INTERTEC DATA SYSTEMS®

# SUPERBRAIN™

USERS MANUAL

$50.00

# USERS MANUAL FOR
# INTERTEC'S
# SUPERBRAIN™
# VIDEO COMPUTER SYSTEM

## IMPORTANT NOTICE

This version of the SuperBrain Users Manual is intended for use with the SuperBrain or SuperBrain QD Video Computer Systems. However, this manual is applicable only for those units with Revision-01 of the Keyboard/CPU module, and version 3.0 or higher of the DOS and boot loader. If you have a Revision-00 Keyboard/CPU module, then use only the First or Second Edition of this manual.

**Document No. 6831010**
**September 1980**

This is the fourth edition of this manual. Your warranty registration form must be returned promptly to assure receipt of future revisions, if any, to this document.

## *** IMPORTANT ***

Do not attempt to write or save programs on your system diskette. It has been 'write protected' by placing a small adhesive aluminum strip over the notch on the right hand side of the diskette. Such attempts will result in a 'WRITE' or 'BAD SECTOR' error.

Before using your SuperBrain please copy the System Diskette onto a new blank diskette - an Intertec 1121010 diskette. If you do not have such a diskette, contact you local dealer. He should be able to supply you with one. If you have any questions concerning this procedure please contact your dealer before proceeding. Failure to do so may result in permanent damage to your System Diskette.

BEFORE APPLYING POWER TO THE MACHINE INSURE THAT NO DISKETTES ARE INSERTED INTO THE MACHINE. NEVER TURN THE MACHINE ON OR OFF WITH DISKETTES INSERTED IN IT. FAILURE TO OBSERVE THIS PRECAUTION WILL MOST DEFINITELY RESULT IN DAMAGE TO THE DISKETTES.

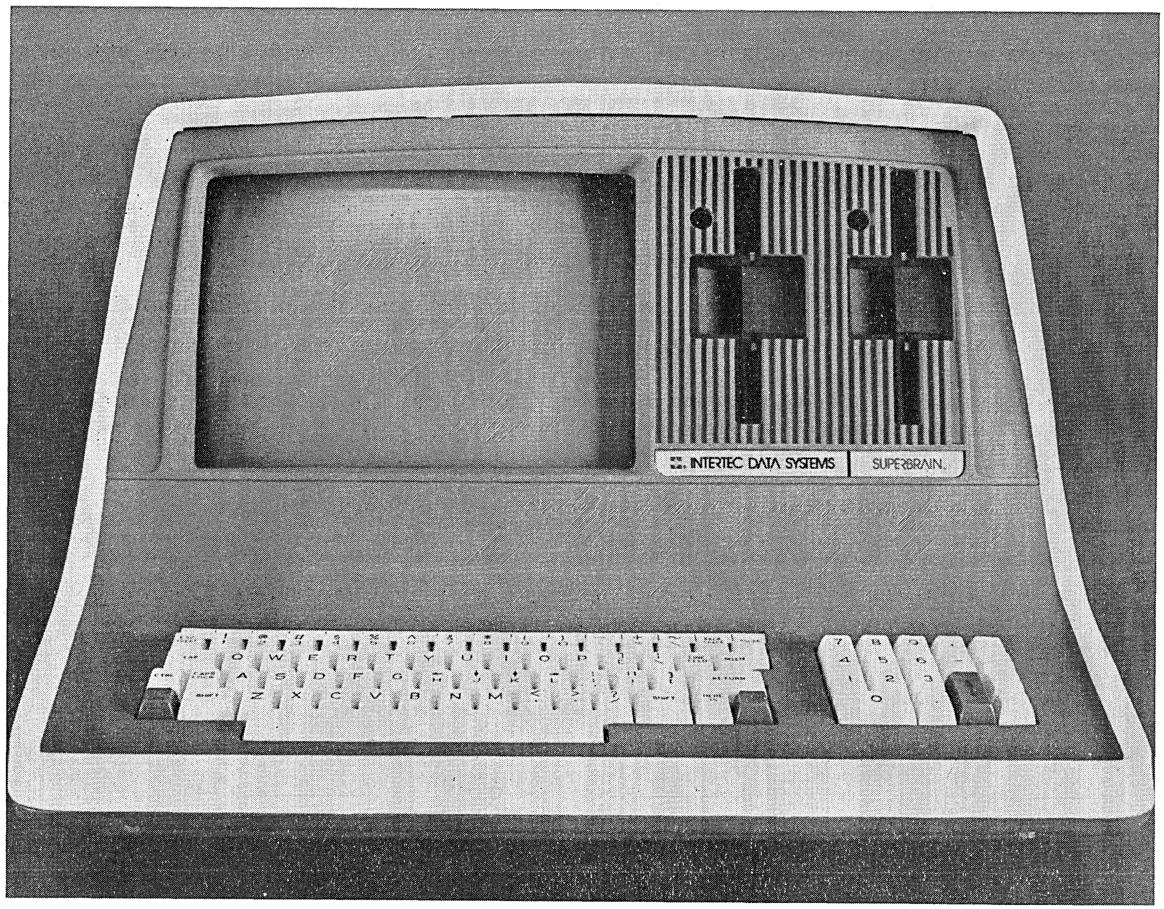
# CONFIDENTIAL
# AND
# PROPRIETARY INFORMATION

Information presented in this manual is furnished for customer reference only and is subject to change.

This document is the property of Intertec Data Systems Corporation, Columbia, South Carolina, and contains confidential and trade secret information. This information may not be transferred from the custody or control of Intertec except as authorized by Intertec and then only by way of loan for limited purposes. It must not be reproduced in whole or in part and must be returned to Intertec upon request and in all events upon completion of the purpose of the loan.

Neither this document nor the information it contains may be used or disclosed to persons not having a need for such use or disclosure consistent with the purpose of the loan without the prior express written consent of Intertec.

INTERTEC DATA SYSTEMS CORPORATION
Columbia, South Carolina



THE SUPERBRAIN VIDEO COMPUTER SYSTEM

# CONGRATULATIONS ON YOUR PURCHASE OF INTERTEC'S SUPERBRAIN

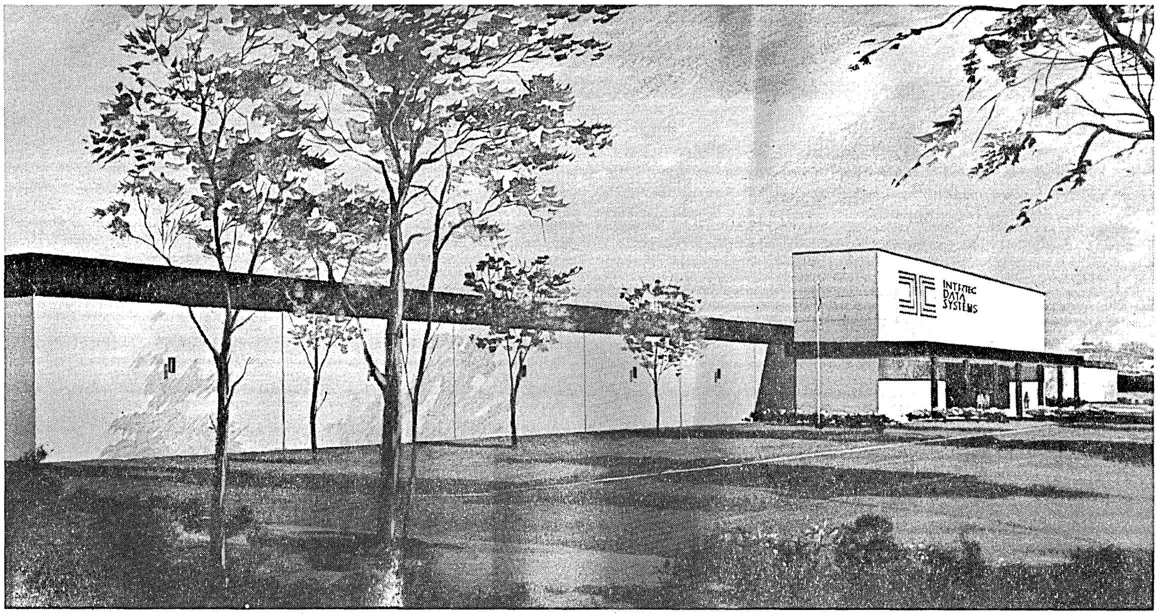
# VIDEO COMPUTER SYSTEM

Your new SuperBrain Video Computer was manufactured at Intertec's new 120,000 square foot plant in Columbia, South Carolina under stringent quality control procedures to insure trouble-free operation for many years. If you should encounter difficulties with the use or operation of your terminal, contact the dealer from whom the unit was purchased for instructions regarding the proper servicing techniques. If service cannot be made available through your dealer, contact Intertec's Customer Service Department at (803) 798-9100.

As with all Intertec products, we would appreciate any comments you may have regarding your evaluation and application for this equipment. For your convenience, we have enclosed a customer comment card at the end of this manual. Please address your comments to:

Product Services Manager
Intertec Data Systems Corporation
2300 Broad River Road
Columbia, South Carolina 29210

The SuperBrain is distributed worldwide through a network of dealer/OEM vendors and through Intertec's own marketing facilities. Contact us at (803) 798-9100 (TWX - 810-666-2115) regarding your requirement for this and other Intertec products.



INTERTEC
DATA
SYSTEMS®
Corporate Headquarters: 2300 Broad River Road, Columbia, South Carolina 29210 • 803 798 9100 • TWX: 810 666 2115

*Intertec's new one hundred and twenty thousand square foot corporate and manufacturing facility in Columbia, South Carolina*

# WILL THE MICROCOMPUTER YOU BUY TODAY STILL BE THE BEST MICROCOMPUTER BUY TOMORROW?

Probably the best test in determining how to spend your microcomputer dollar wisely is to consider the overall versatility of your terminal purchase over the next three to five years. In the fast-paced, ever-changing world of data communications, new features to increase operator and machine efficiency are introduced into the marketplace daily. We at Intertec are acutely aware of this rapid infusion of new ideas into the small systems business. As a result, we have designed the SuperBrain in such a manner as to virtually eliminate the possibility of obsolescence.

Many competitive alternatives to the SuperBrain available today provide only limited capability for high level programming and system expansion. Indeed, most low-cost microcomputer systems presently available quickly become outdated because of the inability to expand the system. Intertec, however, realizes that increased demands for more efficient utilization of programming makes system expansion capability mandatory. That means a lot. Because the more you use your SuperBrain, the more you'll discover its adaptability to virtually any small system requirement. Extensive use of "software-oriented" design concepts instead of conventional "hardware" designs assure you of compatibility with almost any application for which you intend to use the SuperBrain.

Once you read our operator's manual and try out some of the features described herein, we are confident that you too will agree with our "top performance - bottom dollar" approach to manufacturing. The SuperBrain offers you many more extremely flexible features at a lower cost than any other microcomputer we know of on the market today. The use of newly developed technologies, efficient manufacturing processes and consumer-oriented marketing programs enables us to be the first and only major manufacturer to offer such an incredible breakthrough in the microcomputer marketplace.

Browse through our operator's manual and sit down in front of a SuperBrain for a few hours. Then, let us know what you think about our new system. There is a customer comment card enclosed in the rear section of this manual for your convenience.

Thank you for selecting the SuperBrain as your choice for a microcomputer system. We hope you will be selecting it many more times in the future.

# TABLE OF CONTENTS

# INTRODUCTION

# INTRODUCTION

The SuperBrain Video Computer System represents the latest technological advances in the microprocessor industry. The universal adaptability of the SuperBrain CP/M* Disk Operating System satisfies the general purpose requirement for a low cost, high performance microcomputer system.

From the standpoint of human engineering, the SuperBrain has been designed to minimize operator fatigue through the use of a typewriter-oriented keyboard and a remarkably clear display. The SuperBrain displays a total of 1,920 characters arranged in 24 lines with 80 characters per line. The video display is usually crisp and sharp due to Intertec's own specially designed video driver circuitry. And, the high quality, non-glare etched CRT face plate featured on every SuperBrain assures ease of viewing and uniformity of brightness throughout the entire screen.

The SuperBrain's unique internal design assures users of exceptional performance for just a fraction of what they would expect to pay for such "big system" capabilities. The SuperBrain utilizes a single board "microprocessor" design which combines all processor, RAM, ROM, disk controller, and communications electronics on the same printed circuit board. This type of design engineering enables the SuperBrain to deliver superior, competitive performance.

Standard features of every SuperBrain include: two double-density, single-sided mini-floppies with a total of over 350,000 bytes **formatted** disk storage, 32K of dynamic RAM memory — expandable to 64K (in one 32K increment), a universally recognized CP/M* Disk Operating System featuring its own text editor, an assembler for assembly language programming, a program debugger and a disk formatter. Also standard are dual universal RS232 communications ports for serial data transmission between a host computer network via modem or an auxiliary serial printer. A number of transmission rates up to 9600 baud are available and selectable under program control.

Other standard features of the SuperBrain include: special operator convenience keys, dual "restart" keys to insure simplified user operation, a full numeric keypad complement, and a high quality typewriter compatible keyboard. An optional low cost S-100 bus adaptor is available to convert the SuperBrain Z80A data bus into an S-100 data and address compatible protocol. The S-100 adaptor accommodates one S-100 printed circuit board which can be mounted internally.

For reliability, the SuperBrain has been designed around 4 basic modules packaged in an aesthetically pleasing desk-top unit. These major components are: the Keyboard/CPU module, the power supply module, the CRT assembly, and the disk drives themselves. Failure of any component within the terminal may be corrected by simply replacing only the defective module. Individual modules are fastened to the chassis in such a manner to facilitate easy removal and reinstallation. Terminal down-time can be greatly minimized by simply "swapping-out" one of the modules and having component level repair performed at one of Intertec's Service Centers. Spare modules may be purchased from an Intertec marketing office to support those customers who maintain their own "in-house" repair facilities.

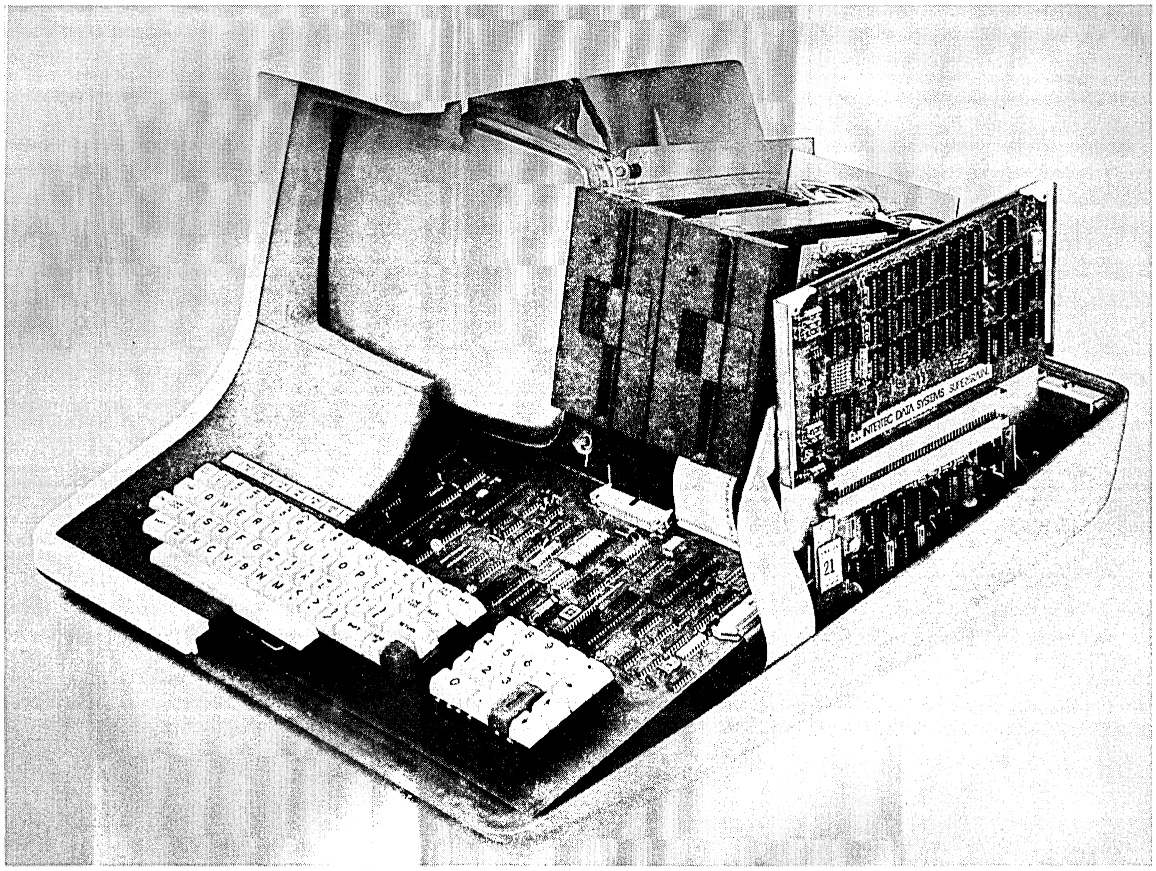
The SuperBrain's cover assembly is exclusively manufactured "in-house" by Intertec. A high-impact structural-foam material is covered with a special "felt-like" paint to enhance the overall appearance. Since the cover assembly is injected-molded, there is virtually no possibility of cracks and disfigurations in the cover itself. And, by manufacturing and finishing the cover assembly in-house, Intertec is able to specify only high quality material on the external and internal cover components of your SuperBrain to insure unparalleled durability over the years to come.

*CP/M is a registered trademark of Digital Research

## INTRODUCTION (continued)

A wide variety of programming tools and options are either planned or available for the SuperBrain. Standard software development tools available from Intertec include Basic, Fortran and Cobol programming languages. A wide variety of applications packages (general ledger, accounts receivable, payroll, inventory, word processing, etc.) are available to operate under SuperBrain CP/M Disk Operating System from leading software vendors in the industry. Disk storage may be increased by adding SuperBrain's S-100 bus adaptor and connecting other auxiliary disk devices, including hard disk drives. And, another model of the SuperBrain - SuperBrain QD - features double density, **double-sided** disk drives which provide over 700,000 bytes of **formatted** data.

The price/performance ratio of the SuperBrain has rarely been equalled in this industry. By employing innovative design techniques, the SuperBrain is not only able to offer a competitive price advantage but boasts many features found only in systems costing three to five times as much. SuperBrain's twin Z80A microprocessors insure extremely fast program execution even when faced with the most difficult programming tasks. And, each unit must pass a grueling 48 hour burn-in before it is shipped to the Customer. By combining advanced microprocessor technology with in-house manufacturing capability and stringent quality control requirements, your SuperBrain should provide unparalleled reliability in any application into which it is placed.



**CUTAWAY VIEW SHOWING MOUNTING OF MAJOR SUBASSEMBLIES.**

## SYSTEM SPECIFICATIONS

| FEATURE | DESCRIPTION |
|---|---|
| CPU | |
| Microprocessors | Twin Z80A's with 4MHZ Clock Frequency. One Z80A (the host processor) performs all processor and screen related functions. The second Z80A is "down-loaded" by the host to execute disk I/O. |
| Word Size | 8 bits |
| Execution Time | 1.0 microseconds register to register |
| Machine Instructions | 158 |
| Interrupt Mode | All interrupts are vectored and reserved. |
| **Floppy Disk** | |
| Storage Capacity | Over 350K (700K + on SuperBrain QD) total bytes of unformatted data on two double density drives. Optional external hard disk storage can be connected using the optional S-100 bus adaptor. |
| Data Transfer Rate | 250K bits/second |
| Average Access Time | 250 milliseconds. 35 milliseconds track-to-track |
| Media | 5 ¼ inch mini-disk |
| Disk Rotation | 300 RPM |
| **Internal Memory** | |
| Dynamic RAM | 32K (64K on Superbrain QD) bytes dynamic RAM. Expandable to 64K in one 32K increment. Optional 32K is socketed. |
| Static RAM | 1K bytes of static RAM is provided in addition to the main processor RAM. This memory is used for program and/or data storage for the auxiliary processor. |
| ROM Storage | 2K bytes standard. Allows ROM "bootstrapping" of system at power-on. |
| CRT | |
| Display Size | 12-inch, P4 phosphor. |
| Display Format | 24 lines x 80 characters per line. |
| Character Font | 5x7 character matrix on a 7x10 character field |
| Display Presentation | Light characters on a dark background. |

*Specifications subject to change without notice or liability.

## SYSTEM SPECIFICATIONS (continued)

| FEATURE | DESCRIPTION |
|---|---|
| Bandwidth | 15 MHZ. |
| Cursor | Reversed image (block cursor) |
| **Communications** | |
| Screen Data Transfer | Memory-mapped at 38 kilobaud. Serial transmission of data at rates up to 9600 bps. |
| Main Interface | RS-232C asynchronous. Synchronous interface optional. |
| Auxiliary Interface | Simplified RS-232C asychronous. Synchronous interface optional. |
| Z80A Data Bus | 40-pin Data Bus connector. |
| S-100 Bus | Connector provided for connection of optional S-100 bus adaptor. |
| Parity | Choice of even, odd, marking, or spacing - under program control. |
| Transmission Mode | Half or Full Duplex. One or two stop bits. |
| Addressable Cursor | Direct Positioning by absolute x, y addressing. |
| **System Utilities** | |
| Disk Operating System | CP/M 2.2 |
| DOS Software | An 8080 disk assembler, debugger, text editor and file handling utilities. |
| **Optional Software** | |
| FORTRAN | ANSI standard. Relocatable, random and sequential disk access. |
| COBOL | ANSI standard. Relocatable, sequential, relative and indexed disk access. |
| BASIC | Sequential and random disk access. Full string manipulation, interpreter. |
| Application Packages | Extensive software development tools are available from leading software vendors including software for the following applications: Payroll, Accounts Receivable, Accounts Payable, Inventory Control, General Ledger and Word Processing. |
| **Keyboard** | |
| Alphanumeric Character Set | Generates all 128 upper and lower case ASCII characters. |

*Specifications subject to change without notice.

## SYSTEM SPECIFICATIONS (continued)

| FEATURE | DESCRIPTION |
|---|---|
| Special Features | 2-Key Rollover, Keyboard lock/unlock - under program control. |
| Numeric Pad | 0-9, decimal point, comma, minus and user-programmable function keys. |
| Cursor Control Keys | Up, down, forward and backward. |
| **Internal Construction** | |
| Cabinetry | Structural foam |
| Component Layout | Four board modular design. All processor related functions and hardware are on a single printed circuit board. All video and power related circuits on separate single boards. |
| Mounting | All modules mounted to base. CRT in a rigid aluminum frame. Disk Drive assemblies are mounted into special bracket for ease of servicing. |
| **Environment** | |
| Weight | Approximately 45 pounds. |
| Physical Dimensions | 14 5/8" (H) x 21 3/8 (W) x 23 1/8 (D) |
| Environment | Operating: 0° to 40° C Storage: 0° to 85° C; 10 to 85% rel. humidity - non-condensing. |
| Power Requirements | 115 VAC, 60 HZ, 3 AMP (optional 230VAC/50HZ model available) |

*Specifications subject to change without notice.

# OPTIONAL VERSUS STANDARD FEATURES

Since each SuperBrain is designed utilizing the latest advances in microprocessor technology, many features which other system vendors offer as options are offered as standard features on the SuperBrain.

The SuperBrain Video Computer is designed to satisfy the universal requirement for a low cost, high performance small business system and, hence, there are virtually no options from which to choose. Basically, available options for the SuperBrain include:

BASIC 80 FROM MICROSOFT - an extensive implementation of Basic language available for Z80 microprocessors. In just three years of use, it has become the world's standard for microcomputer Basic. Basic 80 gives users what they want from a Basic - ease of use plus all of the features that make a micro perform like a minicomputer or large mainframe. Basic 80 meets the requirements of the ANSI subset standard for Basic and supports many unique features rarely found in other Basics.

MICROSOFT FORTRAN 80 - comparable to Fortran compilers on large mainframes and minicomputers. All of ANSI standard Fortran X3.9-1966 is included except the COMPLEX datatype. Therefore, users may take advantage of the many application programs already written in Fortran. Fortran 80 is unique in that it provides a microprocessor Fortran and assembly language development package that generates relocatable object modules. This means that only the subroutines and system routines required to run Fortran 80 programs are loaded before execution. Subroutines can be placed in a system library so that users develop a common set of subroutines that are used in their programs. Also, if only one module of a program is changed, it is necessary to recompile **only** that module.

CENTRONICS-COMPATIBLE PARALLEL INTERFACE(1) - connects directly to SuperBrain's 40 pin Z80A data bus connector and provides for a parallel output as required for Centronics-compatible printers.

S-100 BUS ADAPTOR(2) - connects to SuperBrain's auxiliary Z80A data bus edge card connector and provides for the connection of up to one standard sized S-100 bus board inside the SuperBrain cabinet. Bus adaptor includes ribbon cables, S-100 conversion circuitry, S-100 card guides and a metal mounting bracket to enable the S-100 bus adaptor to be installed on the inside cover just to the right of SuperBrain's twin double-density disk drives.

SYNCHRONOUS INTERFACE - enables synchronous transmission via the auxiliary RS232 serial communications port.

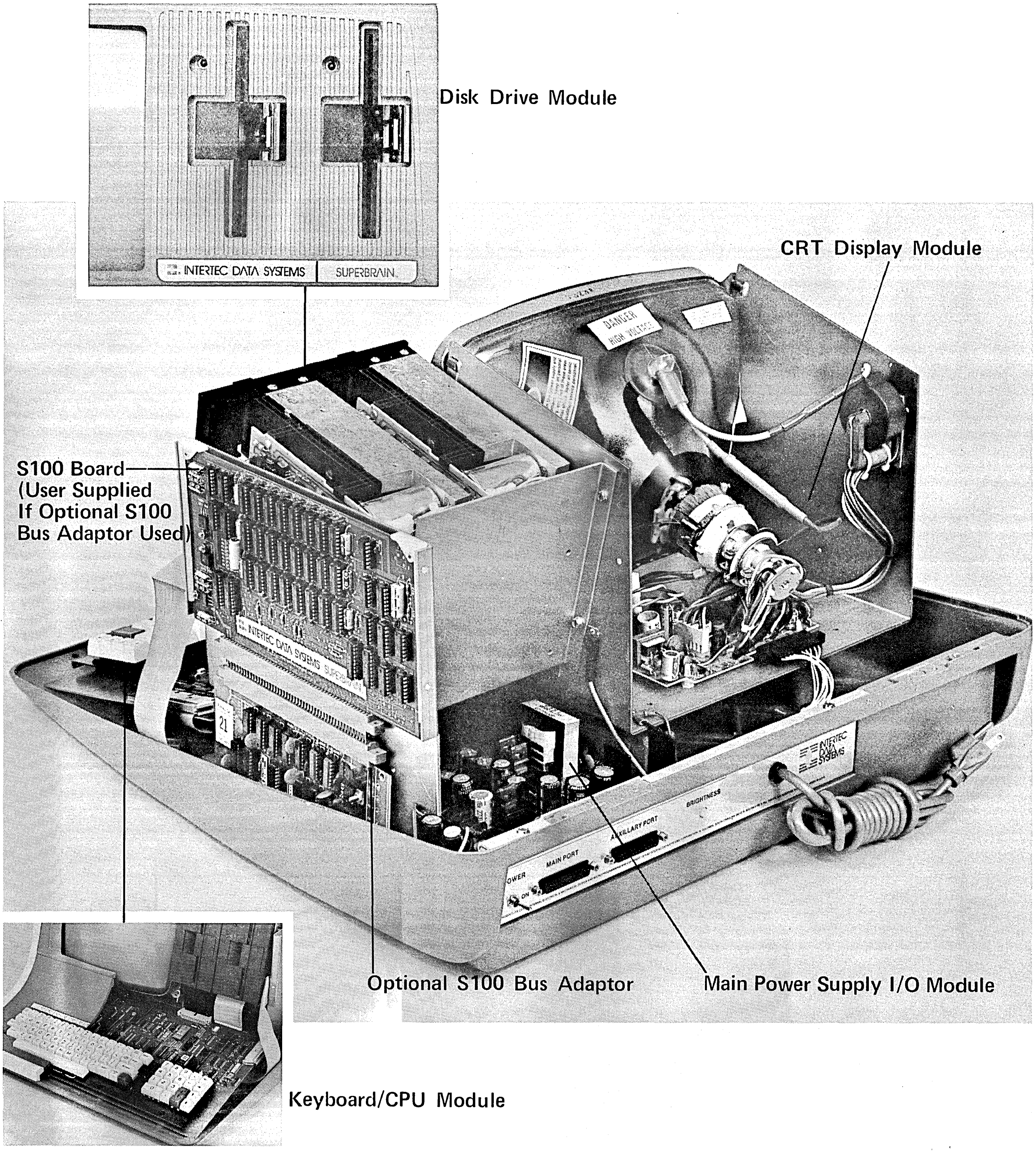
32K DYNAMIC RAM EXPANSION KIT - a set of sixteen 16K RAM chips which plug into existing sockets on the SuperBrain Keyboard/CPU module to enable expansion of the SuperBrain's dynamic memory from 32K to 64K. Also included with the RAM kit is an additional CP/M DOS Diskette which reconfigures the SuperBrain's Operating System to accommodate all 64K of RAM.

(1) Available June, 1980
(2) Available June, 1980

# MAJOR COMPONENTS

## INTERNAL CONSTRUCTION

Perhaps the most remarkable feature of the SuperBrain is its modular construction using only four major subassemblies which are clearly defined in their respective functions so as to facilitate ease of construction and repair. These four subassemblies are shown in figure one and described below.



## INTERNAL CONSTRUCTION (continued)

### KEYBOARD/CPU MODULE

The control section of the SuperBrain Video Computer is based upon the widely acclaimed Z80A microprocessor. The result is far fewer components and the ability to perform a number of functions not possible with any other approach. The Keyboard/CPU module (figure two) contains the SuperBrain's twin Z80 microprocessors. One Z80A (the host processor) performs all processor and screen related functions while the second Z80A can be "downloaded" to execute disk I/O handling routines. The result is extremely fast execution time for even the most sophisticated programs.

In addition to containing the SuperBrain's microprocessor circuitry, the Keyboard/CPU module contains 32K of dynamic RAM with sockets for an additional expansion capability of 32K (see figure three). Also found on this module is: the character and keyboard encoder circuitry, the "bootstrap" ROM, the disk controller and all communications electronics. Power is supplied to and signals are transferred from this module via a single 22 pin ribbon cable connected to the SuperBrain's main power supply module. Connection of this module to the disk drive subassemblies is via a separate ribbon cable. Figure four shows the connectors on the Keyboard/CPU module which are used for interconnecting this module with the disk drive subassemblies, the main power supply and the optional parallel and/or S-100 bus adaptor.

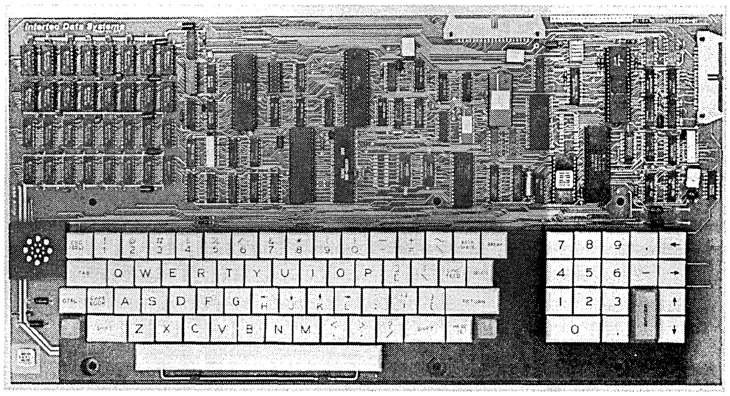


**Figure 2 - SuperBrain Keyboard/CPU Module**

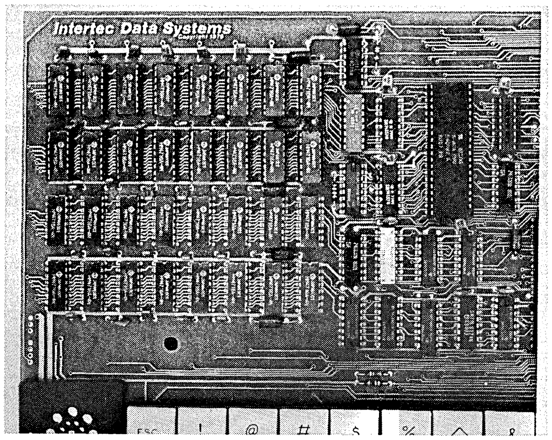


**Figure 3 - Dynamic RAM Section**
**Every SuperBrain is equipped with 32K dynamic RAM - on board expandable to 64K. 16 sockets are provided for the additional 32K of RAM.**

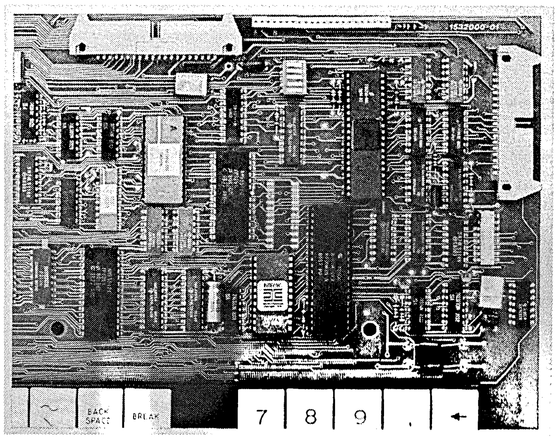


**Figure 4 - Keyboard/CPU Module Connectors**
**The 40 pin connector on the top edge of the card is for connection to SuperBrain's optional parallel and/or S100 bus adaptor. The 40 pin connector on the right edge routes signals to and from the disk drive assembly.**

## INTERNAL CONSTRUCTION (continued)

### CRT DISPLAY MODULE

The CRT Display Module consists of a 12 inch, high resolution, cathode ray tube mounted in a rigid aluminum chassis. The faceplate of the CRT is etched in order to reduce glare on the surface of the screen and provide uniform brightness throughout the entire screen area. The CRT display presentation is arranged in 24 lines of 80 characters per line.

The CRT video driver circuitry is mounted in the base of the CRT chassis to facilitate ease of removal and subsequent repair. In this manner, either the CRT itself or the video circuitry can be easily exchanged without disrupting any of the other major modules within the terminal (see figure five).

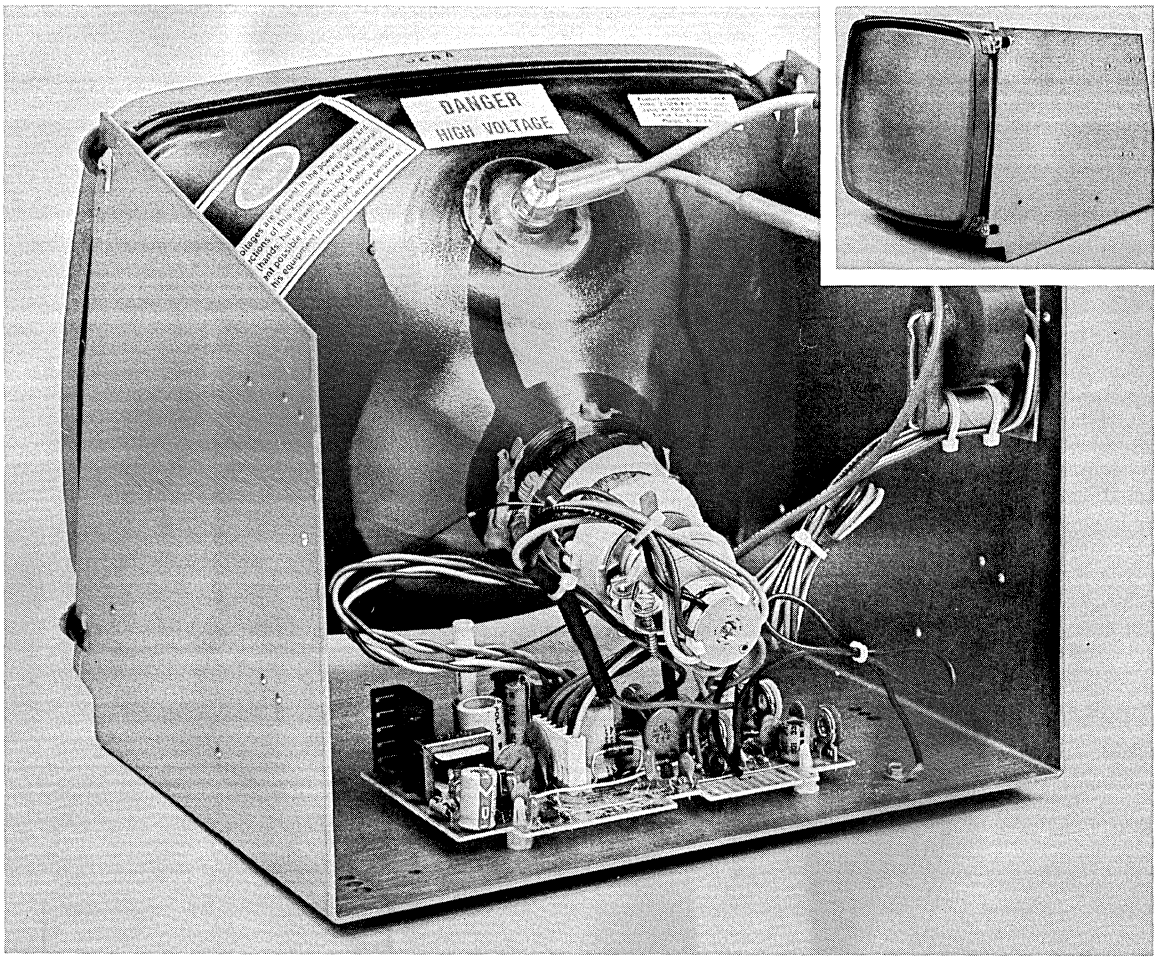


Figure 5 - SuperBrain CRT Display Module
This module is easily removed for service or replacement. A single edge connector is provided for connection to SuperBrain's Power Supply Module.

## INTERNAL CONSTRUCTION (continued)

### MAIN POWER SUPPLY MODULE

The SuperBrain's power supply is a "solid-state, switching" design and employs switching voltage regulators to provide many years of trouble-free service. This design reduces heat dissipation and allows for efficient cooling of the entire terminal with a specially designed whisper fan to reduce environment noise. The entire power supply can be easily removed by unscrewing the three screws holding it into the base of the terminal. Included on the main power supply module are the power off/on switch, the user brightness control and the main and auxiliary RS232 serial ports. By combining the power supply section and external serial communications connections on the same module, the total module count is able to be kept to a minimum thus greatly facilitating ease of field service repair while at the same time minimizing the number of modules required to be stocked to effect competent field repair (refer to figure six).

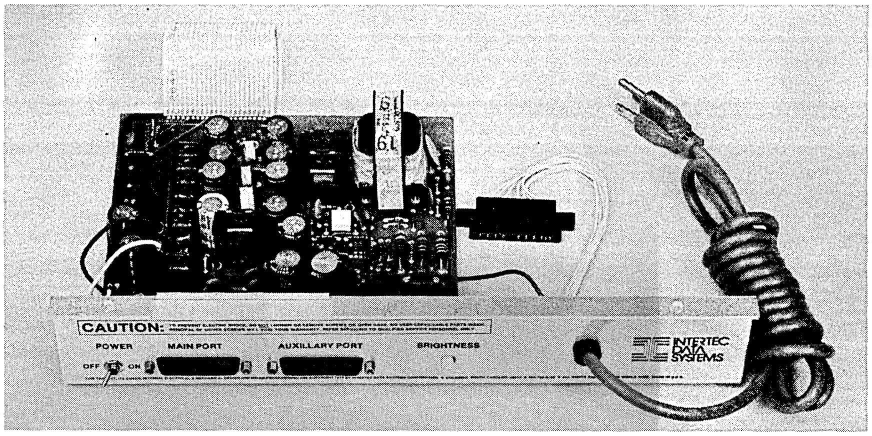


Figure 6 - Main Power Supply

## INTERNAL CONSTRUCTION (continued)

### DISK DRIVE MODULES

Figures seven and eight illustrate the left and right views of the SuperBrain's specially designed double-density disk drive subassembly. Each SuperBrain contains two of these type drives which are mounted conveniently just to the right of the CRT display module on a rugged aluminum mounting bracket which supports the drives so that they are flush mounted with the front "bezel" of the unit. Power to these drives is derived from the Power Supply Module located just behind the drive assemblies themselves. Data to and from these drives is routed via a single 34 pin ribbon cable connecting the drives to the Keyboard/CPU module.

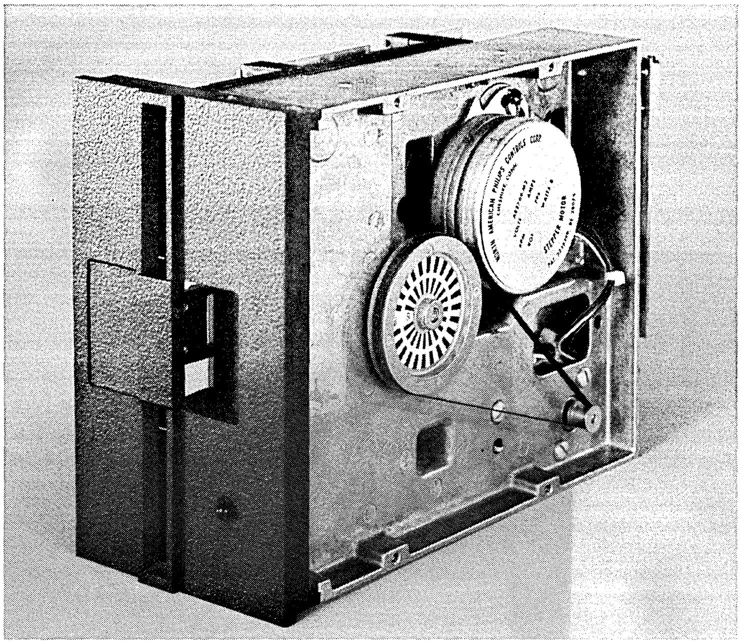


Figure 7 - Top View of SuperBrain Drive Assembly

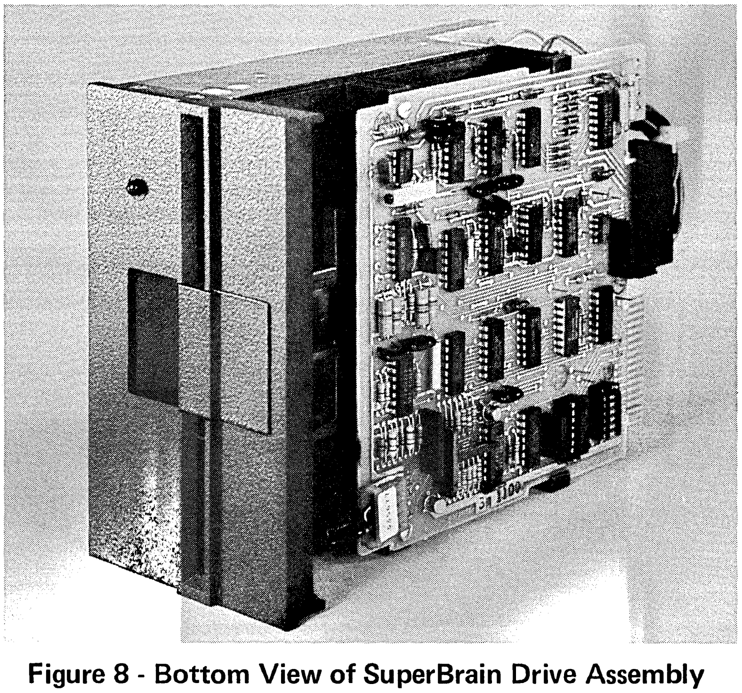


Figure 8 - Bottom View of SuperBrain Drive Assembly

## INTERNAL CONSTRUCTION (continued)

The SuperBrain can be configured to employ an optional module - the S-100 bus adaptor. This adaptor plugs into the SuperBrain's Keyboard/CPU module and mounts internally on the metal bracket supporting the disk drive assemblies. Figure nine shows the SuperBrain with the S-100 bus adaptor and a single S-100 printed circuit card. Figure ten shows the same unit without the S-100 bus module installed.

The S-100 bus adaptor is offered as an optional feature on the SuperBrain for those users who desire to expand the units' capability with the addition of auxiliary disk devices including the new, more popular Winchester-type drives.

A single S-100 card can be easily inserted in the card guide supplied with each S-100 bus adaptor (as shown in figure eleven). NOTE: The S-100 bus adaptor includes cabling, connectors and circuitry to convert the SuperBrain's Z80 data bus into the S-100 bus. The actual S-100 compatible printed circuit board (as is shown in figure eleven) is supplied by the user.

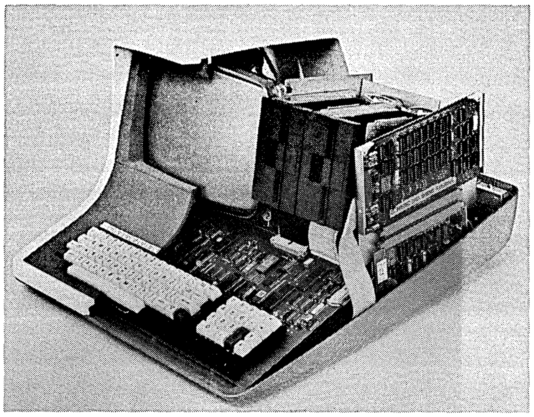


**Figure 9 - SuperBrain with S-100 Bus Adaptor and card installed.**

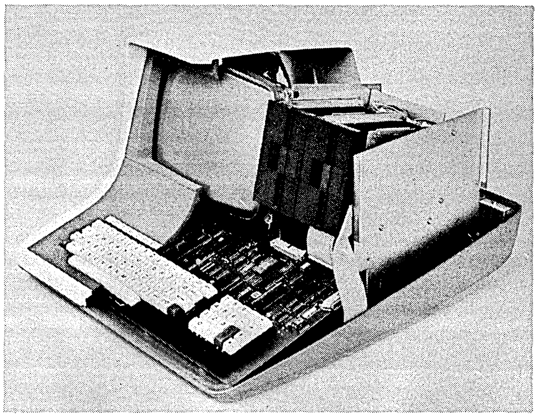


**Figure 10 - SuperBrain with S-100 Bus Adaptor and card removed.**

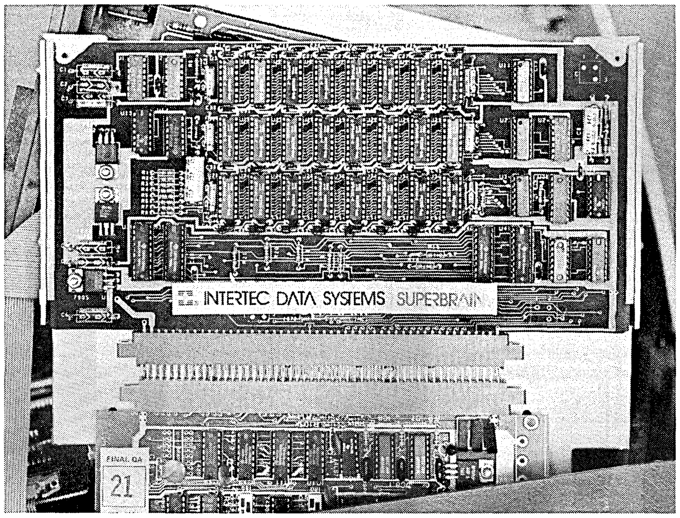


**Figure 11 - SuperBrain S-100 Bus Adaptor**
**Includes adaptor, 100 pin S-100 connector, card guides, mounting bracket and all necessary cabling. The S-100 card is supplied by the user.**

3

# SYSTEM OPERATION

# THEORY OF OPERATION

The SuperBrain contains two Z80 microprocessors. (Reference Figure 3-1) uP1 is the master processor. It communicates with the 64K RAM and the I/O devices (serial port, keyboard encoder, interface controller, and CRT controller). Aside from these devices, it can also access the 2K ROM and DATA BUFFER RAM in the FLOPPY DISK CONTROLLER. uP2 is slaved to uP1 and can only access the 2K ROM, DATA BUFFER, and the DISK INTERFACE. This processor is used exclusively for disk control.

The 32/64 kilobyte main memory consists of up to thirty-two 16K x 1 bit dynamic RAMS. These are divided in four banks (0-3) with each bank containing 16 kilobytes of storage. The RAS-CAS timing sequence necessary for memory access is created by the memory timing generator.

There are two devices that can access memory - uP1 and the CRT Controller. uP1 can read and write to memory while the CRT Controller can only perform the read function. Because each device runs at a different speed, two clock frequencies are required for memory timing. The speed is determined by the selection of the control input to the timing generator. The microprocessor functions require the faster clock.

The CRT-VIDEO CONTROLLER contains three main devices - the CRT Controller which generates all the timing signals for data display; the video generator which produces the character font; and the octal 80-bit shift register which stores one row of video data. (80 characters)

The CRT Controller generates all the timing necessary to display 24 rows of characters with 80 characters per row. Thus the screen can display a total of 1920 characters. These characters are stored in the CRT refresh buffer which is the upper 2048 bytes (2K) of RAM.

Because the CRT buffer is not a separate buffer and the processor must also use the same bus to access memory, this bus must be timeshared between the two. This is accomplished by the CRT controller performing a direct memory access (DMA) cycle which is done at the beginning of each scan row. Each scan row is divided into ten scan lines, therefore during the first scan line time, the controller takes control of the processor bus by generating a bus request. After acquiring the bus, it reads 80 characters from the CRT buffer and loads them into the 80 x 8 shift register. This data is then recirculated in the buffer for the next nine scan lines to produce one row of video characters. Therefore, there are twenty-four DMA cycles performed per vertical frame.

There are also twenty-five interrupts generated - one for each row scan and one extra during vertical blanking. During the first twenty-four, the processor sets or resets the video blanking depending on whether that row is displayed or not. During the vertical blanking interrupt, the address registers in the CRT controller are initialized to the correct top-of-page address and the cursor register is also updated.

The Interface Controller is basically three 8 bit I/O ports (8255). Through this device, the processor can obtain status bits from other devices and react to the status by setting/resetting individual bits in the 8255.

The Keyboard Encoder scans the keyboard for a key depression, determines its position, and generates the correct ASCII code for the key. The processor is flagged by the 'Data Ready' signal via the Interface Controller. The character is then input by the processor.

## THEORY OF OPERATION (continued)

The remaining I/O device is the RS-232-C Serial Interface Port. Presently, it operates only in the asynchronous mode and adheres to a simplified standard protocol. The baud rate is set to 1200 baud by the operating system (Refer to the Technical Bulletin enclosed at the end of this manual.)

As previously mentioned, uP1 has the capability of communicating with the RAM and ROM in the FLOPPY DISK CONTROLLER. It does this to obtain the bootloader from ROM on power-up and system reset and also when transferring disk parameters and data to/from the Data Buffer RAM. Because the amount of main memory used is the maximum that the processor addressing can support different 16K banks of main memory must be switched off line when communicating with the disk RAM or ROM. In these cases Bank 0 (0000H-3FFFH) is switched out when communicating with the ROM, and Bank 2 (8000H-BFFFH) when communicating with the RAM.

The DISK CONTROLLER performs all disk related I/O functions upon command from the main processor. These commands are:

- Restore to track Ø
- Read sector
- Write sector
- Write sector with deleted data mark
- Format

The parameters associated with drive, side, track, and sector numbers are loaded, a status word is set at specified location in the disk RAM. When uP2 receives this status, it sets the 'disk busy' status bit and performs the indicated function. Upon completion, it resets the 'busy' bit thus allowing the main processor (uP1) to retrieve data and status from the RAM.

## GENERAL SPECIFICATIONS

| | |
|---|---|
| POWER | 110/220 VAC 50/60 HZ<br>Dual Switching Power Supplies |
| MEMORY | 32/64K bytes (dynamic) |
| MICROPROCESSOR | Two Z80's operating at 4MHZ |
| SERIAL PORTS | Two asynchronous 'simplified' RS-232-C, programmable ports |
| CRT SCREEN | 24 lines, 80 columns<br>7 x 10 dot character field<br>5 x 7 dot character font<br>50/60 HZ refresh rate |
| FLOPPY DISKS | Two, 5-1/4", double density, MFM<br>Format (Soft sectored) - 512 Bytes/sector; 10 sectors/track<br>35/70 tracks/diskette<br>Capacity - 179K bytes formatted single sided, 35 tracks/diskette<br>358K bytes formatted single sided, 70 tracks/diskette |
| DOS | CP/M, Version 2.2 |

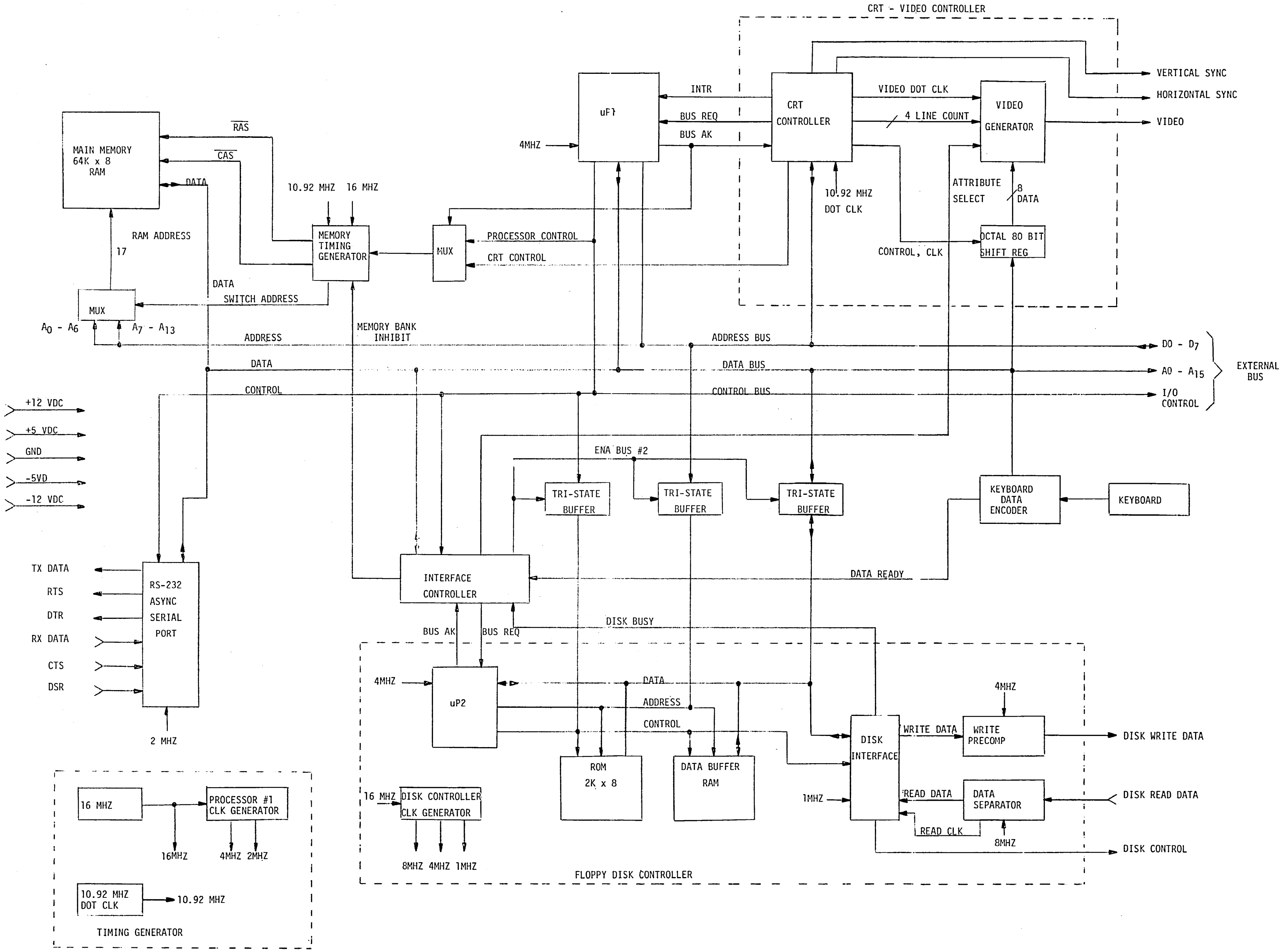


**FIGURE 3-1 SUPERBRAIN KEYBOARD/CPU MODULE BLOCK DIAGRAM**

# INSTALLATION AND OPERATING INSTRUCTIONS

## UNPACKING INSTRUCTIONS

Be sure to use extreme care when unpacking your SuperBrain Video Computer System. The unit should be unpacked with the arrows on the outside facing up. Once you have opened the unit, locate the Operator's Manual which should be placed at the front of the terminal.

If you have ordered additional optional software with your system, it will most likely be attached to the outside of the carton in a gray envelope. Extreme care should be used in opening this envelope so as not to damage any of the delicate diskette media contained inside. The MASTER SYSTEM DISKETTE is located inside the front cover of the Operator's Manual. Be careful not to discard or misplace this diskette as it will be vital for the operation of the equipment in later sections.

Now that you have located your Operator's Manual and system diskette you can proceed to remove all packing material on the top and front of the terminal. Once this has been accomplished, you may now remove the terminal from the shipping carton. In some instances, you may notice that the terminal is somewhat difficult to remove from the carton. This is due to the varying amounts of packing material that is placed in each carton. If you should experience such difficulties, rotate the carton on its side. With the terminal on its side, you should now be able to pull outward on the terminal and separate it from the box. Once the terminal is out of the carton place it on a table and remove the protective plastic bag which should be surrounding the terminal. DO NOT DISCARD THE SHIPPING CARTON UNTIL YOU HAVE COMPLETELY CHECKED OUT THE TERMINAL.

## SET UP

Now that you have removed your SuperBrain Video Computer System from its packing carton, you are ready to begin to set up the system. The first step in this procedure is to verify that your SuperBrain Video Computer System is wired for a line voltage that is available in your area. This can be ascertained by looking on the serial tag located at the right rear of the terminal. This tag should indicate that your unit is set up for either 110 or for a 220 VAC operation. DO NOT ATTEMPT TO CONNECT THE SUPERBRAIN VIDEO COMPUTER SYSTEM TO YOUR LOCAL POWER OUTLET UNLESS THE VOLTAGE AT YOUR OUTLET IS IDENTICAL TO THE ONE SPECIFIED ON THE BACK OF YOUR TERMINAL. Should the voltages differ, contact your dealer at once and do not proceed to connect the SuperBrain Video Computer System to the power outlet.

Before connecting the SuperBrain Video Computer System to the wall outlet, be sure that the power switch located at the left rear corner is turned OFF. You may now proceed to connect your computer system to the wall outlet. After completing this connection, turn the power switch to the 'ON' position. At this time, you should hear a faint "whirring" sound coming from the fan in the computer. After approximately 60 seconds the message 'INSERT DISKETTE INTO DRIVE A' will appear on the screen. If this message does not appear on the screen after approximately 60 seconds, depress the RED key located on the upper right hand corner of the numeric key pad. This key is the master system reset key and should reinitialize the computer system thereby displaying the 'INSERT' message on the screen. If, after several attempts at resetting the equipment you are unable to get this message to appear on the screen, turn the unit off for approximately 3 to 5 minutes and then reapply power to the unit. If you are still unable to get the appropriate message to appear on the screen, contact your Intertec representative.

## SYSTEM DISKETTE

Now that you have power applied to the machine and the 'INSERT DISKETTE' message has been displayed in the upper left hand corner, you are ready to proceed with loading the computer's operating system. This is accomplished by locating the small 5¼" diskette that was packed with the operator's manual. Once you have located this diskette you will notice

## INSTALLATION AND OPERATING INSTRUCTIONS (continued)

that a small adhesive aluminum strip has been placed over the notch on the right hand side of the diskette. This aluminum strip is used to "WRITE PROTECT" the diskette. Therefore, you may only load and/or read programs off of this diskette. If you wish to write or save programs on the system diskette it will be necessary to remove the small adhesive aluminum strip from the diskette. This is NOT RECOMMENDED as it will subject your diskette to accidental errors that may be induced by you while you are getting familiar with the operating system.

You are now ready to proceed with inserting the system diskette into the machine. When facing the front of the machine, you will notice that there are two small openings on the right-hand side of the machine. The first opening (the one furtherest to the left) is designated as DRIVE A. The second opening (the one on the right-hand side of the terminal) is designated as DRIVE B. This distinction is extremely important since the disk operating system can only be loaded from DRIVE A.

Now that you have located the two disk drives on the system, open the disk drive door on DRIVE A (opening closest to your left). The drive can be opened by applying a very slight pressure outward on the small flat door located in the center of the opening. Once the Drive door has been opened, you are now ready to insert the Operating System Diskette. As noted previously, this is the diskette which was packed with your Operator's Manual. The front of the diskette should contain a small white sticker located in the upper left hand corner of the diskette. This diskette should contain a message indicating that it is the SuperBrain DOS Diskette with CP/M Version 2.0. Once you have located this diskette you may insert it into the machine. Be careful to insure that (1) the small aluminum write protect strip is orientated towards the top edge of the diskette and that (2) the label located in the upper left hand corner of the operating system diskette is facing AWAY from the screen towards the right-hand side of the terminal. Once you have orientated the diskette in this fashion, you may now insert it into the terminal. It is **EXTREMELY** important that the diskette be properly orientated before inserting it into the machine since improper orientation will not allow the operating system to properly load. Once the diskette has been placed in the machine, be sure that it has been inserted all the way by applying a gentle pressure on the rear edge of the diskette. Once you are certain that the diskette is fully inserted, you may close the disk drive door. This can be accomplished by applying a slight pressure on the door pulling it back into the direction from which it was originally opened. Once you have closed the door, you will notice a small "swishing" sound. This sound is normal and indicates that the computer is now attempting to load the operating system. Some drives are quieter than others and therefore this noise may not be audible in some cases.

After closing the door the following message should appear in the upper left-hand corner of the screen:

```
XXK SUPERBRAIN DOS VER X.X
A>
```

If this message does not appear on the screen, try depressing the two RED keys located on either side of the keyboard. This should reset the terminal and thereby attempt to reload the operating system. If after several seconds, the message does not appear on the screen, try depressing the RED keys several more times. If repeated depressions of the RED keys do not bring up the indicated message, then open the door on the disk drive A and remove the system diskette and check to see if it was properly inserted. It is extremely important that the diskette be in the proper orientation before attempting to load the operating system. If you are unsure as to the proper orientation of the diskette, please contact the representative from whom you originally purchased your equipment.

## INSTALLATION AND OPERATING INSTRUCTIONS (continued)

After you have checked the orientation of the diskette try reinserting it into DRIVE A (do **NOT** insert the system diskette into DRIVE B as it will not load from DRIVE B). Once the diskette has been reinserted, close the door on DRIVE A and depress the RED key. If after several repeated depressions of the RED keys the message XXK SUPERBRAIN DOS VER X.X does not appear on the terminal then contact your dealer.

### REVIEWING THE SYSTEM DISKETTE

Now that you have successfully loaded the System Diskette and Disk Operating System, (DOS), the SuperBrain is ready to accept your disk operating system commands. At this time we will review several of the commands in the operating system. However, it is recommended that you refer to the appropriate section in this Manual for a detailed description of all such commands (Section 4 - Introduction to CP/M Features and Facilities). The most used system command is the DIR command. This command directs the operating system to display the directory of all programs contained on the system diskette. You may enter this command by simply typing the letters DIR on the keyboard. After you have typed these letters, it is necessary to depress the RETURN key. Depressing this key instructs the computer to process the line of data that you have just typed. After you depress the RETURN key the computer should respond by displaying all of the programs on the system diskette. These programs will appear in the following form:

```
A: ED.COM           A: SYSGEN:COM
A: DDT.COM          A: PIP.COM
A: ASM.COM          A: STAT.COM
A: LOAD.COM         A: SUBMIT.COM
A: DUMP.COM
```

To obtain a better understanding of just what this information means, lets take a look at the first line:

```
A: ED.COM
```

The first letter on this line is a letter A. This tells you that the information following this letter is located on DRIVE A. The colon serves as a separator between the Drive designator ("A") and the file NAME and file TYPE. The file NAME is, in this case, "ED" and the file TYPE is "COM". As such, this line tells the operator that a program called ED (the disk operating system text editor) is located on the "A" drive and is a COM type of file. A more detailed treatment of this information can be found in section 4 of this manual.

**IMPORTANT NOTE:** Some of the disk utility programs have a two digit number suffixed to the File name (i.e. PIP 22). This suffix is used to indicate the actual revision and/or version level of the program.

### DUPLICATING THE OPERATING DISKETTE

Now that you have successfully loaded the Disk Operating System on Drive A, it is important to duplicate this diskette onto another disk. This is necessary in order to preserve the original copy of the diskette and guard against any possible damage to the original media. To generate a copy of the operating system you will first need a NEW BLANK DISKETTE. We recommend an Intertec 1121010 diskette for this purpose. If you do not have any blank diskettes of similar quality, please contact the representative from whom you purchased your equipment. He should be able to supply you with an ample quantity of these diskettes.

Once you have located a new blank diskette, insert it into DRIVE B. Follow the procedures outlined in the previous paragraphs regarding the insertion of the operating system diskette. The only difference is that you will be inserting the new blank diskette into DRIVE B. Be sure and leave the system diskette installed on DRIVE A.

## INSTALLATION AND OPERATING INSTRUCTIONS (continued)

Once you have installed the new blank diskette on DRIVE B, you are now ready to "FORMAT" the new diskette. It is necessary to format all new previously unused diskettes before attempting to transfer data to them. This is necessary because all information is stored on diskettes in what is known as the SOFT SECTORED FORMAT which necessitates the writing of certain information on the disks before user programs can be stored on them.

To format the disk in DRIVE B enter the command 'FORMAT' at the keyboard. Remember to depress the key marked RETURN after typing the words FORMAT. The operating system should now respond by asking you to select the type of diskette being formatted (S or D). This question asks whether the diskette to be formatted is single sided or double sided. Unless you have ordered our new Quad Density SuperBrain QD, the response to this question should be the letter "S" indicating a single sided diskette. After entering the 'S' depress the RETURN key. The operating system will now ask you whether you have a 64K (6) or 32K (3) disk operating system. In most cases, the answer to this question will be 3 (32K). After you have entered the appropriate response to this question the operating system will respond by telling you to place a blank diskette on DRIVE B. Since this has already been done, we are now ready to proceed with formatting the diskette and may do so by entering the letter "F". At this point and time you will hear the disk drive reset to track 0 and begin the formatting process. When a disk is formatted the read/write head positions to track 0 and rewrites each track (there are a total of 35 on each diskette). The screen will also display the current track which is being formatted. This number should range from 0 to 34 for a total of 35 tracks.

After the disk has been completely formatted, the operating system will respond by asking you whether to "REBOOT" the operating system or whether you wish to format another disk. If you wish to format another disk, remove the newly formatted disk from **DRIVE B** and insert a new blank diskette into **DRIVE B.** You may now proceed to format this new diskette by once again entering the letter "F". If you do not wish to format any more diskettes, simply enter a RETURN.

The Operating System should now reload and once again be ready to accept new commands.

Since the intent of this procedure was to copy the original disk operating system we are now ready to begin that procedure. This can be accomplished by entering the following command on the keyboard:

```
   22
   v
PIP B: =*.*
```

After you have entered the above command at the keyboard depress the return key.

The system will now begin to copy all of the programs on DRIVE A over to DRIVE B. As each program is copied, its name will be displayed on the screen. This procedure takes approximately 5 to 10 minutes. After the procedure completes, the control of the operating system will be returned to the user.

Now that you have completed copying the operating system's programs from the A DRIVE to the B DRIVE it is necessary to copy the disk operating system itself (which is located on tracks 0, 1 and 2) onto the DRIVE B. This may be accomplished by entering the following command at the keyboard:

```
SYSGEN22
```

The SYSGEN command is used to generate an operating system and place it on the desired

## INSTALLATION AND OPERATING INSTRUCTIONS (continued)

disk. Once you have entered this command at the keyboard and typed RETURN, the disk operating system will ask you to select which drive that you want to take the source from. The correct answer to this question is the letter "A". After entering "A" depress the RETURN.

The next question the program will ask is where do you want the source to be placed (the destination drive). The correct answer to this is the letter "B" indicating DRIVE B. Once you have entered this, the operating system will be copied from DRIVE A onto DRIVE B.

After this process has been completed the operating system will ask you whether you wish to duplicate another copy or to reload the operating system. The correct response is to simply enter a RETURN which will reload the operating system.

Once the operating system has been reloaded, you may now remove the master disk operating system in DRIVE A. Once this disk has been removed store it in a safe place as you may need it later to generate additional copies of the disk operating system and its programs.

At this point you should have removed the master disk from DRIVE A. Now remove the copy from DRIVE B and reinstall it on DRIVE A and close the door on DRIVE A. After you have completed this, depress the RED reset keys located on either side of the keyboard. This will reset the machine and reload the newly installed operating system off of your new diskette.

**IMPORTANT:** If random garbled information is displayed on the screen at this time, this indicates that you have made an error in the use of the "SYSGEN" program. If this is indeed the case, then remove the new diskette from DRIVE A and reinstall the original master system diskette and repeat the previously outlined procedure for generating a new disk operating system. If you still encounter difficulties, please refer to Section 4 of this manual for more detailed information concerning this procedure.

Now that you have successfully completed the generation of a new system diskette please refer to Section 4 of this manual for a complete description of all of the operating systems utility programs (DDT.COM, PIP.COM, SUBMIT.COM, etc.).

### OPTIONAL SOFTWARE

Numerous optional software packages are available for use with your SuperBrain Video Computer System. Currently available directly from Intertec are such software packages as Microsoft's BASIC, FORTRAN and COBOL. If you would like additional information on these packages please contact your local Intertec representative.

### NEWLY RELEASED SYSTEM PROGRAMS

From time to time, Intertec will be releasing additional 'standard' system programs. Listed below is a brief description of several such programs. A complete description of these and other similar programs can be found in the "software addenda" section of this manual.

FORMAT.COM Allows the user to format blank diskettes. This program must be run on all new diskettes which have not been previously formatted on a SuperBrain Video Computer System. It is important to note that although you may have formatted these diskettes on other systems, this does not necessarily imply that they will work on a SuperBrain unless they have been formatted on a computer of this type. Therefore, in order to insure complete compatibility please format all new diskettes on a SuperBrain Video Computer System before using.

## INSTALLATION AND OPERATING INSTRUCTIONS (continued)

RAMTST.COM This program runs an extensive test on main memory by writing and reading all possible patterns into all locations in the RAM. This program takes approximately 4 to 5 minutes to complete on 32K machines and 8 to 10 minutes on 64K machines. Since different amounts of RAM are contained in the 32 and 64K machines, we have included two RAM test programs. These are: RAMTST32.COM and RAMTST64.COM which are for testing 32 and 64K versions of the SuperBrain Video Computer System. It is important to note that the 64K RAM test program will not execute properly on a 32K machine.

At the end of the RAM test program, the message RAM OK will appear on the screen if the test was completed successfully. If any errors were detected during the test, the computer's bell will turn on and continue in a continuous tone manner until the RED reset key is depressed. If a continuous tone such as this is heard on the computer when executing the RAM test, depress the RED reset and try executing the program several times. If the program continues to produce the audible tone, then please contact the Intertec Service Department.

CONFIGUR.COM This program allows the user to configure all parameters for the RS232 MAIN and AUXILIARY serial port. The selected configuration is then permanently stored on the disk along with the disk operating system. As such, the system will be completely reconfigured each time power is applied to the machine or the RED reset key is depressed.

A complete description of all of these programs can be found in the software addenda section of this manual. In addition to the descriptions contained therein, most newly released system programs will contain a description program along with the actual COM file. This program will be in the form of FILE NAME.DES. As an example of such a program would be 'FORMAT.DES'. This program would contain a description of how the format program operates. Therefore, if you are unable to find an adequate description in the software addenda section of this manual for a program on the disk, please check for a DES version of the program on your disk. If such a program exists, you may display the instructions by simply typing the following command: TYPE FILENAME.DES.

### VIDEO DISPLAY FEATURES AND CONTROL CODES

Various screen control features are available to the operator through the use of 'ESCAPE' sequences. Among these are the following:

Absolute cursor addressing [ESC] [Y] [row] [column] The cursor is positioned to the row and column specified. Refer to the SuperBrain screen layout for specific screen formatting information.

Erase to end of line [ESC] [∽] [K] Data is erased from cursor position to the end of the current line.

Erase to end of page [ESC] [∽] [k] Data is erased from cursor position to the end of the screen.

Display control characters [ESC] [∽] [E] Enable transparent mode. Control characters received are displayed on the screen and are not executed.

## INSTALLATION AND OPERATING INSTRUCTIONS (continued)

Disable control character display [ESC] [∽] [D] Disable the transparent mode.

Other features are also available using the 'CONTROL' key. They are the following:

| | |
|---|---|
| CONTROL [A] | - Home cursor (Row 1, Column 1) |
| CONTROL [F] | - Cursor forward |
| CONTROL [G] | - Ring Bell |
| CONTROL [I] | - Tab |
| CONTROL [K] | - Cursor Up |
| CONTROL [L] | - Clear Screen |
| CONTROL [U] | - Cursor Back |

### MASTER RESET FEATURE

A Master Reset of all terminal hardware may be accomplished by depressing the solid colored RED key located on the upper right hand corner of the numeric keypad. It is important to note that on some versions of the SuperBrain, this reset feature may involve the depression of two RED keys. If this is the case on your computer system, you will notice that the two RED keys are located on the right and left corners of the alphameric section of the keyboard.

### CURSOR CONTROL KEYS

There are three to four cursor control keys located on every SuperBrain Video Computer System. These keys are located on the right-hand side of the numeric keypad. If your computer has a single RED key (keyboard layout A), it will be located in the upper right hand corner of the numeric keypad thereby leaving only three cursor position keys. If your computer is configured with two RED keys (keyboard layout B - one RED key located on each side of the alphanumeric keyboard cluster), then you will have a total of four cursor position keys on the right hand side of the numeric keypad. In either case, these keys will transmit codes to any program running on the SuperBrain. These codes may in-turn be interpreted by the program to result in cursor movement on the screen. It is important to know that these keys will not produce cursor movement when you are in the operating system mode. The reason for this is that CP/M does not define any use of cursor positioning on the screen. As such, depression of these keys while in the operating system mode will result in the control codes assigned to the individual keys being displayed as control codes on the screen.

# INTERFACING INFORMATION

### RS-232-C Serial Interface

The following chart illustrates the pinouts for the MAIN and AUXILIARY serial ports and the direction of signal flow.

## SUPERBRAIN SERIAL PORT PIN ASSIGNMENTS

(For use with Revision 3.0 DOS software or higher and Keyboard/CPU Module Revision 1.0 or higher)

### MAIN PORT

| PIN # | ASSIGNMENT | DIRECTION |
|---|---|---|
| 1 BLU | GND | — |
| 2 BRN | Transmitted Data | (From SB) |
| 3 BLK | Received Data | (To SB) |
| 4 PNK | Request to Send | (From SB) |
| 5 GRY | Clear to send | (To SB) |
| 6 ORG | Data Set Ready | (To SB) |
| 7 WHT | GND | — |
| 15 GRN | Transmit Clock | (To SB) |
| 17 YEL | Receive Clock | (To SB) |
| 20 RED | Data Terminal Ready | (From SB) |
| 22 PRPL | Ring Indicator | (To SB) |
| 24 LBRN | Clock | (From SB) |

### AUXILIARY PORT

| PIN # | ASSIGNMENT | DIRECTION |
|---|---|---|
| 1 | GND | — |
| 2 | Received Data | (To SB) |
| 3 | Transmitted Data | (From SB) |
| 7 | GND | — |
| 20 | Data Terminal Ready | (To SB) |

### Bus Adaptor Interface

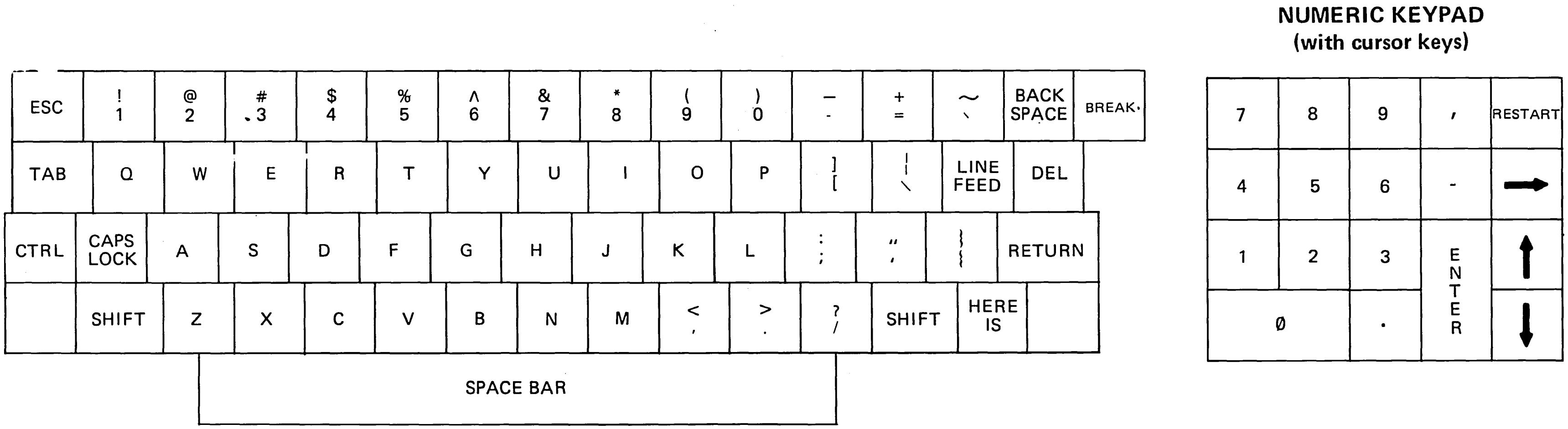
The SuperBrain contains a Z80 bus interface to the main processor bus. These signals are shown in the chart on the following page.

