Ballantine/31597/$9.95 in USA • $12.95 in Canada

IBM PCjr®
User's Handbook

The only manual you'll ever need—
from the moment you plug in your computer through set-up, operation, maintenance, and programming

Weber Systems, Inc. Staff

IBM PCjr®
User's Handbook

IBM PCjr®
User's Handbook

by
Weber Systems Inc. Staff

Ballantine Books • New York

This book is available to organizations for special use.
For further information, direct your inquiries to:
Ballantine Books
Special Sales Department
201 East 50th Street
New York, New York 10022

IBM PC*jr*™ User's Handbook

Library of Congress Catalog Card Number: 83-91222
ISBN: 0-345-31597-9

Manufactured in the United States of America

First Ballantine Books Edition: May 1984
10 9 8 7 6 5 4 3 2 1

Contents

Introduction & Acknowledgements

IBM PC*jr*® User's Handbook is meant to serve as a tutorial as well as an on-going reference guide to the operation and programming of the IBM PC*jr*. All of PC*jr*'s important features are discussed including:

- Keyboard usage
- BASIC programming
- DOS usage
- Logo programming
- Communications
- Graphics
- File handling

A number of examples are included with the text to illustrate the topics being discussed. Terms that may be unfamiliar to the reader will be presented in bold in the text. These terms will be defined in subsequent paragraphs.

Chapter 1 of this book is meant to serve as an introduction to PC*jr* and its related peripherals. Both the entry and the enhanced models are discussed. Topics covered include the system unit, keyboard unit, optional disk drive, 8088 CPU, DOS 2.1, operating system, Microsoft BASIC 2.1, and peripherals such as printers, joysticks, RAM expansion, and modems.

Chapter 2 details the procedures provided in initially installing the PC*jr* as well as its start-up. Keyboard usage and the available peripheral ports are also discussed in chapter 2.

Chapter 3 is meant to serve as an introduction to programming the PC*jr* in Microsoft BASIC version 2.1. The following topics are discussed:

- BASIC start-up
- Program entry
- Listing a program
- Editing a program
- Running a program
- Saving and loading a program

Chapter 4 discusses data types, operators, and variables in the context of BASIC programming. Chapter 5 includes additional fundamental BASIC programming concepts including:

- Input and output
- Tables and arrays
- Functions
- String handling
- Program concatenation

In chapter 6, the use of files in Microsoft BASIC is discussed. Chapter 7 describes techniques for outputting graphics using BASIC commands.

Chapter 8 includes a discussion of the Logo language. Logo is used frequently in educational environments, and is expected to be widely implemented on the PC*jr*.

Chapter 9 consists of a detailed discussion of DOS 2.1 usage on the PC*jr*. Topics covered include:

- DOS start-up
- DOS keyboard usage
- Copying diskettes
- Copying files
- Formatting diskettes
- Backing-up diskettes
- Listing the diskette directory
- Renaming files
- Erasing files
- Analyzing the diskette
- Copying the operating system
- Batch files
- EDLIN
- SORT
- FIND
- MORE

Chapter 10 contains a discussion of using the PC*jr* as a terminal to communicate with other computers using telephone lines as the communications link. Topics covered include:

- Internal modem
- Terminal Emulator program
- Practical example of PC*jr* communications

A number of appendices are included with IBM PC*jr* User's Handbook. These detail the PC*jr* ASCII code set, BASIC reserved words, BASIC error messages, DOS commands, and printer usage with the PC*jr*.

We gratefully acknowledge Rosemary Morrissey and Jeanette Mahrer of IBM Corporation for their assistance.

We also wish to thank Ron Hall and Jeff Jones of the HCS Computer Shoppes for their cooperation and assistance.

