

0119 5F16802148 CONT ; ADDRESSING I, SO LOAD AV(I) INTO REGISTERS

0121 4E792346 MOV E, A! MYI D, 0! LXI H, AV! DAD D! DAD D

MOV C, M! MOV A, C! INX H! MOV B, M

LOW ORDER BYTE IN A AND C, HIGH ORDER BYTE IN B

0125 23 MOV H AND L TO ADDRESS AV(I+1)

INX H

COMPARE VALUE WITH REGS CONTAINING AV(I)

0126 965778239E SUB M! MOV D, A! MOV A, B! INX H! SBB M ; SUBTRACT

BORROW SET IF AV(I+1) > AV(I)

0128 DA3F01 JC INCI ; SKIP IF IN PROPER ORDER

012E B2CA3F01 CHECK FOR EQUAL VALUES

ORA D! JZ INCI ; SKIP IF AV(I) = AV(I+1)
MOV D.M! MOV M.B! DCX H! MOV E.M
MOV M.C! DCX H! MOV M.D! DCX H! MOV M.E

INCREMENT SWITCH COUNT
LXI H,SW! INR M

INCREMENT I
LXI H, I! INR M! JMP COMP

DATA DEFINITION SECTION
DB 0 RESERVE SPACE FOR SWITCH COUNT
DS 1 SPACE FOR INDEX
DW S.100,30,50,20,7,1000,393.100,-32767
EQU ($-AV)/2 COMPUTE N INSTEAD OF PRE
END

16K DDT VER 1.0
NEXT PC
0150 0000 default address (no address or END statement)
-XP3

P=0000 100 Change PC to 100
-UFFFF untrace for 65535 steps

C0Z0M80E10 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 LXI H,0146+0100
-T18 trace 16 steps

C0Z0M80E10 A=81 B=0000 D=0000 H=0146 S=0100 P=0100 LXI H,0146
C0Z0M80E10 A=81 B=0000 D=0000 H=0146 S=0100 P=0103 MVI M,01
C0Z0M80E10 A=81 B=0000 D=0000 H=0146 S=0100 P=0105 LXI H,0147
C0Z0M80E10 A=81 B=0000 D=0000 H=0147 S=0100 P=0108 MVI M,00
C0Z0M80E10 A=81 B=0000 D=0000 H=0147 S=0100 P=010A MOV A,M
C0Z0M80E10 A=00 B=0000 D=0000 H=0147 S=0100 P=010B CP1 09
C1Z0M81E10 A=00 B=0000 D=0000 H=0147 S=0100 P=010D JNC 0113
C1Z0M81E10 A=00 B=0000 D=0000 H=0147 S=0100 P=0110 LXI H,0146
C1Z0M81E10 A=00 B=0000 D=0000 H=0146 S=0100 P=0113 MOV A,M
C1Z0M81E10 A=01 B=0000 D=0000 H=0146 S=0100 P=0114 ORA A
C0Z0M80E10 A=01 B=0000 D=0000 H=0146 S=0100 P=0115 JNZ 0100
C0Z0M80E10 A=01 B=0000 D=0000 H=0146 S=0100 P=0100 LXI H,0146
C0Z0M80E10 A=01 B=0000 D=0000 H=0146 S=0100 P=0103 MVI M,01
C0Z0M80E10 A=01 B=0000 D=0000 H=0146 S=0100 P=0105 LXI H,0147
C0Z0M80E10 A=01 B=0000 D=0000 H=0147 S=0100 P=0108 MVI M,00
C0Z0M80E10 A=01 B=0000 D=0000 H=0147 S=0100 P=010A MOV A,M
-A10D

0110 JC 119 change to a jump on carry
0110 change to a jump on carry

stopped at 16BH
P=010B 100, reset program counter back to beginning of program
-T10, trace execution for 1004 steps

C020000E010 A=00 B=0000 D=0000 H=0147 S=0100 P=0100 LXI H, 0146
C020000E010 A=00 B=0000 D=0000 H=0146 S=0100 P=0103 MVI M, D1
C020000E010 A=00 B=0000 D=0000 H=0146 S=0100 P=0105 LXI H, 0147
C020000E010 A=00 B=0000 D=0000 H=0147 S=0100 P=0108 MVI M, 00
C020000E010 A=00 B=0000 D=0000 H=0147 S=0100 P=010A MOV A, M
C020000E010 A=00 B=0000 D=0000 H=0147 S=0100 P=010B CPI 09
C1200001E010 A=00 B=0000 D=0000 H=0147 S=0100 P=011D JC 0119
C1200001E010 A=00 B=0000 D=0000 H=0147 S=0100 P=0119 MOV E, A
C1200001E010 A=00 B=0000 D=0000 H=0147 S=0100 P=011A MVI D, 00
C1200001E010 A=00 B=0000 D=0000 H=0147 S=0100 P=011C LXI H, 0148
C1200001E010 A=00 B=0000 D=0000 H=0148 S=0100 P=011F DAD D
C0200001E010 A=00 B=0000 D=0000 H=0148 S=0100 P=0120 DAD D
C0200001E010 A=00 B=0000 D=0000 H=0148 S=0100 P=0121 MOV C, M
C0200001E010 A=00 B=0005 D=0000 H=0148 S=0100 P=0122 MOV A, C
C0200001E010 A=05 B=0005 D=0000 H=0148 S=0100 P=0123 INX H
C0200001E010 A=05 B=0005 D=0000 H=0149 S=0100 P=0124 MOV D, M=0125