When using this interface, it is recommended that all signals be buffered so as not to excessively load the main processor bus. The external bus should **ONLY** be utilized for **I/O** devices using addresses 80H to FFH. Memory mapped I/O is **NOT** possible since the SuperBrain is internally configured for 64K of RAM.

## PIN CONNECTIONS FOR EXTERNAL BUS

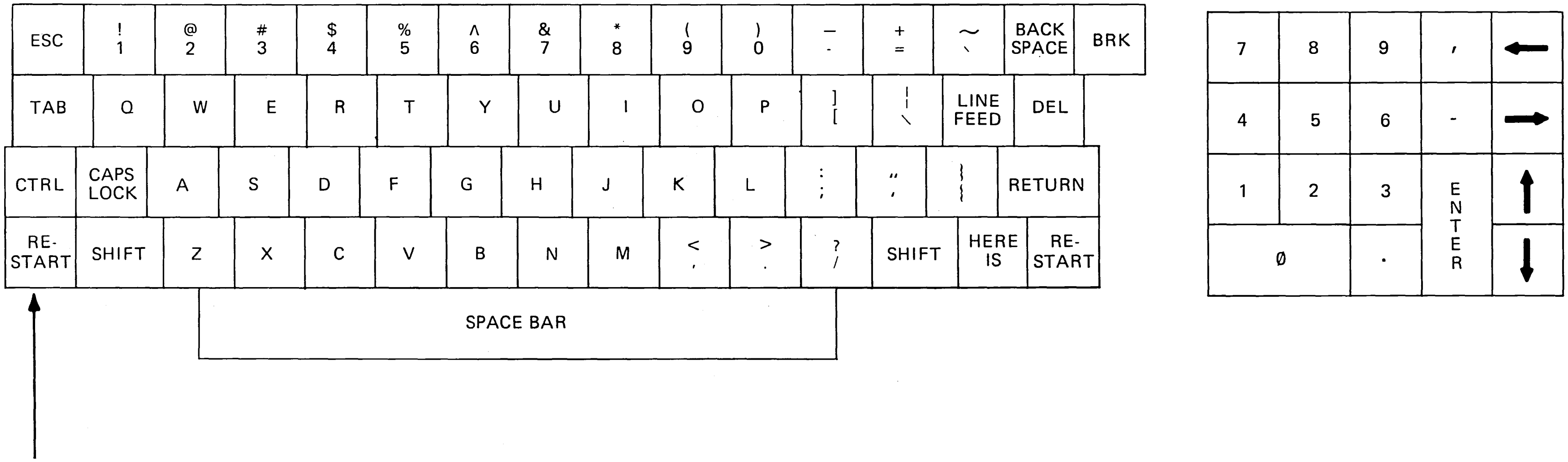
| P/N | SIGNAL NAME | DESCRIPTION |
|---|---|---|
| 1 | | SPARE |
| 2 | SYSRES* | System Reset Output, Low During Power Up Initialize or Reset Depressed |
| 3 | | SPARE |
| 4 | A1∅ | Address Output |
| 5 | A12 | Address Output |
| 6 | A13 | Address Output |
| 7 | A15 | Address Output |
| 8 | GND | Signal Ground |
| 9 | A11 | Address Output |
| 10 | A14 | Address Output |
| 11 | A8 | Address Output |
| 12 | OUT* | Peripheral Write Strobe Output |
| 13 | WR* | Memory Write Strobe Output |
| 14 | | SPARE |
| 15 | RD* | Memory Read Strobe Output |
| 16 | | SPARE |
| 17 | A9 | Address Output |
| 18 | D4 | Bidirectional Data Bus |
| 19 | IN* | Peripheral Read Strobe Output |
| 20 | D7 | Bidirectional Data Bus |
| 21 | | SPARE |
| 22 | D1 | Bidirectional Data Bus |
| 23 | | SPARE |
| 24 | D6 | Bidirectional Data Bus |
| 25 | A∅ | Address Output |
| 26 | D3 | Bidirectional Data Bus |
| 27 | A1 | Address Output |
| 28 | D5 | Bidirectional Data Bus |
| 29 | GND | Signal Ground |
| 30 | D∅ | Bidirectional Data Bus |
| 31 | A4 | Address Bus |
| 32 | D2 | Bidirectional Data Bus |
| 33 | | SPARE |
| 34 | A3 | Address Output |
| 35 | A5 | Address Output |
| 36 | A7 | Address Output |
| 37 | GND | Signal Ground |
| 38 | A6 | Address Output |
| 39 | +5V | 5 Volt Output (Limited Current) |
| 40 | A2 | Address Output |
| NOTE: * implies negative (Logical "∅") true, Input or Output | | |

**Connection points for External Bus**



## SUPERBRAIN KEYBOARD LAYOUT A

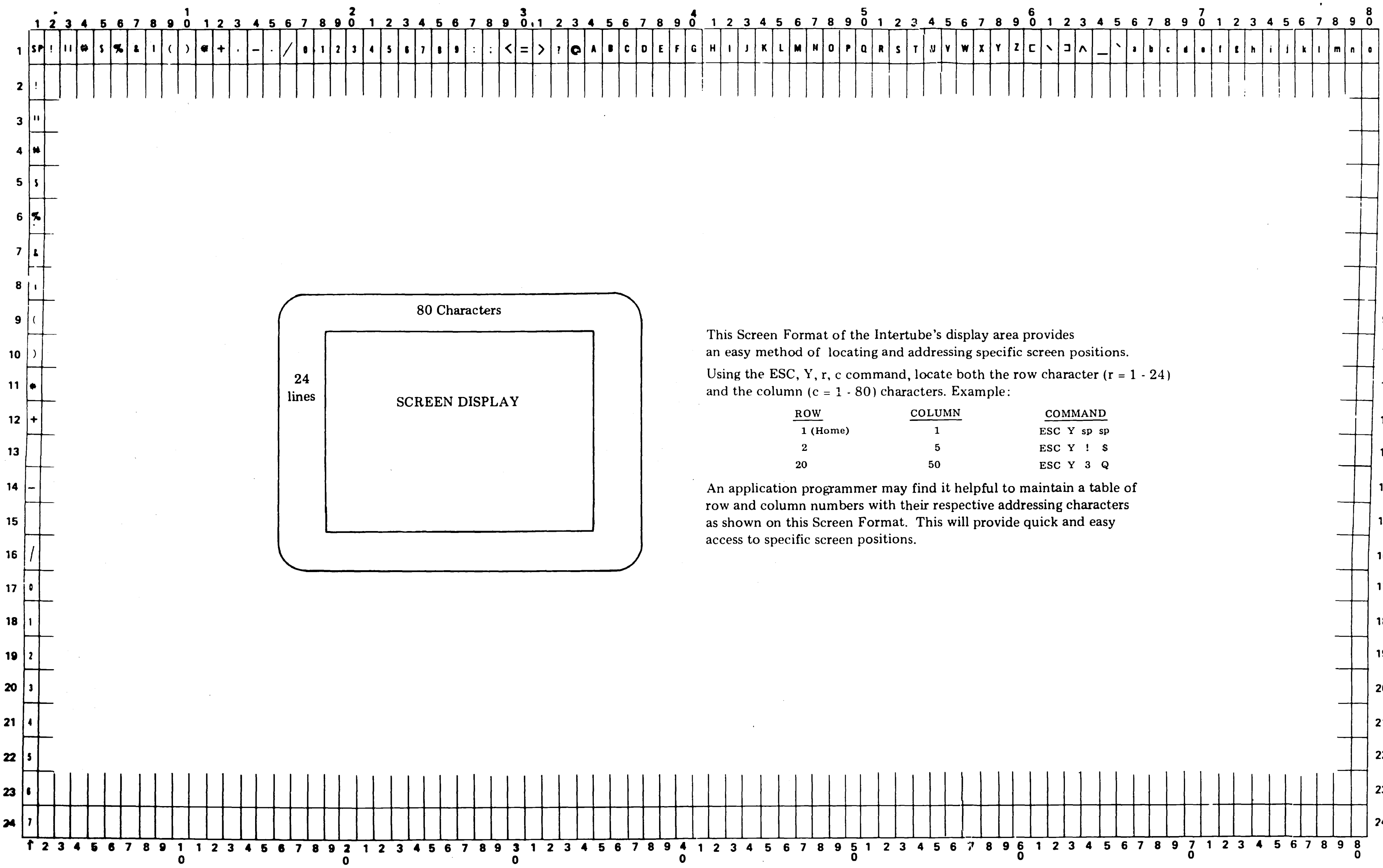
NUMERIC KEYPAD
(with cursor keys)



Special "re-start" sequence key used in conjunction with other re-start key on right side of keyboard will re-load SuperBrain's Disk Operating System. A two-key re-start sequence is used to minimize chance of operator error when system is in operation. Both keys must be depressed simultaneously to reload the operating system.

SUPERBRAIN KEYBOARD LAYOUT B

## SUPERBRAIN SCREEN LAYOUT



SP ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ ` a b c d e f g h i j k l m n o
80 Characters
24 lines
SCREEN DISPLAY

This Screen Format of the Intertube's display area provides an easy method of locating and addressing specific screen positions.

Using the ESC, Y, r, c command, locate both the row character (r = 1 - 24) and the column (c = 1 - 80) characters. Example:

| ROW | COLUMN | COMMAND |
|---|---|---|
| 1 (Home) | 1 | ESC Y sp sp |
| 2 | 5 | ESC Y ! $ |
| 20 | 50 | ESC Y 3 Q |

An application programmer may find it helpful to maintain a table of row and column numbers with their respective addressing characters as shown on this Screen Format. This will provide quick and easy access to specific screen positions.

## INTERPRETING THE ASCII CODE CHART

The figure below illustrates a conventionally arranged ASCII code chart divided into three sections corresponding to control codes (columns 0 and 1) upper case characters (columns 2, 3, 4, and 5), and lower case characters (columns 4 and 5).

| b7 b6 b5 → | | | | | 0 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1 0 0 | 1 0 1 | 1 1 0 | 1 1 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bits | b4 ↓ | b3 ↓ | b2 ↓ | b1 ↓ | column → row ↓ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | 0 | 0 | 0 | 0 | 0 | NUL | DLE | SP | 0 | @ | P | ` | p |
| | 0 | 0 | 0 | 1 | 1 | SOH | DCI | ! | I | A | Q | a | q |
| | 0 | 0 | 1 | 0 | 2 | STX | DC2 | " | 2 | B | R | b | r |
| | 0 | 0 | 1 | 1 | 3 | ETX | DC3 | # | 3 | C | S | c | s |
| | 0 | 1 | 0 | 0 | 4 | EOT | DC4 | $ | 4 | D | T | d | t |
| | 0 | 1 | 0 | 1 | 5 | ENQ | NAK | % | 5 | E | U | e | u |
| | 0 | 1 | 1 | 0 | 6 | ACK | SYN | & | 6 | F | V | f | v |
| | 0 | 1 | 1 | 1 | 7 | BEL | ETB | ' | 7 | G | W | g | w |
| | 1 | 0 | 0 | 0 | 8 | BS | CAN | ( | 8 | H | X | h | x |
| | 1 | 0 | 0 | 1 | 9 | HT | EM | ) | 9 | I | Y | i | y |
| | 1 | 0 | 1 | 0 | 10 | LF | SUB | • | : | J | Z | j | z |
| | 1 | 0 | 1 | 1 | 11 | VT | ESC | + | ; | K | [ | k | { |
| | 1 | 1 | 0 | 0 | 12 | FF | FS | , | < | L | \ | L | ¦ |
| | 1 | 1 | 0 | 1 | 13 | CR | GS | — | = | M | ] | m | } |
| | 1 | 1 | 1 | 0 | 14 | SO | RS | . | < | N | ^ | n | ~ |
| | 1 | 1 | 1 | 1 | 15 | SI | US | / | ? | O | — | o | DEL |

Control codes are not displayable unless in the transparent mode. Some of these codes affect the state of the terminal when they are received by the display electronics. For example, the code SOH causes the cursor to go to the home position, and code DC2 turns on the printer port. Codes which have no defined function in the SuperBrain software are ignored if received. The set of 64 upper case alphanumeric characters is sometimes referred to as "compressed ASCII".

If the terminal is set for upper case operation only (CAPS LOCK), lower case alpha characters from the keyboard are automatically translated and displayed as their upper case equivalents (columns 4 and 5). If the DEL code is received, it is ignored. Lower case characters received from the input RS-232C port are displayed as lower case.

The seven-bit binary code for each character is divided into two parts in this chart. A four-bit number represents the four least significant bits (B1, B2, B3, B4) and a three-bit number represents the three most significant bits (B5, B6, B7). The chart above also is divided into 8 columns and 16 rows. This offers two ways of indicating a particular character's code. The character code is indicated as either a seven-bit binary number or as a column/row number in decimal notation. For example, the character M is represented by the binary number 1001101 or the alternative 4/15 notation. Similarly, the control code VT is represented by the code 00001011 or the alternative 0/11 notation.

# INTRODUCTION TO CP/M FEATURES & FACILITIES

DIGITAL RESEARCH

Post Office Box 579, Pacific Grove, California 93950, (408) 649-3896

4

# AN INTRODUCTION TO CP/M FEATURES AND FACILITIES

REVISION OF JANUARY 1978

## Disclaimer

Digital Research 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, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

# Table of Contents

Section Page

4

4

## 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 8Ø8Ø 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 8Ø8Ø (or Zilog Z-8Ø) 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 Digital Research document entitled "CP/M System Alteration Guide." Although the standard Digital Research version operates on a single-density Intel MDS 8ØØ, 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, Intel-compatible assembler, and debugger subsystems. Optional software includes a powerful Intel-compatible macro assembler, symbolic debugger, along with various high-level languages. When coupled with CP/M´s Console Command Processor, the resulting facilities equal or excel similar large computer facilities.

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 (teletype, CRT, Paper Tape Reader/Punch, and user-defined peripherals), and can be tailored by the user for 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:

SEARCH Look for a particular disk file by name.

OPEN Open a file for further operations.

CLOSE Close a file after processing.

RENAME Change the name of a particular file.

READ Read a record from a particular file.

WRITE Write a record onto the disk.

SELECT Select a particular disk drive for further operations.

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:

```
pppppppp.sss
```

where pppppppp 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

```
pppppppp
```

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.C?M
```

is satisfied by the unambiguous file names

```
XYZ.COM
```

and

```
X3Z.CAM
```

Note that the ambiguous reference

```
*.*
```

is equivalent to the ambiguous file reference

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

while

```
          pppppppp.*
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 unambiquous 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<br>"ALL FILES (Y/N)?"<br>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 = B:A.BAK | The file A.BAK is renamed to A.ASM on drive B. |

If the file ufnl 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 1ØØH (hexadecimal), which is the second page of memory. Thus, if the user's program occupies the area from 1ØØH 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 1ØØH through 3FFH to X.COM. |
| SAVE 4Ø Q | Copies 1ØØH through 28FFH to Q (note that 28 is the page count in 28FFH, and that 28H = 2*16+8 = 4Ø decimal). |
| SAVE 4 X.Y | Copies 1ØØH through 4FFH to X.Y. |

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

| | |
|---|---|
| SAVE 1Ø B:ZOT.COM | Copies 1Ø pages (1ØØH through ØAFFH) 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.PLM

        TYPE  XXX
```

The TYPE command expands tabs (clt-I characters), assumming 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:

```
STAT cr
STAT "command line" cr
```

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.

STAT cr

If the user types an empty command line, the STAT transient calculates the storage remaining on all active drives, and prints a message

```
x: R/W, SPACE: nnnK
```

or

```
x: R/O, SPACE: nnnK
```

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 inadvertantly changing diskettes without performing a warm start). The space remaining on the diskette in drive x is given in kilobytes by nnn.

STAT x: cr

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

```
BYTES REMAINING ON B: nnnK
```

STAT afn cr

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

```
RECS BYTS EX D:FILENAME.TYP
rrrr bbbK ee d:pppppppp.sss
```

where rrrr is the number of 128-byte records allocated to the file, bbb is the number of kilobytes allocated to the file (bbb=rrrr*128/1024), ee is the number of 16K extensions (ee=bbb/16), d is the drive name containing the file (A...Z), pppppppp 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) |
| UR1: | 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 |
| UL1: | 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 MDS 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:  UR1:  UR2:
        PUN: = TTY:  PTP:  UP1:  UP2:
        LST: = TTY:  CRT:  LPT:  UL1:
```

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: = UR1:
          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., ldi and pdi 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

4

```
        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

```
          x.COM
```

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 O 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 alphabetics 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 disk 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), UC1: (console)
PTR: (reader), UR1: (reader), UR2: (reader)
PTP: (punch), UP1: (punch), UP2: (punch)
LPT: (list), UL1: (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 4Ø "nulls" (ASCII Ø´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 1Ø3H, with data returned in location 1Ø9H (parity bit must be zero).

USED IN Computer to Computer Transfer.

OUT: Special PIP output destination which can be patched into the PIP program: PIP CALLs location 1Ø6H with data in register C for each character to transmit. Note that locations 1Ø9H 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 6Ø 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 2Ø 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 RDR: 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.

| | |
|---|---|
| PIP LST: = X.PRN cr | Copy X.PRN to the LST device and terminate the PIP program. |
| PIP cr | Start PIP for a sequence of commands (PIP prompts with "*"). |
| *CON:=X.ASM,Y.ASM,Z.ASM cr | Concatenate three ASM files and copy to the CON device. |
| *X.HEX=CON:,Y.HEX,PTR: cr | 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. |
| *cr | Single carriage return stops PIP. |

`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 6Ø 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 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 ":ØØ" record in X.HEX; then continue the transfer of data by reading Y.ZOT, which contains hex records, including any ":ØØ" 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[p5Ø] Send X.ASM to the LST: device, with line numbers, tabs expanded to every eighth column, and page ejects at every 5Øth line. Note that nt8p6Ø is the assumed parameter list for a PRN file; p5Ø 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 5ØØØ 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, ctl-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 ERAse 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.$$$ on drive A. On termination of editing, change X.ASM on drive B to X.BAK, and change X.$$$ 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

```
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

```
$1  $2  $3   ...  $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 parm#1 ... parm#n are paired with the formal parameters $1 ... $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

```
$$$.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 ctl-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

```
ASM $1
DIR $1.*
ERA *.BAK
PIP $2:=$1.PRN
ERA $1.PRN
```

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

```
ASM X
DIR X.*
ERA *.BAK
PIP PRN:=X.PRN
ERA X.PRN
```

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 aat ØØØØH). 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 1ØØH). 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 8ØØ. |

```
        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 CPMxx.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 TØ 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 MOVCPM 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 MOVCPM commands are given below:

MOVCPM 48 cr — Construct a 48K verskon 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 32CPM48.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 Digital Research Software Licensing Agreement.

## 7. BDOS ERROR MESSAGES.

There are three error situations which the Basic Disk Operating System intercepts during file processsing. 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 Ø through 3 on the MDS, corresponding to CP/M drives A through D, respectively. The CP/M system diskette is inserted into drive Ø, 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 Ø 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 8ØØ controller to reformat with sequential sector numbering (1-26) on each track.

Note: "MDS 8ØØ" and "ISIS" are registered trademarks of Intel Corporation.

4

# OPERATION OF
# THE CP/M CONTEXT EDITOR

DIGITAL RESEARCH

Post Office Box 579, Pacific Grove, California 93950, (408) 649-3896

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

## USER'S MANUAL

## Disclaimer

Digital Research 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, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

# Table of Contents

# ED USER'S MANUAL

5

## 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

$$\text{ED}\left\{\begin{array}{l}\text{<filename>}\\ \\ \text{<filename>.<filetype>}\end{array}\right\}$$

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

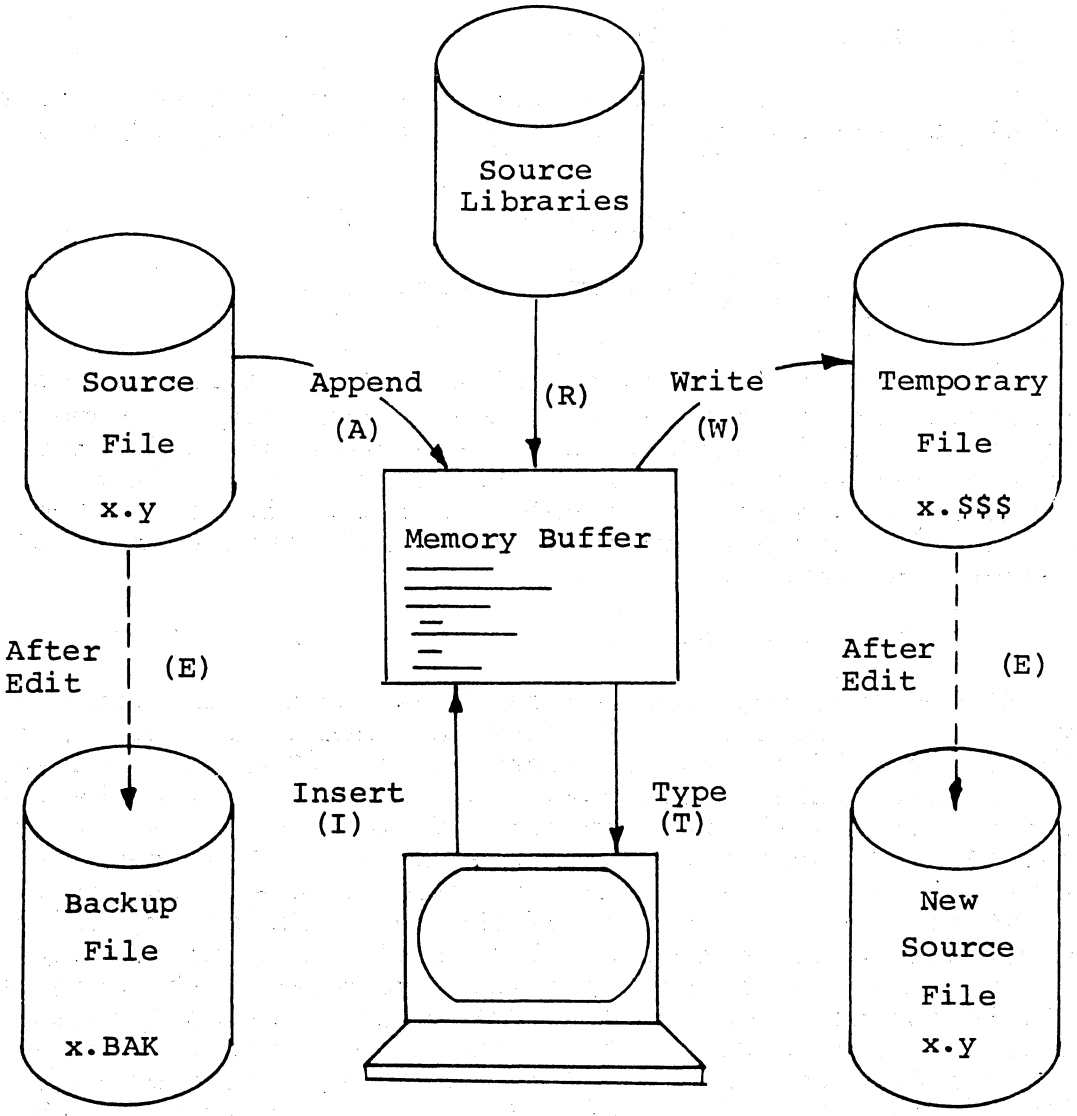
ED operates upon the source file, denoted in Figure 1 by x.y, and passes all text through a memory buffer where the text can be viewed or altered (the number of lines which can be maintained in the memory buffer varies with the line length, but has a total capacity of about 6000 characters in a 16K CP/M 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 alphabetics 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

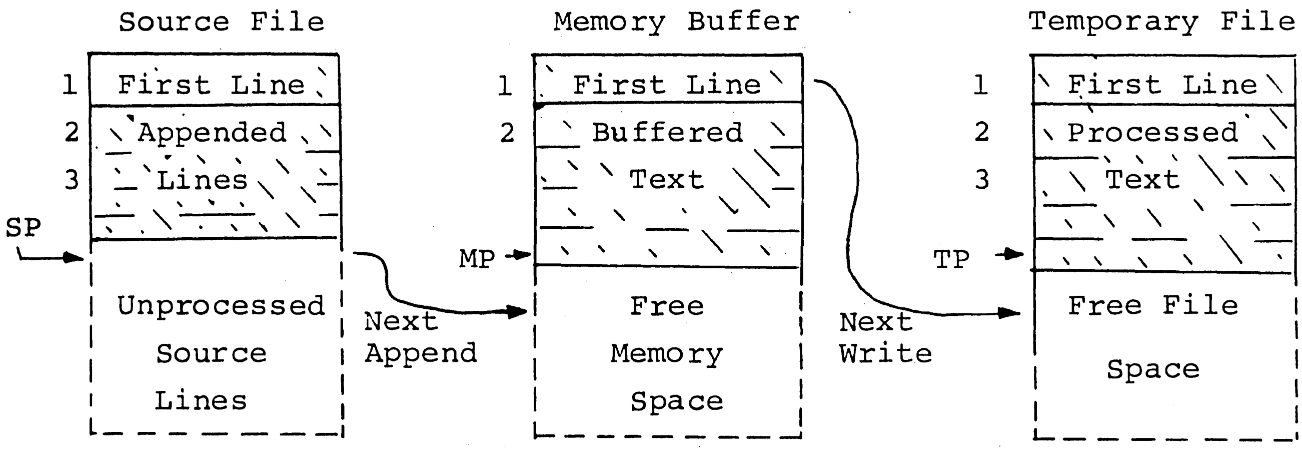
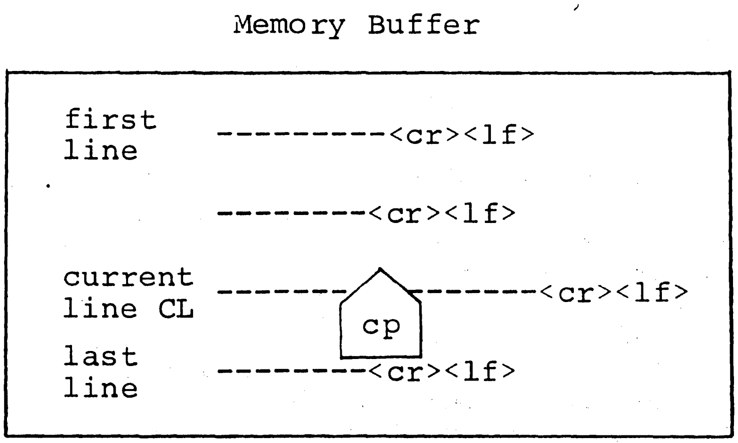


Figure 3. Logical Organization of Memory Buffer



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 id 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 0A fills the current memory buffer to at least half-full, while 0W 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 exceded. 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 $\boxed{cp}$ 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 (ie, 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

```
NOW IS THE<cr><lf>
TIME FOR<cr><lf>
ALL GOOD MEN<cr><lf>^
                    cp
```

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

| Command String | Effect | Resulting Memory Buffer |
|---|---|---|
| 1. B2T<cr> | move to beginning of buffer and type 2 lines: "NOW IS THE TIME FOR" | ^cp NOW IS THE<cr><lf><br>TIME FOR<cr><lf><br>ALL GOOD MEN<cr><lf> |
| 2. 5C0T<cr> | move CP 5 characters and type the beginning of the line "NOW I" | NOW I ^cp S THE<cr><lf> |

|  |  |  |  |
|---|---|---|---|
| 3. | 2L-T<cr> | move two lines down and type previous line "TIME FOR" | NOW IS THE<cr><lf><br>TIME FOR<cr><lf><br>cp ALL GOOD MEN<cr><lf> |
| 4. | -L#K<cr> | move up one line, delte 65535 lines which follow | NOW IS THE<cr><lf> cp |
| 5. | I<cr><br>TIME TO<cr><br>INSERT<cr><br>↑z | insert two lines of text | NOW IS THE<cr><lf><br>TIME TO<cr><lf><br>INSERT<cr><lf> cp |
| 6. | -2L#T<cr> | move up two lines, and type 65535 lines ahead of CP "NOW IS THE" | NOW IS THE<cr><lf> cp<br>TIME TO<cr><lf><br>INSERT<cr><lf> |
| 7. | <cr> | move down one line and type one line "INSERT" | NOW IS THE<cr><lf><br>TIME TO<cr><lf> cp<br>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

$$nF\ c_1c_2 \dots c_k \left\{ \begin{matrix} \text{<cr>} \\ \uparrow z \end{matrix} \right\}$$

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 ↑l (control-l), 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:

| Command String | Effect | Resulting Memory Buffer |
|---|---|---|
| 1. B#T<cr> | move to beginning and type entire buffer | cp NOW IS THE<cr><lf><br>TIME FOR<cr><lf><br>ALL GOOD MEN<cr><lf> |
| 2. FS T<cr> | find the end of the string "S T" | NOW IS T cp HE<cr><lf> |
| 3. FI↑z0TT | find the next "I" and type to the CP then type the remainder of the current line: "TIME FOR" | NOW IS THE<cr><lf><br>TI cp ME FOR<cr><lf><br>ALL GOOD MEN<cr><lf> |

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 $c_1c_2\ldots\ c_n$↑z     or

I $c_1c_2\ldots\ c_n$<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:

| Command String | Effect | Resulting Memory Buffer |
|---|---|---|
| BITHIS IS ↑z<cr> | Insert "THIS IS " at the beginning of the text | THIS IS cp NOW THE <cr><lf><br>TIME FOR<cr><lf><br>ALL GOOD MEN<cr><lf> |

| | | |
|---|---|---|
| FTIME↑z-4DIPLACE↑z<cr> | find "TIME" and delete it; then insert "PLACE" | THIS IS NOW THE<cr><lf><br>PLACE cp FOR<cr><lf><br>ALL GOOD MEN<cr><lf> |
| 3FO↑z-3D5DICHANGES↑<cr> | find third occurrence of "O" (ie the second "O" in GOOD), delete previous 3 characters; then insert "CHANGES" | THIS IS NOW THE <cr><lf><br>PLACE FOR<cr><lf><br>ALL CHANGES cp<cr><lf> |
| -8CISOURCE<cr> | move back 8 characters and insert the line "SOURCE<cr><lf>" | THIS IS NOW THE<cr><lf><br>PLACE FOR<cr><lf><br>ALL SOURCE<cr><lf><br>cp CHANGES<cr><lf> |

ED also provides a single command which combines the F and I commands to perform simple string substitutions. The command takes the form

$$n\ S\ c_1c_2\ldots c_k{\uparrow}z\ d_1d_2\ldots d_m \left\{\begin{matrix}\langle cr\rangle \\ {\uparrow}z\end{matrix}\right\}$$

and has exactly the same effect as applying the command string

$$F\ c_1c_2\ldots c_k{\uparrow}z\text{-}kDId_1d_2\ldots d_m \left\{\begin{matrix}\langle cr\rangle \\ {\uparrow}z\end{matrix}\right\}$$

a total of n times. That is, ED searches the memory buffer starting at the current position of CP and successively substitutes the second string for the first string until the end of buffer, or until the substitution has been performed n times.

As a convenience, a command similar to F is provided by ED which automatically appends and writes lines as the search proceeds. The form is

$$n\ N\ c_1c_2\ldots c_k \left\{\begin{matrix}cr \\ {\uparrow}z\end{matrix}\right\}$$

which searches the entire source file for the nth occurrence of the string $c_1c_2\ldots c_k$ (recall that F fails if the string cannot be found in the current buffer). The operation of the

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

$$n\ J\ c_1c_2...c_k\uparrow z\ d_1d_2...d_m\uparrow z\ e_1e_2...e_q \begin{Bmatrix} \text{<cr>} \\ \uparrow z \end{Bmatrix}$$

with the following action applied n times to the memory buffer: search from the current CP for the next occurrence of the string $c_1c_2...c_k$. If found, insert the string $d_1,d_2...,d_m$, and move CP to follow $d_m$. Then delete all characters following CP up to (but not including) the string $e_1,e_2,...e_q$, leaving CP directly after $d_m$. If $e_1,e_2,...e_q$ cannot be found, then no deletion is made. If the current line is

```
cp NOW IS THE TIME<cr><lf>
```

Then the command

```
JW ↑zWHAT↑z↑l<cr>
```

Results in

```
NOW WHAT cp <cr><lf>
```

(Recall that ↑l represents the pair <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

$$R\ f_1f_2..f_n\uparrow z \qquad \text{or}$$

$$R\ f_1f_2..f_n\text{<cr>}$$

where $f_1f_2..f_n$ 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 characters into the memory buffer.

### 1.9. Repetitive Command Execution

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

$$n\ M\ c_1c_2...c_k \left\{\begin{matrix}\text{<cr>}\\ \uparrow z\end{matrix}\right\}$$

where $c_1c_2...c_k$ represent a string of ED commands, not including 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 occurrences 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) |
| O | 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.

## 3. CONTROL CHARACTERS AND COMMANDS

The following table summarizes the control characters and commands available in ED:

| Control Character | Function |
|---|---|
| ↑c | system reboot |
| ↑e | physical <cr><lf> (not actually entered in command) |
| ↑i | logical tab (cols 1,8, 15,...) |
| ↑l | logical <cr><lf> in search and substitute strings |
| ↑u | line delete |
| ↑z | string terminator |
| rubout | character delete |
| break | discontinue command (e.g., stop typing) |

| Command | Function |
|---|---|
| nA | append lines |
| ±B | begin bottom of buffer |
| ±nC | move character positions |
| ±nD | delete characters |
| E | end edit and close files (normal end) |
| nF | find string |
| H | end edit, close and reopen files |
| I | insert characters |
| nJ | place strings in juxtaposition |
| ±nK | kill lines |
| ±nL | move down/up lines |
| nM | macro definition |
| nN | find next occurrence with autoscan |
| O | return to original file |
| ±nP | move and print pages |
| Q | quit with no file changes |
| R | read library file |
| nS | substitute strings |
| ±nT | type lines |
| ± U | translate lower to upper case if U, no translation if -U |
| nW | write lines |
| nZ | sleep |
| ±n<cr> | move and type (±nLT) |

# 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

```
:4ØØT
```

is interpreted as "type from the current line number through the line whose absolute number is 4ØØ." Combining the two line reference forms, the command

```
345::4ØØT
```

for example, is interpreted as "move to absolute line 345, then type through absolute line 4ØØ." Note that absolute line references of this sort can precede any of the standard ED commands.

A special case of the V command, "ØV", 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

ØX

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 disasterous commands to be typed as single letters, rather than in composite commands. The commands

E (end), H (head), O (original), Q (quit)

must be typed as single letter commands.

ED 1.4 also prints error messages in the form

BREAK "x" AT c

where x is the error character, and c is the command where the error occurred.

# CP/M 2.0 USER'S GUIDE FOR CP/M 1.4 OWNERS

DIGITAL RESEARCH

Post Office Box 579, Pacific Grove, California 93950, (408) 649-3896

6

# CP/M 2.0 USER'S GUIDE
# FOR CP/M 1.4 OWNERS

## Copyright

## Disclaimer

Digital Research 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, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

## Trademarks

# CP/M 2.0 USER'S GUIDE FOR CP/M 1.4 OWNERS

Digital Research, Box 579
Pacific Grove, California

6

## 1. AN OVERVIEW OF CP/M 2.0 FACILITIES.

CP/M 2.0 is a high-performance single-console operating system which uses table driven techniques to allow field reconfiguration to match a wide variety of disk capacities. All of the fundamental file restrictions are removed, while maintaining upward compatibility from previous versions of release 1. Features of CP/M 2.0 include field specification of one to sixteen logical drives, each containing up to eight megabytes. Any particular file can reach the full drive size with the capability to expand to thirty-two megabytes in future releases. The directory size can be field configured to contain any reasonable number of entries, and each file is optionally tagged with read/only and system attributes. Users of CP/M 2.0 are physically separated by user numbers, with facilities for file copy operations from one user area to another. Powerful relative-record random access functions are present in CP/M 2.0 which provide direct access to any of the 65536 records of an eight megabyte file.

All disk-dependent portions of CP/M 2.0 are placed into a BIOS-resident "disk parameter block" which is either hand coded or produced automatically using the disk definition macro library provided with CP/M 2.0. The end user need only specify the maximum number of active disks, the starting and ending sector numbers, the data allocation size, the maximum extent of the logical disk, directory size information, and reserved track values. The macros use this information to generate the appropriate tables and table references for use during CP/M 2.0 operation. Deblocking information is also provided which aids in assembly or disassembly of sector sizes which are multiples of the fundamental 128 byte data unit, and the system alteration manual includes general-purpose subroutines which use the this deblocking information to take advantage of larger sector sizes. Use of these subroutines, together with the table driven data access algorithms, make CP/M 2.0 truly a universal data management system.

6

File expansion is achieved by providing up to 512 logical file extents, where each logical extent contains 16K bytes of data. CP/M 2.0 is structured, however, so that as much as 128K bytes of data is addressed by a single physical extent (corresponding to a single directory entry), thus maintaining compatibility with previous versions while taking full advantage of directory space.

Random access facilities are present in CP/M 2.0 which allow immediate reference to any record of an eight megabyte file. Using CP/M's unique data organization, data blocks are only allocated when actually required and movement to a record position requires little search time. Sequential file access is upward compatible from earlier versions to the full eight megabytes, while random access compatibility stops at 512K byte files. Due to CP/M 2.0's simpler and faster random access, application programmers are encouraged to alter their programs to take full advantage of the 2.0 facilities.

Several CP/M 2.0 modules and utilities have improvements which correspond to the enhanced file system. STAT and PIP both account for file attributes and user areas, while the CCP provides a "login" function to change from one user area to anotner. The CCP also formats directory displays in a more convenient manner and accounts for both CRT and hard-copy devices in its enhanced line editing functions.

The sections below point out the individual differences between CP/M 1.4 and CP/M 2.Ø, with the understanding that the reader is either familiar with CP/M 1.4, or has access to the 1.4 manuals. Additional information dealing with CP/M 2.Ø I/O system alteration is presented in the Digital Research manual "CP/M 2.Ø Alteration Guide."

## 2. USER INTERFACE.

Console line processing takes CRT-type devices into account with three new control characters, shown with an asterisk in the list below (the symbol "ctl" below indicates that the control key is simultaneously depressed):

| | |
|---|---|
| rub/del | removes and echoes last character |
| ctl-C | reboot when at beginning of line |
| ctl-E | physical end of line |
| ctl-H | backspace one character position* |
| ctl-J | (line feed) terminates current input* |
| ctl-M | (carriage return) terminates input |
| ctl-R | retype current line after new line |
| ctl-U | remove current line after new line |
| ctl-X | backspace to beginning of current line* |

In particular, note that ctl-H produces the proper backspace overwrite function (ctl-H can be changed internally to another character, such as delete, through a simple single byte change). Further, the line editor keeps track of the current prompt column position so that the operator can properly align data input following a ctl-U, ctl-R, or ctl-X command.

## 3. CONSOLE COMMAND PROCESSOR (CCP) INTERFACE.

There are four functional differences between CP/M 1.4 and CP/M 2.0 at the console command processor (CCP) level. The CCP now displays directory information across the screen (four elements per line), the USER command is present to allow maintenance of separate files in the same directory, and the actions of the "ERA *.*" and "SAVE" commands have changed. The altered DIR format is self-explanatory, while the USER command takes the form:

```
USER n
```

where n is an integer value in the range 0 to 15. Upon cold start, the operator is automatically "logged" into user area number 0, which is compatible with standard CP/M 1.4 directories. The operator may issue the USER command at any time to move to another logical area within the same directory. Drives which are logged-in while addressing one user number are automatically active when the operator moves to another user number since a user number is simply a prefix which accesses particular directory entries on the active disks.

The active user number is maintained until changed by a subsequent USER command, or until a cold start operation when user 0 is again assumed.

Due to the fact that user numbers now tag individual directory entries, the ERA *.* command has a different effect. In version 1.4, this command can be used to erase a directory which has "garbage" information, perhaps resulting from use of a diskette under another operating system (heaven forbid!). In 2.0, however, the ERA *.* command affects only the current user number. Thus, it is necessary to write a simple utility to erase a nonsense disk (the program simply writes the hexadecimal pattern E5 throughout the disk).

The SAVE command in version 1.4 allows only a single memory save operation, with the potential of destroying the memory image due to directory operations following extent boundary changes. Version 2.0, nowever, does not perform directory operations in user data areas after disk writes, and thus the SAVE operation can be used any number of times without altering the memory image.

## 4. STAT ENHANCEMENTS.

The STAT program has a number of additional functions which allow disk parameter display, user number display, and file indicator manipulation. The command:

```
STAT VAL:
```

produces a summary of the available status commands, resulting in the output:

```
Temp R/O Disk: d:=R/O
Set Indicator: d:filename.typ $R/O $R/W $SYS $DIR
Disk Status  : DSK: d:DSK:
User Status  : USR:
Iobyte Assign:
(list of possible assignments)
```

which gives an instant summary of the possible STAT commands. The command form:

```
STAT d:filename.typ $S
```

where "d:" is an optional drive name, and "filename.typ" is an unambiguous or ambiguous file name, produces the output display format:

```
 Size  Recs  Bytes  Ext Acc
   48    48     6k    1 R/O A:ED.COM
   55    55    12k    1 R/O (A:PIP.COM)
65536   128     2k    2 R/W A:X.DAT
```

where the $S parameter causes the "Size" field to be displayed (without the $S, the Size field is skipped, but the remaining fields are displayed). The Size field lists the virtual file size in records, while the "Recs" field sums the number of virtual records in each extent. For files constructed sequentially, the Size and Recs fields are identical. The "Bytes" field lists the actual number of bytes allocated to the corresponding file. The minimum allocation unit is determined at configuration time, and thus the number of bytes corresponds to the record count plus the remaining unused space in the last allocated block for sequential files. Random access files are given data areas only when written, so the Bytes field contains the only accurate allocation figure. In the case of random access, the Size field gives the logical end-of-file record position and the Recs field counts the logical records of each extent (each of these extents, however, may contain unallocated "holes" even though they are added into the record count). The "Ext" field counts the number of logical 16K extents allocated to the file. Unlike version 1.4, the Ext count does not necessarily correspond to the number of directory entries given to the file, since there can be up to 128K bytes (8 logical extents) directly addressed by a single directory entry, depending upon allocation size (in a special case, there are actually 256K bytes which can be directly addressed by a physical extent).

The "Acc" field gives the R/O or R/W access mode, which is changed using the commands shown below. Similarly, the parentheses

6

shown around the PIP.COM file name indicate that it has the "system" indicator set, so that it will not be listed in DIR commands. The four command forms

```
STAT d:filename.typ $R/O
STAT d:filename.typ $R/W
STAT d:filename.typ $SYS
STAT d:filename.typ $DIR
```

set or reset various permanent file indicators. The R/O indicator places the file (or set of files) in a read-only status until changed by a subsequent STAT command. The R/O status is recorded in the directory with the file so that it remains R/O through intervening cold start operations. The R/W indicator places the file in a permanent read/write status. The SYS indicator attaches the system indicator to the file, while the DIR command removes the system indicator. The "filename.typ" may be ambiguous or unambiguous, but in either case, the files whose attributes are changed are listed at the console when the change occurs. The drive name denoted by "d:" is optional.

When a file is marked R/O, subsequent attempts to erase or write into the file result in a terminal BDOS message

```
Bdos Err on d: File R/O
```

The BDOS then waits for a console input before performing a subsequent warm start (a "return" is sufficient to continue). The command form

```
STAT d:DSK:
```

lists the drive characteristics of the disk named by "d:" which is in the range A:, B:, ..., P:. The drive characteristics are listed in the format:

```
    d: Drive Characteristics
65536: 128 Byte record Capacity
 8192: Kilobyte Drive Capacity
  128: 32  Byte Directory Entries
    Ø: Checked  Directory Entries
 1Ø24: Records/ Extent
  128: Records/ Block
   58: Sectors/ Track
    2: Reserved Tracks
```

where "d:" is the selected drive, followed by the total record capacity (65536 is an 8 megabyte drive), followed by the total capacity listed in Kilobytes. The directory size is listed next, followed by the "checked" entries. The number of checked entries is usually identical to the directory size for removable media, since this mechanism is used to detect changed media during CP/M operation without an intervening warm start. For fixed media, the number is usually zero, since the media is not changed without at least a cold or warm start. The number of records per extent determines the addressing capacity of each directory entry (1Ø24 times 128 bytes, or

128K in the example above). The number of records per block shows the basic allocation size (in the example, 128 records/block times 128 bytes per record, or 16K bytes per block). The listing is then followed by the number of physical sectors per track and the number of reserved tracks. For logical drives which share the same physical disk, the number of reserved tracks may be quite large, since this mechanism is used to skip lower-numbered disk areas allocated to other logical disks. The command form

```
STAT DSK:
```

produces a drive characteristics table for all currently active drives. The final STAT command form is

```
STAT USR:
```

which produces a list of the user numbers which have files on the currently addressed disk. The display format is:

```
Active User : Ø
Active Files: Ø 1 3
```

where the first line lists the currently addressed user number, as set by the last CCP USER command, followed by a list of user numbers scanned from the current directory. In the above case, the active user number is Ø (default at cold start), with three user numbers which have active files on the current disk. The operator can subsequently examine the directories of the other user numbers by logging-in with USER 1, USER 2, or USER 3 commands, followed by a DIR command at the CCP level.

## 5. PIP ENHANCEMENTS.

PIP provides three new functions which account for the features of CP/M 2.Ø. All three functions take the form of file parameters which are enclosed in square brackets following the appropriate file names. The commands are:

| | |
|---|---|
| Gn | Get File from User number n<br>(n in the range Ø - 15) |
| W | Write over R/O files without<br>console interrogation |
| R | Read system files |

The G command allows one user area to receive data files from another. Assuming the operator has issued the USER 4 command at the CCP level, the PIP statement

```
PIP X.Y = X.Y[G2]
```

reads file X.Y from user number 2 into user area number 4. The command

```
PIP A:=A:*.*[G2]
```

copies all of the files from the A drive directory for user number 2 into the A drive directory of the currently logged user number. Note that to ensure file security, one cannot copy files into a different area than the one which is currently addressed by the USER command.

Note also that the PIP program itself is initially copied to a user area (so that subsequent files can be copied) using the SAVE command. The sequence of operations shown below effectively moves PIP from one user area to the next.

```
USER Ø              login user Ø
DDT PIP.COM         load PIP to memory
(note PIP size s)
GØ                  return to CCP
USER 3              login user 3
SAVE s PIP.COM
```

where s is the integral number of memory "pages" (256 byte segments) occupied by PIP. The number s can be determined when PIP.COM is loaded under DDT, by referring to the value under the "NEXT" display. If for example, the next available address is 1DØØ, then PIP.COM requires 1C hexadecimal pages (or 1 times 16 + 12 = 28 pages), and thus the value of s is 28 in the subsequent save. Once PIP is copied in this manner, it can then be copied to another disk belonging to the same user number through normal pip transfers.

Under normal operation, PIP will not overwrite a file which is set to a permanent R/O status. If attempt is made to overwrite a R/O file, the prompt

```
DESTINATION FILE IS R/O, DELETE (Y/N)?
```

is issued. If the operator responds with the character "y" then the file is overwritten. Otherwise, the response

```
** NOT DELETED **
```

is issued, the file transfer is skippped, and PIP continues with the next operation in sequence. In order to avoid the prompt and response in the case of R/O file overwrite, the command line can include the W parameter, as shown below

```
PIP A:=B:*.COM[W]
```

which copies all non-system files to the A drive from the B drive, and overwrites any R/O files in the process. If the operation involves several concatenated files, the W parameter need only be included with the last file in the list, as shown in the following example

```
PIP A.DAT = B.DAT,F:NEW.DAT,G:OLD.DAT[W]
```

Files with the system attribute can be included in PIP transfers if the R parameter is included, otherwise system files are not recognized. The command line

6

```
PIP ED.COM = B:ED.COM[R]
```

for example, reads the ED.COM file from the B drive, even if it has been marked as a R/O and system file. The system file attributes are copied, if present.

It should be noted that downward compatibility with previous versions of CP/M is only maintained if the file does not exceed one megabyte, no file attributes are set, and the file is created by user 0. If compatibility is required with non-standard (e.g., "double density") versions of 1.4, it may be necessary to select 1.4 compatibility mode when constructing the internal disk parameter block (see the "CP/M 2.0 Alteration Guide," and refer to Section 10 which describes BIOS differences).

## 6. ED ENHANCEMENTS.

The CP/M standard program editor provides several new facilities in the 2.Ø release. Experience has shown that most operators use the relative line numbering feature of ED, and thus the editor has the "v" (Verify Line) option set as an initial value. The operator can, of course, disable line numbering by typing the "-v" command. If you are not familiar with the ED line number mode, you may wish to refer to the Appendix in the ED user's guide, where the "v" command is described.

ED also takes file attributes into account. If the operator attempts to edit a read/only file, the message

```
** FILE IS READ/ONLY **
```

appears at the console. The file can be loaded and examined, but cannot be altered in any way. Normally, the operator simply ends the edit session, and uses STAT to change the file attribute to R/W. If the edited file has the "system" attribute set, the message

```
"SYSTEM" FILE NOT ACCESSIBLE
```

is displayed at the console, and the edit session is aborted. Again, the STAT program can be used to change the system attribute, if desired.

Finally, the insert mode ("i") command allows CRT line editing functions, as described in Section 2, above.

## 7. THE XSUB FUNCTION.

An additional utility program is supplied with version 2.Ø of CP/M, called XSUB, which extends the power of the SUBMIT facility to include line input to programs as well as the console command processor. The XSUB command is included as the first line of your submit file and, when executed, self-relocates directly below the CCP. All subsequent submit command lines are processed by XSUB, so that programs which read buffered console input (BDOS function 1Ø) receive their input directly from the submit file. For example, the file SAVER.SUB could contain the submit lines:

```
XSUB
DDT
I$1.HEX
R
GØ
SAVE 1 $2.COM
```

with a subsequent SUBMIT command:

```
SUBMIT SAVER X Y
```

6

which substitutes X for $1 and Y for $2 in the command stream. The XSUB program loads, followed by DDT which is sent the command lines "IX.HEX" "R" and "GØ" thus returning to the CCP. The final command "SAVE 1 Y.COM" is processed by the CCP.

The XSUB program remains in memory, and prints the message

```
(xsub active)
```

on each warm start operation to indicate its presence. Subsequent submit command streams do not require the XSUB, unless an intervening cold start has occurred. Note that XSUB must be loaded after DESPOOL, if both are to run simultaneously.

## 8. BDOS INTERFACE CONVENTIONS.

CP/M 2.0 system calls take place in exactly the same manner as earlier versions, with a call to location 0005H, function number in register C, and information address in register pair DE. Single byte values are returned in register A, with double byte values returned in HL (for reasons of compatibility, register A = L and register B = H upon return in all cases). A list of CP/M 2.0 calls is given below, with an asterisk following functions which are either new or revised from version 1.4 to 2.0. Note that a zero value is returned for out-of range function numbers.

```
 0  System Reset
 1  Console Input
 2  Console Output
 3  Reader Input
 4  Punch Output
 5  List Output
 6* Direct Console I/O
 7  Get I/O Byte
 8  Set I/O Byte
 9  Print String
10* Read Console Buffer
11  Get Console Status
12* Return Version Number
13  Reset Disk System
14  Select Disk
15* Open File
16  Close File
17* Search for First
18* Search for Next
19* Delete File
20  Read Sequential
21  Write Sequential
22* Make File
23* Rename File
24* Return Login Vector
25  Return Current Disk
26  Set DMA Address
27  Get Addr(Alloc)
28* Write Protect Disk
29* Get Addr(R/O Vector)
30* Set File Attributes
31* Get Addr(Disk Parms)
32* Set/Get User Code
33* Read Random
34* Write Random
35* Compute File Size
36* Set Random Record
```

(Functions 28, 29, and 32 should be avoided in application programs to maintain upward compatibility with MP/M.) The new or revised functions are described below.

### Function 6: Direct Console I/O.

Direct Console I/O is supported under CP/M 2.0 for those applications where it is necessary to avoid the BDOS console I/O operations. Programs which currently perform direct I/O through the BIOS should be changed to use direct I/O under BDOS so that they can be fully supported under future releases of MP/M and CP/M.

Upon entry to function 6, register E either contains hexadecimal FF, denoting a console input request, or register E contains an ASCII character. If the input value is FF, then function 6 returns A = 00 if no character is ready, otherwise A contains the next console input character.

If the input value in E is not FF, then function 6 assumes that E contains a valid ASCII character which is sent to the console.

Function 10: Read Console Buffer.

The console buffer read operation remains unchanged except that console line editing is supported, as described in Section 2. Note also that certain functions which return the carriage to the leftmost position (e.g., ctl-X) do so only to the column position where the prompt ended (previously, the carriage returned to the extreme left margin). This new convention makes operator data input and line correction more legible.

Function 12: Return Version Number.

Function 12 has been redefined to provide information which allows version-independent programming (this was previously the "lift head" function which returned HL=0000 in version 1.4, but performed no operation). The value returned by function 12 is a two-byte value, with H = 00 for the CP/M release (H = 01 for MP/M), and L = 00 for all releases previous to 2.0. CP/M 2.0 returns a hexadecimal 20 in register L, with subsequent version 2 releases in the hexadecimal range 21, 22, through 2F. Using function 12, for example, you can write application programs which provide both sequential and random access functions, with random access disabled when operating under early releases of CP/M.

In the file operations described below, DE addresses a file control block (FCB). Further, all directory operations take place in a reserved area which does not affect write buffers as was the case in version 1.4, with the exception of Search First and Search Next, where compatibility is required.

The File Control Block (FCB) data area consists of a sequence of 33 bytes for sequential access, and a series of 36 bytes in the case that the file is accessed randomly. The default file control block normally located at 005CH can be used for random access files, since bytes 007DH, 007EH, and 007FH are available for this purpose. For notational purposes, the FCB format is shown with the following fields:

```
 -----------------------------------------------------------
 |dr|f1|f2|/ /|f8|t1|t2|t3|ex|s1|s2|rc|dØ|/ /|dn|cr|rØ|r1|r2|
 -----------------------------------------------------------
  ØØ Ø1 Ø2 ... Ø8 Ø9 1Ø 11 12 13 14 15 16 ... 31 32 33 34 35
```

where

dr — drive code (Ø - 16)
Ø => use default drive for file
1 => auto disk select drive A,
2 => auto disk select drive B,
...
16=> auto disk select drive P.

f1...f8 — contain the file name in ASCII
upper case, with high bit = Ø

t1,t2,t3 — contain the file type in ASCII
upper case, with high bit = Ø
t1', t2', and t3' denote the
bit of these positions,
t1' = 1 => Read/Only file,
t2' = 1 => SYS file, no DIR list

ex — contains the current extent number,
normally set to ØØ by the user, but
in range Ø - 31 during file I/O

s1 — reserved for internal system use

s2 — reserved for internal system use, set
to zero on call to OPEN, MAKE, SEARCH

rc — record count for extent "ex,"
takes on values from Ø - 128

dØ...dn — filled-in by CP/M, reserved for
system use

cr — current record to read or write in
a sequential file operation, normally
set to zero by user

rØ,r1,r2 — optional random record number in the
range Ø-65535, with overflow to r2,
rØ,r1 constitute a 16-bit value with
low byte rØ, and high byte r1

Function 15: Open File.

The Open File operation is identical to previous definitions, with the exception that byte s2 is automatically zeroed. Note that previous versions of CP/M defined this byte as zero, but made no checks to assure compliance. Thus, the byte is cleared to ensure upward compatibility with the latest version, where it is required.

Function 17: Search for First.

Search First scans the directory for a match with the file given by the FCB addressed by DE. The value 255 (hexadecimal FF) is returned if the file is not found, otherwise a value of A equal to 0, 1, 2, or 3 is returned indicating the file is present. In the case that the file is found, the current DMA address is filled with the record containing the directory entry, and the relative starting position is A * 32 (i.e., rotate the A register left 5 bits, or ADD A five times). Although not normally required for application programs, the directory information can be extracted from the buffer at this position.

An ASCII question mark (63 decimal, 3F hexadecimal) in any position from f1 through ex matches the corresponding field of any directory entry on the default or auto-selected disk drive. If the dr field contains an ASCII question mark, then the auto disk select function is disabled, the default disk is searched, with the search function returning any matched entry, allocated or free, belonging to any user number. This latter function is not normally used by application programs, but does allow complete flexibility to scan all current directory values. If the dr field is not a question mark, the s2 byte is automatically zeroed.

6

Function 18: Search for Next.

The Search Next function is similar to the Search First function, except that the directory scan continues from the last matched entry. Similar to function 17, function 18 returns the decimal value 255 in A when no more directory items match.

Function 19: Delete File.

The Delete File function removes files which match the FCB addressed by DE. The filename and type may contain ambiguous references (i.e., question marks in various positions), but the drive select code cannot be ambiguous, as in the Search and Search Next functions.

Function 19 returns a decimal 255 if the reference file or files could not be found, otherwise a value in the range 0 to 3 is returned.

Function 22: Make File.

The Make File operation is identical to previous versions of CP/M, except that byte s2 is zeroed upon entry to the BDOS.

Function 23: Rename File.

The Actions of the file rename functions are the same as previous releases except that the value 255 is returned if the rename function is unsuccessful (the file to rename could not be found), otherwise a value in the range Ø to 3 is returned.

Function 24: Return Login Vector.

The login vector value returned by CP/M 2.Ø is a 16-bit value in HL, where the least significant bit of L corresponds to the first drive A, and the high order bit of H corresponds to the sixteenth drive, labelled P. Note that compatibility is maintained with earlier releases, since registers A and L contain the same values upon return.

Function 28: Write Protect Current Disk.

The disk write protect function provides temporary write protection for the currently selected disk. Any attempt to write to the disk, before the next cold or warm start operation produces the message

```
Bdos Err on d: R/O
```

Function 29: Get R/O Vector.

Function 29 returns a bit vector in register pair HL which indicates drives which have the temporary read/only bit set. Similar to function 24, the least significant bit corresponds to drive A, while the most significant bit corresponds to drive P. The R/O bit is set either by an explicit call to function 28, or by the automatic software mechanisms within CP/M which detect changed disks.

Function 3Ø: Set File Attributes.

The Set File Attributes function allows programmatic manipulation of permanent indicators attached to files. In particular, the R/O and System attributes (tl' and t2' above) can be set or reset. The DE pair addresses an unambiguous file name with the appropriate attributes set or reset. Function 3Ø searches for a matcn, and changes the matched directory entry to contain the selected inaicators. Indicators f1' through f4' are not presently used, but may be useful for applications programs, since they are not involved in the matching process during file open and close operations. Indicators f5' tnrough f8' and t3' are reserved for future system expansion.

Function 31: Get Disk Parameter Block Address.

The address of the BIOS resident disk parameter block is returned in HL as a result of this function call. This address can be used for either of two purposes. First, the disk parameter values can be extracted for display and space computation purposes, or transient programs can dynamically change the values of current disk parameters when the disk environment changes, if required. Normally, application programs will not require this facility.

Function 32: Set or Get User Code.

An application program can change or interrogate the currently active user number by calling function 32. If register E = FF nexadecimal, then tne value of the current user number is returned in register A, where the value is in the range Ø to 31. If register E is not FF, then the current user number is changed to the value of E (modulo 32).

Function 33: Read Random.

The Read Random function is similar to the sequential file read operation of previous releases, except that the read operation takes place at a particular record number, selected by the 24-bit value constructed from the three byte field following the FCB (byte positions rØ at 33, r1 at 34, and r2 at 35). Note that the sequence of 24 bits is stored with least significant byte first (rØ), middle byte next (r1), and high byte last (r2). CP/M release 2.Ø does not reference byte r2, except in computing the size of a file (function 35). Byte r2 must be zero, however, since a non-zero value indicates overflow past the end of file.

Thus, in version 2.Ø, the rØ,r1 byte pair is treated as a double-byte, or "word" value, which contains the record to read. This value ranges from Ø to 65535, providing access to any particular record of the 8 megabyte file. In order to process a file using random access, the base extent (extent Ø) must first be opened. Altnough the base extent may or may not contain any allocated data, this ensures that the file is properly recorded in the directory, and is visible in DIR requests. The selected record number is then stored into the random record field (rØ,r1), and the BDOS is called to read the record. Upon return from the call, register A either contains an error code, as listed below, or the value ØØ indicating the operation was successful. In the latter case, the current DMA address contains the randomly accessed record. Note that contrary to the sequential read operation, the record number is not advanced. Thus, subsequent random read operations continue to read the same record.

Upon each random read operation, the logical extent and current record values are automatically set. Thus, the file can be sequentially read or written, starting from the current randomly accessed position. Note, however, that in this case, the last randomly read record will be re-read as you switch from random mode to sequential read, and the last record will be re-written as you switch to a sequential write operation. You can, of course, simply advance the random record position following each random read or write to obtain the effect of a sequential I/O operation.

Error codes returned in register A following a random read are listed below.

| | |
|---|---|
| Ø1 | reading unwritten data |
| Ø2 | (not returned in random mode) |
| Ø3 | cannot close current extent |
| Ø4 | seek to unwritten extent |
| Ø5 | (not returned in read mode) |
| Ø6 | seek past physical end of disk |

Error code Ø1 and Ø4 occur when a random read operation accesses a data block which has not been previously written, or an extent which has not been created, which are equivalent conditions. Error 3 does not normally occur under proper system operation, but can be cleared by simply re-reading, or re-opening extent zero as long as the disk is not physically write protected. Error code Ø6 occurs whenever byte r2 is non-zero under the current 2.Ø release. Normally, non-zero return codes can be treated as missing data, with zero return codes indicating operation complete.

Function 34: Write Random.

The Write Random operation is initiated similar to the Read Random call, except that data is written to the disk from the current DMA address. Further, if the disk extent or data block which is the target of the write has not yet been allocated, the allocation is performed before the write operation continues. As in the Read Random operation, the random record number is not changed as a result of the write. The logical extent number and current record positions of the file control block are set to correspond to the random record which is being written. Again, sequential read or write operations can commence following a random write, with the notation that the currently addressed record is either read or rewritten again as the sequential operation begins. You can also simply advance the random record position following each write to get the effect of a sequential write operation. Note that in particular, reading or writing the last record of an extent in random mode does not cause an automatic extent switch as it does in sequential mode under either CP/M 1.4 or CP/M 2.Ø.

The error codes returned by a random write are identical to the random read operation with the addition of error code Ø5, which indicates that a new extent cannot be created due to directory overflow.

Function 35: Compute File Size.

When computing the size of a file, the DE register pair addresses an FCB in random mode format (bytes rØ, r1, and r2 are present). The FCB contains an unambiguous file name which is used in the directory scan. Upon return, the random record bytes contain the "virtual" file size which is, in effect, the record address of the record following the end of the file. if, following a call to function 35, the high record byte r2 is Ø1, then the file contains the maximum record count 65536 in version 2.Ø. Otherwise, bytes rØ and r1 constitute a 16-bit value (rØ is the least significant byte, as before) which is the file size.

Data can be appended to the end of an existing file by simply calling function 35 to set the random record position to the end of file, tnen performing a sequence of random writes starting at the preset record address.

The virtual size of a file corresponds to the physical size when the file is written sequentially. If, instead, the file was created in random mode and "holes" exist in the allocation, then the file may in fact contain fewer records than the size indicates. If, for example, only the last record of an eight megabyte file is written in random mode (i.e., record number 65535), then the virtual size is 65536 records, although only one block of data is actually allocated.

Function 36: Set Random Record.

The Set Random Record function causes the BDOS to automatically produce the random record position from a file which has been read or written sequentially to a particular point. The function can be useful in two ways.

First, it is often necessary to initially read and scan a sequential file to extract the positions of various "key" fields. As each key is encountered, function 36 is called to compute the random record position for the data corresponding to this key. If the data unit size is 128 bytes, the resulting record position is placed into a table with the key for later retrieval. After scanning the entire file and tabularizing the keys and their record numbers, you can move instantly to a particular keyed record by performing a random read using the corresponding random record number which was saved earlier. The scheme is easily generalized when variable record lengths are involved since the program need only store the buffer-relative byte position along with the key and record number in order to find the exact starting position of the keyed data at a later time.

A second use of function 36 occurs when switching from a sequential read or write over to random read or write. A file is sequentially accessed to a particular point in the file, function 36 is called which sets the record number, and subsequent random read and write operations continue from the selected point in the file.

This section is concluded with a rather extensive, but complete example of random access operation. The program listed below performs the simple function of reading or writing random records upon command from the terminal. Given that the program has been created, assembled, and placed into a file labelled RANDOM.COM, the CCP level command:

```
RANDOM X.DAT
```

starts the test program. The program looks for a file by the name X.DAT (in this particular case) and, if found, proceeds to prompt the console for input. If not found, the file is created before the prompt is given. Each prompt takes the form

```
next command?
```

and is followed by operator input, terminated by a carriage return. The input commands take the form

```
nW    nR    Q
```

where n is an integer value in the range 0 to 65535, and W, R, and Q are simple command characters corresponding to random write, random read, and quit processing, respectively. If the W command is issued, the RANDOM program issues the prompt

```
type data:
```

The operator then responds by typing up to 127 characters, followed by a carriage return. RANDOM then writes the character string into the X.DAT file at record n. If the R command is issued, RANDOM reads record number n and displays the string value at the console. If the Q command is issued, the X.DAT file is closed, and the program returns to the console command processor. In the interest of brevity (ok, so the program's not so brief), the only error message is

```
error, try again
```

The program begins with an initialization section where the input file is opened or created, followed by a continuous loop at the label "ready" where the individual commands are interpreted. The default file control block at 005CH and the default buffer at 0080H are used in all disk operations. The utility subroutines then follow, which contain the principal input line processor, called "readc." This particular program shows the elements of random access processing, and can be used as the basis for further program development.

```
                ;*************************************************
                ;*                                               *
                ;* sample random access program for cp/m 2.0     *
                ;*                                               *
                ;*************************************************
0100                    org     100h    ;base of tpa
                ;
0000 =          reboot  equ     0000h   ;system reboot
0005 =          bdos    equ     0005h   ;bdos entry point
                ;
0001 =          coninp  equ     1       ;console input function
0002 =          conout  equ     2       ;console output function
0009 =          pstring equ     9       ;print string until '$'
000a =          rstring equ     10      ;read console buffer
000c =          version equ     12      ;return version number
000f =          openf   equ     15      ;file open function
0010 =          closef  equ     16      ;close function
0016 =          makef   equ     22      ;make file function
0021 =          readr   equ     33      ;read random
0022 =          writer  equ     34      ;write random
                ;
005c =          fcb     equ     005ch   ;default file control block
007d =          ranrec  equ     fcb+33  ;random record position
007f =          ranovf  equ     fcb+35  ;high order (overflow) byte
0080 =          buff    equ     0080h   ;buffer address
                ;
000d =          cr      equ     0dh     ;carriage return
000a =          lf      equ     0ah     ;line feed
                ;
                ;*************************************************
                ;*                                               *
                ;* load SP, set-up file for random access        *
                ;*                                               *
                ;*************************************************
0100 31bc0              lxi     sp,stack
                ;
                ;       version 2.0?
0103 0e0c               mvi     c,version
0105 cd050              call    bdos
0108 fe20               cpi     20h     ;version 2.0 or better?
010a d2160              jnc     versok
                ;       bad version, message and go back
010d 111b0              lxi     d,badver
0110 cdda0              call    print
0113 c3000              jmp     reboot
                ;
                versok:
                ;       correct version for random access
```

```
0116 0e0f          mvi     c,openf ;open default fcb
0118 115c0         lxi     d,fcb
011b cd050         call    bdos
011e 3c            inr     a       ;err 255 becomes zero
011f c2370         jnz     ready
           ;
           ;       cannot open file, so create it
0122 0e16          mvi     c,makef
0124 115c0         lxi     d,fcb
0127 cd050         call    bdos
012a 3c            inr     a       ;err 255 becomes zero
012b c2370         jnz     ready
           ;
           ;       cannot create file, directory full
012e 113a0         lxi     d,nospace
0131 cdda0         call    print
0134 c3000         jmp     reboot  ;back to ccp
           ;
           ;***************************************************
           ;*                                                 *
           ;*  loop back to "ready" after each command        *
           ;*                                                 *
           ;***************************************************
           ;
           ready:
           ;       file is ready for processing
           ;
0137 cde50         call    readcom ;read next command
013a 227d0         shld    ranrec  ;store input record#
013d 217f0         lxi     h,ranovf
0140 3600          mvi     m,0     ;clear high byte if set
0142 fe51          cpi     'Q'     ;quit?
0144 c2560         jnz     notq
           ;
           ;       quit processing, close file
0147 0e10          mvi     c,closef
0149 115c0         lxi     d,fcb
014c cd050         call    bdos
014f 3c            inr     a       ;err 255 becomes 0
0150 cab90         jz      error   ;error message, retry
0153 c3000         jmp     reboot  ;back to ccp
           ;
           ;***************************************************
           ;*                                                 *
           ;* end of quit command, process write              *
           ;*                                                 *
           ;***************************************************
           notq:
           ;       not the quit command, random write?
0156 fe57          cpi     'W'
0158 c2890         jnz     notw
           ;
           ;       this is a random write, fill buffer until cr
015b 114d0         lxi     d,datmsg
015e cdda0         call    print   ;data prompt
```

```
0161 0e7f           mvi     c,127   ;up to 127 characters
0163 21800          lxi     h,buff  ;destination
            rloop:  ;read next character to buff
0166 c5             push    b       ;save counter
0167 e5             push    h       ;next destination
0168 cdc20          call    getchr  ;character to a
016b e1             pop     h       ;restore counter
016c c1             pop     b       ;restore next to fill
016d fe0d           cpi     cr      ;end of line?
016f ca780          jz      erloop
            ;       not end, store character
0172 77             mov     m,a
0173 23             inx     h       ;next to fill
0174 0d             dcr     c       ;counter goes down
0175 c2660          jnz     rloop   ;end of buffer?
            erloop:
            ;       end of read loop, store 00
0178 3600           mvi     m,0
            ;
            ;       write the record to selected record number
017a 0e22           mvi     c,writer
017c 115c0          lxi     d,fcb
017f cd050          call    bdos
0182 b7             ora     a       ;error code zero?
0183 c2b90          jnz     error   ;message if not
0186 c3370          jmp     ready   ;for another record
            ;
            ;*****************************************************
            ;*                                                   *
            ;* end of write command, process read                *
            ;*                                                   *
            ;*****************************************************
            notw:
            ;       not a write command, read record?
0189 fe52           cpi     'R'
018b c2b90          jnz     error   ;skip if not
            ;
            ;       read random record
018e 0e21           mvi     c,readr
0190 115c0          lxi     d,fcb
0193 cd050          call    bdos
0196 b7             ora     a       ;return code 00?
0197 c2b90          jnz     error
            ;
            ;       read was successful, write to console
019a cdcf0          call    crlf    ;new line
019d 0e80           mvi     c,128   ;max 128 characters
019f 21800          lxi     h,buff  ;next to get
            wloop:
01a2 7e             mov     a,m     ;next character
01a3 23             inx     h       ;next to get
01a4 e67f           ani     7fh     ;mask parity
01a6 ca370          jz      ready   ;for another command if 00
01a9 c5             push    b       ;save counter
01aa e5             push    h       ;save next to get
```

```
01ab fe20          cpi                 ;graphic?
01ad d4c80         cnc     putchr      ;skip output if not
01b0 e1            pop     h
01b1 c1            pop     b
01b2 0d            dcr     c           ;count=count-1
01b3 c2a20         jnz     wloop
01b6 c3370         jmp     ready
           ;
           ;***************************************************
           ;*                                                 *
           ;* end of read command, all errors end-up here     *
           ;*                                                 *
           ;***************************************************
           ;
           error:
01b9 11590         lxi     d,errmsg
01bc cdda0         call    print
01bf c3370         jmp     ready
           ;
           ;***************************************************
           ;*                                                 *
           ;* utility subroutines for console i/o             *
           ;*                                                 *
           ;***************************************************
           getchr:
                   ;read next console character to a
01c2 0e01          mvi     c,coninp
01c4 cd050         call    bdos
01c7 c9            ret
           ;
           putchr:
                   ;write character from a to console
01c8 0e02          mvi     c,conout
01ca 5f            mov     e,a         ;character to send
01cb cd050         call    bdos        ;send character
01ce c9            ret
           ;
           crlf:
                   ;send carriage return line feed
01cf 3e0d          mvi     a,cr        ;carriage return
01d1 cdc80         call    putchr
01d4 3e0a          mvi     a,lf        ;line feed
01d6 cdc80         call    putchr
01d9 c9            ret
           ;
           print:
                   ;print the buffer addressed by de until $
01da d5            push    d
01db cdcf0         call    crlf
01de d1            pop     d           ;new line
01df 0e09          mvi     c,pstring
01e1 cd050         call    bdos        ;print the string
01e4 c9            ret
           ;
           readcom:
                   ;read the next command line to the conbuf
01e5 116b0         lxi     d,prompt
01e8 cdda0         call    print   ;command?
01eb 0e0a          mvi     c,rstring
01ed 117a0         lxi     d,conbuf
01f0 cd050         call    bdos    ;read command line
           ;       command line is present, scan it
01f3 21000         lxi     h,0     ;start with 0000
01f6 117c0         lxi     d,conlin;command line
01f9 1a    readc:  ldax    d       ;next command character
01fa 13            inx     d       ;to next command position
01fb b7            ora     a       ;cannot be end of command
01fc c8            rz
           ;       not zero, numeric?
01fd d630          sui     '0'
01ff fe0a          cpi     10      ;carry if numeric
0201 d2130         jnc     endrd
           ;       add-in next digit
0204 29            dad     h       ;*2
0205 4d            mov     c,l
0206 44            mov     b,h     ;bc = value * 2
0207 29            dad     h       ;*4
0208 29            dad     h       ;*8
0209 09            dad     b       ;*2 + *8 = *10
020a 85            add     l       ;+digit
020b 6f            mov     l,a
020c d2f90         jnc     readc   ;for another char
020f 24            inr     h       ;overflow
0210 c3f90         jmp     readc   ;for another char
           endrd:
           ;       end of read, restore value in a
0213 c630          adi     '0'     ;command
0215 fe61          cpi     'a'     ;translate case?
0217 d8            rc
           ;       lower case, mask lower case bits
0218 e65f          ani     101$1111b
021a c9            ret
           ;
           ;*****************************************************
           ;*                                                   *
           ;* string data area for console messages             *
           ;*                                                   *
           ;*****************************************************
           badver:
021b 536f79        db      'sorry, you need cp/m version 2$'
           nospace:
023a 4e6f29        db      'no directory space$'
           datmsg:
024d 547970        db      'type data: $'
           errmsg:
0259 457272        db      'error, try again.$'
           prompt:
026b 4e6570        db      'next command? $'
           ;
           ;****************************************************
           ;*                                                  *
           ;* fixed and variable data area                     *
           ;*                                                  *
           ;****************************************************
Ø27a 21    conbuf: db      conlen  ;length of console buffer
Ø27b       consiz: ds      1       ;resulting size after read
Ø27c       conlin: ds      32      ;length 32 buffer
ØØ21 =     conlen  equ     $-consiz
           ;
Ø29c               ds      32      ;16 level stack
           stack:
Ø2bc               end
```

## 9. CP/M 2.0 MEMORY ORGANIZATION.

Similar to earlier versions, CP/M 2.0 is field-altered to fit various memory sizes, depending upon the host computer memory configuration. Typical base addresses for popular memory sizes are shown in the table below.

| Module | 20k | 24k | 32k | 48k | 64k |
|---|---|---|---|---|---|
| CCP | 3400H | 4400H | 6400H | A400H | E400H |
| BDOS | 3C00H | 4C00H | 6C00H | AC00H | EC00H |
| BIOS | 4A00H | 5A00H | 7A00H | BA00H | FA00H |
| Top of Ram | 4FFFH | 5FFFH | 7FFFH | BFFFH | FFFFH |

The distribution disk contains a CP/M 2.0 system configured for a 20k Intel MDS-800 with standard IBM 8" floppy disk drives. The disk layout is shown below:

| Sector | Track 00 | Module | Track 01 | Module |
|---|---|---|---|---|
| 1 | (Bootstrap Loader) | | 4080H | BDOS + 480H |
| 2 | 3400H | CCP + 000H | 4100H | BDOS + 500H |
| 3 | 3480H | CCP + 080H | 4180H | BDOS + 580H |
| 4 | 3500H | CCP + 100H | 4200H | BDOS + 600H |
| 5 | 3580H | CCP + 180H | 4280H | BDOS + 680H |
| 6 | 3600H | CCP + 200H | 4300H | BDOS + 700H |
| 7 | 3680H | CCP + 280H | 4380H | BDOS + 780H |
| 8 | 3700H | CCP + 300H | 4400H | BDOS + 800H |
| 9 | 3780H | CCP + 380H | 4480H | BDOS + 880H |
| 10 | 3800H | CCP + 400H | 4500H | BDOS + 900H |
| 11 | 3880H | CCP + 480H | 4580H | BDOS + 980H |
| 12 | 3900H | CCP + 500H | 4600H | BDOS + A00H |
| 13 | 3980H | CCP + 580H | 4680H | BDOS + A80H |
| 14 | 3A00H | CCP + 600H | 4700H | BDOS + B00H |
| 15 | 3A80H | CCP + 680H | 4780H | BDOS + B80H |
| 16 | 3B00H | CCP + 700H | 4800H | BDOS + C00H |
| 17 | 3B80H | CCP + 780H | 4880H | BDOS + C80H |
| 18 | 3C00H | BDOS + 000H | 4900H | BDOS + D00H |
| 19 | 3C80H | BDOS + 080H | 4980H | BDOS + D80H |
| 20 | 3D00H | BDOS + 100H | 4A00H | BIOS + 000H |
| 21 | 3D80H | BDOS + 180H | 4A80H | BIOS + 080H |
| 22 | 3E00H | BDOS + 200H | 4B00H | BIOS + 100H |
| 23 | 3E80H | BDOS + 280H | 4B80H | BIOS + 180H |
| 24 | 3F00H | BDOS + 300H | 4C00H | BIOS + 200H |
| 25 | 3F80H | BDOS + 380H | 4C80H | BIOS + 280H |
| 26 | 4000H | BDOS + 400H | 4D00H | BIOS + 300H |

In particular, note that the CCP is at the same position on the disk, and occupies the same space as version 1.4. The BDOS portion, however, occupies one more 256-byte page and the BIOS portion extends through the remainder of track 01. Thus, the CCP is 800H (2048 decimal) bytes in length, the BDOS is E00H (3584 decimal) bytes in length, and the BIOS is up to 380H (898 decimal) bytes in length. In version 2.0, the BIOS portion contains the standard subroutines of 1.4, along with some initialized table space, as described in the following section.

## 10. BIOS DIFFERENCES.

The CP/M 2.0 Basic I/O System differs only slightly in concept from its predecesssors. Two new jump vector entry points are defined, a new sector translation subroutine is included, and a disk characteristics table must be defined. The skeletal form of these changes are found in the program shown below.

