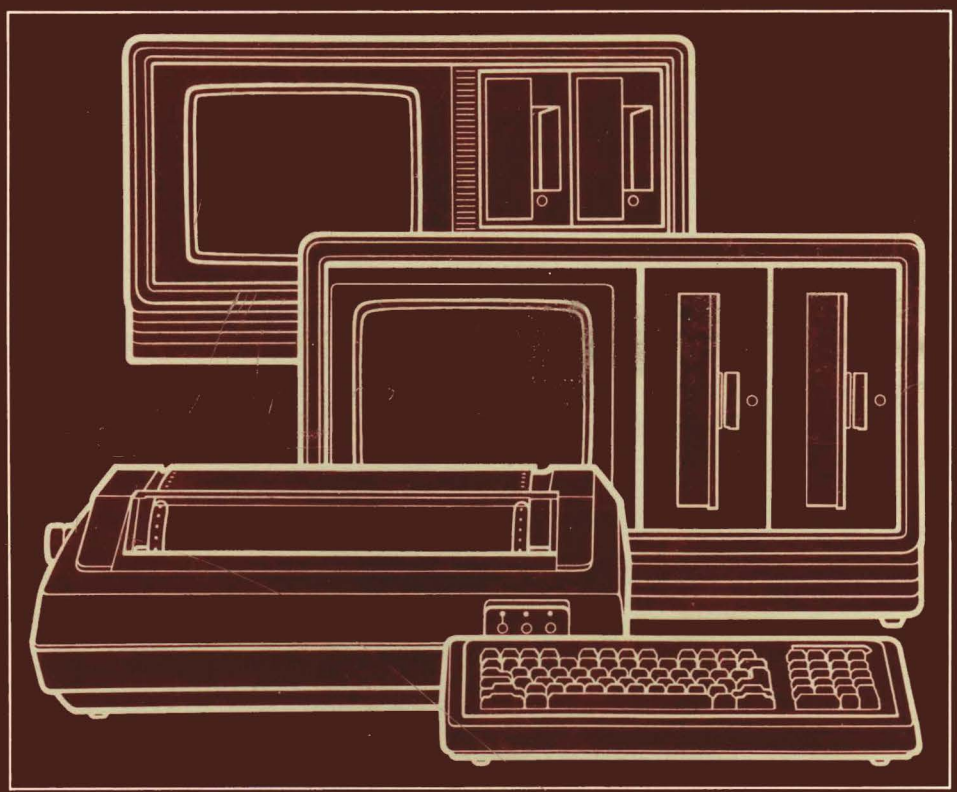
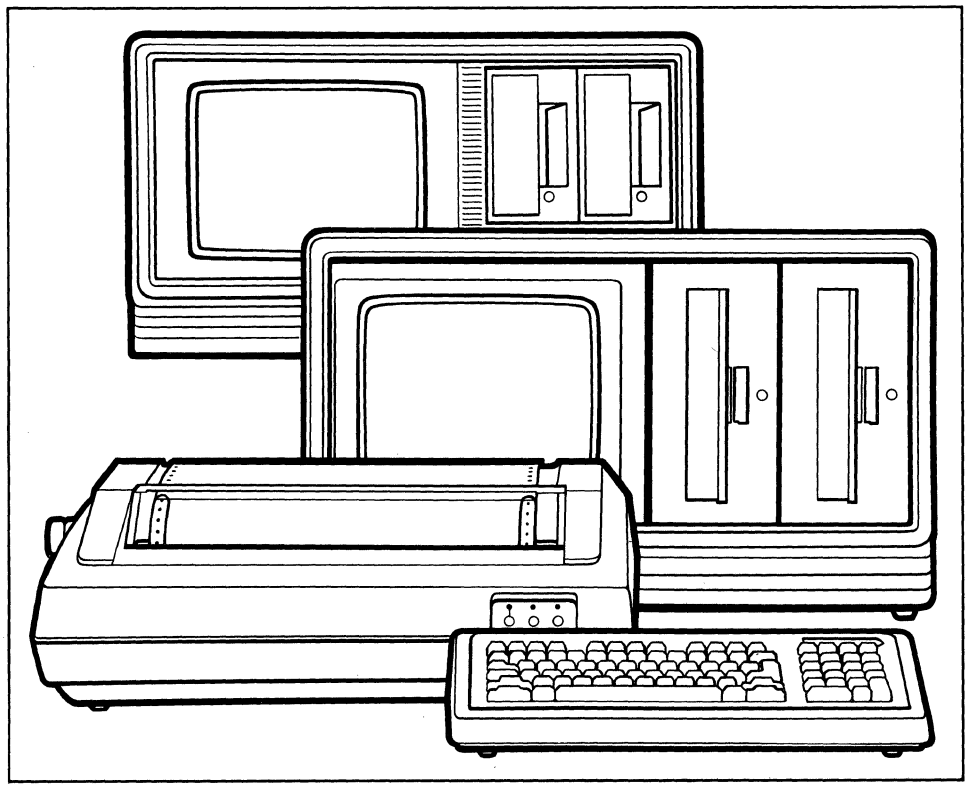
USER'S MANUAL
T200/T250
PERSONAL OFFICE COMPUTER



TOSHIBA

USER'S MANUAL
T200/T250
PERSONAL OFFICE COMPUTER



TOSHIBA

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

INTRODUCTION

Your Toshiba T200 or T250 computer gives you very powerful processing capabilities to keep pace with the challenges of today's Information Age. The extensive capabilities of your system are the result of centuries of evolution in computing power:

1642 *Blaise Pascal built the first known gear-based adding machine.*

1694 *Gottfried Willhelm von Leibniz, the inventor of calculus, improved on Pascal's adding machine. His version could add, subtract, multiply, divide and extract square roots by repeated additions — which is exactly how modern computers handle such problems.*

1835 *With British government financing, Charles Babbage built a machine that not only performed the calculations of the Leibniz machine, but also was a true programmable computer.*

1939 *The first automatic digital computer was built. It operated electromechanically but was quite noisy. The first fully electronic — and silent — computer followed.*

1960 *Transistors began to take over, allowing miniaturization of the formerly gigantic computers.*

1970 *Integrated circuits emerged next. Standards began to develop for the Computer Age.*

1980 *More effective and less costly computers became widely available.*

Your Toshiba table-top T200 has capabilities that once required a computer that filled a large room and cost hundreds of thousands of dollars. Today, at a fraction of that cost, Toshiba's business computers offer more capabilities, greater efficiency, greater ease-of-use and far more reliability than their predecessors of 25 years ago.

Today, Toshiba's T200 and T250 computers are widely used to handle such business accounting applications as general ledger, accounts receivable, accounts payable and payroll. They write orders, keep track of inventory, control production, manage work and manufacturing schedules and file and fill out government and other forms. Toshiba's T200 and T250 computers also provide valuable management aids, allowing fast and precise profit analysis, and quick performance assessment of the sales force, including route management and highlighting of weak or key sales areas. In addition, the T200 and T250 can provide valuable decision making support because they can store and sort through enormous amounts of information at high speeds, allowing managers to simulate the results of various choices they are considering.

Such a list should certainly inspire you toward plans for your new computer. And, soon after you begin to see your computer's capabilities, you will even come up with other uses tailored to meet your needs.

You are to be congratulated on your choice of a system with top quality engineering. Every imaginable aspect of business requirements and operator needs has been taken into consideration during the design process. The following are a few of the system advantages:

- *Your T200 or T250 is compatible with the majority of standard components and programs currently on the market. Numerous programming packages which are already available can be used just as they are. Programs can be purchased from Toshiba to meet various needs. If these are applicable, you need not write programs for your computer.*
- *In addition, new programs can be written should you choose to do so, and existing programs can be modified to suit your system needs.*
- *User-oriented programmable function keys are available.*
- *Large disk storage eliminates many of the problems encountered with older computers.*
- *A high speed, high quality printer provides efficiency, as well as a professional looking product.*
- *The system layout is flexible and easy to use.*

Whether you are an experienced computer professional or a first time user, you are probably experiencing excitement at the prospect of mastering and enjoying the benefits of a new tool, as well as some concern about the process of that mastery. This manual is designed to accompany you through that process with as much enjoyment, and as little frustration, as possible. First time users have the detail necessary to learn what they require and experienced computer users will find the information well-organized for either scanning or in-depth study.

Don't try to understand how to use your computer by reading through the manual. Rather, we recommend that after your computer is installed (described in ***Appendix A)****, you start your journey at the computer itself, with the manual open as a guide beside you.*

Your manual is divided into two major sections: ***Part 1*** *and* ***Part 2.*** *You will begin with Part 1 which describes the system* ***hardware.*** *Hardware is the actual equipment —the keyboard, the main unit, the disks, the printer. Part 1 explains this equipment and how to use it. Illustrations will guide you, both for the T200 and T250.*

Part 2 of the manual will cover the system ***software.*** *Software refers to programs. Programming makes the computer (the hardware) perform various functions such as figure payroll or check inventory. As mentioned, the T200 and T250 are designed so that you can buy "prepackaged" programs, write programs yourself or modify existing programs to meet your needs.*

Part 2 first describes the ***operating system*** *which is software designed to act like your computer's manager. You will learn to interact with and oversee that manager. Part 2 also covers use of the* ***BASIC*** *programming language on your computer. (BASIC is easy to learn and versatile for performing a range of business applications. Learning BASIC, however, is a process all by itself and if you do not already know it, you should consult the MBASIC and CBASIC reference manuals devoted to that purpose.) Part 2 also discusses assembler programming for those who might use it, as well as the use of program debugging and text editing utilities.*

Infrequently used information has been placed in appendices. For ease of reference, an index appears at the end of this manual.

Although every attempt has been made to make the use of this manual as comfortable as possible, the information presented here will only "come alive" as you use it to discover the capabilities of your system. As you persist and master the use of your computer system, you will achieve the satisfaction of seeing your business operations improve greatly.

PART 1

OVERVIEW OF PART 1

Part 1 presents facts about operating the hardware, or equipment, of your computer system. Included are descriptions of:

- Understanding the Basic Components
- Turning the Power On
- Adjusting the Display Screen Brightness
- Using the Cursor
- Using the Keyboard
- Working with the Floppy Disks
- Turning the Power Off

Most of the material in Part 1 applies equally to both the T200 and T250 systems. In cases where there are differences, a unique one-line border is used to the left of the section to indicate whether the material refers to the T200 or T250. This quickly highlights sections you do need to read, as well as those that you do not. The following borders are used:

 = **T200**

 = **T250**

You will observe in Part 1 that all illustrations and charts are placed on the left pages, while text is placed on the **right** pages. Therefore, always begin by reading the right page, and refer to the left page when it is referenced in the text.

Once you have learned the basics about operating the hardware, you will be ready to learn in Part 2 about using the accompanying software.

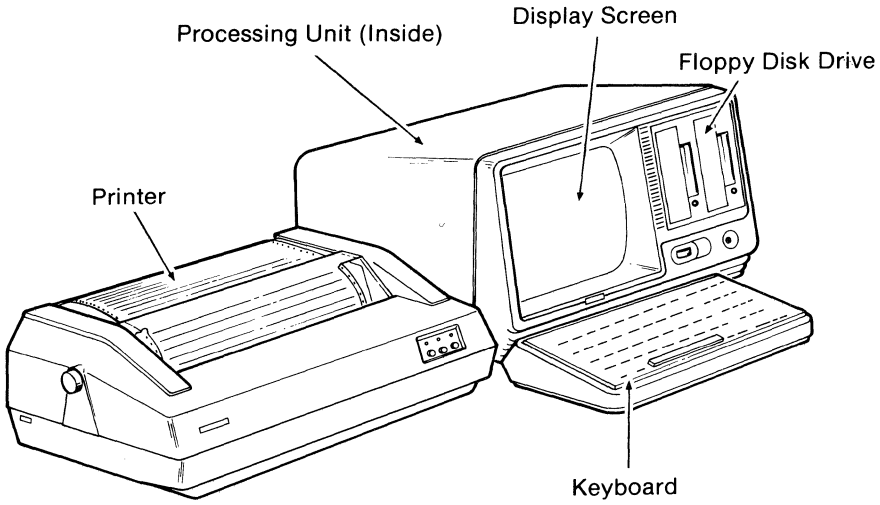


Figure 1.1 T200 *Computer System*

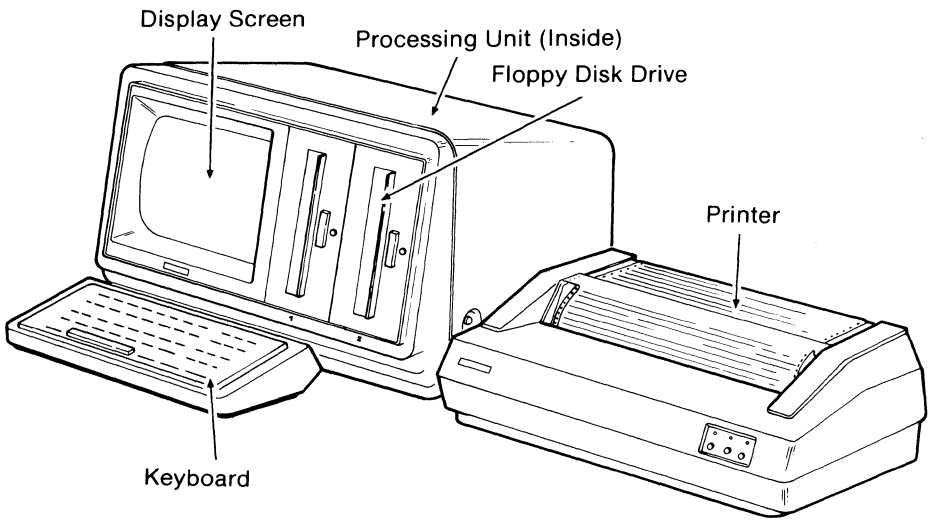


Figure 1.2 T250 *Computer System*

THE FIVE BASIC COMPONENTS

This section provides general information about the five basic components of your T200 or T250, as shown in the figures at left. As needed, following sections will describe the components and their use in more detail.

KEYBOARD

A typewriter-like keyboard that also includes special programming and number keys. You enter data and instructions into the computer via the keyboard, which is movable for your convenience.

DISPLAY SCREEN

A TV-like screen measuring 12 inches diagonally. The screen displays information the user enters via the keyboard, prompts the user for responses and instructions and shows replies from the computer. The screen contains up to 1,920 characters of information, 80 characters per line and 24 lines at a time. The displayed image scrolls up when the screen is filled up and a new line is to be displayed.

PROCESSING UNIT

The "insides" of the main console box. This unit is the control center of the system and does the arithmetic and logic operations, as well as other system control functions. It also contains 65,536 **bytes** of main memory. (One byte holds one character of information.)

FLOPPY DISK DRIVE

Data and programs are stored on what is called a **floppy disk,** which is a thin, flexible disk permanently enclosed in a plastic jacket. These floppy disks are inserted into a floppy disk drive, where information can be written onto the disk or read from it. Since information can be stored, it does not need to be rekeyed constantly on the keyboard.

PRINTER

The printer is used when you want to print information instead of just leaving it on the screen. **Appendix D** describes the use of the printer.

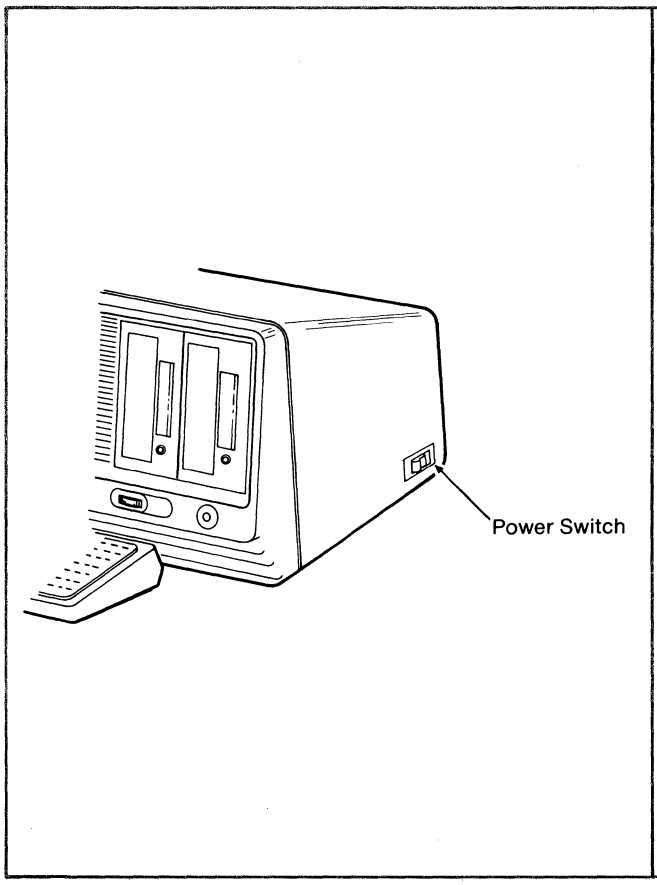


Figure 2.1 T200: *Turning the Power On*

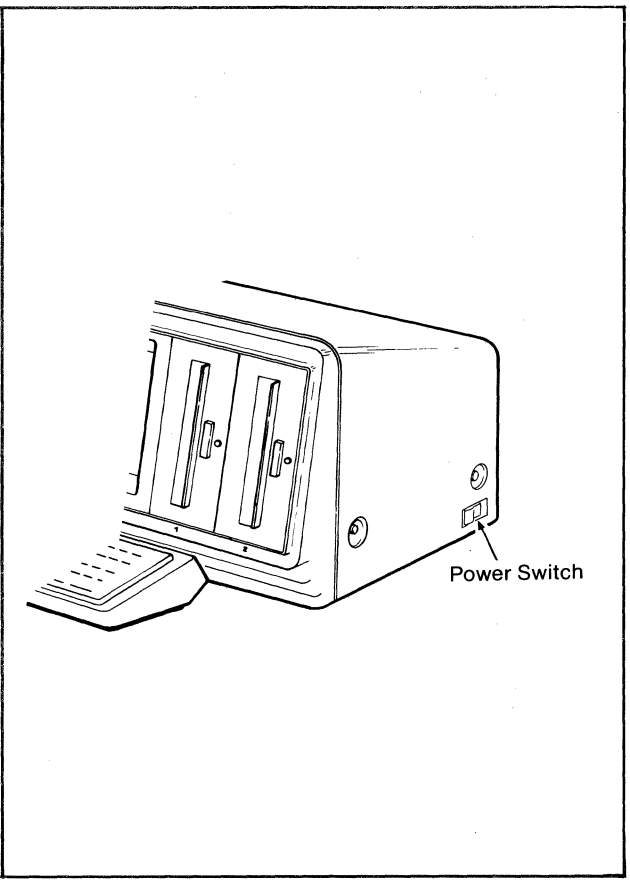


Figure 2.2 T250: *Turning the Power On*

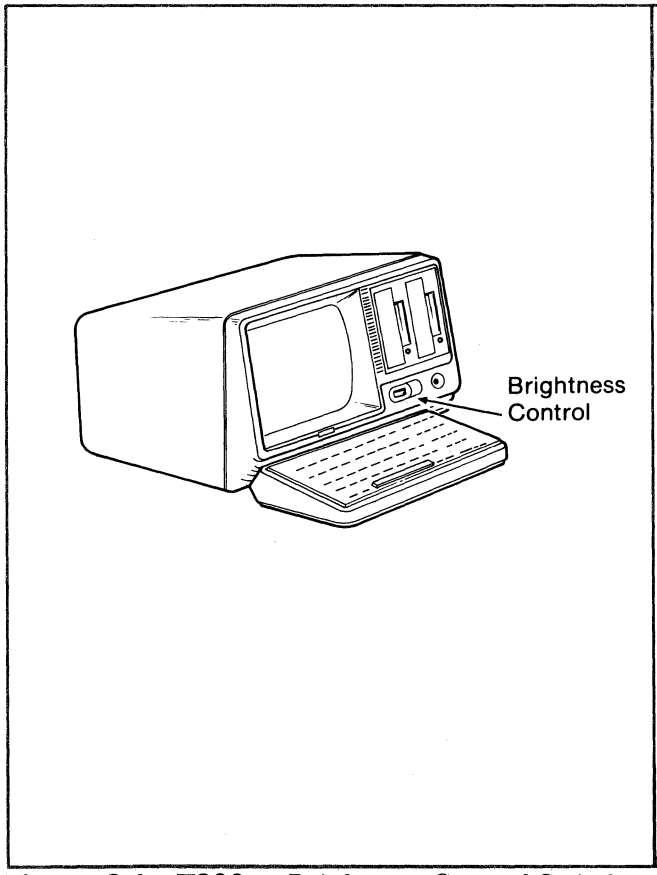


Figure 3.1 T200: *Brightness Control Switch*

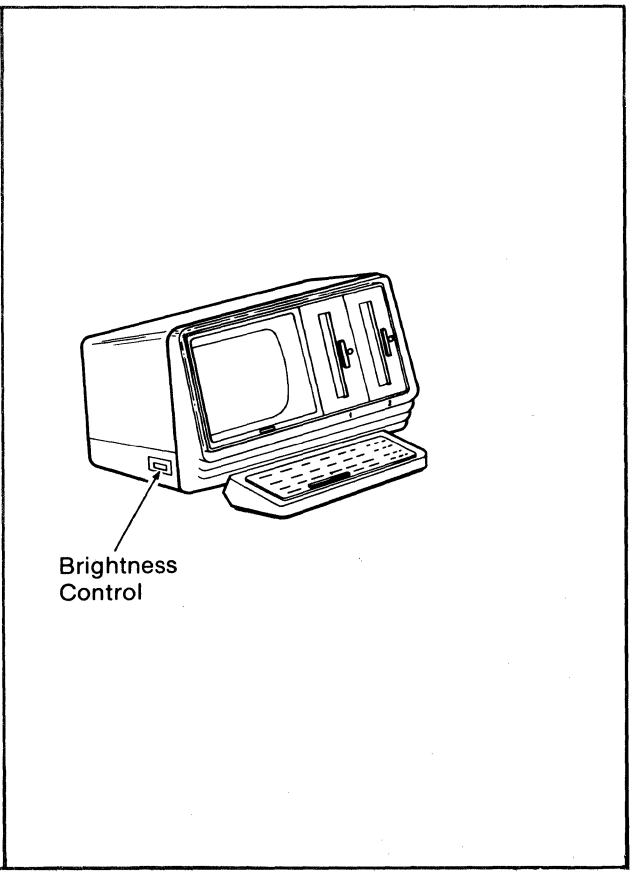


Figure 3.2 T250: *Brightness Control Switch*

TURNING THE POWER ON

The console power switch is located on the back right-hand side of the main console unit. To turn the power on, press the raised side of the switch, as shown in the top figures at left. When the power is turned on, the cooling fan starts running.

When the power has been turned off, wait several seconds before turning it on again.

NOTE: This power switch does not turn on the printer. See **Appendix D** for printer instructions.

ADJUSTING THE DISPLAY SCREEN BRIGHTNESS

The intensity of the characters on the display screen can be adjusted to optimize the comfort of your eyes. Simply turn the control switch. This switch is found (as shown in the figures at left.)

T200 - on the front of the main unit below the disk drives.

T250 - on the left side of the console.

The characters are light green, displayed against a dark green background.

To prolong screen life, keep the brightness setting at the lowest level that allows eye comfort during work sessions and when the system is left idle. Orienting the screen away from glare allows using a lower brightness level.

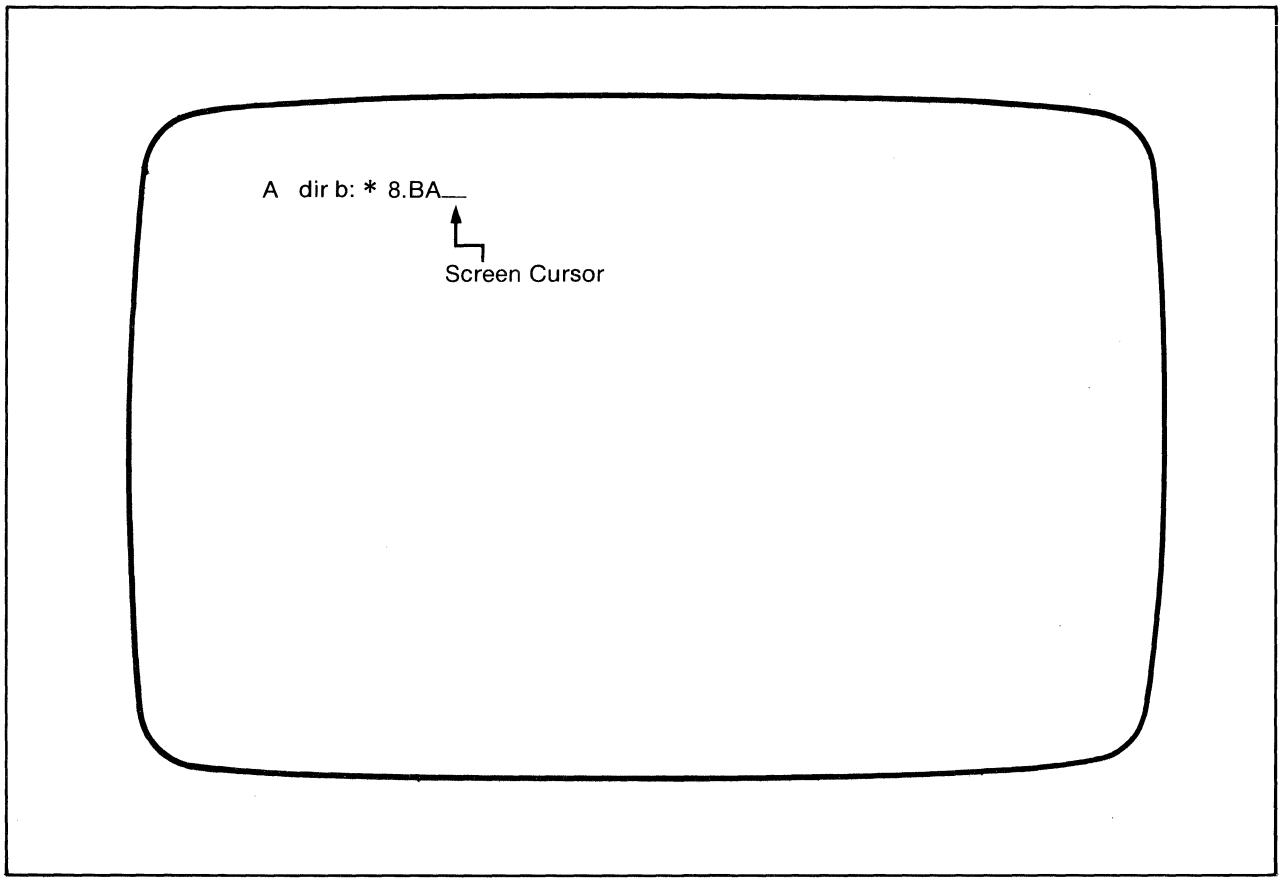


Figure 4 *The Cursor*

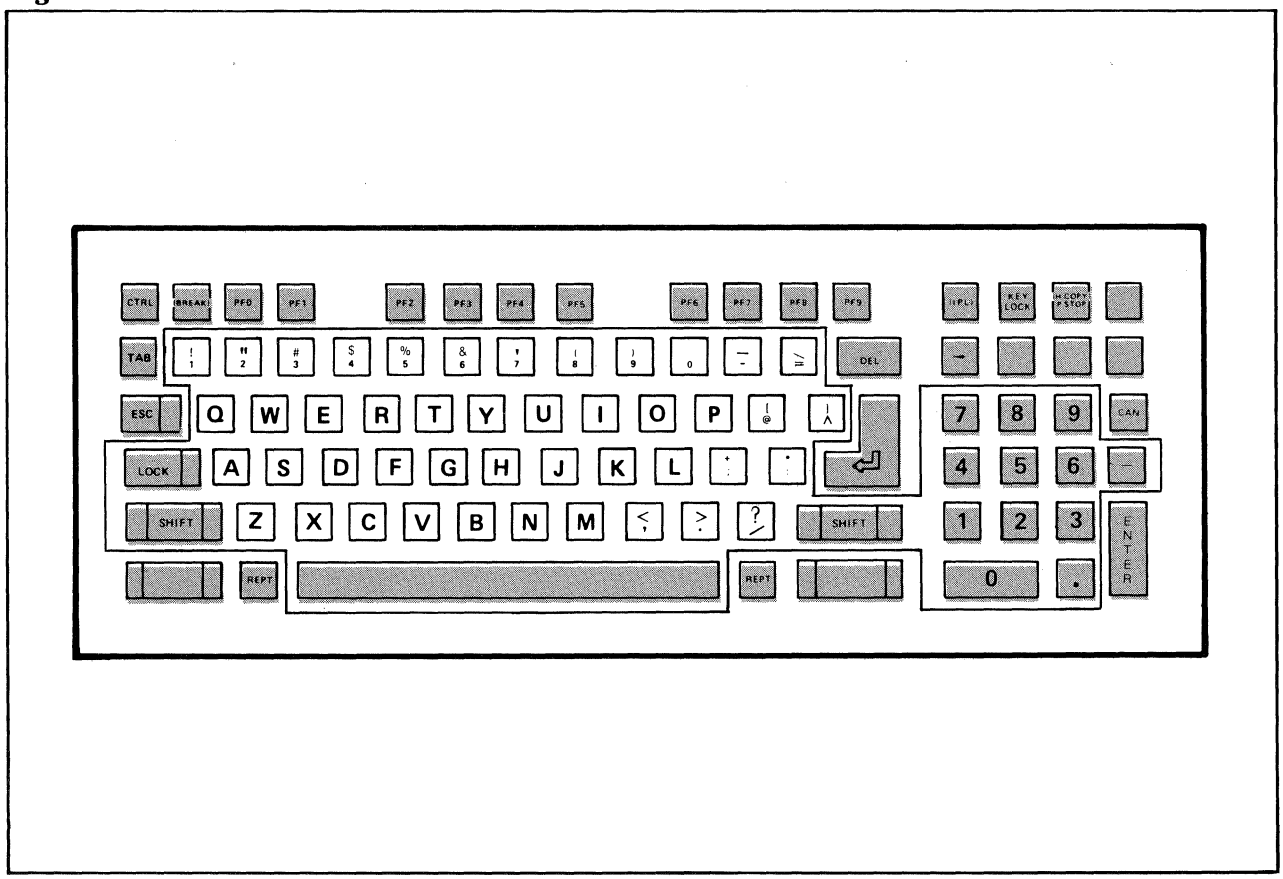


Figure 5 *The Keyboard — Data Keys*

THE CURSOR

The display screen contains a marker that moves like a "bouncing ball" to indicate your place on the screen. The marker, called the **cursor**, is in the shape of a flashing underscore. Top figure at left illustrates the cursor.

When data keys are pressed on the keyboard, the entered characters are displayed on the screen and the cursor advances with each new keystroke.

USING THE KEYBOARD

The keyboard, shown at lower left, contains various types of keys:

- Data keys
- Line editing keys
- Special function keys
- Program function keys

Data Keys

Uppercase and lowercase characters, numeric characters and special symbols are on the white keys. These keys, along with the space bar, the SHIFT key and the LOCK key, are operated as on a standard typewriter.

When the LOCK key is touched, uppercase letters can be entered without holding the SHIFT key. The red light on the LOCK key is lit while the keyboard is in this upper case mode. Touch the LOCK key again to resume the lowercase mode.

The keyboard also has a 10-key numeric pad, on the right side, to aid in entering numbers.