0100 LXI H, 0146
0103 MVI M, 01
0105 LXI H, 0147
0108 MVI M, 00
010A MOV A, M
010B CPI 09
010C JC 0119
0110 LXI H, 0146
0113 MVI A, M
0114 ORA A
0115 JNI Z 0100

0118 RST 07
0119 MOV E, A
011A MVI D, 00
011C LXI H, 0148
-About list with rubact
-S.118, start program from current PC (0125H) and run in real time to 118H
-0127 stopped with an external interrupt 7 from front panel (program was looping indefinitely)

C0200008E010 A=38 B=0064 D=0006 H=0156 S=0100 P=0127 MOV D, A
C0200008E010 A=38 B=0064 D=3806 H=0156 S=0100 P=0128 MOV A, B
C0200008E010 A=00 B=0064 D=3806 H=0156 S=0100 P=0129 INX H
C0200008E010 A=00 B=0064 D=3806 H=0157 S=0100 P=012A SBB M=012B
-0148
0148 05 00 07 00 14 00 1E 00
0150 32 00 64 00 64 00 2C 01 EB 03 01 80 00 00 00 00 00 2 D D
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Automatic breakpoint

P=010B 100, reset program counter back to beginning of program
-T10, trace execution for 1004 steps

List some code from 1004

List more

Start program from current PC (0125H) and run in real time to 118H

Stopped with an external interrupt 7 from front panel (program was looping indefinitely)

Data is sorted, but program doesn't stop.
-G0 2 return to CP/M

DDT SORT.HEX 2 reload the memory image

16K DDT VER 1.0
NEXT PC
015C 0000
-XP

P=0000 100 2 Set PC to beginning of program
-L10D 2 list bad opcode
010D JNC 0119
0118 LXI H,0146
- abort list with rubout
-A10D 2 assemble new opcode
010D JC 119

0118
-L100 2 list starting section of program
0100 LXI H,0146
0103 MVI M,01
0105 LXI H,0147
0108 MVI M,00
- abort list with rubout
-A103 2 change "switch" initialization to 00
0103 MVI M,02

0105
- C return to CP/M with Ctrl-C (G0 works as well)

SAVE 1 SORT.COM 2 save 1 page (256 bytes, from 100H to 1FFH) on disk in case we have to reload later

A>DDT SORT.COM 2 restart DDT with
Saved memory image

16K DDT VER 1.0
NEXT PC
0200 0100 "COM" file always starts with address 100H
- G0 run the program from PC=100H

00118 programmed stop (RST 7) encountered
-D148

0148 05 00 07 00 14 00 1E 00
0150 32 00 64 00 64 00 2C 01 E8 03 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

- G0 2 return to CP/M
ED SORT.ASM  make changes to original program

*H, O3B7TJ, find next ",0"

MVI   M, 0  ; I = 0

*J up one line in text
LXI   H, I  ; ADDRESS INDEX

*J up another line
MVI   M, 1  ; SET TO 1 FOR FIRST ITERATION

*KTj kill line and type next line
LXI   H, I  ; ADDRESS INDEX

*Iw insert new line
MVI   M, 0  ; ZERO SW

*Tw
LXI   H, I  ; ADDRESS INDEX

*NJHC30Tj  JNC*Tj  ; CONTINUE IF I <= (N-2)

*2DIC30Lj  JC CONTINUE IF I <= (N-2)

*E3 source from disk A

hex to disk A

ASM SORT.AA2  ; skip prn file

CP/M ASSEMBLER - VER 1.0

015C next address to assemble

003H USE FACTOR

END OF ASSEMBLY

0DT SORT.HEX, test program changes

16K DDT VER 1.0

NEXT PC
015C 0000
-6188

*0118
-0148

0148 05 00 07 00 1E 00 data sorted

0150 32 00 64 00 64 00 2C 01 E8 03 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2 D D

0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

aborted

-66 return to CP/M - program checks OK.
EXIDY'S CB IOS USERS GUIDE
VERSION 3.0

COPYRIGHT (C) 1980
EXIDY INC.

JANUARY, 1980
COPYRIGHT

Copyright (c) 1980 by Exidy Inc. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Exidy Inc., 390 Java Drive, Sunnyvale, California 94086.

Since this manual is tutorial in nature, permission is granted to reproduce or abstract the example procedures and sample programs for the purposes of inclusion within the reader's programs.

DISCLAIMER

Exidy Inc. makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Exidy Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Exidy Inc. to notify any person of such revision or changes.

TRADEMARKS

CP/M is a registered trademark of Digital Research.
TABLE OF CONTENTS

1. INTRODUCTION........................................1

2. CONFIGURATION AND SYSTEM GENERATION.........2
   A. Hardware for the FDS and the DDS...........2
   B. System Generation.........................3
   C. Options......................................4
   D. Cache Buffer Allocation....................5
   E. Incompatibilities.........................6
   F. Sector Skew Pattern.......................7

3. FEATURES..........................................8
   A. Error Recovery..............................8
   B. CP/M Programming..........................10

4. ERROR MESSAGES.................................11
1. INTRODUCTION

The CBIOS program (Cache Basic Input/Output System) interfaces between CP/M (TM) and the Sorcerer's hardware. The term BIOS was coined by Digital Research, the creators of CP/M. Their term CBIOS stands for Customized Basic Input/Output System, in which the BIOS is customized to the user's hardware. The Exidy program is a CBIOS in this sense. However, the Exidy 'C' stresses the use of disk buffer cache techniques to improve program performance.