```
        org     4000h
        maclib  diskdef
        jmp     boot
;       ...
        jmp     listst  ;list status
        jmp     sectran ;sector translate
        disks   4
;       large capacity drive
bpb     equ     16*1024 ;bytes per block
rpb     equ     bpb/128 ;records per block
maxb    equ     65535/rpb ;max block number
        diskdef 0,1,58,3,bpb,maxb+1,128,0,2
        diskdef 1,1,58,,bpb,maxb+1,128,0,2
        diskdef 2,0
        diskdef 3,1
;
boot:   ret     ;nop
;
listst: xra     a       ;nop
        ret
;
seldsk:
        ;drive number in c
        lxi     h,0     ;0000 in hl produces select error
        mov     a,c     ;a is disk number 0 ... ndisks-1
        cpi     ndisks  ;less than ndisks?
        rnc             ;return with HL = 0000 if not
;       proper disk number, return dpb element address
        mov     l,c
        dad     h       ;*2
        dad     h       ;*4
        dad     h       ;*8
        dad     h       ;*16
        lxi     d,dpbase
        dad     d       ;HL=.dpb
        ret
;
selsec:
        ;sector number in c
        lxi     h,sector
        mov     m,c
        ret
;
sectran:
        ;translate sector BC using table at DE
        xchg            ;HL = .tran
        dad     b       ;single precision tran
```

```
48: ;         dad b again if double precision tran
49:           mov     l,m      ;only low byte necessary here
5Ø: ;         fill both H and L if double precision tran
51:           ret              ;HL = ??ss
52: ;
53: sector:   ds      1
54:           endef
55:           end
```

Referring to the program shown above, lines 3-6 represent the BIOS entry vector of 17 elements (version 1.4 defines only 15 jump vector elements). The last two elements provide access to the "LISTST" (List Status) entry point for DESPOOL. The use of this particular entry point is defined in the DESPOOL documentation, and is no different than the previous 1.4 release. It should be noted that the 1.4 DESPOOL program will not operate under version 2.Ø, but an update version will be available from Digital Research in the near future.

The "SECTRAN" (Sector Number Translate) entry shown in the jump vector at line 6 provides access to a BIOS-resident sector translation subroutine. This mechanism allows the user to specify the sector skew factor and translation for a particular disk system, and is described below.

A macro library is shown in the listing,. called DISKDEF, included on line 2, and referenced in 12-15. Although it is not necessary to use the macro library, it greatly simplifies the disk definition process. You must have access to the MAC macro assembler, of course, to use the DISKDEF facility, while the macro library is included with all CP/M 2.Ø distribution disks. (See the CP/M 2.Ø Alteration Guide for formulas which you can use to hand-code the tables produced by the DISKDEF library).

A BIOS disk definition consists of the following sequence of macro statements:

```
MACLIB   DISKDEF
......
DISKS    n
DISKDEF  Ø,...
DISKDEF  1,...
......
DISKDEF  n-1
......
ENDEF
```

where the MACLIB statement loads the DISKDEF.LIB file (on the same disk as your BIOS) into MAC's internal tables. The DISKS macro call follows, which specifies the number of drives to be configured with your system, where n is an integer in the range 1 to 16. A series of DISKDEF macro calls then follow which define the characteristics of each logical disk, Ø through n-1 (corresponding to logical drives A through P). Note that the DISKS and DISKDEF macros generate in-line fixed data tables, and thus must be placed in a non-executable portion of your BIOS, typically directly following the BIOS jump vector.

The remaining portion of your BIOS is defined following the DISKDEF macros, with the ENDEF macro call immediately preceding the END statement. The ENDEF (End of Diskdef) macro generates the necessary uninitialized RAM areas which are located above your BIOS.

The form of the DISKDEF macro call is

```
        DISKDEF  dn,fsc,lsc,[skf],bls,dks,dir,cks,ofs,[0]
```

where

dn      is the logical disk number, 0 to n-1
fsc     is the first physical sector number (0 or 1)
lsc     is the last sector number
skf     is the optional sector skew factor
bls     is the data allocation block size
dir     is the number of directory entries
cks     is the number of "checked" directory entries
ofs     is the track offset to logical track 00
[0]     is an optional 1.4 compatibility flag

The value "dn" is the drive number being defined with this DISKDEF macro invocation. The "fsc" parameter accounts for differing sector numbering systems, and is usually 0 or 1. The "lsc" is the last numbered sector on a track. When present, the "skf" parameter defines the sector skew factor which is used to create a sector translation table according to the skew. If the number of sectors is less than 256, a single-byte table is created, otherwise each translation table element occupies two bytes. No translation table is created if the skf parameter is omitted (or equal to 0). The "bls" parameter specifies the number of bytes allocated to each data block, and takes on the values 1024, 2048, 4096, 8192, or 16384. Generally, performance increases with larger data block sizes since there are fewer directory references and logically connected data records are physically close on the disk. Further, each directory entry addresses more data and the BIOS-resident ram space is reduced. The "dks" specifies the total disk size in "bls" units. That is, if the bls = 2048 and dks = 1000, then the total disk capacity is 2,048,000 bytes. If dks is greater than 255, then the block size parameter bls must be greater than 1024. The value of "dir" is the total number of directory entries which may exceed 255, if desired. The "cks" parameter determines the number of directory items to check on each directory scan, and is used internally to detect changed disks during system operation, where an intervening cold or warm start has not occurred (when this situation is detected, CP/M automatically marks the disk read/only so that data is not subsequently destroyed). Normally the value of cks = dir when the media is easily changed, as is the case with a floppy disk subsystem. If the disk is permanently mounted, then the value of cks is typically 0, since the probability of changing disks without a restart is quite low. The "ofs" value determines the number of tracks to skip when this particular drive is addressed, which can be used to reserve additional operating system space or to simulate several logical drives on a single large capacity physical drive. Finally, the [0] parameter is included when file compatibility is required with versions of 1.4 which have been modified for higher density disks. This parameter ensures that only 16K is allocated for each directory record, as was the case for previous versions. Normally, this parameter is not included.

For convenience and economy of table space, the special form

```
                    DISKDEF    i,j
```

gives disk i the same characteristics as a previously defined drive j. A standard four-drive single density system, which is compatible with version 1.4, is defined using the following macro invocations:

```
DISKS      4
 DISKDEF   0,1,26,6,1024,243,64,64,2
 DISKDEF   1,0
 DISKDEF   2,0
 DISKDEF   3,0

 ....
 ENDEF
```

with all disks having the same parameter values of 26 sectors per track (numbered 1 through 26), with 6 sectors skipped between each access, 1024 bytes per data block, 243 data blocks for a total of 243k byte disk capacity, 64 checked directory entries, and two operating system tracks.

The definitions given in the program shown above (lines 12 through 15) provide access to the largest disks addressable by CP/M 2.0. All disks have identical parameters, except that drives 0 and 2 skip three sectors on every data access, while disks 1 and 3 access each sector in sequence as the disk revolves (there may, however, be a transparent hardware skew factor on these drives).

The DISKS macro generates n "disk header blocks," starting at address DPBASE which is a label generated by the macro. Each disk header block contains sixteen bytes, and correspond, in sequence, to each of the defined drives. In the four drive standard system, for example, the DISKS macro generates a table of the form:

```
DPBASE  EQU  $
DPE0:   DW   XLT0,0000H,0000H,0000H,DIRBUF,DPB0,CSV0,ALV0
DPE1:   DW   XLT0,0000H,0000H,0000H,DIRBUF,DPB0,CSV1,ALV1
DPE2:   DW   XLT0,0000H,0000H,0000H,DIRBUF,DPB0,CSV2,ALV2
DPE3:   DW   XLT0,0000H,0000H,0000H,DIRBUF,DPB0,CSV3,ALV3
```

where the DPE (disk parameter entry) labels are included for reference purposes to show the beginning table addresses for each drive 0 through 3. The values contained within the disk parameter header are described in detail in the CP/M 2.0 Alteration Guide, but basically address the translation vector for the drive (all reference XLT0, which is the translation vector for drive 0 in the above example), followed by three 16-bit "scratch" addresses, followed by the directory buffer address, disk parameter block address, check vector address, and allocation vector address. The check and allocation vector addresses are generated by the ENDEF macro in the ram area following the BIOS code and tables.

The SELDSK function is extended somewhat in version 2.Ø. In particular, the selected disk number is passed to the BIOS in register C, as before, and the SELDSK subroutine performs the appropriate software or hardware actions to select the disk. Version 2.Ø, however, also requires the SELDSK subroutine to return the address of the selected disk parameter header (DPEØ, DPE1, DPE2, or DPE3, in the above example) in register HL. If SELDSK returns the value HL = ØØØØH, then the BDOS assumes the disk does not exist, and prints a select error mesage at the terminal. Program lines 22 through 36 give a sample CP/M 2.Ø SELDSK subroutine, showing only the disk parameter header address calculation.

The subroutine SECTRAN is also included in version 2.Ø which performs the actual logical to physical sector translation. In earlier versions of CP/M, the sector translation process was a part of the BDOS, and set to skip six sectors between each read. Due differing rotational speeds of various disks, the translation function has become a part of the BIOS in version 2.Ø. Thus, the BDOS sends sequential sector numbers to SECTRAN, starting at sector number Ø. The SECTRAN subroutine uses the sequential sector number to produce a translated sector number which is returned to the BDOS. The BDOS subsequently sends the translated sector number to SELSEC before the actual read or write is performed. Note that many controllers have the capability to record the sector skew on the disk itself, and thus there is no translation necessary. In this case, the "skf" parameter is omitted in the macro call, and SECTRAN simply returns the same value which it receives. The table shown below, for example, is constructed when the standard skew factor skf = 6 is specified in the DISKDEF macro call:

```
XLTØ:  DB    1,7,13,19,25,5,11,17,23,3,9,15,21
       DB    2,8,14,2Ø,26,6,12,18,24,4,1Ø,16,22
```

If SECTRAN is required to translate a sector, then the following process takes place. The sector to translate is received in register pair BC. Only the C register is significant if the sector value does not exceed 255 (B = ØØ in this case). Register pair DE addresses the sector translate table for this drive, determined by a previous call on SELDSK, corresponding to the first element of a disk parameter header (XLTØ in the case shown above). The SECTRAN subroutine then fetches the translated sector number by adding the input sector number to the base of the translate table, to get the indexed translate table address (see lines 46, 47, and 48 in the above program). The value at this location is then returned in register L. Note that if the number of sectors exceeds 255, the translate table contains 16-bit elements whose value must be returned in HL.

Following the ENDEF macro call, a number of uninitialized data areas are defined. These data areas need not be a part of the BIOS which is loaded upon cold start, but must be available between the BIOS and the end of memory. The size of the uninitialized RAM area is determined by EQU statements generated by the ENDEF macro. For a standard four-drive system, the ENDEF macro might produce

```
        4C72 =        BEGDAT EQU $
                      (data areas)
        4DB0 =        ENDDAT EQU $
        013C =        DATSIZ EQU $-BEGDAT
```

which indicates that uninitialized RAM begins at location 4C72H, ends at 4DB0H-1, and occupies 013CH bytes. You must ensure that these addresses are free for use after the system is loaded.

CP/M 2.0 is also easily adapated to disk subsystems whose sector size is a multiple of 128 bytes. Information is provided by the BDOS on sector write operations which eliminates the need for pre-read operations, thus allowing blocking and deblocking to take place at the BIOS level.

See the "CP/M 2.0 Alteration Guide" for additional details concerning tailoring your CP/M system to your particular hardware.

# OPERATION OF THE CP/M DEBUGGER

7

DIGITAL RESEARCH

Post Office Box 579, Pacific Grove, California 93950, (408) 649-3896

# CP/M DYNAMIC DEBUGGING TOOL (DDT)

# USER'S GUIDE

## Disclaimer

Digital Research 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, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

## Table of Contents

Section Page

# 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 a sign-on message in the format

```
nnK DDT-s VER m.m
```

where nn is the memory size (which must match the CP/M system being used), s is the hardware system which is assumed, corresponding to the codes

| | | |
|---|---|---|
| D | - | Digital Research standard version |
| M | - | MDS version |
| I | - | IMSAI standard version |
| O | - | Omron systems |
| S | - | Digital Systems standard version |

and m.m is the revision number.

Following the sign on message, DDT prompts the operator with the character "-" and waits for input commands from the console. The operator can type any of several single character commands, terminated by a carriage return to execute the command. Each line of input can be line-edited using the standard CP/M controls

| | |
|---|---|
| rubout | remove the last character typed |
| ctl-U | remove the entire line, ready for re-typing |
| ctl-C | system reboot |

Any command can be up to 32 characters in length (an automatic carriage return is inserted as the 33rd character), where the first character determines the command type

| | |
|---|---|
| A | enter assembly language mnemonics with operands |
| D | display memory in hexadecimal and ASCII |
| F | fill memory with constant data |
| G | begin execution with optional breakpoints |
| I | set up a standard input file control block |
| L | list memory using assembler mnemonics |
| M | move a memory segment from source to destination |
| R | read program for subsequent testing |
| S | substitute memory values |
| T | trace program execution |
| U | untraced program monitoring |
| X | examine and optionally alter the CPU state |

The command character, in some cases, is followed by zero, one, two, or three hexadecimal values which are separated by commas or single blank characters. All DDT numeric output is in hexadecimal form. In all cases, the commands are not executed until the carriage return is typed at the end of the command.

At any point in the debug run, the operator can stop execution of DDT using either a ctl-C or GØ (jmp to location ØØØØH), and save the current memory image using a SAVE command of the form

```
SAVE n filename.COM
```

where n is the number of pages (256 byte blocks) to be saved on disk. The number of blocks can be determined by taking the high order byte of the top load address and converting this number to decimal. For example, if the highest address in the Transient Program Area is 1234H then the number of pages is 12H, or 18 in decimal. Thus the operator could type a ctl-C during the debug run, returning to the Console Processor level, followed by

```
SAVE 18 X.COM
```

The memory image is saved as X.COM on the diskette, and can be directly executed by simply typing the name X. If further testing is required, the memory image can be recalled by typing

```
          DDT X.COM
```

which reloads previously saved program from loaction 1ØØH 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 rubout 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).

7

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 1ØØH. 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 8Ø8Ø Assembly Language Reference Card for a list of mnemonics), followed by register references and operands in absolute hexadecimal form. Each sucessive 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).

2. The D (Display) Command. The D command allows the operator to view the contents of memory in hexadecimal and ASCII formats. The forms are

```
D
Ds
Ds,f
```

In the first case, memory is displayed from the current display address (initially 1ØØH), and continues for 16 display lines. Each display line takes the form shown below

```
aaaa bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb cccccccccccccccc
```

where aaaa is the display address in hexadecimal, and bb represents data present in memory starting at aaaa. The ASCII characters starting at aaaa are given to the right (represented by the sequence of c´s), where non-graphic characters are printed as a period (.) symbol. Note that both upper and lower case alphabetics are displayed, and thus will appear as upper case symbols on a console device that supports only upper case. Each display line gives the values of 16 bytes of data, except that the first line displayed is truncated so that the next line begins at an address which is a multiple of 16.

The second form of the D command shown above is similar to the first, except that the display address is first set to address s. The third form causes the display to continue from address s through address f. In all cases, the display address is set to the first address not displayed in this command, so that a continuing display can be accomplished by issuing successive D commands with no explicit addresses.

Excessively long displays can be aborted by pushing the rubout key.

3. The F (Fill) Command. The F command takes the form

```
Fs,f,c
```

where s is the starting address, f is the final address, and c is a hexadecimal byte constant. The effect is as follows: DDT stores the constant c at address s, increments the value of s and tests against f. If s exceeds f then the operation terminates, otherwise the operation is repeated. Thus, the fill command can be used to set a memory block to a specific constant value.

4. The G (Go) Command. Program execution is started using the G comand, with up to two optional breakpoint addresses. The G command takes one ot the forms

```
G
Gs
Gs,b
```

```
                    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

```
                    G4ØØ,4ØØ
```

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 FCB 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

```
Ifilename.filetype
```

If the second form is used, and the filetype is either HEX or COM, then subsequent R commands can be used to read the pure binary or hex format machine code (see the R command for further details).

6. The L (List) Command. The L command is used to list assembly language mnemonics in a particular program region. The forms are

```
L
Ls
Ls,f
```

The first command lists twelve lines of disassembled machine code from the current list address. The second form sets the list address to s, and then lists twelve lines of code. The last form lists disassembled code from s through address f. In all three cases, the list address is set to the next unlisted location in preparation for a subsequent L command. Upon encountering an execution breakpoint, the list address is set to the current value of the program counter (see the G and T commands). Again, long typeouts can be aborted using the rubout key during the list process.

7. The M (Move) Command. The M command allows block movement of program or data areas from one location to another in memory. The form is

```
Ms,f,d
```

where s is the start address of the move, f is the final address of the move, and d is the destination address. Data is first moved from s to d, and both addresses are incremented. If s exceeds f then the move operation stops, otherwise the move operation is repeated.

8. The R (Read) Command. The R command is used in conjunction with the I command to read COM and HEX files from the diskette into the transient program area in preparation for the debug run. The forms are

```
R
Rb
```

where b is an optional bias address which is added to each program or data address as it is loaded. The load operation must not overwrite any of the system parameters from ØØØH through ØFFH (i.e., the first page of memory). If b is omitted, then b=ØØØØ is assumed. The R command requires a previous I command, specifying the name of a HEX or COM file. The load address for each record is obtained from each individual HEX record, while an assumed load address of 1ØØH is taken for COM files. Note that any number of R commands can be issued following the I command to re-read the program under test, 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 (1ØØH 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.

1Ø. 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 (0FFFFH) 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

| | | |
|---|---|---|
| C | Carry Flag | (0/1) |
| Z | Zero Flag | (0/1) |

| | | |
|---|---|---|
| M | Minus Flag | (Ø/1) |
| E | Even Parity Flag | (Ø/1) |
| I | Interdigit Carry | (Ø/1) |
| A | Accumulator | (Ø-FF) |
| B | BC register pair | (Ø-FFFF) |
| D | DE register pair | (Ø-FFFF) |
| H | HL register pair | (Ø-FFFF) |
| S | Stack Pointer | (Ø-FFFF) |
| P | Program Counter | (Ø-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 Ø 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 overlayed 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↵
                      tab character          rubout  rubout echo
*I↵  ↑-I  ORG  ↑-I    100H       L_L;START OF TRANSIENT AREA↵
          MVI         B,LEN      ;LENGTH OF VECTOR TO SCAN↵
          MVI         C,0        ;LARGER_RST VALUE SO FAR↵
LOOP__P_O_O_L         LXI        H,VECT   ;BASE OF VECTOR↵
LOOP:     MOV         A,M        ;GET VALUE↵
          SUB         C          ;LARGER VALUE IN C?↵
Rubout deletes character
          JNC         NFOUND     ;JUMP IF LARGER VALUE NOT FOUND↵
;         NEW LARGEST VALUE, STORE IT TO C↵
          MOV         C,A↵
NFOUND:   INX         H          ;TO NEXT ELEMENT↵
          DCR         B          ;MORE TO SCAN?↵
          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↵
↑z *B0P↵
          ORG         100H       ;START OF TRANSIENT AREA
          MVI         B,LEN      ;LENGTH OF VECTOR TO SCAN
          MVI         C,0        ;LARGEST VALUE SO FAR
          LXI         H,VECT     ;BASE OF VECTOR
LOOP:     MOV         A,M        ;GET VALUE
          SUB         C          ;LARGER VALUE IN C?
          JNC         NFOUND     ;JUMP IF LARGER VALUE NOT FOUND
;         NEW LARGEST VALUE, STORE IT TO C
          MOV         C,A
NFOUND:   INX         H          ;TO NEXT ELEMENT
          DCR         B          ;MORE TO SCAN?
          JNZ         LOOP       ;FOR ANOTHER
```

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
*E        ← End of Edit


ASM SCAN   Start Assembler
CP/M ASSEMBLER - VER 1.0

0122
002H USE FACTOR
END OF ASSEMBLY   Assembly Complete - Lock at Program Listing

TYPE SCAN.PRN

Code Address               Source Program
 0100    Machine Code         ORG     100H     ;START OF TRANSIENT AREA
 0100 0608                    MVI     B,LEN    ;LENGTH OF VECTOR TO SCAN
 0102 0E00                    MVI     C,0      ;LARGEST VALUE SO FAR
 0104 211901                  LXI     H,VECT   ;BASE OF VECTOR
 0107 7E            LOOP:     MOV     A,M      ;GET VALUE
 0108 91                      SUB     C        ;LARGER VALUE IN C?
 0109 D20D01                  JNC     NFOUND   ;JUMP IF LARGER VALUE NOT FOUND
                    ;         NEW LARGEST VALUE, STORE IT TO C
 010C 4F                      MOV     C,A
 010D 23            NFOUND:   INX     H        ;TO NEXT ELEMENT
 010E 05                      DCR     B        ;MORE TO SCAN?
 010F C20701                  JNZ     LOOP     ;FOR ANOTHER
                    ;
                    ;         END OF SCAN, STORE C
 0112 79                      MOV     A,C      ;GET LARGEST VALUE
 0113 322101                  STA     LARGE
 0116 C30000                  JMP     0        ;REBOOT
      Code/data listing     ;
      truncated             ;         TEST DATA
 0119 0200040305VECT:         DB      2,0,4,3,5,6,1,5
 0008 =             LEN       EQU     $-VECT   ;LENGTH
 0121 Value of      LARGE:    DS      1        ;LARGEST VALUE ON EXIT
 0122   Equate                END
A>
```

7

```
DDT SCAN.HEX          Start Debugger using hex format machine code
16K DDT VER 1.0
NEXT  PC
0121  0000
-X                last load address +1
                                                                next instruction
                                                                to execute at
C0Z0M0E0I0 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                           PC changed.
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI  B,08
-L100                                                Next instruction
                                                     to execute at Pc=100
0100  MVI  B,08
0102  MVI  C,00
0104  LXI  H,0119
0107  MOV  A,M
0108  SUB  C
0109  JNC  010D       Disassembled Machine
010C  MOV  C,A        Code at 100H.
010D  INX  H          (See Source Listing
010E  DCR  B          for comparison)
010F  JNZ  0107
0112  MOV  A,C
-L
0113  STA  0121
0116  JMP  0000
0119  STAX B
011A  NOP
011B  INR  B          A litte more
011C  INX  B          machine code
011D  DCR  B          (note that Program
011E  MVI  B,01       ends at location 116
0120  DCR  B          with a JMP to 0000)
0121  LXI  D,2200
0124  LXI  H,0200
-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
0116  RST 7      is ever executed.
0117  (single carriage return stops assemble mode)
-L113  List code at 113H to check that RST 7 was properly inserted
0113  STA  0121   in Place of JMP
0116  RST  07
```

```
0117   NOP
0118   NOP
0119   STAX B
011A   NOP
011B   INR  B
011C   INX  B
-
```

-X ↲ Look at registers

```
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI  B,08
```

-I ↲ Execute Program for one step. initial CPU state, before ↲ is executed

```
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI  B,08*0102
```

-I ↲ Trace one step again (note 08H in B) automatic breakpoint

```
C0Z0M0E0I0 A=00 B=0800 D=0000 H=0000 S=0100 P=0102 MVI  C,00*0104
```

-I ↲ Trace again (Register C is cleared)

```
C0Z0M0E0I0 A=00 B=0800 D=0000 H=0000 S=0100 P=0104 LXI  H,0119*0107
```

-T3 ↲ Trace three steps

```
C0Z0M0E0I0 A=00 B=0800 D=0000 H=0119 S=0100 P=0107 MOV  A,M
C0Z0M0E0I0 A=02 B=0800 D=0000 H=0119 S=0100 P=0108 SUB  C
C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0109 JNC  010D*010D
```

-D119 ↲ Display memory starting at 119H. automatic breakpoint at 10DH

Program data

Lowercase x

```
0119 02 00 04 03 05 06 01                                  .......
0120 05 11 00 22 21 00 02 7E EB 77 13 23 EB 0B 78 B1 ..."!..^.W.#..x.
0130 C2 27 01 C3 03 29 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 ................
0150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0160 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 ................
0180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
01A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
01B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
01C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Data is displayed in ASCII with a "." in the position of non-graphic characters

-X ↲ Current CPU state

```
C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=010D INX  H
```

-T5 ↲ Trace 5 steps from current CPU state

```
C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=010D INX  H
C0Z0M0E0I1 A=02 B=0800 D=0000 H=011A S=0100 P=010E DCR  B
C0Z0M0E0I1 A=02 B=0700 D=0000 H=011A S=0100 P=010F JNZ  0107
C0Z0M0E0I1 A=02 B=0700 D=0000 H=011A S=0100 P=0107 MOV  A,M
C0Z0M0E0I1 A=00 B=0700 D=0000 H=011A S=0100 P=0108 SUB  C*0109
```

Automatic Breakpoint

-U5 ↲ Trace without listing intermediate states

```
C0Z1M0E1I1 A=00 B=0700 D=0000 H=011A S=0100 P=0109 JNC  010D*0108
```

-X ↲ CPU State at end of U5 ↲

```
C0Z0M0E1I1 A=04 B=0600 D=0000 H=011B S=0100 P=0108 SUB  C
```

```
-G↲
```
Run Program from current PC until completion (in real-time)

```
*0116
```
breakpoint at 116H, caused by executing RST 7 in machine code

```
-X↲
```
CPU state at end of Program

```
C0Z1M0E1I1 A=00 B=0000 D=0000 H=0121 S=0100 P=0116 RST   07
-XP↲
```
examine and change Program counter

```
P=0116 100↲

-X↲

C0Z1M0E1I1 A=00 B=0000 D=0000 H=0121 S=0100 P=0100 MVI   B,08
-T10↲
```
Trace 10 (hexadecimal) steps

first data element

current largest value

subtract for comparison A<C

```
C0Z1M0E1I1 A=00 B=0000 D=0000 H=0121 S=0100 P=0100 MVI   B,08
C0Z1M0E1I1 A=00 B=0800 D=0000 H=0121 S=0100 P=0102 MVI   C,00
C0Z1M0E1I1 A=00 B=0800 D=0000 H=0121 S=0100 P=0104 LXI   H,0119
C0Z1M0E1I1 A=00 B=0800 D=0000 H=0119 S=0100 P=0107 MOV   A,M
C0Z1M0E1I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0108 SUB   C
C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0109 JNC   010D
C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=010D INX   H
C0Z0M0E0I1 A=02 B=0800 D=0000 H=011A S=0100 P=010E DCR   B
C0Z0M0E0I1 A=02 B=0700 D=0000 H=011A S=0100 P=010F JNZ   0107
C0Z0M0E0I1 A=02 B=0700 D=0000 H=011A S=0100 P=0107 MOV   A,M
C0Z0M0E0I1 A=00 B=0700 D=0000 H=011A S=0100 P=0108 SUB   C
C0Z1M0E1I1 A=00 B=0700 D=0000 H=011A S=0100 P=0109 JNC   010D
C0Z1M0E1I1 A=00 B=0700 D=0000 H=011A S=0100 P=010D INX   H
C0Z1M0E1I1 A=00 B=0700 D=0000 H=011B S=0100 P=010E DCR   B
C0Z0M0E1I1 A=00 B=0600 D=0000 H=011B S=0100 P=010F JNZ   0107
C0Z0M0E1I1 A=00 B=0600 D=0000 H=011B S=0100 P=0107 MOV   A,M*0108
-A109↲
0109 JC 10D↲
010C↲
```
Insert a "hot patch" into the machine code to change the JNC to JC

Program should have moved the value from A into C since A>C. Since this code was not executed, it appears that the JNC should have been a JC instruction

```
-G0↲
```
Stop DDT so that a version of the patched Program can be saved

```
SAVE 1 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 1.0
NEXT  PC
0200 0100
-L100↲
```
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
-XP

P=0100

-T10   Trace to see how patched version operates
                                                   Data is moved from A to C
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI  B,08
C0Z0M0E0I0 A=00 B=0800 D=0000 H=0000 S=0100 P=0102 MVI  C,00
C0Z0M0E0I0 A=00 B=0800 D=0000 H=0000 S=0100 P=0104 LXI  H,0119
C0Z0M0E0I0 A=00 B=0800 D=0000 H=0119 S=0100 P=0107 MOV  A,M
C0Z0M0E0I0 A=02 B=0800 D=0000 H=0119 S=0100 P=0108 SUB  C
C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0109 JC   010D
C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=010C MOV  C,A
C0Z0M0E0I1 A=02 B=0802 D=0000 H=0119 S=0100 P=010D INX  H
C0Z0M0E0I1 A=02 B=0802 D=0000 H=011A S=0100 P=010E DCR  B
C0Z0M0E0I1 A=02 B=0702 D=0000 H=011A S=0100 P=010F JNZ  0107
C0Z0M0E0I1 A=02 B=0702 D=0000 H=011A S=0100 P=0107 MOV  A,M
C0Z0M0E0I1 A=00 B=0702 D=0000 H=011A S=0100 P=0108 SUB  C
C1Z0M1E0I0 A=FE B=0702 D=0000 H=011A S=0100 P=0109 JC   010D
C1Z0M1E0I0 A=FE B=0702 D=0000 H=011A S=0100 P=010D INX  H
C1Z0M1E0I0 A=FE B=0702 D=0000 H=011B S=0100 P=010E DCR  B
C1Z0M0E1I1 A=FE B=0602 D=0000 H=011B S=0100 P=010F JNZ  0107*0107
-X                                                 breakpoint after 16 steps

C1Z0M0E1I1 A=FE B=0602 D=0000 H=011B S=0100 P=0107 MOV  A,M
-G,108   Run from current PC and breakpoint at 108H

*0108
-X                next data item

C1Z0M0E1I1 A=04 B=0602 D=0000 H=011B S=0100 P=0108 SUB  C
-T                 Single Step for a few cycles

C1Z0M0E1I1 A=04 B=0602 D=0000 H=011B S=0100 P=0108 SUB  C*0109
-T

C0Z0M0E0I1 A=02 B=0602 D=0000 H=011B S=0100 P=0109 JC   010D*010C
-X

C0Z0M0E0I1 A=02 B=0602 D=0000 H=011B S=0100 P=010C MOV  C,A
-G   Run to completion

*0116
-X

C0Z1M0E1I1 A=03 B=0003 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↲       End of the S command

0127 7E .↲

-L100↲

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        Review the Code
-L↲

0113  STA   0121
0116  RST   07
0117  NOP
0118  NOP
0119  STAX  B
011A  NOP
011B  INR   B
011C  INX   B
011D  DCR   B
011E  MVI   B,01
0120  DCR   B
-XP↲

P=0116 100↲    Reset the PC

-I↲  Single step, and watch data values

C0Z1M0E1I1 A=03 B=0003 D=0000 H=0121 S=0100 P=0100 MVI   B,08*0102
-T↲

C0Z1M0E1I1 A=03 B=0803 D=0000 H=0121 S=0100 P=0102 MVI   C,00*0104
-T↲
                    count set
                        "largest" set
C0Z1M0E1I1 A=03 B=0800 D=0000 H=0121 S=0100 P=0104 LXI   H,0119*0107
-T↲
                                   base address of data set
C0Z1M0E1I1 A=03 B=0800 D=0000 H=0119 S=0100 P=0107 MOV   A,M*0108
```

```
-T
                 ┌first data item brought to A
C0Z1M0E1I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0108 SUB  C*0109
-T

C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0109 JC   010D*010C
-T

C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=010C MOV  C,A*010D
-T                          ┌first data item moved to C correctly

C0Z0M0E0I1 A=02 B=0802 D=0000 H=0119 S=0100 P=010D INX  H*010E
-T

C0Z0M0E0I1 A=02 B=0802 D=0000 H=011A S=0100 P=010E DCR  B*010F
-T

C0Z0M0E0I1 A=02 B=0702 D=0000 H=011A S=0100 P=010F JNZ  0107*0107
-T

C0Z0M0E0I1 A=02 B=0702 D=0000 H=011A S=0100 P=0107 MOV  A,M*0108
-T               ┌second data item brought to A

C0Z0M0E0I1 A=00 B=0702 D=0000 H=011A S=0100 P=0108 SUB  C*0109
-T               ┌subtract destroys data value which was loaded!!!

C1Z0M1E0I0 A=FE B=0702 D=0000 H=011A S=0100 P=0109 JC   010D*010D
-T

C1Z0M1E0I0 A=FE B=0702 D=0000 H=011A S=0100 P=010D INX  H*010E
-L100

0100  MVI   B,08
0102  MVI   C,00
0104  LXI   H,0119
0107  MOV   A,M
0108  SUB   C    ←———This should have been a CMP so that register A
0109  JC    010D     would not be destroyed.
010C  MOV   C,A
010D  INX   H
010E  DCR   B
010F  JNZ   0107
0112  MOV   A,C
-A108

0108  CMP C      hot patch at 108H changes SUB to CMP

0109

-G0    stop DDT for SAVE
```

7

```
SAVE 1 SCAN.COM↲          Save memory image
A>DDT SCAN.COM↲           Restart DDT
16K DDT VER 1.0
NEXT  PC
0200 0100
-XP↲
P=0100↲
-L116↲
0116  RST  07  ⎫
0117  NOP      ⎪
0118  NOP      ⎬ Look at code to see if it was properly Loaded
0119  STAX B   ⎪ (long typeout aborted with rubout)
011A  NOP      ⎪
- (rubout)     ⎭
-G,116↲   Run from 100H to completion
*0116
-XC↲   Look at Carry (accidental typo)
C1↲
-X↲   Look at CPU state
C1Z1M0E1I1 A=06 B=0006 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.ASM↲   Re-edit the source program, and make both changes
*NSUB↲
*0LT↲              ctl-z
         SUB       C       ;LARGER VALUE IN C?
*SSUB↑ZCMP↑Z0LT↲
         CMP       C       ;LARGER VALUE IN C?
*↲
         JNC       NFOUND  ;JUMP IF LARGER VALUE NOT FOUND
*SNC↑ZC↑Z0LT↲
         JC        NFOUND  ;JUMP IF LARGER VALUE NOT FOUND
*E↲
```

```
ASM SCAN.AAZ  Re-assemble, selecting source from disk A
                                             hex to disk A
CP/M ASSEMBLER - VER 1.0                     Print to Z (selects no Print file)

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  STAX B
011A  NOP
011B  INR  B
- (rubout)

-G100,116     Go from beginning with breakpoint at end

*0116   breakpoint reached
-D121   Look at "LARGE" ——— correct value computed

0121 06 00 22 21 00 02 7E EB 77 13 23 EB 0B 78 B1 .."!..^.W.#..X.
0130 C2 27 01 C3 03 29 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 ................

- (rubout)   aborts long typeout

-G0    stop DDT, debug session complete
```

# OPERATION OF THE CP/M ASSEMBLER

DIGITAL RESEARCH

Post Office Box 579, Pacific Grove, California 93950, (408) 649-3896

# CP/M ASSEMBLER (ASM)
# USER'S GUIDE

## Disclaimer

Digital Research 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, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

# Table of Contents

Section Page

8

CP/M Assembler User´s Guide

## 1. INTRODUCTION.

The CP/M assembler reads assembly language source files from the diskette, and produces 8Ø8Ø 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 8Ø8Ø 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.p1p2p3
```

where p1, p2, and p3 are single letters

p1: A,B, ..., Y   designates the disk name which contains

| | | |
|---|---|---|
| | | the source file |
| 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 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:
X1Y2       X1x2       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 compatability 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. (addresses)

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 1Ø) |
| H | hexadecimal constant (base 16) |

Q is an alternate radix indicator for octal numbers since the letter O is easily confused with the digit Ø. 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 Ø and 1 digits, octal constants can contain digits in the range Ø - 7, while decimal constants contain decimal digits. Hexadecimal constants contain decimal digits as well as hexadecimal digits A (1ØD), 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 Ø 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    11ØØB    1111$ØØØØ$1111$ØØØØB
1234H    ØFFEH    3377O    33$77$22Q
3377o    Øfe3h    1234d    Øffffh
```

### 3.3. Reserved Words.

There are several reserved character sequences which have predefined meanings in the operand field of a statement. The names of 8Ø8Ø registers are given below, which, when encountered, produce the value shown to the right

| | |
|---|---|
| A | 7 |
| B | Ø |
| 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 4ØH).

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 withing 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 Ø - 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 Ø´s become 1´s, 1´s become Ø´s), where b is considered a 16-bit value |

| | |
|---|---|
| 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 logicl 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

```
1Ø+2Ø        1Øh+37Q      L1 /3     (L2+4) SHR 3
('a' and 5fh) + 'Ø'      ('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 Ø-1 which results in the value Øffffh (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 Øffffh which cannot be represented as an 8 bit value), while "ADI (-1) AND ØFFH" is accepted by the assembler since the "AND" operation zeroes the high order bits of the expression.

ie., must fit in one-byte

8

3.6. Precedence of Operators.

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

```
* / MOD SHL SHR          ← AOS Operating system
    - +
    NOT
    AND
   OR XOR
```

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 SHL d      ((a MOD b) * c) SHL 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 assmbly, 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  ØFFFFH      ;DEFINE VALUE OF TRUE
FALSE    EQU  NOT TRUE    ;DEFINE VALUE OF FALSE
;
TTY      EQU  TRUE        ;TRUE IF TTY, FALSE IF CRT
;
TTYBASE EQU   1ØH         ;BASE OF TTY I/O PORTS
CRTBASE EQU   2ØH         ;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 1ØH. 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 2ØH.

### 4.6. The DB Directive.

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 (Ø). 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   Ø,1,2,3,4,5
                  DB   data and Øffh,5,377Q,1+2+3+4
          signon: DB   'please type your name',cr,lf,Ø
                  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 8Ø8Ø processor: the least significant byte of the expression is stored forst in memory, followed by the most significant byte. Examples are

```
          doub:   DW   Øffefh,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 8Ø8Ø microcomputer, which are given in detail in the Intel manual "8Ø8Ø 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 Ø-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 Ø-255 |
| e16 | represents a 16-bit value in the range Ø-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 8Ø8Ø microcomputer CPU. The forms are

| | | |
|---|---|---|
| JMP e16 | JMP L1 | Jump unconditionally to label |
| JNZ e16 | JMP L2 | Jump on non zero condition to label |
| JZ e16 | JMP 1ØØH | Jump on zero condition to label |
| JNC e16 | JNC L1+4 | Jump no carry to label |
| JC e16 | JC L3 | Jump on carry to label |
| JPO e16 | JPO $+8 | Jump on parity odd to label |
| JPE e16 | JPE L4 | Jump on even parity to label |
| JP e16 | JP GAMMA | Jump on positive result to label |
| JM e16 | JM al | Jump on minus to label |
| CALL e16 | CALL S1 | Call subroutine unconditionally |
| CNZ e16 | CNZ S2 | Call subroutine if non zero flag |
| CZ e16 | CZ 1ØØH | Call subroutine on zero flag |
| CNC e16 | CNC S1+4 | Call subroutine if no carry set |
| CC e16 | CC S3 | Call subroutine if carry set |
| CPO e16 | CPO $+8 | Call subroutine if parity odd |
| CPE e16 | CPE S4 | Call subroutine if parity even |
| CP e16 | CP GAMMA | Call subroutine if positive result |
| CM e16 | CM b1$c2 | Call subroutine if minus flag |
| RST e3 | RST Ø | Programmed "restart", equivalent to CALL 8*e3, except one byte call |

| | |
|---|---|
| RET | Return from subroutine |
| RNZ | Return if non zero flag set |
| RZ | Return if zero flag set |
| RNC | Return if no carry |
| RC | Return if carry flag set |
| RPO | Return if parity is odd |
| RPE | Return if parity is even |
| RP | Return if positive result |
| RM | Return if minus flag is set |

## 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).

| | | |
|---|---|---|
| MVI e3,e8 | MVI B,255 | Move immediate data to register A, B, C, D, E, H, L, or M (memory) |
| ADI e8 | ADI 1 | Add immediate operand to A without carry |
| ACI e8 | ACI ØFFH | Add immediate operand to A with carry |
| SUI e8 | SUI L + 3 | Subtract from A without borrow (carry) |
| SBI e8 | SBI L AND 11B | Subtract from A with borrow (carry) |
| ANI e8 | ANI $ AND 7FH | Logical "and" A with immediate data |
| XRI e8 | XRI 1111$ØØØØB | "Exclusive or" A with immediate data |
| ORI e8 | ORI L AND 1+1 | Logical "or" A with immediate data |
| CPI e8 | CPI 'a' | Compare A with immediate data (same as SUI except register A not changed) |
| LXI e3,e16 | LXI B,1ØØH | Load extended immediate to register pair (e3 must be equivalent to B,D,H, or SP) |

## 5.3. Increment and Decrement Instructions.

Instructions are provided in the 8Ø8Ø repetoire for incrementing or decrementing single and double precision registers. The instructions are

| | | |
|---|---|---|
| INR e3 | INR E | Single precision increment register (e3 produces one of A, B, C, D, E, H, L, M) |
| DCR e3 | DCR A | Single precision decrement register (e3 produces one of A, B, C, D, E, H, L, M) |
| INX e3 | INX SP | Double precision increment register pair (e3 must be equivalent to B,D,H, or SP) |
| DCX e3 | DCX B | Double precision decrement register pair (e3 must be equivalent to B,D,H, or SP) |

## 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 | MOV A,B | Move 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 |
| LDAX e3 | LDAX B | Load register A from computed address (e3 must produce either B or D) |
| STAX e3 | STAX D | Store register A to computed address (e3 must produce either B or D) |
| LHLD e16 | LHLD L1 | Load HL direct from location e16 (double precision load to H and L) |
| SHLD e16 | SHLD L5+x | Store HL direct to location e16 (double precision store from H and L to memory) |
| LDA e16 | LDA Gamma | Load register A from address e16 |
| STA e16 | STA X3-5 | Store register A into memory at e16 |
| POP e3 | POP PSW | Load register pair from stack, set SP (e3 must produce one of B, D, H, or PSW) |
| PUSH e3 | PUSH B | Store register pair into stack, set SP (e3 must produce one of B, D, H, or PSW) |
| IN e8 | IN Ø | Load register A with data from port e8 |
| OUT e8 | OUT 255 | Send data from register A to port e8 |
| XTHL | | Exchange data from top of stack with HL |
| PCHL | | Fill program counter with data from HL |
| SPHL | | Fill stack pointer with data from HL |
| XCHG | | Exchange DE pair with HL pair |

## 5.5. Arithmetic Logic Unit Operations.

Instructions which act upon the single precision accumulator to perform arithmetic and logic operations are

| | | |
|---|---|---|
| ADD e3 | ADD B | Add register given by e3 to accumulator without carry (e3 must produce one of A, B, C, D, E, H, or L) |
| ADC e3 | ADC L | Add register to A with carry, e3 as above |
| SUB e3 | SUB H | Subtract reg e3 from A without carry, e3 is defined as above |
| SBB e3 | SBB 2 | Subtract register e3 from A with carry, e3 defined as above |
| ANA e3 | ANA 1+1 | Logical "and" reg with A, e3 as above |
| XRA e3 | XRA A | "Exclusive or" with A, e3 as above |
| ORA e3 | ORA B | Logical "or" with A, e3 defined as above |
| CMP e3 | CMP H | Compare 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 8Ø8Ø 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

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 message 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.

```
ASM SORT↲   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.*↲

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 ;         SORT PROGRAM IN CP/M ASSEMBLY LANGUAGE
                      ;         START AT THE BEGINNING OF THE TRANSIENT PROGRAM AR
 0100 ←                         ORG     100H
      generated machine code
 0100 214601↲         SORT:     LXI     H,SW     ;ADDRESS SWITCH TOGGLE
 0103 3601                      MVI     M,1      ;SET TO 1 FOR FIRST ITERATION
 0105 214701                    LXI     H,I      ;ADDRESS INDEX
 0108 3600                      MVI     M,0      ;I = 0
                      ;
                      ;         COMPARE I WITH ARRAY SIZE
 010A 7E              COMP:     MOV     A,M      ;A REGISTER = I
 010B FE09                      CPI     N-1      ;CY SET IF I < (N-1)
 010D D21901                    JNC     CONT     ;CONTINUE IF I <= (N-2)
                      ;
                      ;         END OF ONE PASS THROUGH DATA
 0110 214601                    LXI     H,SW     ;CHECK FOR ZERO SWITCHES
 0113 7EB7C20001                MOV A,M! ORA A! JNZ SORT ;END OF SORT IF SW=0
                      ;
 0118 FF                        RST 7            ;GO TO THE DEBUGGER INSTEAD OF RE
                      ;truncated
                      ;         CONTINUE THIS PASS
                      ;         ADDRESSING I, SO LOAD AV(I) INTO REGISTERS
 0119 5F16002148CONT:           MOV E,A! MVI D,0! LXI H,AV! DAD D! DAD D
 0121 4E792346                  MOV C,M! MOV A,C! INX H! MOV B,M
                      ;         LOW ORDER BYTE IN A AND C, HIGH ORDER BYTE IN B
                      ;
                      ;         MOV H AND L TO ADDRESS AV(I+1)
 0125 23                        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)
 012B DA3F01                    JC      INCI     ;SKIP IF IN PROPER ORDER
                      ;
                      ;         CHECK FOR EQUAL VALUES
 012E B2CA3F01                  ORA D! JZ INCI ;SKIP IF AV(I) = AV(I+1)
 0132 56702B5E            MOV D,M! MOV M,B! DCX H! MOV E,M
 0136 712B722B73          MOV M,C! DCX H! MOV M,D! DCX H! MOV M,E
                     ;
                     ;    INCREMENT SWITCH COUNT
 013B 21460134            LXI H,SW! INR M
                     ;
                     ;    INCREMENT I
 013F 21470134C3INCI:     LXI H,I! INR M! JMP COMP
                     ;
                     ;    DATA DEFINITION SECTION
 0146 00             SW:  DB       0          ;RESERVE SPACE FOR SWITCH COUNT
 0147                I:   DS       1          ;SPACE FOR INDEX
 0148 050064001EAV:       DW       5,100,30,50,20,7,1000,300,100,-32767
 000A =               N   EQU      ($-AV)/2          ;COMPUTE N INSTEAD OF PRE
 015C    equate value     END
A>TYPE SORT.HEX

:10010000214601360121470136007EFE09D2190140
:100110002146017EB7C20001FF5F16002148011988
:10012000194E79234623965778239EDA3F01B2CAA7
:100130003F0156702B5E712B722B732146013421C7   machine code in
:07014000470134C30A01006E                     HEX format
:10014800050064001E00320014000700E8032C01BB
:0401580064000180BE
:0000000000
A>DDT SORT.HEX   start debug run

16K DDT VER 1.0
NEXT  PC
015C 0000  default address (no address on END statement)
-XP

P=0000 100  Change PC to 100

-UFFFF  untrace for 65535 steps
                                                                abort with
                                                                 rubout
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 LXI  H,0146*0100
-T10  trace $10_{16}$ steps

C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0100 LXI  H,0146
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0103 MVI  M,01
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0105 LXI  H,0147
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=0108 MVI  M,00
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=010A MOV  A,M
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010B CPI  09
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010D JNC  0119
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0110 LXI  H,0146
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0146 S=0100 P=0113 MOV  A,M
C1Z0M1E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0114 ORA  A
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0115 JNZ  0100
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0100 LXI  H,0146
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0103 MVI  M,01
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0105 LXI  H,0147
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=0108 MVI  M,00
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=010A MOV  A,M*010B
-A10D

010D  JC 119  change to a jump on carry                       stopped at
0110                                                            10BH
```

```
-XP
P=010B 100
```

reset program counter back to beginning of program

```
-T10
```

trace execution for 10H steps

```
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0100 LXI  H,0146
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0146 S=0100 P=0103 MVI  M,01
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0146 S=0100 P=0105 LXI  H,0147
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0108 MVI  M,00
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010A MOV  A,M
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010B CPI  09
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010D JC   0119
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0119 MOV  E,A
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=011A MVI  D,00
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=011C LXI  H,0148
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0148 S=0100 P=011F DAD  D
C0Z0M1E0I0 A=00 B=0000 D=0000 H=0148 S=0100 P=0120 DAD  D
C0Z0M1E0I0 A=00 B=0000 D=0000 H=0148 S=0100 P=0121 MOV  C,M
C0Z0M1E0I0 A=00 B=0005 D=0000 H=0148 S=0100 P=0122 MOV  A,C
C0Z0M1E0I0 A=05 B=0005 D=0000 H=0148 S=0100 P=0123 INX  H
C0Z0M1E0I0 A=05 B=0005 D=0000 H=0149 S=0100 P=0124 MOV  B,M*0125
-L100
```

Altered instruction

Automatic breakpoint

```
0100  LXI  H,0146
0103  MVI  M,01
0105  LXI  H,0147
0108  MVI  M,00
010A  MOV  A,M
010B  CPI  09
010D  JC   0119
0110  LXI  H,0146
0113  MOV  A,M
0114  ORA  A
0115  JNZ  0100
-L
```

list some code from 100H

```
0118  RST  07
0119  MOV  E,A
011A  MVI  D,00
011C  LXI  H,0148
```

list more

- abort list with rubout

```
-G,11B
```

start program from current PC (0125H) and run in real time to 11BH

```
*0127
```

stopped with an external interrupt 7 from front panel (program was looping indefinitely)

```
-T4
```

look at looping program in trace mode

```
C0Z0M0E0I0 A=38 B=0064 D=0006 H=0156 S=0100 P=0127 MOV  D,A
C0Z0M0E0I0 A=38 B=0064 D=3806 H=0156 S=0100 P=0128 MOV  A,B
C0Z0M0E0I0 A=00 B=0064 D=3806 H=0156 S=0100 P=0129 INX  H
C0Z0M0E0I0 A=00 B=0064 D=3806 H=0157 S=0100 P=012A SBB  M*012B
-D148

0148 05 00 07 00 14 00 1E 00 ........
0150 32 00 64 00 64 00 2C 01 E8 03 01 80 00 00 00 00 2.D.D.,.........
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

data is sorted, but program doesn't stop.

```
-GØ↲ return to CP/M

DDT SORT.HEX↲ reload the memory image

16K DDT VER 1.0
NEXT  PC
015C 0000
-XP

P=0000 100↲ set PC to beginning of program

-L10D↲ list bad opcode

010D  JNC  0119 ↙
0110  LXI  H,0146
- abort list with rubout

-A10D↲ assemble new opcode

010D  JC 119↲

0110↲

-L100↲ 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↲ change "switch" initialization to ØØ

0103  MVI M,0↲

0105↲

-^C  return to CP/M with ctl-C (GØ works as well)

SAVE 1 SORT.COM↲  save 1 page (256 bytes, from 100H to 1FFH) on disk in case
                                                   we have to reload later

A>DDT SORT.COM↲ restart DDT with
                 saved memory image
16K DDT VER 1.0
NEXT  PC
0200 0100 "COM" file always starts with address 100H
-G↲ run the program from PC=100H

*0118 programmed stop (RST 7) encountered
-D148
                                  ↙ data properly sorted
0148 05 00 07 00 14 00 1E 00 ........
0150 32 00 64 00 64 00 2C 01 E8 03 01 80 00 00 00 00 2.D.D.,.........
0160 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 ................

-GØ↲ return to CP/M
```

```
ED SORT.ASM↲  make changes to original program
 ctl-Z
*N,0^Z0TT↲  find next ",0"
        MVI     M,0     ;I = 0
*-↲ up one line in text
        LXI     H,I     ;ADDRESS INDEX
*-↲ up another line
        MVI     M,1     ;SET TO 1 FOR FIRST ITERATION
*KT↲ kill line and type next line
        LXI     H,I     ;ADDRESS INDEX
*I↲ insert new line
        MVI     M,0     ;ZERO SW
*T↲
        LXI     H,I     ;ADDRESS INDEX
*NJNC^Z0T↲
        JNC*T↲
        CONT    ;CONTINUE IF I <= (N-2)
*-2DIC^Z0LT↲
        JC      CONT    ;CONTINUE IF I <= (N-2)
*E↲
                 source from disk A
                 hex to disk A
ASM SORT.AAZ↲  skip prn file

CP/M ASSEMBLER - VER 1.0

015C next address to assemble
003H USE FACTOR
END OF ASSEMBLY

DDT SORT.HEX↲ test program changes

16K DDT VER 1.0
NEXT  PC
015C 0000
-G100↲

*0118
-D148↲
                                   data sorted
0148 05 00 07 00 14 00 1E 00 ........
0150 32 00 64 00 64 00 2C 01 E8 03 01 80 00 00 00 00 2.D.D.,..........
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

- abort with rubout

-G0↲ return to CP/M - program checks OK.
```

# THE CP/M 2.0 INTERFACE GUIDE

9

DIGITAL RESEARCH

Post Office Box 579, Pacific Grove, California 93950, (408) 649-3896

# CP/M 2.0 INTERFACE GUIDE

9

## Disclaimer

Digital Research 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, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

# CP/M 2.Ø INTERFACE GUIDE

Digital Research, Box 579
Pacific Grove, California

## 1. INTRODUCTION.

This manual describes CP/M, release 2, system organization including the structure of memory and system entry points. The intention 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.

CP/M is logically divided into four parts, called the Basic I/O System (BIOS), the Basic Disk Operating System (BDOS), the Console command processor (CCP), and the Transient Program Area (TPA). The BIOS is a hardware-dependent module which defines the exact low level interface to a particular computer system which is necessary for peripheral device I/O. Although a standard BIOS is supplied by Digital Research, explicit instructions are provided for field reconfiguration of the BIOS to match nearly any hardware environment (see the Digital Research manual entitled "CP/M Alteration Guide"). The BIOS and BDOS are logically combined into a single module 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 backup storage device. 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 and user programs are executed. The lower portion of memory is reserved for system information and is detailed later sections. Memory organization of the CP/M system in shown below:

```
            ---------------------------
   high    |                           |
   memory  |                           |
           |     FDOS (BDOS+BIOS)      |
   FBASE:  |                           |
            ---------------------------
           |                           |
           |            CCP            |
   CBASE:  |                           |
            ---------------------------
           |                           |
           |                           |
           |                           |
           |            TPA            |
           |                           |
   TBASE:  |                           |
            ---------------------------
           |     system  parameters    |
   BOOT:   |                           |
            ---------------------------
```

The exact memory addresses corresponding to BOOT, TBASE, CBASE, and FBASE vary from version to version, and are described fully in the "CP/M Alteration Guide." All standard CP/M versions, however, assume BOOT = 0000H, which is the base of random access memory. The machine code found at location BOOT performs a system "warm start" which loads and initializes the programs and variables necessary to return control to the CCP. Thus, transient programs need only jump to location BOOT to return control to CP/M at the command level. Further, the standard versions assume TBASE = BOOT+Ø1ØØH which is normally location Ø1ØØH. The principal entry point to the FDOS is at location BOOT+ØØØ5H (normally ØØØ5H) where a jump to FBASE is found. The address field at BOOT+ØØØ6H (normally ØØØ6H) contains the value of FBASE and can be used to determine the size of available memory, assuming the CCP is being overlayed by a transient program.

Transient programs are loaded into the TPA and executed as follows. The operator communicates with the CCP by typing command lines following each prompt. Each command line takes one of the forms:

```
command
command filel
command filel file2
```

where "command" is either a built-in function such as 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. The CCP loads the COM file from the disk into memory starting at TBASE and possibly extending up to CBASE.

If the command is followed by one or two file specifications, the CCP prepares one or two file control block (FCB) names in the system parameter area. These optional FCB's are in the form necessary to access files through the FDOS, and are described in the next section.

The transient program receives control from the CCP and begins execution, perhaps using the I/O facilities of the FDOS. The transient program is "called" from the CCP, and thus can simply return to the CCP upon completion of its processing, or can jump to BOOT to pass control back to CP/M. In the first case, the transient program must not use memory above CBASE, while in the latter case, memory up through FBASE-l is free.

The transient program may use the CP/M I/O facilities to communicate with the operator's console and peripheral devices, including the disk subsystem. The I/O system is accessed by passing a "function number" and an "information address" to CP/M through the FDOS entry point at BOOT+ØØØ5H. 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 to the CP/M FDOS. The FDOS, in turn, performs the operation and returns with either a disk read completion indication or an error number indicating that the disk read was unsuccessful. The function numbers and error indicators are given in below.

## 2. OPERATING SYSTEM CALL CONVENTIONS.

The purpose of this section is to provide detailed information for performing direct operating system calls from user programs. Many of the functions listed below, however, are more simply accessed through the I/O macro library provided with the MAC macro assembler, and listed in the Digital Research manual entitled "MAC Macro Assembler: Language Manual and Applications Guide."

CP/M facilities which are available for access by transient programs fall into two general categories: simple device I/O, and disk file I/O. The simple device operations include:

Read a Console Character
Write a Console Character
Read a Sequential Tape Character
Write a Sequential Tape Character
Write a List Device Character
Get or Set I/O Status
Print Console Buffer
Read Console Buffer
Interrogate Console Ready

The FDOS operations which perform disk Input/Output are

Disk System Reset
Drive Selection
File Creation
File Open
File Close
Directory Search
File Delete
File Rename
Random or Sequential Read
Random or Sequential Write
Interrogate Available Disks
Interrogate Selected Disk
Set DMA Address
Set/Reset File Indicators

As mentioned above, access to the FDOS functions is accomplished by passing a function number and information address through the primary entry point at location BOOT+0005H. In general, the function number is passed in register C with the information address in the double byte pair DE. Single byte values are returned in register A, with double byte values returned in HL (a zero value is returned when the function number is out of range). For reasons of compatibility, register A = L and register B = H upon return in all cases. Note that the register passing conventions of CP/M agree with those of Intel's PL/M systems programming language. The list of CP/M function numbers is given below.

| | |
|---|---|
| 0 | System Reset |
| 1 | Console Input |
| 2 | Console Output |
| 3 | Reader Input |
| 4 | Punch Output |
| 5 | List Output |
| 6 | Direct Console I/O |
| 7 | Get I/O Byte |
| 8 | Set I/O Byte |
| 9 | Print String |
| 10 | Read Console Buffer |
| 11 | Get Console Status |
| 12 | Return Version Number |
| 13 | Reset Disk System |
| 14 | Select Disk |
| 15 | Open File |
| 16 | Close File |
| 17 | Search for First |
| 18 | Search for Next |
| 19 | Delete File |
| 20 | Read Sequential |
| 21 | Write Sequential |
| 22 | Make File |
| 23 | Rename File |
| 24 | Return Login Vector |
| 25 | Return Current Disk |
| 26 | Set DMA Address |
| 27 | Get Addr(Alloc) |
| 28 | Write Protect Disk |
| 29 | Get R/O Vector |
| 30 | Set File Attributes |
| 31 | Get Addr(Disk Parms) |
| 32 | Set/Get User Code |
| 33 | Read Random |
| 34 | Write Random |
| 35 | Compute File Size |
| 36 | Set Random Record |

(Functions 28 and 32 should be avoided in application programs to maintain upward compatibility with MP/M.)

Upon entry to a transient program, the CCP leaves the stack pointer set to an eight level stack area with the CCP return address pushed onto the stack, leaving seven levels before overflow occurs. Although this stack is usually not used by a transient program (i.e., most transients return to the CCP though a jump to location 0000H), it is sufficiently large to make CP/M system calls since the FDOS switches to a local stack at system entry. The following assembly language program segment, for example, reads characters continuously until an asterisk is encountered, at which time control returns to the CCP (assuming a standard CP/M system with BOOT = 0000H):

```
BDOS    EQU     0005H       ;STANDARD CP/M ENTRY
CONIN   EQU     1           ;CONSOLE INPUT FUNCTION
;
        ORG     0100H       ;BASE OF TPA
NEXTC:  MVI     C,CONIN     ;READ NEXT CHARACTER
        CALL    BDOS        ;RETURN CHARACTER IN <A>
        CPI     '*'         ;END OF PROCESSING?
        JNZ     NEXTC       ;LOOP IF NOT
        RET                 ;RETURN TO CCP
        END
```

CP/M implements a named file structure on each disk, providing a logical organization which allows any particular file to contain any number of records from completely empty, to the full capacity of the drive. Each drive is logically distinct with a disk directory and file data area. The disk file names are in three parts: the drive select code, the file name consisting of one to eight non-blank characters, and the file type consisting of zero to three non-blank characters. The file type names the generic category of a particular file, while the file name distinguishes individual files in each category. The file types listed below name a few generic categories which have been established, although they are generally arbitrary:

| | | | |
|---|---|---|---|
| ASM | Assembler Source | PLI | PL/I Source File |
| PRN | Printer Listing | REL | Relocatable Module |
| HEX | Hex Machine Code | TEX | TEX Formatter Source |
| BAS | Basic Source File | BAK | ED Source Backup |
| INT | Intermediate Code | SYM | SID Symbol File |
| COM | CCP Command File | $$$ | Temporary File |

Source files are treated as a sequence of ASCII characters, where each "line" of the source file is followed by a carriage-return line-feed sequence (ØDH followed by ØAH). Thus one 128 byte CP/M record could contain several lines of source text. The end of an ASCII file is denoted by a control-Z character (1AH) or a real end of file, returned by the CP/M read operation. Control-Z characters embedded within machine code files (e.g., COM files) are ignored, however, and the end of file condition returned by CP/M is used to terminate read operations.

Files in CP/M can be thought of as a sequence of up to 65536 records of 128 bytes each, numbered from Ø through 65535, thus allowing a maximum of 8 megabytes per file. Note, however, that although the records may be considered logically contiguous, they may not be physically contiguous in the disk data area. Internally, all files are broken into 16K byte segments called logical extents, so that counters are easily maintained as 8-bit values. Although the decomposition into extents is discussed in the paragraphs which follow, they are of no particular consequence to the programmer since each extent is automatically accessed in both sequential and random access modes.

In the file operations starting with function number 15, DE usually addresses a file control block (FCB). Transient programs often use the default file control block area reserved by CP/M at location BOOT+ØØ5CH (normally ØØ5CH) for simple file operations. The basic unit of file information is a 128 byte record used for all file operations, thus a default location for disk I/O is provided by CP/M at location BOOT+ØØ8ØH (normally ØØ8ØH) which is the initial default DMA address (see function 26). All directory operations take place in a reserved area which does not affect write buffers as was the case in release 1, with the exception of Search First and Search Next, where compatibility is required.

The File Control Block (FCB) data area consists of a sequence of 33 bytes for sequential access and a series of 36 bytes in the case that the file is accessed randomly. The default file control block normally located at ØØ5CH can be used for random access files, since the three bytes starting at BOOT+ØØ7DH are available for this purpose. The FCB format is shown with the following fields:

```
---------------------------------------------------------------------
|dr|f1|f2|/ /|f8|t1|t2|t3|ex|s1|s2|rc|dØ|/ /|dn|cr|rØ|r1|r2|
---------------------------------------------------------------------
 ØØ Ø1 Ø2 ... Ø8 Ø9 1Ø 11 12 13 14 15 16 ... 31 32 33 34 35
```

where

```
   dr          drive code (Ø - 16)
               Ø => use default drive for file
               1 => auto disk select drive A,
               2 => auto disk select drive B,
               ...
               16=> auto disk select drive P.

   f1...f8     contain the file name in ASCII
               upper case, with high bit = Ø

   t1,t2,t3    contain the file type in ASCII
               upper case, with high bit = Ø
               t1', t2', and t3' denote the
               bit of these positions,
               t1' = 1 => Read/Only file,
               t2' = 1 => SYS file, no DIR list

    ex         contains the current extent number,
               normally set to ØØ by the user, but
               in range Ø - 31 during file I/O

    s1         reserved for internal system use

    s2         reserved for internal system use, set
               to zero on call to OPEN, MAKE, SEARCH

    rc         record count for extent "ex,"
               takes on values from Ø - 128

    dØ...dn    filled-in by CP/M, reserved for
               system use

    cr         current record to read or write in
               a sequential file operation, normally
               set to zero by user

    rØ,r1,r2   optional random record number in the
               range Ø-65535, with overflow to r2,
               rØ,r1 constitute a 16-bit value with
               low byte rØ, and high byte r1
```

Each file being accessed through CP/M must have a corresponding FCB which provides the name and allocation information for all subsequent file operations. When accessing files, it is the programmer's responsibility to fill the lower sixteen bytes of the FCB and initialize the "cr" field. Normally, bytes 1 through 11 are set to the ASCII character values for the file name and file type, while all other fields are zero.

FCB's are stored in a directory area of the disk, and are brought into central memory before proceeding with file operations (see the OPEN and MAKE functions). The memory copy of the FCB is updated as file operations take place and later recorded permanently on disk at the termination of the file operation (see the CLOSE command).

The CCP constructs the first sixteen bytes of two optional FCB's for a transient by scanning the remainder of the line following the transient name, denoted by "file1" and "file2" in the prototype command line described above, with unspecified fields set to ASCII blanks. The first FCB is constructed at location BOOT+005CH, and can be used as-is for subsequent file operations. The second FCB occupies the d0 ... dn portion of the first FCB, and must be moved to another area of memory before use. If, for example, the operator types

```
PROGNAME B:X.ZOT Y.ZAP
```

the file PROGNAME.COM is loaded into the TPA, and the default FCB at BOOT+005CH is initialized to drive code 2, file name "X" and file type "ZOT". The second drive code takes the default value 0, which is placed at BOOT+006CH, with the file name "Y" placed into location BOOT+006DH and file type "ZAP" located 8 bytes later at BOOT+0075H. All remaining fields through "cr" are set to zero. Note again that it is the programmer's responsibility to move this second file name and type to another area, usually a separate file control block, before opening the file which begins at BOOT+005CH, due to the fact that the open operation will overwrite the second name and type.

If no file names are specified in the original command, then the fields beginning at BOOT+005DH and BOOT+006DH contain blanks. In all cases, the CCP translates lower case alphabetics to upper case to be consistent with the CP/M file naming conventions.

As an added convenience, the default buffer area at location BOOT+0080H is initialized to the command line tail typed by the operator following the program name. The first position contains the number of characters, with the characters themselves following the character count. Given the above command line, the area beginning at BOOT+0080H is initialized as follows:

```
BOOT+0080H:
+00 +01 +02 +03 +04 +05 +06 +07 +08 +09 +10 +11 +12 +13 +14
 14 " " "B" ":" "X" "." "Z" "O" "T" " " "Y" "." "Z" "A" "P"
```

where the characters are translated to upper case ASCII with uninitialized memory following the last valid character. Again, it is the responsibility of the programmer to extract the information from this buffer before any file operations are performed, unless the default DMA address is explicitly changed.

The individual functions are described in detail in the pages which follow.

```
****************************************
*                                      *
*  FUNCTION Ø:  System Reset           *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  ØØH              *
****************************************
```

The system reset function returns control to the CP/M operating system at the CCP level. The CCP re-initializes the disk subsystem by selecting and logging-in disk drive A. This function has exactly the same effect as a jump to location BOOT.

```
****************************************
*                                      *
*  FUNCTION 1:  CONSOLE INPUT          *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  Ø1H              *
*                                      *
*  Returned   Value:                   *
*      Register   A:  ASCII Character  *
****************************************
```

The console input function reads the next console character to register A. Graphic characters, along with carriage return, line feed, and backspace (ctl-H) are echoed to the console. Tab characters (ctl-I) are expanded in columns of eight characters. A check is made for start/stop scroll (ctl-S) and start/stop printer echo (ctl-P). The FDOS does not return to the calling program until a character has been typed, thus suspending execution if a character is not ready.

```
****************************************
*                                      *
*  FUNCTION 2:  CONSOLE OUTPUT         *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  Ø2H              *
*      Register   E:  ASCII Character  *
*                                      *
****************************************
```

The ASCII character from register E is sent to the console device. Similar to function 1, tabs are expanded and checks are made for start/stop scroll and printer echo.

```
***************************************
*                                     *
*  FUNCTION 3:  READER INPUT          *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  Ø3H             *
*                                     *
*  Returned   Value:                  *
*      Register   A:  ASCII Character *
***************************************
```

The Reader Input function reads the next character from the logical reader into register A (see the IOBYTE definition in the "CP/M Alteration Guide"). Control does not return until the character has been read.

```
***************************************
*                                     *
*  FUNCTION 4:  PUNCH OUTPUT          *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  Ø4H             *
*      Register   E:  ASCII Character *
*                                     *
***************************************
```

The Punch Output function sends the character from register E to the logical punch device.

```
***************************************
*                                     *
*  FUNCTION 5:  LIST OUTPUT           *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  Ø5H             *
*      Register   E:  ASCII Character *
*                                     *
***************************************
```

The List Output function sends the ASCII character in register E to the logical listing device.

```
***************************************
*                                     *
*  FUNCTION 6:  DIRECT CONSOLE I/O    *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  Ø6H             *
*      Register   E:  ØFFH (input) or *
*                     char (output)   *
*                                     *
*  Returned   Value:                  *
*      Register   A:  char or status  *
*                     (no value)      *
***************************************
```

Direct console I/O is supported under CP/M for those specialized applications where unadorned console input and output is required. Use of this function should, in general, be avoided since it bypasses all of CP/M's normal control character functions (e.g., control-S and control-P). Programs which perform direct I/O through the BIOS under previous releases of CP/M, however, should be changed to use direct I/O under BDOS so that they can be fully supported under future releases of MP/M and CP/M.

Upon entry to function 6, register E either contains hexadecimal FF, denoting a console input request, or register E contains an ASCII character. If the input value is FF, then function 6 returns A = ØØ if no character is ready, otherwise A contains the next console input character.

If the input value in E is not FF, then function 6 assumes that E contains a valid ASCII character which is sent to the console.

```
***************************************
*                                     *
*  FUNCTION 7:  GET I/O BYTE          *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  Ø7H             *
*                                     *
*  Returned   Value:                  *
*      Register   A:  I/O Byte Value  *
***************************************
```

The Get I/O Byte function returns the current value of IOBYTE in register A. See the "CP/M Alteration Guide" for IOBYTE definition.

```
***************************************
*                                     *
*  FUNCTION 8:  SET I/O BYTE          *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  Ø8H             *
*      Register   E:  I/O Byte Value  *
*                                     *
***************************************
```

The Set I/O Byte function changes the system IOBYTE value to that given in register E.

```
***************************************
*                                     *
*  FUNCTION 9:  PRINT STRING          *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  Ø9H             *
*      Registers DE:  String Address  *
*                                     *
***************************************
```

The Print String function sends the character string stored in memory at the location given by DE to the console device, until a "$" is encountered in the string. Tabs are expanded as in function 2, and checks are made for start/stop scroll and printer echo.

```
****************************************
*                                      *
*  FUNCTION 1Ø: READ CONSOLE BUFFER    *
*                                      *
****************************************
*  Entry Parameters:                   *
*       Register   C:  ØAH             *
*       Registers DE:  Buffer Address  *
*                                      *
*  Returned   Value:                   *
*       Console Characters in Buffer   *
****************************************
```

The Read Buffer function reads a line of edited console input into a buffer addressed by registers DE. Console input is terminated when either the input buffer overflows. The Read Buffer takes the form:

```
    DE: +Ø +1 +2 +3 +4 +5 +6 +7 +8     . . .    +n
        -------------------------------------------
        |mx|nc|cl|c2|c3|c4|c5|c6|c7|    . . .   |??|
        -------------------------------------------
```

where "mx" is the maximum number of characters which the buffer will hold (1 to 255), "nc" is the number of characters read (set by FDOS upon return), followed by the characters read from the console. if nc < mx, then uninitialized positions follow the last character, denoted by "??" in the above figure. A number of control functions are recognized during line editing:

| | |
|---|---|
| rub/del | removes and echoes the last character |
| ctl-C | reboots when at the beginning of line |
| ctl-E | causes physical end of line |
| ctl-H | backspaces one character position |
| ctl-J | (line feed) terminates input line |
| ctl-M | (return) terminates input line |
| ctl-R | retypes the current line after new line |
| ctl-U | removes currnt line after new line |
| ctl-X | backspaces to beginning of current line |

Note also that certain functions which return the carriage to the leftmost position (e.g., ctl-X) do so only to the column position where the prompt ended (in earlier releases, the carriage returned to the extreme left margin). This convention makes operator data input and line correction more legible.

```
***************************************
*                                     *
*  FUNCTION 11: GET CONSOLE STATUS    *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  ØBH             *
*                                     *
*  Returned   Value:                  *
*      Register   A:  Console Status  *
***************************************
```

The Console Status function checks to see if a character has been typed at the console. If a character is ready, the value ØFFH is returned in register A. Otherwise a ØØH value is returned.