KEYBOARD, Continued →

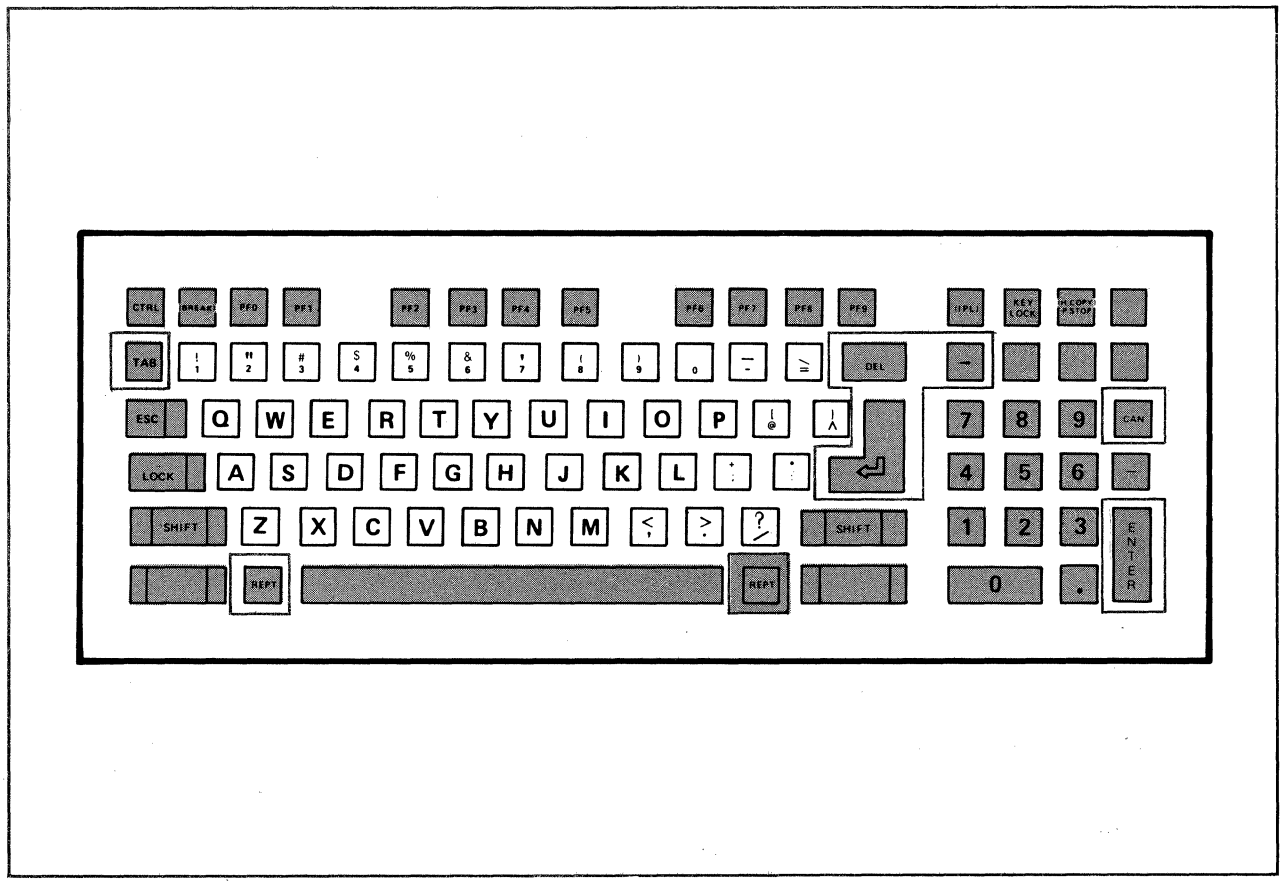
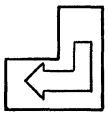


Figure 6 *The Keyboard — Line Editing Keys*

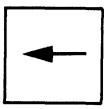
Line Editing Keys

Line editing keys may be pressed when you enter data to send, correct or repeat data. A TAB key is convenient for placement of the data.



The carriage return key terminates the input of a line and sends it to the system.

The ENTER key can be used as a substitute for the carriage return key.



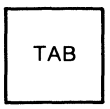
The backspace key moves the cursor back to the left by one character position. This key is used to retype and correct the characters of a line before pressing the carriage return key.

The DEL key deletes the last character typed on the line, but displays it on the screen with an "echo effect". This can be used for more than one character. Example:

- **abcdef** is typed.
- DEL is pressed three times.
- **abcdeffed** is displayed on the screen.
- **abc** is the effective input.

The CAN key cancels the entire input line if pressed before the carriage return key. When the CAN key is pressed, # is displayed, and the input can be retyped from the beginning.

When the REPT key is held down with a character key, the character typed repeats.



The TAB key moves the cursor position to the next tab. Tabs are set on every 8 characters of a line.

KEYBOARD, Continued →

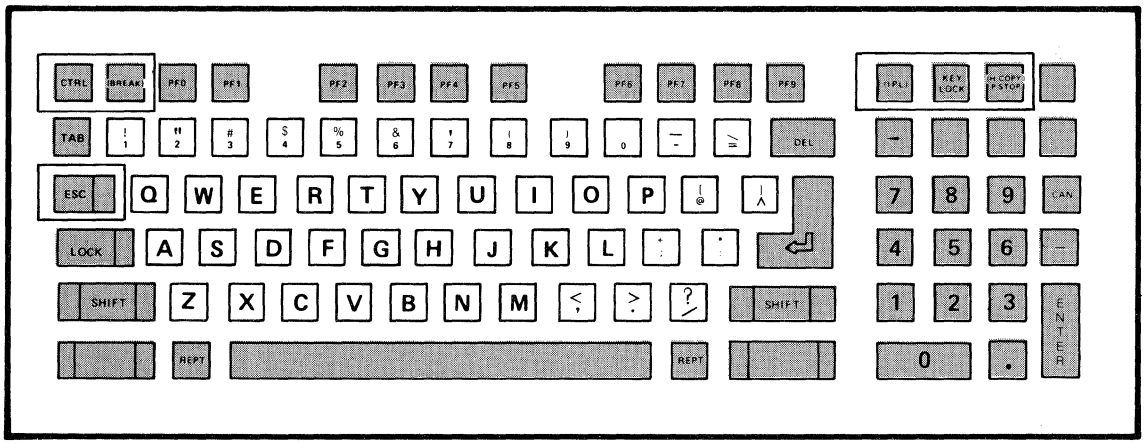


Figure 7 *The Keyboard — Special Function Keys*

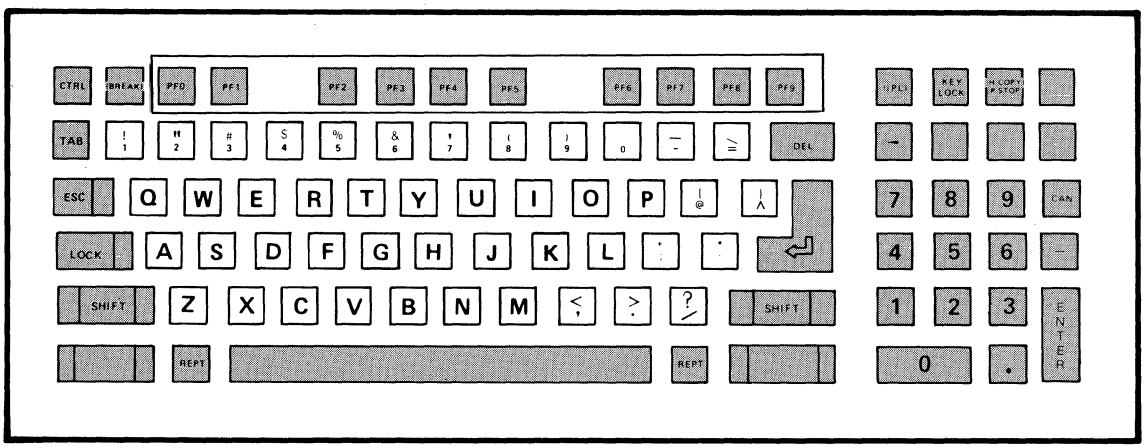


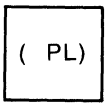
Figure 8 *The Keyboard — Program Function Keys*

Special Function Keys

Pressing special function keys tells the system to perform certain functions.

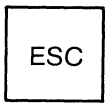
Pressing this key causes all following operating system command interaction (between keyboard and screen) to be printed on the printer. Pressing this key again deactivates the printing. (This does not work while MBASIC is in use.)

Keyboard operation is disabled. The second touch unlocks the keyboard.



When pressed simultaneously with the CTRL key, the system Cold Boot is performed (described under "Using the System Disk").

Reboots the system (described under "Using the System Disk") or interrupts BASIC processing. When this key is pressed, ^**C** is displayed on the screen.



Edits a BASIC input line with the EDIT comand. When this key is pressed, [is displayed.

Performs predefined functions when another key is pressed simultaneously.

Program Function Keys

Ten program function keys, **PF0** through **PF9,** send a specific code to the program executing in the central processor. The following symbols, respectively, are displayed on the screen when one of the program keys is pressed:

{, |, }, §, ^B, ^D, ^F, ^G, ^K or ^N

Appendix B shows the character code table, including a decimal representation of the program function keys.

TAKING CARE OF THE DISKS

- When not in use, floppy disks must be stored in the protective envelopes.
- Do not touch or attempt to clean the data recording surface of the floppy disk.
- Floppy disks may be damaged if twisted, bent, dropped sharply, exposed to sunshine, winter cold, food, liquid, beverages or dust, including smoke.
- Do not write too firmly on the floppy disk label. A felt-tip pen is recommended.
- Do not use erasers on the floppy disk label or near the floppy disk.
- Do not use magnets or magnetized objects near the floppy disk.
- Do not use rubber bands or paper clips on the floppy disk.
- Do not place heavy objects on the floppy disk.

Figure 9 *Taking Care of the Disks*

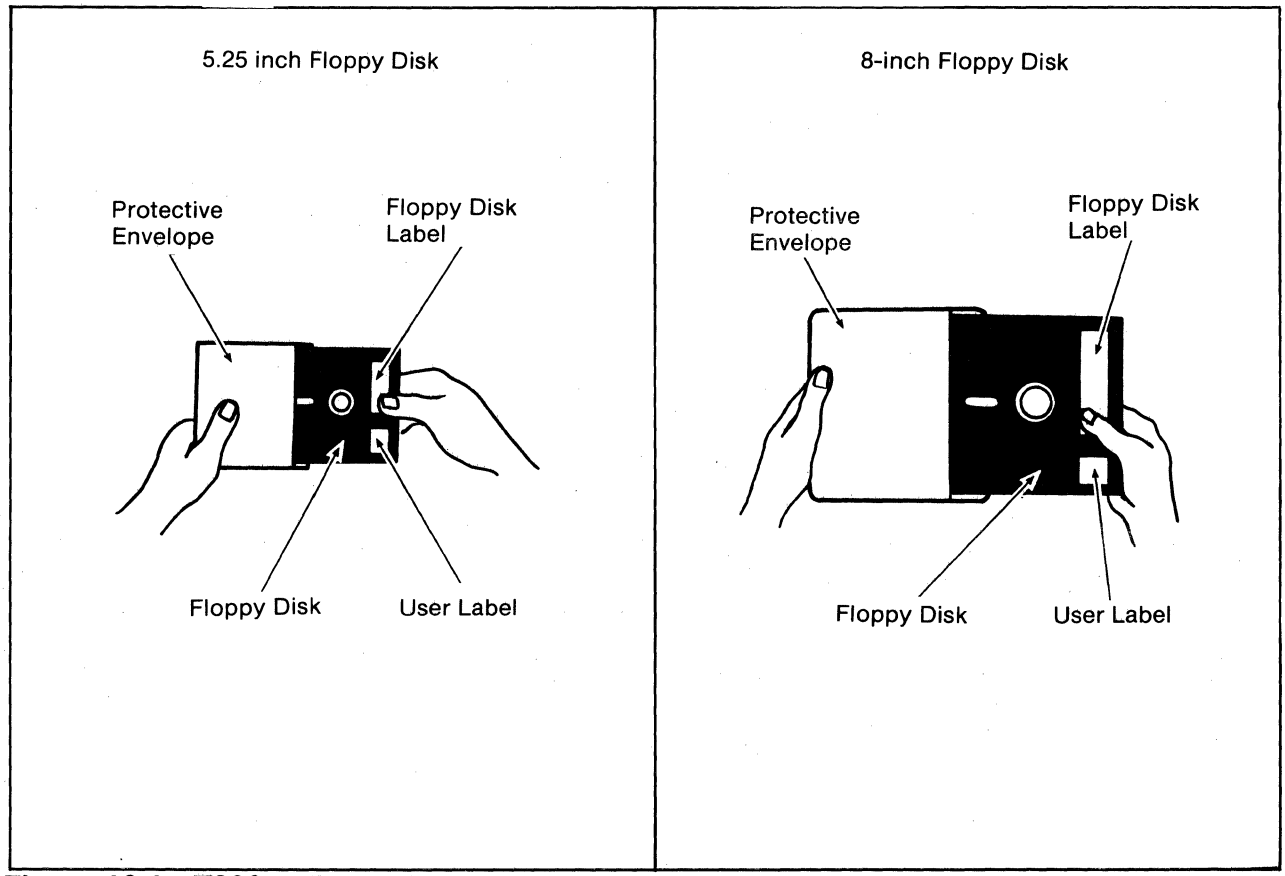


Figure 10.1 T200: *Floppy Disk*

Figure 10.2 T250: *Floppy Disk*

WORKING WITH THE FLOPPY DISKS

You will be working with floppy disks that serve different purposes. The **system disk** contains the operating system which serves as the computer's manager. This disk is delivered with your system, and loaded each time its data is required.

You will also use floppy disks for storing your data or programs. "Prepackaged" programs are available on floppy disks. The amount of information you can store is limited only by the number of floppy disks that you have available.

Regardless of the type of disk you are using, the floppy disks hold information magnetically and require special care. A list of precautions to protect the life of your disks is shown in the chart at left.

Using the Floppy Disks

The T200 and T250 use different sizes of floppy disks. The lower figures at left show the two sizes.

The **T200** has one or two floppy disk drives to mount 5.25-inch two-sided double-density floppy disks.

The **T250** has one or two floppy disk drives that can read and write data on either one-sided single-density 8-inch disks or two-sided double-density 8-inch disks.

The disk type is identified by the label on the jacket. The label shows the number of bytes per sector as "Record Length." "One-sided" is indicated with a **1S** and "two-sided" with a **2D. Appendix C** gives disk characteristics for each type of disk, including the number of sectors and bytes.

Your system may be equipped with one or two disk drives to house the removable disk media. Disk drive **#1** is always the one closest to the screen. Drive **#1** is also called **A.** If the system has two disk drives, the right-hand disk drive **(#2)** is also called **B.**

DISKS, Continued →

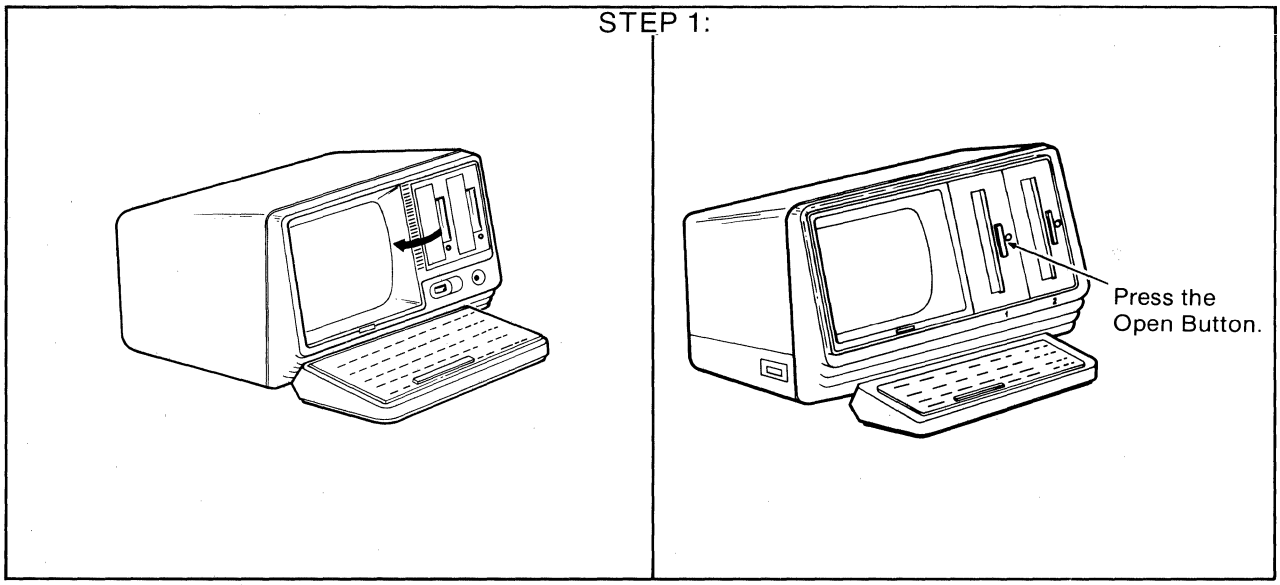


Figure 11.1 T200: *Opening Disk Door*

Figure 11.2 T250: *Opening Disk Door*

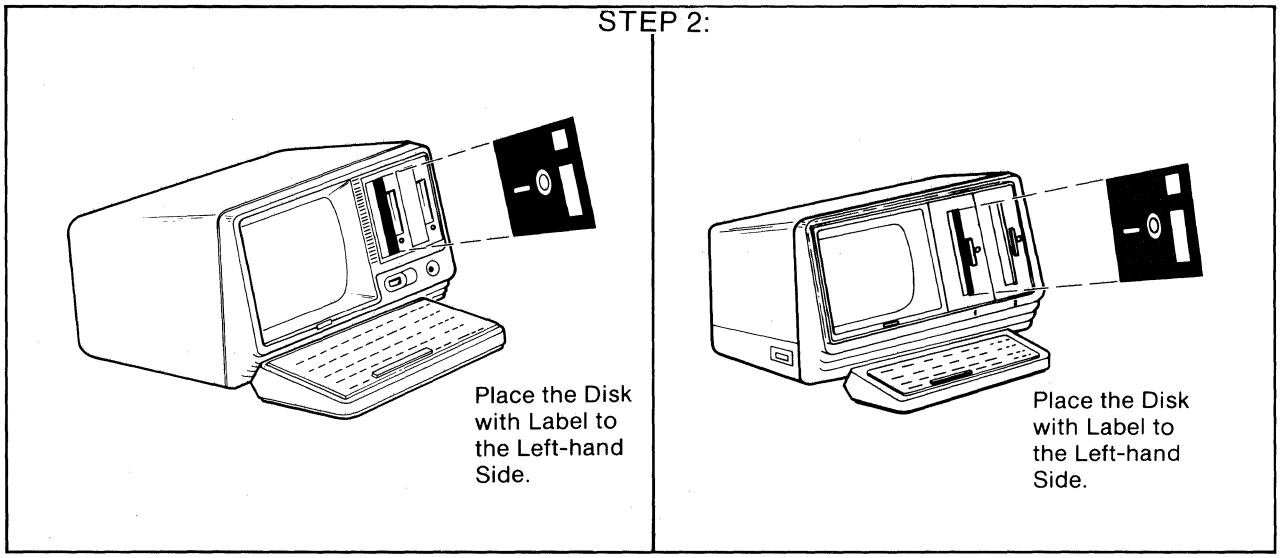


Figure 12.1 T200: *Mounting Disk*

Figure 12.2 T250: *Mounting Disk*

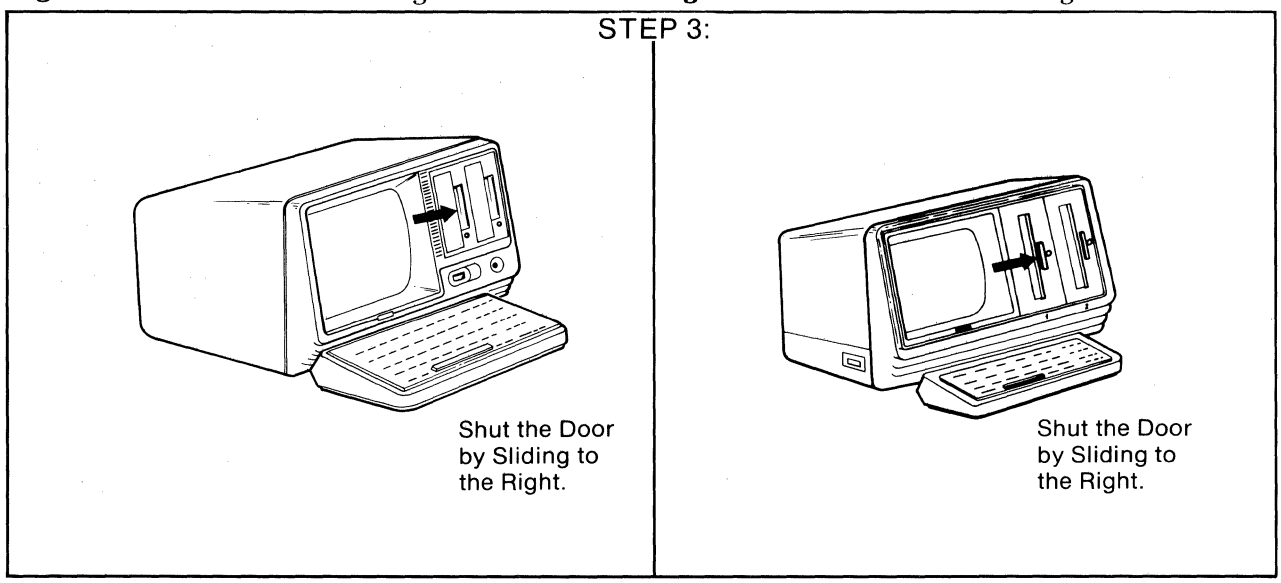


Figure 13.1 T200: *Closing Disk Door*

Figure 13.2 T250: *Closing Disk Door*

Loading and Removing Floppy Disks

LOADING

STEP 1. Open the disk drive door:

T200: Raise the right edge of the door.

T250: Press the disk drive Open Button.

STEP 2. Place the disk in the drive with the label on the jacket to the left side. Be sure the disk is all the way in the drive.

(On the **T250,** a click should be heard.)

STEP 3. Close the door by sliding it firmly to the right.

The three loading steps are illustrated at left.

REMOVING

STEP 1. Confirm that the busy indicator on the disk drive is off. Never remove the disk when the light is on.

STEP 2: Open the disk drive door.

(On the **T250,** press the Open Button. The disk pops out.)

STEP 3: Remove the disk and place it in the protective paper envelope.

STEP 4: Close the door.

NOTE: New disks must be **formatted** before use via software contained in the operating system. See Part 2, "The Operating System."

DISKS, Continued →

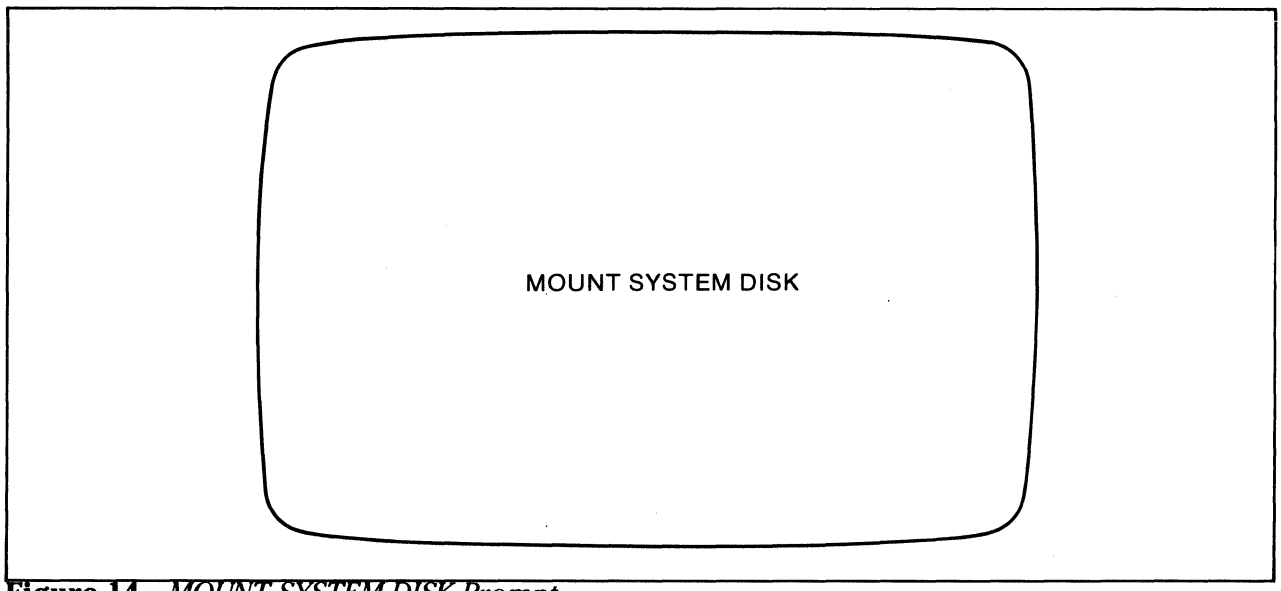


Figure 14 *MOUNT SYSTEM DISK Prompt*

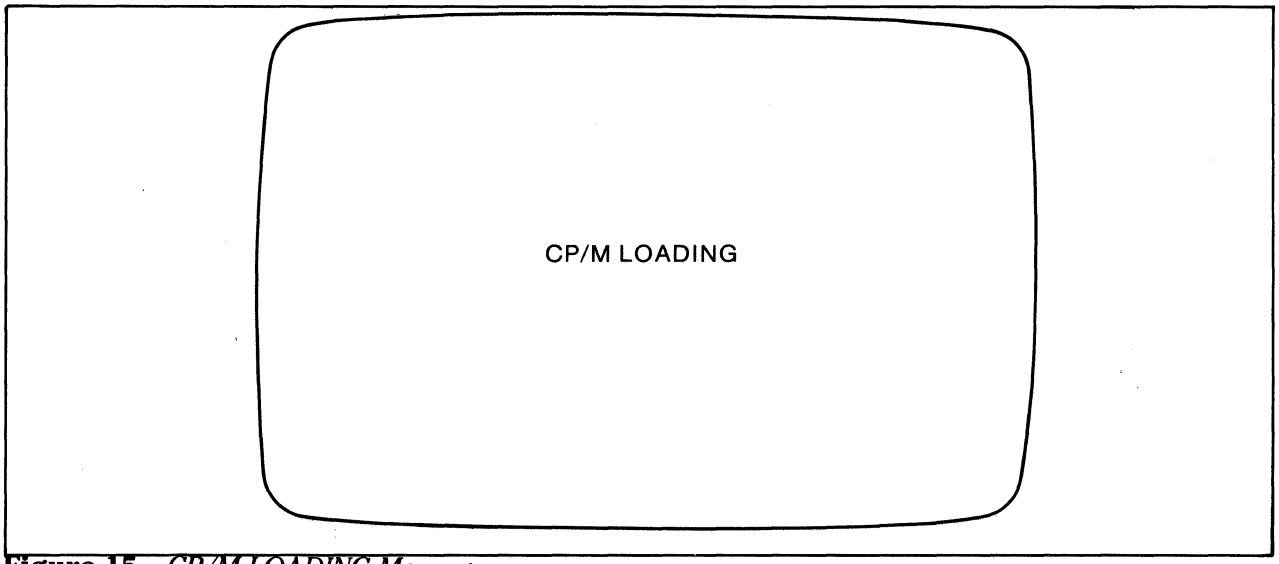


Figure 15 *CP/M LOADING Message*

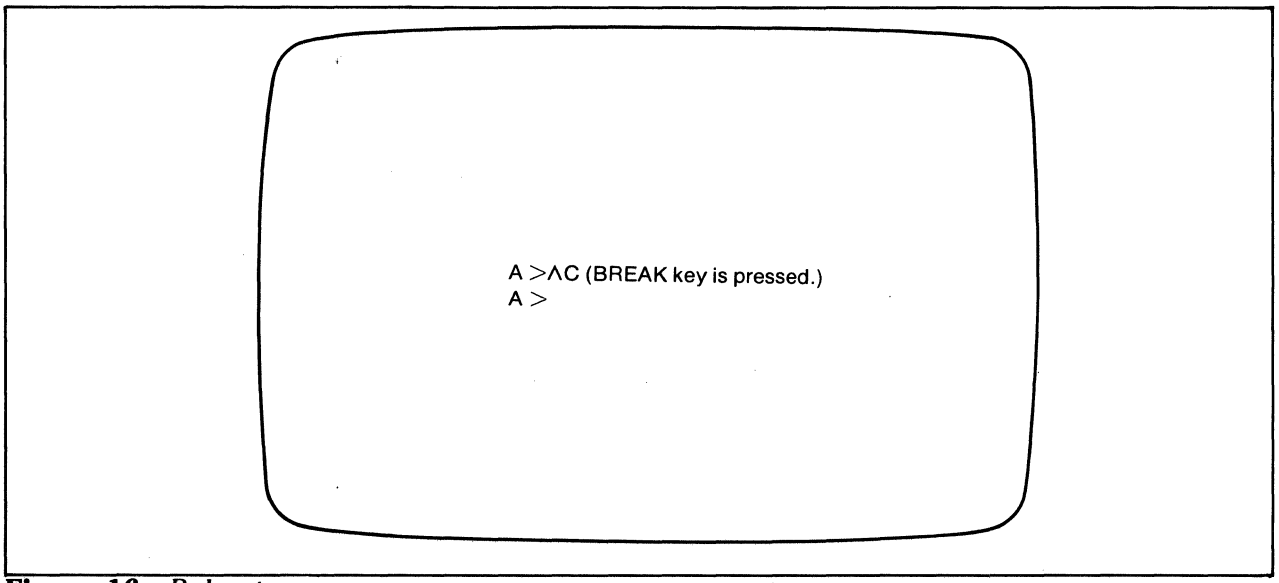


Figure 16 *Reboot*

Using the System Disk

MOUNTING

About five seconds after you have turned on your computer, it will display the statement, **MOUNT SYSTEM DISK.** This prompt is shown in the top figure at left. (If you mounted the system disk before turning on the power, the statement is not displayed and system loading starts automatically.)

You should now place the system disk into disk drive **#1** (drive **A**). Remember, if you have two drives, this is the one closest to the screen.

Mounting the system disk means that you are loading the operating system stored on the system disk into the main memory of the processing unit. While Part 2 will explain how the operating system works, it will help you to know now that all programs are run under the control of the operating system. In addition, some of the instructions that you enter from the keyboard are accepted and processed by the operating system.

NOTE: You may also receive a message to exchange the system disk if you are using a one-sided disk. This is only applicable for the T250.

Using the System Disk

CP/M LOADING (Cold Boot)

After you have successfully mounted the system disk, the message **CP/M LOADING** appears on the screen, as shown in the middle figure at left. For a few seconds while the system disk is loaded internally and the message is displayed, the red busy light is lit on the drive in use.

Once the system disk has been loaded successfully, **A >** appears on the screen. This is known as a "prompt" because it is telling you that you can now enter a command.

This loading procedure is called "Cold Boot." When the console power is already on and you want to reload the operating system by Cold Boot, this can be accomplished by holding the CTRL key and pressing the (IPL) key. If you have not already inserted the system disk, the **MOUNT SYSTEM DISK** prompt will appear.

Using the System Disk

CP/M LOADING (Reboot)

Occasionally, you will receive error messages during the use of your computer. If you do, you will need to **reboot** to restart your operating system.

To reboot, either:

- Press the (BREAK) key, or
- Press the CTRL key and type **C** following the **A >** or **B >** prompt.

These methods are shown in the bottom figure at left.

The operating system will then restart from the outset. This procedure is also called a **warm start.**

NOTE: When the BASIC language is being used (MBASIC), the above methods return MBASIC to command level, rather than performing the reboot.

Winchester Disks

Toshiba plans to introduce Winchester disks in the T200/T250 product line soon. When the Winchester disk is present, it will replace one of the floppy disk drive units.

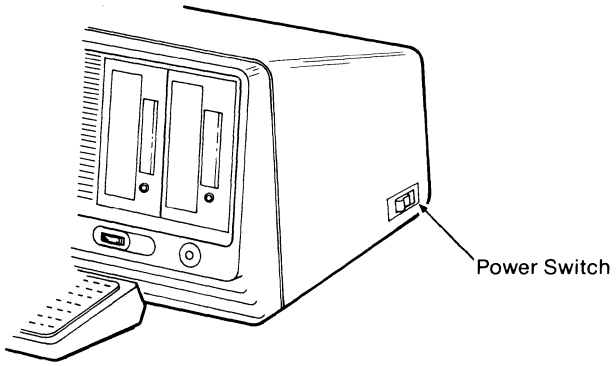


Figure 17.1 T200: *Turning the Power Off*

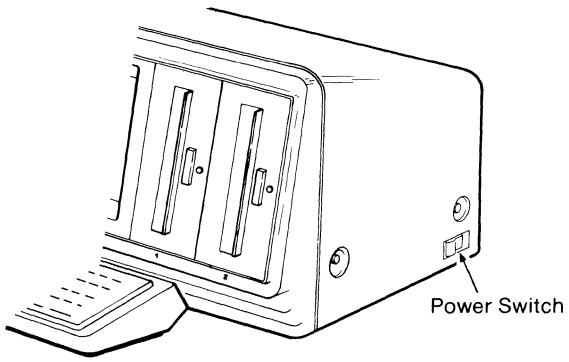


Figure 17.2 *T250:* *Turning the Power Off*

TURNING THE POWER OFF

Follow these steps to turn the power off:

STEP 1. Verify first that:

a. The red light is off on the front of the disk drive.

b. The printer is not printing.

STEP 2. If the disks are inserted in the disk drives, remove them and store in their protective envelopes.

STEP 3. Turn off the power (as shown in the figures at left):

a. On the console

b. On the printer

NOTE: If you have need of communication interfaces, **Appendix E** provides this information.

PART 2

OVERVIEW OF PART 2

Part 2 discusses your computer system's **software**. Whether you buy prepackaged programs, write programs or modify existing ones, you need to understand the operating system software, as well as how to use a computer language on your system. Part 2 includes the following major sections:

- The Operating System
    - Capabilities
    - Parts
    - User Interaction
    - File Names
    - Commands
    - Utilities
    - Error Messages
- BASIC Programs
- Assembler Programs

If you will use only BASIC programs, you do not need to read the assembler programs section of Part 2. Although the debugging tool and text editor are somewhat useful with BASIC, these two facilities are primarily for use with assembly language. Therefore, they are included in the assembler program section.