Portability is the most valuable attribute of CP/M. The clean separation of logical and physical I/O enables it to run on many 8080/Z80 based systems. Digital Research provides the logical I/O in CP/M's Basic Disk Operating System (BDOS). This routes all physical I/O through a BIOS vector. See the "CP/M System Alteration Guide" for a description of the BIOS vector and its functions.

Each hardware system CP/M is used on requires a separate BIOS program. The Exidy interface is tailored for the Sorcerer and the hardware the Sorcerer supports. Because the disk drive is the most complicated piece of hardware CP/M uses, this document focuses on the CBIOS disk interface for the Display Disk System (DDS) soft-sector ed 77 track disk and the Floppy Disk System (FDS) soft-sector ed 77 or 40 track disk.

For clarification, a brief explanation of physical and logical units may benefit the user in our discussion. A physical unit is the actual input/output device and its hardware recording medium. A diskette, for example, physically has 77 tracks, each of which has 16 sectors of 256-bytes. However, an interface between the physical hardware and the user may translate or break up this size into any number of combinations. This level, used by the programmer, is the logical level. That is, the very same diskette may be dealt with by the programmer as a different size from the physical size (say, 77 tracks each of which has 32 sectors of 128-bytes). The software interface makes all necessary adjustments for the input to be understood on the differing physical diskette size.

The Digital Research interface of the BDOS to the BIOS defines the logical CP/M diskette with a logical sector size of 128-bytes. The Exidy physical diskette systems, however, have a 256-byte sector. To compensate for this difference, Exidy splits each physical 256-byte sector in half to form two CP/M logical 128-byte sectors. The CBIOS is responsible for mapping 128-byte logical CP/M sectors to the proper half of a 256-byte physical sector. The physical sectors are skewed or interleaved on the diskette to minimize rotational delay. This skewing pattern is described in detail later.
The disk buffer cache improves performance by buffering reads and writes of the disk in a RAM cache storage area. When CP/M requests a 128-byte sector read from a sector not within the cache buffer, a 256-byte sector from disk must be read. Thus, the cache returns the requested sector to the user, keeping track of the other 128-byte sector half within the 256-byte cache buffer. Should a read request be made for that sector at this point, no disk I/O is required because the sector already exists in the memory cache. This same principle applies to cache buffer writing. That is, only one 256-byte physical sector I/O is written for two CP/M 128-byte logical I/O requests on the same physical sector. If there is no room in the cache buffer for a 256-byte sector during a physical I/O, the CB IOS "pages out" the least recently used sector buffer by writing it to the disk, thereby freeing a buffer slot for use by the I/O.

2. CONFIGURATION AND SYSTEM GENERATION

A. Hardware for the DDS and the FDS

The Exidy CB IOS runs on both the Display Disk System (DDS) and the Floppy Disk Subsystem (FDS). A DDS consists of a Sorcerer II Computer (with keyboard), Display Disk Unit containing a video screen, and two soft sectored Micropolis drives. The Exidy CB IOS may actually support three disk drives connected to the soft sectored disk controller. However, a controller and only two drives are supplied with the DDS. The DDS may be augmented by other peripherals such as a printer, cassettes, etc.. The Exidy CB IOS assigns the logical CP/M devices, Punch and Reader, to serial write and serial read respectively. The List device is assigned to the Sorcerer Centronics parallel printer interface.

On cold boot, CP/M on the DDS outputs the following message:

CP/M on Exidy Sorcerer for 77 Track Disk
32K CP/M VERS 1.42/3
Copyright (C) 1980 Exidy Inc.

A>

There are other Exidy CP/M's for 40 track disks. Exidy's CP/M for the DDS should state 77 tracks.
The Exidy CBIOS also runs on the Floppy Disk System (FDS). An FDS consists of a Sorcerer II Computer (with keyboard), and a Floppy Disk Subsystem containing an MPI floppy disk drive and controller. The Exidy CBIOS may actually support three disk drives connected to the soft sectored disk controller. However, a controller and only one drive are supplied with the standard Floppy Disk Subsystem. The FDS may be augmented by other peripherals such as printers, cassettes, etc. The Exidy CBIOS assigns the logical CP/M devices Punch and Reader, to serial write and serial read respectively. The List device is assigned to the Sorcerer Centronics parallel printer interface. On cold boot, CP/M on the FDS outputs the following message:

CP/M on Exidy Sorcerer for 40 Track Disk
32K CP/M VERS 1.42/3
Copyright (C) 1980 Exidy Inc.

A>

There are other Exidy CP/M's for 77 track disks. The FDS user should insure his system says "For 40 Track Disk."

Apart from configuration differences, CBIOS operates the same on both systems, DDS and FDS.

B. System Generation

Two programs, MOVCPM and SYSGEN, either create a new CP/M system or change its location in RAM. The MOVCPM program obtains a CP/M system image sized appropriately, for example, 32K, 48K, etc. and performs system relocation. SYSGEN takes the system output from MOVCPM and writes it to tracks 0 and 1 of the target diskette. The simplest method of doing this is:

A> MOVCPM 32 * <carriage return>
EXIDY MOVCPM PROGRAM VERSION 3
CONSTRUCTING 32K CP/M VERS 1.4
READY FOR "SYSGEN" OR
"SAVE 40 CPM32.COM"
A>
Notice that "MOVCPM 32 *" is the only user input. At this point MOVCPM has created a 32K CP/M system image in memory and the user may either proceed immediately to SYSGEN or he may save the image on disk as a COM file by typing SAVE 40 CPM32.COM in response to the "A" prompt. The latter procedure provides the user the option of modification with the DDT utility.