```
***************************************
*                                     *
*  FUNCTION 12: RETURN VERSION NUMBER *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  ØCH             *
*                                     *
*  Returned   Value:                  *
*      Registers HL:  Version Number  *
***************************************
```

Function 12 provides information which allows version independent programming. A two-byte value is returned, with H = ØØ designating the CP/M release (H = Ø1 for MP/M), and L = ØØ for all releases previous to 2.Ø. CP/M 2.Ø returns a hexadecimal 2Ø in register L, with subsequent version 2 releases in the hexadecimal range 21, 22, through 2F. Using function 12, for example, you can write application programs which provide both sequential and random access functions, with random access disabled when operating under early releases of CP/M.

```
****************************************
*                                      *
*  FUNCTION 13: RESET DISK SYSTEM      *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  ØDH              *
*                                      *
****************************************
```

The Reset Disk Function is used to programmatically restore the file system to a reset state where all disks are set to read/write (see functions 28 and 29), only disk drive A is selected, and the default DMA address is reset to BOOT+ØØ8ØH. This function can be used, for example, by an application program which requires a disk change without a system reboot.

```
****************************************
*                                      *
*  FUNCTION 14: SELECT DISK            *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  ØEH              *
*      Register   E:  Selected Disk    *
*                                      *
****************************************
```

The Select Disk function designates the disk drive named in register E as the default disk for subsequent file operations, with E = Ø for drive A, 1 for drive B, and so-forth through 15 corresponding to drive P in a full sixteen drive system. The drive is placed in an "on-line" status which, in particular, activates its directory until the next cold start, warm start, or disk system reset operation. If the disk media is changed while it is on-line, the drive automatically goes to a read/only status in a standard CP/M environment (see function 28). FCB's which specify drive code zero (dr = ØØH) automatically reference the currently selected default drive. Drive code values between 1 and 16, however, ignore the selected default drive and directly reference drives A through P.

```
***************************************
*                                     *
*  FUNCTION 15: OPEN FILE             *
*                                     *
***************************************
*  Entry Parameters:                  *
*       Register   C:  ØFH            *
*       Registers DE:  FCB Address    *
*                                     *
*  Returned   Value:                  *
*       Register   A:  Directory Code *
***************************************
```

The Open File operation is used to activate a file which currently exists in the disk directory for the currently active user number. The FDOS scans the referenced disk directory for a match in positions 1 through 14 of the FCB referenced by DE (byte s1 is automatically zeroed), where an ASCII question mark (3FH) matches any directory character in any of these positions. Normally, no question marks are included and, further, bytes "ex" and "s2" of the FCB are zero.

If a directory element is matched, the relevant directory information is copied into bytes dØ through dn of the FCB, thus allowing access to the files through subsequent read and write operations. Note that an existing file must not be accessed until a sucessful open operation is completed. Upon return, the open function returns a "directory code" with the value Ø through 3 if the open was successful, or ØFFH (255 decimal) if the file cannot be found. If question marks occur in the FCB then the first matching FCB is activated. Note that the current record ("cr") must be zeroed by the program if the file is to be accessed sequentially from the first record.

```
****************************************
*                                      *
*  FUNCTION 16: CLOSE FILE             *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  1ØH              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

The Close File function performs the inverse of the open file function. Given that the FCB addressed by DE has been previously activated through an open or make function (see functions 15 and 22), the close function permanently records the new FCB in the referenced disk directory. The FCB matching process for the close is identical to the open function. The directory code returned for a successful close operation is Ø, 1, 2, or 3, while a ØFFH (255 decimal) is returned if the file name cannot be found in the directory. A file need not be closed if only read operations have taken place. If write operations have occurred, however, the close operation is necessary to permanently record the new directory information.

```
***************************************
*                                     *
*  FUNCTION 17: SEARCH FOR FIRST      *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register    C:  11H            *
*      Registers DE:  FCB Address     *
*                                     *
*  Returned   Value:                  *
*      Register    A:  Directory Code *
***************************************
```

Search First scans the directory for a match with the file given by the FCB addressed by DE. The value 255 (hexadecimal FF) is returned if the file is not found, otherwise Ø, 1, 2, or 3 is returned indicating the file is present. In the case that the file is found, the current DMA address is filled with the record containing the directory entry, and the relative starting position is A * 32 (i.e., rotate the A register left 5 bits, or ADD A five times). Although not normally required for application programs, the directory information can be extracted from the buffer at this position.

An ASCII question mark (63 decimal, 3F hexadecimal) in any position from "f1" through "ex" matches the corresponding field of any directory entry on the default or auto-selected disk drive. If the "dr" field contains an ASCII question mark, then the auto disk select function is disabled, the default disk is searched, with the search function returning any matched entry, allocated or free, belonging to any user number. This latter function is not normally used by application programs, but does allow complete flexibility to scan all current directory values. If the "dr" field is not a question mark, the "s2" byte is automatically zeroed.

```
***************************************
*                                     *
*  FUNCTION 18: SEARCH FOR NEXT       *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register    C:  12H            *
*                                     *
*  Returned   Value:                  *
*      Register    A:  Directory Code *
***************************************
```

The Search Next function is similar to the Search First function, except that the directory scan continues from the last matched entry. Similar to function 17, function 18 returns the decimal value 255 in A when no more directory items match.

```
***************************************
*                                     *
*  FUNCTION 19: DELETE FILE           *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  13H             *
*      Registers DE:  FCB Address     *
*                                     *
*  Returned   Value:                  *
*      Register   A:  Directory Code  *
***************************************
```

The Delete File function removes files which match the FCB addressed by DE. The filename and type may contain ambiguous references (i.e., question marks in various positions), but the drive select code cannot be ambiguous, as in the Search and Search Next functions.

Function 19 returns a decimal 255 if the referenced file or files cannot be found, otherwise a value in the range Ø to 3 is returned.

```
***************************************
*                                     *
*  FUNCTION 2Ø: READ SEQUENTIAL       *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  14H             *
*      Registers DE:  FCB Address     *
*                                     *
*  Returned   Value:                  *
*      Register   A:  Directory Code  *
***************************************
```

Given that the FCB addressed by DE has been activated through an open or make function (numbers 15 and 22), the Read Sequential function reads the next 128 byte record from the file into memory at the current DMA address. the record is read from position "cr" of the extent, and the "cr" field is automatically incremented to the next record position. If the "cr" field overflows then the next logical extent is automatically opened and the "cr" field is reset to zero in preparation for the next read operation. The value ØØH is returned in the A register if the read operation was successful, while a non-zero value is returned if no data exists at the next record position (e.g., end of file occurs).

```
****************************************
*                                      *
*  FUNCTION 21: WRITE SEQUENTIAL       *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  15H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

Given that the FCb addressed by DE has been activated through an open or make function (numbers 15 and 22), the Write Sequential function writes the 128 byte data record at the current DMA address to the file named by the FCB. the record is placed at position "cr" of the file, and the "cr" field is automatically incremented to the next record position. If the "cr" field overflows then the next logical extent is automatically opened and the "cr" field is reset to zero in preparation for the next write operation. Write operations can take place into an existing file, in which case newly written records overlay those which already exist in the file. Register A = ØØH upon return from a successful write operation, while a non-zero value indicates an unsuccessful write due to a full disk.

```
****************************************
*                                      *
*  FUNCTION 22: MAKE FILE              *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  16H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

The Make File operation is similar to the open file operation except that the FCB must name a file which does not exist in the currently referenced disk directory (i.e., the one named explicitly by a non-zero "dr" code, or the default disk if "dr" is zero). The FDOS creates the file and initializes both the directory and main memory value to an empty file. The programmer must ensure that no duplicate file names occur, and a preceding delete operation is sufficient if there is any possibility of duplication. Upon return, register A = Ø, 1, 2, or 3 if the operation was successful and ØFFH (255 decimal) if no more directory space is available. The make function has the side-effect of activating the FCB and thus a subsequent open is not necessary.

```
****************************************
*                                      *
*  FUNCTION 23: RENAME FILE            *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register  C:  17H               *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register  A:  Directory Code    *
****************************************
```

The Rename function uses the FCB addressed by DE to change all occurrences of the file named in the first 16 bytes to the file named in the second 16 bytes. The drive code "dr" at position Ø is used to select the drive, while the drive code for the new file name at position 16 of the FCB is assumed to be zero. Upon return, register A is set to a value between Ø and 3 if the rename was successful, and ØFFH (255 decimal) if the first file name could not be found in the directory scan.

```
****************************************
*                                      *
*  FUNCTION 24: RETURN LOGIN VECTOR    *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register  C:  18H               *
*                                      *
*  Returned   Value:                   *
*      Registers HL:  Login Vector     *
****************************************
```

The login vector value returned by CP/M is a 16-bit value in HL, where the least significant bit of L corresponds to the first drive A, and the high order bit of H corresponds to the sixteenth drive, labelled P. A "Ø" bit indicates that the drive is not on-line, while a "1" bit marks an drive that is actively on-line due to an explicit disk drive selection, or an implicit drive select caused by a file operation which specified a non-zero "dr" field. Note that compatibility is maintained with earlier releases, since registers A and L contain the same values upon return.

```
****************************************
*                                      *
*  FUNCTION 25: RETURN CURRENT DISK    *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  19H              *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Current Disk     *
****************************************
```

Function 25 returns the currently selected default disk number in register A. The disk numbers range from Ø through 15 corresponding to drives A through P.

```
****************************************
*                                      *
*  FUNCTION 26: SET DMA ADDRESS        *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  1AH              *
*      Registers DE:  DMA Address      *
*                                      *
****************************************
```

"DMA" is an acronym for Direct Memory Address, which is often used in connection with disk controllers which directly access the memory of the mainframe computer to transfer data to and from the disk subsystem. Although many computer systems use non-DMA access (i.e., the data is transfered through programmed I/O operations), the DMA address has, in CP/M, come to mean the address at which the 128 byte data record resides before a disk write and after a disk read. Upon cold start, warm start, or disk system reset, the DMA address is automatically set to BOOT+ØØ8ØH. The Set DMA function, however, can be used to change this default value to address another area of memory where the data records reside. Thus, the DMA address becomes the value specified by DE until it is changed by a subsequent Set DMA function, cold start, warm start, or disk system reset.

```
***************************************
*                                     *
*  FUNCTION 27: GET ADDR(ALLOC)       *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  1BH             *
*                                     *
*  Returned   Value:                  *
*      Registers HL:  ALLOC Address   *
***************************************
```

An "allocation vector" is maintained in main memory for each on-line disk drive. Various system programs use the information provided by the allocation vector to determine the amount of remaining storage (see the STAT program). Function 27 returns the base address of the allocation vector for the currently selected disk drive. The allocation information may, however, be invalid if the selected disk has been marked read/only. Although this function is not normally used by application programs, additional details of the allocation vector are found in the "CP/M Alteration Guide."

```
***************************************
*                                     *
*  FUNCTION 28: WRITE PROTECT DISK    *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  1CH             *
*                                     *
***************************************
```

The disk write protect function provides temporary write protection for the currently selected disk. Any attempt to write to the disk, before the next cold or warm start operation produces the message

```
Bdos Err on d: R/O
```

```
***************************************
*                                     *
*  FUNCTION 29: GET READ/ONLY VECTOR  *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  1DH             *
*                                     *
*  Returned   Value:                  *
*      Registers HL:  R/O Vector Value*
***************************************
```

Function 29 returns a bit vector in register pair HL which indicates drives which have the temporary read/only bit set. Similar to function 24, the least significant bit corresponds to drive A, while the most significant bit corresponds to drive P. The R/O bit is set either by an explicit call to function 28, or by the automatic software mechanisms within CP/M which detect changed disks.

```
***************************************
*                                     *
*  FUNCTION 3Ø: SET FILE ATTRIBUTES   *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  1EH             *
*      Registers DE:  FCB Address     *
*                                     *
*  Returned   Value:                  *
*      Register   A:  Directory Code  *
***************************************
```

The Set File Attributes function allows programmatic manipulation of permanent indicators attached to files. In particular, the R/O and System attributes (t1' and t2') can be set or reset. The DE pair addresses an unambiguous file name with the appropriate attributes set or reset. Function 3Ø searches for a match, and changes the matched directory entry to contain the selected indicators. Indicators f1' through f4' are not presently used, but may be useful for applications programs, since they are not involved in the matching process during file open and close operations. Indicators f5' through f8' and t3' are reserved for future system expansion.

```
***************************************
*                                     *
*  FUNCTION 31: GET ADDR(DISK PARMS)  *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  1FH             *
*                                     *
*  Returned   Value:                  *
*      Registers HL:  DPB Address     *
***************************************
```

The address of the BIOS resident disk parameter block is returned in HL as a result of this function call. This address can be used for either of two purposes. First, the disk parameter values can be extracted for display and space computation purposes, or transient programs can dynamically change the values of current disk parameters when the disk environment changes, if required. Normally, application programs will not require this facility.

```
***************************************
*                                     *
*  FUNCTION 32: SET/GET USER CODE     *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  2ØH             *
*      Register   E:  ØFFH (get) or   *
*                     User Code (set) *
*                                     *
*  Returned   Value:                  *
*      Register   A:  Current Code or *
*                     (no value)      *
***************************************
```

An application program can change or interrogate the currently active user number by calling function 32. If register E = ØFFH, then the value of the current user number is returned in register A, where the value is in the range Ø to 31. If register E is not ØFFH, then the current user number is changed to the value of E (modulo 32).

```
***************************************
*                                     *
*  FUNCTION 33: READ RANDOM           *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register  C:  21H              *
*      Registers DE:  FCB Address     *
*                                     *
*  Returned  Value:                   *
*      Register  A:  Return Code      *
***************************************
```

The Read Random function is similar to the sequential file read operation of previous releases, except that the read operation takes place at a particular record number, selected by the 24-bit value constructed from the three byte field following the FCB (byte positions rØ at 33, r1 at 34, and r2 at 35). Note that the sequence of 24 bits is stored with least significant byte first (rØ), middle byte next (r1), and high byte last (r2). CP/M does not reference byte r2, except in computing the size of a file (function 35). Byte r2 must be zero, however, since a non-zero value indicates overflow past the end of file.

Thus, the rØ,r1 byte pair is treated as a double-byte, or "word" value, which contains the record to read. This value ranges from Ø to 65535, providing access to any particular record of the 8 megabyte file. In order to process a file using random access, the base extent (extent Ø) must first be opened. Although the base extent may or may not contain any allocated data, this ensures that the file is properly recorded in the directory, and is visible in DIR requests. The selected record number is then stored into the random record field (rØ,r1), and the BDOS is called to read the record. Upon return from the call, register A either contains an error code, as listed below, or the value ØØ indicating the operation was successful. In the latter case, the current DMA address contains the randomly accessed record. Note that contrary to the sequential read operation, the record number is not advanced. Thus, subsequent random read operations continue to read the same record.

Upon each random read operation, the logical extent and current record values are automatically set. Thus, the file can be sequentially read or written, starting from the current randomly accessed position. Note, however, that in this case, the last randomly read record will be re-read as you switch from random mode to sequential read, and the last record will be re-written as you switch to a sequential write operation. You can, of course, simply advance the random record position following each random read or write to obtain the effect of a sequential I/O operation.

Error codes returned in register A following a random read are listed below.

| | |
|---|---|
| 01 | reading unwritten data |
| 02 | (not returned in random mode) |
| 03 | cannot close current extent |
| 04 | seek to unwritten extent |
| 05 | (not returned in read mode) |
| 06 | seek past physical end of disk |

Error code 01 and 04 occur when a random read operation accesses a data block which has not been previously written, or an extent which has not been created, which are equivalent conditions. Error 3 does not normally occur under proper system operation, but can be cleared by simply re-reading, or re-opening extent zero as long as the disk is not physically write protected. Error code 06 occurs whenever byte r2 is non-zero under the current 2.0 release. Normally, non-zero return codes can be treated as missing data, with zero return codes indicating operation complete.

```
****************************************
*                                      *
*  FUNCTION 34: WRITE RANDOM           *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  22H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Return Code      *
****************************************
```

The Write Random operation is initiated similar to the Read Random call, except that data is written to the disk from the current DMA address. Further, if the disk extent or data block which is the target of the write has not yet been allocated, the allocation is performed before the write operation continues. As in the Read Random operation, the random record number is not changed as a result of the write. The logical extent number and current record positions of the file control block are set to correspond to the random record which is being written. Again, sequential read or write operations can commence following a random write, with the notation that the currently addressed record is either read or rewritten again as the sequential operation begins. You can also simply advance the random record position following each write to get the effect of a sequential write operation. Note that in particular, reading or writing the last record of an extent in random mode does not cause an automatic extent switch as it does in sequential mode.

The error codes returned by a random write are identical to the random read operation with the addition of error code Ø5, which indicates that a new extent cannot be created due to directory overflow.

```
***************************************
*                                     *
*  FUNCTION 35: COMPUTE FILE SIZE     *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  23H             *
*      Registers DE:  FCB Address     *
*                                     *
*  Returned   Value:                  *
*      Random Record Field Set        *
***************************************
```

When computing the size of a file, the DE register pair addresses an FCB in random mode format (bytes rØ, r1, and r2 are present). The FCB contains an unambiguous file name which is used in the directory scan. Upon return, the random record bytes contain the "virtual" file size which is, in effect, the record address of the record following the end of the file. if, following a call to function 35, the high record byte r2 is Ø1, then the file contains the maximum record count 65536. Otherwise, bytes rØ and r1 constitute a 16-bit value (rØ is the least significant byte, as before) which is the file size.

Data can be appended to the end of an existing file by simply calling function 35 to set the random record position to the end of file, then performing a sequence of random writes starting at the preset record address.

The virtual size of a file corresponds to the physical size when the file is written sequentially. If, instead, the file was created in random mode and "holes" exist in the allocation, then the file may in fact contain fewer records than the size indicates. If, for example, only the last record of an eight megabyte file is written in random mode (i.e., record number 65535), then the virtual size is 65536 records, although only one block of data is actually allocated.

```
***************************************
*                                     *
*  FUNCTION 36: SET RANDOM RECORD     *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register  C:  24H              *
*      Registers DE: FCB Address      *
*                                     *
*  Returned   Value:                  *
*      Random Record Field Set        *
***************************************
```

The Set Random Record function causes the BDOS to automatically produce the random record position from a file which has been read or written sequentially to a particular point. The function can be useful in two ways.

First, it is often necessary to initially read and scan a sequential file to extract the positions of various "key" fields. As each key is encountered, function 36 is called to compute the random record position for the data corresponding to this key. If the data unit size is 128 bytes, the resulting record position is placed into a table with the key for later retrieval. After scanning the entire file and tabularizing the keys and their record numbers, you can move instantly to a particular keyed record by performing a random read using the corresponding random record number which was saved earlier. The scheme is easily generalized when variable record lengths are involved since the program need only store the buffer-relative byte position along with the key and record number in order to find the exact starting position of the keyed data at a later time.

A second use of function 36 occurs when switching from a sequential read or write over to random read or write. A file is sequentially accessed to a particular point in the file, function 36 is called which sets the record number, and subsequent random read and write operations continue from the selected point in the file.

9

## 3. A SAMPLE FILE-TO-FILE COPY PROGRAM.

The program shown below provides a relatively simple example of file operations. The program source file is created as COPY.ASM using the CP/M ED program and then assembled using ASM or MAC, resulting in a "HEX" file. The LOAD program is the used to produce a COPY.COM file which executes directly under the CCP. The program begins by setting the stack pointer to a local area, and then proceeds to move the second name from the default area at 006CH to a 33-byte file control block called DFCB. The DFCB is then prepared for file operations by clearing the current record field. At this point, the source and destination FCB's are ready for processing since the SFCB at 005CH is properly set-up by the CCP upon entry to the COPY program. That is, the first name is placed into the default fcb, with the proper fields zeroed, including the current record field at 007CH. The program continues by opening the source file, deleting any exising destination file, and then creating the destination file. If all this is successful, the program loops at the label COPY until each record has been read from the source file and placed into the destination file. Upon completion of the data transfer, the destination file is closed and the program returns to the CCP command level by jumping to BOOT.

```
                ;       sample file-to-file copy program
                ;
                ;       at the ccp level, the command
                ;
                ;               copy a:x.y b:u.v
                ;
                ;       copies the file named x.y from drive
                ;       a to a file named u.v on drive b.
                ;
0000 =          boot    equ     0000h   ; system reboot
0005 =          bdos    equ     0005h   ; bdos entry point
005c =          fcbl    equ     005ch   ; first file name
005c =          sfcb    equ     fcbl    ; source fcb
006c =          fcb2    equ     006ch   ; second file name
0080 =          dbuff   equ     0080h   ; default buffer
0100 =          tpa     equ     0100h   ; beginning of tpa
                ;
0009 =          printf  equ     9       ; print buffer func#
000f =          openf   equ     15      ; open file func#
0010 =          closef  equ     16      ; close file func#
0013 =          deletef equ     19      ; delete file func#
0014 =          readf   equ     20      ; sequential read
0015 =          writef  equ     21      ; sequential write
0016 =          makef   equ     22      ; make file func#
                ;
0100                    org     tpa     ; beginning of tpa
0100 311b02             lxi     sp,stack; local stack
                ;
                ;       move second file name to dfcb
0103 0e10               mvi     c,16    ; half an fcb
```

```
Ø1Ø5 116cØØ          lxi     d,fcb2  ; source of move
Ø1Ø8 21daØ1          lxi     h,dfcb  ; destination fcb
Ø1Øb 1a      mfcb:   ldax    d       ; source fcb
Ø1Øc 13              inx     d       ; ready next
Ø1Ød 77              mov     m,a     ; dest fcb
Ø1Øe 23              inx     h       ; ready next
Ø1Øf Ød              dcr     c       ; count 16...Ø
Ø11Ø c2ØbØ1          jnz     mfcb    ; loop 16 times
             ;
             ;       name has been moved, zero cr
Ø113 af              xra     a       ; a = ØØh
Ø114 32faØ1          sta     dfcbcr  ; current rec = Ø
             ;
             ;       source and destination fcb's ready
             ;
Ø117 115cØØ          lxi     d,sfcb  ; source file
Ø11a cd69Ø1          call    open    ; error if 255
Ø11d 1187Ø1          lxi     d,nofile; ready message
Ø12Ø 3c              inr     a       ; 255 becomes Ø
Ø121 cc61Ø1          cz      finis   ; done if no file
             ;
             ;       source file open, prep destination
Ø124 11daØ1          lxi     d,dfcb  ; destination
Ø127 cd73Ø1          call    delete  ; remove if present
             ;
Ø12a 11daØ1          lxi     d,dfcb  ; destination
Ø12d cd82Ø1          call    make    ; create the file
Ø13Ø 1196Ø1          lxi     d,nodir ; ready message
Ø133 3c              inr     a       ; 255 becomes Ø
Ø134 cc61Ø1          cz      finis   ; done if no dir space
             ;
             ;       source file open, dest file open
             ;       copy until end of file on source
             ;
Ø137 115cØØ  copy:   lxi     d,sfcb  ; source
Ø13a cd78Ø1          call    read    ; read next record
Ø13d b7              ora     a       ; end of file?
Ø13e c251Ø1          jnz     eofile  ; skip write if so
             ;
             ;       not end of file, write the record
Ø141 11daØ1          lxi     d,dfcb  ; destination
Ø144 cd7dØ1          call    write   ; write record
Ø147 11a9Ø1          lxi     d,space ; ready message
Ø14a b7              ora     a       ; ØØ if write ok
Ø14b c461Ø1          cnz     finis   ; end if so
Ø14e c337Ø1          jmp     copy    ; loop until eof
             ;
             eofile: ; end of file, close destination
Ø151 11daØ1          lxi     d,dfcb  ; destination
Ø154 cd6eØ1          call    close   ; 255 if error
Ø157 21bbØ1          lxi     h,wrprot; ready message
Ø15a 3c              inr     a       ; 255 becomes ØØ
Ø15b cc61Ø1          cz      finis   ; shouldn't happen
             ;
             ;       copy operation complete, end
```

```
015e 11cc01             lxi     d,normal; ready message
                ;
                finis:  ; write message given by de, reboot
0161 0e09               mvi     c,printf
0163 cd0500             call    bdos    ; write message
0166 c30000             jmp     boot    ; reboot system
                ;
                ;       system interface subroutines
                ;       (all return directly from bdos)
                ;
0169 0e0f       open:   mvi     c,openf
016b c30500             jmp     bdos
                ;
016e 0e10       close:  mvi     c,closef
0170 c30500             jmp     bdos
                ;
0173 0e13       delete: mvi     c,deletef
0175 c30500             jmp     bdos
                ;
0178 0e14       read:   mvi     c,readf
017a c30500             jmp     bdos
                ;
017d 0e15       write:  mvi     c,writef
017f c30500             jmp     bdos
                ;
0182 0e16       make:   mvi     c,makef
0184 c30500             jmp     bdos
                ;
                ;       console messages
0187 6e6f20fnofile:     db      'no source file$'
0196 6e6f209nodir:      db      'no directory space$'
01a9 6f7574fspace:      db      'out of data space$'
01bb 7772695wrprot:     db      'write protected?$'
01cc 636f700normal:     db      'copy complete$'
                ;
                ;       data areas
01da            dfcb:   ds      33      ; destination fcb
01fa =          dfcbcr  equ     dfcb+32 ; current record
                ;
01fb                    ds      32      ; 16 level stack
                stack:
021b                    end
```

Note that there are several simplifications in this particular program. First, there are no checks for invalid file names which could, for example, contain ambiguous references. This situation could be detected by scanning the 32 byte default area starting at location 005CH for ASCII question marks. A check should also be made to ensure that the file names have, in fact, been included (check locations 005DH and 006DH for non-blank ASCII characters). Finally, a check should be made to ensure that the source and destination file names are different. A speed improvement could be made by buffering more data on each read operation. One could, for example, determine the size of memory by fetching FBASE from location 0006H and use the entire remaining portion of memory for a data buffer. In this case, the programmer simply resets the DMA address to the next successive 128 byte area before each read. Upon writing to the destination file, the DMA address is reset to the beginning of the buffer and incremented by 128 bytes to the end as each record is transferred to the destination file.

## 4. A SAMPLE FILE DUMP UTILITY.

The file dump program shown below is slightly more complex than the simple copy program given in the previous section. The dump program reads an input file, specified in the CCP command line, and displays the content of each record in hexadecimal format at the console. Note that the dump program saves the CCP's stack upon entry, resets the stack to a local area, and restores the CCP's stack before returning directly to the CCP. Thus, the dump program does not perform and warm start at the end of processing.

```
                    ; DUMP program reads input file and displays hex data
                    ;
0100                        org     100h
0005 =              bdos    equ     0005h   ;dos entry point
0001 =              cons    equ     1       ;read console
0002 =              typef   equ     2       ;type function
0009 =              printf  equ     9       ;buffer print entry
000b =              brkf    equ     11      ;break key function (true if char
000f =              openf   equ     15      ;file open
0014 =              readf   equ     20      ;read function
                    ;
005c =              fcb     equ     5ch     ;file control block address
0080 =              buff    equ     80h     ;input disk buffer address
                    ;
                    ;       non graphic characters
000d =              cr      equ     0dh     ;carriage return
000a =              lf      equ     0ah     ;line feed
                    ;
                    ;       file control block definitions
005c =              fcbdn   equ     fcb+0   ;disk name
005d =              fcbfn   equ     fcb+1   ;file name
0065 =              fcbft   equ     fcb+9   ;disk file type (3 characters)
0068 =              fcbrl   equ     fcb+12  ;file's current reel number
006b =              fcbrc   equ     fcb+15  ;file's record count (0 to 128)
007c =              fcbcr   equ     fcb+32  ;current (next) record number (0
007d =              fcbln   equ     fcb+33  ;fcb length
                    ;
                    ;       set up stack
0100 210000                 lxi     h,0
0103 39                     dad     sp
                    ;       entry stack pointer in hl from the ccp
0104 221502                 shld    oldsp
                    ;       set sp to local stack area (restored at finis)
0107 315702                 lxi     sp,stktop
                    ;       read and print successive buffers
010a cdc101                 call    setup   ;set up input file
010d feff                   cpi     255     ;255 if file not present
010f c21b01                 jnz     openok  ;skip if open is ok
                    ;
                    ;       file not there, give error message and return
0112 11f301                 lxi     d,opnmsg
0115 cd9c01                 call    err
0118 c35101                 jmp     finis   ;to return
                    ;
                    openok: ;open operation ok, set buffer index to end
011b 3e80                   mvi     a,80h
011d 321302                 sta     ibp     ;set buffer pointer to 80h
                    ;       hl contains next address to print
0120 210000                 lxi     h,0     ;start with 0000
                    ;
                    gloop:
0123 e5                     push    h       ;save line position
0124 cda201                 call    gnb
0127 e1                     pop     h       ;recall line position
0128 da5101                 jc      finis   ;carry set by gnb if end file
012b 47                     mov     b,a
                    ;       print hex values
                    ;       check for line fold
012c 7d                     mov     a,l
012d e60f                   ani     0fh     ;check low 4 bits
012f c24401                 jnz     nonum
                    ;       print line number
0132 cd7201                 call    crlf
                    ;
                    ;       check for break key
0135 cd5901                 call    break
                    ;       accum lsb = 1 if character ready
0138 0f                     rrc             ;into carry
0139 da5101                 jc      finis   ;don't print any more
                    ;
013c 7c                     mov     a,h
013d cd8f01                 call    phex
0140 7d                     mov     a,l
0141 cd8f01                 call    phex
                    nonum:
0144 23                     inx     h       ;to next line number
0145 3e20                   mvi     a,' '
0147 cd6501                 call    pchar
014a 78                     mov     a,b
014b cd8f01                 call    phex
014e c32301                 jmp     gloop
                    ;
                    finis:
                    ;       end of dump, return to ccp
                    ;       (note that a jmp to 0000h reboots)
0151 cd7201                 call    crlf
0154 2a1502                 lhld    oldsp
0157 f9                     sphl
                    ;       stack pointer contains ccp's stack location
0158 c9                     ret             ;to the ccp
                    ;
                    ;
                    ;       subroutines
                    ;
                    break:  ;check break key (actually any key will do)
0159 e5d5c5                 push h! push d! push b; environment saved
015c 0e0b                   mvi     c,brkf
015e cd0500                 call    bdos
0161 c1d1e1                 pop b! pop d! pop h; environment restored
```

```
0164 c9             ret
            ;
            pchar:  ;print a character
0165 e5d5c5         push h! push d! push b; saved
0168 0e02           mvi     c,typef
016a 5f             mov     e,a
016b cd0500         call    bdos
016e c1d1e1         pop b! pop d! pop h; restored
0171 c9             ret
            ;
            crlf:
0172 3e0d           mvi     a,cr
0174 cd6501         call    pchar
0177 3e0a           mvi     a,lf
0179 cd6501         call    pchar
017c c9             ret
            ;
            ;
            pnib:   ;print nibble in reg a
017d e60f           ani     0fh     ;low 4 bits
017f fe0a           cpi     10
0181 d28901         jnc     p10
            ;       less than or equal to 9
0184 c630           adi     '0'
0186 c38b01         jmp     prn
            ;
            ;       greater or equal to 10
0189 c637   p10:    adi     'a' - 10
018b cd6501 prn:    call    pchar
018e c9             ret
            ;
            phex:   ;print hex char in reg a
018f f5             push    psw
0190 0f             rrc
0191 0f             rrc
0192 0f             rrc
0193 0f             rrc
0194 cd7d01         call    pnib    ;print nibble
0197 f1             pop     psw
0198 cd7d01         call    pnib
019b c9             ret
            ;
            err:    ;print error message
            ;       d,e addresses message ending with "$"
019c 0e09           mvi     c,printf        ;print buffer function
019e cd0500         call    bdos
01a1 c9             ret
            ;
            ;
            gnb:    ;get next byte
01a2 3a1302         lda     ibp
01a5 fe80           cpi     80h
01a7 c2b301         jnz     g0
            ;       read another buffer
            ;
            ;
Ø1aa cdceØ1         call    diskr
Ø1ad b7             ora     a       ;zero value if read ok
Ø1ae cab3Ø1         jz      gØ      ;for another byte
            ;       end of data, return with carry set for eof
Ø1b1 37             stc
Ø1b2 c9             ret
            ;
            gØ:     ;read the byte at buff+reg a
Ø1b3 5f             mov     e,a     ;ls byte of buffer index
Ø1b4 16ØØ           mvi     d,Ø     ;double precision index to de
Ø1b6 3c             inr     a       ;index=index+1
Ø1b7 3213Ø2         sta     ibp     ;back to memory
            ;       pointer is incremented
            ;       save the current file address
Ø1ba 218ØØØ         lxi     h,buff
Ø1bd 19             dad     d
            ;       absolute character address is in hl
Ø1be 7e             mov     a,m
            ;       byte is in the accumulator
Ø1bf b7             ora     a       ;reset carry bit
Ø1cØ c9             ret
            ;
            setup:  ;set up file
            ;       open the file for input
Ø1c1 af             xra     a       ;zero to accum
Ø1c2 327cØØ         sta     fcbcr   ;clear current record
            ;
Ø1c5 115cØØ         lxi     d,fcb
Ø1c8 ØeØf           mvi     c,openf
Ø1ca cdØ5ØØ         call    bdos
            ;       255 in accum if open error
Ø1cd c9             ret
            ;
            diskr:  ;read disk file record
Ø1ce e5d5c5         push h! push d! push b
Ø1d1 115cØØ         lxi     d,fcb
Ø1d4 Øe14           mvi     c,readf
Ø1d6 cdØ5ØØ         call    bdos
Ø1d9 c1d1e1         pop b! pop d! pop h
Ø1dc c9             ret
            ;
            ;       fixed message area
Ø1dd 46494cØsignon: db      'file dump version 2.Ø$'
Ø1f3 ØdØa4eØopnmsg: db      cr,lf,'no input file present on disk$'

            ;       variable area
Ø213        ibp:    ds      2       ;input buffer pointer
Ø215        oldsp:  ds      2       ;entry sp value from ccp
            ;
            ;       stack area
Ø217                ds      64      ;reserve 32 level stack
            stktop:
            ;
Ø257                end
```

## 5. A SAMPLE RANDOM ACCESS PROGRAM.

This manual is concluded with a rather extensive, but complete example of random access operation. The program listed below performs the simple function of reading or writing random records upon command from the terminal. Given that the program has been created, assembled, and placed into a file labelled RANDOM.COM, the CCP level command:

```
RANDOM X.DAT
```

starts the test program. The program looks for a file by the name X.DAT (in this particular case) and, if found, proceeds to prompt the console for input. If not found, the file is created before the prompt is given. Each prompt takes the form

```
next command?
```

and is followed by operator input, terminated by a carriage return. The input commands take the form

```
nW   nR   Q
```

where n is an integer value in the range Ø to 65535, and W, R, and Q are simple command characters corresponding to random write, random read, and quit processing, respectively. If the W command is issued, the RANDOM program issues the prompt

```
type data:
```

The operator then responds by typing up to 127 characters, followed by a carriage return. RANDOM then writes the character string into the X.DAT file at record n. If the R command is issued, RANDOM reads record number n and displays the string value at the console. If the Q command is issued, the X.DAT file is closed, and the program returns to the console command processor. In the interest of brevity, the only error message is

```
error, try again
```

The program begins with an initialization section where the input file is opened or created, followed by a continuous loop at the label "ready" where the individual commands are interpreted. The default file control block at ØØ5CH and the default buffer at ØØ8ØH are used in all disk operations. The utility subroutines then follow, which contain the principal input line processor, called "readc." This particular program shows the elements of random access processing, and can be used as the basis for further program development.

```
           ;***************************************************
           ;*                                                 *
           ;* sample random access program for cp/m 2.0       *
           ;*                                                 *
           ;***************************************************
0100                org     100h    ;base of tpa
           ;
0000 =     reboot   equ     0000h   ;system reboot
0005 =     bdos     equ     0005h   ;bdos entry point
           ;
0001 =     coninp   equ     1       ;console input function
0002 =     conout   equ     2       ;console output function
0009 =     pstring  equ     9       ;print string until '$'
000a =     rstring  equ     10      ;read console buffer
000c =     version  equ     12      ;return version number
000f =     openf    equ     15      ;file open function
0010 =     closef   equ     16      ;close function
0016 =     makef    equ     22      ;make file function
0021 =     readr    equ     33      ;read random
0022 =     writer   equ     34      ;write random
           ;
005c =     fcb      equ     005ch   ;default file control block
007d =     ranrec   equ     fcb+33  ;random record position
007f =     ranovf   equ     fcb+35  ;high order (overflow) byte
0080 =     buff     equ     0080h   ;buffer address
           ;
000d =     cr       equ     0dh     ;carriage return
000a =     lf       equ     0ah     ;line feed
           ;
           ;***************************************************
           ;*                                                 *
           ;* load SP, set-up file for random access          *
           ;*                                                 *
           ;***************************************************
0100 31bc0          lxi     sp,stack
           ;
           ;        version 2.0?
0103 0e0c           mvi     c,version
0105 cd050          call    bdos
0108 fe20           cpi     20h     ;version 2.0 or better?
010a d2160          jnc     versok
           ;        bad version, message and go back
010d 111b0          lxi     d,badver
0110 cdda0          call    print
0113 c3000          jmp     reboot
           ;
           versok:
           ;        correct version for random access
0116 0e0f           mvi     c,openf ;open default fcb
0118 115c0          lxi     d,fcb
011b cd050          call    bdos
011e 3c             inr     a       ;err 255 becomes zero
011f c2370          jnz     ready
           ;
           ;        cannot open file, so create it
```

```
Ø122 Øe16          mvi     c,makef
Ø124 115cØ         lxi     d,fcb
Ø127 cdØ5Ø         call    bdos
Ø12a 3c            inr     a       ;err 255 becomes zero
Ø12b c237Ø         jnz     ready
           ;
           ;       cannot create file, directory full
Ø12e 113aØ         lxi     d,nospace
Ø131 cddaØ         call    print
Ø134 c3ØØØ         jmp     reboot  ;back to ccp
           ;
           ;****************************************************
           ;*                                                  *
           ;*  loop back to "ready" after each command          *
           ;*                                                  *
           ;****************************************************
           ;
           ready:
           ;       file is ready for processing
           ;
Ø137 cde5Ø         call    readcom ;read next command
Ø13a 227dØ         shld    ranrec  ;store input record#
Ø13d 217fØ         lxi     h,ranovf
Ø14Ø 36ØØ          mvi     m,Ø     ;clear high byte if set
Ø142 fe51          cpi     'Q'     ;quit?
Ø144 c256Ø         jnz     notq
           ;
           ;       quit processing, close file
Ø147 Øe1Ø          mvi     c,closef
Ø149 115cØ         lxi     d,fcb
Ø14c cdØ5Ø         call    bdos
Ø14f 3c            inr     a       ;err 255 becomes Ø
Ø15Ø cab9Ø         jz      error   ;error message, retry
Ø153 c3ØØØ         jmp     reboot  ;back to ccp
           ;
           ;****************************************************
           ;*                                                  *
           ;* end of quit command, process write               *
           ;*                                                  *
           ;****************************************************
           notq:
           ;       not the quit command, random write?
Ø156 fe57          cpi     'W'
Ø158 c289Ø         jnz     notw
           ;
           ;       this is a random write, fill buffer until cr
Ø15b 114dØ         lxi     d,datmsg
Ø15e cddaØ         call    print   ;data prompt
Ø161 Øe7f          mvi     c,127   ;up to 127 characters
Ø163 218ØØ         lxi     h,buff  ;destination
           rloop:  ;read next character to buff
Ø166 c5            push    b       ;save counter
Ø167 e5            push    h       ;next destination
Ø168 cdc2Ø         call    getchr  ;character to a
Ø16b e1            pop     h       ;restore counter
```

```
Ø16c c1            pop     b       ;restore next to fill
Ø16d feØd          cpi     cr      ;end of line?
Ø16f ca78Ø         jz      erloop
           ;       not end, store character
Ø172 77            mov     m,a
Ø173 23            inx     h       ;next to fill
Ø174 Ød            dcr     c       ;counter goes down
Ø175 c266Ø         jnz     rloop   ;end of buffer?
           erloop:
           ;       end of read loop, store ØØ
Ø178 36ØØ          mvi     m,Ø
           ;
           ;       write the record to selected record number
Ø17a Øe22          mvi     c,writer
Ø17c 115cØ         lxi     d,fcb
Ø17f cdØ5Ø         call    bdos
Ø182 b7            ora     a       ;error code zero?
Ø183 c2b9Ø         jnz     error   ;message if not
Ø186 c337Ø         jmp     ready   ;for another record
           ;
           ;***************************************************
           ;*                                                 *
           ;* end of write command, process read              *
           ;*                                                 *
           ;***************************************************
           notw:
           ;       not a write command, read record?
Ø189 fe52          cpi     'R'
Ø18b c2b9Ø         jnz     error   ;skip if not
           ;
           ;       read random record
Ø18e Øe21          mvi     c,readr
Ø19Ø 115cØ         lxi     d,fcb
Ø193 cdØ5Ø         call    bdos
Ø196 b7            ora     a       ;return code ØØ?
Ø197 c2b9Ø         jnz     error
           ;
           ;       read was successful, write to console
Ø19a cdcfØ         call    crlf    ;new line
Ø19d Øe8Ø          mvi     c,128   ;max 128 characters
Ø19f 218ØØ         lxi     h,buff  ;next to get
           wloop:
Ø1a2 7e            mov     a,m     ;next character
Ø1a3 23            inx     h       ;next to get
Ø1a4 e67f          ani     7fh     ;mask parity
Ø1a6 ca37Ø         jz      ready   ;for another command if ØØ
Ø1a9 c5            push    b       ;save counter
Ø1aa e5            push    h       ;save next to get
Ø1ab fe2Ø          cpi     ' '     ;graphic?
Ø1ad d4c8Ø         cnc     putchr  ;skip output if not
Ø1bØ e1            pop     h
Ø1b1 c1            pop     b
Ø1b2 Ød            dcr     c       ;count=count-1
Ø1b3 c2a2Ø         jnz     wloop
Ø1b6 c337Ø         jmp     ready
```

```
          ;
          ;*****************************************************
          ;*                                                   *
          ;* end of read command, all errors end-up here       *
          ;*                                                   *
          ;*****************************************************
          ;
          error:
Ølb9 1159Ø        lxi     d,errmsg
Ølbc cddaØ        call    print
Ølbf c337Ø        jmp     ready
          ;
          ;*****************************************************
          ;*                                                   *
          ;* utility subroutines for console i/o               *
          ;*                                                   *
          ;*****************************************************
          getchr:
                  ;read next console character to a
Ølc2 ØeØ1         mvi     c,coninp
Ølc4 cdØ5Ø        call    bdos
Ølc7 c9           ret
          ;
          putchr:
                  ;write character from a to console
Ølc8 ØeØ2         mvi     c,conout
Ølca 5f           mov     e,a     ;character to send
Ølcb cdØ5Ø        call    bdos    ;send character
Ølce c9           ret
          ;
          crlf:
                  ;send carriage return line feed
Ølcf 3eØd         mvi     a,cr    ;carriage return
Øld1 cdc8Ø        call    putchr
Øld4 3eØa         mvi     a,lf    ;line feed
Øld6 cdc8Ø        call    putchr
Øld9 c9           ret
          ;
          print:
                  ;print the buffer addressed by de until $
Ølda d5           push    d
Øldb cdcfØ        call    crlf
Ølde dl           pop     d       ;new line
Øldf ØeØ9         mvi     c,pstring
Ølel cdØ5Ø        call    bdos    ;print the string
Øle4 c9           ret
          ;
          readcom:
                  ;read the next command line to the conbuf
Øle5 116bØ        lxi     d,prompt
Øle8 cddaØ        call    print   ;command?
Øleb ØeØa         mvi     c,rstring
Øled 117aØ        lxi     d,conbuf
ØlfØ cdØ5Ø        call    bdos    ;read command line
          ;       command line is present, scan it
```

```
Ø1f3 21ØØØ         lxi     h,Ø     ;start with ØØØØ
Ø1f6 117cØ         lxi     d,conlin;command line
Ø1f9 1a    readc:  ldax    d       ;next command character
Ø1fa 13            inx     d       ;to next command position
Ø1fb b7            ora     a       ;cannot be end of command
Ø1fc c8            rz
           ;       not zero, numeric?
Ø1fd d63Ø          sui     'Ø'
Ø1ff feØa          cpi     1Ø      ;carry if numeric
Ø2Ø1 d213Ø         jnc     endrd
           ;       add-in next digit
Ø2Ø4 29            dad     h       ;*2
Ø2Ø5 4d            mov     c,l
Ø2Ø6 44            mov     b,h     ;bc = value * 2
Ø2Ø7 29            dad     h       ;*4
Ø2Ø8 29            dad     h       ;*8
Ø2Ø9 Ø9            dad     b       ;*2 + *8 = *1Ø
Ø2Øa 85            add     l       ;+digit
Ø2Øb 6f            mov     l,a
Ø2Øc d2f9Ø         jnc     readc   ;for another char
Ø2Øf 24            inr     h       ;overflow
Ø21Ø c3f9Ø         jmp     readc   ;for another char
           endrd:
           ;       end of read, restore value in a
Ø213 c63Ø          adi     'Ø'     ;command
Ø215 fe61          cpi     'a'     ;translate case?
Ø217 d8            rc
           ;       lower case, mask lower case bits
Ø218 e65f          ani     1Ø1$1111b
Ø21a c9            ret
           ;
           ;****************************************************
           ;*                                                  *
           ;* string data area for console messages            *
           ;*                                                  *
           ;****************************************************
           badver:
Ø21b 536f79        db      'sorry, you need cp/m version 2$'
           nospace:
Ø23a 4e6f29        db      'no directory space$'
           datmsg:
Ø24d 54797Ø        db      'type data: $'
           errmsg:
Ø259 457272        db      'error, try again.$'
           prompt:
Ø26b 4e657Ø        db      'next command? $'
           ;
           ;***************************************************
           ;*                                                 *
           ;* fixed and variable data area                    *
           ;*                                                 *
           ;***************************************************
027a 21    conbuf: db      conlen  ;length of console buffer
027b       consiz: ds      1       ;resulting size after read
027c       conlin: ds      32      ;length 32 buffer
0021 =     conlen  equ     $-consiz
           ;
029c               ds      32      ;16 level stack
           stack:
02bc               end
```

Again, major improvements could be made to this particular program to enhance its operation. In fact, with some work, this program could evolve into a simple data base management system. One could, for example, assume a standard record size of 128 bytes, consisting of arbitrary fields within the record. A program, called GETKEY, could be developed which first reads a sequential file and extracts a specific field defined by the operator. For example, the command

```
GETKEY NAMES.DAT  LASTNAME 10 20
```

would cause GETKEY to read the data base file NAMES.DAT and extract the "LASTNAME" field from each record, starting at position 10 and ending at character 20. GETKEY builds a table in memory consisting of each particular LASTNAME field, along with its 16-bit record number location within the file. The GETKEY program then sorts this list, and writes a new file, called LASTNAME.KEY, which is an alphabetical list of LASTNAME fields with their corresponding record numbers. (This list is called an "inverted index" in information retrieval parlance.)

Rename the program shown above as QUERY, and massage it a bit so that it reads a sorted key file into memory. The command line might appear as:

```
QUERY NAMES.DAT LASTNAME.KEY
```

Instead of reading a number, the QUERY program reads an alphanumeric string which is a particular key to find in the NAMES.DAT data base. Since the LASTNAME.KEY list is sorted, you can find a particular entry quite rapidly by performing a "binary search," similar to looking up a name in the telephone book. That is, starting at both ends of the list, you examine the entry halfway in between and, if not matched, split either the upper half or the lower half for the next search. You'll quickly reach the item you're looking for (in log2(n) steps) where you'll find the corresponding record number. Fetch and display this record at the console, just as we have done in the program shown above.

At this point you're just getting started. With a little more work, you can allow a fixed grouping size which differs from the 128 byte record shown above. This is accomplished by keeping track of the record number as well as the byte offset within the record. Knowing the group size, you randomly access the record containing the proper group, offset to the beginning of the group within the record read sequentially until the group size has been exhausted.

Finally, you can improve QUERY considerably by allowing boolean expressions which compute the set of records which satisfy several relationships, such as a LASTNAME between HARDY and LAUREL, and an AGE less than 45. Display all the records which fit this description. Finally, if your lists are getting too big to fit into memory, randomly access your key files from the disk as well. One note of consolation after all this work: if you make it through the project, you'll have no more need for this manual!

FOR COMPUTER TO COMPUTER

9

## 6. SYSTEM FUNCTION SUMMARY.

| FUNC | FUNCTION NAME | INPUT PARAMETERS | OUTPUT RESULTS |
|---|---|---|---|
| Ø | System Reset | none | none |
| 1 | Console Input | none | A = char |
| 2 | Console Output | E = char | none |
| 3 | Reader Input | none | A = char |
| 4 | Punch Output | E = char | none |
| 5 | List Output | E = char | none |
| 6 | Direct Console I/O | see def | see def |
| 7 | Get I/O Byte | none | A = IOBYTE |
| 8 | Set I/O Byte | E = IOBYTE | none |
| 9 | Print String | DE = .Buffer | none |
| 1Ø | Read Console Buffer | DE = .Buffer | see def |
| 11 | Get Console Status | none | A = ØØ/FF |
| 12 | Return Version Number | none | HL= Version* |
| 13 | Reset Disk System | none | see def |
| 14 | Select Disk | E = Disk Number | see def |
| 15 | Open File | DE = .FCB | A = Dir Code |
| 16 | Close File | DE = .FCB | A = Dir Code |
| 17 | Search for First | DE = .FCB | A = Dir Code |
| 18 | Search for Next | none | A = Dir Code |
| 19 | Delete File | DE = .FCB | A = Dir Code |
| 2Ø | Read Sequential | DE = .FCB | A = Err Code |
| 21 | Write Sequential | DE = .FCB | A = Err Code |
| 22 | Make File | DE = .FCB | A = Dir Code |
| 23 | Rename File | DE = .FCB | A = Dir Code |
| 24 | Return Login Vector | none | HL= Login Vect* |
| 25 | Return Current Disk | none | A = Cur Disk# |
| 26 | Set DMA Address | DE = .DMA | none |
| 27 | Get Addr(Alloc) | none | HL= .Alloc |
| 28 | Write Protect Disk | none | see def |
| 29 | Get R/O Vector | none | HL= R/O Vect* |
| 3Ø | Set File Attributes | DE = .FCB | see def |
| 31 | Get Addr(disk parms) | none | HL= .DPB |
| 32 | Set/Get User Code | see def | see def |
| 33 | Read Random | DE = .FCB | A = Err Code |
| 34 | Write Random | DE = .FCB | A = Err Code |
| 35 | Compute File Size | DE = .FCB | rØ, r1, r2 |
| 36 | Set Random Record | DE = .FCB | rØ, r1, r2 |

* Note that A = L, and B = H upon return

# THE CP/M 2.0
# SYSTEM ALTERATION GUIDE

1

DIGITAL RESEARCH

Post Office Box 579, Pacific Grove, California 93950, (408) 649-3896

# CP/M 2.0 ALTERATION GUIDE

10

## Copyright

## Disclaimer

Digital Research 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, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

## Trademarks

# CP/M 2.Ø ALTERATION GUIDE

Digital Research, Box 579
Pacific Grove, California

# 1. INTRODUCTION

The standard CP/M system assumes operation on an Intel MDS-8ØØ 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 any IBM-3741 format compatible drive controller and other peripheral devices.

Although standard CP/M 2.Ø is configured for single density floppy disks, field-alteration features allow adaptation to a wide variety of disk subsystems from single drive minidisks through high-capacity "hard disk" systems. In order to simplify the following adaptation process, we assume that CP/M 2.Ø will first be configured for single density floppy disks where minimal editing and debugging tools are available. If an earlier version of CP/M is available, the customizing process is eased considerably. In this latter case, you may wish to briefly review the system generation process, and skip to later sections which discuss system alteration for non-standard disk systems.

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 your new BIOS into CP/M.

If CP/M is being tailored to your computer system for the first time, the new BIOS requires some relatively simple software development and testing. The standard BIOS is listed in Appendix B, and can be used as a model for the customized package. A skeletal version of the BIOS is given in Appendix C which can serve as the basis 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 can be 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 D. 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 E 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 shown with a following "H" which denotes the hexadecimal radix, and are given for a 20K 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 - 20K. Values for b in various standard memory sizes are

| | |
|---|---|
| 24K: | b = 24K - 20K = 4K = 1000H |
| 32K: | b = 32K - 20K = 12K = 3000H |
| 40K: | b = 40K - 20K = 20K = 5000H |
| 48K: | b = 48K - 20K = 28K = 7000H |
| 56K: | b = 56K - 20K = 36K = 9000H |
| 62K: | b = 62K - 20K = 42K = A800H |
| 64K: | b = 64K - 20K = 44K = B000H |

Note: The standard distribution version of CP/M is set for operation within a 20K memory system. Therefore, you must first bring up the 20K 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 3380H. Code GETSYS so that it starts at location 100H (base of the TPA), as shown in the first part of Appendix d.

(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 3380H (the operating system actually starts 128 bytes later at 3400H).

(4) Review Section 4 and write the PUTSYS program which writes memory starting at 3380H 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 D.

(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 B, and write a simple version which performs a similar function for the customized environment. Use the program given in Appendix C as a model. Call this new BIOS by the name CBIOS (customized BIOS). Implement only the primitive disk operations on a single drive, and simple console input/output functions in this phase.

(7) Test CBIOS completely to ensure that it properly performs console character I/O and disk reads and writes. Be especially careful to ensure that no disk write operations occur accidently 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 placed between locations 4AØØH and 4FFFH. Read the CP/M system using GETSYS and replace the BIOS segment by the new CBIOS 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.

(1Ø) Use GETSYS to bring the copied memory image from the test diskette back into memory at 338ØH, 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 4AØØH. The cold start routine will initialize page zero, then jump to the CCP at location 34ØØH which will call the BDOS, which will call the CBIOS. The CBIOS will be asked by the CCP to read sixteen sectors on track 2, and if successful, CP/M will type "A>", the system prompt.

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 CBIOS.

(11) Upon completion of step (1Ø), 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 retry.

(12) Then 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
```

10

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 Ø, 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 (ctl-C is sufficient) when the 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 procedures. You should take the time to become familiar 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 CBIOS 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
Digital Research

on each copy which is made with your COPY program.

(2Ø) Modify your CBIOS to include the extra functions for punches, readers, signon messages, and so-forth, and add the facilities for a additional disk drives, if desired. 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 CBIOS 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, (assuming media compatiblity, of course) 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 into a named disk file. The disk file can then be loaded, examined, patched, and replaced using the debugger, and system generation program. For further details on the operation of these programs, see the "Guide to CP/M Features and Facilities" manual.

Your CBIOS and BOOT can be modified using ED, 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.

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 2.Ø
            READY FOR "SYSGEN" OR
            "SAVE 34 CPMxx.COM"
```

At this point, an image of a CP/M in the TPA configured for the requested memory size. The memory image is at location Ø9ØØH through 227FH. (i.e., The BOOT is at Ø9ØØH, the CCP is at 98ØH, the BDOS starts at 118ØH, and the BIOS is at 1F8ØH.) Note that the memory image has the standard MDS-8ØØ 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 34 CPMxx.COM
```

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
            23ØØ  Ø1ØØ
            -                               (The DDT prompt)
```

You can then use the display and disassembly commands to examine

(All Information Contained Herein is Proprietary to Digital Research.)

portions of the memory image between 900H and 227FH. 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 900H (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 20K system, for example, the CCP resides at the CP/M address 3400H, but is placed into memory at 980H by the SYSGEN program. Thus, the negative bias, denoted by n, satisfies

```
        3400H + n = 980H, or n = 980H - 3400H
```

Assuming two's complement arithmetic, n = D580H, which can be checked by

```
        3400H + D580H = 10980H = 0980H (ignoring high-order
                                        overflow).
```

Note that for larger systems, n satisfies

```
        (3400H+b) + n = 980H, or
        n = 980H - (3400H + b), or
        n = D580H - b.
```

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

| memory size | bias b | negative offset n |
|---|---|---|
| 20K | 0000H | D580H - 0000H = D580H |
| 24K | 1000H | D580H - 1000H = C580H |
| 32K | 3000H | D580H - 3000H = A580H |
| 40K | 5000H | D580H - 5000H = 8580H |
| 48K | 7000H | D580H - 7000H = 6580H |
| 56K | 9000H | D580H - 9000H = 4580H |
| 62K | A800H | D580H - A800H = 2D80H |
| 64K | B000H | D580H - B000H = 2580H |

Assume, for example, that you want to locate the address x within the memory image loaded under DDT in a 20K 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

```
          H3400,D580
```

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 the BIOS located at (4A00H+b)-n which, when you use the H command, produces an actual address of 1F80H. The disassembly command would thus be

```
          L1F80
```

It is now necessary to patch in your CBOOT and CBIOS routines. The BOOT resides at location 0900H in the memory image. If the actual load address is "n", then to calculate the bias (m) use the command:

```
          H900,n                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 0080H, the command:

```
          H900,80
```

will reply

```
          0980 0880              Sum and difference in hex.
```

Therefore, the bias "m" would be 0880H. To read-in the BOOT, give the command:

```
          ICBOOT.HEX             Input file CBOOT.HEX
```

Then:

```
          Rm                     Read CBOOT with a bias of
                                 m (=900H-n)
```

You may now examine your CBOOT with:

```
         L900
```

We are now ready to replace the CBIOS. Examine the area at 1F80H where the original version of the CBIOS resides. Then type

```
         ICBIOS.HEX              Ready the "hex" file for loading
```

assume that your CBIOS is being integrated into a 20K CP/M system, and thus is origined at location 4A00H. In order to properly locate the CBIOS in the memory image under DDT, we must apply the negative bias n for a 20K system when loading the hex file. This is accomplished by typing

```
          RD580                  Read the file with bias D580H
```

Upon completion of the read, re-examine the area where the CBIOS has been loaded (use an "L1F80" command), to ensure that is was loaded properly. When you are satisfied that the change has been made, return from DDT using a control-C or "G0" command.

Now use SYSGEN to replace 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                 Start the SYSGEN program
SYSGEN VERSION 2.Ø     Sign-on message from SYSGEN
SOURCE DRIVE NAME (OR RETURN TO SKIP)
                       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
                       Place a scratch diskette in
                       drive B, then type return.
FUNCTION COMPLETE
DESTINATION DRIVE NAME (OR RETURN TO REBOOT)
```

Place the scratch diskette in your drive A, and then perform a coldstart to bring up the new CP/M system you have configured.

Test the new CP/M system, and place the Digital Research copyright notice on the diskette, as specified in your Licensing Agreement:

Copyright (c), 1979
Digital Research

## 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 WRITESEC subroutines must be inserted by the user to read and write the specific sectors.

```
;   GETSYS PROGRAM - READ TRACKS Ø AND 1 TO MEMORY AT 338ØH
;   REGISTER                  USE
;      A              (SCRATCH REGISTER)
;      B              TRACK COUNT (Ø, 1)
;      C              SECTOR COUNT (1,2,...,26)
;      DE             (SCRATCH REGISTER PAIR)
;      HL             LOAD ADDRESS
;      SP             SET TO STACK ADDRESS
;
START: LXI    SP,338ØH     ;SET STACK POINTER TO SCRATCH AREA
       LXI    H, 338ØH     ;SET BASE LOAD ADDRESS
       MVI    B, Ø         ;START WITH TRACK Ø
RDTRK:                     ;READ NEXT TRACK (INITIALLY Ø)
       MVI    C,1          ;READ STARTING WITH SECTOR 1
RDSEC:                     ;READ NEXT SECTOR
       CALL   READSEC      ;USER-SUPPLIED SUBROUTINE
       LXI    D,128        ;MOVE LOAD ADDRESS TO NEXT 1/2 PAGE
       DAD    D            ;HL = HL + 128
       INR    C            ;SECTOR = SECTOR + 1
       MOV    A,C          ;CHECK FOR END OF TRACK
       CPI    27
       JC     RDSEC        ;CARRY GENERATED IF SECTOR < 27
;
;   ARRIVE HERE AT END OF TRACK, MOVE TO NEXT TRACK
       INR    B
       MOV    A,B          ;TEST FOR LAST TRACK
       CPI    2
       JC     RDTRK        ;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

       label START if an error occurs
       ..........................................
       POP    H            ;RECOVER HL
       POP    B            ;RECOVER B AND C REGISTERS
       RET                 ;BACK TO MAIN PROGRAM

       END    START
```

Note that this program is assembled and listed in Appendix C for reference purposes, with an assumed origin of 100H. 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 PUTSYS program can be constructed from GETSYS by changing only a few operations in the GETSYS program given above, as shown in Appendix D. The register pair HL become 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 sector given by register C. It is often useful to combine GETSYS and PUTSYS into a single program during the test and development phase, as shown in the Appendix.

10

(All Information Contained Herein is Proprietary to Digital Research.)

## 5. DISKETTE ORGANIZATION

The sector allocation for the standard distribution version of CP/M is given here for reference purposes. The first sector (see table on the following page) contains an optional software boot section. Disk controllers are often set up to bring track Ø, sector 1 into memory at a specific location (often location ØØØØH). The program in this sector, called BOOT, has the responsibility of bringing the remaining sectors into memory starting at location 34ØØH+b. If your controller does not have a built-in sector load, you can ignore the program in track Ø, sector 1, and begin the load from track Ø sector 2 to location 34ØØH+b.

As an example, the Intel MDS-8ØØ hardware cold start loader brings track Ø, sector 1 into absolute address 3ØØØH. Upon loading this sector, control transfers to location 3ØØØH, where the bootstrap operation commences by loading the remainder of tracks Ø, and all of track 1 into memory, starting at 34ØØH+b. The user should note that this bootstrap loader is of little use in a non-MDS environment, although it is useful to examine it since some of the boot actions will have to be duplicated in your cold start loader.

| Track# | Sector# | Page# | Memory Address | CP/M Module name |
|---|---|---|---|---|
| ØØ | Ø1 | | (boot address) | Cold Start Loader |
| ØØ | Ø2 | ØØ | 34ØØH+b | CCP |
| " | Ø3 | " | 348ØH+b | " |
| " | Ø4 | Ø1 | 35ØØH+b | " |
| " | Ø5 | " | 358ØH+b | " |
| " | Ø6 | Ø2 | 36ØØH+b | " |
| " | Ø7 | " | 368ØH+b | " |
| " | Ø8 | Ø3 | 37ØØH+b | " |
| " | Ø9 | " | 378ØH+b | " |
| " | 1Ø | Ø4 | 38ØØH+b | " |
| " | 11 | " | 388ØH+b | " |
| " | 12 | Ø5 | 39ØØH+b | " |
| " | 13 | " | 398ØH+b | " |
| " | 14 | Ø6 | 3AØØH+b | " |
| " | 15 | " | 3A8ØH+b | " |
| " | 16 | Ø7 | 3BØØH+b | " |
| ØØ | 17 | " | 3B8ØH+b | CCP |
| ØØ | 18 | Ø8 | 3CØØH+b | BDOS |
| " | 19 | " | 3C8ØH+b | " |
| " | 2Ø | Ø9 | 3DØØH+b | " |
| " | 21 | " | 3D8ØH+b | " |
| " | 22 | 1Ø | 3EØØH+b | " |
| " | 23 | " | 3E8ØH+b | " |
| " | 24 | 11 | 3FØØH+b | " |
| " | 25 | " | 3F8ØH+b | " |
| " | 26 | 12 | 4ØØØH+b | " |
| Ø1 | Ø1 | " | 4Ø8ØH+b | " |
| " | Ø2 | 13 | 41ØØH+b | " |
| " | Ø3 | " | 418ØH+b | " |
| " | Ø4 | 14 | 42ØØH+b | " |
| " | Ø5 | " | 428ØH+b | " |
| " | Ø6 | 15 | 43ØØH+b | " |
| " | Ø7 | " | 438ØH+b | " |
| " | Ø8 | 16 | 44ØØH+b | " |
| " | Ø9 | " | 448ØH+b | " |
| " | 1Ø | 17 | 45ØØH+b | " |
| " | 11 | " | 458ØH+b | " |
| " | 12 | 18 | 46ØØH+b | " |
| " | 13 | " | 468ØH+b | " |
| " | 14 | 19 | 47ØØH+b | " |
| " | 15 | " | 478ØH+b | " |
| " | 16 | 2Ø | 48ØØH+b | " |
| " | 17 | " | 488ØH+b | " |
| " | 18 | 21 | 49ØØH+b | " |
| Ø1 | 19 | " | 498ØH+b | BDOS |
| Ø1 | 2Ø | 22 | 4AØØH+b | BIOS |
| " | 21 | " | 4A8ØH+b | " |
| " | 23 | 23 | 4BØØH+b | " |
| " | 24 | " | 4B8ØH+b | " |
| " | 25 | 24 | 4CØØH+b | " |
| Ø1 | 26 | " | 4C8ØH+b | BIOS |
| Ø2-76 | Ø1-26 | | | (directory and data) |

## 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" located at 4AØØH+b, as shown below (see Appendices B and C, as well). The jump vector is a sequence of 17 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.

The jump vector at 4AØØH+b takes the form shown below, where the individual jump addresses are given to the left:

```
4AØØH+b    JMP BOOT       ; ARRIVE HERE FROM COLD START LOAD
4AØ3H+b    JMP WBOOT      ; ARRIVE HERE FOR WARM START
4AØ6H+b    JMP CONST      ; CHECK FOR CONSOLE CHAR READY
4AØ9H+b    JMP CONIN      ; READ CONSOLE CHARACTER IN
4AØCH+b    JMP CONOUT     ; WRITE CONSOLE CHARACTER OUT
4AØFH+b    JMP LIST       ; WRITE LISTING CHARACTER OUT
4A12H+b    JMP PUNCH      ; WRITE CHARACTER TO PUNCH DEVICE
4A15H+b    JMP READER     ; READ READER DEVICE
4A18H+b    JMP HOME       ; MOVE TO TRACK ØØ ON SELECTED DISK
4A1BH+b    JMP SELDSK     ; SELECT DISK DRIVE
4A1EH+b    JMP SETTRK     ; SET TRACK NUMBER
4A21H+b    JMP SETSEC     ; SET SECTOR NUMBER
4A24H+b    JMP SETDMA     ; SET DMA ADDRESS
4A27H+b    JMP READ       ; READ SELECTED SECTOR
4A2AH+b    JMP WRITE      ; WRITE SELECTED SECTOR
4A2DH+b    JMP LISTST     ; RETURN LIST STATUS
4A3ØH+b    JMP SECTRAN    ; SECTOR TRANSLATE SUBROUTINE
```

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: the system (re)initialization which results from calls on BOOT and WBOOT, simple character I/O performed by calls on CONST, CONIN, CONOUT, LIST, PUNCH, READER, and LISTST, and diskette I/O performed by calls on HOME, SELDSK, SETTRK, SETSEC, SETDMA, READ, WRITE, and SECTRAN.

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 for an input device is given by an ASCII control-z (1AH). 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 may be used by PIP, but not the BDOS). Further, the LISTST entry is used currently only by DESPOOL, and 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 may 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 just simply return, and the READER routine can return with a 1AH (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:

```
                    most significant        least significant
                    -----------------------------------------
IOBYTE AT  0003H    | LIST     | PUNCH    | READER   | CONSOLE |
                    -----------------------------------------
                     bits 6,7  bits 4,5   bits 2,3   bits 0,1
```

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

10

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 (RDR:)
    2  - user defined reader # 1 (UR1:)
    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 (PUN:)
    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 which 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 retries (10 is standard) 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 00 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 responsibilites 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, including 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 3400H+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 CBIOS, if you have completed your patch). System parameters must be initialized as shown below:

location 0,1,2   set to JMP WBOOT for warm starts (0000H: JMP 4A03H+b)

location 3   set initial value of IOBYTE, if implemented in your CBIOS

location 5,6,7   set to JMP BDOS, which is the primary entry point to CP/M for transient programs. (0005H: JMP 3C06H+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 3400H+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 and 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 parity bit (high order bit) to zero. 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 device. 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 characters 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 device 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.

SELDSK

Select the disk drive given by register C for further operations, where register C contains 0 for drive A, 1 for drive B, and so-forth up to 15 for drive P (the standard CP/M distribution version supports four drives). On each disk select, SELDSK must return in HL the base address of a 16-byte area, called the Disk Parameter Header, described in the Section 10. For standard floppy disk drives, the contents of the header and associated tables does not change, and thus the program segment included in the sample CBIOS performs this operation automatically. If there is an attempt to select a non-existent drive, SELDSK returns HL=0000H as an error indicator. Although SELDSK must return the header address on each call, it is advisable to postpone the actual physical disk select operation until an I/O function (seek, read or write) is actually performed, since disk selects often occur without utimately performing 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 BC contains the track number for subseauent 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 BC can take on values in the range Ø-76 corresponding to valid track numbers for standard floppy disk drives, and Ø-65535 for non-standard disk subsystems.

SETSEC

Register BC contains the sector number (1 through 26) for subsequent 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

Register BC contains the DMA (disk memory access) address for subsequent read or write operations. For example, if B = ØØH and C = 8ØH when SETDMA is called, then all subsequent read operations read their data into 8ØH through ØFFH, and all subsequent write operations get their data from 8ØH through ØFFH, until the next call to SETDMA occurs. The initial DMA address is assumed to be 8ØH. 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 CBIOS 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:

| | |
|---|---|
| Ø | 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 Ø then CP/M assumes that the disk operation completed properly. If an error occurs, however, the CBIOS should attempt at least 1Ø retries 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 ctl-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

10

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.

LISTST Return the ready status of the list device. Used by the DESPOOL program to improve console response during its operation. The value ØØ is returned in A if the list device is not ready to accept a character, and ØFFH if a character can be sent to the printer. Note that a ØØ value always suffices.

SECTRAN Performs sector logical to physical sector translation in order to improve the overall response of CP/M. Standard CP/M systems are shipped with a "skew factor" of 6, where six physical sectors are skipped between each logical read operation. This skew factor allows enough time between sectors for most programs to load their buffers without missing the next sector. In particular computer systems which use fast processors, memory, and disk subsystems, the skew factor may be changed to improve overall response. Note, however, that you should maintain a single density IBM compatible version of CP/M for information transfer into and out of your computer system, using a skew factor of 6. In general, SECTRAN receives a logical sector number in BC, and a translate table address in DE. The sector number is used as an index into the translate table, with the resulting physical sector number in HL. For standard systems, the tables and indexing code is provided in the CBIOS and need not be changed.

## 7. A SAMPLE BIOS

The program shown in Appendix C 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 subroutines. 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 IOBYTE function can be implemented.

## 8. A SAMPLE COLD START LOADER

The program shown in Appendix D 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, between locations 00H and 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.

| Locations<br>from - to | Contents |
|---|---|
| 0000H - 0002H | Contains a jump instruction to the warm start entry point at location 4A03H+b. This allows a simple programmed restart (JMP 0000H) or manual restart from the front panel. |
| 0003H - 0003H | Contains the Intel standard IOBYTE, which is optionally included in the user's CBIOS, as described in Section 6. |
| 0004H - 0004H | Current default drive number (0=A,...,15=P). |
| 0005H - 0007H | 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 "CP/M Interface Guide," 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. |
| 0008H - 0027H | (interrupt locations 1 through 5 not used) |
| 0030H - 0037H | (interrupt location 6, not currently used - reserved) |
| 0038H - 003AH | Restart 7 - Contains a jump instruction into the DDT or SID program when running in debug mode for programmed breakpoints, but is not otherwise used by CP/M. |
| 003BH - 003FH | (not currently used - reserved) |
| 0040H - 004FH | 16 byte area reserved for scratch by CBIOS, but is not used for any purpose in the distribution version of CP/M |
| 0050H - 005BH | (not currently used - reserved) |
| 005CH - 007CH | default file control block produced for a transient program by the Console Command Processor. |
| 007DH - 007FH | Optional default random record position |

0080H - 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 set-up for normal operation under the CP/M system, but can be overwritten by a transient program if the BDOS 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.

## 1Ø. DISK PARAMETER TABLES.

Tables are included in the BIOS which describe the particular characteristics of the disk subsystem used with CP/M. These tables can be either hand-coded, as shown in the sample CBIOS in Appendix C, or automatically generated using the DISKDEF macro library, as shown in Appendix B. The purpose here is to describe the elements of these tables.

In general, each disk drive has an associated (16-byte) disk parameter header which both contains information about the disk drive and provides a scratchpad area for certain BDOS operations. The format of the disk parameter header for each drive is shown below

```
                        Disk    Parameter   Header
     -------------------------------------------------------------
     |  XLT  | ØØØØ | ØØØØ | ØØØØ |DIRBUF|  DPB  |  CSV  |  ALV  |
     -------------------------------------------------------------
        16b    16b    16b    16b    16b    16b     16b     16b
```

where each element is a word (16-bit) value. The meaning of each Disk Parameter Header (DPH) element is

XLT Address of the logical to physical translation vector, if used for this particular drive, or the value ØØØØH if no sector translation takes place (i.e, the physical and logical sector numbers are the same). Disk drives with identical sector skew factors share the same translate tables.

ØØØØ Scratchpad values for use within the BDOS (initial value is unimportant).

DIRBUF Address of a 128 byte scratchpad area for directory operations within BDOS. All DPH's address the same scratchpad area.

DPB Address of a disk parameter block for this drive. Drives with identical disk characteristics address the same disk parameter block.

CSV Address of a scratchpad area used for software check for changed disks. This address is different for each DPH.

ALV Address of a scratchpad area used by the BDOS to keep disk storage allocation information. This address is different for each DPH.

Given n disk drives, the DPH's are arranged in a table whose first row of 16 bytes corresponds to drive Ø, with the last row corresponding to drive n-1. The table thus appears as

```
DPBASE:
        --------------------------------------------------------
    00  |XLT 00| 0000 | 0000 | 0000 |DIRBUF|DBP 00|CSV 00|ALV 00|
        --------------------------------------------------------
    01  |XLT 01| 0000 | 0000 | 0000 |DIRBUF|DBP 01|CSV 01|ALV 01|
        --------------------------------------------------------
                         (and so-forth through)
        --------------------------------------------------------
    n-1|XLTn-1| 0000 | 0000 | 0000 |DIRBUF|DBPn-1|CSVn-1|ALVn-1|
        --------------------------------------------------------
```

where the label DPBASE defines the base address of the DPH table.

A responsibility of the SELDSK subroutine is to return the base address of the DPH for the selected drive. The following sequence of operations returns the table address, with a 0000H returned if the selected drive does not exist.

```
NDISKS   EQU     4  ;NUMBER OF DISK DRIVES
......
SELDSK:
         ;SELECT DISK GIVEN BY BC
         LXI    H,0000H  ;ERROR CODE
         MOV    A,C      ;DRIVE OK?
         CPI    NDISKS   ;CY IF SO
         RNC             ;RET IF ERROR
         ;NO ERROR, CONTINUE
         MOV    L,C      ;LOW(DISK)
         MOV    H,B      ;HIGH(DISK)
         DAD    H        ;*2
         DAD    H        ;*4
         DAD    H        ;*8
         DAD    H        ;*16
         LXI    D,DPBASE ;FIRST DPH
         DAD    D        ;DPH(DISK)
         RET
```

The translation vectors (XLT 00 through XLTn-1) are located elsewhere in the BIOS, and simply correspond one-for-one with the logical sector numbers zero through the sector count-1. The Disk Parameter Block (DPB) for each drive is more complex. A particular DPB, which is addressed by one or more DPH's, takes the general form

```
-------------------------------------------------------------
|  SPT  |BSH|BLM|EXM|  DSM  |  DRM  |AL0|AL1|  CKS  |  OFF  |
-------------------------------------------------------------
   16b   8b  8b  8b    16b     16b   8b  8b    16b     16b
```

where each is a byte or word value, as shown by the "8b" or "16b" indicator below the field.

SPT is the total number of sectors per track

BSH is the data allocation block shift factor, determined by the data block allocation size.

(All Information Contained Herein is Proprietary to Digital Research.)

EXM is the extent mask, determined by the data block allocation size and the number of disk blocks.

DSM determines the total storage capacity of the disk drive

DRM determines the total number of directory entries which can be stored on this drive ALØ,AL1 determine reserved directory blocks.

CKS is the size of the directory check vector

OFF is the number of reserved tracks at the beginning of the (logical) disk.