THE OPERATING SYSTEM

What is the Operating System?

Your computer consists of an interrelated system of devices and programs. While you are the external manager of these components, computers also require an internal manager. For that purpose, the Toshiba T200 and T250 have an operating system designed especially for microcomputers by Digital Research. This system is called **CP/M,** for **CONTROL PROGRAM** for **MICROPROCESSORS.**

Operating System Capabilities

Recall that each time you turn on your computer, you mount the system disk (described in Part 1). This process loads the operating system into your computer. Once you have the operating system loaded into your computer, you can begin to use its capabilities. To do so, you type established commands on the keyboard.

The operating system accepts established commands from the keyboard and translates them into electronic "language" that other parts of the computer can understand.

As you enter information and programs into your computer, you will want a method for keeping track of that data.

The operating system allocates file spaces on disks and allows rapid access to any file. The system allows dynamic allocation of file space, as well as sequential and random file access.

When you write programs, you will need access to the software for that language, and the ability to store the programs.

The operating system supports BASIC, assembler, and other languages. A large number of distinct programs can be stored in both source and machine executable form.

If you write assembly language programs, you will want a method of checking those programs.

The operating system provides a text editor and "debugging" tool.

The Four Parts of the Operating System

In order to carry out its functions, the operating system is divided into four distinct parts. A brief discussion of each will give you a better understanding of how your computer works, and how you need to interact with it.

1. The Console Command Processor (CCP)

You communicate with your computer via the keyboard. In essence, you are setting electronic switches when you press the keys on the keyboard. The **Console Command Processor (CCP)** then reads and translates your commands (switches) into a more complex series of switches. The CCP, therefore, lets you communicate with your computer in simple language similar to English.

For example, the CCP processes commands that list a directory of your files, print the contents of files and control the operation of "transient" programs such as assemblers, editors and debuggers. The standard commands available will be explained in upcoming sections of this manual. In summary, the CCP provides an interface between your keyboard and the remainder of the operating system.

2. The Basic Input Output System (BIOS)

As you know, your T200 or T250 uses at least one disk drive. One of the four parts of the operating system, the **Basic Input Output System (BIOS),** provides access to the disk drive(s). In addition, the BIOS allows you to add other peripherals to your system by changing the peripheral drivers to handle them.

3. The Basic Disk Operating System (BDOS)

In addition to gaining access to the disk drives, the operating system also provides disk management. The part that does this, the **Basic Disk Operating System (BDOS),** controls one or more disk drives that contain independent file directories. The BDOS is the "strategist" in that it implements disk allocation to provide fully dynamic file construction. At the same time, the BDOS minimizes head movement during disk access.

The BDOS allows any file to contain any number of records providing that the file does not exceed the size of any single disk. Each disk can contain up to 256 distinct files. Specific commands are available for working with the files and disks via the BDOS, such as renaming a file.

4. The Transient Program Area (TPA)

The fourth part of the operating system serves as a "juggler" allowing your computer to swap in and overlay additional program segments. The **Transient Program Area (TPA)** holds programs which are loaded from the disk under command of the CCP. For example, during program editing the TPA holds the text editor machine code and data areas. Similarly, programs created under the operating system can be checked out by loading and executing these programs in the TPA.

In addition to the flexibility offered by the TPA, any or all of the four operating system parts can be "overlayed" by an existing program. That is, once a 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. The user's program need only branch to the bootstrap loader at the end of execution, and the complete operating system is reloaded from disk.

See **Appendix G** for patching the operating system.

User Interaction through the CCP

You interact with the operating system primarily through the CCP, which reads and interprets the commands you enter via the keyboard. Upon initial computer startup, you load the system disk into disk drive **#1** (**A**). The CCP displays the message:

Toshiba xxK CP/M VER m.m

where **xx** is the memory size (in kilobytes) which this system manages, and **m.m** is the version number.

In general, the CCP addresses one of two disks which can be inserted into the disk drives (A and B). Following system startup, the operating system automatically logs in disk A and prompts you with the symbol **A >**. This symbol indicates that the CCP is currently addressing disk **A.**

A disk is "logged in" if the CCP is currently addressing it. In order to indicate clearly which disk is logged in, the CCP always prompts you with the disk name followed by the symbol > . When you receive such a prompt, the system will wait until it receives a command from you.

Communicating with the Operating System

As you know, you communicate with the operating system primarily through the CCP. Often, you will be requesting that the computer do certain things with your data. However, when you want to store your data or a program on a disk file, you have to give a name to the file. So before you learn the established commands, it is important that you understand how to formulate file names properly.

Remember, your computer responds to the presence or absence of electrical currents controlled by switches. You must activate the correct keyboard switches for a file name in order for the CCP to translate your request into the desired computer actions with that file.

File Names

Accurately formed file names can consist of just a **primary part** or a primary and **secondary part.** The primary name distinguishes the particular source file. The secondary file name, though optional, is helpful for identification in that it usually specifies the characteristics of files. For example, data files used in Accounts Receivable control may be given a secondary name of **.AR.**

Characteristics of file names are:

- The primary file name consists of one to eight characters.
- The secondary file name, if used, consists of one to three characters.
- If both file names are used, they are separated by a period.
- If just the primary file name is used, it is equivalent to a primary name plus a secondary name consisting of three blanks.
- Uppercase and lowercase characters and numbers are usually used.
- The following special characters can also be used in formulating file names:

  \ ! # $ % & ' ( ) - _ / + @ ^

  The remaining special characters cannot be used for file names.
- Lowercase letters are always translated by the operating system to uppercase when they are entered for command names and file names, unless BASIC is involved. (Lowercase characters entered for a file name with the specially designed BASIC language commands and statements are not translated to upper case.)

If you are working with General Ledger files, you might have file names like the following:

GLTRAX

GLJOURNL

GLACCT.AR

Observe that the longest primary file name (**GLJOURNL**) has the maximum of eight letters. Shorter primary file names (**GLTRAX** and **GLACCT**) are also acceptable, and more desirable since they are logical abbreviations. The secondary file name **.AR** is also used. Note the placement of the period.

For convenience (if you have more than one drive), file names can be prefixed with a drive name (**A** or **B**), followed by a colon (:). The **A** or **B** indicates the disk where the file is located. Examples:

A:GLTRAX

B:GLACCT.AR

Observe that the drive letter and the colon are counted in the maximum of eight characters allowable for the primary part.

NOTE: A special case occurs with secondary names for BASIC files in the following situation. BASIC automatically supplies a **.BAS** if no period (.) appears in the file name, when given with the SAVE command, and the entire file name is less than nine characters long. The SAVE command will be explained in an upcoming section.

Ambiguous File References

When you want to identify a unique file on a particular disk attached to the operating system, you can call out the exact file name. It is termed the **unambiguous** reference, since the file name refers only to that one unique file. You also have the option to specify part of the file name in an **ambiguous,** or "wild card," reference. This type of reference enables you to locate all files, or a subset of all files, in a particular group of files. There are two basic ways to use the wild card file reference. One method uses the * symbol; the other uses the ?.

Using the *

Using the * is the simplest way to make wild card, or ambiguous file references. For example, to get the names of all files beginning with **GL** and having no secondary file name:

TYPING	PRODUCES FILES IN THIS FORM
GL*	GLTRAX
	GLJOURNL

To get the names of files beginning with **GL** and also containing a secondary file name:

TYPING	PRODUCES FILES IN THIS FORM
GL*.*	GLACCT.AR
	GCLACCT.PAY

The following chart summarizes the use of the asterisk (*) in wild card (ambiguous) file reference:

AMBIGUOUS FILE REFERENCE	RECEIVE
.	All files on a disk.
ppppppppp.*	All files on a disk with a primary name of up to 8 specified letters.
*.sss	All files on a disk with a secondary name of up to 3 specified letters.

Using the ?

The question mark symbol (?) matches any character of a file name in the ? position. A maximum of eight question marks to the left of the period in a file name and a maximum of three question marks to the right of the period are allowed (????????.???). In this method of wild card file reference, the total number of characters and question marks determines which files will be found. For example:

TYPING	PRODUCES THESE FILE TYPES
GL???	GLTAX
GL????	GLTRAN
GL??????	GLJOURNL
GL????.??	GLACCT.AR
GL????.???	GLACCT.PAY
??????.AR	GLACCT.AR
??????.PAY	GLACCT.PAY
X?Z.COM	XYZ.COM
X?Z.C?M	XYZ.COM
	X3Z.CAM
????????	All files on a disk with no secondary file name.
????????.???	All files on a disk.

You may have already deduced that some ambiguous file references using the ? symbol would produce the same result as those using the * symbol:

.	=	????????.???
pppppppp.*	=	**pppppppp.???**
*.sss	=	????????.sss
B:*.BAS	=	B:??????.BAS

Note that it is quicker to use the asterick (*) when you are not specifying any of the characters to the left and/or right of the period.

Commands

Now that you have a general understanding of formulating file names, you can begin to learn about the commands, many of which deal with the files in various ways. Two types of commands are available. The first type is called **built-in**. That means that the commands are usable even when the system disk (which contains the Command Console Processor) has been removed from the disk drive. Recall that the operating system is loaded from the system disk into memory when you turn on your computer and perform a "boot" or "cold start." The TPA then executes the built-in commands as you use them.

The second type of command is called **transient**. When you use transient commands, the currently logged disk must have had the system area copied onto it from the actual system disk. While this procedure is described later in "Disk-to-Disk Copying," it is important you understand now this prerequisite for transient command use. The system portion can be copied onto blank floppy disks, as well as those containing programs. Transient commands are also important because they allow you the capacity to define your own additional transient commands. This procedure is described under the LOAD command section.

You can enter commands when the operating system is prompting you for input via the **A>** or **B>** prompt. You type the command on the keyboard and it appears on the screen as you type it. To submit any command to the operating system for action, you always have to press the carriage return key. Remember that the CCP translates these letters into upper case, so you need not bother holding the SHIFT key.

The following pages describe the individual commands. If you will use mainly BASIC programs, you will probably need only three of the built-in and two of the transient commands. These commands will be presented in the first two upcoming sections. Other additional commands will follow. A quick-reference list of the commands is given after that, including unambiguous and/or ambiguous file references as required by the commands.

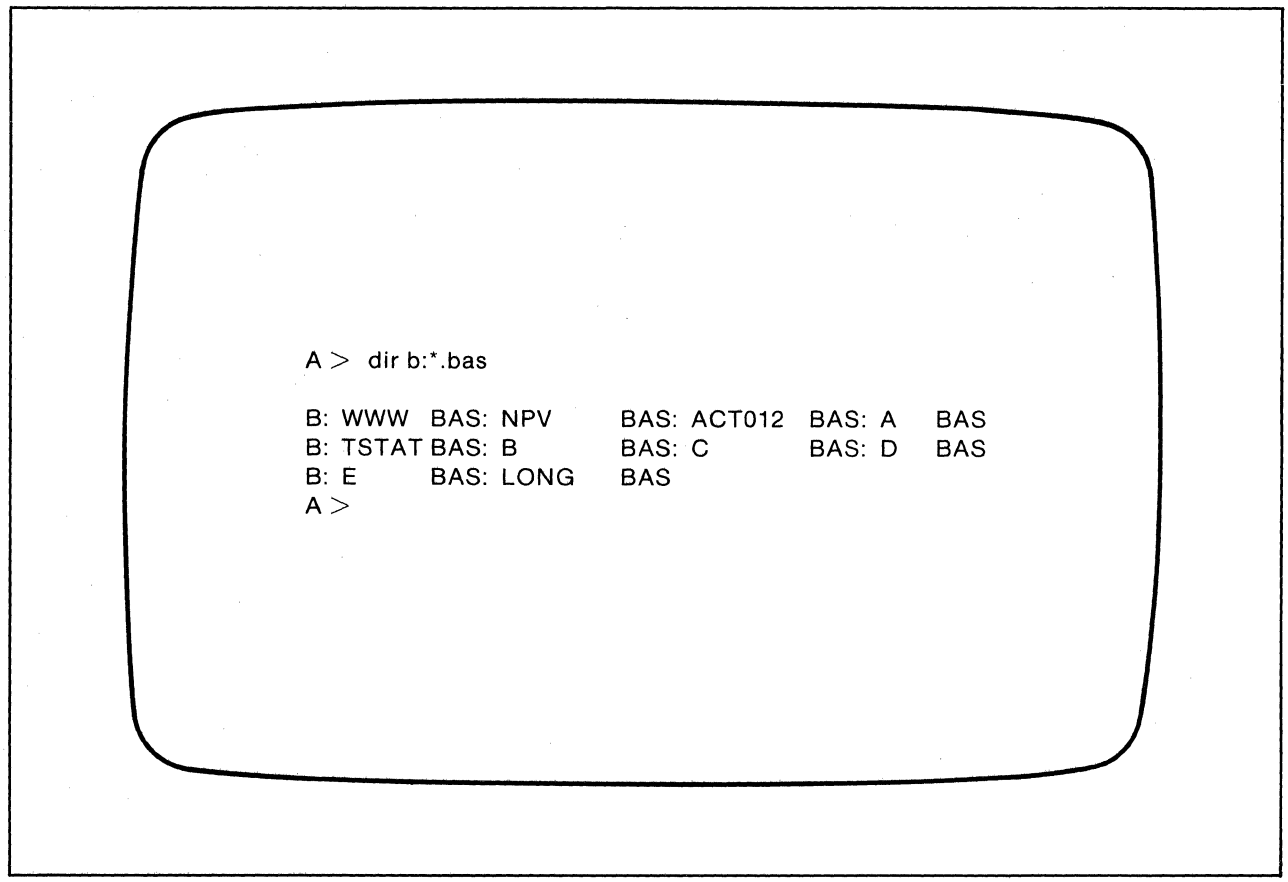


Figure 18 *DIR Command*

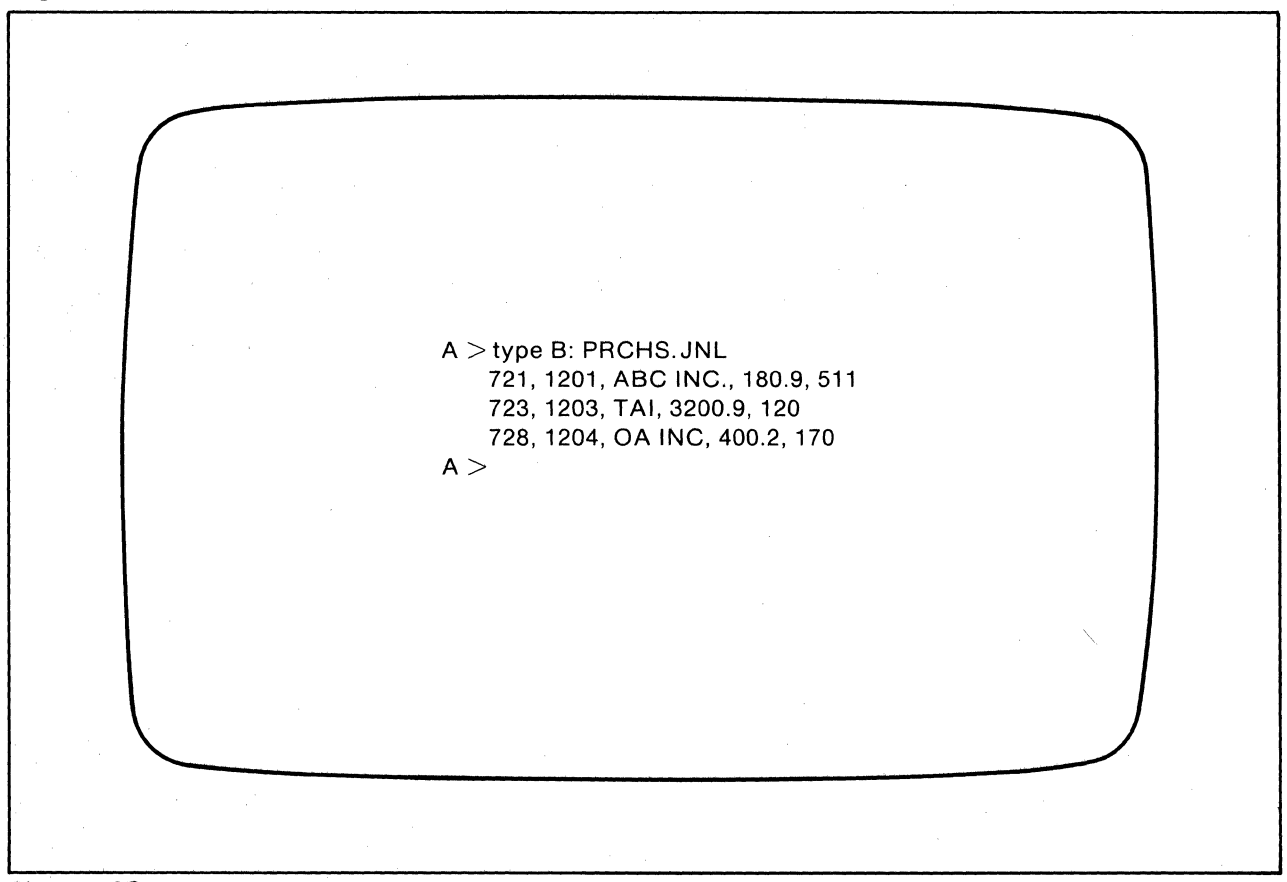


Figure 19 *TYPE Command*

DIR

The **DIR** (**directory**) command causes the names of all files on a disk which satisfy the ambiguous file reference to be displayed on the screen. The following are examples given to demonstrate variations of this command. You of course will have your own file names, and just drive **A** if your system has one disk drive. Remember that you press the carriage return key after a command.

COMMAND FORM	FILE NAMES DISPLAYED
DIR	All files on the current disk.
DIR *.*	Same as DIR above.
DIR A:	All files on the disk inserted in drive **A.**
DIR B:*.TS	All files on the disk in drive **B** with the secondary name **TS.**
DIR B:*.BAS	All BASIC file names on the disk in drive **B.** (This example is shown in the top figure at left.)

Other valid commands are:

DIR X.Y DIR X?Z.C?M DIR B:X.Y

NOTE: If no files can be found on the selected disk to satisfy the directory request, either a **?**, **NOT FOUND** or **NO FILE** is displayed on the screen.

TYPE

The **TYPE** command types on the screen the contents of an unambiguous source file.

COMMAND FORM	DISPLAYS
TYPE X.Y	The contents of file X.Y.
TYPE XXX	The contents of file XXX.
TYPE B:X1	The contents of file X1 on drive **B.**
TYPE B:PRCHS.JNL	The contents of the **PRCHS.JNL** file on drive **B** (shown in the lower figure at left.)

NOTE: The TYPE command expands tabs (CTRL-I characters), assuming tab positions are set at every eighth column.

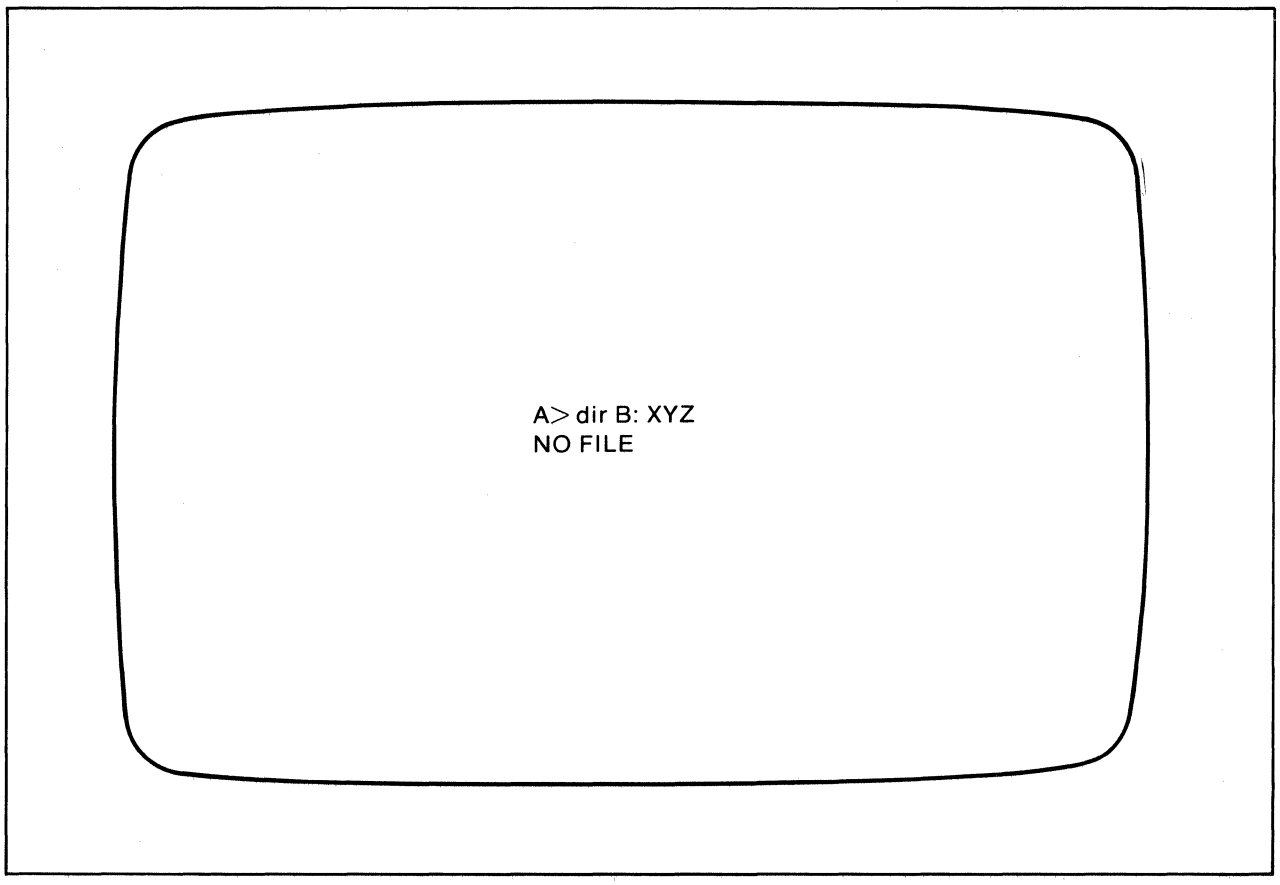


Figure 20 *No File Found*

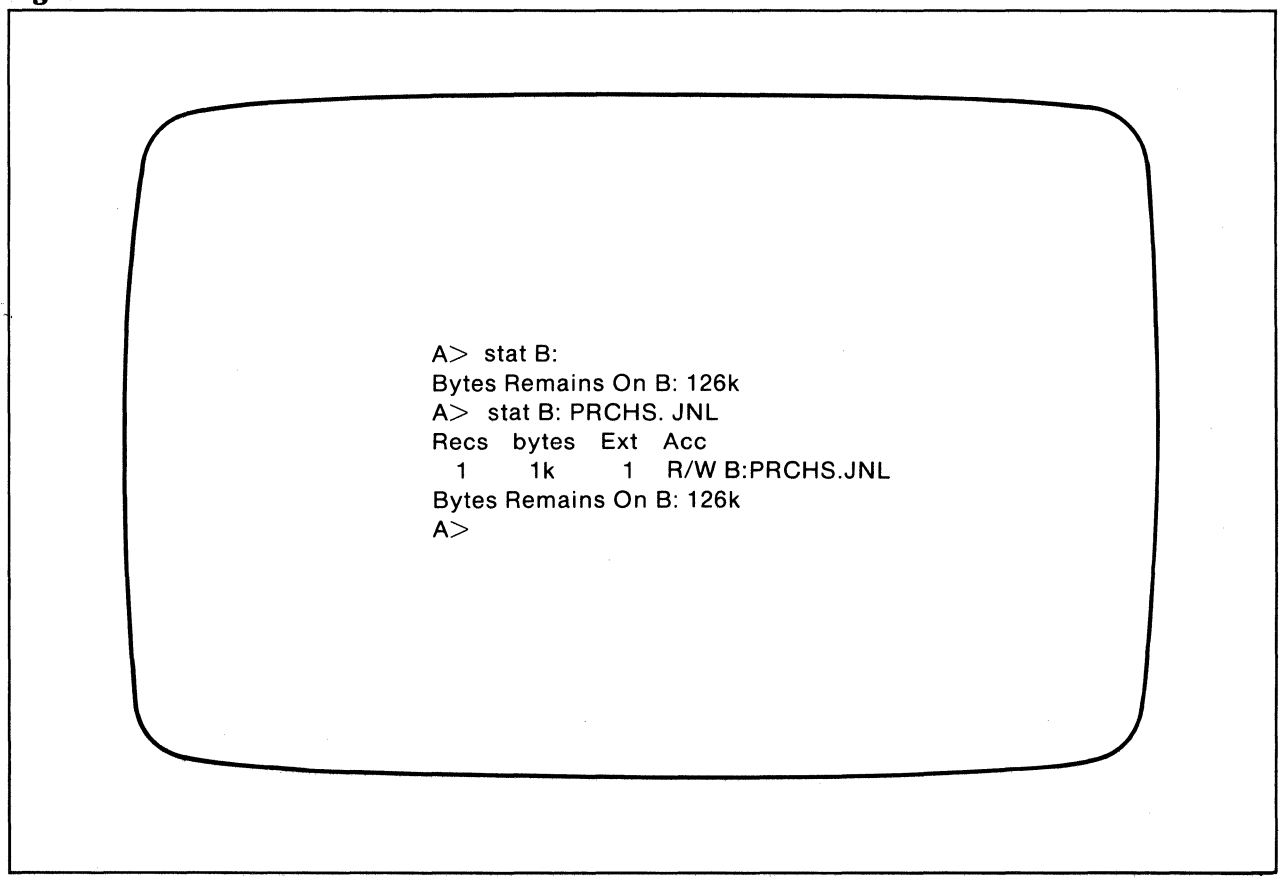


Figure 21 *STAT Command*

ERA

The **ERA (erase)** command removes the specified file(s) from the currently logged disk and makes available the space the file(s) once occupied. You may use either an unambiguous reference to erase one file, or an ambiguous reference to erase a group of files. Once the file is erased, you should not see the file displayed when you enter the DIR command. The figure at top left gives a sample response to such a request.

COMMAND FORM	FILE(S) ERASED FROM CURRENT DISK
ERA X.Y	X.Y
ERA X.*	All files with primary name X.
ERA X?Y.C?M	All files which satisfy X?Y.C?M.
ERA *.*	All files on the current disk. (Before this extreme command is enacted, the CCP displays the message, ALL FILES **(Y/N)**? A **Y** response for yes must be given before the files are actually removed.)
ERA B:*.PRN	All files on drive B which satisfy the ambiguous reference ????????.PRN (independent of the currently logged disk).

PART 2

Transient Commands for BASIC Users

STAT

The **STAT (status)** command provides system status information about file storage.

COMMAND FORM	DISPLAYS
STAT	The storage remaining on all active disk drives (with the drive letter, read/write (RW) or read only (R/O), the remaining space in kilobytes).
STAT B:	The storage remaining on drive **B.** Drive **A** may be currently logged. (See Figure 21).
STAT B:PRCHS.JNL	Information about file PRCHS.JNL on disk **B.** (See Figure 21)
STAT X?Y.C?M	Individual and summarized information about the files (in alphabetical order) which satisfies the ambiguous file reference:

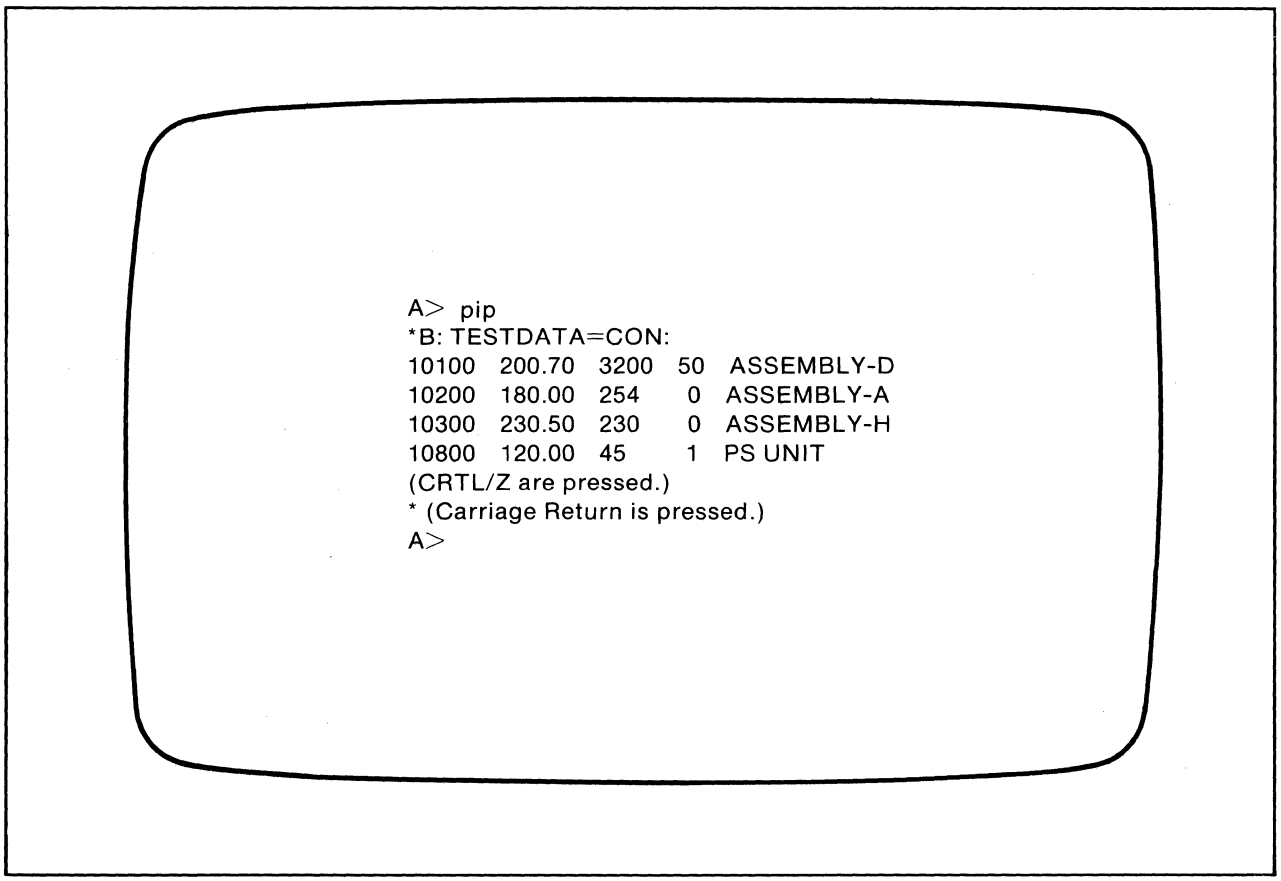


Figure 22 *PIP Command*

RECS = The number of 128-byte records allocated to the file.

BYTES = The number of kilobytes for the file. (BYTES = RECS*128/1024)

EXT = The number of 16K extensions. (EXT = BYTES/16)

FILE NAME/TYP = The primary/secondary file name.

NOTE: The STAT command can also be used for device assignment, as explained in "Other Transient Commands;" and to set a drive to read-only. (See "Write Protecting Disks.")

PIP

The **PIP (Peripheral Interchange Program)** command allows copying and combining of disk files. PIP enables you to work with a **destination** file which receives data and a **source** file (including the keyboard) which delivers the data.

COMMAND FORM	RESULT
Simple Copying:	
PIP X=Y	File Y is copied to File X. File Y remains unchanged.
Linking Files:	
PIP X=Y, Z	Files Y and Z are concatenated (linked together) and copied to File X. Files Y and Z are unchanged.
Different Disks:	
PIP NEW.M=B:OLD.M	A copy of OLD.M is moved from drive **B** to the currently logged disk (**A**). The new file is named NEW.M.
Using Ambiguous File Names:	
PIP A:=B:*.*	All files on disk **B** are copied to **A** with the same file names.
PIP B:TESTDATA=CON:	The file TESTDATA is created on disk **B** by reading the keyboard (CON:) input until the CTRL and Z keys are pressed simultaneously.

The above example is shown in the Figure 22. The user received an asterisk when just PIP was entered.

Ambiguous File, Continued

PIP A:=GL*	All files which satisfy GL* are copied from the currently logged disk to the same file names on drive **A.** Each unambiguous file name is listed as it is copied.
PIP B:=*.COM	All files which have the secondary name COM are copied to drive **B** from the currently logged drive.
PIP A:=B:ZAP.*	All files which have the primary name ZAP are copied from **A** to **B.**

Same File Names on Different Disks:

PIP B:=GAMMA.BAS	Equivalent to B:GAMMA.BAS=GAMMA.BAS
PIP B:=A:GAMMA.BAS	Equivalent to B:GAMMA.BAS=A:GAMMA.BAS

PIP

Other Points Regarding the PIP Command:

- Information from the source is copied left to right to the destination.
- The copy operation can be terminated at any time by pressing any key on the keyboard.
- If you form the PIP command properly, and the destination file exists, it is removed and replaced with the source file data. The destination file is not changed if an error condition exists.
- If the destination file also appears as one or more of the source files, the source file is not altered until the entire concatenation is complete.
- Other less frequently used PIP capabilities are also listed under "Other Transient Commands."