With a MOVCPM image in memory, as it is after the last prompt, the user may do a memory image SYSGEN. In the next example, the system is created on drive B. Note the response to the source drive name prompt is a carriage return. This indicates the source system already exists in memory.

A>SYSGEN <carriage return>
EXIDY SYSGEN VER 1.4/3

SOURCE DRIVE NAME (OR RETURN TO SKIP)
DESTINATION DRIVE NAME (OR RETURN TO REBOOT)B
DESTINATION ON B, THEN TYPE RETURN <carriage return>
CP/M TO BE CREATED WILL RESIDE IN RAM FROM 6000 TO 7F00

The actual disk writing occurs now then this message signs on:

FUNCTION COMPLETE
DESTINATION DRIVE NAME (OR RETURN TO REBOOT)

At this point the system has been created on disk. The user should do a cold boot on the new diskette (in drive A) to verify this. For more details on MOVCPM and SYSGEN, see "An Introduction to CP/M Features and Facilities."

MOVCPM may also be performed without the second "*" parameter MOVCPM 29. In this case, MOVCPM attempts to create and execute in memory a new system of the specified size. However, the system may be destroyed if the given memory size causes the new target system to use memory in either the executing MOVCPM program or the executing CP/M system. Exidy suggests always specifying the second "*" parameter for MOVCPM and using SYSGEN to create a new disk system.

C. Options

Two options come with the CBIOS: Diagnostic Error Messages and Read After Write Data Verification. The default settings are 1) no Diagnostic Error Messages and 2) Read After Write Verification.
With the Diagnostic Error Message option turned off, only fatal errors are reported to the user and recovered soft errors are not. The user selects this option if he wants all errors reported. A Diagnostic Error message shows for any error encountered.

The Read After Write Verification option allows all data to be reread and compared to the write buffer and CRC after the user has written to a physical disk sector. The user may turn off this option, increasing the speed of disk writes by 50 per cent. This may, however, decrease data reliability. We suggest leaving this option set to the default value to assure disk writes are being done successfully.

Only someone familiar with CP/M programming should attempt changing option values. To make these changes, the user creates a disk file with his CP/M system on it. That is, he does a MOVCPM, followed by a SAVE, as described in Section II. A. of this manual. The user then DDT's the CP/M system into memory, altering the contents of absolute location 1F02 hex to reflect the options he wishes, as shown below:

bit 0 = Read After Write Option (hex 01)
bit 2 = Diagnostic Error Message Option (hex 04)

If the value of the bit is 1, then the option is asserted. A bit of 0 turns off the option. Thus the default value is 01 hex for the Read After Write option without Diagnostic Error Messages.

D. Cache Buffer Allocation

The CB IOS dynamically allocates cache buffers whenever CP/M is cold booted. CB IOS determines the top of memory by calling the Sorcerer Monitor routine GETLY (EIA2H). This returns the address of the Monitor Work Area (MWA), located in high memory. CB IOS subtracts 40 hex (64 decimal) from this address for Sorcerer Monitor stack forming an upper buffer address bound. The area from the end of the CB IOS code area to this bound contains all the CB IOS buffers. The maximum buffers possible is seven. A CB IOS buffer is 262 bytes long. The maximum of seven is chosen to insure that the CB IOS is not so overbuffered that more buffer bookkeeping overhead is performed than I/O.
The first parameter of the MOVCPM function controls the number of buffers in the system. For example, if the user has a 32K memory Sorcerer, he may generate his CP/M system through the usual MOVCPM 32 * command. This allocates the minimum of one cache buffer. A MOVCPM 31* command "lies" to CP/M, telling it the machine is 1K smaller than it actually is. This allocates an extra 1K to CBIOS buffers, allowing 4 extra buffers. Thus a "MOVCPM 31 *" command increases that number of buffers from one to five. A "MOVCPM 30 *" command increases that number to seven. (Analogous numbers hold for a 48K Sorcerer, that is, 48 results in one, 47 results in five, and 46 results in seven buffers. If the cold boot is located at BFOO H, then the largest possible CP/M is 47K, using 4 buffers. A 48K CP/M does not work if the boot is at BFOOH.

E. Incompatibilities

All CP/M disk formats, including Exidy's, are incompatible with the Micropolis Disk Operating System (MDOS) and with some other CP/M formats. This incompatibility, especially evident with MDOS, is a result of different sector skewing arrangements. All Exidy disk-based software products only run on Exidy's CP/M TM.