The values of BSH and BLM determine (implicitly) the data allocation size BLS, which is not an entry in the disk parameter block. Given that the designer has selected a value for BLS, the values of BSH and BLM are shown in the table below

| BLS | BSH | BLM |
|---|---|---|
| 1,Ø24 | 3 | 7 |
| 2,Ø48 | 4 | 15 |
| 4,Ø96 | 5 | 31 |
| 8,192 | 6 | 63 |
| 16,384 | 7 | 127 |

where all values are in decimal. The value of EXM depends upon both the BLS and whether the DSM value is less than 256 or greater than 255, as shown in the following table

| BLS | DSM < 256 | DSM > 255 |
|---|---|---|
| 1,Ø24 | Ø | N/A |
| 2,Ø48 | 1 | Ø |
| 4,Ø96 | 3 | 1 |
| 8,192 | 7 | 3 |
| 16,384 | 15 | 7 |

The value of DSM is the maximum data block number supported by this particular drive, measured in BLS units. The product BLS times (DSM+1) is the total number of bytes held by the drive and, of course, must be within the capacity of the physical disk, not counting the reserved operating system tracks.

The DRM entry is the one less than the total number of directory entries, which can take on a 16-bit value. The values of ALØ and AL1, however, are determined by DRM. The two values ALØ and AL1 can together be considered a string of 16-bits, as shown below.

```
-------------------------------------------------
|          AL0          |          AL1          |
-------------------------------------------------
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
-------------------------------------------------
 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
```

where position 00 corresponds to the high order bit of the byte labelled AL0, and 15 corresponds to the low order bit of the byte labelled AL1. Each bit position reserves a data block for number of directory entries, thus allowing a total of 16 data blocks to be assigned for directory entries (bits are assigned starting at 00 and filled to the right until position 15). Each directory entry occupies 32 bytes, resulting in the following table

| BLS | Directory Entries |
|---|---|
| 1,024 | 32 times # bits |
| 2,048 | 64 times # bits |
| 4,096 | 128 times # bits |
| 8,192 | 256 times # bits |
| 16,384 | 512 times # bits |

Thus, if DRM = 127 (128 directory entries), and BLS = 1024, then there are 32 directory entries per block, requiring 4 reserved blocks. In this case, the 4 high order bits of AL0 are set, resulting in the values AL0 = 0F0H and AL1 = 00H.

The CKS value is determined as follows: if the disk drive media is removable, then CKS = (DRM+1)/4, where DRM is the last directory entry number. If the media is fixed, then set CKS = 0 (no directory records are checked in this case).

Finally, the OFF field determines the number of tracks which are skipped at the beginning of the physical disk. This value is automatically added whenever SETTRK is called, and can be used as a mechanism for skipping reserved operating system tracks, or for partitioning a large disk into smaller segmented sections.

To complete the discussion of the DPB, recall that several DPH's can address the same DPB if their drive characteristics are identical. Further, the DPB can be dynamically changed when a new drive is addressed by simply changing the pointer in the DPH since the BDOS copies the DPB values to a local area whenever the SELDSK function is invoked.

Returning back to the DPH for a particular drive, note that the two address values CSV and ALV remain. Both addresses reference an area of uninitialized memory following the BIOS. The areas must be unique for each drive, and the size of each area is determined by the values in the DPB.

The size of the area addressed by CSV is CKS bytes, which is sufficient to hold the directory check information for this particular drive. If CKS = (DRM+1)/4, then you must reserve (DRM+1)/4 bytes for directory check use. If CKS = 0, then no storage is reserved.

The size of the area addressed by ALV is determined by the maximum number of data blocks allowed for this particular disk, and is computed as (DSM/8)+1.

The CBIOS shown in Appendix C demonstrates an instance of these tables for standard 8" single density drives. It may be useful to examine this program, and compare the tabular values with the definitions given above.

## 11. THE DISKDEF MACRO LIBRARY.

A macro library is shown in Appendix F, called DISKDEF, which greatly simplifies the table construction process. You must have access to the MAC macro assembler, of course, to use the DISKDEF facility, while the macro library is included with all CP/M 2.Ø distribution disks.

A BIOS disk definition consists of the following sequence of macro statements:

```
MACLIB   DISKDEF
......
DISKS    n
DISKDEF  Ø,...
DISKDEF  1,...
......
DISKDEF  n-1
......
ENDEF
```

where the MACLIB statement loads the DISKDEF.LIB file (on the same disk as your BIOS) into MAC's internal tables. The DISKS macro call follows, which specifies the number of drives to be configured with your system, where n is an integer in the range 1 to 16. A series of DISKDEF macro calls then follow which define the characteristics of each logical disk, Ø through n-1 (corresponding to logical drives A through P). Note that the DISKS and DISKDEF macros generate the in-line fixed data tables described in the previous section, and thus must be placed in a non-executable portion of your BIOS, typically directly following the BIOS jump vector.

The remaining portion of your BIOS is defined following the DISKDEF macros, with the ENDEF macro call immediately preceding the END statement. The ENDEF (End of Diskdef) macro generates the necessary uninitialized RAM areas which are located in memory above your BIOS.

The form of the DISKDEF macro call is

```
    DISKDEF  dn,fsc,lsc,[skf],bls,dks,dir,cks,ofs,[Ø]
```

where

```
dn     is the logical disk number, Ø to n-1
fsc    is the first physical sector number (Ø or 1)
lsc    is the last sector number
skf    is the optional sector skew factor
bls    is the data allocation block size
dir    is the number of directory entries
cks    is the number of "checked" directory entries
ofs    is the track offset to logical track ØØ
[Ø]    is an optional 1.4 compatibility flag
```

The value "dn" is the drive number being defined with this DISKDEF macro invocation. The "fsc" parameter accounts for differing sector numbering systems, and is usually 0 or 1. The "lsc" is the last numbered sector on a track. When present, the "skf" parameter defines the sector skew factor which is used to create a sector translation table according to the skew. If the number of sectors is less than 256, a single-byte table is created, otherwise each translation table element occupies two bytes. No translation table is created if the skf parameter is omitted (or equal to 0). The "bls" parameter specifies the number of bytes allocated to each data block, and takes on the values 1024, 2048, 4096, 8192, or 16384. Generally, performance increases with larger data block sizes since there are fewer directory references and logically connected data records are physically close on the disk. Further, each directory entry addresses more data and the BIOS-resident ram space is reduced. The "dks" specifies the total disk size in "bls" units. That is, if the bls = 2048 and dks = 1000, then the total disk capacity is 2,048,000 bytes. If dks is greater than 255, then the block size parameter bls must be greater than 1024. The value of "dir" is the total number of directory entries which may exceed 255, if desired. The "cks" parameter determines the number of directory items to check on each directory scan, and is used internally to detect changed disks during system operation, where an intervening cold or warm start has not occurred (when this situation is detected, CP/M automatically marks the disk read/only so that data is not subsequently destroyed). As stated in the previous section, the value of cks = dir when the media is easily changed, as is the case with a floppy disk subsystem. If the disk is permanently mounted, then the value of cks is typically 0, since the probability of changing disks without a restart is quite low. The "ofs" value determines the number of tracks to skip when this particular drive is addressed, which can be used to reserve additional operating system space or to simulate several logical drives on a single large capacity physical drive. Finally, the [0] parameter is included when file compatibility is required with versions of 1.4 which have been modified for higher density disks. This parameter ensures that only 16K is allocated for each directory record, as was the case for previous versions. Normally, this parameter is not included.

For convenience and economy of table space, the special form

```
                    DISKDEF    i,j
```

gives disk i the same characteristics as a previously defined drive j. A standard four-drive single density system, which is compatible with version 1.4, is defined using the following macro invocations:

```
        DISKS       4
        DISKDEF     0,1,26,6,1024,243,64,64,2
        DISKDEF     1,0
        DISKDEF     2,0
        DISKDEF     3,0
        ....
        ENDEF
```

with all disks having the same parameter values of 26 sectors per track (numbered 1 through 26), with 6 sectors skipped between each access, 1024 bytes per data block, 243 data blocks for a total of 243k byte disk capacity, 64 checked directory entries, and two operating system tracks.

The DISKS macro generates n Disk Parameter Headers (DPH's), starting at the DPH table address DPBASE generated by the macro. Each disk header block contains sixteen bytes, as described above, and correspond one-for-one to each of the defined drives. In the four drive standard system, for example, the DISKS macro generates a table of the form:

```
DPBASE  EQU   $
DPE0:   DW    XLT0,0000H,0000H,0000H,DIRBUF,DPB0,CSV0,ALV0
DPE1:   DW    XLT0,0000H,0000H,0000H,DIRBUF,DPB0,CSV1,ALV1
DPE2:   DW    XLT0,0000H,0000H,0000H,DIRBUF,DPB0,CSV2,ALV2
DPE3:   DW    XLT0,0000H,0000H,0000H,DIRBUF,DPB0,CSV3,ALV3
```

where the DPH labels are included for reference purposes to show the beginning table addresses for each drive 0 through 3. The values contained within the disk parameter header are described in detail in the previous section. The check and allocation vector addresses are generated by the ENDEF macro in the ram area following the BIOS code and tables.

Note that if the "skf" (skew factor) parameter is omitted (or equal to 0), the translation table is omitted, and a 0000H value is inserted in the XLT position of the disk parameter header for the disk. In a subsequent call to perform the logical to physical translation, SECTRAN receives a translation table address of DE = 0000H, and simply returns the original logical sector from BC in the HL register pair. A translate table is constructed when the skf parameter is present, and the (non-zero) table address is placed into the corresponding DPH's. The table shown below, for example, is constructed when the standard skew factor skf = 6 is specified in the DISKDEF macro call:

```
XLT0:  DB     1,7,13,19,25,5,11,17,23,3,9,15,21
       DB     2,8,14,20,26,6,12,18,24,4,10,16,22
```

Following the ENDEF macro call, a number of uninitialized data areas are defined. These data areas need not be a part of the BIOS which is loaded upon cold start, but must be available between the BIOS and the end of memory. The size of the uninitialized RAM area is determined by EQU statements generated by the ENDEF macro. For a standard four-drive system, the ENDEF macro might produce

```
4C72 =          BEGDAT EQU $
                (data areas)
4DB0 =          ENDDAT EQU $
013C =          DATSIZ EQU $-BEGDAT
```

which indicates that uninitialized RAM begins at location 4C72H, ends at 4DB0H-1, and occupies 013CH bytes. You must ensure that these addresses are free for use after the system is loaded.

After modification, you can use the STAT program to check your drive characteristics, since STAT uses the disk parameter block to decode the drive information. The STAT command form

```
STAT d:DSK:
```

decodes the disk parameter block for drive d (d=A,...,P) and displays the values shown below:

```
r: 128 Byte Record Capacity
k: Kilobyte Drive  Capacity
d: 32  Byte Directory Entries
c: Checked  Directory Entries
e: Records/ Extent
b: Records/ Block
s: Sectors/ Track
t: Reserved Tracks
```

Three examples of DISKDEF macro invocations are shown below with corresponding STAT parameter values (the last produces a full 8-megabyte system).

```
          DISKDEF 0,1,58,,2048,256,128,128,2
r=4096, k=512, d=128, c=128, e=256, b=16, s=58, t=2

          DISKDEF 0,1,58,,2048,1024,300,0,2
r=16384, k=2048, d=300, c=0, e=128, b=16, s=58, t=2

          DISKDEF 0,1,58,,16384,512,128,128,2
r=65536, k=8192, d=128, c=128, e=1024, b=128, s=58, t=2
```

## 12. SECTOR BLOCKING AND DEBLOCKING.

Upon each call to the BIOS WRITE entry point, the CP/M BDOS includes information which allows effective sector blocking and deblocking where the host disk subsystem has a sector size which is a multiple of the basic 128-byte unit. The purpose here is to present a general-purpose algorithm which can be included within your BIOS which uses the BDOS information to perform the operations automatically.

Upon each call to WRITE, the BDOS provides the following information in register C:

```
Ø   =   normal sector write
1   =   write to directory sector
2   =   write to the first sector
        of a new data block
```

Condition Ø occurs whenever the next write operation is into a previously written area, such as a random mode record update, when the write is to other than the first sector of an unallocated block, or when the write is not into the directory area. Condition 1 occurs when a write into the directory area is performed. Condition 2 occurs when the first record (only) of a newly allocated data block is written. In most cases, application programs read or write multiple 128 byte sectors in sequence, and thus there is little overhead involved in either operation when blocking and deblocking records since pre-read operations can be avoided when writing records.

Appendix G lists the blocking and deblocking algorithms in skeletal form (this file is included on your CP/M disk). Generally, the algorithms map all CP/M sector read operations onto the host disk through an intermediate buffer which is the size of the host disk sector. Throughout the program, values and variables which relate to the CP/M sector involved in a seek operation are prefixed by "sek," while those related to the host disk system are prefixed by "hst." The equate statements beginning on line 29 of Appendix G define the mapping between CP/M and the host system, and must be changed if other than the sample host system is involved.

The entry points BOOT and WBOOT must contain the initialization code starting on line 57, while the SELDSK entry point must be augmented by the code starting on line 65. Note that although the SELDSK entry point computes and returns the Disk Parameter Header address, it does not physically selected the host disk at this point (it is selected later at READHST or WRITEHST). Further, SETTRK, SETTRK, and SETDMA simply store the values, but do not take any other action at this point. SECTRAN performs a trivial trivial function of returning the physical sector number.

The principal entry points are READ and WRITE, starting on lines 11Ø and 125, respectively. These subroutines take the place of your previous READ and WRITE operations.

The actual physical read or write takes place at either WRITEHST or READHST, where all values have been prepared: hstdsk is the host disk number, hsttrk is the host track number, and hstsec is the host sector number (which may require translation to a physical sector number). You must insert code at this point which performs the full host sector read or write into, or out of, the buffer at hstbuf of length hstsiz. All other mapping functions are performed by the algorithms.

This particular algorithm was tested using an 80 megabyte hard disk unit which was originally configured for 128 byte sectors, producing approximately 35 megabytes of formatted storage. When configured for 512 byte host sectors, usable storage increased to 57 megabytes, with a corresponding 400% improvement in overall response. In this situation, there is no apparent overhead involved in deblocking sectors, with the advantage that user programs still maintain the (less memory consuming) 128-byte sectors. This is primarily due, of course, to the information provided by the BDOS which eliminates the necessity for pre-read operations to take place.

10

## APPENDIX A: THE MDS COLD START LOADER

```
            ;       MDS-800 Cold Start Loader for CP/M 2.0
            ;
            ;       Version 2.0 August, 1979
            ;
0000 =      false   equ     0
ffff =      true    equ     not false
0000 =      testing equ     false
            ;
                    if      testing
            bias    equ     03400h
                    endif
                    if      not testing
0000 =      bias    equ     0000h
                    endif
0000 =      cpmb    equ     bias            ;base of dos load
0806 =      bdos    equ     806h+bias       ;entry to dos for calls
1880 =      bdose   equ     1880h+bias      ;end of dos load
1600 =      boot    equ     1600h+bias      ;cold start entry point
1603 =      rboot   equ     boot+3          ;warm start entry point
            ;
3000                org     3000h   ;loaded here by hardware
            ;
1880 =      bdosl   equ     bdose-cpmb
0002 =      ntrks   equ     2               ;tracks to read
0031 =      bdoss   equ     bdosl/128       ;# sectors in bdos
0019 =      bdos0   equ     25              ;# on track 0
0018 =      bdosl   equ     bdoss-bdos0     ;# on track l
            ;
f800 =      mon80   equ     0f800h  ;intel monitor base
ff0f =      rmon80  equ     0ff0fh  ;restart location for mon80
0078 =      base    equ     078h    ;'base' used by controller
0079 =      rtype   equ     base+l  ;result type
007b =      rbyte   equ     base+3  ;result byte
007f =      reset   equ     base+7  ;reset controller
            ;
0078 =      dstat   equ     base    ;disk status port
0079 =      ilow    equ     base+l  ;low iopb address
007a =      ihigh   equ     base+2  ;high iopb address
00ff =      bsw     equ     0ffh    ;boot switch
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:
3000 310001         lxi     sp,stack;in case of call to mon80
            ;       clear disk status
3003 db79           in      rtype
3005 db7b           in      rbyte
            ;       check if boot switch is off
            coldstart:
3007 dbff           in      bsw
3009 e602           ani     02h     ;switch on?
300b c20730         jnz     coldstart
```

```
            ;       clear the controller
3ØØe d37f           out     reset   ;logic cleared
            ;
            ;
3Ø1Ø Ø6Ø2           mvi     b,ntrks ;number of tracks to read
3Ø12 21423Ø         lxi     h,iopbØ
            ;
            start:
            ;
            ;       read first/next track into cpmb
3Ø15 7d             mov     a,l
3Ø16 d379           out     ilow
3Ø18 7c             mov     a,h
3Ø19 d37a           out     ihigh
3Ø1b db78   waitØ:  in      dstat
3Ø1d e6Ø4           ani     4
3Ø1f ca1b3Ø         jz      waitØ
            ;
            ;       check disk status
3Ø22 db79           in      rtype
3Ø24 e6Ø3           ani     11b
3Ø26 feØ2           cpi     2
            ;
                    if      testing
                    cnc     rmon8Ø  ;go to monitor if 11 or 1Ø
                    endif
                    if      not testing
3Ø28 d2ØØ3Ø         jnc     rstart  ;retry the load
                    endif
            ;
3Ø2b db7b           in      rbyte   ;i/o complete, check status
            ;       if not ready, then go to mon8Ø
3Ø2d 17             ral
3Ø2e dcØfff         cc      rmon8Ø  ;not ready bit set
3Ø31 1f             rar             ;restore
3Ø32 e61e           ani     1111Øb  ;overrun/addr err/seek/crc
            ;
                    if      testing
                    cnz     rmon8Ø  ;go to monitor
                    endif
                    if      not testing
3Ø34 c2ØØ3Ø         jnz     rstart  ;retry the load
                    endif
            ;
            ;
3Ø37 11Ø7ØØ         lxi     d,iopbl ;length of iopb
3Ø3a 19             dad     d       ;addressing next iopb
3Ø3b Ø5             dcr     b       ;count down tracks
3Ø3c c2153Ø         jnz     start
            ;
            ;
            ;       jmp boot, print message, set-up jmps
3Ø3f c3ØØ16         jmp     boot
            ;
            ;       parameter blocks
```

```
3Ø42 8Ø      iopbØ:  db      8Øh      ;iocw, no update
3Ø43 Ø4              db      readf    ;read function
3Ø44 19              db      bdosØ    ;# sectors to read trk Ø
3Ø45 ØØ              db      Ø        ;track Ø
3Ø46 Ø2              db      2        ;start with sector 2, trk Ø
3Ø47 ØØØØ            dw      cpmb     ;start at base of bdos
ØØØ7 =       iopbl   equ     $-iopbØ
             ;
3Ø49 8Ø      iopbl:  db      8Øh
3Ø4a Ø4              db      readf
3Ø4b 18              db      bdosl    ;sectors to read on track 1
3Ø4c Ø1              db      1        ;track 1
3Ø4d Ø1              db      1        ;sector 1
3Ø4e 8ØØc            dw      cpmb+bdosØ*128  ;base of second rd
3Ø5Ø                 end
```

APPENDIX B: THE MDS BASIC I/O SYSTEM (BIOS)

```
            ;       mds-8ØØ i/o drivers for cp/m 2.Ø
            ;       (four drive single density version)
            ;
            ;       version 2.Ø august, 1979
            ;
ØØ14 =      vers    equ     2Ø      ;version 2.Ø
            ;
            ;       copyright (c) 1979
            ;       digital research
            ;       box 579, pacific grove
            ;       california, 9395Ø
            ;
4aØØ                org     4aØØh   ;base of bios in 2Øk system
34ØØ =      cpmb    equ     34ØØh   ;base of cpm ccp
3cØ6 =      bdos    equ     3cØ6h   ;base of bdos in 2Øk system
16ØØ =      cpml    equ     $-cpmb  ;length (in bytes) of cpm system
ØØ2c =      nsects  equ     cpml/128;number of sectors to load
ØØØ2 =      offset  equ     2       ;number of disk tracks used by cp
ØØØ4 =      cdisk   equ     ØØØ4h   ;address of last logged disk
ØØ8Ø =      buff    equ     ØØ8Øh   ;default buffer address
ØØØa =      retry   equ     1Ø      ;max retries on disk i/o before e
            ;
            ;       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 = ØØ 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 ØØ
            ;
            ;       (the following calls set-up the io parameter bloc
            ;       mds, which is used to perform subsequent reads an
            ;       seldsk  select disk given by reg-c (Ø,1,2...)
            ;       settrk  set track address (Ø,...76) for sub r/w
            ;       setsec  set sector address (1,...,26)
            ;       setdma  set subsequent dma address (initially 8Øh
            ;
            ;       read/write assume previous calls to set i/o parms
            ;       read    read track/sector to preset dma address
            ;       write   write track/sector from preset dma addres
            ;
            ;       jump vector for indiviual routines
4aØØ c3b34a         jmp     boot
4aØ3 c3c34a wboote: jmp     wboot
4aØ6 c3614b         jmp     const
4aØ9 c3644b         jmp     conin
4aØc c36a4b         jmp     conout
```

```
4aØf c36d4b             jmp     list
4a12 c3724b             jmp     punch
4a15 c3754b             jmp     reader
4a18 c3784b             jmp     home
4a1b c37d4b             jmp     seldsk
4a1e c3a74b             jmp     settrk
4a21 c3ac4b             jmp     setsec
4a24 c3bb4b             jmp     setdma
4a27 c3c14b             jmp     read
4a2a c3ca4b             jmp     write
4a2d c37Ø4b             jmp     listst  ;list status
4a3Ø c3b14b             jmp     sectran
            ;
                        maclib  diskdef ;load the disk definition library
                        disks   4       ;four disks
4a33+=      dpbase      equ     $       ;base of disk parameter blocks
4a33+824aØØ dpeØ:       dw      xltØ,ØØØØh      ;translate table
4a37+ØØØØØØ             dw      ØØØØh,ØØØØh     ;scratch area
4a3b+6e4c73             dw      dirbuf,dpbØ     ;dir buff,parm block
4a3f+Ød4dee             dw      csvØ,alvØ       ;check, alloc vectors
4a43+824aØØ dpe1:       dw      xlt1,ØØØØh      ;translate table
4a47+ØØØØØØ             dw      ØØØØh,ØØØØh     ;scratch area
4a4b+6e4c73             dw      dirbuf,dpb1     ;dir buff,parm block
4a4f+3c4d1d             dw      csv1,alv1       ;check, alloc vectors
4a53+824aØØ dpe2:       dw      xlt2,ØØØØh      ;translate table
4a57+ØØØØØØ             dw      ØØØØh,ØØØØh     ;scratch area
4a5b+6e4c73             dw      dirbuf,dpb2     ;dir buff,parm block
4a5f+6b4d4c             dw      csv2,alv2       ;check, alloc vectors
4a63+824aØØ dpe3:       dw      xlt3,ØØØØh      ;translate table
4a67+ØØØØØØ             dw      ØØØØh,ØØØØh     ;scratch area
4a6b+6e4c73             dw      dirbuf,dpb3     ;dir buff,parm block
4a6f+9a4d7b             dw      csv3,alv3       ;check, alloc vectors
                        diskdef Ø,1,26,6,1Ø24,243,64,64,offset
4a73+=      dpbØ        equ     $               ;disk parm block
4a73+1aØØ               dw      26              ;sec per track
4a75+Ø3                 db      3               ;block shift
4a76+Ø7                 db      7               ;block mask
4a77+ØØ                 db      Ø               ;extnt mask
4a78+f2ØØ               dw      242             ;disk size-1
4a7a+3fØØ               dw      63              ;directory max
4a7c+cØ                 db      192             ;allocØ
4a7d+ØØ                 db      Ø               ;alloc1
4a7e+1ØØØ               dw      16              ;check size
4a8Ø+Ø2ØØ               dw      2               ;offset
4a82+=      xltØ        equ     $               ;translate table
4a82+Ø1                 db      1
4a83+Ø7                 db      7
4a84+Ød                 db      13
4a85+13                 db      19
4a86+19                 db      25
4a87+Ø5                 db      5
4a88+Øb                 db      11
4a89+11                 db      17
4a8a+17                 db      23
4a8b+Ø3                 db      3
```

```
4a8c+09                 db      9
4a8d+0f                 db      15
4a8e+15                 db      21
4a8f+02                 db      2
4a90+08                 db      8
4a91+0e                 db      14
4a92+14                 db      20
4a93+1a                 db      26
4a94+06                 db      6
4a95+0c                 db      12
4a96+12                 db      18
4a97+18                 db      24
4a98+04                 db      4
4a99+0a                 db      10
4a9a+10                 db      16
4a9b+16                 db      22
                        diskdef 1,0
4a73+=      dpb1        equ     dpb0    ;equivalent parameters
001f+=      als1        equ     als0    ;same allocation vector size
0010+=      css1        equ     css0    ;same checksum vector size
4a82+=      xlt1        equ     xlt0    ;same translate table
                        diskdef 2,0
4a73+=      dpb2        equ     dpb0    ;equivalent parameters
001f+=      als2        equ     als0    ;same allocation vector size
0010+=      css2        equ     css0    ;same checksum vector size
4a82+=      xlt2        equ     xlt0    ;same translate table
                        diskdef 3,0
4a73+=      dpb3        equ     dpb0    ;equivalent parameters
001f+=      als3        equ     als0    ;same allocation vector size
0010+=      css3        equ     css0    ;same checksum vector size
4a82+=      xlt3        equ     xlt0    ;same translate table
            ;           endef occurs at end of assembly
            ;
            ;           end of controller - independent code, the remaini
            ;           are tailored to the particular operating environm
            ;           be altered for any system which differs from the
            ;
            ;           the following code assumes the mds monitor exists
            ;           and uses the i/o subroutines within the monitor
            ;
            ;           we also assume the mds system has four disk drive
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
            ;
            ;           mds monitor equates
f800 =      mon80       equ     0f800h  ;mds monitor
ff0f =      rmon80      equ     0ff0fh  ;restart mon80 (boot error)
f803 =      ci          equ     0f803h  ;console character to reg-a
f806 =      ri          equ     0f806h  ;reader in to reg-a
f809 =      co          equ     0f809h  ;console char from c to console o
f80c =      po          equ     0f80ch  ;punch char from c to punch devic
f80f =      lo          equ     0f80fh  ;list from c to list device
f812 =      csts        equ     0f812h  ;console status 00/ff to register
```

```
            ;
            ;       disk ports and commands
ØØ78 =      base    equ     78h      ;base of disk command io ports
ØØ78 =      dstat   equ     base     ;disk status (input)
ØØ79 =      rtype   equ     base+l   ;result type (input)
ØØ7b =      rbyte   equ     base+3   ;result byte (input)
            ;
ØØ79 =      ilow    equ     base+l   ;iopb low address (output)
ØØ7a =      ihigh   equ     base+2   ;iopb high address (output)
            ;
ØØØ4 =      readf   equ     4h       ;read function
ØØØ6 =      writf   equ     6h       ;write function
ØØØ3 =      recal   equ     3h       ;recalibrate drive
ØØØ4 =      iordy   equ     4h       ;i/o finished mask
ØØØd =      cr      equ     Ødh      ;carriage return
ØØØa =      lf      equ     Øah      ;line feed
            ;
            signon: ;signon message: xxk cp/m vers y.y
4a9c Ød0aØa         db      cr,lf,lf
4a9f 323Ø           db      '2Ø'     ;sample memory size
4aal 6b2Ø43f        db      'k cp/m vers '
4aad 322e3Ø         db      vers/lØ+'Ø','.',vers mod lØ+'Ø'
4abØ ØdØaØØ         db      cr,lf,Ø
            ;
            boot:   ;print signon message and go to ccp
            ;       (note: mds boot initialized iobyte at ØØØ3h)
4ab3 31ØØØl         lxi     sp,buff+8Øh
4ab6 219c4a         lxi     h,signon
4ab9 cdd34b         call    prmsg    ;print message
4abc af             xra     a        ;clear accumulator
4abd 32Ø4ØØ         sta     cdisk    ;set initially to disk a
4acØ c3Øf4b         jmp     gocpm    ;go to cp/m
            ;
            ;
            wboot:; loader on track Ø, sector l, which will be skippe
            ;       read cp/m from disk - assuming there is a 128 byt
            ;       start.
            ;
4ac3 318ØØØ         lxi     sp,buff ;using dma - thus 8Ø thru ff ok f
            ;
4ac6 ØeØa           mvi     c,retry ;max retries
4ac8 c5             push    b
            wbootØ: ;enter here on error retries
4ac9 Ø1ØØ34         lxi     b,cpmb   ;set dma address to start of disk
4acc cdbb4b         call    setdma
4acf ØeØØ           mvi     c,Ø      ;boot from drive Ø
4adl cd7d4b         call    seldsk
4ad4 ØeØØ           mvi     c,Ø
4ad6 cda74b         call    settrk   ;start with track Ø
4ad9 ØeØ2           mvi     c,2      ;start reading sector 2
4adb cdac4b         call    setsec
            ;
            ;       read sectors, count nsects to zero
4ade cl             pop     b        ;lØ-error count
4adf Ø62c           mvi     b,nsects
```

```
            rdsec:  ;read next sector
4ae1 c5             push    b       ;save sector count
4ae2 cdc14b         call    read
4ae5 c2494b         jnz     booterr ;retry if errors occur
4ae8 2a6c4c         lhld    iod     ;increment dma address
4aeb 118ØØØ         lxi     d,128   ;sector size
4aee 19             dad     d       ;incremented dma address in hl
4aef 44             mov     b,h
4afØ 4d             mov     c,l     ;ready for call to set dma
4af1 cdbb4b         call    setdma
4af4 3a6b4c         lda     ios     ;sector number just read
4af7 fe1a           cpi     26      ;read last sector?
4af9 daØ54b         jc      rd1
            ;       must be sector 26, zero and go to next track
4afc 3a6a4c         lda     iot     ;get track to register a
4aff 3c             inr     a
4bØØ 4f             mov     c,a     ;ready for call
4bØ1 cda74b         call    settrk
4bØ4 af             xra     a       ;clear sector number
4bØ5 3c     rd1:    inr     a       ;to next sector
4bØ6 4f             mov     c,a     ;ready for call
4bØ7 cdac4b         call    setsec
4bØa c1             pop     b       ;recall sector count
4bØb Ø5             dcr     b       ;done?
4bØc c2e14a         jnz     rdsec
            ;
            ;       done with the load, reset default buffer address
            gocpm:  ;(enter here from cold start boot)
            ;       enable rstØ and rst7
4bØf f3             di
4b1Ø 3e12           mvi     a,12h   ;initialize command
4b12 d3fd           out     revrt
4b14 af             xra     a
4b15 d3fc           out     intc    ;cleared
4b17 3e7e           mvi     a,inte  ;rstØ and rst7 bits on
4b19 d3fc           out     intc
4b1b af             xra     a
4b1c d3f3           out     icon    ;interrupt control
            ;
            ;       set default buffer address to 8Øh
4b1e Ø18ØØØ         lxi     b,buff
4b21 cdbb4b         call    setdma
            ;
            ;       reset monitor entry points
4b24 3ec3           mvi     a,jmp
4b26 32ØØØØ         sta     Ø
4b29 21Ø34a         lxi     h,wboote
4b2c 22Ø1ØØ         shld    1       ;jmp wboot at location ØØ
4b2f 32Ø5ØØ         sta     5
4b32 21Ø63c         lxi     h,bdos
4b35 22Ø6ØØ         shld    6       ;jmp bdos at location 5
4b38 3238ØØ         sta     7*8     ;jmp to mon8Ø (may have been chan
4b3b 21ØØf8         lxi     h,mon8Ø
4b3e 2239ØØ         shld    7*8+1
            ;       leave iobyte set
```

10

```
            ;       previously selected disk was b, send parameter to
4b41 3aØ4ØØ         lda     cdisk   ;last logged disk number
4b44 4f             mov     c,a     ;send to ccp to log it in
4b45 fb             ei
4b46 c3ØØ34         jmp     cpmb
            ;
            ;       error condition occurred, print message and retry
            booterr:
4b49 cl             pop     b       ;recall counts
4b4a Ød             dcr     c
4b4b ca524b         jz      booterØ
            ;       try again
4b4e c5             push    b
4b4f c3c94a         jmp     wbootØ
            ;
            booterØ:
            ;       otherwise too many retries
4b52 215b4b         lxi     h,bootmsg
4b55 cdd34b         call    prmsg
4b58 c3Øfff         jmp     rmon8Ø  ;mds hardware monitor
            ;
            bootmsg:
4b5b 3f626f4        db      '?boot',Ø
            ;
            ;
            const:  ;console status to reg-a
            ;       (exactly the same as mds call)
4b61 c312f8         jmp     csts
            ;
            conin:  ;console character to reg-a
4b64 cdØ3f8         call    ci
4b67 e67f           ani     7fh     ;remove parity bit
4b69 c9             ret
            ;
            conout: ;console character from c to console out
4b6a c3Ø9f8         jmp     co
            ;
            list:   ;list device out
            ;       (exactly the same as mds call)
4b6d c3Øff8         jmp     lo
            ;
            listst:
                    ;return list status
4b7Ø af             xra     a
4b71 c9             ret             ;always not ready
            ;
            punch:  ;punch device out
            ;       (exactly the same as mds call)
4b72 c3Øcf8         jmp     po
            ;
            reader: ;reader character in to reg-a
            ;       (exactly the same as mds call)
4b75 c3Ø6f8         jmp     ri
            ;
            home:   ;move to home position
            ;       treat as track ØØ seek
4b78 ØeØØ           mvi     c,Ø
4b7a c3a74b         jmp     settrk
            ;
            seldsk: ;select disk given by register c
4b7d 21ØØØØ         lxi     h,ØØØØh ;return ØØØØ if error
4b8Ø 79             mov     a,c
4b81 feØ4           cpi     ndisks  ;too large?
4b83 dØ             rnc             ;leave hl = ØØØØ
            ;
4b84 e6Ø2           ani     1Øb     ;ØØ ØØ for drive Ø,1 and 1Ø 1Ø fo
4b86 32664c         sta     dbank   ;to select drive bank
4b89 79             mov     a,c     ;ØØ, Ø1, 1Ø, 11
4b8a e6Ø1           ani     1b      ;mds has Ø,1 at 78, 2,3 at 88
4b8c b7             ora     a       ;result ØØ?
4b8d ca924b         jz      setdrive
4b9Ø 3e3Ø           mvi     a,ØØ11ØØØØb     ;selects drive 1 in bank
            setdrive:
4b92 47             mov     b,a     ;save the function
4b93 21684c         lxi     h,iof   ;io function
4b96 7e             mov     a,m
4b97 e6cf           ani     11ØØ1111b       ;mask out disk number
4b99 bØ             ora     b       ;mask in new disk number
4b9a 77             mov     m,a     ;save it in iopb
4b9b 69             mov     l,c
4b9c 26ØØ           mvi     h,Ø     ;hl=disk number
4b9e 29             dad     h       ;*2
4b9f 29             dad     h       ;*4
4baØ 29             dad     h       ;*8
4ba1 29             dad     h       ;*16
4ba2 11334a         lxi     d,dpbase
4ba5 19             dad     d       ;hl=disk header table address
4ba6 c9             ret
            ;
            ;
            settrk: ;set track address given by c
4ba7 216a4c         lxi     h,iot
4baa 71             mov     m,c
4bab c9             ret
            ;
            setsec: ;set sector number given by c
4bac 216b4c         lxi     h,ios
4baf 71             mov     m,c
4bbØ c9             ret
            sectran:
                            ;translate sector bc using table at de
4bb1 Ø6ØØ           mvi     b,Ø     ;double precision sector number i
4bb3 eb             xchg            ;translate table address to hl
4bb4 Ø9             dad     b       ;translate(sector) address
4bb5 7e             mov     a,m     ;translated sector number to a
4bb6 326b4c         sta     ios
4bb9 6f             mov     l,a     ;return sector number in l
4bba c9             ret
            ;
            setdma: ;set dma address given by regs b,c
```

```
4bbb 69             mov     l,c
4bbc 60             mov     h,b
4bbd 226c4c         shld    iod
4bc0 c9             ret
            ;
            read:   ;read next disk record (assuming disk/trk/sec/dma
4bc1 0e04           mvi     c,readf ;set to read function
4bc3 cde04b         call    setfunc
4bc6 cdf04b         call    waitio  ;perform read function
4bc9 c9             ret             ;may have error set in reg-a
            ;
            ;
            write:  ;disk write function
4bca 0e06           mvi     c,writf
4bcc cde04b         call    setfunc ;set to write function
4bcf cdf04b         call    waitio
4bd2 c9             ret             ;may have error set
            ;
            ;
            ;       utility subroutines
            prmsg:  ;print message at h,l to 0
4bd3 7e             mov     a,m
4bd4 b7             ora     a       ;zero?
4bd5 c8             rz
            ;       more to print
4bd6 e5             push    h
4bd7 4f             mov     c,a
4bd8 cd6a4b         call    conout
4bdb e1             pop     h
4bdc 23             inx     h
4bdd c3d34b         jmp     prmsg
            ;
            setfunc:
            ;       set function for next i/o (command in reg-c)
4be0 21684c         lxi     h,iof   ;io function address
4be3 7e             mov     a,m     ;get it to accumulator for maskin
4be4 e6f8           ani     11111000b       ;remove previous command
4be6 b1             ora     c       ;set to new command
4be7 77             mov     m,a     ;replaced in iopb
            ;       the mds-800 controller req's disk bank bit in sec
            ;       mask the bit from the current i/o function
4be8 e620           ani     00100000b       ;mask the disk select bit
4bea 216b4c         lxi     h,ios           ;address the sector selec
4bed b6             ora     m               ;select proper disk bank
4bee 77             mov     m,a             ;set disk select bit on/o
4bef c9             ret
            ;
            waitio:
4bf0 0e0a           mvi     c,retry ;max retries before perm error
            rewait:
            ;       start the i/o function and wait for completion
4bf2 cd3f4c         call    intype  ;in rtype
4bf5 cd4c4c         call    inbyte  ;clears the controller
            ;
4bf8 3a664c         lda     dbank           ;set bank flags
```

```
4bfb b7                 ora     a               ;zero if drive Ø,1 and nz
4bfc 3e67               mvi     a,iopb and Øffh ;low address for iopb
4bfe Ø64c               mvi     b,iopb shr 8    ;high address for iopb
4cØØ c2Øb4c             jnz     iodrl   ;drive bank 1?
4cØ3 d379               out     ilow            ;low address to controlle
4cØ5 78                 mov     a,b
4cØ6 d37a               out     ihigh   ;high address
4cØ8 c31Ø4c             jmp     waitØ           ;to wait for complete
            ;
            iodrl:  ;drive bank 1
4cØb d389               out     ilow+1Øh        ;88 for drive bank 1Ø
4cØd 78                 mov     a,b
4cØe d38a               out     ihigh+1Øh
            ;
4c1Ø cd594c waitØ:      call    instat          ;wait for completion
4c13 e6Ø4               ani     iordy           ;ready?
4c15 ca1Ø4c             jz      waitØ
            ;
            ;           check io completion ok
4c18 cd3f4c             call    intype          ;must be io complete (ØØ)
            ;           ØØ unlinked i/o complete,   Ø1 linked i/o comple
            ;           1Ø disk status changed      11 (not used)
4c1b feØ2               cpi     1Øb             ;ready status change?
4c1d ca324c             jz      wready
            ;
            ;           must be ØØ in the accumulator
4c2Ø b7                 ora     a
4c21 c2384c             jnz     werror          ;some other condition, re
            ;
            ;           check i/o error bits
4c24 cd4c4c             call    inbyte
4c27 17                 ral
4c28 da324c             jc      wready          ;unit not ready
4c2b 1f                 rar
4c2c e6fe               ani     1111111Øb       ;any other errors?
4c2e c2384c             jnz     werror
            ;
            ;           read or write is ok, accumulator contains zero
4c31 c9                 ret
            ;
            wready: ;not ready, treat as error for now
4c32 cd4c4c             call    inbyte          ;clear result byte
4c35 c3384c             jmp     trycount
            ;
            werror: ;return hardware malfunction (crc, track, seek, e
            ;       the mds controller has returned a bit in each pos
            ;       of the accumulator, corresponding to the conditio
            ;       Ø       - deleted data (accepted as ok above)
            ;       1       - crc error
            ;       2       - seek error
            ;       3       - address error (hardware malfunction)
            ;       4       - data over/under flow (hardware malfunct
            ;       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 condit
               ;       but we will get a permanent error message if it i
               ;       recoverable.  in any case, the not ready conditio
               ;       treated as a separate condition for later improve
               trycount:
               ;       register c contains retry count, decrement 'til z
4c38 0d                dcr     c
4c39 c2f24b            jnz     rewait  ;for another try
               ;
               ;       cannot recover from error
4c3c 3e01              mvi     a,1     ;error code
4c3e c9                ret
               ;
               ;       intype, inbyte, instat read drive bank 00 or 10
4c3f 3a664c    intype: lda     dbank
4c42 b7                ora     a
4c43 c2494c            jnz     intypl  ;skip to bank 10
4c46 db79              in      rtype
4c48 c9                ret
4c49 db89      intypl: in      rtype+10h       ;78 for 0,1  88 for 2,3
4c4b c9                ret
               ;
4c4c 3a664c    inbyte: lda     dbank
4c4f b7                ora     a
4c50 c2564c            jnz     inbytl
4c53 db7b              in      rbyte
4c55 c9                ret
4c56 db8b      inbytl: in      rbyte+10h
4c58 c9                ret
               ;
4c59 3a664c    instat: lda     dbank
4c5c b7                ora     a
4c5d c2634c            jnz     instal
4c60 db78              in      dstat
4c62 c9                ret
4c63 db88      instal: in      dstat+10h
4c65 c9                ret
               ;
               ;
               ;
               ;       data areas (must be in ram)
4c66 00        dbank:  db      0       ;disk bank 00 if drive 0,1
                                       ;          10 if drive 2,3
               iopb:   ;io parameter block
4c67 80                db      80h     ;normal i/o operation
4c68 04        iof:    db      readf   ;io function, initial read
4c69 01        ion:    db      1       ;number of sectors to read
4c6a 02        iot:    db      offset  ;track number
4c6b 01        ios:    db      1       ;sector number
4c6c 8000      iod:    dw      buff    ;io address
               ;
               ;
               ;       define ram areas for bdos operation
```

```
                        endef
4c6e+=      begdat  equu    $
4c6e+       dirbuf: ds      128     ;directory access buffer
4cee+       alvØ:   ds      31
4dØd+       csvØ:   ds      16
4d1d+       alv1:   ds      31
4d3c+       csv1:   ds      16
4d4c+       alv2:   ds      31
4d6b+       csv2:   ds      16
4d7b+       alv3:   ds      31
4d9a+       csv3:   ds      16
4daa+=      enddat  equ     $
Ø13c+=      datsiz  equ     $-begdat
4daa                end
```

# APPENDIX C: A SKELETAL CBIOS

```
              ;       skeletal cbios for first level of cp/m 2.0 altera
              ;
0014 =        msize   equ     20      ;cp/m version memory size in kilo
              ;
              ;       "bias" is address offset from 3400h for memory sy
              ;       than 16k (referred to as "b" throughout the text)
              ;
0000 =        bias    equ     (msize-20)*1024
3400 =        ccp     equ     3400h+bias      ;base of ccp
3c06 =        bdos    equ     ccp+806h        ;base of bdos
4a00 =        bios    equ     ccp+1600h       ;base of bios
0004 =        cdisk   equ     0004h   ;current disk number 0=a,...,15=p
0003 =        iobyte  equ     0003h   ;intel i/o byte
              ;
4a00                  org     bios    ;origin of this program
002c =        nsects  equ     ($-ccp)/128     ;warm start sector count
              ;
              ;       jump vector for individual subroutines
4a00 c39c4a           jmp     boot            ;cold start
4a03 c3a64a wboote:   jmp     wboot           ;warm start
4a06 c3114b           jmp     const           ;console status
4a09 c3244b           jmp     conin           ;console character in
4a0c c3374b           jmp     conout          ;console character out
4a0f c3494b           jmp     list            ;list character out
4a12 c34d4b           jmp     punch           ;punch character out
4a15 c34f4b           jmp     reader          ;reader character out
4a18 c3544b           jmp     home            ;move head to home positi
4a1b c35a4b           jmp     seldsk          ;select disk
4a1e c37d4b           jmp     settrk          ;set track number
4a21 c3924b           jmp     setsec          ;set sector number
4a24 c3ad4b           jmp     setdma          ;set dma address
4a27 c3c34b           jmp     read            ;read disk
4a2a c3d64b           jmp     write           ;write disk
4a2d c34b4b           jmp     listst          ;return list status
4a30 c3a74b           jmp     sectran         ;sector translate
              ;
              ;       fixed data tables for four-drive standard
              ;       ibm-compatible 8" disks
              ;       disk parameter header for disk 00
4a33 734a00 dpbase:   dw      trans,0000h
4a37 000000           dw      0000h,0000h
4a3b f04c8d           dw      dirbf,dpblk
4a3f ec4d70           dw      chk00,all00
              ;       disk parameter header for disk 01
4a43 734a00           dw      trans,0000h
4a47 000000           dw      0000h,0000h
4a4b f04c8d           dw      dirbf,dpblk
4a4f fc4d8f           dw      chk01,all01
              ;       disk parameter header for disk 02
4a53 734a00           dw      trans,0000h
4a57 000000           dw      0000h,0000h
4a5b f04c8d           dw      dirbf,dpblk
4a5f 0c4eae           dw      chk02,all02
```

```
            ;       disk parameter header for disk Ø3
4a63 734aØØ         dw      trans,ØØØØh
4a67 ØØØØØØ         dw      ØØØØh,ØØØØh
4a6b fØ4c8d         dw      dirbf,dpblk
4a6f lc4ecd         dw      chkØ3,allØ3
            ;
            ;       sector translate vector
4a73 Ø1Ø7Ød trans:  db      1,7,13,19       ;sectors 1,2,3,4
4a77 19Ø5Øb         db      25,5,11,17      ;sectors 5,6,7,8
4a7b 1703Ø9         db      23,3,9,15       ;sectors 9,1Ø,11,12
4a7f 15Ø2Ø8         db      21,2,8,14       ;sectors 13,14,15,16
4a83 141aØ6         db      2Ø,26,6,12      ;sectors 17,18,19,2Ø
4a87 1218Ø4         db      18,24,4,1Ø      ;sectors 21,22,23,24
4a8b 1Ø16           db      16,22           ;sectors 25,26
            ;
            dpblk:  ;disk parameter block, common to all disks
4a8d laØØ           dw      26              ;sectors per track
4a8f Ø3             db      3               ;block shift factor
4a9Ø Ø7             db      7               ;block mask
4a91 ØØ             db      Ø               ;null mask
4a92 f2ØØ           dw      242             ;disk size-l
4a94 3fØØ           dw      63              ;directory max
4a96 cØ             db      192             ;alloc Ø
4a97 ØØ             db      Ø               ;alloc l
4a98 1ØØØ           dw      16              ;check size
4a9a Ø2ØØ           dw      2               ;track offset
            ;
            ;       end of fixed tables
            ;
            ;       individual subroutines to perform each function
            boot:   ;simplest case is to just perform parameter initi
4a9c af             xra     a               ;zero in the accum
4a9d 3203ØØ         sta     iobyte          ;clear the iobyte
4aaØ 3204ØØ         sta     cdisk           ;select disk zero
4aa3 c3ef4a         jmp     gocpm           ;initialize and go to cp/
            ;
            wboot:  ;simplest case is to read the disk until all sect
4aa6 318ØØØ         lxi     sp,8Øh          ;use space below buffer f
4aa9 ØeØØ           mvi     c,Ø             ;select disk Ø
4aab cd5a4b         call    seldsk
4aae cd544b         call    home            ;go to track ØØ
            ;
4abl Ø62c           mvi     b,nsects        ;b counts # of sectors to
4ab3 ØeØØ           mvi     c,Ø             ;c has the current track
4ab5 16Ø2           mvi     d,2             ;d has the next sector to
            ;       note that we begin by reading track Ø, sector 2 s
            ;       contains the cold start loader, which is skipped
4ab7 21ØØ34         lxi     h,ccp           ;base of cp/m (initial lo
            loadl:  ;load one more sector
4aba c5             push    b       ;save sector count, current track
4abb d5             push    d       ;save next sector to read
4abc e5             push    h       ;save dma address
4abd 4a             mov     c,d     ;get sector address to register c
4abe cd924b         call    setsec  ;set sector address from register
4acl cl             pop     b       ;recall dma address to b,c
```

10

```
4ac2 c5             push    b       ;replace on stack for later recal
4ac3 cdad4b         call    setdma  ;set dma address from b,c
            ;
            ;       drive set to 0, track set, sector set, dma addres
4ac6 cdc34b         call    read
4ac9 fe00           cpi     00h     ;any errors?
4acb c2a64a         jnz     wboot   ;retry the entire boot if an erro
            ;
            ;       no error, move to next sector
4ace e1             pop     h       ;recall dma address
4acf 118000         lxi     d,128   ;dma=dma+128
4ad2 19             dad     d       ;new dma address is in h,l
4ad3 d1             pop     d       ;recall sector address
4ad4 c1             pop     b       ;recall number of sectors remaini
4ad5 05             dcr     b       ;sectors=sectors-1
4ad6 caef4a         jz      gocpm   ;transfer to cp/m if all have bee
            ;
            ;       more sectors remain to load, check for track chan
4ad9 14             inr     d
4ada 7a             mov     a,d     ;sector=27?, if so, change tracks
4adb fe1b           cpi     27
4add daba4a         jc      load1   ;carry generated if sector<27
            ;
            ;       end of current track, go to next track
4ae0 1601           mvi     d,1     ;begin with first sector of next
4ae2 0c             inr     c       ;track=track+1
            ;
            ;       save register state, and change tracks
4ae3 c5             push    b
4ae4 d5             push    d
4ae5 e5             push    h
4ae6 cd7d4b         call    settrk  ;track address set from register
4ae9 e1             pop     h
4aea d1             pop     d
4aeb c1             pop     b
4aec c3ba4a         jmp     load1   ;for another sector
            ;
            ;       end of load operation, set parameters and go to c
            gocpm:
4aef 3ec3           mvi     a,0c3h  ;c3 is a jmp instruction
4af1 320000         sta     0       ;for jmp to wboot
4af4 21034a         lxi     h,wboote        ;wboot entry point
4af7 220100         shld    1       ;set address field for jmp at 0
            ;
4afa 320500         sta     5       ;for jmp to bdos
4afd 21063c         lxi     h,bdos  ;bdos entry point
4b00 220600         shld    6       ;address field of jump at 5 to bd
            ;
4b03 018000         lxi     b,80h   ;default dma address is 80h
4b06 cdad4b         call    setdma
            ;
4b09 fb             ei              ;enable the interrupt system
4b0a 3a0400         lda     cdisk   ;get current disk number
4b0d 4f             mov     c,a     ;send to the ccp
4b0e c30034         jmp     ccp     ;go to cp/m for further processin
```

```
                ;
                ;
                ;       simple i/o handlers (must be filled in by user)
                ;       in each case, the entry point is provided, with s
                ;       to insert your own code
                ;
                const:  ;console status, return Øffh if character ready,
4b11                    ds      1Øh     ;space for status subroutine
4b21 3eØØ               mvi     a,ØØh
4b23 c9                 ret
                ;
                conin:  ;console character into register a
4b24                    ds      1Øh     ;space for input routine
4b34 e67f               ani     7fh     ;strip parity bit
4b36 c9                 ret
                ;
                conout: ;console character output from register c
4b37 79                 mov     a,c     ;get to accumulator
4b38                    ds      1Øh     ;space for output routine
4b48 c9                 ret
                ;
                list:   ;list character from register c
4b49 79                 mov     a,c     ;character to register a
4b4a c9                 ret             ;null subroutine
                ;
                listst: ;return list status (Ø if not ready, 1 if ready)
4b4b af                 xra     a       ;Ø is always ok to return
4b4c c9                 ret
                ;
                punch:  ;punch character from register c
4b4d 79                 mov     a,c     ;character to register a
4b4e c9                 ret             ;null subroutine
                ;
                ;
                reader: ;read character into register a from reader devic
4b4f 3e1a               mvi     a,1ah   ;enter end of file for now (repla
4b51 e67f               ani     7fh     ;remember to strip parity bit
4b53 c9                 ret
                ;
                ;
                ;       i/o drivers for the disk follow
                ;       for now, we will simply store the parameters away
                ;       in the read and write subroutines
                ;
                home:   ;move to the track ØØ position of current drive
                ;       translate this call into a settrk call with param
4b54 Øe ØØ              mvi     c,Ø     ;select track Ø
4b56 cd7d4b             call    settrk
4b59 c9                 ret             ;we will move to ØØ on first read
                ;
                seldsk: ;select disk given by register c
4b5a 21ØØØØ             lxi     h,ØØØØh ;error return code
4b5d 79                 mov     a,c
4b5e 32ef4c             sta     diskno
4b61 feØ4               cpi     4       ;must be between Ø and 3
```

```
4b63 d0             rnc             ;no carry if 4,5,...
            ;       disk number is in the proper range
4b64                ds      10      ;space for disk select
            ;       compute proper disk parameter header address
4b6e 3aef4c         lda     diskno
4b71 6f             mov     l,a     ;l=disk number 0,1,2,3
4b72 2600           mvi     h,0     ;high order zero
4b74 29             dad     h       ;*2
4b75 29             dad     h       ;*4
4b76 29             dad     h       ;*8
4b77 29             dad     h       ;*16 (size of each header)
4b78 11334a         lxi     d,dpbase
4b7b 19             dad     d       ;hl=.dpbase(diskno*16)
4b7c c9             ret
            ;
            settrk: ;set track given by register c
4b7d 79             mov     a,c
4b7e 32e94c         sta     track
4b81                ds      10h     ;space for track select
4b91 c9             ret
            ;
            setsec: ;set sector given by register c
4b92 79             mov     a,c
4b93 32eb4c         sta     sector
4b96                ds      10h     ;space for sector select
4ba6 c9             ret
            ;
            sectran:
                    ;translate the sector given by bc using the
                    ;translate table given by de
4ba7 eb             xchg            ;hl=.trans
4ba8 09             dad     b       ;hl=.trans(sector)
4ba9 6e             mov     l,m     ;l = trans(sector)
4baa 2600           mvi     h,0     ;hl= trans(sector)
4bac c9             ret             ;with value in hl
            ;
            setdma: ;set dma address given by registers b and c
4bad 69             mov     l,c     ;low order address
4bae 60             mov     h,b     ;high order address
4baf 22ed4c         shld    dmaad   ;save the address
4bb2                ds      10h     ;space for setting the dma addres
4bc2 c9             ret
            ;
            read:   ;perform read operation (usually this is similar
            ;       so we will allow space to set up read command, th
            ;       common code in write)
4bc3                ds      10h     ;set up read command
4bd3 c3e64b         jmp     waitio  ;to perform the actual i/o
            ;
            write:  ;perform a write operation
4bd6                ds      10h     ;set up write command
            ;
            waitio: ;enter here from read and write to perform the ac
            ;       operation.  return a 00h in register a if the ope
            ;       properly, and 01h if an error occurs during the r
```

```
      ;
      ;   in this case, we have saved the disk number in 'd
      ;           the track number in 'track' (0-76
      ;           the sector number in 'sector' (1-
      ;           the dma address in 'dmaad' (0-655
4be6            ds   256   ;space reserved for i/o drivers
4ce6 3e01       mvi  a,1   ;error condition
4ce8 c9         ret        ;replaced when filled-in
      ;
      ;   the remainder of the cbios is reserved uninitiali
      ;   data area, and does not need to be a part of the
      ;   system memory image (the space must be available,
      ;   however, between "begdat" and "enddat").
      ;
4ce9  track:  ds   2       ;two bytes for expansion
4ceb  sector: ds   2       ;two bytes for expansion
4ced  dmaad:  ds   2       ;direct memory address
4cef  diskno: ds   1       ;disk number 0-15
      ;
      ;   scratch ram area for bdos use
4cf0 =  begdat  equ  $     ;beginning of data area
4cf0  dirbf:  ds   128     ;scratch directory area
4d70  all00:  ds   31      ;allocation vector 0
4d8f  all01:  ds   31      ;allocation vector 1
4dae  all02:  ds   31      ;allocation vector 2
4dcd  all03:  ds   31      ;allocation vector 3
4dec  chk00:  ds   16      ;check vector 0
4dfc  chk01:  ds   16      ;check vector 1
4e0c  chk02:  ds   16      ;check vector 2
4e1c  chk03:  ds   16      ;check vector 3
      ;
4e2c =  enddat  equ  $     ;end of data area
013c =  datsiz  equ  $-begdat;size of data area
4e2c            end
```

# APPENDIX D: A SKELETAL GETSYS/PUTSYS PROGRAM

```
                 ;       combined getsys and putsys programs from Sec 4.
                 ;       Start the programs at the base of the TPA

Ø1ØØ                     org     Ø1ØØh

ØØ14 =           msize   equ     2Ø                ; size of cp/m in Kbytes

                 ; "bias" is the amount to add to addresses for > 2Øk
                 ;       (referred to as "b" throughout the text)

ØØØØ =           bias    equ     (msize-2Ø)*1Ø24
34ØØ =           ccp     equ     34ØØh+bias
3cØØ =           bdos    equ     ccp+Ø8ØØh
4aØØ =           bios    equ     ccp+16ØØh

                 ;       getsys programs tracks Ø and 1 to memory at
                 ;       388Øh + bias

                 ;       register                usage
                 ;         a                 (scratch register)
                 ;         b                 track count (Ø...76)
                 ;         c                 sector count (1...26)
                 ;         d,e               (scratch register pair)
                 ;         h,l               load address
                 ;         sp                set to stack address

                 gstart:                                   ; start of getsys
Ø1ØØ 318Ø33              lxi     sp,ccp-ØØ8Øh              ; convenient plac
Ø1Ø3 218Ø33              lxi     h,ccp-ØØ8Øh               ; set initial loa
Ø1Ø6 Ø6ØØ                mvi     b,Ø                       ; start with trac
                 rd$trk:                                   ; read next track
Ø1Ø8 ØeØ1                mvi     c,1                       ; each track star
                 rd$sec:
Ø1Øa cdØØØ3              call    read$sec                  ; get the next se
Ø1Ød 118ØØØ              lxi     d,128                     ; offset by one s
Ø11Ø 19                  dad     d                         ;   (hl=hl+128)
Ø111 Øc                  inr     c                         ; next sector
Ø112 79                  mov     a,c                       ; fetch sector nu
Ø113 fe1b                cpi     27                        ;   and see if la
Ø115 daØaØ1              jc      rdsec                     ; <, do one more

                 ; arrive here at end of track, move to next track

Ø118 Ø4                  inr     b                         ; track = track+1
Ø119 78                  mov     a,b                       ; check for last
Ø11a feØ2                cpi     2                         ; track = 2 ?
Ø11c daØ8Ø1              jc      rd$trk                    ; <, do another

                 ; arrive here at end of load, halt for lack of anything b

Ø11f fb                  ei
Ø12Ø 76                  hlt
```

```
            ;       putsys program, places memory image starting at
            ;       388Øh + bias back to tracks Ø and 1
            ;       start this program at the next page boundary

Ø2ØØ                org     ($+Ø1ØØh) and ØffØØh

            put$sys:
Ø2ØØ 318Ø33         lxi     sp,ccp-ØØ8Øh            ; convenient plac
Ø2Ø3 218Ø33         lxi     h,ccp-ØØ8Øh             ; start of dump
Ø2Ø6 Ø6ØØ           mvi     b,Ø                     ; start with trac
            wr$trk:
Ø2Ø8 ØeØ1           mvi     c,1                     ; start with sect
            wr$sec:
Ø2Øa cdØØØ4         call    write$sec               ; write one secto
Ø2Ød 118ØØØ         lxi     d,128                   ; length of each
Ø21Ø 19             dad     d                       ; <hl>=<hl> + 128
Ø211 Øc             inr     c                       ; <c> = <c> + 1
Ø212 79             mov     a,c                     ; see if
Ø213 fe1b           cpi     27                      ;   past end of t
Ø215 daØaØ2         jc      wr$sec                  ; no, do another

            ;  arrive here at end of track, move to next track

Ø218 Ø4             inr     b                       ; track = track+1
Ø219 78             mov     a,b                     ; see if
Ø21a feØ2           cpi     2                       ;   last track
Ø21c daØ8Ø2         jc      wr$trk                  ; no, do another

            ;       done with putsys, halt for lack of anything bette

Ø21f fb             ei
Ø22Ø 76             hlt


            ; user supplied subroutines for sector read and write

            ;       move to next page boundary

Ø3ØØ                org     ($+Ø1ØØh) and ØffØØh

            read$sec:
                    ; read the next sector
                    ; track in <b>,
                    ; sector in <c>
                    ; dmaaddr in <hl>

Ø3ØØ c5             push    b
Ø3Ø1 e5             push    h

            ; user defined read operation goes here
Ø3Ø2                ds      64

Ø342 e1             pop     h
Ø343 c1             pop     b
```

```
Ø344 c9             ret

Ø4ØØ                org     ($+Ø1ØØh) and ØffØØh    ; another page bo

          write$sec:

                    ; same parameters as read$sec

Ø4ØØ c5             push    b
Ø4Ø1 e5             push    h

          ; user defined write operation goes here
Ø4Ø2                ds      64

Ø442 el             pop     h
Ø443 cl             pop     b
Ø444 c9             ret

          ; end of getsys/putsys program

Ø445                end
```

## APPENDIX E: A SKELETAL COLD START LOADER

```
            ; this is a sample cold start loader which, when modified
            ; resides on track ØØ, sector Ø1 (the first sector on the
            ; diskette).  we assume that the controller has loaded
            ; this sector into memory upon system start-up (this pro-
            ; gram can be keyed-in, or can exist in 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" (34ØØh + "bias").  in a 2Øk
            ; memory system, the value of "bias" is ØØØØh, with large
            ; values for increased memory sizes (see section 2). afte
            ; loading the cp/m system, the clod start loader branches
            ; to the "boot" entry point of the bios, which begins at
            ; "bios" + "bias." the cold start loader is not used un-
            ; til the system is powered up again, as long as the bios
            ; is not overwritten.  the origin is assumed at ØØØØh, an
            ; must be changed if the controller brings the cold start
            ; loader into another area, or if a read/only memory area
            ; is used.

ØØØØ                org     Ø               ; base of ram in cp/m

ØØ14 =      msize   equ     2Ø              ; min mem size in kbytes

ØØØØ =      bias    equ     (msize-2Ø)*1Ø24 ; offset from 2Øk system
3400 =      ccp     equ     34ØØh+bias      ; base of the ccp
4aØØ =      bios    equ     ccp+16ØØh       ; base of the bios
Ø300 =      biosl   equ     Ø3ØØh           ; length of the bios
4aØØ =      boot    equ     bios
19ØØ =      size    equ     bios+biosl-ccp  ; size of cp/m system
ØØ32 =      sects   equ     size/128        ; # of sectors to load

            ;       begin the load operation

            cold:
ØØØØ Ø102ØØ         lxi     b,2             ; b=Ø, c=sector 2
ØØØ3 1632           mvi     d,sects         ; d=# sectors to load
ØØØ5 21ØØ34         lxi     h,ccp           ; base transfer address

            lsect:  ; load the next sector

            ;       insert inline code at this point to
            ;       read one 128 byte sector from the
            ;       track given in register b, sector
            ;       given in register c,
            ;       into the address given by <hl>
            ;
            ; branch to location "cold" if a read error occurs
            ;       *************************************************
            ;       *
            ;       *       user supplied read operation goes here...
            ;       *
            ;       *************************************************

0008 c36b00         jmp     past$patch      ; remove this when patche
000b                ds      60h

            past$patch:
            ; go to next sector if load is incomplete
006b 15             dcr     d               ; sects=sects-1
006c ca004a         jz      boot            ; head for the bios

            ;       more sectors to load
            ;
            ; we aren't using a stack, so use <sp> as scratch registe
            ;       to hold the load address increment

006f 318000         lxi     sp,128          ; 128 bytes per sector
0072 39             dad     sp              ; <hl> = <hl> + 128

0073 0c             inr     c               ; sector = sector + 1
0074 79             mov     a,c
0075 fe1b           cpi     27              ; last sector of track?
0077 da0800         jc      lsect           ; no, go read another

            ; end of track, increment to next track

007a 0e01           mvi     c,1             ; sector = 1
007c 04             inr     b               ; track = track + 1
007d c30800         jmp     lsect           ; for another group
0080                end                     ; of boot loader
```

```
;       CP/M 2.0 disk re-definition library
;
;       Copyright (c) 1979
;       Digital Research
;       Box 579
;       Pacific Grove, CA
;       93950
;
;       CP/M logical disk drives are defined using the
;       macros given below, where the sequence of calls
;       is:
;
;       disks   n
;       diskdef parameter-list-0
;       diskdef parameter-list-1
;       ...
;       diskdef parameter-list-n
;       endef
;
;       where n is the number of logical disk drives attached
;       to the CP/M system, and parameter-list-i defines the
;       characteristics of the ith drive (i=0,1,...,n-1)
;
;       each parameter-list-i takes the form
;               dn,fsc,lsc,[skf],bls,dks,dir,cks,ofs,[0]
;       where
;       dn      is the disk number 0,1,...,n-1
;       fsc     is the first sector number (usually 0 or 1)
;       lsc     is the last sector number on a track
;       skf     is optional "skew factor" for sector translate
;       bls     is the data block size (1024,2048,...,16384)
;       dks     is the disk size in bls increments (word)
;       dir     is the number of directory elements (word)
;       cks     is the number of dir elements to checksum
;       ofs     is the number of tracks to skip (word)
;       [0]     is an optional 0 which forces 16K/directory en
;
;       for convenience, the form
;               dn,dm
;       defines disk dn as having the same characteristics as
;       a previously defined disk dm.
;
;       a standard four drive CP/M system is defined by
;               disks   4
;               diskdef 0,1,26,6,1024,243,64,64,2
;       dsk     set     0
;               rept    3
;       dsk     set     dsk+1
;               diskdef %dsk,0
;               endm
;               endef
;
;       the value of "begdat" at the end of assembly defines t
;       beginning of the uninitialize ram area above the bios,
;       while the value of "enddat" defines the next location
;       following the end of the data area.  the size of this
;       area is given by the value of "datsiz" at the end of t
;       assembly.  note that the allocation vector will be qui
;       large if a large disk size is defined with a small blo
;       size.
;
dskhdr  macro   dn
;;      define a single disk header list
dpe&dn: dw      xlt&dn,0000h    ;translate table
        dw      0000h,0000h     ;scratch area
        dw      dirbuf,dpb&dn   ;dir buff,parm block
        dw      csv&dn,alv&dn   ;check, alloc vectors
        endm
;
disks   macro   nd
;;      define nd disks
ndisks  set     nd      ;;for later reference
dpbase  equ     $       ;base of disk parameter blocks
;;      generate the nd elements
dsknxt  set     0
        rept    nd
        dskhdr  %dsknxt
dsknxt  set     dsknxt+1
        endm
        endm
;
dpbhdr  macro   dn
dpb&dn  equ     $               ;disk parm block
        endm
;
ddb     macro   data,comment
;;      define a db statement
        db      data            comment
        endm
;
ddw     macro   data,comment
;;      define a dw statement
        dw      data            comment
        endm
;
gcd     macro   m,n
;;      greatest common divisor of m,n
;;      produces value gcdn as result
;;      (used in sector translate table generation)
gcdm    set     m       ;;variable for m
gcdn    set     n       ;;variable for n
gcdr    set     0       ;;variable for r
        rept    65535
gcdx    set     gcdm/gcdn
gcdr    set     gcdm - gcdx*gcdn
        if      gcdr = 0
        exitm
        endif
```

```
gcdm    set     gcdn
gcdn    set     gcdr
        endm
        endm
;
diskdef macro   dn,fsc,lsc,skf,bls,dks,dir,cks,ofs,k16
;;      generate the set statements for later tables
        if      nul lsc
;;      current disk dn same as previous fsc
dpb&dn  equ     dpb&fsc ;equivalent parameters
als&dn  equ     als&fsc ;same allocation vector size
css&dn  equ     css&fsc ;same checksum vector size
xlt&dn  equ     xlt&fsc ;same translate table
        else
secmax  set     lsc-(fsc)       ;;sectors 0...secmax
sectors set     secmax+1;;number of sectors
als&dn  set     (dks)/8 ;;size of allocation vector
        if      ((dks) mod 8) ne 0
als&dn  set     als&dn+1
        endif
css&dn  set     (cks)/4 ;;number of checksum elements
;;      generate the block shift value
blkval  set     bls/128 ;;number of sectors/block
blkshf  set     0       ;;counts right 0's in blkval
blkmsk  set     0       ;;fills with 1's from right
        rept    16      ;;once for each bit position
        if      blkval=1
        exitm
        endif
;;      otherwise, high order 1 not found yet
blkshf  set     blkshf+1
blkmsk  set     (blkmsk shl 1) or 1
blkval  set     blkval/2
        endm
;;      generate the extent mask byte
blkval  set     bls/1024        ;;number of kilobytes/block
extmsk  set     0       ;;fill from right with 1's
        rept    16
        if      blkval=1
        exitm
        endif
;;      otherwise more to shift
extmsk  set     (extmsk shl 1) or 1
blkval  set     blkval/2
        endm
;;      may be double byte allocation
        if      (dks) > 256
extmsk  set     (extmsk shr 1)
        endif
;;      may be optional [0] in last position
        if      not nul k16
extmsk  set     k16
        endif
;;      now generate directory reservation bit vector
dirrem  set     dir     ;;# remaining to process
```

```
dirbks  set     bls/32  ;;number of entries per block
dirblk  set     0       ;;fill with 1's on each loop
        rept    16
        if      dirrem=0
        exitm
        endif
;;      not complete, iterate once again
;;      shift right and add 1 high order bit
dirblk  set     (dirblk shr 1) or 8000h
        if      dirrem > dirbks
dirrem  set     dirrem-dirbks
        else
dirrem  set     0
        endif
        endm
        dpbhdr  dn      ;;generate equ $
        ddw     %sectors,<;sec per track>
        ddb     %blkshf,<;block shift>
        ddb     %blkmsk,<;block mask>
        ddb     %extmsk,<;extnt mask>
        ddw     %(dks)-1,<;disk size-1>
        ddw     %(dir)-1,<;directory max>
        ddb     %dirblk shr 8,<;alloc0>
        ddb     %dirblk and 0ffh,<;alloc1>
        ddw     %(cks)/4,<;check size>
        ddw     %ofs,<;offset>
;;      generate the translate table, if requested
        if      nul skf
xlt&dn  equ     0                       ;no xlate table
        else
        if      skf = 0
xlt&dn  equ     0                       ;no xlate table
        else
;;      generate the translate table
nxtsec  set     0       ;;next sector to fill
nxtbas  set     0       ;;moves by one on overflow
        gcd     %sectors,skf
;;      gcdn = gcd(sectors,skew)
neltst  set     sectors/gcdn
;;      neltst is number of elements to generate
;;      before we overlap previous elements
nelts   set     neltst  ;;counter
xlt&dn  equ     $                       ;translate table
        rept    sectors ;;once for each sector
        if      sectors < 256
        ddb     %nxtsec+(fsc)
        else
        ddw     %nxtsec+(fsc)
        endif
nxtsec  set     nxtsec+(skf)
        if      nxtsec >= sectors
nxtsec  set     nxtsec-sectors
        endif
nelts   set     nelts-1
        if      nelts = 0
```

```
nxtbas  set     nxtbas+1
nxtsec  set     nxtbas
nelts   set     neltst
        endif
        endm
        endif   ;;end of nul fac test
        endif   ;;end of nul bls test
        endm
;
defds   macro   lab,space
lab:    ds      space
        endm
;
lds     macro   lb,dn,val
        defds   lb&dn,%val&dn
        endm
;
endef   macro
;;      generate the necessary ram data areas
begdat  equ     $
dirbuf: ds      128     ;directory access buffer
dsknxt  set     Ø
        rept    ndisks  ;;once for each disk
        lds     alv,%dsknxt,als
        lds     csv,%dsknxt,css
dsknxt  set     dsknxt+1
        endm
enddat  equ     $
datsiz  equ     $-begdat
;;      db Ø at this point forces hex record
        endm
```

APPENDIX G:  BLOCKING AND DEBLOCKING ALGORITHMS.

```
;*****************************************************
;*                                                   *
;*      Sector Deblocking Algorithms for CP/M 2.0    *
;*                                                   *
;*****************************************************
;
;       utility macro to compute sector mask
smask   macro   hblk
;;      compute log2(hblk), return @x as result
;;      (2 ** @x = hblk on return)
@y      set     hblk
@x      set     0
;;      count right shifts of @y until = 1
        rept    8
        if      @y = 1
        exitm
        endif
;;      @y is not 1, shift right one position
@y      set     @y shr 1
@x      set     @x + 1
        endm
        endm
;
;*****************************************************
;*                                                   *
;*         CP/M to host disk constants               *
;*                                                   *
;*****************************************************
blksiz  equ     2048            ;CP/M allocation size
hstsiz  equ     512             ;host disk sector size
hstspt  equ     20              ;host disk sectors/trk
hstblk  equ     hstsiz/128      ;CP/M sects/host buff
cpmspt  equ     hstblk * hstspt ;CP/M sectors/track
secmsk  equ     hstblk-1        ;sector mask
        smask   hstblk          ;compute sector mask
secshf  equ     @x              ;log2(hstblk)
;
;*****************************************************
;*                                                   *
;*        BDOS constants on entry to write           *
;*                                                   *
;*****************************************************
wrall   equ     0               ;write to allocated
wrdir   equ     1               ;write to directory
wrual   equ     2               ;write to unallocated
;
;*****************************************************
;*                                                   *
;*      The BDOS entry points given below show the   *
;*      code which is relevant to deblocking only.   *
;*                                                   *
;*****************************************************
;
```

```
;       DISKDEF macro, or hand coded tables go here
dpbase  equ     $               ;disk param block base
;
boot:
wboot:
        ;enter here on system boot to initialize
        xra     a               ;Ø to accumulator
        sta     hstact          ;host buffer inactive
        sta     unacnt          ;clear unalloc count
        ret
;
seldsk:
        ;select disk
        mov     a,c             ;selected disk number
        sta     sekdsk          ;seek disk number
        mov     l,a             ;disk number to HL
        mvi     h,Ø
        rept    4               ;multiply by 16
        dad     h
        endm
        lxi     d,dpbase        ;base of parm block
        dad     d               ;hl=.dpb(curdsk)
        ret
;
settrk:
        ;set track given by registers BC
        mov     h,b
        mov     l,c
        shld    sektrk          ;track to seek
        ret
;
setsec:
        ;set sector given by register c
        mov     a,c
        sta     seksec          ;sector to seek
        ret
;
setdma:
        ;set dma address given by BC
        mov     h,b
        mov     l,c
        shld    dmaadr
        ret
;
sectran:
        ;translate sector number BC
        mov     h,b
        mov     l,c
        ret
;
```

```
;*****************************************************
;*                                                   *
;*      The READ entry point takes the place of      *
;*      the previous BIOS defintion for READ.        *
;*                                                   *
;*****************************************************
read:
        ;read the selected CP/M sector
        mvi     a,1
        sta     readop          ;read operation
        sta     rsflag          ;must read data
        mvi     a,wrual
        sta     wrtype          ;treat as unalloc
        jmp     rwoper          ;to perform the read
;
;*****************************************************
;*                                                   *
;*      The WRITE entry point takes the place of     *
;*      the previous BIOS defintion for WRITE.       *
;*                                                   *
;*****************************************************
write:
        ;write the selected CP/M sector
        xra     a               ;Ø to accumulator
        sta     readop          ;not a read operation
        mov     a,c             ;write type in c
        sta     wrtype
        cpi     wrual           ;write unallocated?
        jnz     chkuna          ;check for unalloc
;
;       write to unallocated, set parameters
        mvi     a,blksiz/128    ;next unalloc recs
        sta     unacnt
        lda     sekdsk          ;disk to seek
        sta     unadsk          ;unadsk = sekdsk
        lhld    sektrk
        shld    unatrk          ;unatrk = sectrk
        lda     seksec
        sta     unasec          ;unasec = seksec
;
chkuna:
        ;check for write to unallocated sector
        lda     unacnt          ;any unalloc remain?
        ora     a
        jz      alloc           ;skip if not
;
;       more unallocated records remain
        dcr     a               ;unacnt = unacnt-1
        sta     unacnt
        lda     sekdsk          ;same disk?
        lxi     h,unadsk
        cmp     m               ;sekdsk = unadsk?
        jnz     alloc           ;skip if not
;
;       disks are the same
```

```
        lxi     h,unatrk
        call    sektrkcmp       ;sektrk = unatrk?
        jnz     alloc           ;skip if not
;
;       tracks are the same
        lda     seksec          ;same sector?
        lxi     h,unasec
        cmp     m               ;seksec = unasec?
        jnz     alloc           ;skip if not
;
;       match, move to next sector for future ref
        inr     m               ;unasec = unasec+1
        mov     a,m             ;end of track?
        cpi     cpmspt          ;count CP/M sectors
        jc      noovf           ;skip if no overflow
;
;       overflow to next track
        mvi     m,Ø             ;unasec = Ø
        lhld    unatrk
        inx     h
        shld    unatrk          ;unatrk = unatrk+1
;
noovf:
        ;match found, mark as unnecessary read
        xra     a               ;Ø to accumulator
        sta     rsflag          ;rsflag = Ø
        jmp     rwoper          ;to perform the write
;
alloc:
        ;not an unallocated record, requires pre-read
        xra     a               ;Ø to accum
        sta     unacnt          ;unacnt = Ø
        inr     a               ;1 to accum
        sta     rsflag          ;rsflag = 1
;
;*****************************************************
;*                                                   *
;*      Common code for READ and WRITE follows       *
;*                                                   *
;*****************************************************
rwoper:
        ;enter here to perform the read/write
        xra     a               ;zero to accum
        sta     erflag          ;no errors (yet)
        lda     seksec          ;compute host sector
        rept    secshf
        ora     a               ;carry = Ø
        rar                     ;shift right
        endm
        sta     sekhst          ;host sector to seek
;
;       active host sector?
        lxi     h,hstact        ;host active flag
        mov     a,m
        mvi     m,1             ;always becomes 1
        ora     a               ;was it already?
        jz      filhst          ;fill host if not
;
;       host buffer active, same as seek buffer?
        lda     sekdsk
        lxi     h,hstdsk        ;same disk?
        cmp     m               ;sekdsk = hstdsk?
        jnz     nomatch
;
;       same disk, same track?
        lxi     h,hsttrk
        call    sektrkcmp       ;sektrk = hsttrk?
        jnz     nomatch
;
;       same disk, same track, same buffer?
        lda     sekhst
        lxi     h,hstsec        ;sekhst = hstsec?
        cmp     m
        jz      match           ;skip if match
;
nomatch:
        ;proper disk, but not correct sector
        lda     hstwrt          ;host written?
        ora     a
        cnz     writehst        ;clear host buff
;
filhst:
        ;may have to fill the host buffer
        lda     sekdsk
        sta     hstdsk
        lhld    sektrk
        shld    hsttrk
        lda     sekhst
        sta     hstsec
        lda     rsflag          ;need to read?
        ora     a
        cnz     readhst         ;yes, if 1
        xra     a               ;0 to accum
        sta     hstwrt          ;no pending write
;
match:
        ;copy data to or from buffer
        lda     seksec          ;mask buffer number
        ani     secmsk          ;least signif bits
        mov     l,a             ;ready to shift
        mvi     h,0             ;double count
        rept    7               ;shift left 7
        dad     h
        endm
;       hl has relative host buffer address
        lxi     d,hstbuf
        dad     d               ;hl = host address
        xchg                    ;now in DE
        lhld    dmaadr          ;get/put CP/M data
        mvi     c,128           ;length of move
        lda     readop          ;which way?
        ora     a
        jnz     rwmove          ;skip if read
;
;       write operation, mark and switch direction
        mvi     a,1
        sta     hstwrt          ;hstwrt = 1
        xchg                    ;source/dest swap
;
rwmove:
        ;C initially 128, DE is source, HL is dest
        ldax    d               ;source character
        inx     d
        mov     m,a             ;to dest
        inx     h
        dcr     c               ;loop 128 times
        jnz     rwmove
;
;       data has been moved to/from host buffer
        lda     wrtype          ;write type
        cpi     wrdir           ;to directory?
        lda     erflag          ;in case of errors
        rnz                     ;no further processing
;
;       clear host buffer for directory write
        ora     a               ;errors?
        rnz                     ;skip if so
        xra     a               ;0 to accum
        sta     hstwrt          ;buffer written
        call    writehst
        lda     erflag
        ret
;
;*****************************************************
;*                                                   *
;*      Utility subroutine for 16-bit compare        *
;*                                                   *
;*****************************************************
sektrkcmp:
        ;HL = .unatrk or .hsttrk, compare with sektrk
        xchg
        lxi     h,sektrk
        ldax    d               ;low byte compare
        cmp     m               ;same?
        rnz                     ;return if not
;       low bytes equal, test high 1s
        inx     d
        inx     h
        ldax    d
        cmp     m       ;sets flags
        ret
;
```

```
;*****************************************************
;*                                                   *
;*      WRITEHST performs the physical write to      *
;*      the host disk, READHST reads the physical    *
;*      disk.                                        *
;*                                                   *
;*****************************************************
writehst:
        ;hstdsk = host disk #, hsttrk = host track #,
        ;hstsec = host sect #. write "hstsiz" bytes
        ;from hstbuf and return error flag in erflag.
        ;return erflag non-zero if error
        ret
;
readhst:
        ;hstdsk = host disk #, hsttrk = host track #,
        ;hstsec = host sect #. read "hstsiz" bytes
        ;into hstbuf and return error flag in erflag.
        ret
;
;*****************************************************
;*                                                   *
;*      Unitialized RAM data areas                   *
;*                                                   *
;*****************************************************
;
sekdsk: ds      1               ;seek disk number
sektrk: ds      2               ;seek track number
seksec: ds      1               ;seek sector number
;
hstdsk: ds      1               ;host disk number
hsttrk: ds      2               ;host track number
hstsec: ds      1               ;host sector number
;
sekhst: ds      1               ;seek shr secshf
hstact: ds      1               ;host active flag
hstwrt: ds      1               ;host written flag
;
unacnt: ds      1               ;unalloc rec cnt
unadsk: ds      1               ;last unalloc disk
unatrk: ds      2               ;last unalloc track
unasec: ds      1               ;last unalloc sector
;
erflag: ds      1               ;error reporting
rsflag: ds      1               ;read sector flag
readop: ds      1               ;1 if read operation
wrtype: ds      1               ;write operation type
dmaadr: ds      2               ;last dma address
hstbuf: ds      hstsiz          ;host buffer
;
```

```
;*****************************************************
;*                                                   *
;*      The ENDEF macro invocation goes here         *
;*                                                   *
;*****************************************************
        end
```

# MICROSOFT BASIC 80
# REFERENCE MANUAL

1

# MICROSOFT

# BASIC-80

## release 5.0

# reference manual

11

Revision 1

# Introduction

BASIC-80 is the most extensive implementation of BASIC available for the 8080 and Z80 microprocessors. In its fifth major release (Release 5.0), BASIC-80 meets the ANSI qualifications for BASIC, as set forth in document BSRX3.60-1978. Each release of BASIC-80 consists of three upward compatible versions: 8K, Extended and Disk . This manual is a reference for all three versions of BASIC-80, release 5.0 and later. This manual is also a reference for Microsoft BASIC-86 and the Microsoft BASIC Compiler. BASIC-86 is currently available in Extended and Disk Standalone versions, which are comparable to the BASIC-80 Extended and Disk Standalone versions.