The following two sections:

Other Built-in Commands

Other Transient Commands

explain command forms for use with languages other than BASIC. Therefore, if you plan to use just BASIC, skip over these sections to "Line Editing and Output Control."

REN

The **REN (rename)** command renames a specified file that exists on a disk. The currently logged disk is assumed to contain the file to be renamed.

COMMAND FORM	RESULT
REN X.Y=Q.R	The name of file Q.R is changed to X.Y.
REN XYZ.COM=XYZ.XXX	The name of file XYZ.XXX is changed to XYZ.COM.
REN A:X.ASM=Y.ASM	The file Y.ASM is renamed to X.ASM on drive **A.** (If one name is preceded by a drive name and the other is not, both file names are assumed to be on the same disk.)
REN B:A.ASM=B:A.BAK	The file A.BAK is renamed to A.ASM on drive **B.**

NOTE: If the file name you wish to use is already present on the drive, the REN command will respond with the error message, **FILE EXISTS.** The change will not be performed.

If the file name you wish to rename does not exist on the specified disk, **NOT FOUND** is printed on the screen.

The **SAVE** command literally saves information by placing a specified number of pages (256-byte blocks) onto disk from the TPA. SAVE also names this as a file. In the operating system distribution system, the TPA starts at 100H (hexadecimal), which is the second page of memory. Thus, if the user's program occupies the area from 100H through 2FFH, the SAVE command must specify two pages of memory. The machine code file can subsequently be loaded and executed. The SAVE operation can be used any number of times without altering the memory image.

COMMAND FORM	RESULT
SAVE 3 X.COM	Copies 100H through 3FFH to X.COM.
SAVE 40 Q	Copies 100H through 28FFH to Q. (Note: 28 is the page count in 28FFH, and 28H=2*16+8=40 decimal.)
SAVE 4 X.Y	Copies 100H through 4FFH to X.Y.
SAVE 10 B:ZOT.COM	Copies 10 pages (100H through OAFFH) to the file ZOT.COM on drive B.

Other Transient Commands

ASM

The **ASM (assembler)** command loads the assembler and assembles the specified program from disk. You provide a file name after the ASM command. The secondary file name ASM is assumed and thus need not be specified.

COMMAND FORM	RESULT
ASM GAMMA	The two-pass assembler is automatically executed for the source file GAMMA.
ASM B:ALPHA	The assembler is loaded from the currently logged drive and operates on the source program ALPHA.ASM on drive **B.**

If assembly errors occur during the second pass, the errors are displayed on the screen. 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 using the TYPE command, or sent to a peripheral device using PIP. Note also that the PRN file contains the original source program, augmented by miscellaneous assembly information in the leftmost 16 columns (for example, program addresses and hexadecimal machine code). Thus, the PRN file can serve as a backup for the original source file. If the source file is accidentally removed or destroyed, the PRN file can be edited. (See the text editor section.) The editing is accomplished by removing the leftmost 16 characters of each line (issuing a single editor "macro" command). The resulting file is identical to the original source file and can be renamed (REN) from PRN to ASM for subsequent editing and assembly. The file

x.HEX

is also produced which contains 8080 machine language in Intel "hex" format suitable for subsequent loading and execution. (See the LOAD command.) For complete details of the operating system's assembly language program, see the assembly programs section.

DDT

The **DDT (Dynamic Debugging Tool)** command is used to load the debugger into the TPA and start execution. The use of this command is explained in detail under the debugging tool section.

DUMP

The **DUMP** command initiates a program which types the contents of an unambiguous disk file on the screen 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 DEL key during printout.

ED

The **ED** command controls the program which is the operating system's context editor. It allows creation and alteration of ASCII files in the operating system environment. The use of this command is explained in detail in the text editor section.

LOAD

The **LOAD** command reads the specified unambiguous file, which is assumed to contain **hex** format machine code, and produces a memory image file which can be subsequently executed. The file name 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 > which is 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 subsequently be executed any number of times simply by typing the primary name. In this way, the user can "invent" new commands in the CCP. (Formatted 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 the DDT.

SUBMIT

The **SUBMIT** command allows CP/M commands to be batched together for automatic processing. The unambiguous file name given in the SUBMIT command must be the name of a file which exists on the currently logged disk, with an assumed file type of **SUB.** The SUBMIT command takes the form:

SUBMIT filename p1 p2 p3 . . . pn

where **p1** through **pn** are actual parameter values.

The SUB file is created with the ED program like any other file. It contains CP/M prototype commands, with dummy parameters that allow substitution of actual parameter values at execution time. The dummy parameters take the form:

$i

where **i** is an integer. For the first such parameter, i must equal 1, for the second i=2, for the third i=3, and so on. For example, SUBMIT file TYPICAL.SUB might contain:

DIR $1:$2 (cr)

PIP A:=$1:$2 (cr)

where **$1** and **$2** are the dummy parameters that function as variables, accepting the values of the actual parameters at execution time. The actual parameter values following the file name are substituted into the dummy parameters. If no errors occur, the file with substituted parameters is processed sequentially by CP/M.

When the SUBMIT transient is executed, the actual parameter values pl . . . pn are paired with the dummy parameters $1 . . . $n in the prototype commands. If the number of dummy and actual parameters does not correspond, then the SUBMIT function is aborted with an error message on the screen.

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 pressing the DEL key when the command is read and echoed. In this case, the $$$.SUB file is removed, and the subsequent commands come from the keyboard. Command processing is also aborted if the CCP detects an error in any of the commands. Programs which execute under CP/M can abort processing of command files when error conditions occur simply by erasing any existing $$$.SUB file.

In order to introduce dollar signs into a SUBMIT file, you may type a **$$** which reduces to a single **$** within the command 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 (press carriage return)

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.**

THE XSUB FUNCTION

XSUB 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 10) 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
G0
SAVE 1 $2.COM
```

with a subsequent SUBMIT command:

```
SUBMIT SAVER X Y
```

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 "G0" 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.

SYSGEN

This utility adds operating system patch TPATCH to the released version of CP/M, generating a version 2.xx BIOS and writing it on a newly formatted disk on drive **B.** To perform SYSGEN:

following the system prompt, type SYSGEN

place the newly formated disk in drive **B.**

NOTE: BREAK will abort the SYSGEN process.

* * * * * * *

The following two commands, PIP and STAT, were also discussed under commands for BASIC users. More uses for them are detailed below.

PIP

PIP 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), **LST:** (list) while the physical devices are

CRT: (console or list)

LPT: (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.) The destination device must be capable of receiving data (that is, data cannot be sent to the punch), and the source devices must be capable of generating data (that is, the LST: device cannot be read).

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

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

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

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

OUT: Special PIP output destination which can be patched into the PIP program: PIP CALLs location 106H with data in register C for each character to transmit. Note that locations 109H through 1FFH of the PIP memory image are not used and can be replaced by special purpose drives using the debugging tool.

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

File and device names can be interspersed in the PIP commands. In each case, the specific device is read until end-of-file (CTRL-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 (CTRL-Z) is appended to the result for ASCII files. Note that 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 DEL 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.

Valid PIP commands are shown below

COMMAND FORM	RESULT
PIP LST:=X.PRN	Copies X.PRN to the LST device and terminates the PIP program.
PIP	Starts PIP for a sequence of commands (PIP prompts with *).
*CON:=X.ASM,Y.ASM,Z.ASM	Concatenates three ASM files and copies to the CON device.
*X.HEX=CON:,Y.HEX,PTR:	Creates a HEX file by reading the CON (until a CTRL-Z is typed), followed by data from Y.HEX, followed by data from PTR until a CTRL-Z is encountered.
PIP PUN:=NUL:,X.ASM,EOF:,NUL:	Send 40 nulls to the punch device; then copy the X.ASM file to the punch, followed by an end-of-file (CTRL-Z) and 40 more null characters.

You can stop PIP with a single carriage return.

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 (CTRL-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 l. Leading zeros 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=l or is excluded altogether, page ejects occur every 60 lines. If the F parameter is used, form feed suppression takes place before the new page ejects are inserted.

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

Tn Expand **Tabs** (CTRL-I characters) to every **n**th column during the transfer of characters to the destination from the source.

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

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

W **Write** over read only files without console interrogation. If the operation involves several concatenated files, the W parameter need only be included with the last file in the list.

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

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

COMMAND FORM	RESULT
PIP X.ASM=B:[v]	Copies X.ASM from drive **B** to the current drive and verifies that the data was properly copied.
PIP LPT:=X.ASM[nt8u]	Copies X.ASM to the LPT: device; numbers each line, expands tabs to every eighth column, and translates lower case alphabetics to upper case.
PIP PUN:=X.HEX[i],Y.AOT[h]	First copies X.HEX to the PUN: device and ignores the trailing ":00" record in X.HEX; then continues the transfer of data by reading Y.ZOT, which contains hex records, including any ":00" records which it contains.

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

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

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 skipped, 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]

The above 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, the system files are not recognized. The command line

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 an R/O and system file. The system file attributes are copied, if present.

STAT

The **STAT** command allows control over the physical to logical device assignment. (See the IOBYTE function in the system entry points section, and the "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: Input device (i.e., RS232 port).

PUN Output device (i.e., RS232 port).

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:

CRT: Cathode ray tube device (high speed console).

BAT: Batch processing (console is current RDR:, output goes to current LST: device).

LPT: Line printer (Centronics port).

COM: RS232 Communications port.

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

STAT VAL: [carriage return]

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

CON: =	CRT:	LPT:	BAT:	COM:
RDR: =	CRT:	COM:	COM:	COM:
PUN: =	CRT:	COM:	LPT:	LPT:
LST: =	CRT:	LPT:	LPT:	COM:

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: [carriage return]

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:	=	COM:
PUN:	=	COM:
LST:	=	LPT:

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

STAT ldl = pdl, ld2 =pd2, . . . , ldn = pdn [carriage return]

where ldl through ldn are logical device names, and pdl through pdn are compatible physical device names (that is, 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: [carriage return]

STAT PUN:=TTY:, LST: = LPT:, RDR: = TTY: [carriage return]

COMMAND QUICK-REFERENCE LIST

COMMAND CALL	BASIC/ OTHER	TRANS/* BUILTIN	AFN/ UFN**	DESCRIPTION OF COMMAND	MANUAL PAGE #
ASM	Other	Trans	UFN	Loads assembler	42
DIR	BASIC	Builtin	AFN	Lists file directory	35
DDT	Other	Trans	UFN	Loads debugger	42
DUMP	Other	Trans	UFN	Dumps file in hex	42
ED	Other	Trans	UFN	Loads editor	43
ERA	BASIC	Builtin	Either	Erases a file	37
LOAD	Other	Trans	UFN	Loads file in hex	43
PIP	BASIC	Trans	Either	Loads Per. Inter. Prog.	39, 46
REN	Other	Builtin	UFN	Renames a file	41
SAVE	Other	Builtin	UFN	Saves pages from TPA	41
STAT	BASIC	Trans	Either	Gives status (space)	37, 49
SUBMIT	Other	Trans	UFN	Submits file for proc.	43
SYSGEN	Other	Trans	---	Creates new CP/M disk	46
TYPE	BASIC	Builtin	UFN	Types file contents	35

* For transient commands, the currently logged disk must contain a system disk portion. This is not required for built-in commands.

** AFN = Ambiguous File Name

UFN = Unambiguous File Name

Line Editing and Output Control

All users occasionally make mistakes while typing at the keyboard. The CCP offers various ways for correcting mistakes. Three of the methods (backspace, DEL key, CAN key) are for correcting the input of **data.** These were discussed in Part 1 under "Line Editing Keys." These three methods can also be used for correcting the entry of **commands**. Additional line editing of commands, both for mistakes and other purposes, is performed by pressing the CTRL key and one other specified key simultaneously:

CTRL - C (BREAK KEY)	Reboots the operating system (a warm start) when used at the beginning of a line. May be used when error messages are encountered.
CTRL - E	Returns the carriage from the end of a line to the beginning of the next line, but the line is not submitted to the computer for action until the carriage return key is pressed. (CCP command lines can generally be up to 128 characters in length.)
CTRL - H (← KEY)	Backspaces one character position.
CTRL - J	Terminates the current input (line feed).
CTRL - M	Terminates the input (carriage return).
CTRL - R	Retypes the current command line, deleting any characters corrected with the DEL key (types a "clean" line). This is helpful since the DEL key leaves the wrong character on display, even though the correct key is also displayed and is the only one effectively sent to the computer for action.
CTRL - U	Deletes the entire line typed at the console. (Same result as pressing the CAN key.)
CTRL - X	Same as CTRL - U.
CTRL - Z	Ends the input from the keyboard. (This is used for PIP and ED commands.)

In contrast to the above input controls, output controls are also available:

CTRL - P	Copies 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 this same key combination is typed again.
CTRL - S	Temporarily stops the output displayed on the screen so that a segment of the output can be viewed. When the next CTRL and S combination is simultaneously typed, the program execution and output continue.

In addition, pressing the key labeled:

(H-COPY) P.STOP	Sends all subsequent output to both the screen and the printer until this key is pressed again.

Disk to Disk Copying

On the T250, to copy disks, type UTIL and carriage return. This brings the following copy options to the screen in menu format:

1. Copy entire disk from Drive A to Drive B.
2. Copy CP/M system tracks only.
3. Copy disk in Drive A using one drive only.

F. Go to FORMAT menu.

X. Exit this menu program and return to CP/M.

Then select the type of copy you wish to perform from the menu.

To copy disks on the T200, type COPY.COM. This initiates a series of three screens that step you through the copy process. Screen #1 displays:

TOSHIBA T200 CP/M COPY UTILITY

Step 1> Insert SOURCE disk in Drive # 1

Step 2> Insert FORMATTED disk in Drive # 2

Press RETURN to begin copy or BREAK to end

After pressing RETURN, Screen # 2 displays the following information:

TOSHIBA CP/M COPY UTILITY

Copying Disk Now . . . . . Please wait

Track # X X

During the copy operation, the X X indicating the track number will cycle from 00 through 35 and then from 36 through 70. Then, the screen will contain the information shown on Screen # 3, unless one of these error messages appear on Screen #2:

Bad Disk Please Try Another !

OR

Not a FORMATTED disk in Drive #2

Screen #3 will contain the following information if the copy is completed successfully:

TOSHIBA T200 CP/M COPY UTILITY

Copy Now Complete ! . . . . Remove Disks

Press RETURN to continue copy or BREAK to end

If there is a problem completing the copy successfully at this point, line 2 of Screen #3 displays:

Copy NOT Complete . . . . Please Try Again

Setting Up New Disks (Formatting)

Before you use new floppy disks for storing data, you must first "format" them. This process is necessary since it prepares the format of your disks for storing information properly. Appendix F describes the storage layout of floppy disks.

On the T250, to perform formatting, begin with the menu used above for the T250 under "Disk to Disk Copying." Select option F from the menu, which will bring the following format options to the screen in menu format:

1. Format the disk in Drive B as a 256 sector double sided double density disk.
2. Format the disk in Drive B as a 128 sector single sided disk.

C. Go to COPY menu.

X. EXIT this menu program and return to CP/M.

Select the type of formatting you want from the menu.

On the T200, to format disks, type FORMAT.COM. This initiates a series of three screens that step you through the copy process. Screen # 1 displays:

TOSHIBA T200 CP/M FORMAT UTILITY

Step1> Insert A NEW disk in Drive # 2

REMEMBER>>If this disk has data on it this will erase it totally.

Press RETURN to begin format or BREAK to end

After pressing RETURN, Screen # 2 displays:

TOSHIBA T200 CP/M FORMAT UTILITY

Formatting Disk Now . . . . . Please wait

Track # X X

During formatting, the track number cycles from 00 - 35 twice. The system then displays Screen # 3 unless this error message appears on screen # 2:

Bad Disk Please Try Another !

If the formatting is completed successfully, Screen # 3 displays:

TOSHIBA T200 CP/M FORMAT UTILITY

Format Now Complete ! . . . . Remove Disk

Press RETURN to continue format or BREAK to end

If there is an error during the formatting procedure, line 2 of Screen # 3 displays:

Format NOT Complete . . . . Please Try Again

The Currently Logged Disk/Switching Disks

As described under "User Interaction through the CCP," the currently logged disk is the one identified on the screen via the prompt symbol **A >** for drive **A,** or **B >** for drive **B.**

If you have two disk drives and want to switch disks, you simply type the disk name you want (**A** or **B**), followed by a colon (:). This must be done when the CCP is in a receptive mode, prompting you for some type of input. The sequence of prompts and commands shown below might occur after the operating system is initially loaded from drive A.

PROMPTS/COMMANDS	EXPLANATION OF COMMANDS
xxK CP/M VER 2.2	
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.**

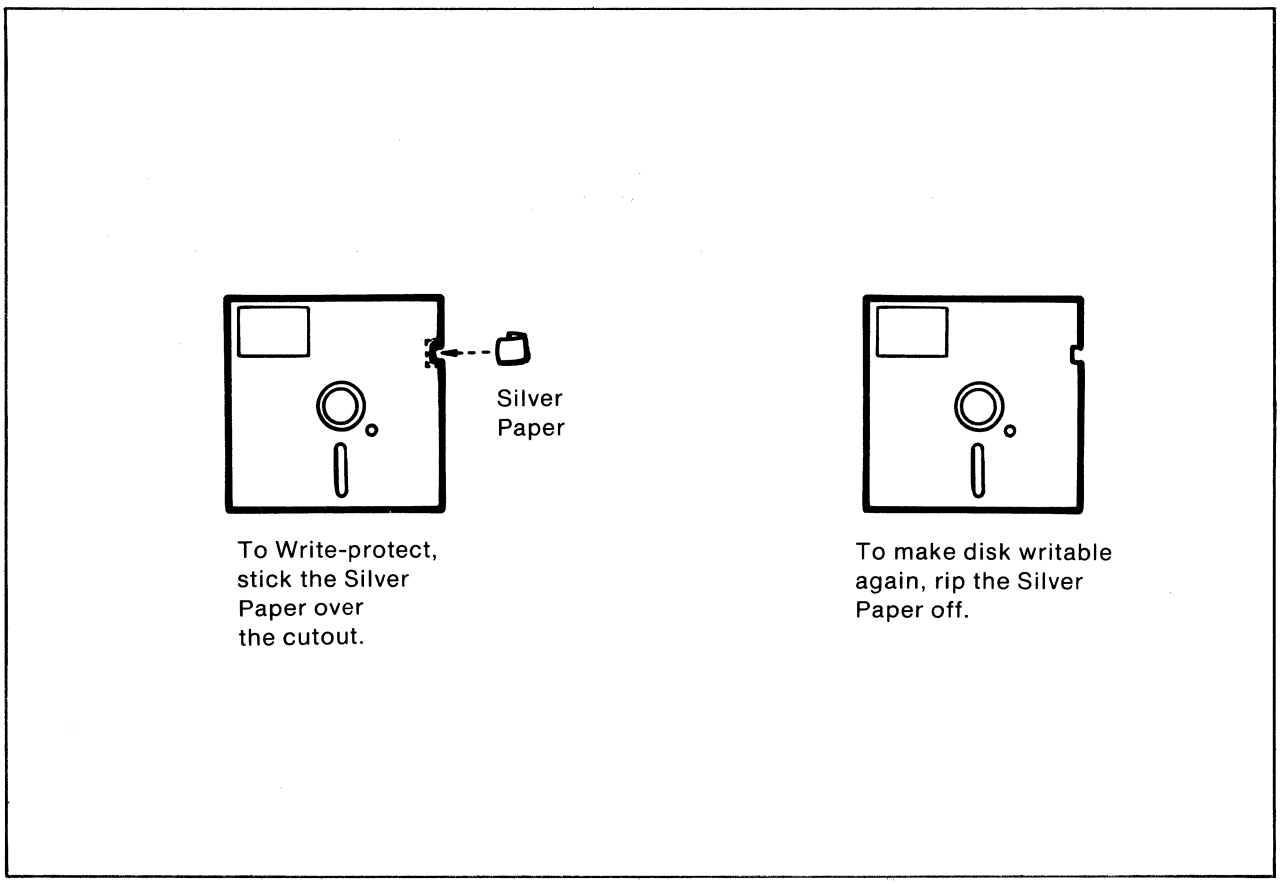


Figure 23 *Write-Protecting T200 Disks*

Write-Protecting Disks

Floppy disks are like cassettes in that you can record new information over prior information. The T200 and T250 offer different ways of protecting the contents stored on a disk from erroneously being covered over with new information.

T200; The 5.25-inch disks can be **write-protected** by sticking a protective silver paper on the disk jacket. (Shown in figure 23.)

T250: The way to protect disks on the T250 is to set the disk drive itself to a **read only** mode (rather than **read and write**). Use the STAT command in the following form:

STAT x:R/O

where **x** is drive **A** or **B.** Any disk mounted on the drive will not accept the write operation until the next warm or cold start.

NOTE: When you have disks that contain important information, it is recommended that you make backup disks by copying (PIP command or the disk copying operation).

If you try to write to a disk in a read only situation, you will receive an error message (see next section).

Error Messages

In the course of using your computer, you may encounter error messages displayed on your screen. The list below gives the messages possible, as well as steps for correcting the

ERROR MESSAGE	EXPLANATION	CORRECTION
NO FILE or **NOT FOUND**	The operating system cannot find a file.	• Check the directory for file names on both the **A** and **B** drives. The file may not exist on either drive directory. • Make sure you have the most current data disk. • Retype the file name in case you did not spell it correctly the first time.
FILE EXISTS	The file name already exists and may not be used again.	• Select a new name for the new file, or rename the file that already exists.
PIP?	The PIP program is not stored on currently logged disk.	1. Check the directory of the operating system for PIP.COM. 2. Load the disk that has the PIP.COM program.

ERROR MESSAGE	EXPLANATION	CORRECTION
BDOS ERR ON x (where **x** is drive **A** or **B**)		• Make sure the power to the disk drive is on. • Make sure the disk drive door is completely shut. • Make sure the disk is formatted and not too worn.
BDOS ERR ON x: BAD SECTOR	A problem exists in reading or writing to the disk.	• Make sure the disk has been mounted on the drive. • Check to see if the disk is worn out or damaged. • Verify that the disk is a type recommended by Toshiba (see **Appendix C**). • Recover from this situation by pressing the CTRL and C keys simultaneously to reboot the system.

NOTE: You may also press the RETURN key which ignores the bad sector. However, using the RETURN key in this situation may destroy your disk integrity if the operation is a directory write. In this case, make sure you have adequate backup. Check with your Toshiba representative if your system reports this error more than once a month.

BDOS ERR ON x: READ ONLY	An attempt has been made to write to a disk which has been set to **read only** in a STAT command, or when the drive has been set to read only by BDOS.	• Reboot the operating system by pressing the CTRL and C keys simultaneously, or perform a cold start when the disks are changed. (The drive is returned to read **and** write capacity.)
BDOS ERR ON x: SELECT	An attempt has been made to address a drive other than **A** or **B**.	• Reboot the system by pressing the RETURN or ENTRY key. • If necessary, restart the computer.

USING BASIC PROGRAMS

This section describes how to run programs written in the **BASIC** language. There are many versions of BASIC, and two are offered with the T200 and T250 systems. These are the CP/M version of Microsoft's BASIC-80 (**MBASIC**) and Digital Research Incorporated's **CBASIC**.

If you are not familiar with these languages, please refer to the MBASIC Reference Manual and the CBASIC Reference Manual that you have received. The rules of writing BASIC programs, the meaning of BASIC statements and commands, as well as sophisticated methods for modifying programs, are described in these manuals. You can then refer to the following section in order to learn the specifics of using the BASIC language in conjunction with your Toshiba T200 or T250.

CBASIC is a variation of a compiler language. This means that a program is written with an editor and then compiled. The programs used to compile (**CBAS2**) and to run (**CRUN2**) it are separate. After the source code of a program has been created, the CBASIC compiler generates intermediate code that the computer executes.

The intermediate code is derived from a CBASIC (or later version called **CBASIC 2**) compiled source program. It can be executed using the CRUN2 command. This command assumes that a program has already been compiled.

The version of MBASIC that is provided with your Toshiba T200 or T250 is an interpretive form of BASIC. It executes the source program statements directly by interpreting into object code. This avoids the step of compiling which in turn decreases the program development time.

Flow chart showing process of creating CBASIC program.

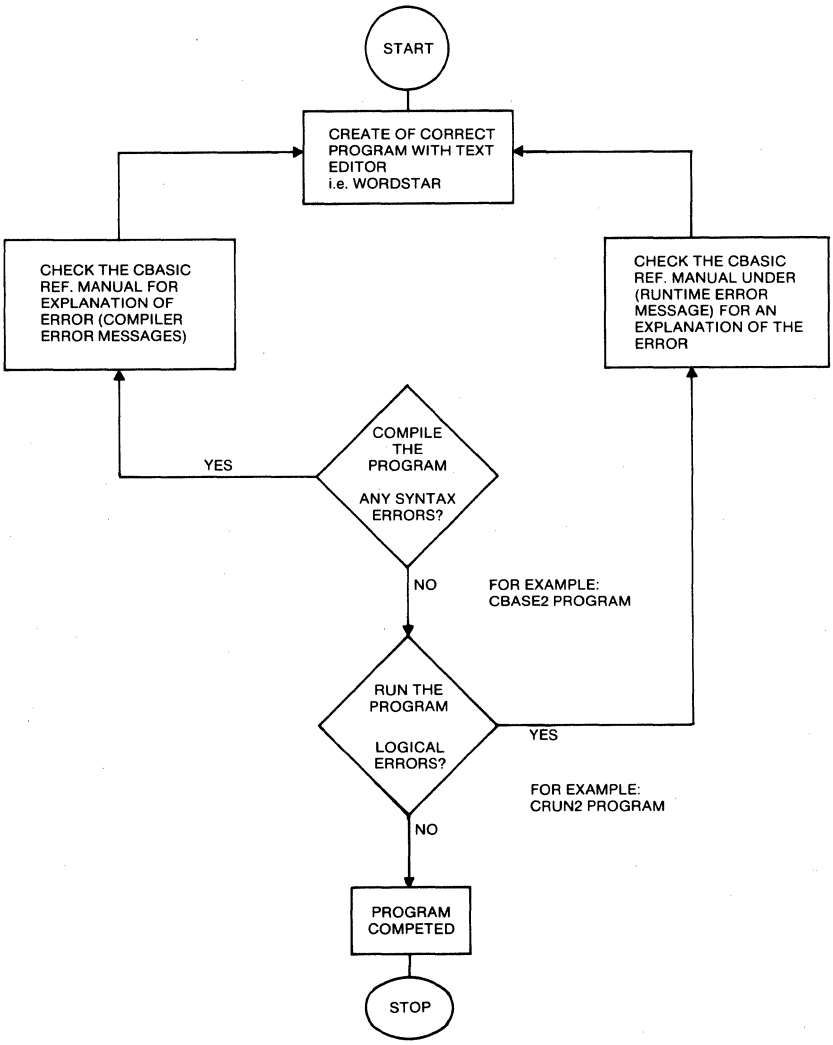


Figure 24 *Creating a CBASIC Program.*

Using CBASIC

To write a CBASIC program, first you create the source statements using a text editor or word processing program of your preference. Be sure to use a text editor such as **WordStar,** which has a mode or creating text which is a **nondocument** mode. Most text editors, in the normal word processing mode, will put in many characters and set certain bit flags which the CBASIC compiler cannot understand. Once you have created the source program (following the rules and statement explanations of the CBASIC reference manual) you compile the program. This is done by typing (while at the CP/M level):

A>CBAS2 Program name

Where "program name" is the name of the program you created.

NOTE: The program must have the secondary file name of **.BAS** or the compiler will not find it. Also, the primary filename can be no longer than 8 characters. Example: **PROCESS.BAS**. Make sure you follow these rules and give the program the same name when you create it with the text editor that you use to reference it from CP/M.

The compile will flag any errors it finds during compilation. These error messages can be found in the back of the CBASIC reference manual under "Compiler Error Messages." You must go back and correct any mistakes by using the text editor. Repeat this cycle until the program compiles successfully with no errors found. Now you are ready to try running the program. To do this type:

A>CRUN2 Program name

Any errors found during execution will be flagged by the runtime module (CRUN2.COM) and you will be taken out to CP/M level. Go to the back of the CBASIC reference manual to find an explanation of the error message under "Runtime error messages." Then go back to the text editor, make the necessary changes to the program, recompile it, and try running it again until it runs without errors.

The process involved in writing a CBASIC program is illustrated by the flow chart on the left.

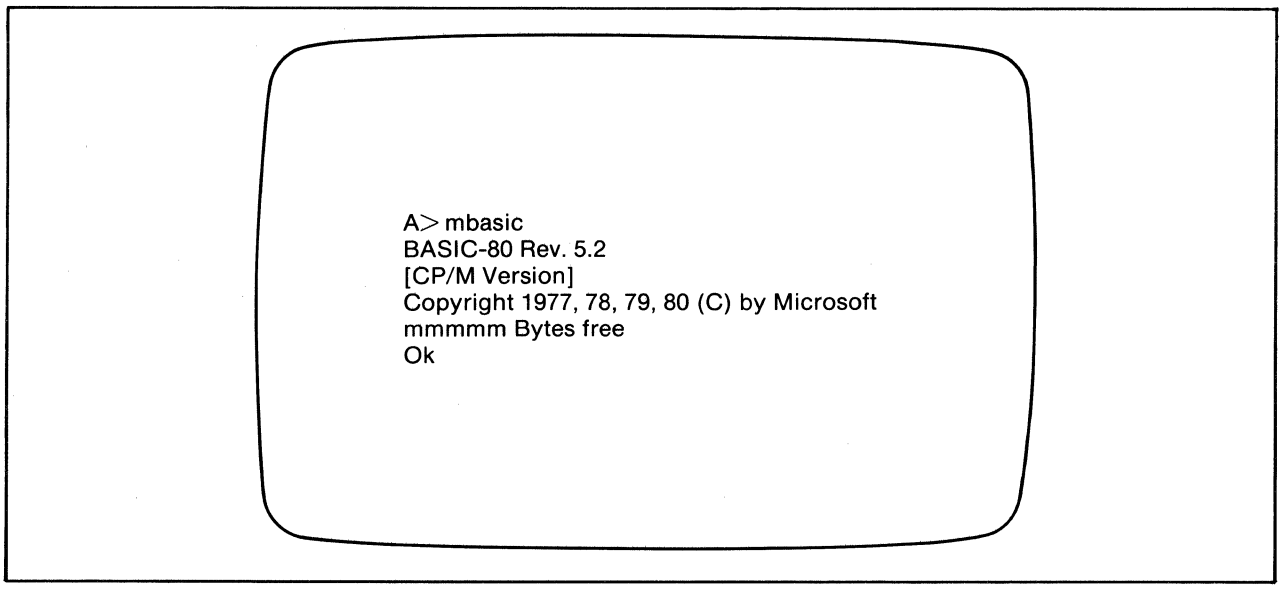


Figure 25 *Initiating MBASIC*

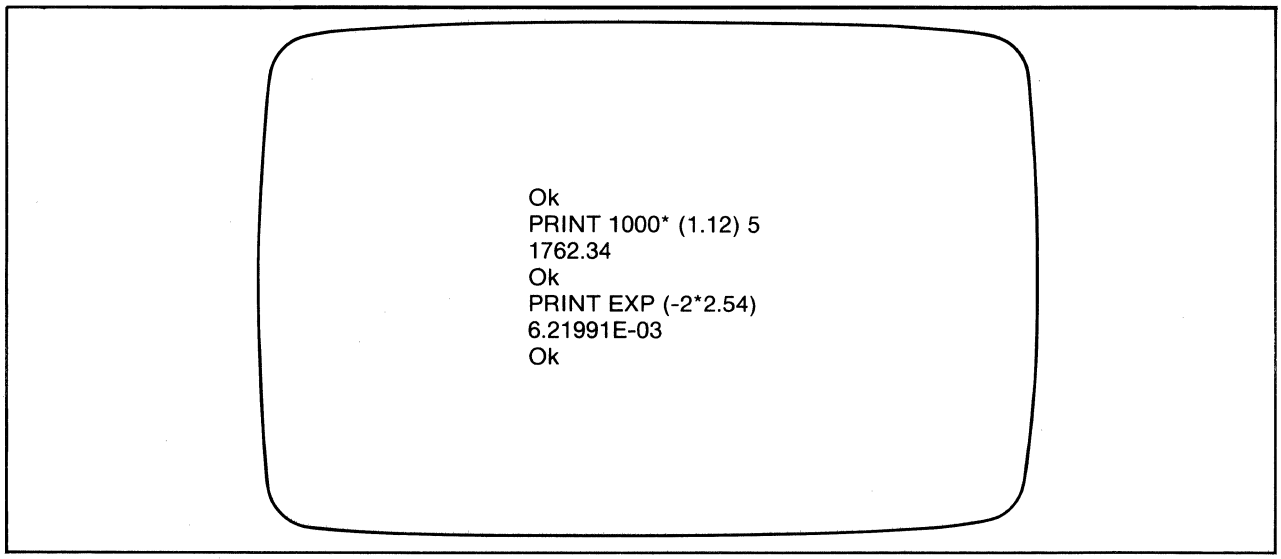


Figure 26 *BASIC Direct Mode*

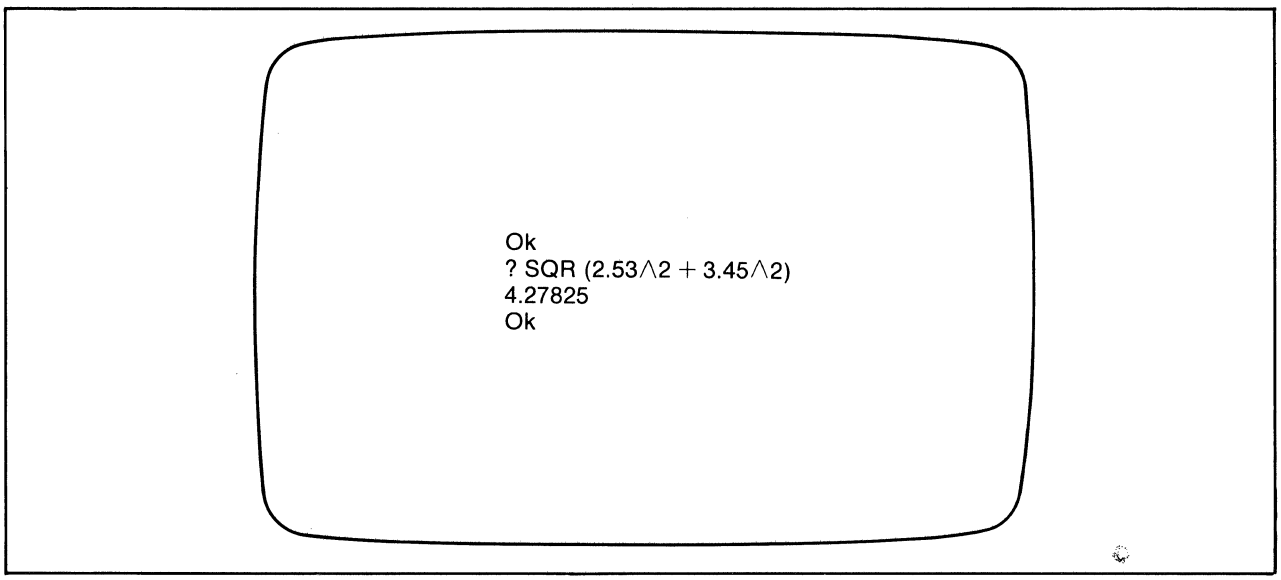


Figure 27 *"?" for PRINT*

Initiating MBASIC

After you load the operating system into your computer, you receive the command prompt **A>** on the screen. You have learned that at this point your system is waiting for a command from you. If you want to work with MBASIC, be sure the system disk is in drive A and enter the name of the file which contains the BASIC program. This file, called **MBASIC**, is on the system disk. As with any command, you then press the carriage return key to send the command to the operating system. (See the figure at top left.)

About eight second after you send MBASIC, the system will reply as shown in the top figure at left. The prompt **OK** is displayed when BASIC is ready to accept your commands to BASIC. At this point, you may remove the system disk from the drive since MBASIC has been loaded into memory until the next warm or cold start.

BASIC Commands

BASIC accepts predefined commands. If you enter anything other than the established commands, and press the carriage return key, you will see the error message **Syntax error** on the screen. This means that BASIC cannot understand what you are trying to request. (Error messages and their causes are explained in the MBASIC and CBASIC Reference Manuals.) Therefore, in order to work with BASIC, you must use the established terminology. You can enter BASIC commands and statements in either of two modes: **direct** or **indirect.**

The Direct Mode

In the direct mode, BASIC commands and statements are entered directly without line numbers. These commands and statements begin with a key word such as **PRINT**. An input line is submitted to BASIC when the carriage return key is pressed and the commands and statements are executed. As shown in the middle figure at left, the direct mode may be used as a "calculator" for quick computations that do not require a complete program. A BASIC statement **PRINT** precedes the arithmetic operation you want to calculate

As shown in the bottom figure at left, a shorthand method is available (as the exception) for the use of PRINT. A question mark (**?**) can be used in place of **PRINT**.

The Indirect Mode

The indirect mode is used for entering programs. Program lines are preceded by line numbers and are stored in memory. The upcoming section, "Entering a BASIC Program," describes this process.