The Exidy skewing pattern follows for those interested in developing translation programs.
### Sector Skew Pattern

<table>
<thead>
<tr>
<th>CP/M logical 128-byte sector</th>
<th>Exidy 256-byte Sector (Physical Sector, first/last half)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16, first</td>
</tr>
<tr>
<td>2</td>
<td>16, last</td>
</tr>
<tr>
<td>3</td>
<td>13, first</td>
</tr>
<tr>
<td>4</td>
<td>13, last</td>
</tr>
<tr>
<td>5</td>
<td>10, first</td>
</tr>
<tr>
<td>6</td>
<td>10, last</td>
</tr>
<tr>
<td>7</td>
<td>7, first</td>
</tr>
<tr>
<td>8</td>
<td>7, last</td>
</tr>
<tr>
<td>9</td>
<td>4, first</td>
</tr>
<tr>
<td>10</td>
<td>4, last</td>
</tr>
<tr>
<td>11</td>
<td>1, first</td>
</tr>
<tr>
<td>12</td>
<td>1, last</td>
</tr>
<tr>
<td>13</td>
<td>14, first</td>
</tr>
<tr>
<td>14</td>
<td>14, last</td>
</tr>
<tr>
<td>15</td>
<td>11, first</td>
</tr>
<tr>
<td>16</td>
<td>11, last</td>
</tr>
<tr>
<td>17</td>
<td>8, first</td>
</tr>
<tr>
<td>18</td>
<td>8, last</td>
</tr>
<tr>
<td>19</td>
<td>5, first</td>
</tr>
<tr>
<td>20</td>
<td>5, last</td>
</tr>
<tr>
<td>21</td>
<td>2, first</td>
</tr>
<tr>
<td>22</td>
<td>2, last</td>
</tr>
<tr>
<td>23</td>
<td>15, first</td>
</tr>
<tr>
<td>24</td>
<td>15, last</td>
</tr>
<tr>
<td>25</td>
<td>12, first</td>
</tr>
<tr>
<td>26</td>
<td>12, last</td>
</tr>
<tr>
<td>27</td>
<td>9, first</td>
</tr>
<tr>
<td>28</td>
<td>9, last</td>
</tr>
<tr>
<td>29</td>
<td>6, first</td>
</tr>
<tr>
<td>30</td>
<td>6, last</td>
</tr>
<tr>
<td>31</td>
<td>3, first</td>
</tr>
<tr>
<td>32</td>
<td>3, last</td>
</tr>
</tbody>
</table>

Note that Exidy physical sectors are numbered 1 to 16, and are 256 physical bytes long. CP/M logical sectors are numbered 1-32 and are 128-bytes long. Thus, two CP/M sectors fit in one Exidy physical sector.
3. FEATURES

A. Error Recovery

The CBIOS includes extensive automatic error detection, recovery, and reporting facilities. The Read After Write option, active by default, is the only error detection function controlled by the user. When a disk I/O error occurs, recovery is fully automatic in the following steps:

1. CBIOS retries operation: up to 5 times until successful.
2. If the error still exists, it steps one track in/out alternately for a total of 6 times and repeats step 1 again.
3. If error still exists, then it deselects/reselects drive and then homes to track 0, up to 2 times, repeating 1 and 2. If error still exists, the error is treated as "permanent" and unrecoverable and the operation is aborted.

These error recovery steps are performed in nested fashion. That is, a separate counter is maintained for each error retry state, 1, 2, and 3. If step 1 fails, (its counter reaching 5), then step 2 is performed and its counter incremented. Meanwhile, the step 1 counter is reset, and its process again performed. If successive errors cause the step 2 counter to reach its maximum, then step 3 is performed, and its counter incremented. Both the first and second counters are reset, and step 1 is reinitiated. Thus a total of sixty (5x6x2) retry steps are performed before the error is declared non-recoverable. This retry process can take up to 75 seconds.

If the error is non-recoverable, the CBIOS issues an error message stating:

n DRV: ERR CODE= D

n here identifies the drive A, B, or C. Further identification of the error code follows the message.

The CBIOS then returns the error to its caller, the CP/M Basic Disk Operating System (BDOS). BDOS reports the error to the user, in less descriptive terms than the CBIOS in the following message:

BDOS ERR ON n: BAD SECTOR
The BDOS operation is suspended until the user hits any key except control-C. When any other key is hit, the BDOS retries the I/O. If the user wishes to end error processing, he must hit control-C or reset to the Sorcerer Monitor and perform either a warm boot (GO 0) or a cold boot.

If the user chooses the Diagnostic Error Message option, each error issues an I/O error message even if recovered by the CBIOS. In the event of a nonrecoverable error, the CBIOS prints 60 diagnostic error messages before declaring the error nonrecoverable and issuing the above error message. This procedure slows down recovery considerably. Only technicians diagnosing disk-related hardware errors should use this option.

One peculiar "error" of CP/M systems is the write protect error. The CBIOS shows this error message to the user:

```
   n DRV: ERR CODE=B
```

However, the CBIOS doesn't report the error to the BDOS. Thus the BDOS thinks it is writing to a disk, but cannot because it is a write protected disk. The BDOS discovers the error only after it reads back the directory data and it does not agree with what it remembered having "written". This usually results in the following error message:

```
BDOS ERR ON n: R/O
```

The write protect error occurs when CP/M performs "token" directory writes upon reading each new extent of a file. Thus if a PIP (Peripheral Interchange Program) is performed on a large (>16K) file from a write protected diskette to a writable diskette, the token directory writes cause write protect errors on the write protected source diskette. If the CBIOS returned the write protect error to the BDOS, the user could never copy files from a write protected diskette (even though only reads are to be done). These write protect errors on a diskette used only for input can be ignored as a peculiarity in CP/M.
B. **CP/M Programming**

Cache BIOS does not immediately, upon user request, execute disk writes. At any given moment there may be "dirty" buffers in the cache, that is, buffers which should be written to disk. Writing such buffers to disk is called "flushing the cache". The typical user who interfaces to the CBIOs through the BDOS, that is, does logical file I/O, documented in the "Interface Guide", does not need to be aware of the flushing mechanism. The cache is automatically flushed upon BDOS file closing. Only the user who performs direct CBIOs I/O through the vector needs to be aware of cache flushing. The cache is flushed when:

1. An I/O error occurs
2. A write to the directory track occurs
3. A CBIOs disk select occurs

Programmers using the non-standard CBIOs I/O functions and not the standard BDOS ones should be careful and account for caching processes.