There are significant differences between the 5.0 release of BASIC-80 and the previous releases (release 4.51 and earlier). If you have programs written under a previous release of BASIC-80, check Appendix A for new features in 5.0 that may affect execution.

The manual is divided into three large chapters plus a number of appendices. Chapter 1 covers a variety of topics, largely pertaining to information representation when using BASIC-80. Chapter 2 contains the syntax and semantics of every command and statement in BASIC-80, ordered alphabetically. Chapter 3 describes all of BASIC-80's intrinsic functions, also ordered alphabetically. The appendices contain information pertaining to individual operating systems; plus lists of error messages, ASCII codes, and math functions; and helpful information on assembly language subroutines and disk I/O.

# BASIC-80 Reference Manual

## CONTENTS

# CHAPTER 1

# GENERAL INFORMATION ABOUT BASIC-80

## 1.1 INITIALIZATION

The procedure for initialization will vary with different implementations of BASIC-80. Check the appropriate appendix at the back of this manual to determine how BASIC-80 is initialized with your operating system.

## 1.2 MODES OF OPERATION

When BASIC-80 is initialized, it types the prompt "Ok". "Ok" means BASIC-80 is at command level, that is, it is ready to accept commands. At this point, BASIC-80 may be used in either of two modes: the direct mode or the indirect mode.

In the direct mode, BASIC commands and statements are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a "calculator" for quick computations that do not require a complete program.

The indirect mode is the mode used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

## 1.3 LINE FORMAT

Program lines in a BASIC program have the following format (square brackets indicate optional):

```
nnnnn BASIC statement[:BASIC statement...] <carriage return>
```

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A BASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of:

```
     72 characters in 8K BASIC-80
    255 characters in Extended and Disk BASIC-80.
```

In Extended and Disk versions, it is possible to extend a logical line over more than one physical line by use of the terminal's <line feed> key. <Line feed> lets you continue typing a logical line on the next physical line without entering a <carriage return>. (In the 8K version, <line feed> has no effect.

### 1.3.1 Line Numbers

Every BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range 0 to 65529. In the Extended and Disk versions, a period (.) may be used in EDIT, LIST, AUTO and DELETE commands to refer to the current line.

## 1.4 CHARACTER SET

The BASIC-80 character set is comprised of alphabetic characters, numeric characters and special characters.

The alphabetic characters in BASIC-80 are the upper case and lower case letters of the alphabet.

The numeric characters in BASIC-80 are the digits 0 through 9.

The following special characters and terminal keys are recognized by BASIC-80:

| Character | Name |
|---|---|
|  | Blank |
| ; | Semicolon |
| = | Equal sign or assignment symbol |
| + | Plus sign |
| - | Minus sign |
| * | Asterisk or multiplication symbol |
| / | Slash or division symbol |
| ^ | Up arrow or exponentiation symbol |
| ( | Left parenthesis |
| ) | Right parenthesis |
| % | Percent |
| # | Number (or pound) sign |
| $ | Dollar sign |
| ! | Exclamation point |
| [ | Left bracket |
| ] | Right bracket |
| , | Comma |
| . | Period or decimal point |
| ' | Single quotation mark (apostrophe) |
| : | Colon |
| & | Ampersand |
| ? | Question mark |
| < | Less than |
| > | Greater than |
| \ | Backslash or integer division symbol |
| @ | At-sign |
| _ | Underscore |
| <rubout> | Deletes last character typed. |
| <escape> | Escapes Edit Mode subcommands. See Section 2.16. |
| <tab> | Moves print position to next tab stop. Tab stops are every eight columns. |
| <line feed> | Moves to next physical line. |
| <carriage return> | Terminates input of a line. |

### 1.4.1 Control Characters

The following control characters are in BASIC-80:

Control-A Enters Edit Mode on the line being typed.

Control-C Interrupts program execution and returns to BASIC-80 command level.

Control-G Rings the bell at the terminal.

Control-H Backspace. Deletes the last character typed.

Control-I Tab. Tab stops are every eight columns.

Control-O Halts program output while execution continues. A second Control-O restarts output.

Control-R Retypes the line that is currently being typed.

Control-S Suspends program execution.

Control-Q Resumes program execution after a Control-S.

Control-U Deletes the line that is currently being typed.

## 1.5 CONSTANTS

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

1. Integer constants Whole numbers between -32768 and +32767. Integer constants do not have decimal points.

2. Fixed Point constants Positive or negative real numbers, i.e., numbers that contain decimal points.

3. Floating Point constants

Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The exponent must be in the range -38 to +38. Examples:

```
235.988E-7 = .0000235988
2359E6 = 2359000000
```

(Double precision floating point constants use the letter D instead of E. See Section 1.5.1.)

4. Hex constants

Hexadecimal numbers with the prefix &H. Examples:

```
     &H76
     &H32F
```

5. Octal constants

Octal numbers with the prefix &O or &. Examples:

```
     &O347
     &1234
```

### 1.5.1 Single And Double Precision Form For Numeric Constants

In the 8K version of BASIC-80, all numeric constants are single precision numbers. They are stored with 7 digits of precision, and printed with up to 6 digits.

In the Extended and Disk versions, however, numeric constants may be either single precision or double precision numbers. With double precision, the numbers are stored with 16 digits of precision, and printed with up to 16 digits.

11

A single precision constant is any numeric constant that has:

1. seven or fewer digits, or
2. exponential form using E, or
3. a trailing exclamation point (!)

A double precision constant is any numeric constant that has:

1. eight or more digits, or
2. exponential form using D, or
3. a trailing number sign (#)

Examples:

| Single Precision Constants | Double Precision Constants |
|---|---|
| 46.8 | 345692811 |
| -7.09E-06 | -1.09432D-06 |
| 3489.0 | 3489.0# |
| 22.5! | 7654321.1234 |

## 1.6 VARIABLES

Variables are names used to represent values that are used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

### 1.6.1 Variable Names And Declaration Characters

BASIC-80 variable names may be any length, however, in the 8K version, only the first two characters are significant. In the Extended and Disk versions, up to 40 characters are significant. The characters allowed in a variable name are letters and numbers, and the decimal point is allowed in Extended and Disk variable names. The first character must be a letter. Special type declaration characters are also allowed -- see below.

A variable name may not be a reserved word. The Extended and Disk versions allow embedded reserved words; the 8K version does not. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all BASIC-80 commands, statements, function names and operator names.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign ($) as the last character. For example: A$ = "SALES REPORT". The dollar sign is a variable type declaration character, that is, it "declares" that the variable will represent a string.

In the Extended and Disk versions, numeric variable names may declare integer, single or double precision values. (All numeric values in 8K are single precision.) The type declaration characters for these variable names are as follows:

| | |
|---|---|
| % | Integer variable |
| ! | Single precision variable |
| # | Double precision variable |

The default type for a numeric variable name is single precision.

Examples of BASIC-80 variable names follow.

In Extended and Disk versions:

| | |
|---|---|
| PI# | declares a double precision value |
| MINIMUM! | declares a single precision value |
| LIMIT% | declares an integer value |

In 8K, Extended and Disk versions:

| | |
|---|---|
| N$ | declares a string value |
| ABC | represents a single precision value |

11

In the Extended and Disk versions of BASIC-80, there is a second method by which variable types may be declared. The BASIC-80 statements DEFINT, DEFSTR, DEFSNG and DEFDBL may be inclcded in a program to declare the types for certain variable names. These statements are described in detail in Section 2.12.

### 1.6.2 Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with integers or integer expressions. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimensional array, T(1,4) would reference a value in a two-dimensional array, and so on.

## 1.7 TYPE CONVERSION

When necessary, BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)
   Example:

   ```
   10 A% = 23.42
   20 PRINT A%
   RUN
    23
   ```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.
   Examples:

   ```
   10 D# = 6#/7       The arithmetic was performed
   20 PRINT D#        in double precision and the
   RUN                result was returned in D#
    .8571428571428571 as a double precision value.

   10 D = 6#/7        The arithmetic was performed
   20 PRINT D         in double precision and the
   RUN                result was returned to D (single
    .857143           precision variable), rounded and
                      printed as a single precision
                      value.
   ```

3. Logical operators (see Section 1.8.3) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs.

4. When a floating point value is converted to an integer, the fractional portion is rounded.
   Example:

   ```
   10 C% = 55.88
   20 PRINT C%
   RUN
    56
   ```

5. If a double precision variable is assigned a single precision value, only the first seven digits, rounded, of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E-8 times the original single precision value.
   Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
 2.04  2.039999961853027
```

## 1.8 EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC-80 may be divided into four categories:

1. Arithmetic

2. Relational

3. Logical

4. Functional

11

### 1.8.1 Arithmetic Operators

The arithmetic operators, in order of precedence, are:

| Operator | Operation | Sample Expression |
|---|---|---|
| ∧ | Exponentiation | X∧Y |
| - | Negation | -X |
| *,/ | Multiplication, Floating<br>Point Division | X*Y<br>X/Y |
| +,- | Addition, Subtraction | X+Y |

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

| Algebraic Expression | BASIC Expression |
|---|---|
| $X+2Y$ | X+Y*2 |
| $X-\frac{Y}{Z}$ | X-Y/Z |
| $\frac{XY}{Z}$ | X*Y/Z |
| $\frac{X+Y}{Z}$ | (X+Y)/Z |
| $(X^2)^Y$ | (X∧2)∧Y |
| $X^{Y^Z}$ | X∧(Y∧Z) |
| $X(-Y)$ | X*(-Y) Two consecutive operators must be separated by parentheses. |

### 1.8.1.1 Integer Division And Modulus Arithmetic -

Two additional operators are available in Extended and Disk versions of BASIC-80: Integer division and modulus arithmetic.

Integer division is denoted by the baskslash (\). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer. For example:

```
10\4 = 2
25.68\6.99 = 3
```

The precedence of integer division is just after multiplication and floating point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

```
10.4 MOD 4 = 2 (10/4=2 with a remainder 2)
25.68 MOD 6.99 = 5 (26/7=3 with a remainder 5)
```

The precedence of modulus arithmetic is just after integer division.

### 1.8.1.2 Overflow And Division By Zero -

If, during the evaluation of an expression, a division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

### 1.8.2 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then used to make a decision regarding program flow. (See IF, Section 2.26.)

| Operator | Relation Tested | Expression |
|---|---|---|
| = | Equality | X=Y |
| <> | Inequality | X<>Y |
| < | Less than | X<Y |
| > | Greater than | X>Y |
| <= | Less than or equal to | X<=Y |
| >= | Greater than or equal to | X>=Y |

(The equal sign is also used to assign a value to a variable. See LET, Section 2.30.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X+Y < (T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z. More examples:

```
IF SIN(X)<0 GOTO 1000
IF I MOD J <> 0 THEN K=K+1
```

### 1.8.3 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

| X | NOT X |
|---|---|
| 1 | 0 |
| 0 | 1 |

AND

| X | Y | X AND Y |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

OR

| X | Y | X OR Y |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

XOR

| X | Y | X XOR Y |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

IMP

| X | Y | X IMP Y |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

EQV

| X | Y | X EQV Y |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF, Section 2.26). For example:

```
     IF D<200 AND F<4 THEN 80
     IF I>10 OR K<0 THEN 50
     IF NOT P THEN 100
```

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator maybe used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

| | |
|---|---|
| 63 AND 16=16 | 63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16 |
| 15 AND 14=14 | 15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110) |
| -1 AND 8=8 | -1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8 |
| 4 OR 2=6 | 4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110) |
| 10 OR 10=10 | 10 = binary 1010, so 1010 OR 1010 = 1010 (10) |
| -1 OR -2=-1 | -1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1. |
| NOT X=-(X+1) | The two's complement of any integer is the bit complement plus one. |

### 1.8.4 Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. BASIC-80 has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC-80's intrinsic functions are described in Chapter 3.

BASIC-80 also allows "user defined" functions that are written by the programmer. See DEF FN, Section 2.11.

### 1.8.5 String Operations

Strings may be concatenated using +. For example:

```
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=     <>     <     >     <=     >=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"     where B$ = "8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

## 1.9 INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the RUBOUT key or with Control-H. Rubout surrounds the deleted character(s) with backslashes, and Control-H has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type Control-U. A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. BASIC-80 will automatically replace the old line with the new line.

More sophisticated editing capabilities are provided in the Extended and Disk versions of BASIC-80. See EDIT, Section 2.16.

To delete the entire program that is currently residing in memory, enter the NEW command. (See Section 2.41.) NEW is usually used to clear memory prior to entering a new program.

## 1.10 ERROR MESSAGES

If BASIC-80 detects an error that causes program execution to terminate, an error message is printed. In the 8K version, only the error code is printed. In the Extended and Disk versions, the entire error message is printed. For a complete list of BASIC-80 error codes and error messages, see Appendix J.

# CHAPTER 2

# BASIC-80 COMMANDS AND STATEMENTS

All of the BASIC-80 commands and statements are described in this chapter. Each description is formatted as follows:

Format: Shows the correct format for the instruction. See below for format notation.

Versions: Lists the versions of BASIC-80 in which the instruction is available.

Purpose: Tells what the instruction is used for.

Remarks: Describes in detail how the instruction is used.

Example: Shows sample programs or program segments that demonstrate the use of the instruction.

## Format Notation

Wherever the format for a statement or command is given, the following rules apply:

1. Items in capital letters must be input as shown.

2. Items in lower case letters enclosed in angle brackets (< >) are to be supplied by the user.

3. Items in square brackets ([ ]) are optional.

4. All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.

5. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).

## 2.1 AUTO

Format: AUTO [<line number>[,<increment>]]

Versions: Extended, Disk

Purpose: To generate a line number automatically after every carriage return.

Remarks: AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing Control-C. The line in which Control-C is typed is not saved. After Control-C is typed, BASIC returns to command level.

Example:

| | |
|---|---|
| AUTO 100,50 | Generates line numbers 100, 150, 200 ... |
| AUTO | Generates line numbers 10, 20, 30, 40 ... |

## 2.2 CALL

Format: CALL <variable name>[(<argument list>)]

Version: Extended, Disk

Purpose: To call an assembly language subroutine.

Remarks: The CALL statement is one way to transfer program flow to an assembly language subroutine. (See also the USR function, Section 3.40)

<variable name> contains an address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name. <argument list> contains the arguments that are passed to the assembly language subroutine.

The CALL statement generates the same calling sequence used by Microsoft's FORTRAN, COBOL and BASIC compilers.

Example:

```
110 MYROUT=&HD000
120 CALL MYROUT(I,J,K)
      .
      .
      .
```

## 2.3 CHAIN

Format: CHAIN [MERGE] <filename>[,[<line number exp>] [,ALL][,DELETE<range>]]

Version: Disk

Purpose: To call a program and pass variables to it from the current program.

Remarks: <filename> is the name of the program that is called. Example:

```
CHAIN"PROG1"
```

<line number exp> is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. Example:

```
CHAIN"PROG1",1000
```

<line number exp> is not affected by a RENUM command.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed. See Section 2.7. Example:

```
CHAIN"PROG1",1000,ALL
```

If the MERGE option is included, it allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd. Example:

```
CHAIN MERGE"OVRLAY",1000
```

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option. Example:

```
CHAIN MERGE"OVRLAY2",1000,DELETE 1000-5000
```

The line numbers in <range> are affected by the RENUM command.

NOTE: The Microsoft BASIC compiler does not support the ALL, MERGE, and DELETE options to CHAIN. If you wish to maintain compatibility with the BASIC compiler, it is recommended that COMMON be used to pass variables and that overlays not be used.

11

## 2.4 CLEAR

Format: CLEAR[,[<expression1>][,<expression2>]]

Versions: 8K, Extended, Disk

Purpose: To set all numeric variables to zero and all string variables to null; and, optionally, to set the end of memory and the amount of stack space.

Remarks: <expression1> is a memory location which, if specified, sets the highest location available for use by BASIC-80.

<expression2> sets aside stack space for BASIC. The default is 1000 bytes or one-eighth of the available memory, whichever is smaller.

NOTE: In previous versions of BASIC-80, <expression1> set the amount of string space and <expression2> set the end of memory. BASIC-80, release 5.0 and later, allocates string space dynamically. An "Out of string space" error occurs only if there is no free memory left for BASIC to use.

Examples:

```
CLEAR

CLEAR ,32768

CLEAR,,2000

CLEAR,32768,2000
```

## 2.5 CLOAD

Formats: CLOAD <filename>

CLOAD? <filename>

CLOAD* <array name>

Versions: 8K (cassette), Extended (cassette)

Purpose: To load a program or an array from cassette tape into memory.

Remarks: CLOAD executes a NEW command before it loads the program from cassette tape. <filename> is the string expression or the first character of the string expression that was specified when the program was CSAVEd.

CLOAD? verifies tapes by comparing the program currently in memory with the file on tape that has the same filename. If they are the same, BASIC-80 prints Ok. If not, BASIC-80 prints NO GOOD.

CLOAD* loads a numeric array that has been saved on tape. The data on tape is loaded into the array called <array name> specified when the array was CSAVE*ed.

CLOAD and CLOAD? are always entered at command level as direct mode commands. CLOAD* may be entered at command level or used as a program statement. Make sure the array has been DIMensioned before it is loaded. BASIC-80 always returns to command level after a CLOAD, CLOAD? or CLOAD* is executed. Before a CLOAD is executed, make sure the cassette recorder is properly connected and in the Play mode, and the tape is possitioned correctly.

See also CSAVE, Section 2.9.

NOTE: CLOAD and CSAVE are not included in all implementations of BASIC-80.

Example:

```
CLOAD "MAX2"
```

Loads file "M" into memory.

11

## 2.6 CLOSE

Format: CLOSE[[#]<file number>[,[#]<file number...>]]

Version: Disk

Purpose: To conclude I/O to a disk file.

Remarks: <file number> is the number under which the file was OPENed. A CLOSE with no arguments closes all open files.

The association between a particular file and file number terminates upon execution of a CLOSE. The file may then be reOPENed using the same or a different file number; likewise, that file number may now be reused to OPEN any file.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always CLOSE all disk files automatically. (STOP does not close disk files.)

Example: See Appendix B.

## 2.7 COMMON

Format: COMMON <list of variables>

Version: Disk

Purpose: To pass variables to a CHAINed program.

Remarks: The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
       .
       .
       .
```

## 2.8 CONT

Format: CONT

Versions: 8K, Extended, Disk

Purpose: To continue program execution after a Control-C has been typed, or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (? or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. With the Extended and Disk versions, CONT may be used to continue execution after an error.

CONT is invalid if the program has been edited during the break. In 8K BASIC-80, execution cannot be CONTinued if a direct mode error has occurred during the break.

Example: See example Section 2.61, STOP.

## 2.9 CSAVE

Formats: CSAVE <string expression>

CSAVE* <array variable name>

Versions: 8K (cassette), Extended (cassette)

Purpose: To save the program or an array currently in memory on cassette tape.

Remarks: Each program or array saved on tape is identified by a filename. When the command CSAVE <string expression> is executed, BASIC-80 saves the program currently in memory on tape and uses the first character in <string expression> as the filename. <string expression> may be more than one character, but only the first character is used for the filename.

When the command CSAVE* <array variable name> is executed, BASIC-80 saves the specified array on tape. The array must be a numeric array. The elements of a multidimensional array are saved with the leftmost subscript changing fastest.

CSAVE may be used as a program statement or as a direct mode command.

Before a CSAVE or CSAVE* is executed, make sure the cassette recorder is properly connected and in the Record mode.

See also CLOAD, Section 2.5.

NOTE: CSAVE and CLOAD are not included in all implementations of BASIC-80.

Example:

```
CSAVE "TIMER"
```

Saves the program currently in memory on cassette under filename "T".

## 2.10 DATA

Format: DATA <list of constants>

Versions: 8K, Extended, Disk

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s). (See READ, Section 2.54)

Remarks: DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement (Section 2.57).

Example: See examples in Section 2.54, READ.

## 2.11 DEF FN

Format: DEF FN<name>[(<parameter list>)]=<function definition>

Versions: 8K, Extended, Disk

Purpose: To define and name a function that is written by theuser.

Remarks: <name> must be a legal variable name. This name, preceded by FN, becomes the name of the function. <parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas. <function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call. (Remember, in the 8K version only one argument is allowed in a function call, therefore the DEF FN statement will contain only one variable.)

In Extended and Disk BASIC-80, user-defined functions may be numeric or string; in 8K, user-defined string functions are not allowed. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example:

```
          .
          .
410 DEF FNAB(X,Y)=X∧3/Y∧2
420 T=FNAB(I,J)
          .
          .
```

Line 410 defines the function FNAB. The function is called in line 420.

## 2.12 DEFINT/SNG/DBL/STR

Format: DEF<type> <range of letters>
where <type> is INT, SNG, DBL, or STR

Versions: Extended, Disk

Purpose: To declare variable types as integer, single precision, double precision, or string.

Remarks: A DEFtype statement declares that the variable names beginning with the letter(s) specified will be that type variable. However, a type declaration character always takes precedence over a DEFtype statement in the typing of a variable.

If no type declaration statements are encountered, BASIC-80 assumes all variables without declaration characters are single precision variables.

Examples:

| | |
|---|---|
| 10 DEFDBL L-P | All variables beginning with the letters L, M, N, O, and P will be double precision variables. |
| 10 DEFSTR A | All variables beginning with the letter A will be string variables. |

## 2.13 DEF USR

Format: DEF USR[<digit>]=<integer expression>

Versions: Extended, Disk

Purpose: To specify the starting address of an assembly language subroutine.

Remarks: <digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine. See Appendix C, Assembly Language Subroutines.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example:

```
.
.
.
200 DEF USR0=24000
210 X=USR0(Y∧2/2.89)
.
.
.
```

## 2.14 DELETE

Format: DELETE[<line number>][-<line number>]

Versions: Extended, Disk

Purpose: To delete program lines.

Remarks: BASIC-80 always returns to command level after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

Examples:

| | |
|---|---|
| DELETE 40 | Deletes line 40 |
| DELETE 40-100 | Deletes lines 40 through 100, inclusive |
| DELETE-40 | Deletes all lines up to and including line 40 |

11

## 2.15 DIM

Format: DIM <list of subscripted variables>

Versions: 8K, Extended, Disk

Purpose: To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see Section 2.46).

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example:

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
     .
     .
     .
```

## 2.16 EDIT

Format: EDIT <line number>

Versions: Extended, Disk

Purpose: To enter Edit Mode at the specified line.

Remarks: In Edit Mode, it is possible to edit portions of a line without retyping the entire line. Upon entering Edit Mode, BASIC-80 types the line number of the line to be edited, then it types a space and waits for an Edit Mode subcommand.

### Edit Mode Subcommands

Edit Mode subcommands are used to move the cursor or to insert, delete, replace, or search for text within a line. The subcommands are not echoed. Most of the Edit Mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When a preceding integer is not specified, it is assumed to be 1.

Edit Mode subcommands may be categorized according to the following functions:

1. Moving the cursor

2. Inserting text

3. Deleting text

4. Finding text

5. Replacing text

6. Ending and restarting Edit Mode

11

NOTE

In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and $ represents the Escape (or Altmode) key.

1. Moving the Cursor

Space Use the space bar to move the cursor to the right. [i]Space moves the cursor i spaces to the right. Characters are printed as you space over them.

Rubout In Edit Mode, [i]Rubout moves the cursor i spaces to the left (backspaces). Characters are printed as you backspace over them.

2. Inserting Text

I I<text>$ inserts <text> at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, type Escape. If Carriage Return is typed during an Insert command, the effect is the same as typing Escape and then Carriage Return. During an Insert command, the Rubout or Delete key on the terminal may be used to delete characters to the left of the cursor. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell (Control-G) is typed and the character is not printed.

X The X subcommand is used to extend the line. X moves the cursor to the end of the line, goes into insert mode, and allows insertion of text as if an Insert command had been given. When you are finished extending the line, type Escape or Carriage Return.

3. Deleting Text

D [i]D deletes i characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, iD deletes the remainder of the line.

H H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for replacing statements at the end of a line.

4. Finding Text

S The subcommand [i]S<ch> searches for the ith occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor will stop at the end of the line. All characters passed over during the search are printed.

K The subcommand [i]K<ch> is similar to [i]S<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

5. Replacing Text

C The subcommand C<ch> changes the next character to <ch>. If you wish to change the next i characters, use the subcommand iC, followed by i characters. After the ith new character is typed, change mode is exited and you will return to Edit Mode.

6. Ending and Restarting Edit Mode

<cr> Typing Carriage Return prints the remainder of the line, saves the changes you made and exits Edit Mode.

E The E subcommand has the same effect as Carriage Return, except the remainder of the line is not printed.

Q The Q subcommand returns to BASIC-80 command level, *without* saving any of the changes that were made to the line during Edit Mode.

L The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in Edit Mode. L is usually used to list the line when you first enter Edit Mode.

A The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.

NOTE

> If BASIC-80 receives an unrecognizable command or illegal character while in Edit Mode, it prints a bell (Control-G) and the command or character is ignored.

### Syntax Errors

When a Syntax Error is encountered during execution of a program, BASIC-80 automatically enters Edit Mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and type Carriage Return (or the E subcommand), BASIC-80 reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination, first exit Edit Mode with the Q subcommand. BASIC-80 will return to command level, and all variable values will be preserved.

### Control-A

To enter Edit Mode on the line you are currently typing, type Control-A. BASIC-80 responds with a carriage return, an exclamation point (!) and a space. The cursor will be positioned at the first character in the line. Proceed by typing an Edit Mode subcommand.

NOTE

Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT." will enter Edit Mode at the current line. (The line number symbol "." always refers to the current line.)

## 2.17 END

Format: END

Versions: 8K, Extended, Disk

Purpose: To terminate program execution, close all files and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. BASIC-80 always returns to command level after an END is executed.

Example:

```
520 IF K>1000 THEN END ELSE GOTO 20
```

11

## 2.18 ERASE

Format: ERASE <list of array variables>

Versions: 8K, Extended, Disk

Purpose: To eliminate arrays from a program.

Remarks: Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Redimensioned array" error occurs.

NOTE: The Microsoft BASIC compiler does not support ERASE.

Example:

```
 .
 .
 .
450 ERASE A,B
460 DIM B(99)
 .
 .
 .
```

## 2.19 ERR AND ERL VARIABLES

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use IF 65535 = ERL THEN ...
Otherwise, use

```
IF ERR = error code THEN ...

IF ERL = line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. BASIC-80's error codes are listed in Appendix J. (For Standalone Disk BASIC error codes, see Appendix H.)

## 2.20 ERROR

Format: ERROR <integer expression>

Versions: Extended, Disk

Purpose: 1) To simulate the occurrence of a BASIC-80 error; or 2) to allow error codes to be defined by the user.

Remarks: The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC-80 (see Appendix J), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by BASIC-80's error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to BASIC-80.) This user-defined error code may then be conveniently handled in an error trap routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, BASIC-80 responds with the message UNPRINTABLE ERROR. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example 1:

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

Or, in direct mode:

```
Ok
ERROR 15              (you type this line)
String too long       (BASIC-80 types this line)
Ok
```

```
Example 2:      .
                .
                .
               110 ON ERROR GOTO 400
               120 INPUT "WHAT IS YOUR BET";B
               130 IF B > 5000 THEN ERROR 210
                .
                .
                .
               400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
               410 IF ERL = 130 THEN RESUME 120
                .
                .
                .
```

## 2.21 FIELD

Format: FIELD[#]<file number>,<field width> AS <string variable>...

Version: Disk

Purpose: To allocate space for variables in a random file buffer.

Remarks: To get data out of a random buffer after a GET or to enter data before a PUT, a FIELD statement must have been executed.

<file number> is the number under which the file was OPENed. <field width> is the number of characters to be allocated to <string variable>. For example,

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does NOT place any data in the random file buffer. (See LSET/RSET and GET.)

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 128.)

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

Example: See Appendix B.

NOTE: *Do not use a FIELDed variable name in an INPUT or LET statement.* Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

## 2.22 FOR...NEXT

Format:

```
FOR <variable>=x TO y [STEP z]
  .
  .
  .
NEXT [<variable>][,<variable>...]
```

where x, y and z are numeric expressions.

Versions: 8K, Extended, Disk

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: <variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if thevalue of the counter is now greater than the final value (y). If it is not greater, BASIC-80 branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

Nested Loops

FOR...NEXT loops may be nested, that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be

11

omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1:

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
 1  20
 3  30
 5  40
 7  50
 9  60
Ok
```

Example 2:

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3:

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1  2  3  4  5  6  7  8  9  10
Ok
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set. (Note: Previous versions of BASIC-80 set the initial value of the loop variable before setting the final value; i.e., the above loop would have executed six times.)

## 2.23 GET

Format: GET [#]<file number>[,<record number>]

Version: Disk

Purpose: To read a record from a random disk file into a random buffer.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

Example: See Appendix B.

## 2.24 GOSUB...RETURN

Format:

```
GOSUB <line number>
    .
    .
    .
RETURN
```

Versions: 8K, Extended, Disk

Purpose: To branch to and return from a subroutine.

Remarks: <line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC-80 to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertant entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

Example:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

## 2.25 GOTO

Format: GOTO <line number>

Versions: 8K, Extended, Disk

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Remarks: If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example:

```
LIST
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 4          AREA = 78.5
R = 7          AREA = 153.86
R = 12         AREA = 452.16
?Out of data in 10
Ok
```

11

## 2.26 IF...THEN[...ELSE] AND IF...GOTO

Format: IF <expression> THEN <statement(s)> | <line number>

[ELSE <statement(s)> | <line number>]

Format: IF <expression> GOTO <line number>

[ELSE <statement(s)> | <line number>]

Versions: 8K, Extended, Disk

NOTE: The ELSE clause is allowed only in Extended and Disk versions.

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. (ELSE is allowed only in Extended and Disk versions.) Extended and Disk versions allow a comma before THEN.

### Nesting of IF Statements

In the Extended and Disk versions, IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
     THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example

```
IF A=B THEN IF B=C THEN PRINT "A=C"
     ELSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error results unless a statement with the specified line number had previously been entered in the indirect mode.

NOTE: When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1:

```
200 IF I THEN GET#1,I
```

This statement GETs record number I if I is not zero.

Example 2:

```
100 IF(I<20)*(I>10) THEN DB=1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
        .
        .
        .
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3:

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer, otherwise output goes to the terminal.

11

## 2.27 INPUT

Format: INPUT[;][<"prompt string">;]<list of variables>

Versions: 8K, Extended, Disk

Purpose: To allow input from the terminal during program execution.

Remarks: When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to input data does not echo a carriage return/line feed sequence.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the messsage "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

In the 8K version, INPUT is illegal in the direct mode.

Examples:

```
10 INPUT X
20 PRINT X "SQUARED IS" X∧2
30 END
RUN
?  5      (The 5 was typed in by the user
           in response to the question mark.)
 5 SQUARED IS 25
Ok


LIST
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R∧2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS?  7.4  (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?
etc.
```

## 2.28 INPUT#

Format: INPUT#<file number>,<variable list>

Version: Disk

Purpose: To read data items from a sequential disk file and assign them to program variables.

Remarks: <file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma.

If BASIC-80 is scanning the sequential data file for a string item, leading spaces, carriage returns and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

Example: See Appendix B.

## 2.29 KILL

Format: KILL <filename>

Version: Disk

Purpose: To delete a file from disk.

Remarks: If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs.

KILL is used for all types of disk files: program files, random data files and sequential data files.

Example:

```
200 KILL "DATA1"
```

See also Appendix B.

## 2.30 LET

Format: [LET] <variable>=<expression>

Versions: 8K, Extended, Disk

Purpose: To assign the value of an expression to a variable.

Remarks: Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
      .
      .
      .

    or

110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
      .
      .
      .
```

## 2.31 LINE INPUT

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

Versions: Extended, Disk

Purpose: To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks: The prompt string is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to <string variable>.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT may be escaped by typing Control-C. BASIC-80 will return to command level and type Ok. Typing CONT resumes execution at the LINE INPUT.

Example: See Example, Section 2.32, LINE INPUT#.

## 2.32 LINE INPUT#

Format: LINE INPUT#<file number>,<string variable>

Version: Disk

Purpose: To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Remarks: <file number> is the number under which the file was OPENed. <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC-80 program saved in ASCII mode is being read as data by another program.

Example:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES    234,4    MEMPHIS
LINDA JONES    234,4    MEMPHIS
Ok
```

## 2.33 LIST

Format 1: LIST [<line number>]

Versions: 8K, Extended, Disk

Format 2: LIST [<line number>[-[<line number>]]]

Versions: Extended, Disk

Purpose: To list all or part of the program currently in memory at the terminal.

Remarks: BASIC-80 always returns to command level after a LIST is executed.

Format 1: If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program or by typing Control-C.) If <line number> is included, the 8K version will list the program beginning at that line; and the Extended and Disk versions will list only the specified line.

Format 2: This format allows the following options:

1. If only the first number is specified, that line and all higher-numbered lines are listed.
2. If only the second number is specified, all lines from the beginning of the program through that line are listed.
3. If both numbers are specified, the entire range is listed.

11

Examples: Format 1:

| | |
|---|---|
| LIST | Lists the program currently in memory. |
| LIST 500 | In the 8K version, lists all programs lines from 500 to the end.<br>In Extended and Disk, lists line 500. |

Format 2:

| | |
|---|---|
| LIST 150- | Lists all lines from 150 to the end. |
| LIST -1000 | Lists all lines from the lowest number through 1000. |
| LIST 150-1000 | Lists lines 150 through 1000, inclusive. |

## 2.34 LLIST

Format: LLIST [<line number>[-[<line number>]]]

Versions: Extended, Disk

Purpose: To list all or part of the program currently in memory at the line printer.

Remarks: LLIST assumes a 132-character wide printer.

BASIC-80 always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST, Format 2.

NOTE: LLIST and LPRINT are not included in all implementations of BASIC-80.

Example: See the examples for LIST, Format 2.

## 2.35 LOAD

Format: LOAD <filename>[,R]

Version: Disk

Purpose: To load a file from disk into memory.

Remarks: <filename> is the name that was used when the file was SAVEd. (With CP/M, the default extension .BAS is supplied.)

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the "R" option is used with LOAD, the program is RUN after it is LOADed, and all open data files are kept open. Thus, LOAD with the "R" option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

Example:

```
LOAD "STRTRK",R
```

## 2.36 LPRINT AND LPRINT USING

Format: LPRINT [<list of expressions>]

LPRINT USING <"format string">;<list of expressions>

Versions: Extended, Disk

Purpose: To print data at the line printer.

Remarks: Same as PRINT and PRINT USING, except output goes to the line printer. See Section 2.49 and Section 2.50.

LPRINT assumes a 132-character-wide printer.

NOTE: LPRINT and LLIST are not included in all implementations of BASIC-80.

## 2.37 LSET AND RSET

Format:

```
LSET <string variable> = <string expression>
RSET <string variable> = <string expression>
```

Version: Disk

Purpose: To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks: If <string expression> requires fewer bytes than were FIELDed to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See the MKI$, MKS$, MKD$ functions, Section 3.25.

Examples:

```
150 LSET A$=MKS$(AMT)
160 LSET D$=DESC($)
```

See also Appendix B.

NOTE: LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

## 2.38 MERGE

Format: MERGE <filename>

Version: Disk

Purpose: To merge a specified disk file into the program currently in memory.

Remarks: <filename> is the name used when the file was SAVEd. (With CP/M, the default extension .BAS is supplied.) The file must have been SAVEd in ASCII format. (If not, a "Bad file mode" error occurs.)

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on disk into the program in memory.)

BASIC-80 always returns to command level after executing a MERGE command.

Example: MERGE "NUMBRS"

11

## 2.39 MID$

Format: MID$(<string exp1>,n[,m])=<string exp2>

where n and m are integer expressions and <string exp1> and <string exp2> are string expressions.

Versions: Extended, Disk

Purpose: To replace a portion of one string with another string.

Remarks: The characters in <string exp1>, beginning at position n, are replaced by the characters in <string exp2>. The optional m refers to the number of characters from <string exp2> that will be used in the replacement. If m is omitted, all of <string exp2> is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of <string exp1>.

Example:

```
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
RUN
KANSAS CITY, KS
```

MID$ may also be used as a function that returns a substring of a given string. See Section 3.24.

## 2.40 NAME

Format: NAME <old filename> AS <new filename>

Version: Disk

Purpose: To change the name of a disk file.

Remarks: <old filename> must exist and <new filename> must not exist; otherwise an error will result. After a NAME command, the file exists on the same disk, in the same area of disk space, with the newname.

Example:

```
Ok
NAME "ACCTS" AS "LEDGER"
Ok
```

In this example, the file that was formerly named ACCTS will now be named LEDGER.

## 2.41 NEW

Format: NEW

Versions: 8K, Extended, Disk

Purpose: To delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. BASIC-80 always returns to command level after a NEW is executed.

## 2.42 NULL

Format: NULL <integer expression>

Versions: 8K, Extended, Disk

Purpose: To set the number of nulls to be printed at the end of each line.

Remarks: For 10-character-per-second tape punches, <integer expression> should be >=3. When tapes are not being punched, <integer expression> should be 0 or 1 for Teletypes and Teletype-compatible CRTs. <integer expression> should be 2 or 3 for 30 cps hard copy printers. The default value is 0.

Example:

```
Ok
NULL 2
Ok
100 INPUT X
200 IF X<50 GOTO 800
       .
       .
       .
```

Two null characters will be printed after each line.

11

## 2.43 ON ERROR GOTO

Format: ON ERROR GOTO <line number>

Versions: Extended, Disk

Purpose: To enable error trapping and specify the first line of the error handling subroutine.

Remarks: Once error trapping has been enabled all errors detected, including direct mode errors (e.g., Syntax errors), will cause a jump to the specified error handling subroutine. If <line number> does not exist, an "Undefined line" error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC-80 to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

NOTE: If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

Example:

```
10 ON ERROR GOTO 1000
```

## 2.44 ON...GOSUB AND ON...GOTO

Format: ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

Versions: 8K, Extended, Disk

Purpose: To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks: The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is negative, zero or greater than the number of items in the list, an "Illegal function call" error occurs.

Example:

```
100 ON L-1 GOTO 150,300,320,390
```

## 2.45 OPEN

Format: OPEN <mode>,[#]<file number>,<filename>,[<reclen>]

Version: Disk

Purpose: To allow I/O to a disk file.

Remarks: A disk file must be OPENed before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

<mode> is a string expression whose first character is one of the following:

| | |
|---|---|
| O | specifies sequential output mode |
| I | specifies sequential input mode |
| R | specifies random input/output mode |

<file number> is an integer expression whose value is between one and fifteen. The number is then associated with the file for as long as it is OPEN and is used to refer other disk I/O statements to the file.

<filename> is a string expression containing a name that conforms to your operating system's rules for disk filenames.

<reclen> is an integer expression which, if included, sets the record length for random files. The default record length is 128 bytes. See also page A-3.

NOTE: A file can be OPENed for sequential input or random access on more than one file number at a time. A file may be OPENed for output, however, on only one file number at a time.

Example:

```
10 OPEN "I",2,"INVEN"
```

See also Appendix B.

## 2.46 OPTION BASE

Format: OPTION BASE n
where n is 1 or 0

Versions: Extended, Disk

Purpose: To declare the minimum value for array subscripts.

Remarks: The default base is 0. If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is one.

## 2.47 OUT

Format: OUT I,J
where I and J are integer expressions in the range 0 to 255.

Versions: 8K, Extended, Disk

Purpose: To send a byte to a machine output port.

Remarks: The integer expression I is the port number, and the integer expression J is the data to be transmitted.

Example:

```
100 OUT 32,100
```

## 2.48 POKE

Format: POKE I,J
where I and J are integer expressions

Versions: 8K, Extended, Disk

Purpose: To write a byte into a memory location.

Remarks: The integer expression I is the address of the memory location to be POKEd. The integer expression J is the data to be POKEd. J must be in the range 0 to 255. In the 8K version, I must be less than 32768. In the Extended and Disk versions, I must be in the range 0 to 65536.

With the 8K version, data may be POKEd into memory locations above 32768 by supplying a negative number for I. The value of I is computed by subtracting 65536 from the desired address. For example, to POKE data into location 45000, I = 45000-65536, or -20536.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See Section 3.27.

POKE and PEEK are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

Example:

```
10 POKE &H5A00,&HFF
```

11

## 2.49 PRINT

Format: PRINT [<list of expressions>]

Versions: 8K, Extended, Disk

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

### Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC-80 divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, BASIC-80 goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 10∧(-6) is output as .000001 and 10∧(-7) is output as 1E-7. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 10∧(-16) is output as .0000000000000001 and 10∧(-17) is output as 1D-17.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1:

```
10 X=5
20 PRINT X+5, X-5, X*(-5), X∧5
30 END
RUN
 10            0             -25            3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2:

```
LIST
10 INPUT X
20 PRINT X "SQUARED IS" X∧2 "AND";
30 PRINT X "CUBED IS" X∧3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

Example 3:

```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

## 2.50 PRINT USING

Format: PRINT USING <"format string">;<list of expressions>

Versions: Extended, Disk

Purpose: To print strings or numbers using a specified format.

Remarks and Examples: <list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. <"format string">, enclosed in quotation marks, is comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

### String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!" Specifies that only the first character in the given string is to be printed.

"\n spaces\" Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right. Example:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\  \";A$;B$
50 PRINT USING "\   \";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT  !!
```

"&" Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input. Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```

## Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

# A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

. A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##;".78
 0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##   ";10.2,5.3,66.789,.234
10.20   5.30   66.79   0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+ A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

-	A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##   ";-68.95,2.4,55.6,-.9
-68.95   +2.40   +55.60   -0.90

PRINT USING "##.##-   ";-68.95,22.449,-7.01
68.95-   22.45    7.01-
```

**	A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "**#.#   ";12.39,-0.9,765.1
*12.4   *-0.9   765.1
```

$$	A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
 $456.78
```

**$	The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

,	A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^^^^) format.

```
PRINT USING "####,.##";1234.5
1,234.50

PRINT USING "####.##,";1234.5
1234.50,
```

^^^^

Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##^^^^";234.56
 2.35E+02
```

```
PRINT USING ".####^^^^-";888888
 .8889E+06
```

```
PRINT USING "+.##^^^^";123
+.12E+03
```

_

An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

The literal character itself may be an underscore by placing "__" in the format string.

%

If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22
```

```
PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

11

## 2.51 PRINT# AND PRINT# USING

Format: PRINT#<filenumber>,[USING<"format string">;]<list of exps>

Version: Disk

Purpose: To write data to a sequential disk file.

Remarks: <filenumber> is the number used when the file was OPENed for output. <"format string"> is comprised of formatting characters as described in Section 2.50, PRINT USING. The expressions in <list of expressions> are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it will be input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement

```
PRINT#1,A$;B$
```

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;",";B$
```

The image written to disk is

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$="   93604-1". The statement

```
PRINT#1,A$;B$
```

would write the following image to disk:

```
CAMERA, AUTOMATIC   93604-1
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "AUTOMATIC   93604-1" to B$. To separate these strings properly on the disk, write double quotes to the disk image using CHR$(34). The statement

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to disk:

```
"CAMERA, AUTOMATIC""   93604-1"
```

and the statement

```
INPUT#1,A$,B$
```

11

would input "CAMERA, AUTOMATIC" to A$ and "   93604-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$###.##,";J;K;L
```

For more examples using PRINT#, see Appendix B.

See also WRITE#, Section 2.68.

## 2.52 PUT

Format: PUT [#]<file number>[,<record number>]

Version: Disk

Purpose: To write a record from a random buffer to a random disk file.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the record will have the next available record number (after the last PUT). The largest possible record number is 32767.

Example: See Appendix B.

## 2.53 RANDOMIZE

Format: RANDOMIZE [<expression>]

Versions: Extended, Disk

Purpose: To reseed the random number generator.

Remarks: If <expression> is omitted, BASIC-80 suspends program execution and asks for a value by printing

```
     Random Number Seed (0-65529)?
```

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (0-65529)? 3  (user types 3)
 .88598  .484668  .586328  .119426  .709225
Ok
RUN
Random Number Seed (0-65529)? 4 (user types 4
for new sequence)
 .803506  .162462  .929364  .292443  .322921
Ok
RUN
Random Number Seed (0-65529)? 3 (same sequence
as first RUN)
 .88598  .484668  .586328  .119426  .709225
Ok
```

## 2.54 READ

Format: READ <list of variables>

Versions: 8K, Extended, Disk

Purpose: To read values from a DATA statement and assign them to variables. (See DATA, Section 2.10.)

Remarks: A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statment. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see RESTORE, Section 2.57)

Example 1:

```
 .
 .
 .
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
 .
 .
 .
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2:

```
LIST
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY         STATE         ZIP
DENVER,      COLORADO      80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

11

## 2.55 REM

Format: REM <remark>

Versions: 8K, Extended, Disk

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

In the Extended and Disk versions, remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM.

Example:

```
     .
     .
     .
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
     .
     .
     .
```

or, with Extended and Disk versions:

```
     .
     .
     .
120 FOR I=1 TO 20     'CALCULATE AVERAGE VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
     .
     .
     .
```

## 2.56 RENUM

Format: RENUM [[<new number>][,[<old number>][,<increment>]]]

Versions: Extended, Disk

Purpose: To renumber program lines.

Remarks: <new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

NOTE: RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

Examples:

| | |
|---|---|
| RENUM | Renumbers the entire program. The first new line number will be 10. Lines will increment by 10. |
| RENUM 300,,50 | Renumbers the entire program. The first new line number will be 300. Lines will increment by 50. |
| RENUM 1000,900,20 | Renumbers the lines from 900 up so they start with line number 1000 and increment by 20. |

11

## 2.57 RESTORE

Format: RESTORE [<line number>]

Versions: 8K, Extended, Disk

Purpose: To allow DATA statements to be reread from a specified point.

Remarks: After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
      .
      .
      .
```

## 2.58 RESUME

Formats: RESUME

RESUME 0

RESUME NEXT

RESUME <line number>

Versions: Extended, Disk

Purpose: To continue program execution after an error recovery procedure has been performed.

Remarks: Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME
or
RESUME 0 Execution resumes at the statement which caused the error.

RESUME NEXT Execution resumes at the statement immediately following the one which caused the error.

RESUME <line number> Execution resumes at <line number>.

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to be printed.

Example:

```
10 ON ERROR GOTO 900
   .
   .
   .
900 IF (ERR=230)AND(ERL=90) THEN PRINT "TRY
AGAIN":RESUME 80
   .
   .
   .
```

11

## 2.59 RUN

Format 1: RUN [<line number>]

Versions: 8K Extended, Disk

Purpose: To execute the program currently in memory.

Remarks: If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. BASIC-80 always returns to command level after a RUN is executed.

Example: RUN

Format 2: RUN <filename>[,R]

Version: Disk

Purpose: To load a file from disk into memory and run it.

Remarks: <filename> is the name used when the file was SAVEd. (With CP/M and ISIS-II, the default extension .BAS is supplied.)

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain OPEN.

Example: RUN "NEWFIL",R

See also Appendix B.

## 2.60 SAVE

Format: SAVE <filename>[,A | ,P]

Version: Disk

Purpose: To save a program file on disk.

Remarks: <filename> is a quoted string that conforms to your operating system's requirements for filenames. (With CP/M, the default extension .BAS is supplied.) If <filename> already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

Examples:

```
SAVE"COM2",A
SAVE"PROG",P
```

See also Appendix B.

11

## 2.61 STOP

Format: STOP

Versions: 8K, Extended, Disk

Purpose: To terminate program execution and return to command level.

Remarks: STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

```
Break in line nnnnn
```

Unlike the END statement, the STOP statement does not close files.

BASIC-80 always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see Section 2.8).

Example:

```
10 INPUT A,B,C
20 K=A∧2*5.3:L=B∧3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```

## 2.62 SWAP

Format: SWAP <variable>,<variable>

Versions: Extended, Disk

Purpose: To exchange the values of two variables.

Remarks: Any type variable may be SWAPped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example:

```
LIST
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
 ONE FOR ALL
 ALL FOR ONE
Ok
```

11

## 2.63 TRON/TROFF

Format: TRON

TROFF

Versions: Extended, Disk

Purpose: To trace the execution of program statements.

Remarks: As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example:

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINTJ;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
[10][20][30][40] 1  10  20
[50][60][30][40] 2  20  30
[50][60][70]
Ok
TROFF
Ok
```

## 2.64 WAIT

Format: WAIT <port number>, I[,J]
where I and J are integer expressions

Versions: 8K, Extended, Disk

Purpose: To suspend program execution while monitoring the status of a machine input port.

Remarks: The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, BASIC-80 loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero.

CAUTION: It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to manually restart the machine.

Example:

```
100 WAIT 32,2
```

11

## 2.65 WHILE...WEND

Format:

```
WHILE <expression>
   .
   .
[<loop statements>]
   .
   .
WEND
```

Versions: Extended, Disk

Purpose: To execute a series of statements in a loop as long as a given condition is true.

Remarks: If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example:

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115       FLIPS=0
120       FOR I=1 TO J-1
130               IF A$(I)>A$(I+1) THEN
                          SWAP A$(I),A$(I+1):FLIPS=1
140       NEXT I
150 WEND
```

## 2.66 WIDTH

Format: WIDTH [LPRINT] <integer expression>

Versions: Extended, Disk

Purpose: To set the printed line width in number of characters for the terminal or line printer.

Remarks: If the LPRINT option is omitted, the line width is set at the terminal. If LPRINT is included, the line width is set at the line printer.

<integer expression> must have a value in the range 15 to 255. The default width is 72 characters.

If <integer expression> is 255, the line width is "infinite," that is, BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

## 2.67 WRITE

Format: WRITE[<list of expressions>]

Version: Disk

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of the expressions are output at the terminal. The expressions in the list may be numeric and/or string expressions, and they must be separated by commas.

When the printed items are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement, Section 2.49.

Example:

```
10 A=80:B=90:C$=THAT'S ALL
20 WRITE A,B,C$
RUN
 80, 90,"THAT'S ALL"
Ok
```

## 2.68 WRITE#

Format: WRITE#<file number>,<list of expressions>

Version: Disk

Purpose: To write data to a sequential file.

Remarks: <file number> is the number under which the file was OPENed in "O" mode. The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

Example: Let A$="CAMERA" and B$="93604-1". The statement:

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.

11

# CHAPTER 3

# BASIC-80 FUNCTIONS

The intrinsic functions provided by BASIC-80 are presented in this chapter. The functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

| | |
|---|---|
| X and Y | Represent any numeric expressions |
| I and J | Represent integer expressions |
| X$ and Y$ | Represent string expressions |

If a floating point value is supplied where an integer is required, BASIC-80 will round the fractional portion and use the resulting integer.

## 3.1 ABS

Format: ABS(X)

Versions: 8K, Extended, Disk

Action: Returns the absolute value of the expression X.

Example:

```
PRINT ABS(7*(-5))
 35
Ok
```

## 3.2 ASC

Format: ASC(X$)

Versions: 8K, Extended, Disk

Action: Returns a numerical value that is the ASCII code of the first character of the string X$. (See Appendix L for ASCII codes.) If X$ is null, an "Illegal function call" error is returned.

Example:

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

See the CHR$ function for ASCII-to-string conversion.

## 3.3 ATN

Format: ATN(X)

Versions: 8K, Extended, Disk

Action: Returns the arctangent of X in radians. Result is in the range -pi/2 to pi/2. The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example:

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.24905
Ok
```

## 3.4 CDBL

Format: CDBL(X)

Versions: Extended, Disk

Action: Converts X to a double precision number.

Example:

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67  454.6700134277344
Ok
```

11

## 3.5 CHR$

Format: CHR$(I)

Versions: 8K, Extended, Disk

Action: Returns a string whose one element has ASCII code I. (ASCII codes are listed in Appendix L.) CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent (CHR$(7)) as a preface to an error message, or a form feed could be sent (CHR$(12)) to clear a CRT screen and return the cursor to the home position.

Example:

```
PRINT CHR$(66)
B
Ok
```

See the ASC function for ASCII-to-numeric conversion.

## 3.6 CINT

Format: CINT(X)

Versions: Extended, Disk

Action: Converts X to an integer by rounding the fractional portion. If X is not in the range -32768 to 32767, an "Overflow" error occurs.

Example:

```
PRINT CINT(45.67)
 46
Ok
```

See the CDBL and CSNG functions for converting numbers to the double precision and single precision data type. See also the FIX and INT functions, both of which return integers.

## 3.7 COS

Format: COS(X)

Versions: 8K, Extended, Disk

Action: Returns the cosine of X in radians. The calculation of COS(X) is performed in single precision.

Example:

```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.84212
Ok
```

## 3.8 CSNG

Format: CSNG(X)

Versions: Extended, Disk

Action: Converts X to a single precision number.

Example:

```
10 A# = 975.3421#
20 PRINT A#; CSNG(A#)
RUN
 975.3421  975.342
Ok
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types.

## 3.9 CVI, CVS, CVD

Format: CVI(<2-byte string>)
CVS(<4-byte string>)
CVD(<8-byte string>)

Version: Disk

Action: Convert string values to numeric values. Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Example:

```
 .
 .
 .
70 FIELD #1,4 AS N$, 12 AS B$, ...
80 GET #1
90 Y=CVS(N$)
 .
 .
 .
```

See also MKI$, MKS$, MKD$, Section 3.25 and Appendix B.

## 3.10 EOF

Format: EOF(<file number>)

Version: Disk

Action: Returns -1 (true) if the end of a sequential file has been reached. Use EOF to test for end-of-file while INPUTting, to avoid "Input past end" errors.

Example:

```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
 .
 .
 .
```

## 3.11 EXP

Format: EXP(X)

Versions: 8K, Extended, Disk

Action: Returns e to the power of X. X must be <=87.3365. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:

```
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.5982
Ok
```

## 3.12 FIX

Format: FIX(X)

Versions: Extended, Disk

Action: Returns the truncated integer part of X. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

Examples:

```
PRINT FIX(58.75)
 58
Ok

PRINT FIX(-58.75)
-58
Ok
```

11

## 3.13 FRE

Format:

```
FRE(0)
FRE(X$)
```

Versions: 8K, Extended, Disk

Action: Arguments to FRE are dummy arguments. If the argument is 0 (numeric), FRE returns the number of bytes in memory not being used by BASIC-80. If the argument is a string, FRE returns the number of free bytes in string space.

Example:

```
PRINT FRE(0)
 14542
Ok
```

## 3.14 HEX$

Format:

```
HEX$(X)
```

Versions: Extended, Disk

Action: Returns a string which represents the hexadecimal value of the decimal argument. X is rounded to an integer before HEX$(X) is evaluated.

Example:

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

See the OCT$ function for octal conversion.

## 3.15 INP

Format: INP(I)

Versions: 8K, Extended, Disk

Action: Returns the byte read from port I. I must be in the range 0 to 255. INP is the complementary function to the OUT statement, Section 2.47.

Example:

```
100 A=INP(255)
```

## 3.16 INPUT$

Format: INPUT$(X[,[#]Y])

Version: Disk

Action: Returns a string of X characters, read from the terminal or from file number Y. If the terminal is used for input, no characters will be echoed and all control characters are passed through except Control-C, which is used to interrupt the execution of the INPUT$ function.

Example 1:

```
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE IN
HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

11

Example 2:

```
   .
   .
   .
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
   .
   .
   .
```

## 3.17 INSTR

Format: INSTR([I,]X$,Y$)

Versions: Extended, Disk

Action: Searches for the first occurrence of string Y$ in X$ and returns the position at which the match is found. Optional offset I sets the position for starting the search. I must be in therange 0 to 255. If I>LEN(X$) or if X$ is null or if Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions or string literals.

Example:

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2  6
Ok
```

## 3.18 INT

Format: INT(X)

Versions: 8K, Extended, Disk

Action: Returns the largest integer <=X.

Examples:

```
PRINT INT(99.89)
 99
Ok

PRINT INT(-12.11)
-13
Ok
```

See the FIX and CINT functions which also return integer values.

## 3.19 LEFT$

Format: LEFT$(X$,I)

Versions: 8K, Extended, Disk

Action: Returns a string comprised of the leftmost I characters of X$. I must be in the range 0 to 255. If I is greater than LEN(X$), the entire string (X$) will be returned. If I=0, the null string (length zero) is returned.

Example:

```
10 A$ = "BASIC-80"
20 B$ = LEFT$(A$,5)
30 PRINT B$
BASIC
Ok
```

Also see the MID$ and RIGHT$ functions.

## 3.20 LEN

Format: LEN(X$)

Versions: 8K, Extended, Disk

Action: Returns the number of characters in X$. Non-printing characters and blanks are counted.

Example:

```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
 16
Ok
```

11

## 3.21 LOC

Format: LOC(<file number>)

Version: Disk

Action: With random disk files, LOC returns the next record number to be used if a GET or PUT (without a record number) is executed. With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was OPENed.

Example:

```
200 IF LOC(1)>50 THEN STOP
```

## 3.22 LOG

Format: LOG(X)

Versions: 8K, Extended, Disk

Action: Returns the natural logarithm of X. X must be greater than zero.

Example:

```
PRINT LOG(45/7)
 1.86075
Ok
```

## 3.23 LPOS

Format: LPOS(X)

Versions: Extended, Disk

Action: Returns the current position of the line printer print head within the line printer buffer. Does not necessarily give the physical position of the print head. X is a dummy argument.

Example:

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

## 3.24 MID$

Format: MID$(X$,I[,J])

Versions: 8K, Extended, Disk

Action: Returns a string of length J characters from X$ beginning with the Ith character. I and J must be in the range 0 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(X$), MID$ returns a null string.

Example:

```
LIST
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
Ok
RUN
GOOD EVENING
Ok
```

Also see the LEFT$ and RIGHT$ functions.

11

## 3.25 MKI$, MKS$, MKD$

Format:

```
MKI$(<integer expression>)
MKS$(<single precision expression>)
MKD$(<double precision expression>)
```

Version: Disk

Action: Convert numeric values to string values. Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

Example:

```
90 AMT=(K+T)
100 FIELD #1, 8 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
  .
  .
```

See also CVI, CVS, CVD, Section 3.9 and Appendix B.

## 3.26 OCT$

Format: OCT$(X)

Versions: 8K, Extended, Disk

Action: Returns a string which represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

Example:

```
PRINT OCT$(24)
 30
Ok
```

See the HEX$ function for hexadecimal conversion.

## 3.27 PEEK

Format: PEEK(I)

Versions: 8K, Extended, Disk

Action: Returns the byte (decimal integer in the range 0 to 255) read from memory location I. With the 8K version of BASIC-80, I must be less than 32768. To PEEK at a memory location above 32768, subtract 65536 from the desired address. With Extended and Disk BASIC-80, I must be in the range 0 to 65536. PEEK is the complementary function to the POKE statement, Section 2.48.

Example: A=PEEK(&H5A00)

## 3.28 POS

Format: POS(I)

Versions: 8K, Extended, Disk

Action: Returns the current cursor position. The leftmost position is 0. X is a dummy argument.

Example: IF POS(X)>60 THEN PRINT CHR$(13)

Also see the LPOS function.

11

## 3.29 RIGHT$

Format: RIGHT$(X$,I)

Versions: 8K, Extended, Disk

Action: Returns the rightmost I characters of string X$. If I=LEN(X$), returns X$. If I=0, the null string (length zero) is returned.

Example:

```
10 A$="DISK BASIC-80"
20 PRINT RIGHT$(A$,8)
RUN
BASIC-80
Ok
```

Also see the MID$ and LEFT$ functions.

## 3.30 RND

Format: RND[(X)]

Versions: 8K, Extended, Disk

Action: Returns a random number between 0 and 1. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (see RANDOMIZE, Section 2.53). However, X<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

Example:

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 24  30  31  51  5
Ok
```

## 3.31 SGN

Format: SGN(X)

Versions: 8K, Extended, Disk

Action: If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns -1.

Example: ON SGN(X)+2 GOTO 100,200,300 branches to 100 if X is negative, 200 if X is 0 and 300 if X is positive.

## 3.32 SIN

Format: SIN(X)

Versions: 8K, Extended, Disk

Action: Returns the sine of X in radians. SIN(X) is calculated in single precision. COS(X)=SIN(X+3.14159/2).

Example:

```
PRINT SIN(1.5)
 .997495
Ok
```

11

## 3.33 SPACE$

Format: SPACE$(X)

Versions: 8K, Extended, Disk

Action: Returns a string of spaces of length X. The expression X is rounded to an integer and must be in the range 0 to 255.

Example:

```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
  1
   2
    3
     4
      5
Ok
```

Also see the SPC function.

## 3.34 SPC

Format: SPC(I)

Versions: 8K, Extended, Disk

Action: Prints I blanks on the terminal. SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255.

Example:

```
PRINT "OVER" SPC(15) "THERE"
OVER               THERE
Ok
```

Also see the SPACE$ function.

## 3.35 SQR

Format: SQR(X)

Versions: 8K, Extended, Disk

Action: Returns the square root of X. X must be >=0.

Example:

```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10            3.16228
 15            3.87298
 20            4.47214
 25            5
Ok
```

## 3.36 STR$

Format: STR$(X)

Versions: 8K, Extended, Disk

Action: Returns a string representation of the value of X.

Example:

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
    .
    .
    .
```

Also see the VAL function.

11

## 3.37 STRING$

Formats: STRING$(I,J)
STRING$(I,X$)

Versions: Extended, Disk

Action: Returns a string of length I whose characters all have ASCII code J or the first character of X$.

Example:

```
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

## 3.38 TAB

Format: TAB(I)

Versions: 8K, Extended, Disk

Action: Spaces to position I on the terminal. If the current print position is already beyond space I, TAB has no effect. Space 0 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 0 to 255. TAB may only be used in PRINT and LPRINT statements.

Example:

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
RUN
NAME                     AMOUNT

G. T. JONES              $25.00
Ok
```

## 3.39 TAN

Format: TAN(X)

Versions: 8K, Extended, Disk

Action: Returns the tangent of X in radians. TAN(X) is calculated in single precision. If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:

```
10 Y = Q*TAN(X)/2
```

## 3.40 USR

Format: USR[<digit>](X)

Versions: 8K, Extended, Disk

Action: Calls the user's assembly language subroutine with the argument X. <digit> is allowed in the Extended and Disk versions only. <digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. See Appendix C.

Example:

```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
     .
     .
     .
```

11

## 3.41 VAL

Format: VAL(X$)

Versions: 8K, Extended, Disk

Action: Returns the numerical value of string X$. If the first character of X$ is not +, -, &, or a digit, VAL(X$)=0.

Example:

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN
PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815 THEN
PRINT NAME$ TAB(25) "LONG BEACH"
   .
   .
   .
```

See the STR$ function for numeric to string conversion.

## 3.42 VARPTR

Format 1: VARPTR(<variable name>)

Versions: Extended, Disk

Format 2: VARPTR(#<file number>)

Version: Disk

Action: Format 1: Returns the address of the first byte of data identified with <variable name>. A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise an "Illegal function call" error results. Any type variable name may be used (numeric, string, array), and the address returned will be an integer in the range 32767 to -32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

NOTE: All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

Format 2: Returns the starting address of the disk I/O buffer assigned to <file number>.

In Standalone Disk BASIC, VARPTR(#<file number>) returns the first byte of the file block. See Appendix H.

Example:

```
100 X=USR(VARPTR(Y))
```

# APPENDIX A

## New Features in BASIC-80, Release 5.0

The execution of BASIC programs written under Microsoft BASIC, release 4.51 and earlier may be affected by some of the new features in release 5.0. Before attempting to run such programs, check for the following:

1. New reserved words: CALL, CHAIN, COMMON, WHILE, WEND, WRITE, OPTION BASE, RANDOMIZE.

2. Conversion from floating point to integer values results in rounding, as opposed to truncation. This affects not only assignment statements (e.g., I%=2.5 results in I%=3), but also affects function and statement evaluations (e.g., TAB(4.5) goes to the 5th position, A(1.5) yeilds A(2), and X=11.5 MOD 4 yields 0 for X).

3. The body of a FOR...NEXT loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step. See Section 2.22.

4. Division by zero and overflow no longer produce fatal errors. See Section 1.8.1.2.

5. The RND function has been changed so that RND with no argument is the same as RND with a positive argument. The RND function generates the same sequence of random numbers with each RUN, unless RANDOMIZE is used. See Sections 2.53 and 3.30.

6. The rules for PRINTing single precision and double precision numbers have been changed. See Section 2.49.

7. If the argument to ON...GOTO is out of range, an error message results and execution halts.

8. String space is allocated dynamically, and the first argument in a two-argument CLEAR statement will be ignored. See Section 2.4.

9. Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.), or with a carriage return causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

10. There are two new field formatting characters for use with PRINT USING. An ampersand is used for variable length string fields, and an underscore signifies a literal character in a format string.

11. If the expression supplied with the WIDTH statement is 255, BASIC uses an "infinite" line width, that is, it does not insert carriage returns. WIDTH LPRINT may be used to set the line width at the line printer. See Section 2.66.

12. The at-sign and underscore are no longer used as editing characters.

13. Variable names are significant up to 40 characters and can contain embedded reserved words. However, reserved words must now be delimited by spaces. To maintain compatibility with earlier versions of BASIC, spaces will be automatically inserted between adjoining reserved words and variable names. WARNING: This insertion of spaces may cause the end of a line to be truncated if the line length is close to 255 characters.

14. BASIC programs may be saved in a protected binary format. See SAVE, Section 2.60.

## CP/M and ISIS-II BASIC-80

In CP/M and ISIS-II BASIC-80, release 5.0, a number of additions have been made to disk I/O capability:

1. After a GET statement, INPUT# and LINE INPUT# may be done to read characters from the random file buffer. PRINT#, PRINT# USING, and WRITE# may also be used to put characters in the random file buffer before a PUT statement.

   In the case of WRITE#, BASIC-80 pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

2. /S:<max record size> may be added at the end of the command line to set the maximum record size for use with random files. The default record size is 128 bytes.

A new feature has been added to the INPUT statement. A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B$ will print the prompt with no question mark.

# APPENDIX B

## BASIC-80 Disk I/O

Disk I/O procedures for the beginning BASIC-80 user are examined in this appendix. If you are new to BASIC-80 or if you're getting disk related errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to your operating system's requirements for filenames. The CP/M operating system will append a default extension .BAS to the filename given in a SAVE, RUN, MERGE or LOAD command.

### B.1 PROGRAM FILE COMMANDS

Here is a review of the commands and statements used in program file manipulation.

SAVE "filename"[,A] Writes to disk the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)

LOAD "filename"[,R] Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and access the same data files.

11

RUN "filename"[,R]

RUN "filename" loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open.

MERGE "filename"

Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level.

KILL"filename"

Deletes the file from the disk. "filename" may be a program file, or a sequential or random access data file.

NAME

To change the name of a disk file, execute the NAME statement, NAME "oldfile" AS "newfile". NAME may be used with program files, random files, or sequential files.

## B.2 PROTECTED FILES

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
     SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited.

## B.3 DISK DATA FILES - SEQUENTIAL AND RANDOM I/O

There are two types of disk data files that may be created and accessed by a BASIC-80 program: sequential files and random access files.

### B.3.1 Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored, one item after another (sequentially), in the order it is sent and is read back in the same way.

The statements and functions that are used with sequential files are:

```
   OPEN    PRINT#         INPUT#        WRITE#
           PRINT# USING   LINE INPUT#

   CLOSE    EOF   LOC
```

The following program steps are required to create a sequential file and access the data in the file:

| | |
|---|---|
| 1. OPEN the file in "O" mode. | OPEN "O",#1,"DATA" |
| 2. Write data to the file using the PRINT# statement. (WRITE# maybe used instead.) | PRINT#1,A$;B$;C$ |
| 3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode. | CLOSE#1<br>OPEN "I",#1,"DATA" |
| 4. Use the INPUT# statement to read data from the sequential file into the program. | INPUT#1,X$,Y$,Z$ |

Program B-1 is a short program that creates a sequential file, "DATA", from information you input at the terminal.

11

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN


NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

PROGRAM B-1 - CREATE A SEQUENTIAL DATA FILE

Now look at Program B-2. It accesses the file "DATA" that was created in Program B-1 and displays the name of everyone hired in 1978.

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

PROGRAM B-2 - ACCESSING A SEQUENTIAL FILE

Program B-2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
     PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

11

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed. A sector is a 128-byte block of data.

B.3.1.1 <u>Adding Data To A Sequential File</u> -
If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. The following procedure can be used to add data to an existing file called "NAMES".

1. OPEN "NAMES" in "I" mode.
2. OPEN a second file called "COPY" in "O" mode.
3. Read in the data in "NAMES" and write it to "COPY".
4. CLOSE "NAMES" and KILL it.
5. Write the new information to "COPY".
6. Rename "COPY" as "NAMES" and CLOSE.
7. Now there is a file on disk called "NAMES" that includes all the previous data plus the new data you just added.

Program B-3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disk file. Remember, LINE INPUT# will read in characters from the disk until it sees a carriage return (it does not stop at quotes or commas) or until it has read 255 characters.