```
Ok
new
Ok
auto
10 rem Net Present Value Calculation
20 input "Enter revenue for each year"; R(1), R(2), R(3), R(4), R(5)
30 input "Enter cost for each year"; C(1), C(2), C(3), C(4), C(5)
40 input "Interest rate"; D
50 for N=1 to 4
60 NPV=NPV + (R(N) - E(N)/(1 + D) ∧ N
70 print "Net Present Value . ."; NPV
80 goto 20
90 end
100 ∧ C (BREAK key is pressed.)
Ok
list
10 REM Net Present Value Calculation
20 INPUT "Enter revenue for each year"; R(1), R(2), R(3), R(4), R(5)
30 INPUT "Enter cost for each year"; C(1), C(2), C(3), C(4), C(5)
40 INPUT "Interest rate"; D
50 for N=1 to 4
60 NPV=NPV + (R(N) - E(N)/(1 + D) ∧ N
70 PRINT "Net Present Value . . "; NPV
80 GOTO 20
90 END
OK
```

Figure 28 *A BASIC Program*

Entering a BASIC Program

A BASIC program consists of program lines composed in the following manner:

line number BASIC statement

A BASIC program line always begins with a line number and ends with a carriage return. The line numbers may be typed in, or for additional convenience, automatically generated by the **AUTO** command.

The figure at top left shows a sample BASIC program. Observe that two commands are given before the program lines begin. The first command, **NEW**, tells the system that you are entering a new program. This command clears all lines stored in memory. The second command, **AUTO**, is used (as descibed above) to generate line numbers automatically. To terminate automatic line numbering, hold the CTRL key and touch the C key, or simply touch the (BREAK) key. The line in which CTRL and C are typed is not submitted and BASIC returns to command level.

After the program was entered, the command **LIST** was typed in order to display the program just stored in memory.

NOTE: Command and statement keywords, such as **REM**, **INPUT** and **PRINT**, were typed in the example at top left in lowercase. They were automatically converted to uppercase, as shown in the **LIST**ed program on the next page.

```
Ok
50 for N=1 to 5
15 dim R(5), C(5)
65 next N
45 NPV=0
80
61 print R(N)−C(N), NPV
list
10 REM Net Present Value Calculation
15 DIM R(5), C(5)
20 INPUT "Enter revenue for each year"; R(1), R(2), R(3), R(4), R(5)
30 INPUT "Enter cost for each year"; C(1), C(2), C(3), C(4), C(5)
40 INPUT "Interest rate"; D
45 NPV=0
50 FOR N=1 TO 5
60 NPV=NPV + (R(N) − C(N))/(1 + D) ∧ N
61 PRINT R(N)−C(N), NPV
65 NEXT N
70 PRINT "Net Present Value . ."; NPV
90 END
Ok
list
Ok
```

Figure 29 *Program Corrections/LIST*

```
Ok
RUN
Enter revenue for each year? 200, 340, 800, 450, 400
Enter cost for each year? 1010, 150, 20, 20, 20
Interest rate? 0.25
−810                    −648
 190                    −526.4
 780                    −127.04
 430                      49.0881
 380                     173.607
Net Present Value . .173.607
```

Figure 30 *RUN Command*

Correcting A BASIC Program

- If an incorrect character is entered as a line is typed, use the left arrow key to move the cursor to the incorrect character position and continue typing the line as desired.
- If a program line that is currently in memory needs correction, retype the line using the same line number, as shown below:

```
Ok
50 for N=1 to 5
```

BASIC automatically replaces the old line with the new line.

- If lines need to be inserted to a program in memory, simply type the lines with the appropriate numbers, as shown below (line numbers indicate the order in which the program lines are stored in memory):

```
15 dim R(5), C(5)
65 next N
45 NPV=0
61 print R(N)-C(N), NPV
```

- If lines in memory need to be deleted, type the line number and press the carriage return:

```
80
```

- If you have made a correction, confirm it by using the **LIST** command which prints out the program currently in memory.

The program at top left shows some corrections and then the **LIST**ed program. The **LLIST** command prints the program on the printer.

Running a BASIC Program

To execute the program in memory, type the **RUN** command. The program is executed, and the results are given. In the example at lower left, the program requested yearly revenue and cost as well as interest rate. The program then figured and printed out the net present value.

Storing A Program on Disk

The program entered from the keyboard is not permanently recorded in memory. The program may be erased and written over by other programs, or deleted when you turn the

```
10 REM Purchase Journal
20 OPEN "O", #1, "B:PRCHS.JNL"
30 INPUT "Voucher No."; N
40 IF N = 9999 THEN 200
50 INPUT "Name of Supplier"; S$
60 INPUT "Date"; M
70 INPUT "Amount"; CREDIT
80 INPUT "Debit: No. and Amount"; J,DEBIT
90 IF J=120 OR J=511 OR  J=521 OR  J=531 OR J=170 THEN 160 ELSE 100
100 PRINT "120 . . .Merchandise Inventory"
110 PRINT "511 . . .Head Office Expense"
120 PRINT "521 . . .South Branch Expense"
130 PRINT "531 . . .North Branch Expense"
140 PRINT "170 . . .Office Equipment"
150 GOTO 80
160 IF CREDIT <> DEBIT THEN 170 ELSE 180
170 PRINT "error"
175 GOTO 70
180 PRINT#1,M;",";N;",";S$;",";CREDIT;",";J
190 GOTO 30
200 CLOSE
210 END
```

Figure 31 *A File-handling Program*

system's power off. Before this occurs, the program should be written from the memory to a disk by the **SAVE** command. In the following example, the **SAVE** command saves the file named NPV:

```
Ok
SAVE "B:NPV"
Ok
```

The "B:" indicates that the program is to be stored on a disk which is inserted in drive B (#2). If you have removed the system disk and mounted your disk on the drive A (#1), type:

```
SAVE "A:file name"
```

Or, if your disk is mounted on the currently active disk drive, you do not need to reference a drive and can simply type:

```
SAVE "file name"
```

The saved program can be recalled at any time from the disk and loaded into memory for program changes or execution via the LOAD command:

```
Ok
LOAD "B:NPV"
Ok
```

In dealing with programs, it is important that you understand two types of files:

- Program files
- Data files

Program files contain the saved programs. **Data files** contain data which are read or written by the executing programs.

For example, a data file used for an inventory control application may contain relatively fixed information such as merchandise name, price and vendor name, as well as inventory status information.

Data files are created with specified names, loaded with contents, modified or appended by the BASIC application programs. These include **OPEN**, **PRINT#**, **INPUT#**, **WRITE#**, **CLOSE**, **GET**, **PUT** and so on.

The figure at left shows an example of a file-handling program.

Activating Saved BASIC Programs

You will want to activate programs that you write and/or purchase. These programs are stored on floppy disks with unique file names. To activate the execution of a program, insert the appropriate disk into drive **B** (**#2**) and enter any of the following command sequences:

```
A>MBASIC  B:file name
```

```
Ok
LOAD  "B:file name",R
```

```
Ok
RUN  "B:file name"
```

```
Ok
LOAD  "B:file name"
RUN
```

If you want to execute two or more programs in a specified sequence, you can write the commands to initiate these programs in order in a "command file." You then type a single SUBMIT command which causes the program to be executed automatically in a batch mode:

```
A>SUBMIT  B:command-file name
```

Each program should contain a **SYSTEM** statement to return to the operating system when it is finished, as described below.

Terminating BASIC

To exit from BASIC and return to the operating system, type **SYSTEM**. You then receive the prompt **A>**.

If you have removed the system disk, mount it on drive **A** before you type SYSTEM or the message, **SET DISK IN DRIVE A OR DISK ERRORS** appears on the screen.

NOTE: To interrupt program execution and return to BASIC command level, hold the CTRL key and press the C key, or simply press the (BREAK) key.

USING ASSEMBLER PROGRAMS

The CP/M assembler, system entry points, debugger and text editor are used by advanced users who have a special need for working with them. The assembler reads assembly language source files from the disk and produces 8080 machine language in Intel hex format. The CP/M assembler is initiated by typing:

ASM filename or

ASM filename.parms

In both cases, the assembler assumes there is a file on the disk with the name

filename.ASM

which contains an 8080 assembly language source file. The first and second forms shown above differ only in that the second form allows parameters to be passed to the assembler to control source file access and hex and print file destinations.

In either case, the CP/M assembler loads, and prints the message

CP/M ASSEMBLER VER n.n

where n.n is the current version number. In the case of the first command, the assembler reads the source file with assumed file type **ASM** and creates two output files:

filename.HEX and

filename.PRN

The **HEX** file contains the machine code corresponding to the original program in Intel hex format, and the **PRN** file contains an annotated listing showing generated machine code, error flags and source lines. If errors occur during translation, they will be listed in the **PRN** file as well as at the console.

The second command form can be used to redirect input and output files from their defaults. In this case, the "parms" portion of the command is a three-letter group which specifies the origin of the source file, the destination of the hex file and the destination of the print file. The form is

filename.plp2p3

where **pl**, **p2**, and **p3** are single letters

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

PART 2

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 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.

Program Format

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

line# label operation operand ;comment

where any or all of the fields may be present in a particular instance. Each assembly 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 treated as an end-of-the line by the assembler. (Thus, multiple assembly language statements can be written on the same physical line if separated by exclamation point 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. 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 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$name
x:	yxl:	longer$named$data:
x1y2	x1x2	x234$5678$9012$3456:

The operation field contains either an assembly directive, or pseudo operation, or an 8080 machine operation code. The pseudo operations and machine operation code are described below.

The operand field of the statement, in general, contains an expression formed out of constants and labels, along with arithmetic and logical operations on these elements. Again, the complete details of properly formed expressions are given below.

The comment field contains arbitrary characters following the ; symbol until the next real or logical end-of-line. These characters are read, listed, and otherwise ignored by the assembler. In order to maintain compatibility with the Processor Technology assembler, the CP/M assembler also treats 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.

Forming the Operand

In order to describe completely the operation codes and pseudo operations, it is necessary to present first 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.

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 reserve memory space (for example, a **MOV** instruction or a DS pseudo operation), 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.

Numeric Constants

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

B	binary constant (base 2)
O	octal constant (base 8)
Q	octal constant (base 8)
D	decimal constant (base 10)
H	hexadecimal constant (base 16)

Q is an alternate radix indicator for octal numbers since the letter **O** is easily confused with the digit **0**. Any numeric constant which does not terminate with a radix indicator is assumed to be a decimal constant.

A constant is thus composed as a sequence of digits, followed by an optional radix indicator, where the digits are in the appropriate range for the radix. That is, binary constants must be composed of 0 and 1 digits, octal constants can contain digits in the range 0 - 7, while decimal constants contain decimal digits. Hexadecimal constants contain decimal digits as well as hexadecimal digits A (10D), B (11D), C (12D), D (13D), E (14D) and F (15D). Note that the leading digit of a hexadecimal constant must be a decimal digit in order to avoid confusing a hexadecimal constant with an identifier (a leading 0 will always suffice). A constant composed in this manner must equate 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 **$** signs are allowed within constants to improve their readability. Finally, the radix indicator is translated to upper case if a lower case letter is encountered. The following are all valid instances of numeric constants:

1234	1234D	1100B	1111$0000$1111$0000B
1234H	0FFEH	33770	33$77$22Q
3377o	0fe3h	1234d	0ffffh

Reserved Words

There are several reserved character sequences which have predefined meanings in the operand field of a statement. The names of 8080 registers are given below, which, when encountered, produce the value shown to the right:

A	7
B	0
C	1
D	2
E	3
H	4
L	5
M	6
SP	6
PSW	6

(Again, lower case names have the same value as their upper case equivalents). Machine instructions can also be used in the operand field, and equate to their internal codes. In the case of instructions which require operands, where specific operand becomes a part of the binary bit pattern of the instruction (for example, MOV A,B), the value of the instruction (in this case MOV) is the bit pattern of the instruction with zeroes in the operand fields (for example, MOV produces 40H).

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

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; 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.'

Arithmetic and Logical Operators

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

OPERATOR	EXPLANATION
a + b	Unsigned arithmetic sum of a and b
a − b	Unsigned arithmetic difference between a and b
+ b	Unary plus (produces b)
− b	Unary minus (identical to 0 − b)
a * b	Unsigned magnitude multiplication of a and b
a / b	Unsigned magnitude division of a by b
a MOD b	Remainder after a / b
NOT b	Logical inverse of b (all 0's become 1's; 1's become 0's), where b is considered a 16-bit value

a AND b	Bit-by-bit logical *and* of a and b
a OR b	Bit-by-bit logical *or* of a and b
a XOR b	Bit-by-bit logical exclusive *or* of a and b
a SHL b	The value which results from shifting a to the left by an amount b, with zero fill.
a SHR b	The value which results from shifting a to the right by an amount b, with zero fill.

In each case, **a** and **b** represent simple operands (labels, numeric constants, reserved words and one or two character strings), or fully enclosed parenthesized subexpressions such as:

```
10+20          10h+37Q      L1/3      (L2+4) SHR 3
('a' and 5fh) + '0'                 ('B'+B) OR (PSW+M)
(1+(2+c)) shr (A-(B+1))
```

Note that all computations are performed at assembly time as 16-bit unsigned operations. Thus, **−1** is computed as **0−1** which results in the value **0ffffh** (that is, all 1's). The resulting expression must fit the operation code in which it is used. If, for example, the expression is used in an ADI (add immediate) instruction, then the high order eight bits of the expression must be zero. As a result, the operation "ADI −1" produces an error message (**−1** becomes **0ffffh** which cannot be represented as an 8-bit value), while **ADI (−1) AND 0FFH** is accepted by the assembler since the **AND** operation zeroes the high order bits of the expression.

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 unparenthesized expressions is listed below. Operators listed first have highest precedence (they are applied first in an unparentesized 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
- +
NOT
AND
OR XOR
```

Thus, the expressions shown to the left below are interpreted by the assembler as the fully parenthesized 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. 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.

Assembler Directives

Assembler directives are used to set labels to specific values during assembly, perform conditional assembly, define storage areas and specify starting addresses in the program. Each assembler directive is denoted by a "pseudo operation" which appears in the operation field of the line. The acceptable pseudo operations are:

OPERATION	EXPLANATION
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.

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.)

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.)

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 communication 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.

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.

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 a part of a program which communicates with either a teletype or a CRT console (but not both) by selecting a particular value for TTY before the assembly begins:

```
TRUE      EQU     0FFFFH      ;DEFINE VALUE OF TRUE
FALSE     EQU     NOT TRUE    ;DEFINE VALUE OF FALSE
;
TTY       EQU     TRUE        ;TRUE IF TTY, FALSE IF CRT
;
TTYBASE   EQU     10H         ;BASE OF TTY I/O PORTS
CRTBASE   EQU     20H         ;BASE OF CRT I/O PORTS
          IF      TTY         ;ASSEMBLE RELATIVE TO TTYBASE
CONIN     EQU     TTYBASE     ;CONSOLE INPUT
CONOUT    EQU     TTYBASE+1   ;CONSOLE OUTPUT
```

```
            ENDIF
;
            IF      NOT TTY     ;ASSEMBLE RELATIVE TO CRTBASE
CONIN       EQU     CRTBASE     ;CONSOLE INPUT
CONOUT      EQU     CRTBASE+1   ;CONSOLE OUTPUT
            ENDIF
            ...
            IN      CONIN       ;READ CONSOLE DATA
            ...
            OUT     CONOUT      ;WRITE CONSOLE DATA
```

In this case, the program would assemble for an environment where a teletype is connected, based at port 10H. The statement defining TTY could be changed to:

```
TTY     EQU     FALSE
```

and, in this case, the program would assemble for a CRT based at port 20H.

The DB Directive

The **DB** directive allows the programmer to define initialized 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 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. (They must stand alone between the commas). Note that ACSII characters are always placed in memory with the parity bit reset (0). Further, recall that there is no translation from lower to upper case within strings. The optional label can be used to reference the data area throughout the remainder of the program. Examples of valid **DB statements** are:

```
data:   DB   0,1,2,3,4,5
        DB   data and 0fffh,5,377Q,1+2+3+4
signon: DB   'please type your name',cr,1f,0
        DB   'AB' SHR 8, 'C', 'DE' AND 7FH
```

The DW Directives

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 are

not allowed. In all cases, the data storage is consistent with the 8080 processor: the least significant byte of the expression is stored first in memory, followed by the most significant byte. Examples are:

```
doub:  DW   Offefh,doub+4,signon-$,255+255
       DW   'a', 5, 'ab', 'CD', 6 shl 8 or 11b
```

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
```

Operation Codes

Assembly language operation codes form the principal part of assembly language programs, and form the operation field of the instruction. In general, ASM accepts all the standard mnemonics for the Intel 8080 microcomputer, which are given in detail in the Intel manual "8080 Assembly Language Programming Manual." Labels are optional on each input line and, if included, take the value of the instruction address immediately before the instruction is issued. The individual operators are listed briefly in the following sections for completeness. The Intel manuals should be referenced for exact operator details. In each case:

e3	Represents a 3-bit value in the range 0-7 which can be one of the predefined registers A,B,C,D,E,H,L,M,SP or PSW
e8	Represents an 8-bit value in the range 0-255
e16	Represents a 16-bit value in the range 0-65535

which can themselves be formed from an arbitrary combination of operands and operators. In some cases, the operands are restricted to particular values within the allowable range, such as the PUSH instruction. These cases will be noted as they are encountered.

In the sections which follow, each operator code is listed in its most general form, along with a specific example with a short explanation and special restrictions.

Jumps, Calls and Returns

The **Jump, Call and Return** instructions allow several different forms which test the condition flags set in the 8080 microcomputer CPU. The forms are:

JMP	e16	JMP	L1	Jump unconditionally to label
JNZ	e16	JMP	L2	Jump on non-zero condition to label
JZ	e16	JMP	100H	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	a1	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	100H	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	0	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

Immediate Operand Instructions

Several instructions are available which load single or double precision registers, or single precision memory cells with constant values. Instructions which perform immediate arithmetic or logical operations on the accumulator (register A) are also available.

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 0FFH	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$0000B	"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,100H	Load extended immediate to register pair (e3 must be equivalent to B, D, H or SP).

Increment and Decrement Instructions

Instructions are provided in the 8080 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).

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 0	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.

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 (3e 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).

Control Instructions

The four remaining instructions are categorized as control:

HLT	Halt the 8080 processor
DI	Disable the interrupt system
EI	Enable the interrupt system
NOP	No operation

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:

ERROR CODE	EXPLANATION
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 (for example, macros) are recognized, but flagged in this version.
O	**Overflow:** expression is too complicated (too many pending operators) to compute. 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 messages are printed which are due to terminal error conditions:

ERROR/MESSAGE	EXPLANATION
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 (for example, 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.

A Sample Session

The following section shows interaction with the assembler and debugger in the development of a simple language program.

```
ASM SORT↵  assemble SORT.ASM


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

generated machine code

truncated

```
0100                ;       SORT PROGRAM IN CP/M ASSEMBLY LANGUAGE
                    ;       START AT THE BEGINNING OF THE TRANSIENT PROGRAM AR
0100                .       ORG     100H
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 RE8
                    ;
                    ;       CONTINUE THIS PASS
                    ;       ADDRESSING I, SO LOAD AV(I) INTO REGISTERS
0119    5F16002148  CONT:   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                  IHX     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 INCL ;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
:07014000470134C30A01006E
:1001480050064001E00320014000700E8032C01BB
:040158006400018 0BE
:0000000000
A>DDT SORT.HEX↵
16K DDT VER 1.0
NEXT PC
015C 0000   default address (no address on END statement)
-XP↵
P=0000 100↵  Change PC to 100                                   about with rubout
-UFFFF↵  untrace for 65535 steps                                      ↓
C0Z0M0E0I0  A=00  B=0000  D=0000  H=0000  S=0100  P=0100  LXI H,0146*0100
-T10↵  trace 10₁₆ 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 108H
0110↵
```

```
-XP↵
P=010B 100↵   reset program counter back to beginning of program
-T10↵   trace execution for 10H steps
```

Altered instruction

```
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
```

Automatic breakpoint

```
-L100↵
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
```

list some code from 100H

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

list more

above list with rubout

```
-G, 118↵   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
```

data is sorted, but program doesn't stop

```
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  00  ..................
```

```
-G0↵  return to CP/M
DDT SORT. HEX↵  reload the memory image


P=0000  100↵  set PC to beginning of program
-L10D↵  list bad op code
010D  JNC  0119
0110  LXI  H,0146
   abort list with rubout
-A10D↵  assemble new op code
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 00
0103  MVI  M,0↵
0105↵
-^C  return to CP/M with ctl-c (G0 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 alway starts with address 100H
-G↵  run the program from PC=100H
*0118  programmed stop (RST7) 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  ................

-G0↵  return to CP/M
```

PART 2

```
ED SORT .ASM↵ make changes to original program
Ctl −Z
     ^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
skip prn file

```
ASM  SORT. AAZ↵

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 ....................
```

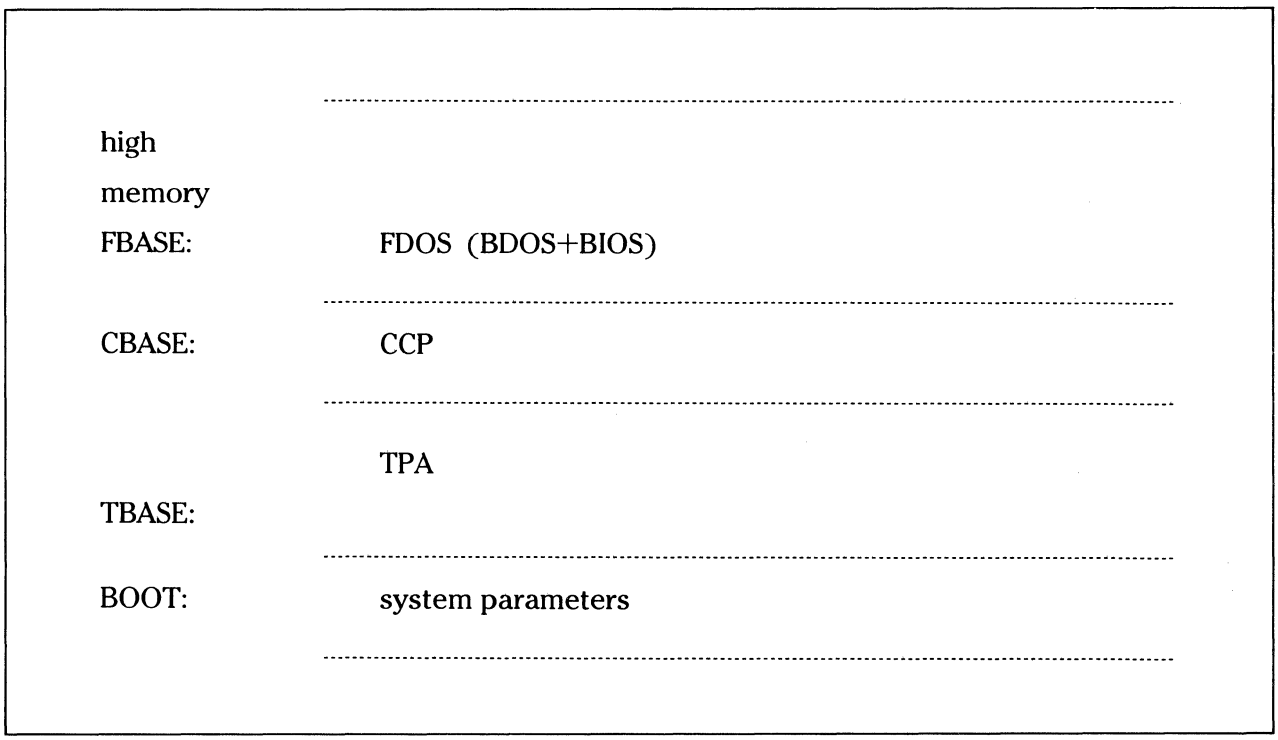
abort with rubout