The best guide to CP/M programming is Digital Research's "CP/M Interface Guide." Although the guide is accurate and informative, additional information may help the user overcome any problems he may encounter, as listed below.

1. The address of the last SETDMA call is used as the CP/M directory buffer of the BDOS Open, Close, and Delete commands. These calls destroy the buffer set for the last disk I/O. This feature is visible when the CCP SAVE command destroys the last 128 bytes of memory it writes to disk after closing the file it just saved. **THUS MORE THAN ONE SAVE CANNOT BE DONE IN A ROW.**

2. The BDOS search commands (function numbers 17 and 18) do not work as indicated in the "CP/M Interface Guide". The following provides accurate information:

   a. The Search command (17) does not return a byte pointer. Instead, it returns the index of the found file (within the directory) to register A, or to 255 if a match is not found. The index of the file is within the range of 0 to 127, since the Exidy CP/M contains up to 128 directory entries. Directory entries are 32 bytes per entry, thus there are four entries per sector. The BDOS, searching for the...
desired file, reads directory sectors into its DMA buffer, located from 80 to FF hex. These facts provide the basis for the following formula. The File Control Block (FCB) for the found file is located at:

\[ 80H + \text{MOD(index,4)} \times 32 \]

The BDOS returns "index" and MOD is the modulus function which returns the remainder of "index" divided by 4.

b. Only after the initial search (17), may search (18) occur. The Interface Guide incorrectly states that an FCB parameter is required. Actually, the FDB from the previous search call (17) is used. The parameter returns in the A register and is a directory index exactly as described above in 2a.

4. ERROR MESSAGES

The following is the error message format with an explanation of the various error codes:

\[ \text{x DRV: ERR CODE=c} \]

where \( x \) represents the drive on which the error occurred (A, B, or C) and \( c \) represents one of the following:

A – Disk Select Error—The disk selected was not drive "A", "B", or "C".

B – Write Protect Error—The attempt to write to a write protected diskette was not reported to the BDOS as an error. See III. B., CP/M Programming.

C – Disk Track Out of Range—A track number was detected past the end of the disk, indicating that the CP/M is very sick.

D – Non-Recoverable Disk I/O Error—All retries have failed to eliminate a read or write error. A more complete description follows this section.

E – Insufficient Memory for Disk Cache Buffers—At initialization, cache buffers are allocated. This error occurs if there is not sufficient space for one cache buffer, see II. D, Cache Buffer Allocation.

F – Error on Cache Flush—An error occurred while the cache was flushed. The last CP/M job or command should be redone.
A non-recoverable disk I/O error (Code D) has a few possible causes, which is one of the following:

(I/O Type): TRK=ttH, SCTR=ssH: STAT=bbH *

Where I/O Type is of the following:

READ ERR - Error occurred during read operation
WRITE ERR - Error occurred during write operation
RDAFTWR VERIFY ERR - The read after write option was selected, and the verify did not agree. The write however, was done successfully.

Note that track and sector values (tt and ss) are expressed in hexadecimal.

The status value (bb) is eight bits of status flags from the disk controller, expressed in hexadecimal. Only the status flags listed below are used for error indicators.

04H (bit 2) - Lost data (data overrun/underrun)
08H (bit 3) - CRC error
10H (bit 4) - Record Not Found (RNF)
20H (bit 5) - Write Fault for Write command
40H (bit 6) - Write Protect Flag
80H (bit 7) - Not Ready Flag

These codes show up in messages like this:

A DRV: ERR CODE = D
READ ERR: TRK=02, SCTR=05: STAT=10H

This message would tell the user that he has a non-recoverable disk I/O error (ERR CODE = D) on the second track of sector 5 and its status, 10H refers to bit 4, Record Not Found.
I. INTRODUCTION

The operation of the CP/M CBIOS on the hard-sectored Micropolis Mod II (77 track) and I (38 track) is very similar to its operation on the DDS and the FDS. Therefore, all the information described for those systems is accurate for users with the Micropolis Mods I and II. The only differences are listed here:

II. HARDWARE FOR THE MICROPOLIS MOD I AND II DUAL DISK UNIT WITH S100 AND DDS WITH HARD-SECTORED 77 TRACK DISKS.

The Exidy CBIOS runs on a Sorcerer system with S100 Expansion and Micropolis Dual Disk Units or with the hard-sectored DDS disk. The dual disk is a Micropolis quad density, and the associated disk controller is for hard 16 sector per track diskettes. The CBIOS supports up to 4 drives, allowing the usage of two dual disk units on a system.

Upon cold boot, CP/M should output the following message on the system:

CP/M on Exidy Sorcerer
32K CP/M VERS. 1.42/3
Copyright (C) 1979 Exidy Inc.

It should be noted that the Mod I has 38 tracks and the Mod II or hard DDS has 77.

II. OPTIONS

All three systems (Micropolis Mods I and II and DDS Hard Sectored 77 track disks) have both the options described earlier, Diagnostic Error and Read After Write Data Verification. An additional option, the Pre-Write Read Verification is featured on these systems. By default this third option is set.