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

PROGRAM B-3 - ADDING DATA TO A SEQUENTIAL FILE

The error trapping routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

### B.3.2 Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk -- it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

```
OPEN    FIELD   LSET/RSET   GET

PUT     CLOSE   LOC

MKI$    CVI
MKS$    CVS
MKD$    CVD
```

### B.3.2.1 Creating A Random File -

The following program steps are required to create a random file.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes.If the record length is omitted, the default is 128 bytes.

   ```
   OPEN "R",#1,"FILE",32
   ```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

   ```
   FIELD #1 20 AS N$,
    4 AS A$, 8 AS P$
   ```

3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.

   ```
   LSET N$=X$
   LSET A$=MKS$(AMT)
   LSET P$=TEL$
   ```

4. Write the data from the buffer to the disk using the PUT statement.

   ```
   PUT #1,CODE%
   ```

Look at Program B-4. It takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

NOTE

Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

```
10 OPEN "R"#1,"FILE"
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

PROGRAM B-4 - CREATE A RANDOM FILE

B.3.2.2 <u>Access A Random File</u> -
The following program steps are required to access a random file:

```
1.  OPEN the file in "R" mode.     OPEN "R",#1,"FILE",32

2.  Use the FIELD statement to     FIELD #1 20 AS N$,
    allocate space in the random          4 AS A$, 8 AS P$
    buffer for the variables that
    will be read from the file.
```

NOTE:
In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer.

   ```
   GET #1,CODE%
   ```

4. The data in the buffer may now be acessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values.

   ```
   PRINT N$
   PRINT CVS(A$)
   ```

Program B-5 accesses the random file "FILE" that was created in Program B-4. By inputting the three-digit code at the terminal, the information associated with that code is read from the file and displayed.

```
10 OPEN "R",#1,"FILE"
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

PROGRAM B-5 - ACCESS A RANDOM FILE

The LOC function, with random files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file#1 is higher than 50.

Program B-6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

## PROGRAM B-6 - INVENTORY

```
120 OPEN "R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT "BAD FUNCTION NUMBER":GOTO
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;" IN STOCK":GOTO 600
630 Q%=Q%-S%
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;" REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
```

```
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY";CVI(Q$) TAB(50)
    "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER":GOTO 840
    ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

# APPENDIX C

## Assembly Language Subroutines

All versions of BASIC-80 have provisions for interfacing with assembly language subroutines. The USR Function allows assembly language subroutines to be called in the same way BASIC's intrinsic functions are called.

NOTE

> The addresses of the DEINT, GIVABF, MAKINT and FRCINT routines are stored in locations that must be supplied individually for different implementations of BASIC.

### C.1 MEMORY ALLOCATION

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s). BASIC uses all memory available from its starting location up, so only the topmost locations in memory can be set aside for user subroutines.

When an assembly language subroutine is called, the stack pointer is set up for 8 levels (16 bytes) of stack storage. If more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

11

The assembly language subroutine may be loaded into memory by means of the system monitor, or the BASIC POKE statement, or (if the user has the MACRO-80 or FORTRAN-80 package) routines may be assembled with MACRO-80 and loaded using LINK-80.

## C.2 USR FUNCTION CALLS - 8K BASIC

The starting address of the assembly language subroutine must be stored in USRLOC, a two-byte location in memory that is supplied individually with different implementations of BASIC-80. With 8K BASIC, the starting address may be POKEd into USRLOC. Store the low order byte first, followed by the high order byte.

The function USR will call the routine whose address is in USRLOC. Initially USRLOC contains the address of ILLFUN, the routine that gives the "Illegal function call" error. Therefore, if USR is called without changing the address in USRLOC, an "Illegal function call" error results.

The format of a USR function call is

```
     USR(argument)
```

where the argument is a numeric expression. To obtain the argument, the assembly language subroutine must call the routine DEINT. DEINT places the argument into the D,E register pair as a 2-byte, 2's complement integer. (If the argument is not in the range -32768 to 32767, an "Illegal function call" error occurs.)

To pass the result back from an assembly language subroutine, load the value in register pair [A,B], and call the routine GIVABF. If GIVABF is not called, USR(X) returns X. To return to BASIC, the assembly language subroutine must execute a RET instruction.

For example, here is an assembly language subroutine that multiplies the argument by 2:

```
USRSUB: CALL DEINT      ;put arg in D,E
        XCHG            ;move arg to H,L
        DAD H           ;H,L=H,L+H,L
        MOV A,H         ;move result to A,B
        MOV B,L
        JMP GIVABF      ;pass result back and RETurn
```

Note that valid results will be obtained from this routine for arguments in the range -16384<=x<=16383. The single instruction JMP GIVABF has the same effect as:

```
        CALL GIVABF
        RET
```

To return additional values to the program, load them into memory and read them with the PEEK function.

There are several methods by which a program may call more than one USR routine. For example, the starting address of each routine may be POKEd into USRLOC prior to each USR call, or the argument to USR could be an index into a table of USR routines.

## C.3 USR FUNCTION CALLS - EXTENDED AND DISK BASIC

In the Extended and Disk versions, the format of the USR function is

```
     USR[<digit>](argument)
```

where <digit> is from 0 to 9 and the argument is any numeric or string expression. <digit> specifies which USR routine is being called, and corresponds with the digit supplied in the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. The address given in the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the type of argument that was given. The value in A may be one of the following:

| Value in A | Type of Argument |
|---|---|
| 2 | Two-byte integer (two's complement) |
| 3 | String |
| 4 | Single precision floating point number |
| 8 | Double precision floating point number |

11

If the argument is a number, the [H,L] register pair points to the Floating Point Accumulator (FAC) where the argument is stored.

If the argument is an integer:

FAC-3 contains the lower 8 bits of the argument and
FAC-2 contains the upper 8 bits of the argument.

If the argument is a single precision floating point number:

FAC-3 contains the lowest 8 bits of mantissa and

FAC-2 contains the middle 8 bits of mantissa and FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative). FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

If the argument is a double precision floating point number:

FAC-7 through FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the [D,E] register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

CAUTION: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. Example:

```
A$ = "BASIC-80"+""
```

This will copy the string literal into string space and will prevent alteration of program text during a subroutine call.

Usually, the value returned by a USR function is the same type (integer, string, single precision or double precision) as the argument that was passed to it. However, calling the MAKINT routine returns the integer in [H,L] as the value of the function, forcing the value returned by the function to be integer. To execute MAKINT, use the following sequence to return from the subroutine:

```
   PUSH     H      ;save value to be returned
   LHLD    xxx     ;get address of MAKINT routine
   XTHL            ;save return on stack and
                   ;get back [H,L]
   RET             ;return
```

Also, the argument of the function, regardless of its type, may be forced to an integer by calling the FRCINT routine to get the integer value of the argument in [H,L]. Execute the following routine:

```
   LXI      H      ;get address of subroutine
                   ;continuation
   PUSH     H      ;place on stack
   LHLD    xxx     ;get address of FRCINT
   PCHL
SUB1: . . . . .
```

## C.4 CALL STATEMENT

Extended and Disk BASIC-80 user function calls may also be made with the CALL statement. The calling sequence used is the same as that in Microsoft's FORTRAN, COBOL and BASIC compilers.

A CALL statement with no arguments generates a simple "CALL" instruction. The corresponding subroutine should return via a simple "RET." (CALL and RET are 8080 opcodes - see an 8080 reference manual for details.)

A subroutine CALL with arguments results in a somewhat more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine. That parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each,regardless of type.

The method of passing the parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).

2. If the number of parameters is greater than 3, they are passed as follows:

    1. Parameter 1 in HL.

    2. Parameter 2 in DE.

    3. Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

11

Note that, with this scheme, the subroutine must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for correct number or type of parameters.

If the subroutine expects more than 3 parameters, and needs to transfer them to a local data area, there is a system subroutine which will perform this transfer. This argument transfer routine is named $AT (located in the FORTRAN library, FORLIB.REL), and is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (i.e., the total number of arguments minus 2). The subroutine is responsible for saving the first two parameters before calling $AT. For example, if a subroutine expects 5 parameters, it should look like:

```
SUBR:   SHLD    P1      ;SAVE PARAMETER 1
        XCHG
        SHLD    P2      ;SAVE PARAMETER 2
        MVI     A,3     ;NO. OF PARAMETERS LEFT
        LXI     H,P3    ;POINTER TO LOCAL AREA
        CALL    $AT     ;TRANSFER THE OTHER 3 PARAMETERS
        .
        .
        .
        [Body of subroutine]
        .
        .
        .
        RET             ;RETURN TO CALLER
P1:     DS      2       ;SPACE FOR PARAMETER 1
P2:     DS      2       ;SPACE FOR PARAMETER 2
P3:     DS      6       ;SPACE FOR PARAMETERS 3-5
```

A listing of the argument transfer routine AT$ follows.

```
;        ARGUMENT TRANSFER
;[B,C]   POINTS TO 3RD PARAM.
;[H,L]   POINTS TO LOCAL STORAGE FOR PARAM 3
;[A]     CONTAINS THE # OF PARAMS TO XFER(TOTAL-2)


        ENTRY   $AT
$AT:    XCHG                    ;SAVE [H,L] IN [D,E]
        MOV     H,B
        MOV     L,C             ;[H,L] = PTR TO PARAMS
AT1:    MOV     C,M
        INX     H
        MOV     B,M
        INX     H               ;[B,C] = PARAM ADR
        XCHG                    ;[H,L] POINTS TO LOCAL STORAGE
        MOV     M,C
        INX     H
        MOV     M,B
        INX     H               ;STORE PARAM IN LOCAL AREA
        XCHG                    ;SINCE GOING BACK TO AT1
        DCR     A               ;TRANSFERRED ALL PARAMS?
        JNZ     AT1             ;NO, COPY MORE
        RET                     ;YES, RETURN
```

When accessing parameters in a subroutine, don't forget that they are pointers to the actual arguments passed.

NOTE

It is entirely up to the programmer to see to it that the arguments in the calling program match in number, type, and length with the parameters expected by the subroutine. This applies to BASIC subroutines, as well as those written in assembly language.

## C.5 INTERRUPTS

Assembly language subroutines can be written to handle interrupts. All interrupt handling routines should save the stack, register A-L and the PSW. Interrupts should always be re-enabled before returning from the subroutine, since an interrupt automatically disables all further interrupts once it is received. The user should be aware of which interrupt vectors are free in the particular version of BASIC that has been supplied. Note to CP/M users: in CP/M BASIC, all interrupt vectors are free.)

11

# APPENDIX D

## BASIC-80 with the CP/M Operating System

The CP/M version of BASIC-80 (MBASIC) is supplied on a standard size 3740 single density diskette. The name of the file is MBASIC.COM. (A 28K or larger CP/M system is recommended.)

To run MBASIC, bring up CP/M and type the following:

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

The system will reply:

```
     xxxx Bytes Free
     BASIC-80 Version 5.0
     (CP/M Version)
     Copyright 1978 (C) by Microsoft
     Created: dd-mmm-yy
     Ok
```

MBASIC is the same as Disk BASIC-80 as described in this manual, with the following exceptions:

### D.1 INITIALIZATION

11

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

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

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

If /F:<number of files> is present, it sets the number of disk data files that may be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes of memory. If the /F option is omitted, the number of files defaults to 3.

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

NOTE

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

Examples:

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

## D.2 DISK FILES

Disk filenames follow the normal CP/M naming conventions. All filenames may include A: or B: as the first two characters to specify a disk drive, otherwise the currently selected drive is assumed. A default extension of .BAS is used on LOAD, SAVE, MERGE and RUN <filename> commands if no "." appears in the filename and the filename is less than 9 characters long.

## D.3 FILES COMMAND

Format: `FILES[<filename>]`

Purpose: To print the names of files residing on the current disk.

Remarks: If <filename> is omitted, all the files on the currently selected drive will be listed. <filename> is a string formula which may contain question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the filename or extension will match any file or any extension.

Examples:

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

## D.4 RESET COMMAND

Format: `RESET`

Purpose: To close all disk files and write the directory information to a diskette before it is removed from a disk drive.

Remarks: Always execute a RESET command before removing a diskette from a disk drive. Otherwise, when the diskette is used again, it will not have the current directory information written on the directory track.

RESET closes all open files on all drives and writes the directory track to every diskette with open files.

11

## D.5 LOF FUNCTION

Format: LOF(<file number>)

Action: Returns the number of records present in the last extent read or written. If the file does not exceed one extent (128 records), then LOF returns the true length of the file.

Example:

```
110 IF NUM%>LOF(1) THEN PRINT "INVALID ENTRY"
```

## D.6 EOF

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

## D.7 MISCELLANEOUS

1. CSAVE and CLOAD are not implemented.

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

3. FRCINT is at 103 hex and MAKINT is at 105 hex. (Add 1000 hex for ADDS versions, 4000 for SBC CP/M versions.)

# APPENDIX E

## BASIC-80 with the ISIS-II Operating System

With ISIS-II, BASIC-80 is the same as described in this manual, with the following exceptions:

### E.1 INITIALIZATION

The initialization dialog has been replaced by a set of options which are placed after the MBASIC command to ISIS-II. The format of the command line is:

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

If <filename> is present, BASIC proceeds as if a RUN <filename> command were typed after initialization is complete. A default extension of .BAS is used if none is supplied.

If /F:<number of files> is present, it sets the number of disk data files that may be open at any one time during the execution of a BASIC program. The maximum is six and the default is three. The /M:<highest memory location> option sets the highest memory location that will be used by BASIC. Use this option to reserve memory locations above BASIC for assembly language subroutines.

At initialization, the system will reply:

```
xxxx Bytes Free
BASIC-80 Version x.x
(ISIS-II Version)
Copyright 1978 (C) by Microsoft
```

## E.2 LINE PRINTER I/O

To send output to the printer during execution of a BASIC program, open the line printer as if it were a disk file:

```
50 N=4
100 OPEN "O",N,":LP:"
 .
 .
 .
120 PRINT #N,A,B,C
```

Since BASIC buffers disk I/O, you may want to force buffers out by CLOSEing the printer channel.

To LIST a program on the line printer, use:

```
SAVE":LP:",A
```

## E.3 ATTRIB STATEMENT

In ISIS-II BASIC-80, the ATTRIB statement sets file attributes. The format of the statement is:

```
ATTRIB <filename string>,<attribute string>
```

The attribute string consists of F, W, S or I for the attribute, followed by a 1 to set the attribute or a 0 to reset.

Examples:

```
ATTRIB "INFO.DAT","W1"
ATTRIB "GHOST.BAS","I1"
ATTRIB ":F1:SYSFIL","W1F1S1I1"
ATTRIB A$,B$
```

## E.4 MISCELLANEOUS

Note these other differences for ISIS-II BASIC:

1. MAKINT is located at xxxxx hex, and GIVINT is located at xxxxx hex.

2. There is no FILES command in ISIS-II BASIC. Filenames do <u>not</u> default to .BAS on SAVEs, LOADs, and MERGEs.

# APPENDIX F

## BASIC-80 with the TEKDOS Operating System

The operation of BASIC-80 with the TEKDOS operating system is the same as described in this manual with the following exceptions:

1. At initialization, BASIC asks MEMORY SIZE? If you respond with a carriage return, BASIC will use all available memory. If you respond with a memory location (in decimal), BASIC will use memory only up to that location. This lets you reserve space at the top of memory for assembly language subroutines.

2. The number of disk files that may be open at one time defaults to 5.

3. LPRINT and LLIST are not implemented. Instead, open a file to the printer.

4. TEKDOS does not support random disk I/O. The corresponding BASIC-80 statements (PUT, GET, OPEN"R", etc.) are inoperable under TEKDOS.

5. Control-C works only once due to a bug in TEKDOS. If you interrupt a running program or a LIST command with Control-C, BASIC appears to be in "single statement" mode. To clear this condition, exit BASIC with a SYSTEM command and re-enter BASIC with an XEQ BASIC. Avoid using the AUTO command, since it requires a Control-C to return to BASIC command level.

# APPENDIX G

## BASIC-80 with the INTEL SBC and MDS Systems

### G.1 INITIALIZATION

The paper tape of BASIC-80 supplied for SBC and MDS systems is in Intel-compatible hex format. Use the monitor's R command to load the tape, then execute the G command to start BASIC-80. The command is:

```
     .G4000
```

BASIC will respond:

```
     Memory size?
```

If you want BASIC to use all available RAM, just type a carriage return. If you want to reserve space at the top of memory for machine language subroutines, enter the highest memory address (in decimal) that BASIC may use.

```
     Terminal Width?
```

(8K versions only) Respond with the number of characters for the output line width in PRINT statements. The default is 72 characters. (Extended versions use WIDTH command.)

```
     Want SIN-COS-TAN-ATN?
```

Type Y to retain these functions, type N to delete them, or type A to delete ATN only.

### G.2 SUBROUTINE ADDRESSES

In the 8K version of SBC and MDS BASIC-80, DEINT is located at 0043 hex and GIVABF is located at 0045 hex. USRLOC is at xxxx hex. In the Extended version, FRCINT is located at xxxx hex, and MAKINT is located at xxxx hex.

## G.3 LLIST AND LLPRINT

LLIST and LPRINT are not implemented.

# APPENDIX H

## Standalone Disk BASIC

Standalone Disk BASIC is an easily implemented, self-contained version of BASIC-80 that runs on almost any 8080 or Z80 based disk hardware without an operating system. Standalone Disk BASIC incorporates several unique disk I/O methods that make faster and more efficient use of disk access and storage.

Random access with Standalone BASIC is faster than other disk operating systems because the file allocation table is kept in memory and updated periodically on the diskette. Therefore, there is no need for index blocks for random files, and there is no need to distinguish between random and sequential files. Because there are no index blocks, there is no large per-file-overhead either in memory or on disk. Binary SAVEs and LOADs are also faster because they are optimized by cluster, i.e., an entire cluster is read or written at one time, instead of a single sector.

To initialize Standalone Disk BASIC, insert the BASIC diskette and power up the system. In one- or two-drive systems, BASIC asks if there are two drives. In systems with more than two drives, BASIC asks for the number of drives. BASIC then asks how many files, i.e., how many disk files may be open at one time. Answer with a number from 0 to 15, or, for a default of 1 file per drive, just enter a carriage return.

The operation of Standalone Disk BASIC is the same as Disk BASIC-80 as described in this manual, with the following exceptions:

### H.1 FILENAMES

Disk filenames are six characters with an optional three-character extension that is preceded by a decimal point. If a decimal point appears in a filename after fewer than six characters, the name is blank-filled to six characters and the next three characters are the extension.

If the filename is six or fewer characters with no decimal point, there is no extension. If the filename is more than six characters, BASIC inserts a decimal point after the sixth character and uses the next three characters as an extension. (Any additional characters are ignored.)

## H.2 DISK FILES

The FILES command prints the names of the files residing on a disk. The format is: [L]FILES[<drive number>]

LFILES outputs to the line printer. In addition to the filename, the size of each file, in clusters, is output. A cluster is the minimum unit of allocation for a file -- it is one-half of a track. Filenames of files created with OPEN or ASCII SAVE are listed with a space between the name and extension. Filenames of binary files created with binary SAVE are listed with a decimal point between the name and extension. The protected file option with SAVE is not supported in Standalone Disk BASIC.

## H.3 FPOS

The FPOS function:

```
     FPOS(<file number>)
```

is the same as BASIC-80's LOC function except it returns the number of the physical sector where <filenumber> is located. (BASIC-80's LOC function and CP/M BASIC-80's LOF function are also implemented.)

## H.4 DSKI$/DSKO$

The DSKO$ statement:

```
     DSKO$<drive>,<track>,<sector>,<string expression>
```

writes the string on the specified sector. The maximum length for the string is 128 characters. A string of fewer than 128 characters is zero-filled at the end to 128 characters.

DSKI$ is the complementary function to the DSKO$ statement. DSKI$ returns the contents of a sector to a string variable name. The format is:

```
     DSKI$(<drive>,<track>,<sector>)
```

Example: A$=DSKI$(0,I,J)

## H.5 MOUNT COMMAND

Before a diskette can be used for file operations (i.e., any disk I/O besides DSKI$, DSKO$, or IBM or USR modes), it must be MOUNTed. The format of the command is:

```
     MOUNT[<drive>[,<drive>...]]
```

MOUNT with no arguments mounts all drives. When a diskette is mounted, BASIC reads the File Allocation Table (see Section H.11.2) from the diskette into memory and checks it for errors. If there are no errors, the disk is mounted. If an error is found, BASIC reads one or both of the back-up allocation tables from the diskette in an attempt to mount the disk; and a warning message, "x copies of allocation bad on drive y", is issued. x is 1 or 2 and y is the drive number. When a warning occurs, it is a good idea to make a new copy of the diskette. If all copies of the allocation table are bad or if a free entry is encountered in the file chain, a fatal error--"Bad allocation table"--is given and the diskette will not be mounted.

While a disk is mounted, BASIC occasionally writes the allocation table to the directory track, but it does not check for errors unless the read after write attribute is set for that drive (see SET statement).

## H.6 REMOVE COMMAND

REMOVE is the complement of MOUNT. Before a diskette can be taken out of the drive, a REMOVE command must be executed. The format of the command is:

```
     REMOVE[<drive>[,<drive>...]]
```

REMOVE writes three copies of the current allocation table to disk and follows the same error-check procedure as MOUNT. MOUNT and REMOVE replace the RESET command that is in BASIC-80.

NOTE

ALWAYS do a REMOVE before taking a diskette out of a drive. If you do not, the diskette you took out will not have an updated and checked allocation table, and the data on the next diskette inserted will be destroyed when the wrong allocation table is written to the directory track.

## H.7 SET STATEMENT

The SET statement determines the attributes of the currently mounted disk drive, a currently open file, or a file that need not be open. The format of the SET statement is:

```
SET<drive> | #<file> | <filename>,<attribute string>
```

<attribute string> is a string of characters that determines what attributes are set. Any characters other than the following are ignored:

| | |
|---|---|
| R | Read after write |
| P | Write protect |
| E | EBCDIC conversion (if available) |

Attributes are assigned in the following order:

1. MOUNT command
   When a MOUNT is done for a particular drive, the first byte of the information sector on the diskette (track 35, sector 20 for floppy; track 18, sector 13 for minifloppy) contains the attributes for the disk. (octal values: R=100, P=20, E=40)

2. SET<drive>,<attribute string> Statement
   This statement sets the current attributes for the disk, in memory, while it is mounted. The attributes are not permanently recorded and apply only while the disk is mounted.

3. When a file is created, the permanent file attributes recorded on the disk will be the same as the current drive attributes.

4. SET<filename>,<attribute string> Statement
   This statement changes the permanent file attributes that are stored in the directory entry for that file. It does not affect the drive attributes.

5. When an existing file is OPENed, the attributes of the file number are those of the directory entry.

6. SET#<file number>,<attribute string> Statement
   This statement changes the attributes for that file number but does not change the directory entry.

Examples:

| | |
|---|---|
| SET 1,"R" | Force read after write checking on all output to drive 1 |
| SET #1,"R" | Force read after write for all output to |

| | |
|---|---|
| | file 1 while it is open |
| SET #1,"P" | Give write protect error if any output is attempted to file 1 |
| SET "TEST","P" | Protect TEST from deletion and modification |
| SET 1,"" | Turn off all attributes for drive 1 |

## H.8 ATTR$ FUNCTION

ATTR$ returns a string of the current attributes for a drive, currently open file, or file that need not be open. The format of ATTR$ is:

```
ATTR$(<drive>   #<file number>   <filename>)
```

For example:

```
SET 1,"R":A$=ATTR$(1):PRINT A$
R
Ok
```

## H.9 OPEN STATEMENT

The format for the OPEN statement in Standalone BASIC is:

```
OPEN <filename> [FOR <mode>] AS [#]<file number>
```

where <mode> is one of the following:

```
INPUT
OUTPUT
APPEND
IBM
USR
```

The mode determines only the initial positioning within the file and the actions to be taken if the file does not exist. The action taken in each mode is:

INPUT — The initial position is at the start of the file. An error is returned if the file is not found.

OUTPUT — The initial position is at the start of the file. A new file is always created.

APPEND — The initial position is at the end of the file. An error is returned if the file is not found.

IBM The initial position is after the last DSKI$ or DSKO$. The file is then set up to write contiguous. No file search is done. (The same effect may be achieved in many cases by altering the FORMAT program. See Section H.11.2.1.)

USR Same as IBM mode except, instead of write contiguous, USR0 is called and returns the next track/sector number. The USR0 routine should read the current track/sector from B,C and return the next location in B,C. When USR0 is first called, B,C contains the track and sector number of the previous DSKI$ or DSKO$.

If the FOR <mode> clause is omitted, the initial position is at the start of the file. If the file is not found, it is created.

Note that variable length records are not supported in Standalone Disk BASIC. All records are 128 bytes in length.

USR mode is especially useful for creating diskettes that require sector mapping. This is the case if the diskette is intended for use on another system, for example, a CP/M system. Instead of opening the file for write contiguous (IBM mode), the USR0 routine may be used to map the sectors logically, as required by the other system.

When a file is OPENed FOR APPEND, the file mode is set to APPEND and the record number is set to the last record of the file. The program may subsequently execute disk I/O statements that move the pointer elsewhere in the file. When the last record is read, the file mode is reset to FILE and the pointer is left at the end of the file. Then, if you wish to append another record, execute:

```
GET#n,LOF(n)
```

This positions the pointer at the end of the file in preparation for appending.

At any one time, it is possible to have a particular filename OPEN under more than one file number. This allows different attributes to be used for different purposes. Or, for program clarity, you may wish to use different file numbers for different methods of access. Each file number has a different buffer, so changes made under one file are not accessible to (or affected by) the other numbers until that record is written (e.g., GET#n,LOC(n)).

## H.10 DISK I/O

A GET or PUT (i.e., random access) cannot be done on a file that is OPEN FOR IBM or OPEN FOR USR. Otherwise, GET/PUT may be executed along with PRINT#/INPUT# on the same file, which makes midfile updating possible. The statement formats for GET, PUT, PRINT#, and INPUT# are the same as those in BASIC-80. The action of each statement in Standalone BASIC is as follows:

GET — If the "buffer changed" flag is set, write the buffer to disk. Then execute the GET (read the record into the buffer), and reset the position for sequential I/O to the beginning of the buffer.

PUT — Execute the PUT (write the buffer to the specified record number), and set the "sequential I/O is illegal" flag until a GET is done.

INPUT# — If the buffer is empty, write it if the "Buffer changed" flag is set, then read the next buffer.

PRINT# — Set the "buffer changed" flag. If the buffer is full, write it to disk. Then, if end of file has not been reached, read the next buffer.

### H.10.1 File Format

For a single density floppy, each file requires 137 bytes: 9 bytes plus the 128-byte buffer. Because the File Allocation Table keeps random access information for all files, random and sequential files are identical on the disk. The only distinction is that sequential files have a Control-Z (32 octal) as the last character of the last sector. When this sector is read, it is scanned from the end for a non-zero byte. If this byte is Control-Z, the size of the buffer is set so that a PRINT overwrites this byte. If the byte is not Control-Z, the size is set so the last null seen is overwritten.

Any sequential file can be copied in random mode and remain identical. If a file is written to disk in random mode (i.e., with PUT instead of PRINT) and then read in sequential mode, it will still have proper end of file detection.

## H.11 DISK ALLOCATION INFORMATION

With Standalone Disk BASIC, storage space on the diskette is allocated beginning with the cluster closest to the current position of the head. (This method is optimized for writing. Custom versions can be optimized for reading.) Disk allocation information is placed in memory when the disk is mounted and is periodically written back to the disk. Because this allocation information is kept in memory, there is no need for index blocks for random files, and there is no need to distinguish between random and sequential files.

### H.11.1 Directory Format

On the diskette, each sector of the directory track contains eight file entries. Each file entry is 16 bytes long and formatted as follows:

**Bytes** **Usage**

0-8 Filename, 1 to 9 characters. The first character may not be 0 or 255.

9 Attribute:
Octal
200 Binary file
100 Force read after write check
40 EBCDIC file
20 Write protected file
Excluding 200, these bits are the same for the disk attribute byte which is the first byte of the information sector.

10 Pointer into File Allocation Table to the first cluster of the file's cluster chain.

11-15 Reserved for future expansion.

If the first byte of a filename is zero, that file entry slot is free. If the first byte is 255, that slot is the last occupied slot in the directory, i.e., this flags the end of the directory.

### H.11.2 Drive Information

For each disk drive that is MOUNTed, the following information is kept in memory:

1. Attributes
   Drive attributes are read from the information sector when the drive is mounted and may be changed with the SET statement. Current attributes may be examined with the ATTR$ function.

2. Track Number
   This is the current track while the disk is mounted. Otherwise, track number contains 255 as a flag that the disk is not mounted.

3. Modification Counter
   This counter is incremented whenever an entry in the File Allocation Table is changed. After a given number of changes has been made, the File Allocation Table is written to disk.

4. Number of Free Clusters
   This is calculated when the drive is mounted, and updated whenever a file is deleted or a cluster is allocated.

5. File Allocation Table
   The File Allocation Table has a one-byte entry for every cluster allocated on the disk. If the cluster is free, this entry is 255. If the cluster is reserved, this entry is 254. If the cluster is the last cluster of the file, this entry is 300 (octal) plus the number of sectors from this cluster that were used. Otherwise, the entry is a pointer to the next cluster of the file. The File Allocation Table is read into memory when the drive is mounted, and updated:

   1. When a file is deleted

   2. When a file is closed

   3. When modifications to the table total twice the number of sectors in a cluster (this can be changed in custom versions)

   4. When modifications to the table have been made and the disk head is on (or passes) the directory track.

11

H.11.2.1 FORMAT Program - Before mounting a drive with a new diskette, run BASIC's FORMAT program to initialize the directory (set all bytes to 255), set the information sector to 0, and set all the File Allocation Table entries (except the directory track entry (254)) to "free" (255).

The FORMAT program is:

```
10 CLEAR 1500
20 A$=STRING$(255,128)
30 B$=STRING$(35*2,255)+STRING$(2,254)+STRING$(56,255)
40 FOR S=1 TO 19:DSKO$ 1,35,5,A$:NEXT
50 FOR S=21 TO 25 STEP 2:DSKO$ 1,35,S,B$
60 DKSO$ 1,35,S+1,A$:NEXT
70 DSKO$ 1,39,20,CHR$(0)
```

After running FORMAT and MOUNTing the drive, files will be allocated as usual, i.e., on either side of the directory track.

The FORMAT program may be altered to pre-allocate selected files. For instance, you may wish to use the FORMAT program to pre-allocate files contiguously (as they would be allocated in IBM mode). Then IBM and BASIC files may both exist on the diskette. The altered FORMAT program must also write the name of the file(s) to the directory track (i.e., files1-8 in sector 1, files 9-16 in sector 2, etc.), so BASIC knows where the files start.

### H.11.3 File Block

Each file on the disk has a file block that contains the following information:

1. File Mode (byte 0)
   This is the first byte (byte 0) of the file block, and its location may be read with VARPTR(#filenumber). The location of any other byte in the file block is relative to the file mode byte. The file mode byte is one of the following:

| (octal) | |
|---|---|
| 1 | Input only |
| 2 | Output only |
| 4 | File mode |
| 10 | Append mode |
| 20 | Delete file |
| 40 | IBM mode |
| 100 | Special format (USR) |
| 200 | Binary save |

NOTE

> It is not recommended that the user attempt to modify the next four bytes of the File Allocation Table. Many unforeseen complications may result.

2. Pointer to the File Allocation Table entry for the first cluster allocated to the file (+1)

3. Pointer to the File Allocation Table entry for the last cluster accessed (+2)

4. Last sector accessed (+3)

5. Disk number of file (+4)

6. The size of the last buffer read (+5). This is 128 unless the last sector of the file is not full (i.e., Control-Z).

7. The current position in the buffer (+6). This is the offset within the buffer for the next print or input.

8. File flag (+7), is one of the following:

| Octal | |
|---|---|
| 100 | Read after write check |
| 40 | Read/Write EBCDIC, not ASCII<br>(Not available in all versions.) |
| 20 | File write protected |
| 10 | Buffer changed by PRINT |
| 4 | PUT has been done. PRINT/INPUT are<br>errors until a GET is done.<br>(See Section H.10.) |
| 2 | Flags buffer is empty |

9. Terminal position for TAB function and comma in PRINT statements (+8)

10. Beginning of sector buffer (+9), 128 bytes in length

## H.12 ADVANCED USES OF FILE BUFFERS

1. Information may be passed from one program to another by FIELDing it to an unopened file number (not #0). The FIELD buffer is not cleared as long as the file is not OPENed.

2. The FIELDed buffer for an unopened file can also be used to format strings. For example, an 80-character string could be placed into a FIELDed buffer with LSET. The strings could then be accessed as four 20-character strings using their FIELDed variable names. For example:

```
100 FIELD#1, 80 AS A$
200 FIELD#1, 20 AS A1$, 20 AS A2$, 20 AS A3$, 20 AS A4$
300 LINE INPUT "CUSTOMER INFORMATION: ";B$
400 LSET A$=B$
500 PRINT "NAME ";A1$;"SSN: ";A2$
```

3. FIELD#0 may be used as a temporary buffer, but note that this buffer is cleared after each of the following commands: FILES, LOAD, SAVE, MERGE, RUN, DSKO$, MOUNT, OPEN.

4. The effect of PRINT[USING]# into a string may be achieved by printing to a FIELDed buffer and then accessing it without reopening the file. To assure that this temporary buffer is not written to the disk, return the pointer to the beginning of the buffer and reset the "buffer changed" flag as follows:

```
10 OPEN "D" FOR IBM AS 1:REM THIS DOESN'T USE SPACE
20 PRINT USING#1 ...
30 P=PEEK(6+VARPTR(#1)):REM OPTIONAL, TO GET LENGTH OF PRINT
USING
40 FIELD#1 ... AS ...
50 Y=7+VARPTR(#1)
60 POKE Y,PEEK(Y AND &360):REM RESET BUFFER CHANGED FLAG
70 POKE 6+VARPTR,0:REM CLEAR POSITION IN BUFFER
```

## H.13 STANDALONE BASIC DISK ERRORS

| | |
|---|---|
| 50 | FIELD overflow |
| 51 | Internal error |
| 52 | Bad file number |
| 53 | File not found |
| 54 | File already open |
| 55 | Disk not mounted |
| 56 | Disk I/O error |
| 57 | File already exists |
| 59 | Disk already mounted |
| 61 | Input past end |
| 62 | Bad file name |
| 63 | Direct statement in file |
| 64 | Bad allocation table |
| 65 | Bad drive number |
| 66 | Bad track/sector |
| 67 | File write protected |
| 68 | Disk offline |
| 69 | Deleted record |
| 70 | Rename across disks |
| 71 | Sequential after PUT |
| 72 | Sequential I/O only |
| 73 | File not OPEN |

# APPENDIX I

## Converting Programs to BASIC-80

If you have programs written in a BASIC other than BASIC-80, some minor adjustments may be necessary before running them with BASIC-80. Here are some specific things to look for when converting BASIC programs.

### I.1 STRING DIMENSIONS

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the BASIC-80 statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for BASIC-80 string concatenation.

In BASIC-80, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

| Other BASIC | BASIC-80 |
|---|---|
| X$=A$(I) | X$=MID$(A$,I,1) |
| X$=A$(I,J) | X$=MID$(A$,I,J-I+1) |

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

| Other BASIC | 8K BASIC-80 |
|---|---|
| A$(I)=X$ | A$=LEFT$(A$,I-1)+X$+MID$(A$,I+1) |
| A$(I,J)=X$ | A$=LEFT$(A$,I-1);X$;MID$(A$,J+1) |
|  | Ext. and Disk BASIC-80 |
| A$(I)=X$ | MID$(A$,1,1)=X$ |
| A$(I,J)=X$ | MID$(A$,I,J-I+1)=X$ |

11

## I.2 MULTIPLE ASSIGNMENTS

Some BASICs allow statements of the form:

```
     10 LET B=C=0
```

to set B and C equal to zero. BASIC-80 would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
     10 C=0:B=0
```

## I.3 MULTIPLE STATEMENTS

Some BASICs use a backslash ( ) to separate multiple statements on a line. With BASIC-80, be sure all statements on a line are separated by a colon (:).

## I.4 MAT FUNCTIONS

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

# APPENDIX J

## Summary of Error Codes and Error Messages

| Code | Number | Message |
|---|---|---|
| BS | 9 | Subscript out of range<br>An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts. |
| CN | 17 | Can't continue<br>An attempt is made to continue a program that:<br>1. has halted due to an error,<br>2. has been modified during a break in execution, or<br>3. does not exist. |
| DD | 10 | Redimensioned array<br>Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array. |
| FC | 5 | Illegal function call<br>A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of:<br>1. a negative or unreasonably large subscript<br>2. a negative or zero argument with LOG<br>3. a negative argument to SQR<br>4. a negative mantissa with a non-integer exponent |

5. a call to a USR function for which the starting address has not yet been given

6. an improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.

ID 12 Illegal direct
A statement that is illegal in direct mode is entered as a direct mode command.

NF 1 NEXT without FOR
A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.

OD 4 Out of data
A READ statement is executed when there are no DATA statements with unread data remaining in the program.

OM 7 Out of memory
A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.

OS 14 Out of string space
String variables exceed the allocated amount of string space. Use CLEAR to allocate more string space, or decrease the size and number of strings.

OV 6 Overflow
The result of a calculation is too large to be represented in BASIC-80's number format. If underflow occurs, the result is zero and execution continues without an error.

SN 2 Syntax error
A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).

ST 16 String formula too complex
A string expression is too long or too complex. The expression should be broken into smaller expressions.

TM 13 Type mismatch
A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.

RG 3 Return without GOSUB
A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.

UF 18 Undefined user function
A USR function is called before the function definition (DEF statement) is given.

UL 8 Undefined line
A line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE is to a nonexistent line.

/0 11 Division by zero
A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

<u>Extended and Disk Versions Only</u>

19 No RESUME
An error trapping routine is entered but contains no RESUME statement.

20 RESUME without error
A RESUME statement is encountered before an error trapping routine is entered.

21 Unprintable error
An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.

22 Missing operand
An expression contains an operator with no operand following it.

23 Line buffer overflow
An attempt is made to input a line that has too many characters.

26 FOR without NEXT
A FOR was encountered without a matching NEXT.

29 WHILE without WEND
A WHILE statement does not have a matching WEND.

30 WEND without WHILE
A WEND was encountered without a matching WHILE.

## Disk Errors

50 Field overflow
A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

51 Internal error
An internal malfunction has occurred in Disk BASIC-80. Report to Microsoft the conditions under which the message appeared.

52 Bad file number
A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

53 File not found
A LOAD, KILL or OPEN statement references a file that does not exist on the current disk.

54 Bad file mode
An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file or to execute an OPEN with a file mode other than I, O, or R.

55 File already open
A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.

57 Disk I/O error
An I/O error occurred on a disk I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error.

58 File already exists
The filename specified in a NAME statement is identical to a filename already in use on the disk.

61 Disk full
All disk storage space is in use.

62 Input past end
An INPUT statement is exeucted after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.

63 Bad record number
In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.

64 Bad file name
An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).

66 Direct statement in file
A direct statement is encountered while LOADing an ASCII-format file. The LOAD is terminated.

67 Too many files
An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

11

# APPENDIX K

## Mathematical Functions

### Derived Functions

Functions that are not intrinsic to BASIC-80 may be calculated as follows:

| Function | BASIC-80 Equivalent |
|---|---|
| SECANT | SEC(X)=1/COS(X) |
| COSECANT | CSC(X)=1/SIN(X) |
| COTANGENT | COT(X)=1/TAN(X) |
| INVERSE SINE | ARCSIN(X)=ATN(X/SQR(-X*X+1)) |
| INVERSE COSINE | ARCCOS(X)=-ATN (X/SQR(-X*X+1))+1.5708 |
| INVERSE SECANT | ARCSEC(X)=ATN(X/SQR(X*X-1))<br>+SGN(SGN(X)-1)*1.5708 |
| INVERSE COSECANT | ARCCSC(X)=ATN(X/SQR(X*X-1))<br>+(SGN(X)-1)*1.5708 |
| INVERSE COTANGENT | ARCCOT(X)=ATN(X)+1.5708 |
| HYPERBOLIC SINE | SINH(X)=(EXP(X)-EXP(-X))/2 |
| HYPERBOLIC COSINE | COSH(X)=(EXP(X)+EXP(-X))/2 |
| HYPERBOLIC TANGENT | TANH(X)=EXP(-X)/EXP(X)+EXP(-X))*2+1 |
| HYPERBOLIC SECANT | SECH(X)=2/(EXP(X)+EXP(-X)) |
| HYPERBOLIC COSECANT | CSCH(X)=2/(EXP(X)-EXP(-X)) |
| HYPERBOLIC COTANGENT | COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1 |
| INVERSE HYPERBOLIC<br>SINE | ARCSINH(X)=LOG(X+SQR(X*X+1)) |
| INVERSE HYPERBOLIC<br>COSINE | ARCCOSH(X)=LOG(X+SQR(X*X-1) |
| INVERSE HYPERBOLIC<br>TANGENT | ARCTANH(X)=LOG((1+X)/(1-X))/2 |
| INVERSE HYPERBOLIC<br>SECANT | ARCSECH(X)=LOG((SQR(-X*X+1)+1)/X) |
| INVERSE HYPERBOLIC<br>COSECANT | ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X |
| INVERSE HYPERBOLIC<br>COTANGENT | ARCCOTH(X)=LOG((X+1)/(X-1))/2 |

## APPENDIX L

## ASCII Character Codes

| ASCII Code | Character | ASCII Code | Character | ASCII Code | Character |
|---|---|---|---|---|---|
| 000 | NUL | 043 | + | 086 | V |
| 001 | SOH | 044 | , | 087 | W |
| 002 | STX | 045 | - | 088 | X |
| 003 | ETX | 046 | . | 089 | Y |
| 004 | EOT | 047 | / | 090 | Z |
| 005 | ENQ | 048 | 0 | 091 | [ |
| 006 | ACK | 049 | 1 | 092 | \ |
| 007 | BEL | 050 | 2 | 093 | ] |
| 008 | BS | 051 | 3 | 094 | ∧ |
| 009 | HT | 052 | 4 | 095 | < |
| 010 | LF | 053 | 5 | 096 | ' |
| 011 | VT | 054 | 6 | 097 | a |
| 012 | FF | 055 | 7 | 098 | b |
| 013 | CR | 056 | 8 | 099 | c |
| 014 | SO | 057 | 9 | 100 | d |
| 015 | SI | 058 | : | 101 | e |
| 016 | DLE | 059 | ; | 102 | f |
| 017 | DC1 | 060 | < | 103 | g |
| 018 | DC2 | 061 | = | 104 | h |
| 019 | DC3 | 062 | > | 105 | i |
| 020 | DC4 | 063 | ? | 106 | j |
| 021 | NAK | 064 | @ | 107 | k |
| 022 | SYN | 065 | A | 108 | l |
| 023 | ETB | 066 | B | 109 | m |
| 024 | CAN | 067 | C | 110 | n |
| 025 | EM | 068 | D | 111 | o |
| 026 | SUB | 069 | E | 112 | p |
| 027 | ESCAPE | 070 | F | 113 | q |
| 028 | FS | 071 | G | 114 | r |
| 029 | GS | 072 | H | 115 | s |
| 030 | RS | 073 | I | 116 | t |
| 031 | US | 074 | J | 117 | u |
| 032 | SPACE | 075 | K | 118 | v |
| 033 | ! | 076 | L | 119 | w |
| 034 | " | 077 | M | 120 | x |
| 035 | # | 078 | N | 121 | y |
| 036 | $ | 079 | O | 122 | z |
| 037 | % | 080 | P | 123 | { |
| 038 | & | 081 | Q | 124 | \| |
| 039 | ' | 082 | R | 125 | |
| 040 | ( | 083 | S | 126 | } |
| 041 | ) | 084 | T | 127 | DEL |
| 042 | * | 085 | U | | |

ASCII codes are in decimal
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

# INDEX

11

# Microsoft
# Software Problem Report

Use this form to report errors or problems in: □ Microsoft BASIC-80<br>□ Microsoft BASIC-86<br>□ Microsoft BASIC Compiler

Date ____________________

Report only one problem per form.

Describe your hardware and operating system: ______________________

_____________________________________________________________________

BASIC Release number: ____________________

Please supply a concise description of the problem and the circumstances surrounding its occurrence. If possible, reduce the problem to a simple test case. Otherwise, include all programs and data in machine readable form (preferably on a diskette). If a patch or interim solution is being used, please describe it.

This form may also be used to describe suggested enhancements to Microsoft BASIC.

Problem Description:

-over-

Did you find errors in the BASIC-80 Reference Manual?
If so, please include page numbers and describe:

Fill in the following information before returning this form:

Name________________________________ Phone______________________

Organization____________________________________________________

Address_______________________ City___________ State____ Zip______

Return form to:     Microsoft
                    10800 NE Eighth, Suite 819
                    Bellevue, WA 98004

# MICROSOFT UTILITY SOFTWARE MANUAL

# MICROSOFT

# utility software manual

12

Microsoft
Utility Software Manual

## CONTENTS

# SECTION 1

# MACRO-80 Assembler

## 1.1 Format of MACRO-80 Commands

### 1.1.1 MACRO-80 Command Strings

To run MACRO-80, type M80 followed by a carriage return. MACRO-80 will return the prompt "*" (with the DTC operating system, the prompt is ">"), indicating it is ready to accept commands. The format of a MACRO-80 command string is:

```
objprog-dev:filename.ext,list-dev:filename.ext=
   source-dev:filename.ext
```

objprog-dev:
The device on which the object program is to be written.

list-dev:
The device on which the program listing is written.

source-dev:
The device from which the source-program input to MACRO-80 is obtained. If a device name is omitted, it defaults to the currently selected drive.

filename.ext
The filename and filename extension of the object program file, the listing file, and the source file. Filename extensions may be omitted. See Section 4 for the default extension supplied by your operating system.

Either the object file or the listing file or both may be omitted. If neither a listing file nor an object file is desired, place only a comma to the left of the equal sign. If the names of the object file and the listing file are omitted, the default is the name of the source file.

Examples:

| | |
|---|---|
| *=SOURCE.MAC | Assemble the program SOURCE.MAC and place the object in SOURCE.REL |
| *,LST:=TEST | Assemble the program TEST.MAC and list on device LST |

12

| | |
|---|---|
| *SMALL,TTY:=TEST | Assemble the program TEST.MAC, place the object in SMALL.REL and list on TTY |

### 1.1.2 MACRO-80 Switches

A number of different switches may be given in the MACRO-80 command string that will affect the format of the listing file. Each switch must be preceded by a slash (/):

| Switch | Action |
|---|---|
| O | Print all listing addresses, etc. in octal. (Default for Altair DOS) |
| H | Print all listing addresses, etc. in hexadecimal.<br>(Default for non-Altair versions) |
| R | Force generation of an object file. |
| L | Force generation of a listing file. |
| C | Force generation of a cross reference file. |
| Z | Assemble Z80 (Zilog format) mnemonics. (Default for Z80 operating systems) |
| I | Assemble 8080 mnemonics. (Default for 8080 operating systems) |

Examples:

| | |
|---|---|
| *=TEST/L | Compile TEST.MAC with object file TEST.REL and listing file TEST.LST |
| *LAST,LAST/C=MOD1 | Compile MOD1.MAC with object file LAST.REL and cross reference file LAST.CRF for use with CREF-80<br>(See Section 1.12) |

## 1.2 Format of MACRO-80 Source Files

In general, MACRO-80 accepts a source file that is almost identical to source files for INTEL compatible assemblers. Input source lines of up to 132 characters in length are acceptable.

MACRO-80 preserves lower case letters in quoted strings and comments. All symbols, opcodes and pseudo-opcodes typed in lower case will be converted to upper case.

NOTE

> If the source file includes line numbers from an editor, each byte of the line number must have the high bit on. Line numbers from Microsoft's EDIT-80 Editor are acceptable.

### 1.2.1 Statements

Source files input to MACRO-80 consist of statements of the form:

[label:[:]] [operator] [arguments] [;comment]

With the exception of the ISIS assembler $ controls (see Section 1.10), it is not necessary that statements begin in column 1. Multiple blanks or tabs may be used to improve readability.

If a label is present, it is the first item in the statement and is immediately followed by a colon. If it is followed by two colons, it is declared as PUBLIC (see ENTRY/PUBLIC, Section 1.5.10). For exmple:

```
     FOO::    RET
```

is equivalent to

```
     PUBLIC   FOO
     FOO:     RET
```

The next item after the label (or the first item on the line if no label is present) is an operator. An operator may be an opcode (8080 or Z80 mnemonic), pseudo-op, macro call or expression. The evaluation order is as follows:

1. Macro call

2. Opcode/Pseudo operation

3. Expression

Instead of flagging an expression as an error, the assembler treats it as if it were a DB statement (see Section 1.5.4).

The arguments following the operator will, of course, vary in form according to the operator.

A comment always begins with a semicolon and ends with a carriage return. A comment may be a line by itself or it may be appended to a line that contains a statement. Extended comments can be entered using the .COMMENT pseudo operation (see Section 1.5.19).

### 1.2.2 Symbols

MACRO-80 symbols may be of any length, however, only the first six characters are significant. The following characters are legal in a symbol:

A-Z 0-9 $ . ? @

With the 8080/Z80 assembler, the underline character is also legal in a symbol. A symbol may not start with a digit. When a symbol is read, lower case is translated into upper case. If a symbol reference is followed by ## it is declared external (see also the EXT/EXTRN pseudo-op, Section 1.5.12).

### 1.2.3 Numeric Constants

The default base for numeric constants is decimal. This may be changed by the .RADIX pseudo-op (see Section 1.5.21). Any base from 2 (binary) to 16 (hexadecimal) may be selected. When the base is greater than 10, A-F are the digits following 9. If the first digit of the number is not numeric (i.e., A-F), the number must be preceded by a zero. This eliminates the use of zero as a leading digit for octal constants, as in previous versions of MACRO-80.

Numbers are 16-bit unsigned quantities. A number is always evaluated in the current radix unless one of the following special notations is used:

| | |
|---|---|
| nnnnB | Binary |
| nnnnD | Decimal |
| nnnnO | Octal |
| nnnnQ | Octal |
| nnnnH | Hexadecimal |
| X'nnnn' | Hexadecimal |

Overflow of a number beyond two bytes is ignored and the result is the low order 16-bits.

A character constant is a string comprised of zero, one or two ASCII characters, delimited by quotation marks, and used in a non-simple expression. For example, in the statement

```
        DB      'A' + 1
```

'A' is a character constant. But the statement

```
        DB      'A'
```

uses 'A' as a string because it is in a simple expression. The rules for character constant delimiters are the same as for strings.

A character constant comprised of one character has as its value the ASCII value of that character. That is, the high order byte of the value is zero, and the low order byte is the ASCII value of the character. For example, the value of the constant 'A' is 41H.

A character constant comprised of two characters has as its value the ASCII value of the first character in the high order byte and the ASCII value of the second character in the low order byte. For example, the value of the character constant "AB" is 41H*256+42H.

### 1.2.4 Strings

A string is comprised of zero or more characters delimited by quotation marks. Either single or double quotes may be used as string delimiters. The delimiter quotes may be used as characters if they appear twice for every character occurrence desired. For example, the statement

```
        DB      "I am ""great"" today"
```

stores the string

```
                I am "great" today
```

If there are zero characters between the delimiters, the string is a null string.

## 1.3 Expression Evaluation

### 1.3.1 Arithmetic and Logical Operators

The following operators are allowed in expressions. The operators are listed in order of precedence.

NUL

LOW, HIGH

*, /, MOD, SHR, SHL

Unary Minus

+, -

EQ, NE, LT, LE, GT, GE

NOT

AND

OR, XOR

Parentheses are used to change the order of precedence. During evaluation of an expression, as soon as a new operator is encountered that has precedence less than or equal to the last operator encountered, all operations up to the new operator are performed. That is, subexpressions involving operators of higher precedence are computed first.

All operators except +, -, *, / must be separated from their operands by at least one space.

The byte isolation operators (HIGH, LOW) isolate the high or low order 8 bits of an Absolute 16-bit value. If a relocatable value is supplied as an operand, HIGH and LOW will treat it as if it were relative to location zero.

### 1.3.2 Modes

All symbols used as operands in expressions are in one of the following modes: Absolute, Data Relative, Program (Code) Relative or COMMON. (See Section 1.5 for the ASEG, CSEG, DSEG and COMMON pseudo-ops.) Symbols assembled under the ASEG, CSEG (default), or DSEG pseudo-ops are in Absolute, Code Relative or Data Relative mode respectively. The number of COMMON modes in a program is determined by the number of COMMON blocks that have been named with the COMMON pseudo-op. Two COMMON symbols are not in the same mode unless they are in the same COMMON block.

In any operation other than addition or subtraction, the mode of both operands must be Absolute.

If the operation is addition, the following rules apply:

1. At least one of the operands must be Absolute.

2. Absolute + <mode> = <mode>

If the operation is subtraction, the following rules apply:

1. <mode> - Absolute = <mode>

2. <mode> - <mode> = Absolute
   where the two <mode>s are the same.

Each intermediate step in the evaluation of an expression must conform to the above rules for modes, or an error will be generated. For example, if FOO, BAZ and ZAZ are three Program Relative symbols, the expression

```
FOO + BAZ - ZAZ
```

will generate an R error because the first step (FOO + BAZ) adds two relocatable values. (One of the values must be Absolute.) This problem can always be fixed by inserting parentheses. So that

```
FOO + (BAZ - ZAZ)
```

is legal because the first step (BAZ - ZAZ) generates an Absolute value that is then added to the Program Relative value, FOO.

12

### 1.3.3 Externals

Aside from its classification by mode, a symbol is either External or not External. (See EXT/EXTRN, Section 1.5.12.) An External value must be assembled into a two-byte field. (Single-byte Externals are not supported.) The following rules apply to the use of Externals in expressions:

1. Externals are legal only in addition and subtraction.

2. If an External symbol is used in an expression, the result of the expression is always External.

3. When the operation is addition, either operand (but not both) may be External.

4. When the operation is subtraction, only the first operand may be External.

## 1.4 Opcodes as Operands

8080 opcodes are valid one-byte ·operands. Note that only the first byte is a valid operand. For example:

```
MVI    A,(JMP)
ADI    (CPI)
MVI    B,(RNZ)
CPI    (INX H)
ACI    (LXI B)
MVI    C,MOV A,B
```

Errors will be generated if more than one byte is included in the operand -- such as (CPI 5), LXI B,LABEL1) or (JMP LABEL2).

Opcodes used as one-byte operands need not be enclosed in parentheses.

NOTE

Opcodes are not valid operands in Z80 mode.

## 1.5 Pseudo Operations

### 1.5.1 ASEG

```
ASEG
```

ASEG sets the location counter to an absolute segment of memory. The location of the absolute counter will be that of the last ASEG (default is 0), unless an ORG is done after the ASEG to change the location. The effect of ASEG is also achieved by using the code segment (CSEG) pseudo operation and the /P switch in LINK-80. See also Section 1.5.27.

### 1.5.2 COMMON

```
COMMON /<block name>/
```

COMMON sets the location counter to the selected common block in memory. The location is always the beginning of the area so that compatibility with the FORTRAN COMMON statement is maintained. If <block name> is omitted or consists of spaces, it is considered to be blank common. See also Section 1.5.27.

### 1.5.3 CSEG

```
CSEG
```

CSEG sets thelocation counter to the code relative segment of memory. The location will be that of the last CSEG (default is 0), unless an ORG is done after the CSEG to change the location. CSEG is the default condition of the assembler (the INTEL assembler defaults to ASEG). See also Section 1.5.27.

### 1.5.4 Define Byte

```
DB      <exp>[,<exp>...]

DB      <string>[<string>...]
```

The arguments to DB are either expressions or strings. DB stores the values of the expressions or the characters of the strings in successive memory locations beginning with the current location counter.

Expressions must evaluate to one byte. (If the high byte of the result is 0 or 255, no error is given; otherwise, an A error results.)

Strings of three or more characters may not be used in expressions (i.e., they must be immediately followed by a comma or the end of the line). The characters in a string are stored in the order of appearance, each as a one-byte value with the high order bit set to zero.

Example:

```
0000'   4142            DB      'AB'
0002'   42              DB      'AB' AND 0FFH
0003'   41 42 43        DB      'ABC'
```

12

### 1.5.5 Define Character

```
DC      <string>
```

DC stores the characters in <string> in successive memory locations beginning with the current location counter. As with DB, characters are stored in order of appearance, each as a one-byte value with the high order bit set to zero. However, DC stores the last character of the string with the high order bit set to one. An error will result if the argument to DC is a null string.

### 1.5.6 Define Space

```
DS      <exp>
```

DS reserves an area of memory. The value of <exp> gives the number of bytes to be allocated. All names used in <exp> must be previously defined (i.e., all names known at that point on pass 1). Otherwise, a V error is generated during pass 1 and a U error may be generated during pass 2. If a U error is not generated during pass 2, a phase error will probably be generated because the DS generated no code on pass 1.

### 1.5.7 DSEG

```
DSEG
```

DSEG sets the location counter to the Data Relative segment of memory. The location of the data relative counter will be that of the last DSEG (default is 0), unless an ORG is done after the DSEG to change the location. See also Section 1.5.27.

### 1.5.8 Define Word

```
DW      <exp>[,<exp>...]
```

DW stores the values of the expressions in successive memory locations beginning with the current location counter. Expressions are evaluated as 2-byte (word) values.

### 1.5.9 END

```
END     [<exp>]
```

The END statement specifies the end of the program. If <exp> is present, it is the start address of the program. If <exp> is not present, then no start address is passed to LINK-80 for that program.

### 1.5.10 ENTRY/PUBLIC

```
ENTRY  <name>[,<name>...]
 or
PUBLIC <name>[,<name>...]
```

ENTRY or PUBLIC declares each name in the list as internal and therefore available for use by this program and other programs to be loaded concurrently. All of the names in the list must be defined in the current program or a U error results. An M error is generated if the name is an external name or common-blockname.

### 1.5.11 EQU

```
<name> EQU <exp>
```

EQU assigns the value of <exp> to <name>. If <exp> is external, an error is generated. If <name> already has a value other than <exp>, an M error is generated.

### 1.5.12 EXT/EXTRN

```
EXT    <name>[,<name>...]
 or
EXTRN  <name>[,<name>...]
```

EXT or EXTRN declares that the name(s) in the list are external (i.e., defined in a different program). If any item in the list references a name that is defined in the current program, an M error results. A reference to a name where the name is followed immediately by two pound signs (e.g., NAME##) also declares the name as external.

### 1.5.13 NAME

NAME ('modname')

NAME defines a name for the module. Only the first six characters are significant in a module name. A module name may also be defined with the TITLE pseudo-op. In the absence of both the NAME and TITLE pseudo-ops, the module name is created from the source file name.

### 1.5.14 Define Origin

ORG <exp>

The location counter is set to the value of <exp> and the assembler assigns generated code starting with that value. All names used in <exp> must be known on pass 1, and the value must either be absolute or in the same area as the location counter.

### 1.5.15 PAGE

PAGE [<exp>]

PAGE causes the assembler to start a new output page. The value of <exp>, if included, becomes the new page size (measured in lines per page) and must be in the range 10 to 255. The default page size is 50 lines per page. The assembler puts a form feed character in the listing file at the end of a page.

### 1.5.16 SET

<name> SET <exp>

SET is the same as EQU, except no error is generated if <name> is already defined.

### 1.5.17 SUBTTL

SUBTTL <text>

SUBTTL specifies a subtitle to be listed on the line after the title (see TITLE, Section 1.5.18) on each page heading. <text> is truncated after 60 characters. Any number of SUBTTLs may be given in a program..

### 1.5.18 TITLE

```
TITLE <text>
```

TITLE specifies a title to be listed on the first line of each page. If more than one TITLE is given, a Q error results. The first six characters of the title are used as the module name unless a NAME pseudo operation is used. If neither a NAME or TITLE pseudo-op is used, the module name is created from the source filename.

### 1.5.19 .COMMENT

```
.COMMENT <delim><text><delim>
```

The first non-blank character encountered after .COMMENT is the delimiter. The following <text> comprises a comment block which continues until the next occurrence of <delimiter> is encountered. For example, using an asterisk as the delimiter, the format of the comment block would be:

```
.COMMENT  *
any amount of text entered
here as the comment block
    .
    .
    .      *
;return to normal mode
```

### 1.5.20 .PRINTX

```
.PRINTX <delim><text><delim>
```

The first non-blank character encountered after .PRINTX is the delimiter. The following text is listed on the terminal during assembly until another occurrence of the delimiter is encountered. .PRINTX is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches. For example:

```
IF     CPM
.PRINTX /CPM version/
ENDIF
```

NOTE

.PRINTX will output on both passes. If only one printout is desired, use the IF1 or IF2 pseudo-op.

### 1.5.21 .RADIX

```
.RADIX <exp>
```

The default base (or radix) for all constants is decimal. The .RADIX statement allows the default radix to be changed to any base in the range 2 to 16. For example:

```
LXI    H,0FFH
.RADIX 16
LXI    H,0FF
```

The two LXIs in the example are identical. The <exp> in a .RADIX statement is always in decimal radix, regardless of the current radix.

### 1.5.22 .REQUEST

```
.REQUEST <filename>[,<filename>...]
```

.REQUEST sends a request to the LINK-80 loader to search the filenames in the list for undefined globals before searching the FORTRAN library. The filenames in the list should be in the form of legal MACRO-80 symbols. They should not include filename extensions or disk specifications. The LINK-80 loader will supply its default extension and will assume the currently selected disk drive.

### 1.5.23 .Z80

.Z80 enables the assembler to accept Z80 opcodes. This is the default condition when the assembler is running on a Z80 operating system. Z80 mode may also be set by appending the Z switch to the MACRO-80 command string -- see Section 1.1.2.

### 1.5.24 .8080

.8080 enables the assembler to accept 8080 opcodes. This is the default condition when the assembler is running on an 8080 operating system. 8080 mode may also be set by appending the I switch to the MACRO-80 command string -- see Section 1.1.2.

### 1.5.25 Conditional Pseudo Operations

The conditional pseudo operations are:

| | |
|---|---|
| IF/IFT | True if <exp> is not 0. |
| IFE/IFF <exp> | True if <exp> is 0. |
| IF1 | True if pass 1. |
| IF2 | True if pass 2. |
| IFDEF <symbol> | True if <symbol> is defined or has been declared External. |
| IFNDEF <symbol> | True if <symbol> is undefined or not declared External. |
| IFB <arg> | True if <arg> is blank. The angle brackets around <arg> are required. |
| IFNB <arg> | True if <arg> is not blank. Used for testing when dummy parameters are supplied. The angle brackets around <arg> are required. |

All conditionals use the following format:

```
IFxx   [argument]
  .
  .
  .
[ELSE
  .
  .
  .    ]
ENDIF
```

12

Conditionals may be nested to any level. Any argument to a conditional must be known on pass 1 to avoid V errors and incorrect evaluation. For IF, IFT, IFF, and IFE the expression must involve values which were previously defined and the expression must be absolute. If the name is defined after an IFDEF or IFNDEF, pass 1 considers the name to be undefined, but it will be defined on pass 2.

ELSE
Each conditional pseudo operation may optionally be used with the ELSE pseudo operation which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a

given IF, and an ELSE is always bound to the most recent, open IF. A conditional with more than one ELSE or an ELSE without a conditional will cause a C error.

ENDIF
Each IF must have a matching ENDIF to terminate the conditional. Otherwise, an 'Unterminated conditional' message is generated at the end of each pass. An ENDIF without a matching IF causes a C error.

### 1.5.26 Listing Control Pseudo Operations

Output to the listing file can be controlled by two pseudo-ops:

.LIST and .XLIST

If a listing is not being made, these pseudo-ops have no effect. .LIST is the default condition. When a .XLIST is encountered, source and object code will not be listed until a .LIST is encountered.

The output of cross reference information is controlled by .CREF and .XCREF. If the cross reference facility (see Section 1.12) has not been invoked, .CREF and .XCREF have no effect. The default condition is .CREF. When a .XCREF is encountered, no cross reference information is output until .CREF is encountered.

The output of MACRO/REPT/IRP/IRPC expansions is controlled by three pseudo-ops: .LALL, .SALL, and .XALL. .LALL lists the complete macro text for all expansions. .SALL lists only the object code produced by a macro and not its text. .XALL is the default condition; it is similar to .SALL, except a source line is listed only if it generates object code.

### 1.5.27 Relocation Pseudo Operations

The ability to create relocatable modules is one of the major features of MACRO-80. Relocatable modules offer the advantages of easier coding and faster testing, debugging and modifying. In addition, it is possible to specify segments of assembled code that will later be loaded into RAM (the Data Relative segment) and ROM/PROM (the Code Relative segment). The pseudo operations that select relocatable areas are CSEG and DSEG. The ASEG pseudo-op is used to generate non-relocatable (absolute) code. The COMMON pseudo-op creates a common data area for every COMMON block that is named in the program.

The default mode for the assembler is Code Relative. That is, assembly begins with a CSEG automatically executed and the location counter in the Code Relative mode, pointing to location 0 in the Code Relative segment of memory. All subsequent instructions will be assembled into the Code Relative segment of memory until an ASEG or DSEG or COMMON pseudo-op is executed. For example, the first DSEG encountered sets the location counter to location zero in the Data Relative segment of memory. The following code is asembled in the Data Relative mode, that is, it is assigned to the Data Relative segment of memory. If a subsequent CSEG is encountered, the location counter will return to the next free location in the Code Relative segment and so on.

The ASEG, DSEG, CSEG pseudo-ops never have operands. If you wish to alter the current value of the location counter, use the ORG pseudo-op.

### ORG Pseudo-op

At any time, the value of the location counter may be changed by use of the the ORG pseudo-op. The form of the ORG statement is:

```
ORG     <exp>
```

where the value of <exp> will be the new value of the location counter in the current mode. All names used in <exp> must be known on pass 1 and the value of <exp> must be either Absolute or in the current mode of the location counter. For example, the statements

```
DSEG
ORG     50
```

set the Data Relative location counter to 50, relative to the start of the Data Relative segment of memory.

### LINK-80

The LINK-80 linking loader (see Section 2 of this manual) combines the segments and creates each relocatable module in memory when the program is loaded. The origins of the relocatable segments are not fixed until the program is loaded and the origins are assigned by LINK-80. The command to

LINK-80 may contain user-specified origins through the use of the /P (for Code Relative) and /D (for Data and COMMON segments) switches.

For example, a program that begins with the statements

```
        ASEG
        ORG     800H
```

and is assembled entirely in Absolute mode will always load beginning at 800 unless the ORG statement is changed in the source file. However, the same program, assembled in Code Relative mode with no ORG statement, may be loaded at any specified address by appending the /P:<address> switch to the LINK-80 command string.

### 1.5.28 Relocation Before Loading

Two pseudo-ops, .PHASE and .DEPHASE, allow code to be located in one area, but executed only at a different,specified area.

For example:

```
  0000'                            .PHASE   100H
  0100     CD 0106       FOO:      CALL     BAZ
  0103     C3 0007'                JMP      ZOO
  0106     C9            BAZ:      RET
                                   .DEPHASE
  0007'    C3 0005       ZOO:      JMP      5
```

All labels within a .PHASE block are defined as the absolute value from the origin of the phase area. The code, however, is loaded in the current area (i.e., from 0' in this example). The code within the block can later be moved to 100H and executed.

## 1.6 Macros and Block Pseudo Operations

The macro facilities provided by MACRO-80 include three repeat pseudo operations: repeat (REPT), indefinite repeat (IRP), and indefinite repeat character (IRPC). A macro definition operation (MACRO) is also provided. Each of these four macro operations is terminated by the ENDM pseudo operation.

### 1.6.1 Terms

For the purposes of discussion of macros and block operations, the following terms will be used:

1. <dummy> is used to represent a dummy parameter. All dummy parameters are legal symbols that appear in the body of a macro expansion.

2. <dummylist> is a list of <dummy>s separated by commas.

3. <arglist> is a list of arguments separated by commas. <arglist> must be delimited by angle brackets. Two angle brackets with no intervening characters (<>) or two commas with no intervening characters enter a null argument in the list. Otherwise an argument is a character or series of characters terminated by a comma or >. With angle brackets that are nested inside an <arglist>, one level of brackets is removed each time the bracketed argument is used in an <arglist>. (See example, Section 1.6.5.) A quoted string is an acceptable argument and is passed as such. Unless enclosed in brackets or a quoted string, leading and trailing spaces are deleted from arguments.

4. <paramlist> is used to represent a list of actual parameters separated by commas. No delimiters are required (the list is terminated by the end of line or a comment), but the rules for entering null parameters and nesting brackets are the same as described for <arglist>. (See example, Section 1.6.5.)

### 1.6.2 REPT-ENDM

```
REPT   <exp>
  .
  .
  .
ENDM
```

The block of statements between REPT and ENDM is repeated <exp> times. <exp> is evaluated as a 16-bit unsigned number. If <exp> contains any external or undefined terms, an error is generated. Example:

```
SET    0
REPT   10     ;generates DB1-DB10
SET    X+1
DB     X
ENDM
```

### 1.6.3 IRP-ENDM

```
        IRP     <dummy>,<arglist>
         .
         .
         .
        ENDM
```

The <arglist> must be enclosed in angle brackets. The number of arguments in the <arglist> determines the number of times the block of statements is repeated. Each repetition substitutes the next item in the <arglist> for every occurrence of <dummy> in the block. If the <arglist> is null (i.e., <>), the block is processed once with each occurrence of <dummy> removed. For example:

```
        IRP     X,<1,2,3,4,5,6,7,8,9,10>
        DB      X
        ENDM
```

generates the same bytes as the REPT example.

### 1.6.4 IRPC-ENDM

```
        IRPC    <dummy>,string (or <string>)
         .
         .
         .
        ENDM
```

IRPC is similar to IRP but the arglist is replaced by a string of text and the angle brackets around the string are optional. The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of <dummy> in the block. For example:

```
        IRPC    X,0123456789
        DB      X+1
        ENDM
```

generates the same code as the two previous examples.

### 1.6.5 MACRO

Often it is convenient to be able to generate a given sequence of statements from various places in a program, even though different parameters may be required each time the sequence is used. This capability is provided by the MACRO statement. The form is

```
<name> MACRO <dummylist>
          .
          .
          .
       ENDM
```

where <name> conforms to the rules for forming symbols. <name> is the name that will be used to invoke the macro. The <dummy>s in <dummylist> are the parameters that will be changed (replaced) each time the MACRO is invoked. The statements before the ENDM comprise the body of the macro. During assembly, the macro is expanded everytime it is invoked but, unlike REPT/IRP/IRPC, the macro is not expanded when it is encountered.

The form of a macro call is

```
<name> <paramlist>
```

where <name> is the name supplied in the MACRO definition, and the parameters in <paramlist> will replace the <dummy>s in the MACRO <dummylist> on a one-to-one basis. The number of items in <dummylist> and <paramlist> is limited only by the length of a line. The number of parameters used when the macro is called need not be the same as the number of <dummy>s in <dummylist>. If there are more parameters than <dummmy>s, the extras are ignored. If there are fewer, the extra <dummy>s will be made null. The assembled code will contain the macro expansion code after each macro call.

NOTE

A dummy parameter in a MACRO/REPT/IRP/IRPC is always recognized exclusively as a dummmy parameter. Register names such as A and B will be changed in the expansion if they were used as dummy parameters.

Here is an example of a MACRO definition that defines a macro called FOO:

```
FOO     MACRO   X
Y       SET     0
        REPT    X
Y       SET     Y+1
        DB      Y
        ENDM
        ENDM
```

This macro generates the same code as the previous three examples when the call

```
        FOO     10
```

is executed.

Another example, which generates the same code, illustrates the removal of one level of brackets when an argument is used as an arglist:

```
FOO     MACRO   X
        IRP     Y,<X>
        DB      Y
        ENDM
        ENDM
```

When the call

```
        FOO     <1,2,3,4,5,6,7,8,9,10>
```

is made, the macro expansion looks like this:

```
        IRP     Y,<1,2,3,4,5,6,7,8,9,10>
        DB      Y
        ENDM
```

### 1.6.6 ENDM

Every REPT, IRP, IRPC and MACRO pseudo-op must be terminated with the ENDM pseudo-op. Otherwise, the 'Unterminated REPT/IRP/IRPC/MACRO' message is generated at the end of each pass. An unmatched ENDM causes an O error.

### 1.6.7 EXITM

The EXITM pseudo-op is used to terminate a REPT/IRP/IRPC or MACRO call. When an EXITM is executed, the expansion is exited immediately and any remaining expansion or repetition is not generated. If the block containing the EXITM is nested within another block, the outer level continues to be expanded.

### 1.6.8 LOCAL

```
LOCAL  <dummylist>
```

The LOCAL pseudo-op is allowed only inside a MACRO definition. When LOCAL is executed, the assembler creates a unique symbol for each <dummy> is <dummylist> and substitutes that symbol for each occurrence of the <dummy> in the expansion. These unique symbols are usually used to define a label within a macro, thus eliminating multiply-defined labels on successive expansions of the macro. The symbols created by the assembler range from ..0001 to ..FFFF. Users will therefore want to avoid the form ..nnnn for their own symbols. If LOCAL statements are used, they must be the first statements in the macro definition.

### 1.6.9 Special Macro Operators and Forms

& The ampersand is used in a macro expansion to concatenate text or symbols. A dummy parameter that is in a quoted string will not be substituted in the expansion unless it is immediately preceded by &. To form a symbol from text and a dummy, put & between them. For example:

```
ERRGEN  MACRO   X
ERROR&X:PUSH    B
        MVI     B,'&X'
        JMP     ERROR
        ENDM
```

In this example, the call ERRGEN A will generate:

```
ERRORA: PUSH    B
        MVI     B,'A'
        JMP     ERROR
```

;; In a block operation, a comment preceded by two semicolons is not saved as part of the expansion (i.e., it will not appear on the listing even under .LALL). A comment preceded by one semicolon, however, will be preserved and appear in the expansion.

! When an exclamation point is used in an argument, the next character is entered literally (i.e., !; and <;> are equivalent).

NUL NUL is an operator that returns true if its argument (a parameter) is null. The remainder of a line after NUL is considered to be the argument to NUL. The conditional

```
IF NUL argument
```

is false if, during the expansion, the first character of the argument is anything other than a semicolon or carriage return. It is recommended that testing for null parameters be done using the IFB and IFNB conditionals.

## 1.7 Using Z80 Pseudo-ops

When using the 8080/Z80 assembler, the following Z80 pseudo-ops are valid. The function of each pseudo-op is equivalent to that of its 8080 counterpart.

| Z80 pseudo-op | Equivalent 8080 pseudo-op |
|---|---|
| COND | IFT |
| ENDC | ENDIF |
| *EJECT | PAGE |
| DEFB | DB |
| DEFS | DS |
| DEFW | DW |
| DEFM | DB |
| DEFL | SET |
| GLOBAL | PUBLIC |
| EXTERNAL | EXTRN |

The formats, where different, conform to the 8080 format. That is, DEFB and DEFW are permitted a list of arguments (as are DB and DW), and DEFM is permitted a string or numeric argument (as is DB).

## 1.8 Sample Assembly

```
A>M80

*EXMPL1,TTY:=EXMPL1

        MAC80 3.2        PAGE     1


                                 00100    ;CSL3(P1,P2)
                                 00200    ;SHIFT P1 LEFT CIRCULARLY 3 BITS
                                 00300    ;RETURN RESULT IN P2
                                 00400            ENTRY   CSL3
                                 00450    ;GET VALUE OF FIRST PARAMETER
                                 00500    CSL3:
  0000'   7E                     00600            MOV     A,M
  0001'   23                     00700            INX     H
  0002'   66                     00800            MOV     H,M
  0003'   6F                     00900            MOV     L,A
                                 01000    ;SHIFT COUNT
  0004'   06 03                  01100            MVI     B,3
  0006'   AF                     01200    LOOP:   XRA     A
                                 01300    ;SHIFT LEFT
  0007'   29                     01400            DAD     H
                                 01500    ;ROTATE IN CY BIT
  0008'   17                     01600            RAL
  0009'   85                     01700            ADD     L
  000A'   6F                     01800            MOV     L,A
                                 01900    ;DECREMENT COUNT
  000B'   05                     02000            DCR     B
                                 02100    ;ONE MORE TIME
  000C'   C2 0006'               02200            JNZ     LOOP
  000F'   EB                     02300            XCHG
                                 02400    ;SAVE RESULT IN SECOND PARAMETER
  0010'   73                     02500            MOV     M,E
  0011'   23                     02600            INX     H
  0012'   72                     02700            MOV     M,D
  0013'   C9                     02800            RET
                                 02900            END


        MAC80 3.2        PAGE     S


CSL3    0000I'  LOOP    0006'


No  Fatal error(s)
```

## 1.9 MACRO-80 Errors

MACRO-80 errors are indicated by a one-character flag in column one of the listing file. If a listing file is not being printed on the terminal, each erroneous line is also printed or displayed on the terminal. Below is a list of the MACRO-80 Error Codes:

A Argument error
Argument to pseudo-op is not in correct format or is out of range (.PAGE 1; .RADIX 1; PUBLIC 1; STAX H; MOV M,M; INX C).