```
−G0↵  return to CP/M - program check OK
```

System Entry Points

This section describes CP/M system organization, including the structure of memory and sytem entry points.

The BIOS and BDOS are logically combined into a single module with a common entry point, 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, that is, 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 in later sections. Memory organization of the CP/M system is shown below:



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 program 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+0100H which is normally location 0100H. The principal entry point to the FDOS is at location BOOT+005H (normally 0005H) where a jump to FBASE is found. The address field at BOOT+0006h (normally 0006H) 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 file1

command file1 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. Therefore, it 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. Therefore, it 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. In the latter case, memory up through FBASE-1 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+0005H. 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 below.

Operating System Call Conventions

The purpose of this section is to provide detailed information for performing direct operating system calls from user programs. 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 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 No.
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 (most transients return to the CCP through a jump to location 000H), 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**	L/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 (0DH followed by 0AH). Thus, one 128-byte CP/M record could contain several lines of source text. The end of an ASCII file is denoted by a CTRL-Z character (1AH) or a real end-of-file, returned by the CP/M read operation. CTRL-Z characters embedded within machine code files (for example, 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 65,536 records of 128 bytes each, numbered from 0 through 65,535, 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 operation 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+005CH (normally 005CH) 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+0080H (normally 0080H) 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, with the exception of Search First and Search Next, where compatibilty 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 the three bytes starting at BOOT+007DH 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	d0	/ /	dn	cr	r0	r1	r2
00	01	02	...	08	09	10	11	12	13	14	15	16	...	31	32	33	34	35

where

dr	drive code (0 - 16) 0 = 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 = 0
t1, t2, t3	contain the file type in ASCII upper case, with the high bit = 0 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 at 00 by the user, but in range 0 - 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 0 - 128
d0 . . . 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 the user
r0, r1, r2	optional random record number in the range 0-65535, with overflow to r2, r0, r1 constitute a 16-bit value with low byte r0, 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 16 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 is later recorded permanently on disk at the termination of the file operation (see the CLOSE command).

The CCP constructs the first 16 bytes of two optional FCB's for a transient by scanning the remainder of the line following the transient name, denoted by **file1** and **file 2** 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+006DH, 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 on the following pages, completed by a summary list of the functions.

FUNCTION 0: SYSTEM RESET

Entry Parameters:

Register **C: 00H**

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: 01H**

Returned Value:

Register **A: ASCII Character**

The **Console Input** function reads the next console character to register A. Characters, along with carriage return, line feed and backspace (CTRL-H) are echoed to the console. Tab characters (CTRL-I) are expanded in columns of eight characters. A check is made for start/stop scroll (CTRL-S) and start/stop printer echo (CTRL-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: 02H**

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: 03H**

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: 04H**

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: 05H**

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: 06H**

Register **E: 0FFH (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 (for example, CTRL-S and CTRL-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 supported under future releases of 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 7: GET I/O BYTE

Entry Parameters:

Register **C: 007H**

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."

FUNCTION 8: SET I/O BYTE

Entry Parameters:

Register **C: 08H**

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: 09H**

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 10: READ CONSOLE BUFFER

Entry Parameters:

Register **C: 0AH**

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 the input buffer overflows. The Read Buffer takes the form:

DE:	+0	+1	+2	+3	+4	+5	+6	+7	+8	...	+n
	mx	nc	c1	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. The nc < mx, then uninitialized positions follow the last character, denoted by **??** in the above figure. Various control functions are recognized during line editing (DEL, CRTL-C, E, H, J, M, R, U and X). These are described in the line editing section. Note also that certain functions which return the carriage to the leftmost position do so only to the column position where the prompt ended.

FUNCTION 11: GET CONSOLE STATUS

Entry Parameters:

Register **C: 0BH**

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 0FFH is returned in register A. Otherwise, a 00H value is returned.

FUNCTION 12: RETURN VERSION NUMBER

Entry Parameters:

Register **C: 0CH**

Returned Value:

Registers **HL: Version Number**

Function 12 provides information which allows **version independent programming**. A two-byte value is returned, with H=00 designating the CP/M release. CP/M returns a hexadecimal in register L, in the range of 21, 22 through 2F. Using function 12, for example, you can write application programs which provide both sequential and random access functions.

FUNCTION 13: RESET DISK SYSTEMS

Entry Parameters:

Register **C: 0DH**

The **Reset Disk** function is used to restore programmatically 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=0080H. 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: 0EH**

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=0 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=00H) 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: 0FH**

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 sl 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 **d0** 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 successful open operation is completed. Upon return, the open function returns a "directory code" with the value 0 through 3 if the open was successful, or 0FFH (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: 10H**

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 0, 1, 2 or 3, while a 0FFH (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 record permanently 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 0, 1, 2 or 3 is returned indicating the file is present. If 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 (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 file name and type may contain ambiguous references (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 cannot be found, otherwise a value in the range 0 to 3 is returned.

FUNCTION 20: 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 00H is returned in register A if the read operation was successful. A non-zero vaue is returned in register A if the read operation was successful. A non-zero value is returned if no data exists at the next record position (for example, an 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=00H 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 (that is, 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=0, 1, 2 or 3 if the operation was successful and 0FFH (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. Byte s2 is zeroed upon entry to the BDOS.

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 0 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 0 and 3 if the rename was successful, and 0FFH (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 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, labeled P. A **0** bit indicates that the drive is not on-line. A **1** bit marks a 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. 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 0 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 (the data is transferred 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+0080H. 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 **Write Protect Disk** 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 x: R/O

where **x** is the disk drive.

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 30: 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 (+**1'** and +**2'**) can be set or reset. The DE pair addresses an unambiguous file name with the appropriate attributes set or reset. Function 30 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 program can dynamically change the value 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: 20H**

Register **E: 0FFH (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=0FFH, then the value of the current user number is returned in register A, where the value is in the range 0 to 31. If the register E is not 0FFH, 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 r0 at 33, rl at 34 and r2 at 35). Note that the sequence of 24 bits is stored with the least significant byte first (r0), 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 r0, r1 byte pair is treated as a double-byte, or **word** value, which contains the record to read. This value ranges from 0 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 0) 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 (r0, 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 00 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 subsequent 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 codes 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. Normally, non-zero return codes can be treated as missing data, with zero return codes indicating that the operation is 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 exception of error code 05, 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 r0, 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 01, then the file contains the maximum record count 65536. Otherwise, bytes r0 and r1 constitute a 16-bit value (r0 is the least significant byte, as before) which is the file size.

Data can be appended to the end of an existing file simply by 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 8-megabyte file is written in random mode (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 produce automatically the random record position from a file which has been read or written sequentially to a particular point. The function can be used in two ways.

First, it is often necessary initially to 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.

System Function Summary

#	FUNCTION NAME	INPUT PARAMETERS	OUTPUT RESULTS
0	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
10	Read Console Buffer	DE = .Buffer	see def

#	FUNCTION NAME	INPUT PARAMETERS	OUTPUT RESULTS
11	Get Console Status	none	A = 00/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	E = .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
20	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 Addr(=R/O Vector)	none	HL = R/O Vect**
30	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	r0, r1, r2
36	Set Random Record	DE = .FCB	r0, r1, r2

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

A Sample File-to-File Copy Program

The following program provides a relatively simple example of file operations. The program source file is created as **COPY.ASM,** using the CP/M **ED** program. It is then assembled using **ASM** or **MAC**, resulting in a **HEX** file. The **LOAD** program is then 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. This means that 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 existing 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 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
0105  116c00    lxi     d,fcb2  ; source of move
0108  21da01    lxi     h,dfcb  ; destination fcb
010b 1a         mfcb:   ldax    d       ; source fcb
010c 13         inx     d       ; ready next
010d 77         mov     m,a     ; dest fcb
010e 23         inx     h       ; ready next
010f 0d         dcr     c       ; count 16. . . 0
0110 c20b01     jnz     mfcb    ; loop 16 times
                ;
                ;       name has been moved, zero cr
0113 af         xra     a       ; a =00h
0114 32fa01     sta     dfcbcr  ; current rec = 0
                ;
                ;       source and destination fcb's ready
                ;
0117 115c00     lxi     d,sfcb  ; source file
011a cd6901     call    open    ; error if 255
011d 118701     lxi     d,nofile ; ready message
0120 3c         inr     a       ; 255 becomes 0
0121 cc6101     cz      finis   ; done if no file
                ;
                ;       source file open, prep destination
0124 11da01     lxi     d,dfcb  ; destination
0127  cd7301    call    delete  ; remove if present
```

```
                ;
012a 11da01             lxi     d,dfcb  ; destination
012d cd8201             call    make    ; create the file
0130 119601             lxi     d,nodir ; ready message
0133 3c                 inr     a       ; 255 becomes 0
0134 cc6101             cz      finis   ; done if no dir space
                ;
                ;       source file open, dest file open
                ;       copy until end of file on source
                ;
0137 115c00     copy:   lxi     d,sfcb  ; source
013a cd7801             call    read    ; read next record
013d b7                 ora     a       ; end of file?
013e c25101             lnz     eofile  ; skip write if so
                ;
                ;       not end of file, write the record
0141 11da01             lxi     d,dfcb  ; destination
0144 cd7d01             call    write   ; write record
0147 11a901             lxi     d,space ; ready message
014a b7                 ora     a       ; 00 if write ok
014b c46101             cnz     finis   ; end if so
014e c33701             jmp     copy    ; loop until eof
                ;
                eofile: ;  end of file, close destination
0151 11da01             lxi     d,dfcb  ; destination
0154 cd6e01             call    close   ; 255 if error
0157 21bb01             lxi     h,wrprot ; ready message
015a 3c                 inr     a       ; 255 becomes 00
015b cc6101             cz      finis   ; shouldn't happen
                ;
                ;       copy operation complete, 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.

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 before returning directly to the CCP. Thus, the dump program does not perform a 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 =          1f      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 cdo500             call bdos
0161 cldle1             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 cldle1             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     lbp
01a5 fe80               cpi     80h
01a7 c2b301             jnz     g0
                ;       read another buffer
                ;
                ;
01aa cdce01             call    diskr
01ad b7                 ora     a       ; zero value if read ok
01ae cab301             jz      g0      ; for another byte
                ;       end of data, return with carry set for eof
01b1 37                 stc
01b2 c9                 ret
                ;
                g0:     ;read the byte at buff+reg a
01b3 5f                 mov     e,a     ; 1s byte of buffer index
01b4 1600               mvi     d,0     ; double precision index to de
01b6 3c                 inr     a       ; index=index+1
01b7 321302             sta     ibp     ; back to memory
                ;       pointer is incremented
```

```
                ;       save the current file address
01ba 218000             lxi     h,buff
01bd 19                 dad     d
                ;       absolute character address is in hl
01be 7e                 mov     a,m
                ;       byte is in the accumulator
01bf b7                 ora     a       ; reset carry bit
01c0 c9                 ret
                ;
                setup:  ; set up file
                ;       open the file for input
01c1 af                 xra     a       ;zero to accum
01c2 327c00             sta     fcbcr   ; clear current record
                ;
01c5 115c00             lxi     d,fcb
01c8 0e0f               mvi     c,openf
01ca cd0500             call    bdos
                ;       255 in accum if open error
01cd c9                 ret
                ;
                diskr:  ; read disk file record
01ce e5d5c5             push h! push d! push b
01d1 115c00             lxi     d,fcb
01d4 0e14               mvi     c,readf
01d6 cd0500             call    bdos
01d9 cldle1             pop b! pop d! pop h
01dc c9                 ret
                ;
                ;       fixed message area
01dd 46494c0 signon:    db      'file dump version 2.0$'
01f3 0d0a4e0 opnmsg:    db      cr, 1f, 'no input file present on disk$'

                ;       variable area
0213            ibp:    ds      2       ; input buffer pointer
0215            oldsp:  ds      2       ; entry sp value from ccp
                ;
                ;       stack area
0217                    ds      64      ; reserve 32 level stack
                stktop:
                ;
0257                    end
```

A Sample Random Access Program

The following program is a rather extensive, but complete, example of a random access operation. The program performs the simple function of reading or writing random random records upon command from the terminal. Given that the program has been created, assembled and placed into a file labeled **RANDOM.COM**, the CPP 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 and 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. For 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 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
                ;
                ;
                notg:
                ;       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
015 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,1f        ; 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,1
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     1           ;+digit
020b 6f                 mov     1,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
                ;
                ;
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. Starting at both ends of the list, you examine the entry halfway inbetween. If not matched, you split either the upper half or the lower half for the next search. You'll quickly reach the item you are looking for (in log2(n) steps) where you will find the corresponding record number. Fetch and display this record at the console, just as has been done in the program shown above.

At this point, you are 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.

You can improve QUERY considerably by allowing boolean expressions which compute the set of records which satisfy several relationships. Examples are 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.

The Dynamic Debugging Tool

The **Dynamic Debugging Tool (DDT)** program allows dynamic interactive testing and debugging of programs generated in the CP/M environment. The DDT is generally used for assembler language programs. 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 Processor, and thus resides directly below the BDOS 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 TPA 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-D VEDR m.m

where **nn** is the memory size (which must match the CP/M system being used) 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.

DEL	remove the last character typed
CTRL U	remove the entire line, ready for retyping
CTRL 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 CTRL-C or G0 (jump to location 0000H), 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 TPA is 1234H then the number of pages is 12H, or 18 in decimal. Thus the operator could type a CTRL-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 disk, and can be directly executed simply by 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 location 100H through page 18 (12FFH). The machine state is not a part of the COM file. The program must be restarted from the beginning in order to test it properly.

DDT Commands

The individual commands are given below in some detail. With each command, 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 DEL key should be used instead if the program is executing a T or U command). In the explanation of each command, the command letter is shown in some cases with numbers separated by commas,

where the numbers are represented by lower case letters. These numbers are always assumed to be in a hexadecimal radix, and from one to four digits in length (longer numbers will be automatically truncated on the right).

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

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

As

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

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

Note that the assembler/disassembler portion of DDT can be overlayed by the transient program being tested, in which case the DDT program responds with an error condition when the A and L commands are used. (Refer to the debugging example at the end of this section.)

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 100H), 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 ccccccccccccccc

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. They 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 DEL key.

3. The F (Fill) 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** command with up to two optional breakpoint addresses. The G command takes one of the forms:

G	Gs,b,c
Gs	G,b
Gs,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 an 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 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. There is no intervention between the starting address and the break address by DDT. Therefore, if the program under test does not reach a breakpoint, control cannot return to DDT without executing an 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. If the current program counter is 1234H, then the commands:

G,1234

and

G400,400

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

5. The I (Input) Command. The **I** command allows the operator to insert a file name into the default file control block at 5CH. (The file control block created by CP/M for transient programs is placed at this location.) 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 by using the DEL 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 disk 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 000H through 0ffH (that is, the first page of memory). If b is omitted, then b=0000 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 100H is taken for COM files. Note that any number of R commands can be issued following the I command to reread 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 (100H for COM files, or taken from the last record if a HEX file is specified).

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

Ss

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

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

T

Th

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 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 occurs. A breakpoint can be forced in the trace mode by typing a DEL 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 disk drive, run in real-time, avoiding I/O timing problems. Programs running in trace mode execute approximately 500 times more slowly than in 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 an 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 DEL key to get control back to DDT during trace, rather than executing an RST 7. This ensures 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	(0/1)
E	Even Parity Flag	(0/1)
I	Interdigit Carry	(0/1)
A	Accumulator	(0/FF)
B	BC register pair	(0-FFFF)
D	DE register pair	(0-FFFF)
H	HL register pair	(0-FFFF)
S	Stack pointer	(0-FFFF)
P	Program Counter	(0-FFFF)

In the first case, the CPU register state is displayed in the format:

CfZfMfEfIf A=bb B=dddd D=dddd H=dddd P=dddd inst

where **f** is a 0 or 1 flag value, **bb** is a byte value, and **dddd** is a double byte quantity corresponding to the register pair. The **inst** field contains the disassembled instruction which occurs at the location addressed by the CPU state's program counter.

The second form allows display and optional alteration of register values, where r is one of the registers given above (C, Z, M, E, I, A, B, D, H, S or P). In each case, the flag or register value is first displayed at the console. The DDT program then accepts input from the console. If a carriage return is typed, then the flag or register value is not altered. If a value in the proper range is typed, then the flag or register value is altered. Note that BC, DE and HL are displayed as register pairs. Thus, the operator types the entire register pair when B, C, or the BC pair is altered.

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 CCP. Although it is 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. This further reduces 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). The trace and display (T and X) commands list the **inst** field of the display in hexadecimal, rather than as a decoded instruction.

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
*I                                        rubout echo
↵  ↑-I   ORG   ↑-I  100H    L-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↵
 Rubout  SUB       C        ;LARGER VALUE IN C?↵
 deletes JNC       NFOUND   ;JUMP IF LARGER VALUE NOT FOUND↵
;character 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↵
↑Z       END↵
*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
```

```
ASM SCAN        Start Assembler
CP/M ASSEMBLER - VER 1.0

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

```
TYPE SCAN.PRN
Code Address    Source Program
    Machine Code
0100                    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  0200040305  VECT: DB      2, 0, 4, 3, 5, 6, 1, 5
0008  =           LEN   EQU     $-VECT      ;LENGTH
0121              LARGE: DS     1           ;LARGEST VALUE ON EXIT
0122                    END
A>
      Value of
      Equate
```

```
DDT SCAN HEX↵      Start Debugger using hex format machine code

16K DDT VER 1.0
NEXT PC                                              next instruction
0121 0000                                            to execute at
-X↵                                                  PC=0
      last load address +1
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0000 OUT 7F
-XP↵
                Examine registers before debug run
P=0000 100↵  Change PC to 100
                                                     Next instruction
-X↵  Look at registers again          PC changed     to execute at PC=100
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI B,00
-L100↵

0100  MVI   B,08
0102  MVI   C,00
0104  LXI   H,0119
0107  MOV   A,M
0108  SUB   C
0109  JNC   010D
010C  MOV   C,A
010D  INX   H
010E  DCR   B
010F  JNZ   0107
0112  MOV   A,C
-L↵

0113  STA   0121
0116  JMP   0000
0119  STAX  B
011A  NOP
011B  INR   B
011C  INX   B
011D  DCR   B
011E  MVI   B,01
0120  DCR   B
0121  LXI   D,2200
0124  LXI   H,0200
-A116↵  enter inline assembly mode to change the JMP to 0000 into a RST 7, which
0116  RST 7↵   will cause the program under test to return to DDT if 116H
                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
0116  RST   07  ←— in place of JMP
```

Disassembled Machine Code at 100H (See Source Listing for comparison)

A little more machine code (note that program ends at location 116 with a JMP to 0000)

```
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
-T    Execute Program for one stop.   initial CPU state, before  is executed

C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI B,08*0102
-T    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
-T    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=0000 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
                                        Automatic breakpoint at 10DH
      Display memory starting at 119H.
                                       Program data
0119  02  00  04  03  05  06  01       .......                  lower case x
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     Data is displayed
0170  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00     in ASCII with a "•"
0180  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00     in the position of
0190  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00     non-graphic
01A0  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00     characters
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
-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
-U5                                                     Automatic
       Trace without listing intermediate states         Breakpoint
C0Z0M0E01I1 A=00 B=0700 D=0000 H=011A S=0100 P=0109 JNC 010D*0108
-X     CPU State at end of US

C0Z0M0E01I1 A=04 B=0600 D=0000 H=011B S=0100 P=0108 SUB C
```

PART 2

```
-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↵
```
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

```
0109 JC 10D↵

010C↵

-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
        Trace to see how patched version operates
-T10
                              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↵
1023 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
-L↵                    } Review the Code
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

-T↵   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
```

```
SAVE 1 SCAN.COM↵          Save memory image
                          Restart DDT
A>DDT SCAN.COM↵

16K DDT VER 1.0
NEXT PC
0200 0100
-XP↵

P=0100↵

-L116↵

0116  RST  07
0117  NOP
0118  NOP
0119  STAX B
011A  NOP
- (rubout)

                          } Look at code to see if it was properly loaded
                            (long typeout aborted with 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↑ZOLT↵
      CMP       C     ;LARGER VALUE IN C?
*
      JNC       NFOUND ;JUMP IF LARGER VALUE NOT FOUND
*SNC↑ZC↑ZOLT↵
      JC        NFOUND ;JUMP IF LARGER VALUE NOT FOUND
*F↵
```

```
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  08  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
```

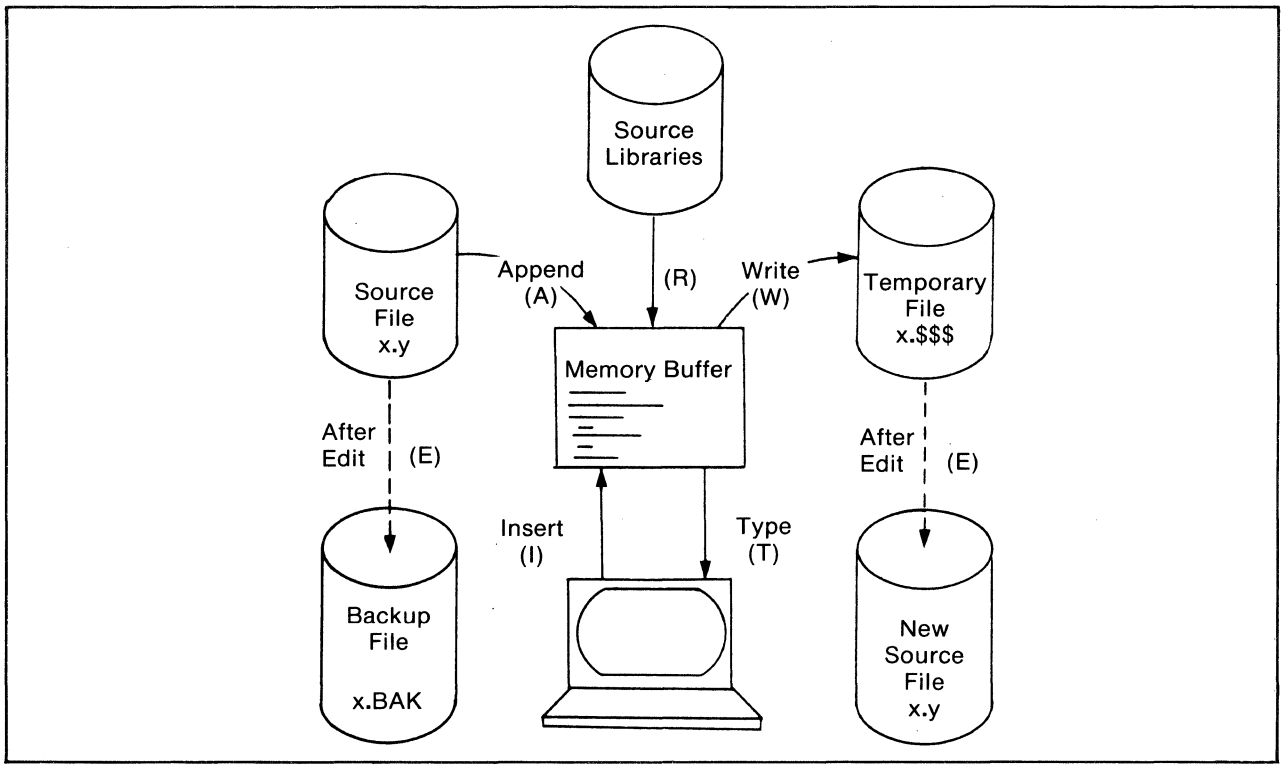


Figure 32 *Overall ED Operation*

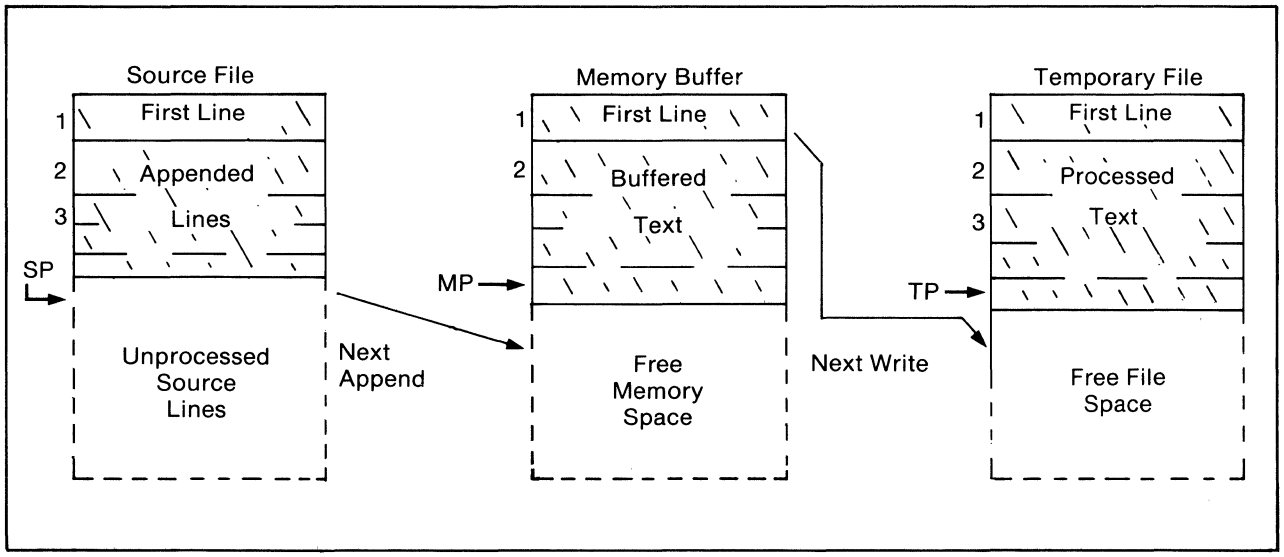


Figure 33 *Memory Buffer Organization*

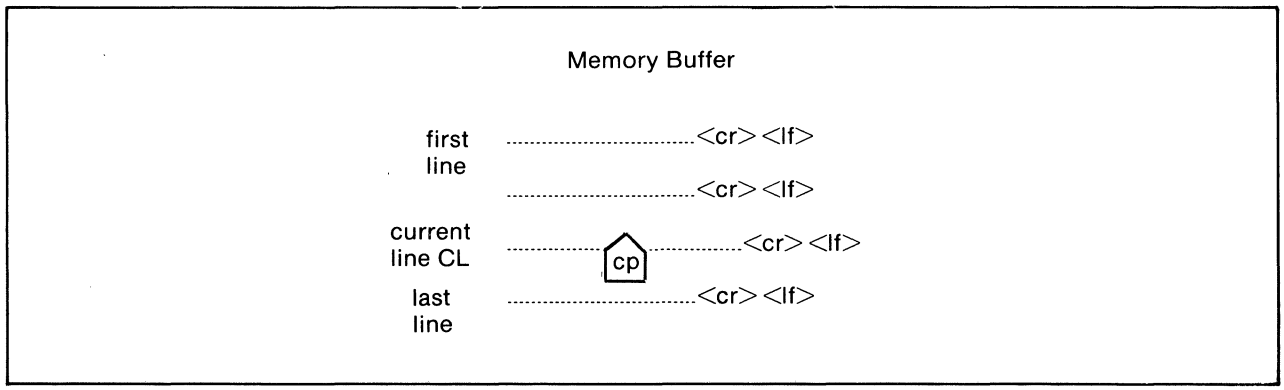


Figure 34 *Logical Organization of Memory Buffer*

The Text Editor

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

ED filename

ED filename.filetype

In general, ED reads segments of the source file given by **filename** or **filename.filetype** into central memory, where the file is manipulated by the operator, and subsequently written back to disk after alterations. If the source file does not exist before editing, it is created by ED and initialized to empty. The overall operation of ED is shown in **figure 32.**

ED Operation

ED operates upon the source file, denoted in figure 32, by **x.y**, and passes all text through a memory buffer where the text can be viewed or altered. 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 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 33.**

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.

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.

COMMAND FORM*	RESULT
nA	Appends the next n unprocessed source lines from the source file at SP to the end of the memory buffer at MP. Increments SP and MP by n.
nW	Writes the first n lines of the memory buffer to the temporary file free space. Shifts the remaining lines n+1 through MP to the top of the memory buffer. Increments TP by n.
E	Ends the edit. Copies all buffered text to temporary file, and copies all unprocessed source lines to the temporary file. Renames files as described previously.
H	Moves 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).

COMMAND FORM*	RESULT
O	Returns to original file. The memory buffer is emptied, the temporary file is deleted, and the SP is returned to position 1 of the source file. The effects of the previous editing commands are thus nullified.
Q	Quits edit with no file alterations, return to CP/M.

*Each command is followed by a carriage return.

A number of special cases should be considered. 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 one line, respectively. In addition, if a pound sign (#) is given in place of n, then the integer 65535 is assumed (the largest value of 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 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 assumed if the memory bufffer size is exceeded. You may then enter any command (such as W) which does not increase memory requirements. The remainder of any partial line not read during the overflow will be brought into memory on the next successful append.

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 34**. The dashes represent characters of the source lines of indefinite length, terminated by carriage return (cr) and line feed (lf) characters, and ⬠cp⬡ represents the imaginary character pointer. Note that the CP is always located **ahead** of the first character of the first line, **behind** the last character of the last line, or **between** two characters. The current line CL is the source line which contains the CP.

Memory Buffer Operation

Upon initiation of ED, the memory buffer is empty (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 (carriage return)

ED then accepts any number of input lines, where each line terminates with a cr (the lf is supplied automatically), until a CTRL-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)

CTRL-Z

leaves the memory buffer as shown below:

NOW IS THE (cr) (lf)
TIME FOR (cr) (lf)
ALL GOOD MEN (cr) (lf)

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 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.

COMMAND FORM*	RESULT
±**B**	Moves CP to beginning of memory buffer if +, and to bottom if –.
±**nC**	Moves CP by ±n characters (toward front of buffer if +), counting the (cr) (lf) as two distinct characters.
±**nD**	Deletes **n** characters ahead of CP if plus, and behind CP if minus.
±**nk**	Kills (removes) ±**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**	If **n=0** then moves CP to the beginning of the current line (if it is not already there). If **n does not equal 0,** then first moves the CP to the beginning of the current line, and then moves 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**	If **n=0** then types the contents of the current line up to CP. If **n=1** then types the contents of the current line from CP to the end of the line. If **n** is greater than 1 then types the current line along with **n-1** lines which follow, if + is specified. Similarly, if **n** is greater than **1** and – is given, types the previous n lines, up to the CP. The break key can be depressed to abort long type-outs.
±**n**	Equivalent to ±**nLT**, which moves up or down and types a single line.

*Each command is followed by a carriage return.

Command Strings

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

DEL	Remove the last character
CTRL-U	Delete the entire line
CTRL-C	Re-initialize the CP/M System
CTRL-E	Return carriage for long lines without transmitting buffer (max 128 characters)

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)	Moves to beginning of buffer and types 2 lines: **NOW IS THE** **TIME FOR**	NOW IS THE(cr) (lf) TIME FOR(cr) (lf) ALL GOOD MEN(cr) (lf)
2.	5C0T(cr)	Moves CP 5 characters and types the beginning of the line **NOW I**	NOW I⟨cp⟩S the(cr) (lf)
3.	2L-T(cr)	Moves two lines down and types previous line **TIME FOR**	NOW IS THE(cr) (lf) TIME FOR(cr) (lf) ALL GOOD MEN(cr) (lf)
4.	-L#K(cr)	Moves up one line, deletes 65535 lines which follow	NOW IS THE (cr) (lf) ⟨cp⟩
5.	I(cr) TIME TO(cr) INSERT(cr) CTRL-Z	Inserts two lines of text	NOW IS THE(cr) (lf) TIME TO(cr) (lf) INSERT(cr) (lf) ⟨cp⟩
6.	-2L#T(cr)	Moves up two lines, and types 65535 lines ahead of CP **NOW IS THE**	NOW IS THE(cr) (lf) ⟨cp⟩ TIME TO (cr) (lf) INSERT(cr) (lf)
7.	(cr)	Moves down one line and types one line **INSERTS**	NOW IS THE(cr) (lf) TIME TO(cr) (lf) ⟨cp⟩ INSERT(cr) (lf)

Text Search and Alteration

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

$$nF\ c_1c_2...c_k \left\{ \begin{matrix} (cr) \\ (CTRL\text{-}Z) \end{matrix} \right\}$$

where c_1 through c_k represent the characters to match followed by either a (cr) or CTRL-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 CTRL-L, which is replaced by the pair of symbols (cr) (lf).

The following commands illustrate the use of the F command:

	COMMAND STRING BUFFER	EFFECT	RESULTING MEMORY
1.	B#T(cr)	Moves to beginning and types entire buffer	NOW IS THE(cr) (lf) TIME FOR(cr) (lf) ALL GOOD MEN(cr) (lf)
2.	FS T(cr)	Finds the end of the string **S T**	NOW IS T HE(cr) (lf)
3.	FI (up arrow up) 0TT	Finds the next **I** and types to the CP, then types the remainder of the current line: **TIME FOR**	NOW IS THE (cr) (lf) TI ME FOR(cr) (lf) 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:

$$I\ c_1c_2 \ldots c_n\ (\text{CTRL-Z}) \quad \text{or}$$

$$I\ c_1c_2 \ldots c_n(\text{cr})$$

where $\mathbf{c_1}$ through $\mathbf{c_2}$ are characters to insert. If the insertion string is terminated by a (CTRL-Z), the characters $\mathbf{c_1}$ through $\mathbf{c_n}$ are inserted directly following the CP, and the CP is moved directly after character c_n. The action is the same if the command 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 (CTRL-Z) (cr)	Inserts **THIS IS** at the beginning of the text	THIS IS NOW THE(cr)(lf) TIME FOR(cr)(lf) ALL GOOD MEN(cr)(lf)
FTIME (CTRL-Z)-4DIPLACE(CTRL-Z)(cr)	Finds TIME and deletes it; then inserts **PLACE**	THIS IS NOW THE(cr)(lf) PLACE for (cr)(lf) ALL GOOD MEN(cr)(lf)
3FO(CTRL-Z)-3DICHANGES(CTRL-Z)(cr)	Finds third occurrence of **O** (the second **O** in GOOD), deletes previous 3 characters, then inserts **CHANGES**	THIS IS NOW THE(cr)(lf) PLACE FOR(cr)(lf) ALL CHANGES (cr)(lf)

COMMAND STRING	EFFECT	RESULTING MEMORY BUFFER
−8CISOURCE(cr)	Moves back 8 characters And inserts the line **SOURCE** (cr)(lf)	THIS IS NOW THE(cr)(lf) PLACE FOR(cr)(lf) ALL SOURCE(cr)(lf) (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:

$$n\ S\ c_1c_2...c_k(\text{CTRL-Z})\ d_1d_2...d_m \left\{\begin{matrix}(\text{cr})\\ \text{CTRL-Z}\end{matrix}\right\}$$

and has exactly the same effect as applying the command string:

$$F\ c_1c_2...c_k(\text{CTRL-Z})\text{-kDI}d_1d_2...d_m \left\{\begin{matrix}(\text{cr})\\ \text{CTRL-Z}\end{matrix}\right\}$$

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:

$$n\ N\ c_1c_2...c_k \left\{\begin{matrix}(\text{cr})\\ \text{CTRL-Z}\end{matrix}\right\}$$

which searches the entire source file for the nth occurrence of the string $\mathbf{c_1c_2...c_k}$ (recall the 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 (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:

$$n\ J\ c_1c_2...c_k(\text{CTRL-Z})\ d_1d_2...d_m(\text{CTRL-Z})\ e_1e_2...e_q \left\{\begin{matrix}(\text{cr})\\ \text{CTRL-Z}\end{matrix}\right\}$$

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

(cp) NOW IS THE TIME(cr)(lf)

Then the command:

JW (CTRL-Z)WHAT(CTRL-Z)(CTRL-Z)1(cr)

Results in:

NOW WHAT (cp) (cr)(lf)

Recall that (CTRL-Z) 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.

Source Libraries

ED also allows the inclusion of **source libraries** during the editing process with the R command. The form of this command is

$$R\ f_1f_2...f_n(\text{CTRL-Z})z \qquad \text{or}$$
$$R\ f_1f_2...f_n(\text{cr})$$

where $\mathbf{f_1f_2...fn}$ is the name of a source file on the disk with an 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.

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:

$$n\ M\ c_1c_2...c_k \left\{\begin{array}{c}(\text{cr})\\ \text{CTRL-Z}\end{array}\right\}$$

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. (For example, 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(CTRL-Z)-5DIDELTA(CTRL-Z)0TT(cr)

or equivalently

MSGAMMA(CTRL-Z)DELTA(CTRL-Z)0TT(cr)

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 (for example, 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 X

where **x** is the disk drive. 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.

Control Characters and Commands

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

CONTROL CHARACTER	FUNCTION
CTRL-C	System reboot
CTRL-E	Physical (cr)(lf) (not actually entered in command)
CTRL-I	Logical tab (columns 1, 8, 15, ...)
CTRL-L	Logical (cr)(lf) in search and substitute strings
CTRL-U	Line delete
CTRL-Z	String terminator
DEL	Character delete
Break	Discontinue command (for example, stop typing)

Summary of Commands

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

COMMAND	FUNCTION
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)

Line Numbers

ED 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 five blanks.

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

345:T

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

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

:400T

is interpreted as **type from the current line number through the line whose absolute number is 400**. Note that absolute line references of this sort can precede any of the standard Ed commands.

Free Space Interrogation

A special case of the V command (0V) 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.

Block Move Facility

ED 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 lines accumulate one after another in this file, and can be retrieved simply by 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 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

0X

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 CTRL-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.)

Errors

Due to common typographical errors, ED requires that several potentially disastrous commands 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 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.

Other Notes on ED

ED has 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 carriage returns. Although the CP/M has a limited memory work space area (approximately 5000 characters in a 16K CP/M system), the file size which can be edited is not limited, since data is easily "paged" through this work area.