The Pre-write Read option verifies that the disk head is located on the proper track before the write operation is performed. This verification is performed by reading the header of a sector on the track and insuring that the number in the header agrees with the requested track. This option causes no performance degradation due to the interleaving sectors on the disk. The user is advised to leave this option set.
The bit designations remain the same for the other two options, with bit 1 (02 hex) assigned to Pre-Write Read Verification Option. If the value of the bit is 1, then the option is asserted. Thus the default value is 03 hex, with Read After Write, Pre-Write Read Verification and no Diagnostic Error Messages options asserted.

III. Error Messages

The status value, expressed in hexadecimal at the conclusion of Error Messages is not the same for any of these three systems. Instead, the lower digit value of STAT in an error code is interpreted as follows (the upper digit has no meaning for the user):

<table>
<thead>
<tr>
<th>Stat</th>
<th>Message/Explanation</th>
</tr>
</thead>
</table>
| X1   | HOME ERR
     | Did 90 step-outs but could not find track 0 status. |
| X2   | SEEK ERROR
     | (Not currently implemented.) |
| X3   | SYNC BYTE NOT OFFH
     | During a read operation, the sync byte was not FF. |
| X4   | HDR BYTE NOT ZERO
     | Header byte with non-zero value (all ten header bytes must be zero). |
| X5   | TS ID MISMATCH
     | Track/sector ID read from diskette do not match track/sector requested. |
| X6   | CKSNS DON'T AGREE
     | Checksum read from a sector does not match the checksum computed from the data read. |
| X7   | DATA COMPARE ERROR
     | Read after write found that data did not agree with the data written. |
COPYRIGHT

Copyright (c) 1980 by Exidy Inc.. All rights reserved. No part of this publication may be reproduced, transmitted transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Exidy Inc., 390 Java Drive, Sunnyvale, California 94086.

Since this manual is tutorial in nature, permission is granted to reproduce or abstract the example procedures and sample programs for the purposes of inclusion within the reader's programs.

DISCLAIMER

Exidy Inc. makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Exidy Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Exidy Inc. to notify any person of such revision or changes.

TRADEMARKS

CP/M is a registered trademark of Digital Research.
# TABLE OF CONTENTS

1. INTRODUCTION ........................................... 1  
2. USE ......................................................... 2  
3. SAMPLE RUN ............................................... 3  
   3.1 EXCOPY with Multiple Drives ...................... 3  
   3.2 EXCOPY with One Drive ........................... 3  
   3.3 Format Only ....................................... 5  
4. ERROR MESSAGES .......................................... 6  
   4.1 Cannot format ..................................... 6  
   4.2 Write protected .................................. 6  
   4.3 Write error ....................................... 6  
   4.4 Read back error ................................. 6  
   4.5 Additional Messages ............................. 6  
5. RECOVERY ................................................... 7
1. INTRODUCTION TO EXCOPY

EXCOPY (TM) is a CP/M (TM) program which formats or copies formatted, soft-sectored diskettes on the Exidy Floppy Disk Subsystem (FDS) and Display Disk System (DDS). The copying operation automatically formats the destination diskette on either a single or a multiple drive configuration. The copy program minimizes the number of read/write cycles performed by determining the amount of RAM available as a copy buffer, using as much of it as possible. Disk formatting without copying may also be performed by the program. The user must not violate any software licensing agreements when copying diskettes.
2. USE

The practical user will want to secure back-up copies of all his important files, protecting against any possibility of losing data and enabling the user to read and write to the disk. Also, the user may wish to format only, that is, create on a new disk proper tracks and sectors to read and write to.

EXCOPY is called from CP/M by simply entering "EXCOPY" on the CP/M command line. The program signs on and requests an indication of one of these responses: "C" (or carriage return) for copy, "F" for format only, or "E" (or control-C) for exit back to CP/M.

If copying is requested, the program asks if more than one disk drive is configured and available for the copying operation. The response "Y" or "N" indicates yes or no. If more than one drive is indicated, a message instructs you to place the source diskette in drive A and the destination diskette in drive B. Copying automatically occurs after striking any key.

If only one drive is indicated, a message instructs you to first insert the destination diskette in the A drive for formatting. Hitting any key triggers this process. Then, alternately place source and destination diskettes in the A drive as requested by the console messages.

WARNING: Copying and Formatting destroys any previous information on the destination diskette. BE SURE THERE ARE NO IMPORTANT FILES ON THE DESTINATION DISKETTE, AS THEY WILL BE LOST. ALSO, PAY CAREFUL ATTENTION TO THE DRIVE YOU PLACE YOUR SOURCE AND DESTINATION DISKETTE IN. Any mixup will lose all information on your source diskette. Exidy suggests putting write-protect tabs on your source diskette to guard against any such mixup.

After copying, the console asks if more is desired. If your response is "Y" (yes), the cycle is repeated. Otherwise, the program directs the operator to place a system diskette in the A drive, hitting any key to re-boot the system.

Formatting without copying may also be requested. In this case, the program asks the user to specify which drive he wishes to format, "A", "B", or "C". After completion, the program asks if more is desired, and repeats the cycle if "Y" is entered.
3. **SAMPLE RUN:**

3.1 **EXCOPY With Multiple Drives**

Here is a sample of the console I/O when using EXCOPY. For clarification, user input is underlined to differentiate from program output:

A> EXCOPY (return)

Exidy Disk Copy & Format Program
For xx track diskettes. Ver 1.1
Copyright (C) 1980 Exidy Inc.