C Conditional nesting error
ELSE without IF, ENDIF without IF, two ELSEs on one IF.

D Double Defined symbol
Reference to a symbol which is multiply defined.

E External error
Use of an external illegal in context (e.g., FOO SET NAME##; MVI A,2-NAME##).

M Multiply Defined symbol
Definition of a symbol which is multiply defined.

N Number error
Error in a number, usually a bad digit (e.g., 8Q).

O Bad opcode or objectionable syntax
ENDM, LOCAL outside a block; SET, EQU or MACRO without a name; bad syntax in an opcode (MOV A:); or bad syntax in an expression (mismatched parenthesis, quotes, consecutive operators, etc.).

P Phase error
Value of a label or EQU name is different on pass 2.

Q Questionable
Usually means a line is not terminated properly. This is a warning error (e.g. MOV A,B,).

R Relocation
Illegal use of relocation in expression, such as abs-rel. Data, code and COMMON areas are relocatable.

U Undefined symbol
A symbol referenced in an expression is not defined. (For certain pseudo-ops, a V error is printed on pass 1 and a U on pass 2.)

V Value error
On pass 1 a pseudo-op which must have its value known on pass 1 (e.g., .RADIX, .PAGE, DS, IF, IFE, etc.), has a value which is undefined. If the symbol is defined later in the program, a U error will not appear on the pass 2 listing.

Error Messages:

'No end statement encountered on input file'
No END statement: either it is missing or it is not parsed due to being in a false conditional, unterminated IRP/IRPC/REPT block or terminated macro.

'Unterminated conditional'
At least one conditional is unterminated at the end of the file.

'Unterminated REPT/IRP/IRPC/MACRO'
At least one block is unterminated.

[xx] [No] Fatal error(s) [,xx warnings]
The number of fatal errors and warnings. The message is listed on the CRT and in the list file.

## 1.10 Compatibility with Other Assemblers

The $EJECT and $TITLE controls are provided for compatability with INTEL's ISIS assembler. The dollar sign must appear in column 1 only if spaces or tabs separate the dollar sign from the control word. The control

```
$EJECT
```

is the same as the MACRO-80 PAGE pseudo-op.
The control

```
$TITLE('text')
```

is the same as the MACRO-80 SUBTTL <text> pseudo-op.

The INTEL operands PAGE and INPAGE generate Q errors when used with the MACRO-80 CSEG or DSEG

12

pseudo-ops. These errors are warnings; the assembler ignores the operands.

When MACRO-80 is entered, the default for the origin is Code Relative 0. With the INTEL ISIS assembler, the default is Absolute 0.

With MACRO-80, the dollar sign ($) is a defined constant that indicates the value of the location counter at the start of the statement. Other assemblers may use a decimal point or an asterisk. Other constants are defined by MACRO-80 to have the following values:

```
A=7     B=0     C=1     D=2     E=3
H=4     L=5     M=6     SP=6    PSW=6
```

## 1.11 Format of Listings

On each page of a MACRO-80 listing, the first two lines have the form:

```
[TITLE text]    MAC80 3.2    PAGE x[-y]
[SUBTTL text]
```

where:

1. TITLE text is the text supplied with the TITLE pseudo-op, if one was given in the source program.

2. x is the major page number, which is incremented only when a form feed is encountered in the source file. (When using Microsoft's EDIT-80 text editor, a form feed is inserted whenever a page mark is done.) When the symbol table is being printed, x = 'S'.

3. y is the minor page number, which is incremented whenever the .PAGE pseudo-op is encountered in the source file, or whenever the current page size has been filled.

4. SUBTTL text is the text supplied with the SUBTTL pseudo-op, if one was given in the source program.

Next, a blank line is printed, followed by the first line of output.

A line of output on a MACRO-80 listing has the following form:

```
[crf#]     [error] loc#m    xx    xxxx ...    source
```

If cross reference information is being output, the first item on the line is the cross reference number, followed by a tab.

A one-letter error code followed by a space appears next on the line, if the line contains an error. If there is no error, a space is printed. If there is no cross reference number, the error code column is the first column on the listing.

The value of the location counter appears next on the line. It is a 4-digit hexadecimal number or 6-digit octal number, depending on whether the /O or /H switch was given in the MACRO-80 command string.

The character at the end of the location counter value is the mode indicator. It will be one of the following symbols:

| | |
|---|---|
| ' | Code Relative |
| " | Data Relative |
| ! | COMMON Relative |
| <space> | Absolute |
| * | External |

Next, three spaces are printed followed by the assembled code. One-byte values are followed by a space. Two-byte values are followed by a mode indicator. Two-byte values are printed in the opposite order they are stored in, i.e., the high order byte is printed first. Externals are either the offset or the value of the pointer to the next External in the chain.

The remainder of the line contains the line of source code, as it was input.

### 1.11.1 Symbol Table Listing

12

In the symbol table listing, all the macro names in the program are listed alphabetically, followed by all the symbols in the program, listed alphabetically. After each symbol, a tab is printed, followed by the value of the symbol. If the symbol is Public, an I is printed immediately after the value. The next character printed will be one of the following.

| | |
|---|---|
| U | Undefined symbol. |
| C | COMMON block name. (The "value" of the COMMON block is its length (number of bytes) in hexadecimal or octal.) |
| * | External symbol. |
| <space> | Absolute value. |
| ' | Program Relative value. |
| " | Data Relative value. |
| ! | COMMON Relative value. |

## 1.12 Cross Reference Facility

The Cross Reference Facility is invoked by typing CREF80. In order to generate a cross reference listing, the assembler must output a special listing file with embedded control characters. The MACRO-80 command string tells the assembler to output this special listing file. (See Section 1.5.26 for the .CREF and .XCREF pseudo-ops.) /C is the cross reference switch. When the /C switch is encountered in a MACRO-80 command string, the assembler opens a .CRF file instead of a .LST file.

Examples:

| | |
|---|---|
| *=TEST/C | Assemble file TEST.MAC and create object file TEST.REL and cross reference file TEST.CRF. |
| *T,U=TEST/C | Assemble file TEST.MAC and create object file T.REL and cross reference file U.CRF. |

When the assembler is finished, it is necessary to call the cross reference facility by typing CREF80. The command string is:

```
*listing file=source file
```

Possible command strings are: The default extension for the source file is .CRF. The /L switch is ignored, and any other switch will cause an error message to be sent to the terminal. Possible command strings are:

| | |
|---|---|
| *=TEST | Examine file TEST.CRF and generate a cross reference listing file TEST.LST. |
| *T=TEST | Examine file TEST.CRF and generate a cross reference listing file T.LST. |

Cross reference listing files differ from ordinary listing files in that:

1. Each source statement is numbered with a cross reference number.

2. At the end of the listing, variable names appear in alphabetic order along with the numbers of the lines on which they are referenced or defined. Line numbers on which the symbol is defined are flagged with '#'.

12

# SECTION 2

# LINK-80 Linking Loader

## 2.1 Format of LINK-80 Commands

### 2.1.1 LINK-80 Command Strings

To run LINK-80, type L80 followed by a carriage return. LINK-80 will return the prompt "*" (with the DTC operating system, the prompt is ">"), indicating it is ready to accept commands. Each command to LINK-80 consists of a string of filenames and switches separated by commas:

```
objdev1:filename.ext/switch1,objdev2:filename.ext,...
```

If the input device for a file is omitted, the default is the currently logged disk. If the extension of a file is omitted, the default is .REL. After each line is typed, LINK will load or search (see /S below) the specified files. After LINK finishes this process, it will list all symbols that remained undefined followed by an asterisk.

Example:

```
*MAIN

 DATA     0100    0200

 SUBR1*       (SUBR1 is undefined)

 DATA     0100    0300

*SUBR1
*/G           (Starts Execution - see below)
```

Typically, to execute a FORTRAN and/or COBOL program and subroutines, the user types the list of filenames followed by /G (begin execution). Before execution begins, LINK-80 will always search the system library (FORLIB.REL or COBLIB.REL) to satisfy any unresolved external references. If the user wishes to first search libraries of his own, he should append the filenames that are followed by /S to the end of the loader command string.

## 2.1.2 LINK-80 Switches

A number of switches may be given in the LINK-80 command string to specify actions affecting the loading process. Each switch must be preceded by a slash (/). These switches are:

Switch Action

R

Reset. Put loader back in its initial state. Use /R if you loaded the wrong file by mistake and want to restart. /R takes effect as soon as it is encountered in a command string.

E or E:Name

Exit LINK-80 and return to the Operating System. The system library will be searched on the current disk to satisfy any existing undefined globals. The optional form E:Name (where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program. Use /E to load a program and exit back to the monitor.

G or G:Name

Start execution of the program as soon as the current command line has been interpreted. The system library will be searched on the current disk to satisfy any existing undefined globals if they exist. Before execution actually begins, LINK-80 prints three numbers and a BEGIN EXECUTION message. The three numbers are the start address, the address of the next available byte, and the number of 256-byte pages used. The optional form G:Name (where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program.

N

If a FILENAME>/N is specified, the program will be saved on disk under the selected name (with a default extension of .COM for CP/M) when a /E or /G is done. A jump to the start of the program is inserted if needed so the program can run properly (at 100H for CP/M).

12

P and D

/P and /D allow the origin(s) to be set for the next program loaded. /P and /D take effect when seen (not deferred), and they have no effect on programs already loaded. The form is /P:ADDRESS> or /D:ADDRESS>, where ADDRESS> is the desired origin in the current typeout radix. (Default radix for non-MITS versions is hex. /O sets radix to octal; /H to hex.) LINK-80 does a default /P:LINK origin>+3 (i.e., 103H for CP/M and 4003H for ISIS) to leave room for the jump to the start address.

NOTE: Do not use /P or /D to load programs or data into the locations of the loader's jump to the start address (100H to 102H for CPM and 2800H to 2802H for DTC), unless it is to load the start of the program there. If programs or data are loaded into these locations, the jump will not be generated.

If no /D is given, data areas are loaded before program areas for each module. If a /D is given, all Data and Common areas are loaded starting at the data origin and the program area at the program origin. Example:

```
*/P:200,FOO
Data     200    300
*/R
*/P:200 /D:400,FOO
Data     400    480
Program 200     280
```

U

List the origin and end of the program and data area and all undefined globals as soon as the current command line has been interpreted. The program information is only printed if a /D has been done. Otherwise, the program is stored in the data area.

M

List the origin and end of the program and data area, all defined globals and their values, and all undefined globals followed by an asterisk. The program information is only printed if a /D has been done. Otherwise, the program is stored in the data area.

S — Search the filename immediately preceding the /S in the command string to satisfy any undefined globals.

Examples:

```
*/M             List all globals

*MYPROG,SUBROT,MYLIB/S
                Load MYPROG.REL and SUBROT.REL and
                then search MYLIB.REL to satisfy
                any remaining undefined globals.

*/G             Begin execution of main program
```

## 2.2 Sample Link

```
A>L80
*EXAMPL,EXMPL1/G
DATA    3000    30AC
[304F   30AC    49]
[BEGIN EXECUTION]

          1792          14336
         14336         -16383
        -16383             14
            14            112
           112            896
A>
```

## 2.3 Format of LINK Compatible Object Files

NOTE

Section 2.3 is reference material for users who wish to know the load format of LINK-80 relocatable object files. Most users will want to skip this section, as it does not contain material necessary to the operation of the package.

LINK-compatible object files consist of a bit stream. Individual fields within the bit stream are not aligned on byte boundaries, except as noted below. Use of a bit stream for relocatable object files keeps the size of object files to a minimum, thereby decreasing the number of disk reads/writes.

12

There are two basic types of load items: Absolute and Relocatable. The first bit of an item indicates one of these two types. If the first bit is a 0, the following 8 bits are loaded as an absolute byte. If the first bit is a 1, the next 2 bits are used to indicate one of four types of relocatable items:

00 Special LINK item (see below).

01 Program Relative. Load the following 16 bits after adding the current Program base.

10 Data Relative. Load the following 16 bits after adding the current Data base.

11 Common Relative. Load the following 16 bits after adding the current Common base.

Special LINK items consist of the bit stream 100 followed by:

a four-bit control field

an optional A field consisting of a two-bit address type that isthe same as the two-bit field above except 00 specifies absolute address

an optional B field consisting of 3 bits that give a symbol length and up to 8 bits for each character of the symbol

A general representation of a special LINK item is:

```
1 00xxxx  yy nn    zzz + characters of symbol name
          --------  -------------------------------
          A field             B field
```

| | |
|---|---|
| xxxx | Four-bit control field (0-15 below) |
| yy | Two-bit address type field |
| nn | Sixteen-bit value |
| zzz | Three-bit symbol length field |

The following special types have a B-field only:

| | |
|---|---|
| 0 | Entry symbol (name for search) |
| 1 | Select COMMON block |
| 2 | Program name |

3 Request library search
4 Reserved for future expansion

The following special LINK items have both an A field and a B field:

5 Define COMMON size
6 Chain external (A is head of address chain, B is name of external symbol)
7 Define entry point (A is address, B is name)
8 Reserved for future expansion

The following special LINK items have an A field only:

9 External + offset. The A value will be added to the two bytes starting at the current location counter immediately before execution.
10 Define size of Data area (A is size)
11 Set loading location counter to A
12 Chain address. A is head of chain, replace all entries in chain with current location counter.
The last entry in the chain has an address field of absolute zero.
13 Define program size (A is size)
14 End program (forces to byte boundary)

The following special Link item has neither an A nor a B field:

15 End file

## 2.4 LINK-80 Error Messages

LINK-80 has the following error messages:

?No Start Address A /G switch was issued, but no main program had been loaded.

?Loading Error The last file given for input was not a properly formatted LINK-80 object file.

?Out of Memory Not enough memory to load program.

?Command Error Unrecognizable LINK-80 command.

?<file> Not Found <file>, as given in the command string, did not exist.

```
%2nd COMMON Larger /XXXXXX/
```

The first definition of COMMON block /XXXXXX/ was not the largest definition. Reorder module loading sequence or change COMMON block definitions.

```
%Mult. Def. Global YYYYYY
```

More than one definition for the global (internal) symbol YYYYYY was encountered during the loading process.

```
%Overlaying ┌Program┐ Area  ,Start = xxxx
            └Data   ┘       ,Public = <symbol name>(xxxx)
                            ,External = <symbol name>(xxxx)
```

A /D or /P will cause already loaded data to be destroyed.

```
?Intersecting ┌Program┐ Area
              └Data   ┘
```

The program and data area intersect and an address or external chain entry is in this intersection. The final value cannot be converted to a current value since it is in the area intersection.

```
?Start Symbol - <name> - Undefined
```

After a /E: or /G: is given, the symbol specified was not defined.

```
Origin ┌Above┐ Loader Memory, Move Anyway (Y or N)?
       └Below┘
```

After a /E or /G was given, either the data or program area has an origin or top which lies outside loader memory (i.e., loader origin to top of memory). If a Y <cr> is given, LINK-80 will move the area and continue. If anything else is given, LINK-80 will exit. In either case, if a /N was given, the image will already have been saved.

```
?Can't Save Object File
```

A disk error occurred when the file was being saved.

## 2.5 Program Break Information

LINK-80 stores the address of the first free location in a global symbol called $MEMRY if that symbol has been defined by a program loaded. $MEMRY is set to the top of the data area +1.

NOTE

> If -D is given and the data origin is less than the program area, the user must be sure there is enough room to keep the program from being destroyed. This is particularly true with the disk driver for FORTRAN-80 which uses $MEMRY to allocate disk buffers and FCB's.

# SECTION 3

## LIB-80 Library Manager (CP/M Versions Only)

LIB-80 is the object time library manager for CP/M versions of FORTRAN-80 and COBOL-80. LIB-80 will be interfaced to other operating systems in future releases of FORTRAN-80 and COBOL-80.

### 3.1 LIB-80 Commands

To run LIB-80, type LIB followed by a carriage return. LIB-80 will return the prompt "*" (with the DTC operating system, the prompt is ">"), indicating it is ready to accept commands. Each command in LIB-80 either lists information about a library or adds new modules to the library under construction.

Commands to LIB-80 consists of an optional destination filename which sets the name of the library being created, followed by an equal sign, followed by module names separated by commas. The default destination filename is FORLIB.LIB. Examples:

```
*NEWLIB=FILE1 <MOD2>, FILE3,TEST

*SIN,COS,TAN,ATAN
```

Any command specifying a set of modules concatenates the modules selected onto the end of the last destination filename given. Therefore,

```
*FILE1,FILE2 <BIGSUB>, TEST
```

isequivalent to

```
 *FILE1
 *FILE2 <BIGSUB>
 *TEST
```

### 3.1.1 Modules

A module is typically a FORTRAN or COBOL subprogram, main program or a MACRO-80 assembly that contains ENTRY statements.

The primary function of LIB-80 is to concatenate modules in .REL files to form a new library. In order to extract modules from previous libraries or .REL files, a powerful syntax has been devised to specify ranges of modules within a .REL file.

The simplest way to specify a module within a file is simply to use the name of the module. For example:

```
 SIN
```

But a relative quantity plus or minus 255 may also be used. For example:

```
SIN+1
```

specifies the module after SIN and

```
SIN-1
```

specifies the one before it.

Ranges of modules may also be specified by using two dots:

..SIN means all modules up to and including SIN.

SIN.. means all modules from SIN to the end of the file.

SIN..COS means SIN and COS and all the modules in between.

Ranges of modules and relative offsets may also be used in combination:

```
SIN+1..COS-1
```

To select a given module from a file, use the name of the file followed by the module(s) specified enclosed in angle brackets and separated by commas:

```
FORLIB <SIN..COS>
```

or

```
MYLIB.REL <TEST>
```

or

```
BIGLIB.REL <FIRST,MIDDLE,LAST>
```

etc.

If no modules are selected from a file, then *all*

12

the modules in the file are selected:

```
TESTLIB.REL
```

## 3.2 LIB-80 Switches

A number of switches are used to control LIB-80 operation. These switches are always preceded by a slash:

/O Octal - set Octal typeout mode for /L command.

/H Hex - set Hex typeout mode for /L command (default).

/U List the symbols which would remain undefined on a search through the file specified.

/L List the modules in the files specified and symbol definitions they contain.

/C (Create) Throw away the library under construction and start over.

/E Exit to CP/M. The library under construction (.LIB) is revised to .REL and any previous copy is deleted.

/R Rename - same as /E but does not exit to CP/M on completion.

## 3.3 LIB-80 Listings

To list the contents of a file in cross reference format, use /L:

```
*FORLIB/L
```

When building libraries, it is important to order the modules such that any intermodule references are "forward." That is, the module containing the global reference should physically appear ahead of the module containing the entry point. Otherwise, LINK-80 may not satisfy all global references on a single pass through the library.

Use /U to list the symbols which could be undefined in a single pass through a library. If a module in the library makes a backward reference to a symbol in another module, /U will list that symbol. Example:

```
*SYSLIB/U
```

NOTE: Since certain modules in the standard FORTRAN and COBOL systems are always force-loaded, they will be listed as undefined by /U but will not cause a problem when loading FORTRAN or COBOL programs.

Listings are currently always sent to the terminal; use control-P to send the listing to the printer.

## 3.4 Sample LIB Session

```
A>LIB

*TRANLIB=SIN,COS,TAN,ATAN,ALOG
*EXP
*TRANLIB.LIB/U
*TRANLIB.LIB/L
  .
  .
  .
(List of symbols in TRANLIB.LIB)
  .
  .
  .
*/E
A>
```

## 3.5 Summary of Switches and Syntax

```
/O  Octal - set listing radix
/H  Hex - set listing radix
/U  List undefineds
/L  List cross reference
/C  Create - start LIB over
/E  Exit - Rename .LIB to .REL and exit
/R  Rename - Rename .LIB to .REL
```

```
module::=module name {+ or - number}

module sequence ::=

module | ..module | module.. | module1..module2

file specification::=filename {<module sequence> {,<module sequence

command::= {library filename=}  {list of file specifications}
    {list of switches}
```

# SECTION 4

## Operating Systems

This section describes the use of MACRO-80 and LINK-80 under the different disk operating systems. The examples shown in this section assume that the FORTRAN-80 compiler is in use. If you are using the COBOL-80 compiler, substitute "COBOL" wherever "F80" appears, and substitute the extension ".COB" wherever ".FOR" appears.

## 4.1 CPM

### Create a Source File

Create a source file using the CPM editor. Filenames are up to eight characters long, with 3-character extensions. FORTRAN-80 source filenames should have the extension FOR, COBOL-80 source filenames should have the extension COB, and MACRO-80 source filenames should have the extension MAC.

### Compile the Source File

Before attempting to compile the program and produce object code for the first time, it is advisable to do a simple syntax check. Removing syntax errors will eliminate the necessity of recompiling later. To perform the syntax check on a source file called MAX1.FOR, type

```
A>F80 ,=MAX1
```

This command compiles the source file MAX1.FOR without producing an object or listing file. If necessary, return to the editor and correct any syntax errors.

To compile the source file and produce an object and listing file, type

```
A>F80 MAX1,MAX1=MAX1
```

or

```
A>F80 =MAX1/L
```

The compiler will create a REL (relocatable) file called MAX1.REL and a listing file called MAX1.PRN.

### Loading, Executing and Saving the Program (Using LINK-80)

To load the program into memory and execute it, type

```
A>L80 MAX1/G
```

To exit LINK-80 and save the memory image (object code), type

```
A>L80 MAX1/E,MAX1/N
```

When LINK-80 exits, three numbers will be printed: the starting address for execution of the program, the end address of the program and the number of 256-byte pages used. For example

```
[210C 401A 48]
```

If you wish to use the CPM SAVE command to save a memory image, the number of pages used is the argument for SAVE. For example

```
A>SAVE 48 MAX1.COM
```

NOTE

> CP/M always saves memory starting at 100H and jumps to 100H to begin execution. Do not use /P or /D to set the origin of the program or data area to 100H, unless program execution will actually begin at 100H.

An object code file has now been saved on the disk under the name specified with /N or SAVE (in this case MAX1). To execute the program simply type the program name

```
A>MAX1
```

### CPM - Available Devices

| | |
|---|---|
| A:, B:, C:, D: | disk drives |
| HSR: | high speed reader |
| LST: | line printer |
| TTY: | Teletype or CRT |

### CPM Disk Filename Standard Extensions

| | |
|---|---|
| FOR | FORTRAN-80 source file |
| COB | COBOL-80 source file |
| MAC | MACRO-80 object file |
| REL | relocatable object file |
| PRN | listing file |
| COM | absolute file |

### CPM Command Lines

CPM command lines and files are supported; i.e., a COBOL-80, FORTRAN-80, MACRO-80 or LINK-80 command line may be placed in the same line with the CPM run command. For example, the command

```
A>F80 =TEST
```

causes CPM to load and run the FORTRAN-80 compiler, which then compiles the program TEST.FOR and creates the file TEST.REL. This is equivalent to the following series of commands:

```
A>F80
*=TEST
*^C
A>
```

## 4.2 DTC Microfile

### Create a Source File

Create a source file using the DTC editor. Filenames are up to five characters long, with 1-character extensions. COBOL-80, FORTRAN-80 and MACRO-80 source filenames should have the extension T.

### Compile the Source File

Before attempting to compile the program and produce object code for the first time, it is advisable to do a simple syntax check. Removing syntax errors will eliminate the necessity of recompiling later. To perform the syntax check on the source file called MAX1,type

```
*F80 ,=MAX1
```

This command compiles the source file MAX1 without producing an object or listing file. If necessary, return to the editor and correct any syntax errors.

To compile the source file MAX1 and produce an object and listing file, type

```
      *F80 MAX1,MAX1=MAX1
 or
      *F80 =MAX1/L/R
```

The compiler will create a relocatable file called MAX1.O and a listing file called MAX1.L.

### Loading, Executing and Saving the Program (Using LINK-80)

To load the program into memory and execute it, type

```
*L80 MAX1/G
```

To save the memory image (object code), type

```
*L80 MAX1/E
```

which will exit from LINK-80, return to the DOS monitor and print three numbers: the starting address for execution of the program, the end address of the program, and the number of 256-byte pages used. For example

```
[210C 401A 48]
```

Use the DTC SAVE command to save a memory image. For example

```
*SA MAX1 2800 401A 2800
```

2800H (24000Q) is the load address used by the DTC Operating System.

NOTE

> If a /P:ADDRESS> or /D:ADDRESS> has been included in the loader command to specify an origin other than the default (2800H), make sure the low address in the SAVE command is the same as the start address of the program.

An object code file has now been saved on the disk under the name specified in the SAVE command (in this case MAX1). To execute the program, simply type

```
*RUN MAX1
```

12

### DTC Microfile - Available Devices

| | |
|---|---|
| DO:, D1:, D2:, D3: | disk drives |
| TTY: | Teletype or CRT |
| LIN: | communications port |

### DTC Disk Filename Standard Extensions

| | |
|---|---|
| T | COBOL-80, FORTRAN-80 or MACRO-80 source file |
| O | relocatable object file |
| L | listing file |

DTC Command Lines
DTC command lines are supported as described in Section 4.1, CPM Command Lines.

## 4.3 Altair DOS

Create a Source File
Create a source file using the Altair DOS editor. The name of the file should have four characters, and the first character must be a letter. For example, to create a file called MAX1, initialize DOS and type

```
.EDIT MAX1
```

The editor will respond

```
CREATING FILE
00100
```

Enter the program. When you are finished entering and editing the program, exit the editor.

Compile the Source File
Load the compiler by typing

```
.F80
```

The compiler will return the prompt character "*".

Before attempting to compile the program and produce object code for the first time, it is advisable to do a simple syntax check. Removing syntax errors will eliminate the necessity of recompiling later. To perform the syntax check on the source file called MAX1, type

```
*,=&MAX1.
```

(The editor stored the program as &MAX1) Typing ,=&MAX1. compiles the source file MAX1 without producing an object or listing file. If necessary, return to the editor and correct any syntax errors.

To compile the source file MAX1 and produce an object and listing file, type

```
*MAX1R,&MAX1=&MAX1.
```

The compiler will create a REL (relocatable) file called MAX1RREL and a listing file called &MAX1LST. The REL filename must be entered as five characters instead of four, so it is convenient to use the source filename plus R.

After the source file has been compiled and a prompt has been printed, exit the compiler. If the computer uses interrupts with the terminal, type Control C. If not, actuate the RESET switch on the computer front panel. Either action will return control to the monitor.

Using LINK-80

Load LINK-80 by typing

```
.L80
```

LINK-80 will respond with a "*" prompt. Load the program into memory by entering the name of the program REL file

```
*MAX1R
```

Executing and Saving the Program

Now you are ready to either execute the program that is in memory or save a memory image (object code) of the program on disk. To execute the program, type

```
*/G
```

To save the memory image (object code), type

```
*/E
```

which will exit from LINK-80, return to the DOS monitor and print three numbers: the starting address for execution of the program, the end address of the program, and the number of 256-byte pages used. For example

```
[26301 44054 35]
```

Usethe DOS SAVE command to save a memory image. Type

```
.SAV MAX1 0 17100 44054 26301
```

17100 is the load address used by Altair DOS for the floppy disk. (With the hard disk, use 44000.)

An object code file has now been saved on the disk under the name specified in the SAVE command (in this case MAX1). To execute the program, simply type the program name

```
.MAX1
```

12

### Altair DOS - Available Devices

| | |
|---|---|
| FO:, F1:, F2:, ... | disk drives |
| TTY: | Teletype or CRT |

### Altair DOS Disk Filename Standard Extensions

| | |
|---|---|
| FOR | FORTRAN-80 source file |
| COB | COBOL-80 source file |
| MAC | MACRO-80 source file |
| REL | relocatable object file |
| LST | listing file |

### Command Lines

Command lines are not supported by Altair DOS.

## 4.4 ISIS-II

### Create a Source File

Create a source file using the ISIS-II editor. Filenames are up to six characters long, with 3-character extensions. FORTRAN-80 source filenames should have the extension FOR and COBOL-80 source filenames should have the extension COB. MACRO-80 source filenames should have the extension MAC.

### Compile the Source File

Before attempting to compile the program and produce object code for the first time, it is advisable to do a simple syntax check. Removing syntax errors will eliminate the necessity of recompiling later. To perform the syntax check on the source file called MAX1.FOR, type

```
-F80 ,=MAX1
```

This command compiles the source file MAX1.FOR without producing an object or listing file. If necessary, return to the editor and correct any syntax errors.

To compile the source file MAX1.FOR and produce an object and listing file, type

```
-F80 MAX1,MAX1=MAX1
```

or

```
-F80 =MAX1/L/R
```

The compiler will create a REL (relocatable) file called MAX1.REL and a listing file called MAX1.LST.

### Loading, Saving and Executing the Program (Using LINK-80)

To load the program into memory and execute it, type

```
-L80 MAX1/G
```

To save the memory image (object code), type

```
-L80 MAX1/E,MAX1/N
```

which will exit from LINK-80, return to the ISIS-II monitor and print three numbers: the starting address for execution of the program, the end address of the program, and the number of 256-byte pages used. For example

```
[210C 401A 48]
```

An object code file has now been saved on the disk under the name specified with /N (in this case MAX1).

### ISIS-II - Available Devices

| | |
|---|---|
| :FO:, :F1:, :F2:, ... | disk drives |
| TTY: | Teletype or CRT |
| LST: | line printer |

### ISIS-II Disk Filename Standard Extensions

| | |
|---|---|
| FOR | FORTRAN-80 source file |
| COB | COBOL-80 source file |
| MAC | MACRO-80 source file |
| REL | relocatable object file |
| LST | listing file |

### ISIS-II Command Lines

ISIS-II command lines are supported as described in Section 4.1, CPM Command Lines.

12

# Index

# SERVICE INFORMATION

Page East

1st Stop - Schatz Rd
left turn

3 or 4 Blocks - Stop Sign

Adie Rd - left on

1 block to Centerline - Turn Right, then

left on Corner
#2405

St. Louis Info Sys
314432-3181

Westport
Centerline Dr.

modules - Video - $75
Pow Sup - $75
CPU - $225
Drives - $150

## SERVICING PROCEDURES

Your SuperBrain Video Terminal is warranted to the original purchaser for 90 days from date of shipment. This warranty covers the adjustment or replacement F.O.B. Intertec's plant in Columbia, South Carolina of any part or parts which in Intertec's judgment shall disclose to have been originally defective. A complete statement of your warranty rights is contained on the inside back cover of this manual.

In order to validate your SuperBrain warranty, the Warranty Registration Form (contained in this section) must be completed in full and returned to Intertec Data Systems within 10 days of receipt of this equipment. Be sure to include the serial number of the specific terminal you are registering. The serial number of your terminal can be found on the rear I/O panel next to the power cord. A Customer Comment Card is also enclosed for your convenience if you desire to make comments regarding the overall operation and/or adaptability of the SuperBrain to your particular application. Completion of the Customer Comment Card is optional.

**IF SERVICE IS EVER REQUIRED:**
If you should encounter difficulties with the use or operation of this terminal, contact the supplier from whom the unit was purchased for instructions regarding the proper servicing techniques. Service procedures differ from dealer to dealer but most Intertec authorized service dealers can provide local, on-site servicing of this equipment on a per-call or maintenance contract basis. Plus, a variety of service programs are available directly from the factory including extended warranty, a module exchange program and on-site contract maintenance from over 50 locations in the U.S. Contact our National Service Department at the factory for rates and availability if you desire to participate in one of these programs. If you are not covered under one of the three programs described above and service cannot be made available through your local supplier, contact Intertec's Customer Service Department at (803) 798-9100. Be prepared to give the following information when you call:

1) The serial number of the equipment which is defective. If you are returning individual modules to the factory for repair, it will be necessary to have the serial number of the individual modules also. The serial number of the entire terminal may be found on the rear I/O panel just to the right of the power cord. Module serial numbers are listed on white stickers placed in conspicuous locations on each major module or subassembly of the terminal.

   NOTE: Individual modules cannot be returned to the factory for repair unless you originally purchased your unit directly from the factory. If your unit was purchased through a Dealer or OEM vendor, and you desire factory repair, then the entire terminal must be returned.
2) The name and location of the Dealer and/or Agent from which the unit was purchased.
3) A complete description of the alleged failure (including the nature and cause of the failure if readily available).

The Customer Service Department will issue you Return Material Authorization Number (RMA Number) which will be valid for a period of 30 days. This RMA Number will be your official authorization to return equipment to IDSC for repair only. The Customer Service Department will also give you an estimate, if requested, of the time it should take to process and repair your equipment. Turnaround time on repairs varies depending on workloads and availability of parts but normally your equipment will be repaired and returned to you within 10 working days of its receipt. If your repair is urgent, you may authorize a special $50 Emergency Repair fee and have your equipment repaired and returned within no more than 48 hours of its receipt at our Service Center. Ask the Customer Service Department for more information about this program.

## SERVICING PROCEDURES (continued)

IMPORTANT: Any equipment returned to Intertec without an RMA Number will result in the equipment being refused and possible cancellation of your SuperBrain warranty. Also if your RMA Number expires, you must request a new number. Equipment arriving at Intertec bearing expired RMA Number will also be refused.

After securing an RMA Number from the Customer Service Department, return the specified modules and/or complete terminals to Intertec, freight prepaid, at the address below. NOTE: The RMA Number must be plainly marked and visible on your shipping label to prevent the equipment from being refused at Intertec's Receiving Department.

ATTN: SUPERBRAIN SERVICE CENTER
Intertec Data Systems Corporation
2300 Broad River Road
Columbia, South Carolina 29210

To aid our technicians in troubleshooting and correcting your reported malfunction, please complete an Intertec Equipment Malfunction Report (contained in this section) and enclose it with the equipment you intend to return to the factory.

Be sure a declared value equal to the price of the unit is shown on the Bill of Lading, Express Receipt of Air Freight Bill, whichever is applicable. Risk of loss or damage to equipment during the time it is in transit either to or from Intertec's facilities is your sole responsibility. A declared value must be placed on your Bill of Lading to insure substantiation of your freight claim if shipping damage or loss is incurred.

All equipment returned to an Intertec Service Center **must** be freight prepaid. Equipment not prepaid on arrival at Intertec's Receiving Department cannot be accepted. Upon repair of the defective equipment, it will be returned to you, F.O.B. the factory in Columbia, via UPS or equivalent ground transportation unless you specify otherwise.

## INSTRUCTIONS FOR HANDLING LOST OR DAMAGED EQUIPMENT

The goods described on your Packing Slip were delivered to the Transportation Company at Intertec's premises in complete and good condition. If any of the goods called for on this Packing Slip are short or damaged, you must file a claim WITH THE TRANSPORTATION COMPANY FOR THE AMOUNT OF THE DAMAGE AND/OR LOSS.

### IF LOSS OR DAMAGE IS EVIDENT AT TIME OF DELIVERY:

If any of the goods called for on your Packing Slip are short or damaged at the time of delivery, ACCEPT THEM, **but insist that the Freight Agent make a damaged or short notation on your Freight Bill or Express Receipt and sign it.**

### IF DAMAGE OR LOSS IS CONCEALED AND DISCOVERED AT A LATER DATE:

If any concealed loss or damage is discovered, notify your local Freight Agent or Express Agent AT ONCE and request him to make an inspection. This is absolutely necessary. Unless you do this, the Transportation Company will not consider your claim for loss or damage valid. If the agent refuses to make an inspection, you should draw up an affidavit to the effect that you notified him on a certain date and that he failed to make the necessary inspection.

After you have ascertained the extent of the loss or damage, **ORDER THE REPLACEMENT PARTS OR COMPLETE NEW UNITS FROM THE FACTORY.** We will ship to you **and bill you for the cost.** This new invoice will then be a part of your claim for reimbursement from the Transportation Company. This together with other papers, will properly support your claim.

## SERVICING PROCEDURES (continued)

IMPORTANT: The claims adjustment procedure for UPS shipments varies somewhat from the procedure listed above for regular motor and air freight shipments. If your equipment was shipped via UPS and sustained either damage or loss, the UPS representative in your area **must** initiate the claim by inspecting the goods and assigning a freight claim number to the damaged equipment. The representative will attach a "Call Tag" to the outside of the equipment box which will be your authorization to return the merchandise to our factory for claim adjustment. Upon receipt of this damaged equipment, we will perform the necessary repairs, process the appropriate paperwork with UPS and return the equipment to you. Please allow time for processing of any type claim. Normal time for proper processing of a UPS claim is 15-30 working days.

Remember, it is extremely important that you **do not give the Transportation Company a clear receipt if damage or shortages are evident upon delivery.** It is equally important that you call for an inspection if the loss or damage is discovered later. DO NOT, UNDER ANY CIRCUMSTANCES, ORDER THE TRANSPORTATION COMPANY TO RETURN SHIPMENT TO OUR FACTORY OR REFUSE SHIPMENT UNLESS WE HAVE AUTHORIZED SUCH RETURN.

## ADDITIONAL TECHNICAL DOCUMENTATION

Detailed technical documentation (i.e. schematics) describing the operation of the SuperBrain Video Terminal and the electrical interconnection of its various modules is available at nominal cost directly from Intertec Data Systems Corporation. However, due to the confidentiality of this technical information, it will be necessary to sign and return the Documentation Non-Disclosure Agreement (appearing on the next page) denoting your concurrence with its terms and conditions.

The handling and processing costs of SuperBrain technical documentation is $50. Due to the large amount of requests being processed and the relatively small handling costs involved, we **must** request that you **enclose payment** ($50) upon return of your Non-Disclosure Agreement. Normally the documents will be mailed to you within 15 to 30 days after receipt of your payment and a signed copy of the Agreement. (IMPORTANT: The technical documentation will be mailed to the address listed at the top of the Non-Disclosure Agreement.) For prompt processing of your documentation request, please forward your signed agreement and payment to:

Customer Service Department<br>
Intertec Data Systems Corporation<br>
2300 Broad River Road<br>
Columbia, South Carolina 29210

NOTE: Formal technical documentation for the SuperBrain will be sent to you normally within 10-15 days of receipt of your payment and signed Non-Disclosure agreement.

IMPORTANT: Payment **must** accompany your Non-Disclosure Agreement. Agreements sent to us without payment will be discarded without notice.

# HARDWARE ADDENDUMS

14

INTERTEC DATA SYSTEMS

® Corporate Headquarters: 2300 Broad River Road, Columbia, South Carolina 29210 • 803/798-9100 • TWX: 810-666-2115

# ENGINEERING CHANGE ORDER

IDS - 910A

SUBJECT Expanding SuperBrain Memory Size From 32K to 64K on Revision 1 CPU Modules
NOTE: This ECO is for Revision 1 CPU Modules only! Refer to ECO #119001 for instructions for Revision 0 CPU Modules.

PRODUCT SuperBrain DATE January, 1980 ECO # 010004 PAGE 1 OF 1

ASSEMBLY NAME/NUMBER Keyboard/CPU Module

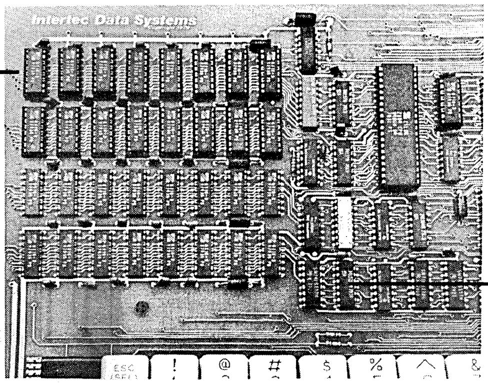
## BACKGROUND AND IMPLEMENTATION INFORMATION:

Standard SuperBrain terminals are supplied with 32K dynamic RAM but can be expanded to 64K. The instructions below detail the proper installation of the optional 32K RAM expansion kit.

## INSTALLATION:

1) Remove cover and locate RAM bank at upper left corner of Keyboard/CPU Module. (See Figure 1)
2) Install all sixteen RAM chips in the two upper rows of eight sockets each being careful to notice that all the chips are inserted correctly. (NOTE: The notch on each chip should be pointing toward the top of the board.)
3) After all sixteen RAM chips have been installed, find the small bare wire jumper located between the two chips designated '74LS157' and '74LS155'. (See Figure 1)
4) Cut the LEFT end of the jumper (end closest to the '74LS157') and reconnect it to the pad just to the RIGHT of the other end (the two pads are approximately 0.1" apart). Installation of the additional 32K is now complete.

**Figure 1 - Location of SuperBrain RAM Bank**



Install extra 32K of RAM into these 16 sockets (8 sockets each row) 16 RAM chips are required to enable the extra 32K memory.

Remove and reconnect jumper located in this area

## OPERATION:

To operate the SuperBrain with 64K, insert a DOS diskette (configured for 64K) into drive A. The sign-on message must read as follows:

```
64K        SUPERBRAIN        DOS        VER
A
```

IMPORTANT: Do not attempt to operate the unit with a DOS Diskette configured for 32K. It will not work properly in a 64K machine. The program which allows for configuration of the SuperBrain in 32 or 64K of RAM is entitled 'CPM6420.COM' and is contained on your CP/M DOS Diskette. The CP/M program MOVCPM, which reconfigures the size of the Disk Operating System, is not supplied on the DOS diskette. Intertec offers only two RAM configurations - 32K and 64K - so there should be no need to reconfigure the operating system to any other size.

*See reverse side for additional information*

THIS ENGINEERING CHANGE ORDER AFFECTS:

□MATERIAL(S)/COMPONENTS(S) USED □PACKAGING/SHIPPING □OTHER

□PRODUCTION PROCEDURES ☒SERVICING/PROCEDURES

THIS CHANGE PREVENTS: N/A

CHANGE FROM:

32K SuperBrain Operation

CHANGE TO:

64K SuperBrain Operation

INITIATED BY: DEPARTMENT: Product Services APPROVED BY:

| | |
|---|---|
| THIS ECO DISTRIBUTED TO: | KIT AND ORDERING INFORMATION |
| □ ENGINEERING | KIT AVAILABLE? □ YES □ NO |
| □ OPERATIONS | KIT NUMBER Order by description |
| □ QUALITY ASSURANCE | PRICING: $350 for additional 32K RAM |
| □ SHIPPING & RECEIVING | F.O.B. factory, Columbia, S.C. |
| ☒ CUSTOMER SERVICE | |
| ☒ MARKETING | |
| ☒ FIELD SERVICE | CONTACT THE CUSTOMER SERVICE DEPARTMENT AT THE NUMBER AND ADDRESS ON REVERSE SIDE TO OBTAIN FURTHER INFORMATION AND/OR TO ORDER THIS KIT. |
| □ CUSTOMER LIST | |
| ☒ CUSTOMER AS REQUESTED | |

# SOFTWARE ADDENDUMS

15

INTERTEC
DATA
SYSTEMS®
Corporate Headquarters: 2300 Broad River Road, Columbia, South Carolina 29210 • 803/798-9100 • TWX: 810-666-2115

## SUPERBRAIN DOS 3.0 DESCRIPTION

DOS 3.0 has two major differences from the previous versions of SUPERBRAIN DOS. First, DOS 3.0 incorporates CP/M 2.2 and secondly, the physical sector length of the diskette has been changed from 128 bytes/sector to 512 bytes/sector. An increased diskette capacity (40 kilobytes per diskette) is the result of these alterations.

The updated DOS requires the use of a VERSION 3.0 or higher PROM Bootloader. The version number can be easily verified by performing a system reset with no diskettes in the drives. The version number will be displayed in the sign-on message.

Also included on this diskette are four different operating systems to facilitate the copying of 128 byte/sector diskettes to 512 byte/sector diskettes. They are the following:

32CPM5/5.COM – 32K DOS; Disk A is 512 bytes/sector;
Disk B is 512 bytes/sector

32CPM5/1.COM – 32K DOS; Disk A is 512 bytes/sector;
Disk B is 128 bytes/sector

64CPM5/5.COM – 64K DOS; Disk A is 512 bytes/sector;
Disk B is 512 bytes/sector

64CPM5/1.COM – 64K DOS; Disk A is 512 bytes/sector;
Disk B is 128 bytes/sector

The distribution copy is initialized as a 64K system with both disk drives programmed to accept a 512 byte/sector diskette. This was done so a copy of the distribution diskette can be easily made before attempting to change operating systems.

**NOTE: THE STANDARD SUPERBRAIN IS SHIPPED WITH A 32K MEMORY. THEREFORE, A 32K DOS MUST BE GENERATED BEFORE PERFORMING ANY FILE MANIPULATIONS.**

### RECOMMENDED OPERATING PROCEDURES

To insure that the proper operating system for your SUPERBRAIN version is utilized, the following procedure should be performed. This procedure describes the generation of an operating system on a newly formatted diskette.

1) Insert a blank diskette into Disk B.
2) Format the diskette using the FORMAT30.COM program. Type 'FORMAT30' (Return)

**NOTE: DRIVE B CAN ONLY BE FORMATTED IF IT IS NOT DESIGNATED AS A 128 BYTE/SECTOR DRIVE.**

3) Select one of the two 512/512 operating systems and put it on Disk B.

**EXAMPLE:**

If you have a 32K system and you want to copy the distribution diskette, do the following:

Type '32CPM5/5' (RETURN)
System responds with
'SOURCE DRIVE NAME (OR RETURN TO SKIP)
Type RETURN
System responds with
DESTINATION DRIVE NAME (OR RETURN TO REBOOT)
Type 'B'
System responds with
DESTINATION ON B, THEN TYPE RETURN
Type RETURN
When function is complete, type RETURN.

4) Remove the diskettes and interchange them in the drives.
5) Do a system reset. The system should sign-on with the generated DOS message.
6) Copy the programs from Disk B to Disk A using the PIP program.

Now that a back-up copy has been generated, any one of the four operating systems can be put on disk A by following the above procedures and using Disk A as the destination.

To copy 128 byte/sector diskettes to a 512 byte/sector diskette, use either 64CPM5/1.COM or 32CPM5/1.COM. Put the 512 byte/sector diskette in Disk A and the 128 byte/sector diskette in Disk B.

## 32K BIOS PROGRAM

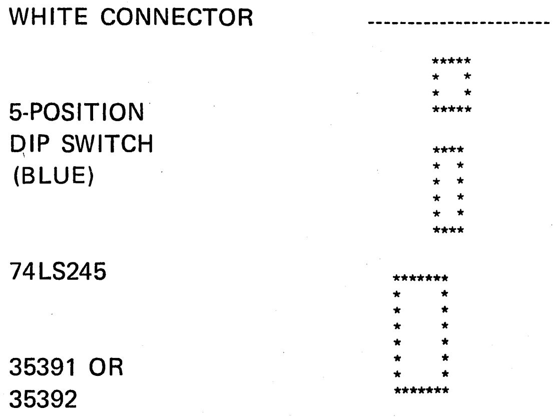
The BIOS portion of the DOS is supplied as a source program (32BS5/5.ASM) to facilitate the modification of the software drivers for peripheral devices. This program can be edited, assembled, and integrated into the DOS. Any extra routines should only be added in the designated area of the BIOS program.

## SOFTWARE/HARDWARE COMPATIBILITY

DOS 3.0 can only be operated on SUPERBRAIN units that have a REV-01 processor PC Board, and then only certain REV-01 boards quality. If your system does not have a REV-01 board, then DOS 3.0 cannot be used on that system. However, REV-01 boards that do not qualify can be factory retrofitted to support DOS 3.0.

To determine if your machine can support DOS 3.0 software, it is necessary to visually inspect the Processor board. This is done in the following manner:

1) Remove power from unit.
2) Close door on disk drives.
3) Remove cover by removing four screws. (2-front, 2-rear)
4) To determine if the board is at the correct revision level, the number 1532000-01 should be on the top right hand corner of the board and there should be two blue ribbon-cable connectors mounted on the board.
   If the board does not meet both conditions, it will not support DOS 3.0
5) If the board is REV-01, one more condition must be met. There should not be a 35391 or 35392 IC installed in the location shown in the diagram on the attached page. If an IC is present in that location and you would like to use DOS 3.0, contact the factory for pricing and scheduling.



Direct all inquiries to:

CUSTOMER SERVICE
INTERTEC DATA SYSTEMS
2300 BROAD RIVER ROAD
COLUMBIA, SC 29210

TELEPHONE: 803/798-9100

INTERTEC DATA SYSTEMS
® Corporate Headquarters: 2300 Broad River Road, Columbia, South Carolina 29210 • 803/798-9100 • TWX: 810-666-2115

**TECHNICAL BULLETIN**
IDS - 912A

DATE OF THIS RELEASE November, 1979 PAGE 1 OF 6 BULLETIN # B119004

ASSEMBLY NAME/NUMBER DOS DISKETTE PRODUCT SuperBrain

REFERENCE ECO # N/A DISTRIBUTED TO ______ APPROVED ______

## USING THE "INP:" AND "OUT:" FEATURES OF PIP TO FACILITATE FILE TRANSFERS TO AND FROM THE SUPERBRAIN

The SuperBrain is presently equipped with one RS-232-C serial interface port (labeled 'Main' on the rear panel). This interface is programmed for the following mode:

↑ Actually Aux. Port
This procedure for old Superbrains

Asynchronous
1200 Baud
8 bits
1 Stop Bit
No Parity

This port is also wired so that the SuperBrain appears as a processor rather than as a terminal. If it is to be used as a terminal, pins 2 and 3 in the RS-232-C cable must be interchanged.

Files can be transferred using the PIP program as described in Section 6.4 of the Operator's Manual entitled "An Introduction to CP/M Features and Facilities." When the SuperBrain transmits serial data, the destination is designated as a list (LST:) device; when receiving, the source device is considered a reader (RDR:).

The serial port may also be considered as an input (INP:) or output (OUT:) port. When used in this mode, the operator has the option of communicating to the sending/receiving device via the SuperBrain console before actual files are transferred.

Files transferred via the serial port must be in Intel hex format or ASCII. Binary files must be converted to hex files by utilizing the HEXGEN.ASM program before being sent to the SuperBrain. BASIC files must be saved in the ASCII format it they are to be transferred to the serial interface.

(NOTE: When ASCII files are transferred using the INP: or OUT: format, all data entered by the Operator on the console will also appear in the ASCII file. Undesired data must then be edited by using ED.COM).

Sequence of Operation:

1. Connect SuperBrain MAIN port to console input of host computer. Be sure host computer is set to 1200 baud.

2. The largest program that can be transferred by PIP is 25K. If programs are larger than 25K, then programs most be broken down into smaller segments 25K or smaller.

3. All commands must be entered on the SuperBrain in the following sequence:

DATE OF THIS RELEASE November, 1979 PAGE 2 OF 6 BULLETIN # B119004

ASSEMBLY NAME/NUMBER DOS DISKETTE PRODUCT SuperBrain

REFERENCE ECO # N/A DISTRIBUTED TO APPROVED

A. To transfer ASCII file - ABC. ASM - from SuperBrain to host:

```
A> PIP OUT: = ABC. ASM↲          (Keyboard entry)
ECHO (Y/N) Y                      (Computer responds)
                                  (Keyboard entry)
+                                 (Computer responds)
```

Now the SuperBrain will act like a dumb terminal for host computer. Any keyboard entry will be sent to host computer and displayed on screen.

```
+ PIP ABC.HST = CON:↲             (Keyboard entry)
                                  (Computer responds)
(CTRL) (B)                        (Keyboard entry - these two
                                   keys at the same time)
```

NOTE: Underlined characters are typed by customer.
"↲ " represents a carriage return.

Now the file is being transferred and should be displayed on the screen. When the file has been transferred the operating system will show the prompt symbol.

```
A> PIP OUT: = EOF:↲               (Keyboard entry)
ECHO  (Y/N) Y                     (Computer responds)
                                  (Keyboard entry)
+                                 (Computer responds)
(CTRL) (B)                        (Keyboard entry - these two
                                   keys at the same time)
```

Now the file transfer has been completed; both computers should return to the operating system.

B. To transfer binary file - ABC.COM - from SuperBrain to host:

```
A> PIP ABC.TST = INP:↲            (Keyboard entry)
ECHO  (Y/N) Y                     (Computer responds)
                                  (Keyboard entry)
+                                 (Computer responds)
```

Now the SuperBrain will act like a dumb terminal for the host computer. Any keyboard entry will be sent to the host computer and displayed on the screen.

```
+ PIP ABC.HEX = CON:↲             (Keyboard entry)
```

DATE OF THIS RELEASE November, 1979 PAGE 3 OF 6 BULLETIN # B119004

ASSEMBLY NAME/NUMBER DOS DISKETTE PRODUCT SuperBrain

REFERENCE ECO # N/A DISTRIBUTED TO APPROVED

NOTE: The binary file on the SuperBrain will be transferred in INTEL HEX format. After the transfer use LOAD or DDT and SAVE to change. HEX file to a binary, COM file.

```
(CTRL) (Z)                        (Keyboard entry - these two
                                   keys at the same time)
End of file, Control Z?           (Computer responds)
(CTRL) (Z)                        (Keyboard entry - these
                                   two keys)
```

Now the host computer is set up to input a file. The SuperBrain will return to the operating system with its prompt.

```
A> HEXDUMP ABC.COM ↲               (Keyboard entry)
```

At this point the file will be transferred in HEX format and displayed on the screen. When the transfer is complete the SuperBrain will return to the operating system.

C. To transfer ASCII file - ABC.PRN - to SuperBrain from host:

```
A> PIP ABC.PRN = INP: ↲            (Keyboard entry)
ECHO  (Y/N) Y ↲                    (Computer responds)
                                   (Keyboard entry)
+                                  (Computer responds)
```

Now the SuperBrain is ready for input from host. The keyboard entry will be sent to the host and displayed on the screen. Now set up commands to output from the host.

```
+ PIP CON: = ABC.PRN ↲             (Keyboard entry)
```

The file ABC.PRN on the host is now being input to the SuperBrain and displayed on the screen. After the file has been transferred, the SuperBrain should return to the operating system; if it does not, then type (CTRL) (Z) simultaneously.

D. To transfer binary file - ABC.COM - to SuperBrain from host:

(NOTE: Before transferring to SuperBrain, either HEXGEN.ASM or HEXDUMP.COM must be transferred to the host.)

~~NOTE~~

~~NOTE HEXDUMP ONLY PRINTS OUT A LISTING OF THE DESIGNATED FILE IN ASCII (ie, THE ASCII equivalent of binary)~~

Contact the Customer Service Department at the address above for additional information on this bulletin.

|  |  |  |  |
|---|---|---|---|
| DATE OF THIS RELEASE November, 1979 | PAGE 4 OF 6 | BULLETIN # B119004 | |
| ASSEMBLY NAME/NUMBER DOS DISKETTE | | PRODUCT SuperBrain | |
| REFERENCE ECO# N/A | DISTRIBUTED TO | APPROVED | |

1) Using HEXDUMP. COM

```
A> PIP ABC.HEX = INP: [H]             (Keyboard entry)
ECHO  (Y/N)  Y                        (Computer responds)
                                      (Keyboard entry)
+                                     (Computer responds)
```

Now the SuperBrain ready to accept input. NOTE: Since a binary file is transferred in INTEL HEX format, the .HEX file on the SuperBrain can be changed using LOAD or DDT and SAVE, to a binary file.

```
+ HEXDUMP ABC.COM                     (Keyboard entry)
```

The file is now being transferred and also displayed on the screen. When the transfer is complete, the SuperBrain will return to the operating system.

2) Using HEXGEN. ASM
Look at source listing:

```
     ORG          6000H

     LXI          SP 6400H
     LXI          D, 6000H      *ending address
     LXI          H, 100H       *beginning address
```

The origin and the SP will need to be modified for your particular system. (For example: 32K system use ORG 5000H, and SP, 5400H.) You may also change H,D to suit program size; register H is loaded with the end address of the program to be transferred and register D has the beginning address (most programs begin at 100H). Now run assembler to generate HEXGEN.HEX. You are ready to begin.

```
A> PIP ABC.HEX = INP:  [H]            (Keyboard entry)
ECHO  (Y/N)  Y                        (Computer responds)
                                      (Keyboard entry)
+                                     (Computer responds)
```

At this point the SuperBrain is ready for input and the host must be set up to output the HEX file.

```
+DDT                                  (Keyboard entry)
Version 1.4                           (Computer responds)
-
```

DATE OF THIS RELEASE November, 1979 PAGE 5 OF 6 BULLETIN # B119004

ASSEMBLY NAME/NUMBER DOS DISKETTE PRODUCT SuperBrain

REFERENCE ECO # N/A DISTRIBUTED TO APPROVED

Now we have loaded DDT into the host system.

```
-IABC.COM↵                          (Keyboard entry)
-R↵
NEXT PC                             (Computer responds)
0A00 0100                           (These two numbers are the
                                     end and starting address)
-IHEXGEN.HEX↵                       (Keyboard entry)
-R↵
NEXT PC                             (Computer responds)
60B8 0100
```

At this point the host computer has 2 programs loaded into memory, one above the other. One is the program to be transferred and the other to generate the INTEL HEX format.

```
-G6000↵                             (Keyboard entry)
                                    (The number is the same
                                     as ORG in the source listing)
```

Now the file is being transferred and will be displayed on the screen. After the program has been transferred, the SuperBrain will return to the operating system.

3) To change back to a binary file, follow this procedure:

```
A> LOAD ABC.HEX↵                    (Keyboard entry)

LAST ADDRESS XXXX                   (Computer responds)
FIRST ADDRESS XXXX
BYTES READ XXXX
RECORDS WRITTEN XX

A>
```

Now there are two files: one HEX and one binary.

or

```
A> DDT ABC.HEX↵                     (Keyboard entry)
Version 1.4                         (Computer responds)
Next PC
ABCD 0100
-
(CTRL) (C)                          (Keyboard entry - both keys
                                     at same time)
```

DATE OF THIS RELEASE November, 1979 PAGE 6 OF 6 BULLETIN # B119004

ASSEMBLY NAME/NUMBER DOS DISKETTE PRODUCT SuperBrain

REFERENCE ECO # N/A DISTRIBUTED TO APPROVED

```
A> SAVE XX ABC.COM                (Keyboard entry)

NOTE:  XX = A times 16 + B
            under NEXT
```

Now there are two files: one .HEX and one binary.

*Contact the Customer Service Department at the address above for additional information on this bulletin.*

INTERTEC DATA SYSTEMS®

Corporate Headquarters: 2300 Broad River Road, Columbia, South Carolina 29210 • 803/798-9100 • TWX: 810-666-2115

**TECHNICAL BULLETIN**
IDS - 912A

DATE OF THIS RELEASE January 10, 1980 PAGE 1 OF 1 BULLETIN # B010008

ASSEMBLY NAME/NUMBER Main Power Supply PRODUCT SuperBrain & InterTube

REFERENCE ECO # N/A DISTRIBUTED TO InterTube & SuperBrain Resellers APPROVED

## COMPATIBILITY INFORMATION FOR REVISION 3 AND 4 OF THE INTERTUBE AND SUPERBRAIN MAIN POWER SUPPLY MODULE/ASSEMBLY

Revision 4 of the SuperBrain Main Power Supply Module is compatible only with Revision 1 of the SuperBrain Keyboard/CPU Module and any revision level of the InterTube Processor Module.

Revisions 1 - 3 of the SuperBrain Main Power Supply can be used only with Revision 0 of the SuperBrain Keyboard/CPU Module and any revision level of the InterTube Processor Module.

CAUTION: Attempts by the customer to connect incompatible Power Supply Modules with either Keyboard/CPU Modules or Processor Modules will cause severe, irreparable damage to all modules connected in this manner.

Since compatibility must be observed when interchanging modules, it is necessary for all customers to specify the revision level of any module which is requested to be sent from our Service Department prior to return of a defective module. Revision levels of all modules/subassemblies are listed as a suffix number of the standard Intertec module part number. Example: Intertec number 1424002-04 would specify revision level "4" of the SuperBrain Main Power Supply.

*Contact the Customer Service Department at the address above for additional information on this bulletin.*

INTERTEC DATA SYSTEMS

**TECHNICAL BULLETIN**
IDS - 912A

® Corporate Headquarters: 2300 Broad River Road, Columbia, South Carolina 29210 • 803/798-9100 • TWX: 810-666-2115

DATE OF THIS RELEASE January 10, 1980 PAGE 1 OF 1 BULLETIN # B010009

ASSEMBLY NAME/NUMBER Keyboard/CPU Module PRODUCT SuperBrain

REFERENCE ECO # N/A DISTRIBUTED TO Manuals and as requested APPROVED

## SUPERBRAIN CPU MODULE REVISION 1
## SERIAL COMMUNICATIONS DIPSWITCH SETTING PROTOCOL

Starting with Revision 1 of the SuperBrain Keyboard/CPU Module (all factory produced units effective January 10, 1980) there exists a small 5 position dipswitch located in the upper right hand corner of this module. This switch is used to control various clock parameters to and from the MAIN USART. For normal use these switches should be set as follows:

1 - OFF, 2 - OFF, 3 - ON, 4 - ON, 5 - OFF

Listed below is a brief description of the function of each of these switches:

1 - External TX Clock to MAIN USART - Originates from Pin #15 on MAIN RS232 connector at rear of terminal.

2 - External RX Clock to MAIN USART - Originates from Pin #17 on MAIN RS232 connector at rear of terminal.

3 - Internal TX Clock to MAIN USART - When on this switch enables the built-in baud rate generator (Western Digital BR-1941). NOTE: When this switch is in the 'ON' position switch 1 MUST be in the 'OFF' position.

4 - Internal RX Clock to MAIN USART - When this switch is in the 'ON' position switch 2 MUST be in the 'OFF' position.

5 - Internal Baud Clock to MAIN Port - This switch enables the transmission of the internal baud rate clock (Western Digital BR-1941) to the main RS232 port - this signal will appear on Pin #24 of the main port when this switch is in the 'ON' position. If this switch is not used, it should be left in the 'OFF' position to avoid any possible conflict with external RS232 signals.

*Contact the Customer Service Department at the address above for additional information on this bulletin.*

## STATEMENT OF LIMITED WARRANTY

For ninety (90) days from the date of shipment from our manufacturing plant at 2300 Broad River Road, Columbia, South Carolina, Intertec warrants, to the original purchaser only, that its products, excluding software products, will be free of defective parts or components and agrees to replace or repair any defective component which, in Intertec's judgment, shall disclose to have been originally defective. Intertec neither offers nor implies any warranty whatsoever on any software products. Furthermore, Intertec's obligations under this limited warranty are subject to the following conditions:

## LIMITED WARRANTY REPAIRS

Unless authorized by written statement from Intertec, all repairs must be done by Intertec at our plant in Columbia, South Carolina. Return of any and all parts and/or equipment must be freight prepaid and accompanied by an Intertec Return Material Authorization number which must be clearly visible on the customer's shipping label. Return of parts or equipment contrary to this policy shall result in the material being refused, and the customer being invoiced for any replacement parts, if any were previously issued, at Intertec's standard prices.

When making repairs or replacing parts in accordance with this limited warranty, Intertec reserves the right to alter and/or modify specifications of this equipment.

Upon completion of the repairs, Intertec will return the equipment, freight collect, directly to the customer from whom it was sent via UPS or equivalent ground transportation.

Authorization to return equipment for repair can be obtained by writing Intertec at the address stated herein or by calling our Customer Service Department at 803/798-9100.

In the event Intertec shall authorize repair of its equipment, in writing, by an authorized repair agent, then Customer shall bear all shipping, packing, inspection and insurance costs necessary to effectuate repairs under this warranty.

## EXCLUSIONS

The Limited Warranty provided by Intertec Data Systems Corporation does not include:

(a) Any damage or defect caused by injuries received in shipment or any damage caused by unauthorized repairs or adjustments. The risk of loss or damage to the equipment shall pass to the Customer upon delivery by Intertec to the carrier at Intertec's premises.

(b) Repair, damage or increase in service time caused by failure to continually provide a suitable installation environment including, but not limited to, the failure to provide, or the failure of, adequate electrical power, air-conditioning, or humidity control.

(c) Repair, damage or increase in service time caused by accident or disaster, which shall include, but not be limited to, fire, flood, water, wind, lightning, transportation neglect, misuse and alterations, which shall include, but not be limited to, any deviation from the original physical, mechanical or electrical design of the product.

(d) Any statements made about the equipment by salesman, dealers or agents unless such statements are in a written document signed by an officer of Intertec Data Systems Corporation. Such statements do not constitute warranties, shall not be relied on by the buyer, and are not part of the contract for sale.

(e) Any damage arising out of any application for its products other than for normal commercial and industrial use, unless such application is, upon request, specifically approved in writing by Intertec. Intertec products are sophisticated data processing units and are not sold or distributed for personal, family or household purposes.

(f) Software, including either source code, object code or any computer program used in connection with our equipment, whether purchased directly from Intertec or from an independent source.

## WAIVER OF ALL EXPRESS OR IMPLIED WARRANTIES

Our limited warranty to repair or replace defective parts or components for ninety (90) days after shipment from our Columbia plant is being offered in lieu of all express or implied warranties.

INTERTEC MAKES NO EXPRESS WARRANTY OTHER THAN THE LIMITED WARRANTY SET FORTH ABOVE, CONCERNING THIS PRODUCT OR ITS COMPONENTS, NOR DO WE IMPLIEDLY WARRANT ITS MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

All statements, technical information and recommendations contained in this and related documents are based on tests we believe to be reliable, but the accuracy or completeness thereof is not guaranteed.

THE FOREGOING LIMITED WARRANTIES ARE IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, EXCEPT AS TO CONSUMER GOODS IN WHICH CASE THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY ONLY FOR THE PERIOD OF THE LIMITED WARRANTY.

PURCHASERS OF CONSUMER PRODUCTS SHOULD NOTE THAT SOME STATES DO NOT ALLOW FOR THE EXCLUSION OF CONSEQUENTIAL DAMAGES OR THE LIMITATION OR THE DURATION OF IMPLIED WARRANTIES SO THE ABOVE EXCLUSION AND LIMITATION MAY NOT BE APPLICABLE.

THIS LIMITED WARRANTY GIVES THE PURCHASER SPECIFIC LEGAL RIGHTS, AND THE PURCHASER MAY ALSO HAVE OTHER RIGHTS WHICH MAY VARY FROM STATE TO STATE.

## LIMITATION OF REMEDIES

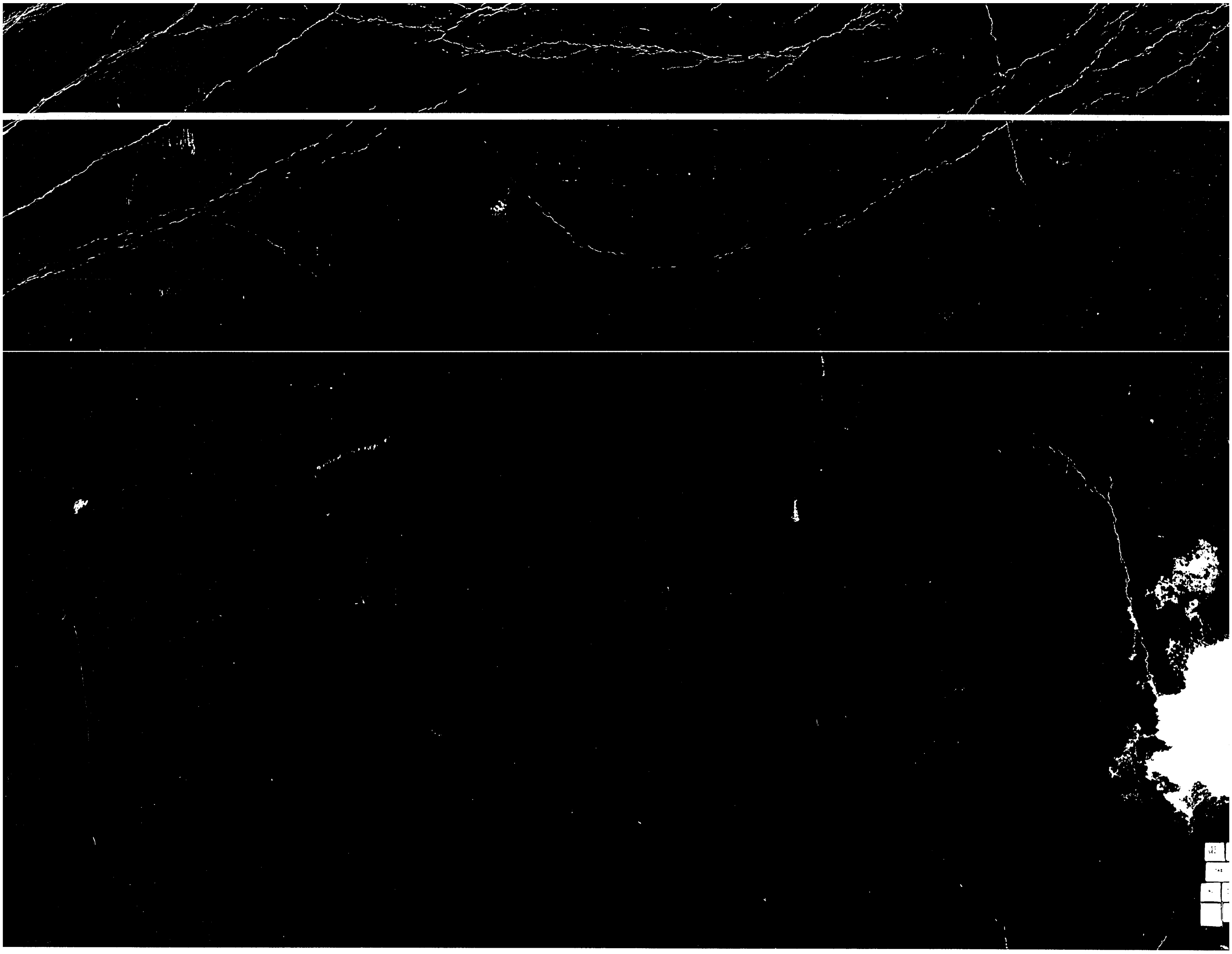
INTERTEC SHALL NOT BE LIABLE FOR ANY INJURY, LOSS OR DAMAGE, DIRECT OR CONSEQUENTIAL, TO PERSONS OR PROPERTY CAUSED EITHER DIRECTLY OR INDIRECTLY BY THE USE OR INABILITY TO USE ITS PRODUCTS AND/OR DOCUMENTS. SUCH LIMITATION IN LIABILITY SHALL REMAIN IN FULL FORCE AND EFFECT EVEN WHEN INTERTEC MAY HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH INJURIES, LOSSES OR DAMAGES.

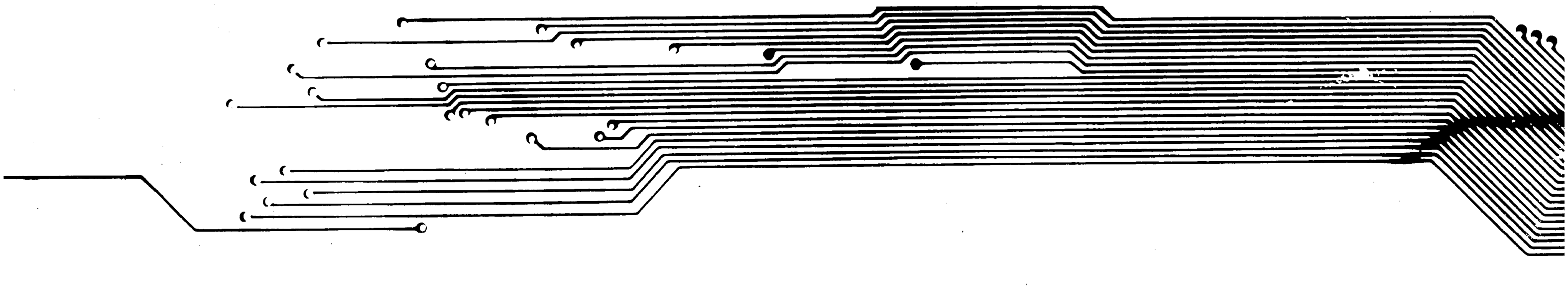
Before purchasing or using, the Customer shall determine the suitability of Intertec's products and documents for his intended use and assumes all risk and liability whatsoever in connection therewith.

THE LIMITED WARRANTY TO REPLACE OR REPAIR PARTS OR COMPONENTS FOR NINETY(90) DAYS IS THE EXCLUSIVE REMEDY PROVIDED TO THE CUSTOMER AND THE LIABILITY OF INTERTEC WITH RESPECT TO ANY OTHER CONTRACT, SALE OR ANYTHING DONE IN CONNECTION THEREWITH, WHETHER IN CONTRACT, IN TORT, UNDER ANY WARRANTY, OR OTHERWISE, SHALL NOT EXCEED THE PRICE OF THE PART OR COMPONENT ON WHICH SUCH LIABILITY IS BASED.

Rights under this warranty are not assignable without the express prior consent, in writing, of Intertec Data Systems Corporation, and, regarding the terms of such consent in writing, the assignee shall have no greater rights than his assignor.

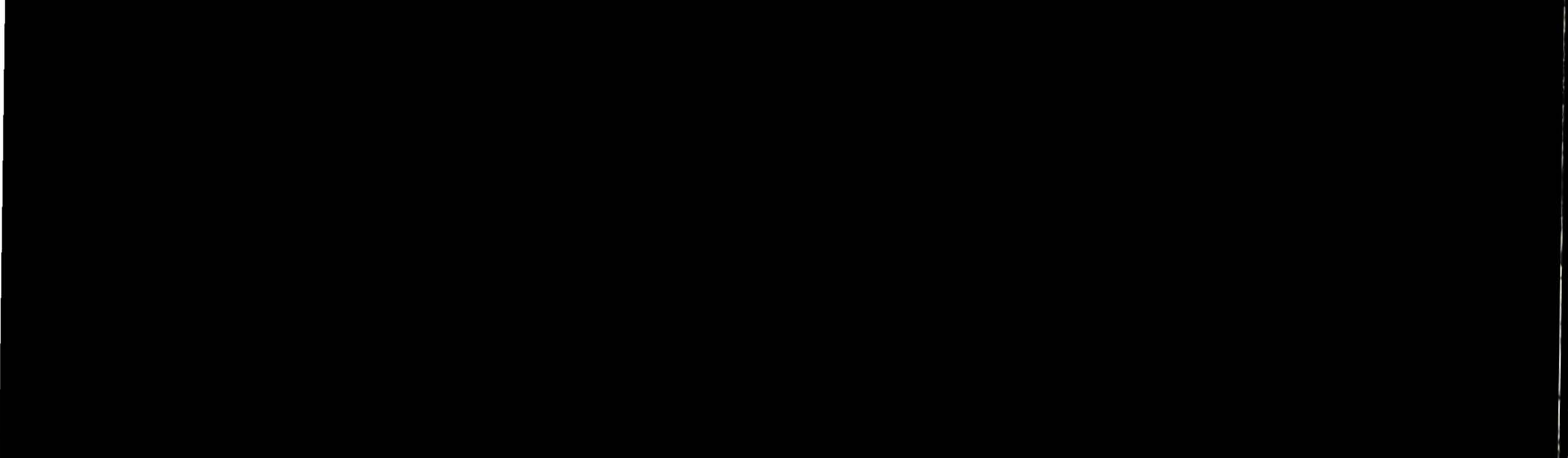
In the event the Customer has any problem or complaints arising out of any breach of our limited warranty, including a failure to make repairs in accordance with the warranty, or unsuccessful repair attempts by an authorized repair facility, the Customer is encouraged to inform Intertec, in writing, of his or her problem or complaint. Any such writing should be addressed to Intertec Data Systems Corporation, 2300 Broad River Road, Columbia, South Carolina, 29210 and should be marked with the phrase "Warranty Claim."





INTERTEC
DATA
SYSTEMS®

CORPORATE HEADQUARTERS • 2300 BROAD RIVER ROAD • COLUMBIA, SOUTH CAROLINA 29210 • 803/798-9100



INTERTEC
DATA
SYSTEMS®

CORPORATE HEADQUARTERS • 2300 BROAD RIVER ROAD • COLUMBIA, SOUTH CAROLINA 29210 • 803/798-9100