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

Given that the operator has typed

ED X.ASM (cr)

the ED program creates an intermediate work file with the name

X.$$$

to hold the edited data during the ED run. Upon completion of ED, the X.ASM file (original file) is renamed to X.BAK, and the edited work file is renamed to X.ASM. Thus, the X.BAK file contains the original (unedited) file, and the X.ASM file contains the newly edited file. The programmer 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 commands 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 programmer can abort the edit at any point (reboot, power failure, CTRL-C or Q command) without destroying the original file. In this case, the BAK file is not created, and the original file is always intact.

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

ED file name x:

where the file name exists on the currently logged disk and is to be edited, and x is the name of an alternate drive. The ED program reads and processes the source file, and writes the new file to the specified alternate drive using the file name given in the command above. Upon completion of processing, the original file becomes the backup file. Thus, if the programmer is addressing disk A, the following command is valid:

ED X.ASM B:

which edits the file **X.ASM** on drive, 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 accidentally destroying a source file. In this case, the programmer 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 follow:

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

ED B:X.ASM A: Edits the file X.ASM on drive B to the temporary file X.$$$ on drive A. On termination of editing, changes X.ASM on drive B to X.BAK, and changes X.$$$ on drive A to X.ASM.

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 programmer 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.

APPENDICES

APPENDIX A
INSTALLATION

The first step in installing your computer is to make sure that various requirements are met. These are described at the beginning of this section.

Environment Requirements

Operating Environment:

Temperature	10°...32°C (50°...90°F)
Relative humidity	20...80%
Temperature gradient	10°C/hour
Vibration	Max. 0.3G at 2-60 Hz
Altitude	Up to 2,440m above sea level

Nonoperating Environment:

Temperature	−10°...55°C (−14°...131°F)
Relative humidity	5...95%
Temperature gradient	10°C/hour
Vibration	Max. 0.6G at 3-60 Hz

Other Conditions:

Do not install the system in a place exposed to the following conditions:

- A great deal of dust
- Strong vibrations
- Strong electromagnetic field
- Corrosive gases
- Sunshine
- Non-horizontal surface

Physical Dimensions:

Size	Width	Height	Depth	
T200 Console	550	329	432	mm
T250 Console	620	390	470	mm
Keyboard	444	73	215	mm
Printer	560	188	368	mm

Weight

T200 Console with dual drive unit	30 kg
T250 Console with dual drive unit	39 kg
Keyboard	1 kg
Printer	15 kg

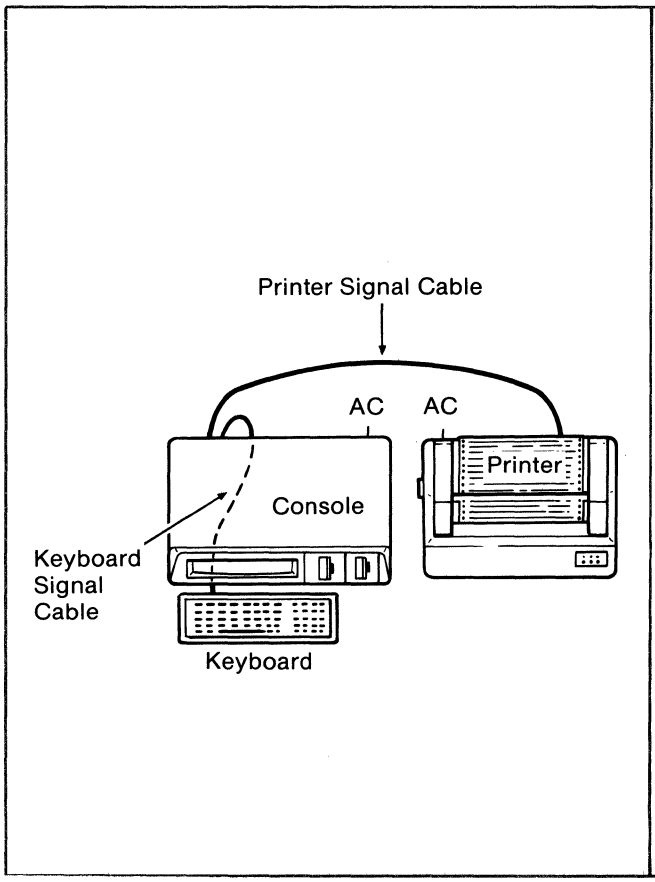


Figure 35 *Signal Cable Connection*

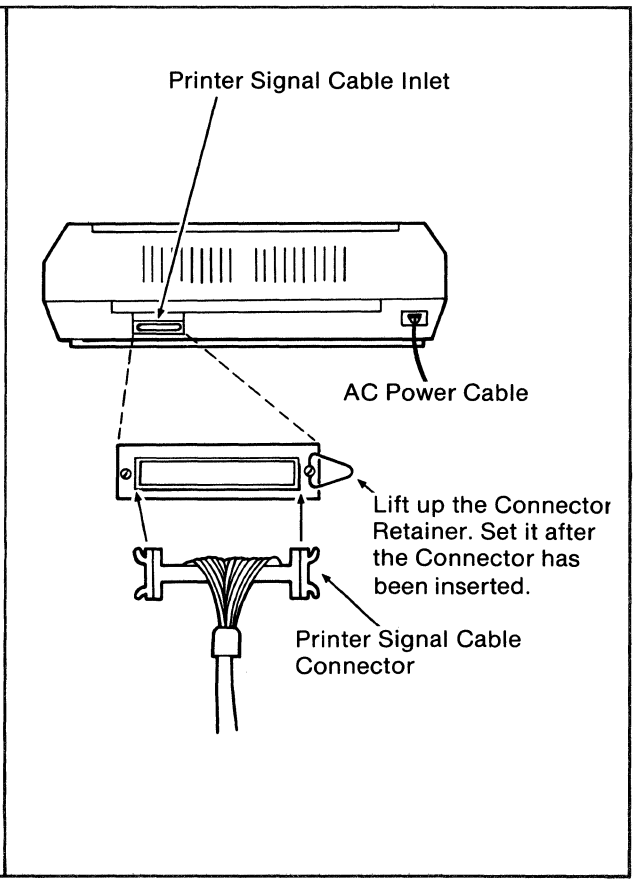


Figure 36 *Connecting Printer Signal Cable*

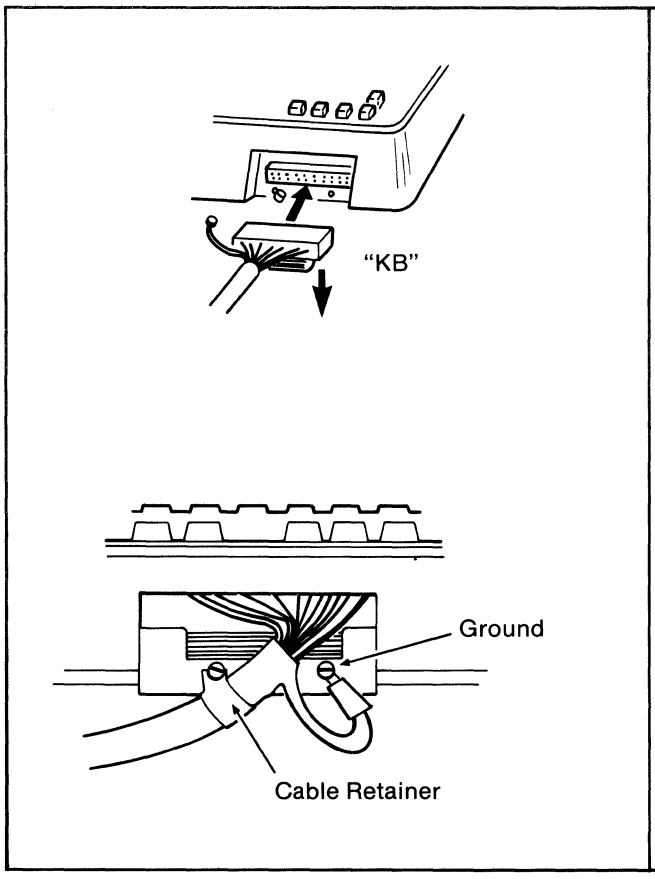


Figure 37 *Connecting Keyboard Signal Cable*

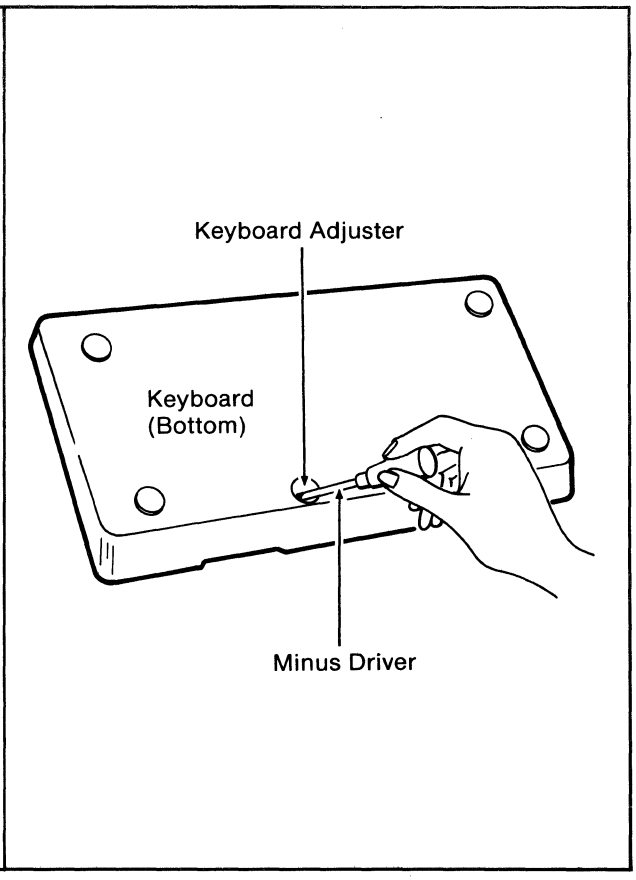


Figure 38 *Keyboard Click Adjuster*

Electrical Requirements:

Two 15-ampere, single-phase grounded receptacles are required for each T200 or T250 system. Power cords are attached to the console and the printer each 2.5 meters (8.3 feet) long. The AC voltage requirements are:

60 Hz ± ½

115 V ± 10%

The power requirements are:

Console and Keyboard	Max. 0.3 kVA
Printer	Max. 0.2 kVA

After the various requirements have been satisfied, you connect the cabling.

Cabling

Two interunit signal cables come out of the back of the console. One connects to the keyboard. The other connects to the printer.

The printer signal cable has a 36-pin connector with plastic hood. This is to be connected to the inlet located at the back of the printer. See **figure 35**.

The keyboard signal cable has a 50-pin connector with grounding lead. Connect the signal cable to the inlet located at the back of the keyboard.

Remove the cable retainer screw on the keyboard inlet.

Attach the connector to inlet with the label "KB" and pull the tab facing downward.

When you are sure that the connector is placed correctly, push it firmly.

Attach the cable retainer around the cable and the grounding lead to the ground screw.

See **figures 36 and 37**.

After you have satisfied the various requirements and hooked up the cabling, adjust the keyboard and you will be ready to use your computer.

Adjusting the Keyboard Click

The keyboard has an adjuster for key-in click sound. It is located at the bottom of the keyboard as shown in **figure 38**. Turning the adjuster to the right increases the click sound.

APPENDIX B
CHARACTER CODE TABLE

b8	b7	b6	b5	b4	b3	b2	b1		0	0	0	0	0	0	0	0
									0	0	0	0	1	1	1	1
									0	0	1	1	0	0	1	1
									0	1	0	1	0	1	0	1
									0	1	2	3	4	5	6	7
				0	0	0	0	0		P-STOP CTRL/P	SP	0	@	P		p
				0	0	0	1	1	CTRL/A	CTRL/Q	!	1	A	Q	a	q
				0	0	1	0	2	PF4	CTRL/R	"	2	B	R	b	r
				0	0	1	1	3	BREAK CTRL/C	CTRL/S	#	3	C	S	c	s
				0	1	0	0	4	PF5	CTRL/T	$	4	D	T	d	t
				0	1	0	1	5	CTRL/E	CAN CRTL/U	%	5	E	U	e	u
				0	1	1	0	6	PF6	CTRL/V	&	6	F	V	f	v
				0	1	1	1	7	PF7	CTRL/W	'	7	G	W	g	w
				1	0	0	0	8	CTRL/H	CTRL/X	(	8	H	X	h	x
				1	0	0	1	9	TAB	CRTL/Y	)	9	I	Y	i	y
				1	0	1	0	A	CTRL/J	CTRL/Z	*	:	J	Z	j	z
				1	0	1	1	B	PF8	ESC	+	;	K	[	k	PFO
				1	1	0	0	C	CTRL/L		,	<	L	\	l	PF1
				1	1	0	1	D	CR		−	=	M	]	m	PF2
				1	1	1	0	E	PF9		.	>	N	^	n	PF3
				1	1	1	1	F	CTRL/O	■	/	?	O	_	o	DEL

APPENDICES

- The carriage return is sent when either the carriage return key or the ENTER key is pressed.
- To implement reverse video, press the CTRL and 1 keys. Pressing these keys again resumes the normal display mode.
- The decimal representation of the program function keys is as follows:

PF0	123
PF1	124
PF2	125
PF3	126
PF4	2
PF5	4
PF6	6
PF7	7
PF8	11
PF9	14

The BASIC programs can use these decimal values to decide which program function key is pressed.

Example:

```
110     LET     A$=INPUT$(1)
120     IF  ASC(A$)=2 THEN 300
        ELSE    400
        .
        .
        .
300     REM PROCESSING FOR PF4 KEY
```

The statement 110 accepts the input of one character (without the carriage return) from the keyboard. The statement decides if the key PF4 has been pressed and branches accordingly.

APPENDIX C
DISK CHARACTERISTICS

T200:

Sectors per cylinder	32
Bytes per sector	256
Storage capacity in bytes	286,720

T250:

	1-sided	**2-sided**
Sectors per cylinder	26	52
Bytes per sector	128	256
Storage capacity in bytes	256,256	1,025,024

APPENDIX D
THE PRINTER

Toshiba offers a choice of printers for use with your T200 or T250. The standard matrix printer has a printing speed of **125 characters per second**. It is capable of printing **upper and lower case characters, numeric characters** and **special symbols.** The printer prints in both directions to reduce print time. Other features are as follows:

Print font	9 X 7 dot matrix
Characters per line	136
Character spacing	10 characters per inch
Line spacing	6 lines per inch
Multicopies	Up to 3 including original

Operating the Printer

Turn on Power

The power switch is located on the back left side. When the power is turned on, the POWER light is lit on the control panel. See **figure 39**.

Select Mode

If the SELECT light is not lit on the control panel, **press** the SELECT push button. This sets the printer to the SELECT mode. The printer can print information when it is in the SELECT mode.

Deselect Mode

The LINE FEED and TOP OF FORM push buttons are used to feed the paper manually. These buttons are enabled when the printer is in the DESELECT mode.

- If the SELECT light is **on**, **push** the SELECT button. This places the printer in the DESELECT mode and turns the SELECT **light off**.
- **One** line feed takes place each time the LINE FEED button is pressed.
- When the TOP OF FORM button is **pressed**, the paper advances to the next top-of-page position. There are **66 print lines** from a top-of-page to the next. See **figure 40**.

Paper Loading

Before setting the new paper, **press** the TOP OF FORM button in the DESELECT mode. Insert the paper through the back of the printer. See **figure 41**.

Open the front view cover. **Lift** the paper holders up. **Grasp** the end of the paper. See **figure 42**. Make sure that the print ribbon is not loosened. If it is loosened, open the top cover and tighten the ribbon.

Lift up the paper and **set** the holes on both sides of paper to the tractor pins. See **figure 43**.

Adjust the vertical paper position right on the first print line of a paper at this time. See **figure 44**.

Press down the paper holders. See **figure 45**

Turn the feed knob for vertical alignment of paper. See **figure 46**.

Paper Empty Status

The end of paper is detected by a microswitch located in front of the platen. When this occurs, the PAPER EMPTY light is **lit** on the control panel and the SELECT light is turned **off**. At this time, if the SELECT button is **pressed**, the PAPER EMPTY light will go out after printing one character. The PAPER EMPTY light is lit again and this step can be repeated.

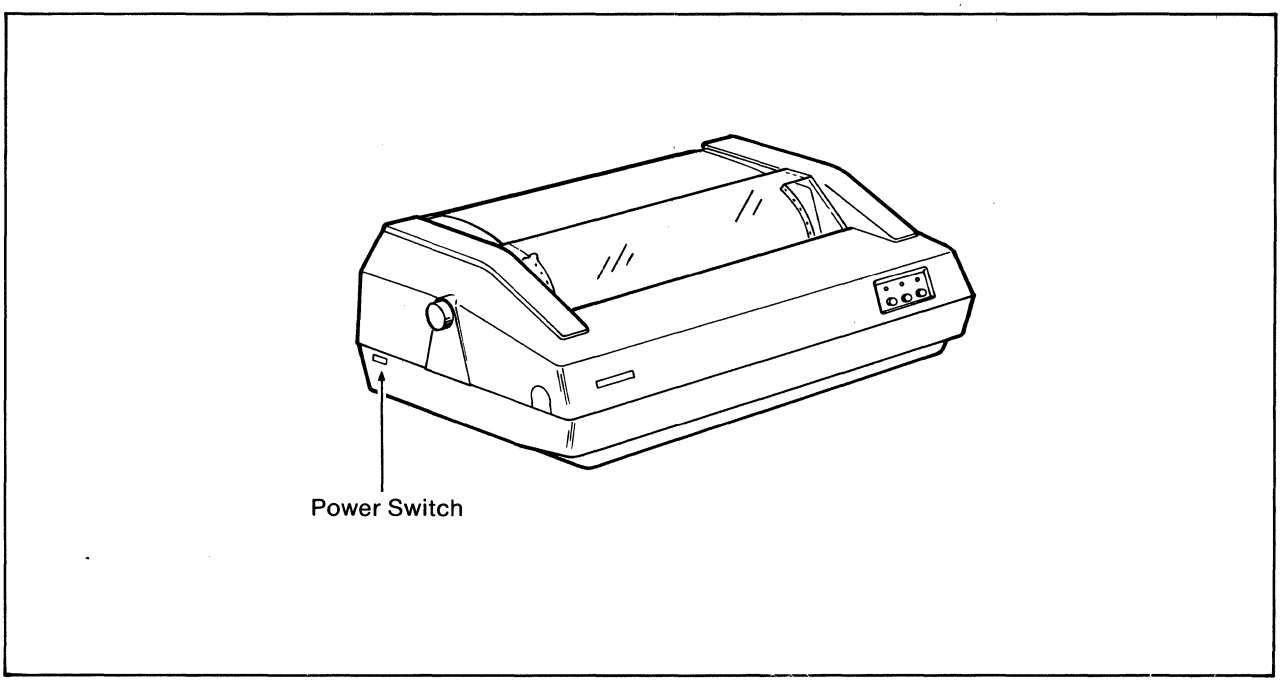


Figure 39 *Printer Power Switch*

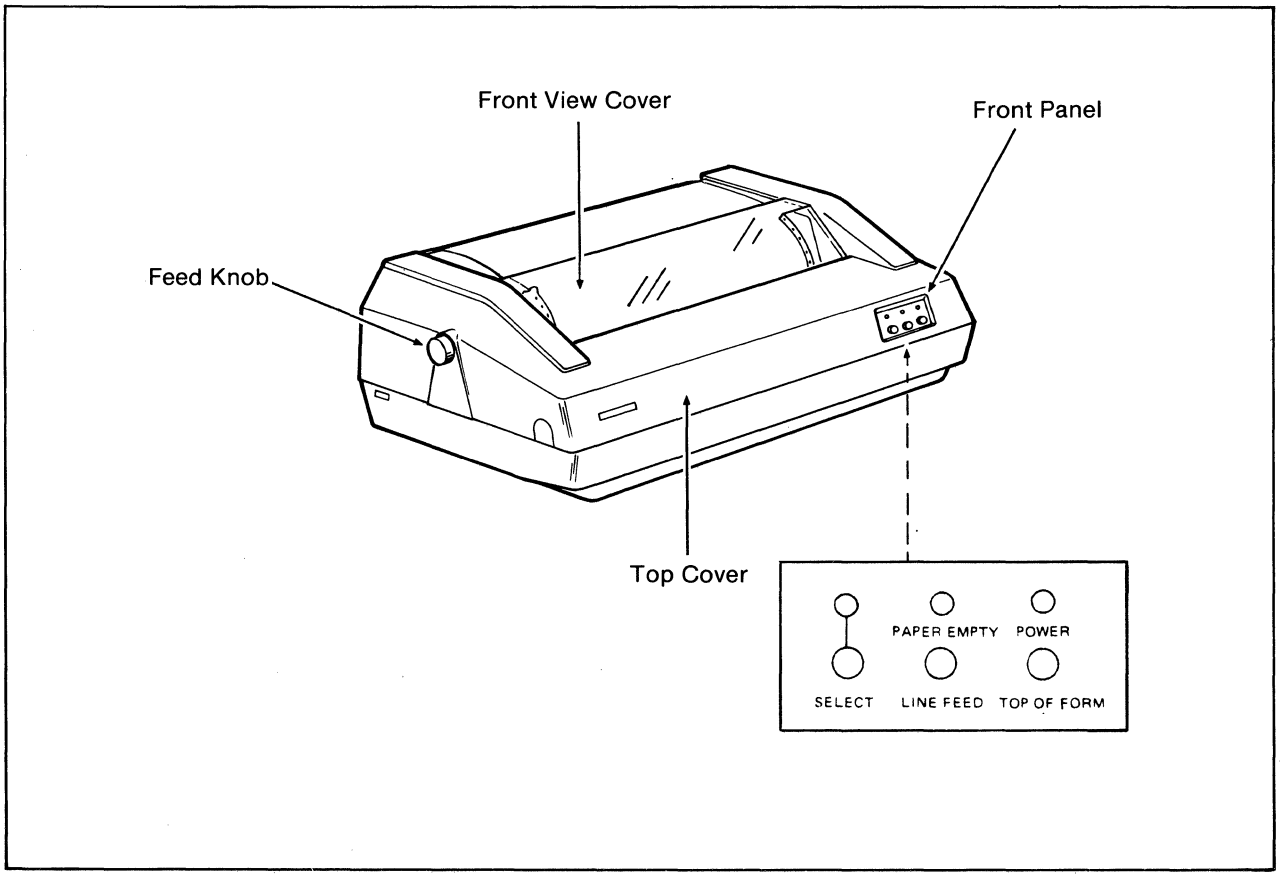


Figure 40 *Printer*

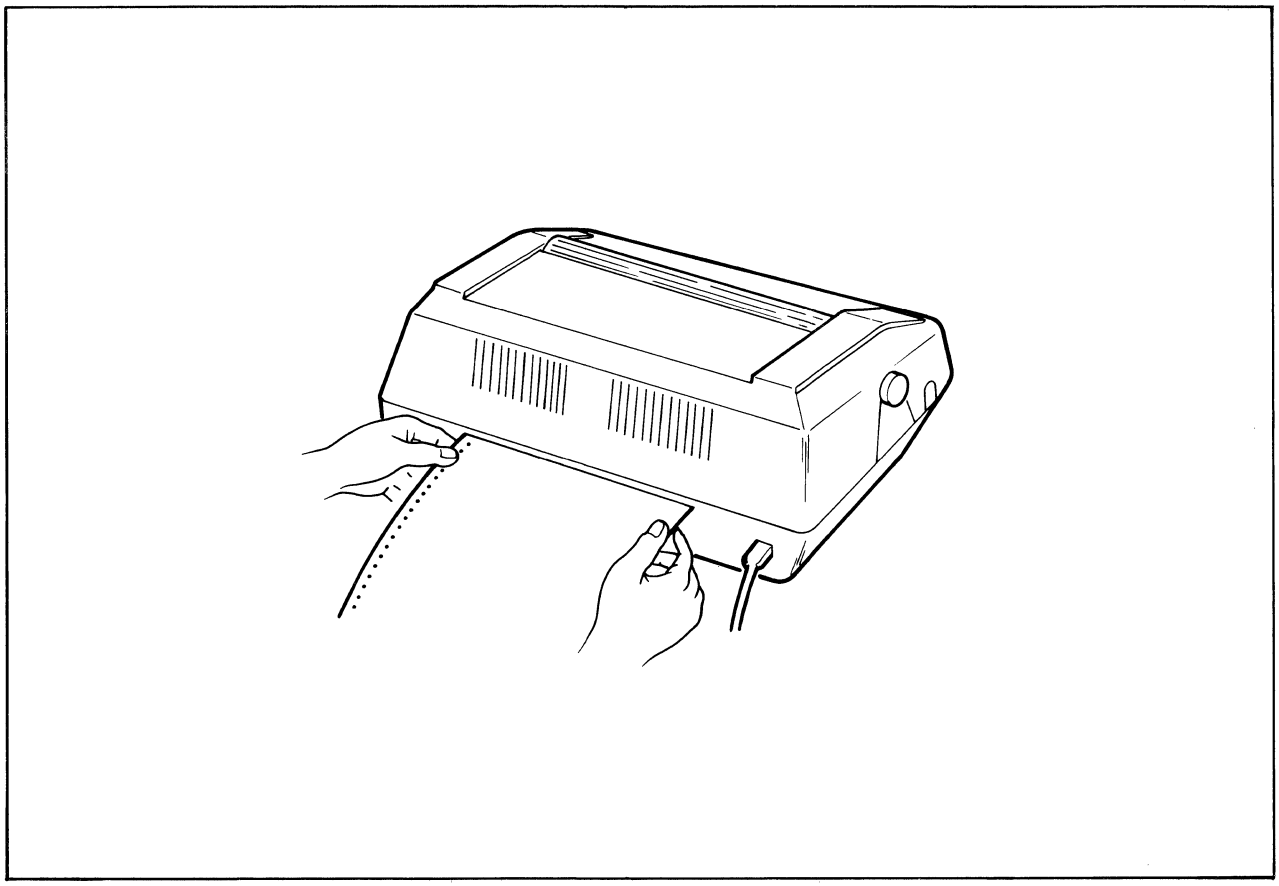


Figure 41 *Inserting the Paper*

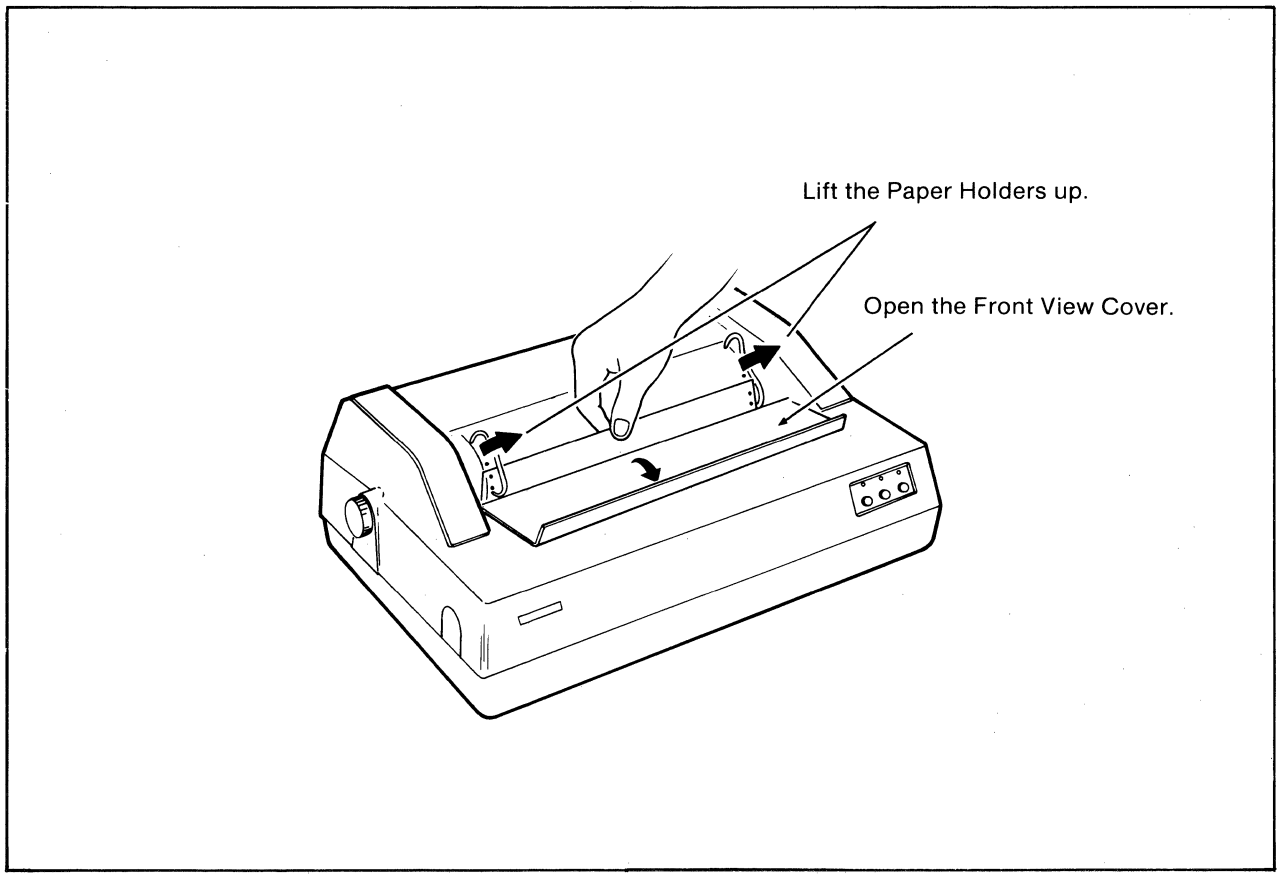


Figure 42 *Opening the Paper Cover*

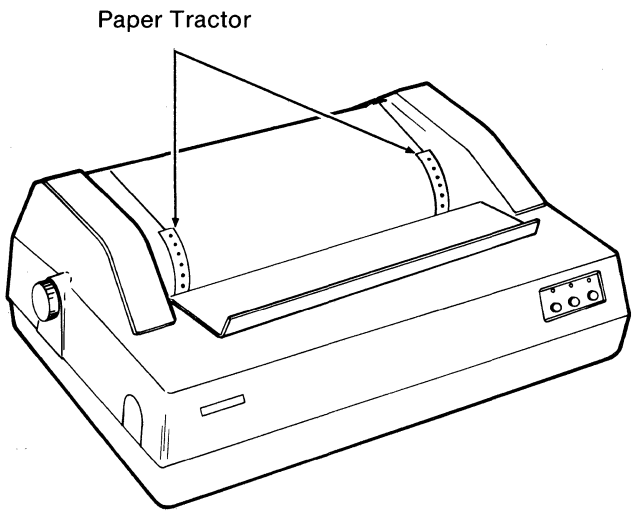


Figure 43 *Setting the Paper on Tractor Pins*

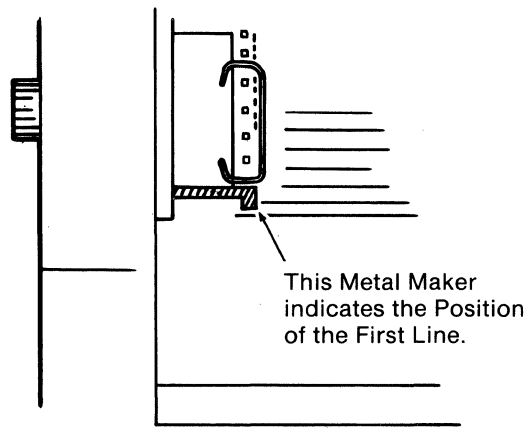


Figure 44 *Adjusting the Paper Position*

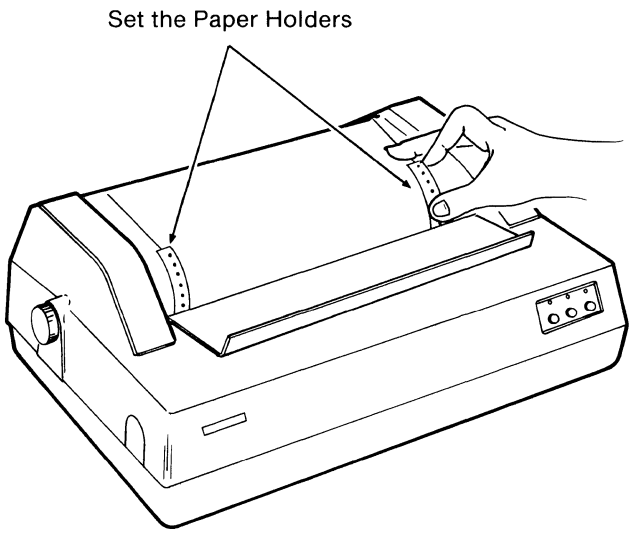


Figure 45 *Setting the Paper Holders*

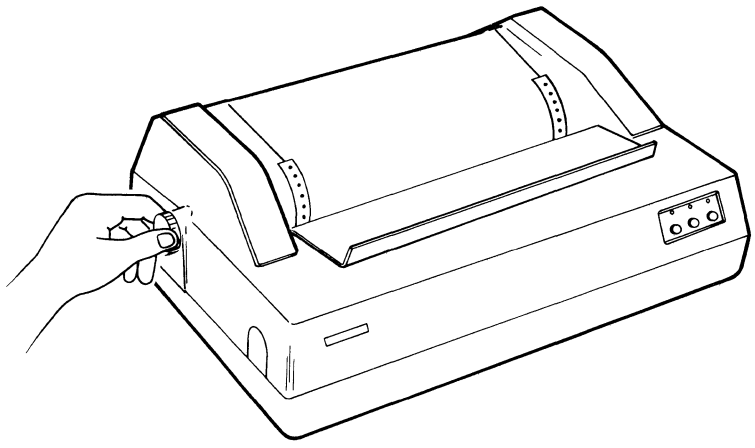


Figure 46 *Turning the Paper Feed Knob*

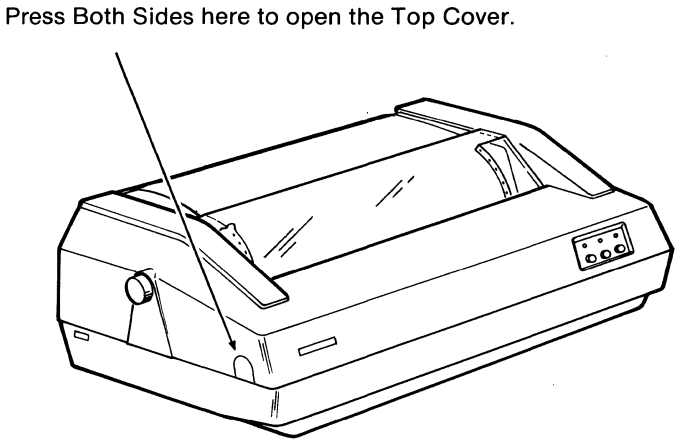


Figure 47 *Opening the Top Cover*

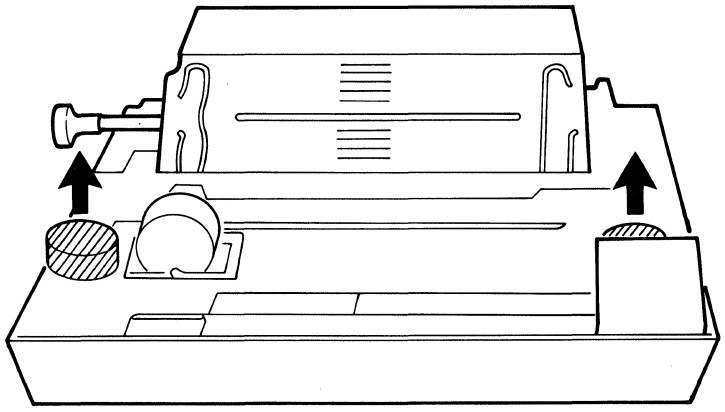


Figure 48 *Removing the Ribbon Spools*

To look at the ribbon, **press** the latch on both sides of the cover and open it. See **figure 47.**

Remove the ribbon and spools which are loaded. The spools come out vertically. See **figure 48.**

Inspect the removed ribbon. If the upper and lower portions of the ribbon are worn out, the ribbon cannot be used. Keep the empty spool. If only one side of the ribbon has been used, **turn** the spools over and load the ribbon again.