Format only, Copy, or Exit (F/C/E)?... C

Do you have more than one drive configured in this system (Y/N)? Y

Put source diskette in drive A and destination diskette in drive B then Hit any key when ready. (any key)

(copied commences)

Good Copy.

More (Y/N)? N

Place system diskette in drive A and Hit any key when ready to reboot. any key

(CP/M reboots)

xx, here, stands for either 40 track diskettes in the FDS or 77 track diskettes in the DDS.

3.2 **EXCOPY With One Drive**

The following is the console I/O when invoking EXCOPY with only one drive configured on the Sorcerer. An asterisk (*) indicates the point where the program waits until any key is hit. Track numbers indicated will vary depending on size of RAM. In this example, RAM is 32K. Once again, be sure to begin by placing the DESTINATION diskette in the drive. Should you confuse it with the source diskette, all information will be permanently lost.
A>EXCOPY (return)

Exidy Disk Copy & Format Program
For xx track diskettes. Ver 1.1
Copyright (C) 1980 Exidy Inc.

Format only, Copy, or Exit (F/C/E)?...(return)

Do you have more than one drive configured in this system (Y/N)? N

Place destination diskette in drive A
For initial formatting and
Hit any key when ready. (any key)

(formating commences)

<table>
<thead>
<tr>
<th>Tracks</th>
<th>Tracks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source:* 0-- 6</td>
<td>Destination:* 0-- 6</td>
</tr>
<tr>
<td>Source:* 7-- 13</td>
<td>Destination:* 7-- 13</td>
</tr>
<tr>
<td>Source:*14--20</td>
<td>Destination:*14--20</td>
</tr>
<tr>
<td>Source:*21--27</td>
<td>Destination:*21--27</td>
</tr>
<tr>
<td>Source:*28--34</td>
<td>Destination:*28--34</td>
</tr>
<tr>
<td>Source:*35--39</td>
<td>Destination:*28--39</td>
</tr>
</tbody>
</table>

Good Copy.

More (Y/N)? [N]

Place system diskette in drive A and
Hit any key when ready to reboot. [any key]

(cold boots system diskette)

Note: The above example pertained to FDS only in the screen display of track numbers. The difference for DDS, also with a 32K RAM, is as follows:

<table>
<thead>
<tr>
<th>Tracks</th>
<th>Tracks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source:* 0-- 6</td>
<td>Destination:* 0-- 6</td>
</tr>
<tr>
<td>Source:* 7-- 13</td>
<td>Destination:* 7-- 13</td>
</tr>
<tr>
<td>Source:*14--20</td>
<td>Destination:*14--20</td>
</tr>
<tr>
<td>Source:*21--27</td>
<td>Destination:*21--27</td>
</tr>
<tr>
<td>Source:*28--34</td>
<td>Destination:*28--34</td>
</tr>
<tr>
<td>Source:*35--41</td>
<td>Destination:*35--41</td>
</tr>
<tr>
<td>Source:*42--48</td>
<td>Destination:*42--48</td>
</tr>
<tr>
<td>Source:*49--55</td>
<td>Destination:*49--55</td>
</tr>
<tr>
<td>Source:*56--62</td>
<td>Destination:*56--62</td>
</tr>
<tr>
<td>Source:*63--69</td>
<td>Destination:*63--69</td>
</tr>
<tr>
<td>Source:*70--76</td>
<td>Destination:*70--76</td>
</tr>
</tbody>
</table>
3.3 Format Only

The console I/O for a Format Only operation would appear as follows:

A>EXCOPY (return)

Exidy Disk Copy & Format Program
For xx track diskettes. Ver 1.1
Copyright (C) 1980 Exidy Inc.

Format only, Copy, or Exit (F/C/E)?...F
Select drive (A,B, or C)...B

Place diskette in drive B for formatting then
Hit any key when ready. any key

(formatting commences)

More (Y/N)? N
(etc.)
4. ERROR MESSAGES

Several conditions may display error messages on the console, as follows:

4.1 Cannot format, try again

This message appears when the disk controller either fails to either write a track with the formatting data, or after writing, the controller cannot read back each sector on the track in question. An improper destination diskette (such as a hard-sectored diskette) or a worn or damaged one may cause this. Also, malfunctioning disk drive hardware which prevents formatting may return this message. Try fresh media and double-check the hardware. The program automatically restarts after each error to allow another attempt.

4.2 Destination is write protected.

This message is given if the destination diskette has a write protect tab covering its write protect notch. Either remove the tab or use an unprotected diskette.

4.3 Write error on track # xx

This message is returned during formatting if the write operation cannot be performed. The track number (xx) is specified.

4.4 Read back error on track # xx

This message is displayed if a track can be written to but cannot be read back after repeatedly attempting to do so. Again, the track number is specified as (xx).

4.5 Additional Message

I can't find your boot address!!!!
Please enter it...[hhhh]

This occurs if a non-Exidy boot-strap controller prom is used and the copy program can't determine where the cold bootprogram is addressed. In this event, simply enter the hexadecimal address of your boot-strap prom. This should not occur if Exidy hardware and software is used.
5. **Recovery**

If the copy program has errors while reading the source, writing the destination, or verifying the destination diskette, the program repeatedly attempts the operation until successful. If, after many attempts, the operation still has errors, a message is displayed and the error retry continues indefinitely.

These messages are:

**Recovering from read errors, track # xx**

**Recovering from write errors, track # xx**

If it appears that the copy program cannot recover from an error, reset the computer to end processing and check ground and other connections between computer and disk drive. Also check the diskette for wear.