If the new ribbon is to be used, **attach** the end of the new ribbon to the empty spool and **wind** the ribbon until the eyelet is wound over the empty spool. See **figure 49.**

To **load** the ribbon, thread the ribbon through the ribbon guides as indicated in **figure 50.** **Close** the top cover.

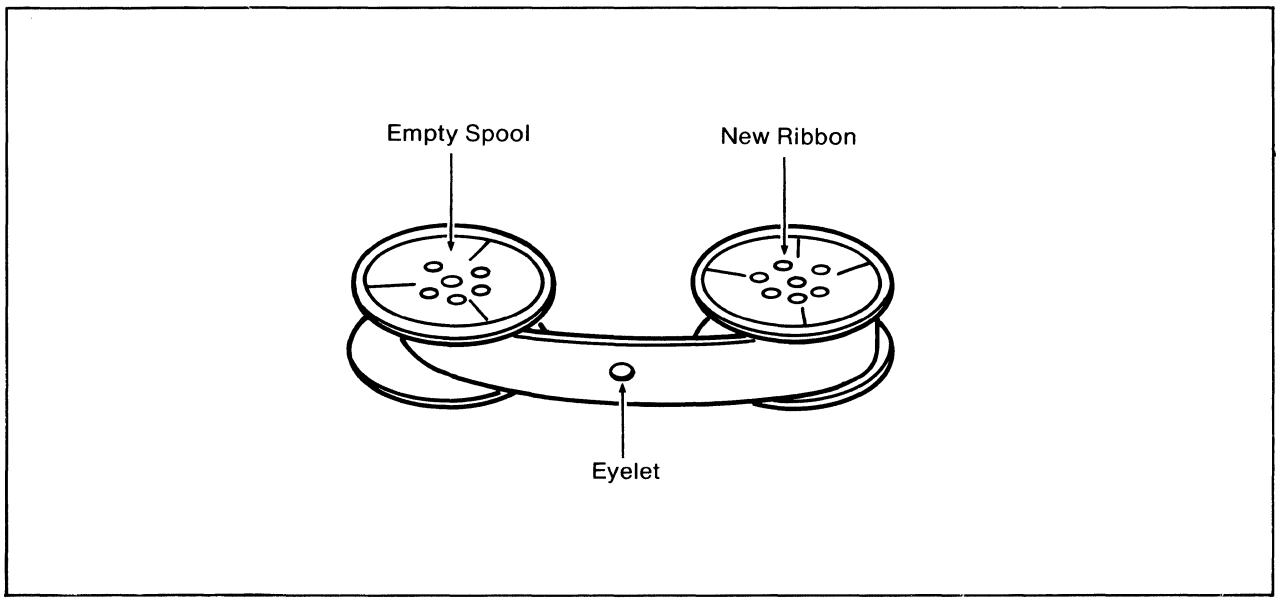


Figure 49 *The Ribbon Spools*

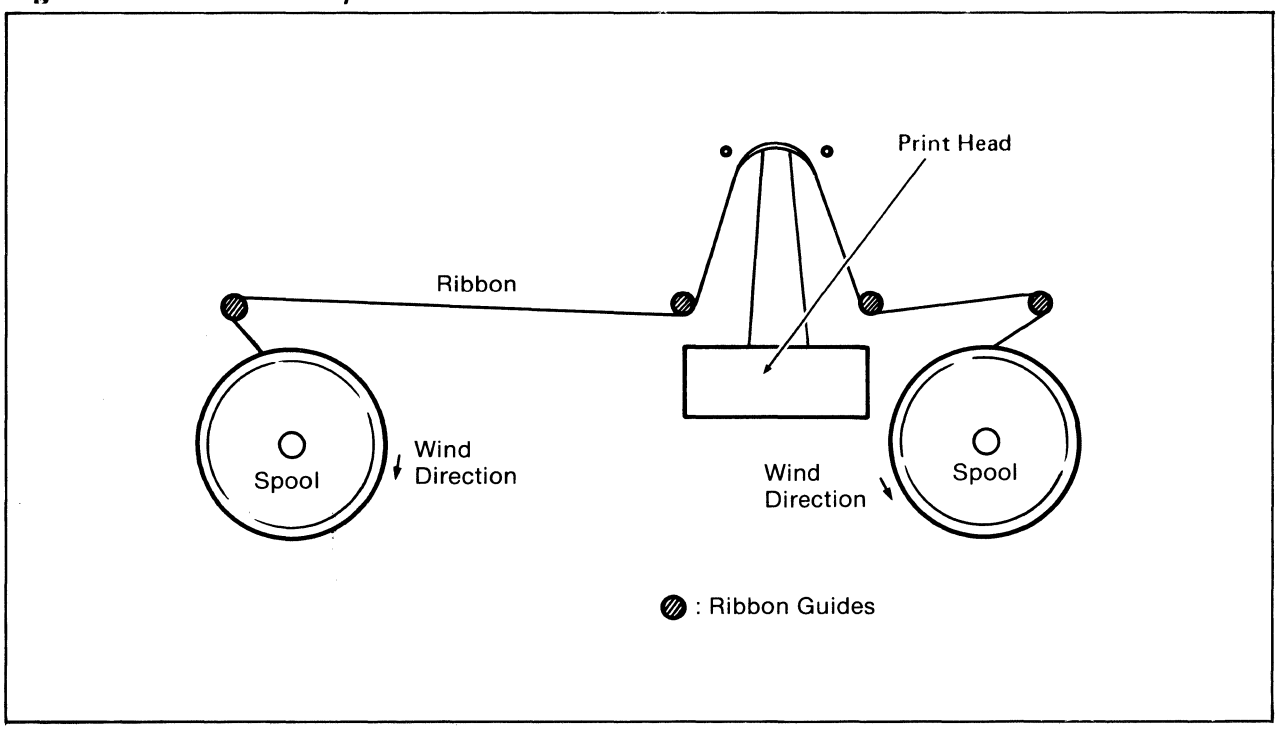


Figure 50 *Loading the Ribbon Spools*

APPENDIX E
COMMUNICATION INTERFACE

The T200/T250 system is equipped with a communication interface which conforms to the EIA RS-232C standard. With this interface, the T200/T250 system can connect a device, such as a modem, which also has the RS-232C interface.

The communication interface consists of the following components:

- An Intel 8251A comparable device that is a universal synchronous/asynchronous receiver/transmitter (USART).
- An 8253 Timer.
- Additional electronic circuits.

To communicate with the external device which is connected to this interface, the user has to prepare communication control program routines, including:

- Writing the transmission rate to the 8253 Timer.
- Writing a mode byte to USART.
- Writing command bytes to USART.
- Transmitting or receiving data through USART.
- Reading the USART status byte.

If the station address is used, the terminal ID can be read from the switch register port.

The following communication features are assumed in subsequent descriptions:

- Asynchronous communication.
- Half-duplex transmission.
- A transmission rate of 110, 150, 300, 600, 1200, 2400 or 4800 bps.

Communication Interface Lines

Data and control lines of the communication interface are as shown in the figure below:

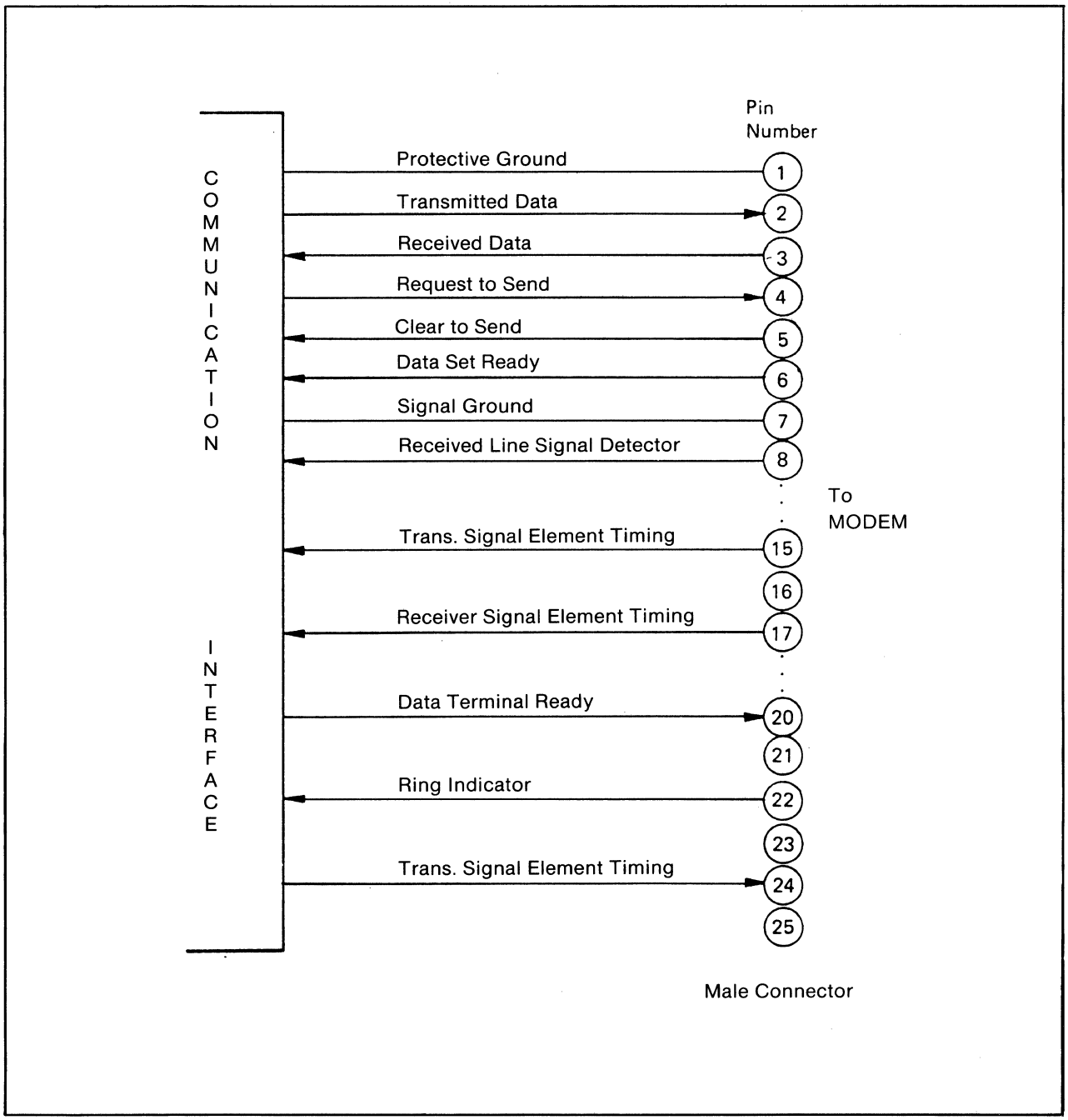


Figure 51 *Communication Interface Signals*

When connected to a modem, the interface lines are used as follows:

Protective Ground	Bonds the chassis ground of the modem to that of the communication interface.
Signal Ground	Provides a common ground reference between the modem and the communication interface.
Transmitted Data	Transfers serial data from the communication interface to the modem.
Received Data	Transfers serial data from the modem to the communication interface.
Request to Send	Requests that the modem prepare itself to transmit (by establishing carrier).
Clear to Send	Indicates that the modem has established a connection with a remote modem and is ready to transmit data.
Data Set Ready	Indicates that the modem is ready to transmit and receive data in the data mode.
Data Terminal Ready	Indicates that the modem is ready to transmit and receive data in the data mode.
Ring Indicator	Indicates that the modem is receiving a ringing signal from the telephone system.
Received Line Signal Detector	Indicates that the modem is receiving a signal which is suitable for demodulation.
Transmitter Signal Element Timing	Provides the signal element timing for the transmitted data.
Receiver Signal Element Timing	Provides the signal element timing for the received data.

The signal level is as follows:

Max Allowable	+25V
Maximum	+15V
Minimum	+3V
−3V	Minimum
−15V	Maximum
−25V	Max allowable

I/O Ports for the Communication Interface

The communication interface is operated by a program which writes and reads control and data bytes on the I/O ports shown in the table below:

I/O Port in Hex	Read/ Write	Function
A0	R/W	8253 Timer Counter #0
A1	R/W	8253 Timer Counter #1
A2	R/W	8253 Timer Counter #2
A3	W	8253 Timer Control
A4	R/W	USART Data
A5	R/W	USART Control Status
A6	R/W	CCM Mode Register
A7	R	CI/CTS Status Register

Communication Interface Functions

Setting the 8253 Timer

If the internal clock is used for communication, the clock rate used must be written to the 8253 ports. For asynchronous communication, write as follows:

Port	Function	Contents Written
A3	Control	"B6" (hex)
A2	Counter	
	110 baud	1135 (Decimal)
	150	832
	300	416
	600	208
	1200	104
	2400	52
	4800	26

NOTE: To write on Port A2, send the least significant byte first, then the most significant byte.

Initializing the USART:

Before starting the data transfer, the USART must be loaded with the control bytes through the Port A5. The control bytes define the USART operations such as:

- Stop bit length
- Parity checking
- Character code length
- Baud rate

There are two types of control bytes:

Mode instruction
Command instruction

The mode instruction is loaded once after resetting the USART (internally or externally). The command instruction is loaded following the mode instruction. Subsequently, the command instructions can be loaded any time during the data transfer.

The mode instruction byte is in the following format:

S2 S1 EP PEN L2 L1 B2 B1

S1S1: Stop bit length
01 ... 1 bit
10 ... 1-1½ bits
11 ... 2 bits

EP: Parity check
0 ... Odd
1 ... Even

PEN: parity enable/disable
0 ... Disable
1 ... Enable

L2: Character length
00 ... 5 bits
01 ... 6
10 ... 7
11 ... 8

B2B1: Baud rate
10 ... x16 (when 8253 Timer is used)
11 ... x64

Example:

For a 7-bit character, even parity check, start/stop bit each 1 bit, write "7A" (hex) to port A5.

The command instruction byte specifies enabling/disabling data transfer, error resetting, modem control and so on. It is in the following format:

EH IR RTS ER SBRK RxE DTR TxEN

EH: Hunt sync character
1 ... Search
0 ... No operation

IR: Internal reset
1 ... Reset USART (to accept mode byte)
0 ... No operation

RTS: Request to send
1 ... Set RTS on (establish carrier)
0 ... No operation

ER: Error reset
1 ... Reset error flags (PE, OE, FE)
0 ... No operation

SBRK: Send break character
1 ... Send break character
0 ... Normal transmission mode

RxE: Receive enable
1 ... Enable
0 ... Disable

DTR: Data terminal ready
1 ... Set DTR on
0 ... No operation

TxEN: Transmit enable
1 ... Enable
0 ... Disable

Example:

To set "Request to Send" on, write "25" (hex) to port A5.

Reading the Status

The USART, when in operation, allows the communication status to be read at any time by the program from port A5. The status byte has the following format:

DSR SYN DET FE OE PE TxE Rx RDY Tx RDY

DSR: Data set ready signal from modem
SYNDET: Sync detected

Error flags

FE: Framing error indicating that a valid stop bit had not been detected.

OE: Overrun error indicating that a character has not been read out before the next character arrives.

PE: Parity error

TxE: Transmitter empty
This indicates whether the USART has a character pending transmission or not.

RxRDY: Receiver reader
This indicates that the USART has a character which has been received.

TxRDY: Transmitter ready
This indicates that the USART can accept a character to transmit.

The program should check the status byte before and after the data transfer. For example, prior to sending a byte to the USART, the status is checked if TxRDY is on. Prior to receiving a byte from the USART, the status is checked if RxRDY is on.
Each data byte is read or written through port A4.

CCM Mode Register

This register is used to set up the clock or to clear the receive buffer.

M2 M1 RATE CHG RxD INH TxD WA

M2M1: 00 ... Asynchronous
01 ... Synchronous (modem clock)
10 ... Synchronous (internal clock)
11 ... Synchronous (direct)
This must conform to the 8253 Timer setting.

RATE CHG: Change modem speed
0 ... Normal
1 ... Half

RxDINH: Inhibit receiving

TxDWA: Send back internally the transmitted data

NOTE: In asynchronous and half-duplex communication, the first data received sometimes results in an error. To avoid this, we recommend that the receive buffer be cleared before data is received, by the following procedures:

Set RxDINH=1, TxDWA=1

Send a dummy data byte

Reset RxDINH=0, TxDWA=0

Send an error reset command

Read a data byte

CI/CTS Status Register

This register provides the additional status information of the interface.

CI CD CTS TxDWA * * *

CI: Call indicator

CD: Carrier detected

CTS: Clear to send

TxDWA: Indicates TxDWA of the CCM mode register

Terminal ID

The terminal ID is obtained by the following steps:

- Write "02" (hex) to the data register port C1 (hex).
- Read from the switch register port D2 (hex).

The terminal ID is given in one byte.

NOTE: You must enable the interrupts before any transmission to the CCM. Upon completion of transmission, disable interrupts.

Summary of Commands

REMOTE COMMANDS	LEAD-IN REQD (X)*	ASCII CODE	DECIMAL
Home cursor	X	DC2	18
Up cursor	X	FF	12
Down cursor	X	VT	11
Left cursor		BS	8
Right cursor		DLE	16
Address cursor	X	DC1,X+96, Y+96**	17,X+96,Y+96**
Read cursor address	X	ENQ	5
Clear screen	X	FS	28
Clear foreground	X	GS	29
Clear to end of line	X	SI	15
Clear to end of screen	X	CAN	24
Delete line	X	DC3	19
Insert line	X	SUB	26

*Lead-in code:

HEX 7E

Decimal 126

**The value 96 should be added only in those cases where the values of X and Y are 30 or less.

APPENDIX F
FLOPPY DISK STORAGE LAYOUT

The floppy disks have the layout shown in the figure below:

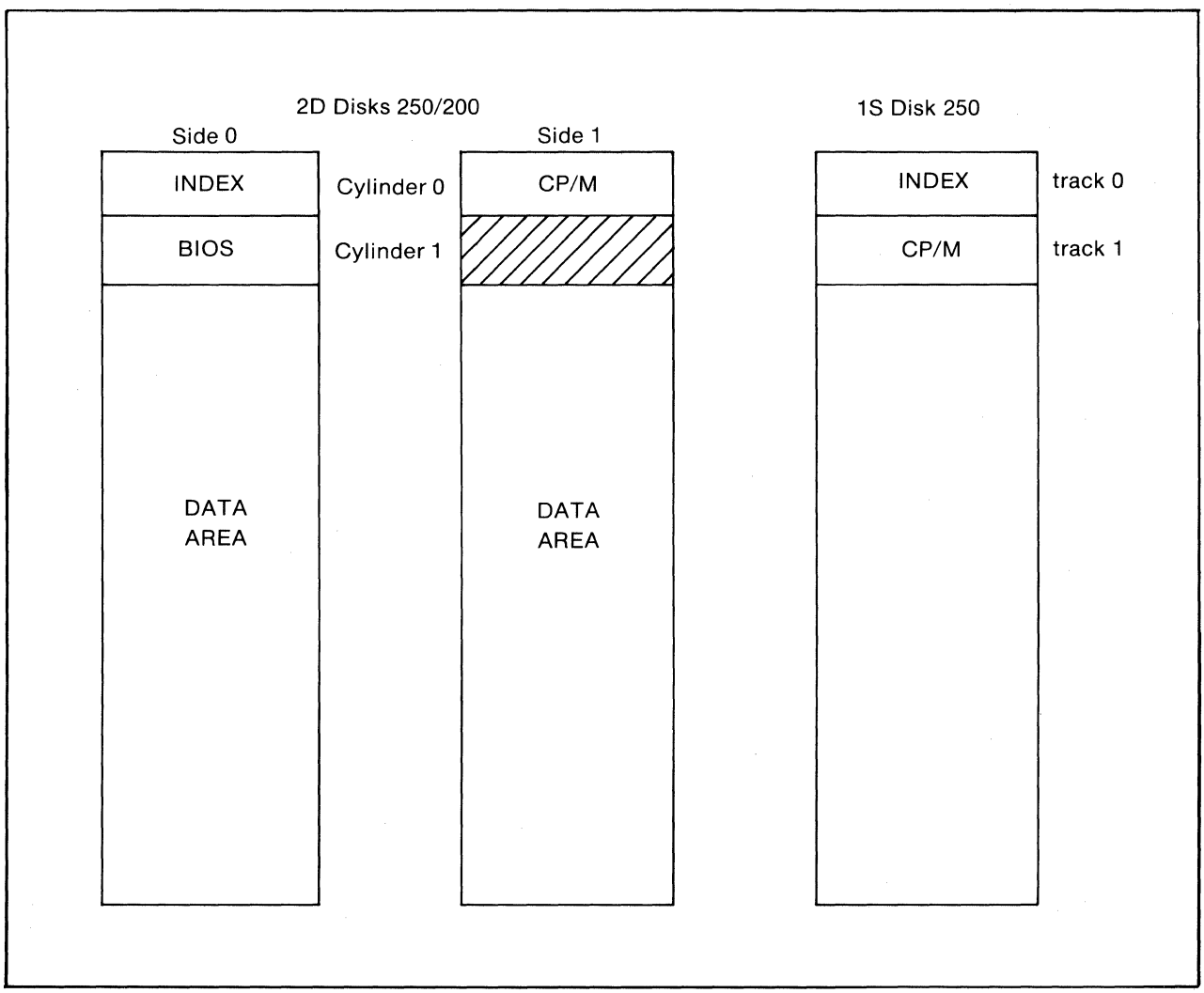


Figure 52 *Disk Storage Layout*

APPENDIX G
PATCHING CP/M

Although CP/M's default parameters suffice for normal operation, some special circumstances may require patching the operating system. The PATCH utility allows modifying the operating system to suit these special circumstances.

To execute the PATCH utility:

— Place the system disk or disk containing PATCH program in drive A.

— Press the BREAK key. CP/M should display the A> prompt.

— Type PATCH and press the ENTER key. The following menu should be displayed:

```
TOSHIBA BIOS Patch Utility
Command?
Patch commands
0    Return to CP/M
1    Change read after write
2    Change cursor shape
3    Change baud rate
4    Change parity, # stop/data bits
5    Change communications command
6    Change patch flag
```

To modify CP/M, enter one of the Patch commands. All entries are 1 or 2 hexadecimal bytes (indicated by the prompt). If you do not want to modify a parameter, just enter RETURN to prompt for data for that command and you will be returned to the main menu for another command.

The following table explains the patch commands and their legal values:

COMMAND	LEGAL VALUES	VALUE FUNCTION
0	—	Return to CP/M
1	0	Default; no read after write.
	1	Read after write; disk I/O is slower. **NOTE:** 1 value causes CP/M to perform write verification of floppy disks to ensure data was correctly written to disk. This slows disk I/O and normally is not required with the reliability of today's disks.
2	49	Default; blinking underscore cursor.
	40	Blinking box shaped cursor.
	69	Slow blinking underscore cursor.
	60	Slow blinking box shaped cursor.
	29	Stationary underscore cursor.
	20	Stationary box shaped cursor. **NOTE:** Other values may produce unpredictable results. If the cursor disappears, reselect option 2 and enter 49 and return in response to prompt. This should return the standard cursor.

COMMAND	LEGAL VALUES	VALUE FUNCTION
3	01A0	Default 300 BAUD
	00D0	600 BAUD
	0068	1200 BAUD
	0034	2400 BAUD
	001A	4800 BAUD
	000D	9600 BAUD **NOTE:** Default is 300 BAUD, no parity, 2 stop bits. Normally, only the baud rate needs to be changed. Other parameters may be set by referring to APPENDIX E, COMMUNICATION INTERFACE.
4	6E	Default. **NOTE:** refer to APPENDIX E, COMMUNICATION INTERFACE, for parameters to change the number of data and stop bits for each character, and to select odd or even parity.
5	B6	Default. **NOTE:** There is normally no need for the default value.
6	0	Sets drive B back to normal status.
	1	Sets drive B to patched status. **NOTE:** This option allows changing the disk patched flag. It is normally used after using the utility DIR128 to force disk B to use 128 directory entries that the old disk used. If the option is set to 0, CP/M uses the disk normally. However, if the option is set to 1 (which is done by DIR128) the disk1 is forced into just one format, determined by the last disk placed in the drive.

If you receive the error message:

Function must be 0-6

it means you entered an illegal PATCH command. The system responds to the 0 command with:

TOSHIBA BIOS Patch Utility Operation Completed
A>

INDEX

Toshiba America, Inc. ("TAI")
Information Systems Division
2441 Michelle Drive
Tustin, California 92680

LIMITED 90 DAY WARRANTY FOR TAI INFORMATION PROCESSING EQUIPMENT

This equipment and any word processing software which may be contained in the same package as this equipment (collectively, the "Equipment") is warranted free from defects in workmanship and material, software errors and non-conformity with TAI's Standard Performance Specifications included in the package in which the Equipment is shipped, for a period of 90 days after installation. Installation shall be deemed to have occurred not later than ten (10) days following the date of purchase. Any Equipment which fails to meet this express warranty during the relevant warranty period will be repaired or replaced free of charge as necessary to cure such failure. Carry-in service can be obtained during the warranty period by bringing or mailing your unit to any authorized TAI service center. To locate the authorized TAI service center nearest you, refer to the authorized TAI service center directory enclosed or write to the TAI Warranty Division at the address specified above. A receipt of purchase or other proof of the date of purchase or installation of the Equipment will be required before any warranty service will be performed.

THIS WARRANTY SHALL BE EFFECTIVE ONLY IF THE WARRANTY REGISTRATION CARD BELOW IS COMPLETED, SIGNED AND RETURNED TO TAI BY MAIL NOT LATER THAN TEN (10) DAYS FOLLOWING INSTALLATION OF THE EQUIPMENT COVERED, AND RETURN OF THE COMPLETED WARRANTY REGISTRATION CARD TO TAI WITHIN SUCH PERIOD IS A PRE-CONDITION FOR WARRANTY COVERAGE AND FOR WARRANTY SERVICES TO BE PERFORMED.

This warranty applies only to Equipment which is new and unopened on the date of purchase and which is located within the United States, Canada, Puerto Rico or Mexico during the entire relevant warranty period. This warranty is contingent upon normal and proper use of the Equipment and does not cover damage which occurs in shipment or damage or failure resulting in whole or in part from alteration, unusual physical or electrical stress, misuse, failure to follow the most current instructions promulgated by TAI with respect to proper use of the equipment, abuse, neglect, fire, accident, flood, act of God, improper installation or improper maintenance, or any defect or error in any hardware, peripheral device or software other than the Equipment covered by this warranty. This warranty does not cover Equipment on which the original identification marks or serial numbers have been removed or altered.

TAI'S OBLIGATION TO REPAIR OR REPLACE ANY EQUIPMENT WHICH FAILS TO MEET THE EXPRESS WARRANTY SET FORTH ABOVE SHALL BE THE SOLE AND EXCLUSIVE REMEDY FOR A BREACH OF SUCH WARRANTY. THE ABOVE EXPRESS WARRANTY IS THE SOLE WARRANTY MADE BY TAI WITH RESPECT TO THE EQUIPMENT AND IS IN LIEU OF ALL OTHER WARRANTIES BY TAI, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, EXCEPT AS TO CONSUMER GOODS IN WHICH CASE THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY FOR THE PERIOD OF THE EXPRESS WARRANTY. THE EQUIPMENT COVERED BY THIS WARRANTY IS MARKETED AND SOLD BY TAI AS A BUSINESS RATHER THAN CONSUMER PRODUCT AND IS NOT INTENDED FOR A PERSONAL, FAMILY OR HOUSEHOLD USE.

UNDER NO CIRCUMSTANCES WILL TAI BE LIABLE FOR ANY CONSEQUENTIAL, INCIDENTAL, SPECIAL OR EXEMPLARY DAMAGES ARISING OUT OF OR CONNECTED WITH THE DELIVERY, SALE, USE OR PERFORMANCE OF THE EQUIPMENT, EVEN IF TAI IS APPRISED OF THE LIKELIHOOD OF SUCH DAMAGES OCCURRING. SOME STATES DO NOT ALLOW FOR THE EXCLUSION OF CONSEQUENTIAL DAMAGES OR THE LIMITATION OF IMPLIED WARRANTIES WITH RESPECT TO CONSUMER PRODUCTS SO THE ABOVE EXCLUSION MAY NOT BE APPLICABLE IF THE COVERED EQUIPMENT CONSTITUTES A CONSUMER PRODUCT UNDER APPLICABLE LAW.

IN NO EVENT SHALL TAI'S LIABILITY (WHETHER IN CONTRACT, TORT OR OTHERWISE) FOR DAMAGES ARISING OUT OF OR RELATED TO A BREACH OF THE ABOVE EXPRESS WARRANTY OR THE SALE, DELIVERY, USE OR PERFORMANCE OF THE EQUIPMENT EXCEED THE PURCHASE PRICE OF THE EQUIPMENT. SUCH LIMITATION OF LIABILITY SHALL, WITHOUT LIMITATION, BE APPLICABLE IN THE EVENT THAT THE SOLE REMEDY OF REPAIR OR REPLACEMENT FOR A BREACH OF THE ABOVE EXPRESS WARRANTY FAILS OF ITS ESSENTIAL PURPOSE OR OTHERWISE IS UNENFORCEABLE.

This warranty gives you specific legal rights and you may also have other rights which may vary from state to state.

Any word processing software included in the Equipment is licensed for use on TAI word processing equipment pursuant to TAI's End-User Word Processing Software License Agreement and is not sold.

If a problem with this Equipment develops during the warranty period, first contact the dealer from which you purchased it or an authorized TAI service center. If the problem is not handled to your satisfaction, then write to the TAI Warranty Division at the company address indicated above.

RETAIN THIS PORTION FOR YOUR RECORDS