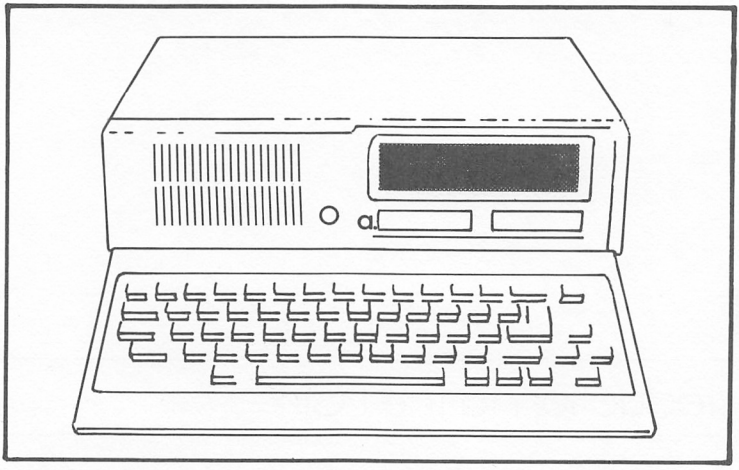
1

Introduction to the PCjr

Overview

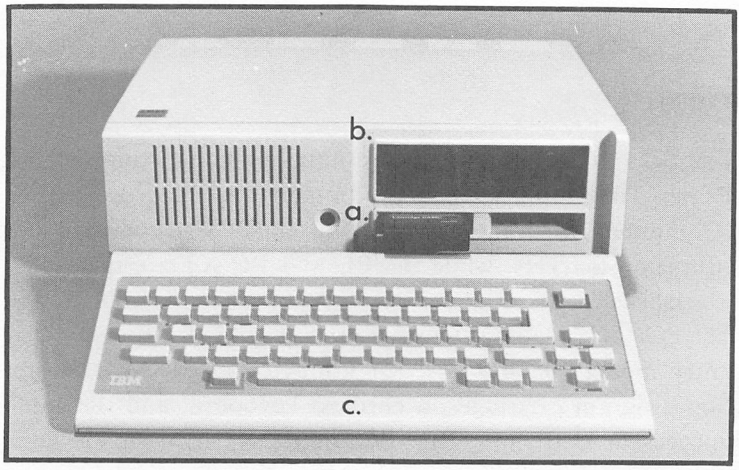
The IBM PC*jr* is expected to reshape the home computer market. The PC*jr* is IBM's follow-up to the tremendously successful IBM Personal Computer. Apparently, IBM intends the PC*jr* to appeal to the home and education markets, while the PC and PC XT are targeted at the business market.

The PC*jr* is available as two separate models (see figures 1-1 and 1-2). The entry model includes 64K (or **kilobytes**) of RAM, two slots for inserting program cartridges, a cordless keyboard, and the Intel 8088 **microprocessor** used on the IBM PC and PC XT models. The enhanced model contains an additional 64K Memory and Display Expansion board, as well as a slim-line 5¼ inch floppy diskette drive which is capable of storing 360K. The Memory and Display Expansion board increases user RAM to 128K and allows 80 characters to be displayed on each line of text on the screen.



a. cartridge slots

FIGURE 1-1. IBM PC*jr* (Entry Model)



a. cartridge slots b. diskette drive c. keyboard

FIGURE 1-2. IBM PC*jr* (Enhanced Model)

The IBM PC*jr* consists of three basic units; the **keyboard, System Unit,** and **power supply/transformer** (see figures 1-3 to 1-5). The System Unit contains the heart of the PC*jr*, the Intel 8088 microprocessor, 64 or 128K of RAM, the cartridge slots, and the Microsoft BASIC interpreter in **ROM** (read-only memory).

The keyboard allows the PC*jr* user to communicate with the System Unit. Unlike the PC and PC XT models, the PC*jr* keyboard does not necessarily have to be connected to the System Unit with a cable. The PC*jr* keyboard is cordless. Communications with the System Unit is accomplished via an infrared optical signal.

If more than one PC*jr* is being used concurrently in a room, it is necessary to connect the keyboard to the System Unit using the optional keyboard connection cable. Otherwise, the keyboard can be used without the cable to communicate with the System Unit. The advantage of cordless communications is especially useful in a classroom situation -- as the System Unit can be placed in a central location and the keyboard can be passed around the classroom. The effective range of cordless keyboard communications is approximately 20 feet.

The PC*jr* keyboard is powered by four AA batteries. It contains 62 **programmable*** keys arranged in a typewriter-like sequence.

A display must be added to your PC*jr* in order for it to be a useful device. The PC*jr* can utilize either a television set, the IBM Color Display, or a standard video monitor as a display device. Since the PC*jr* offers built-in color graphics, a color display device offers the greatest advantage.

Unlike the IBM PC and PC XT models, the PC*jr*'s power supply is not built into the System Unit. Rather, the power supply/transformer (see figure 1-5) is housed in a separate unit which is attached to the System Unit by a power cord.

---

* Programmable keys allow any letter, number, function, or command to be assigned.

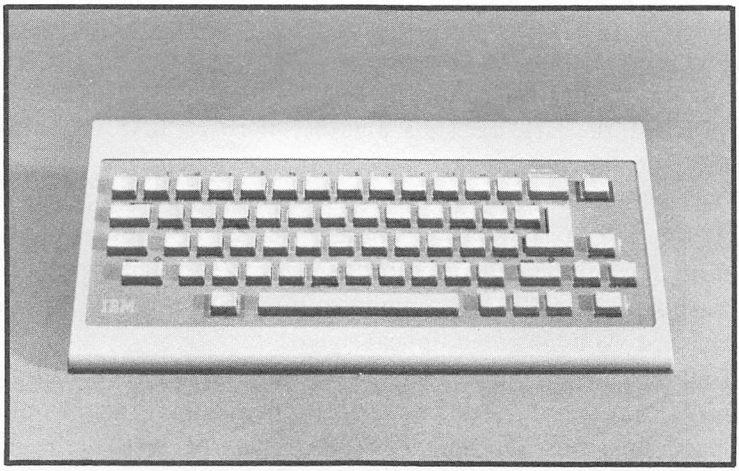


FIGURE 1-3. IBM PC*jr* Keyboard Unit.

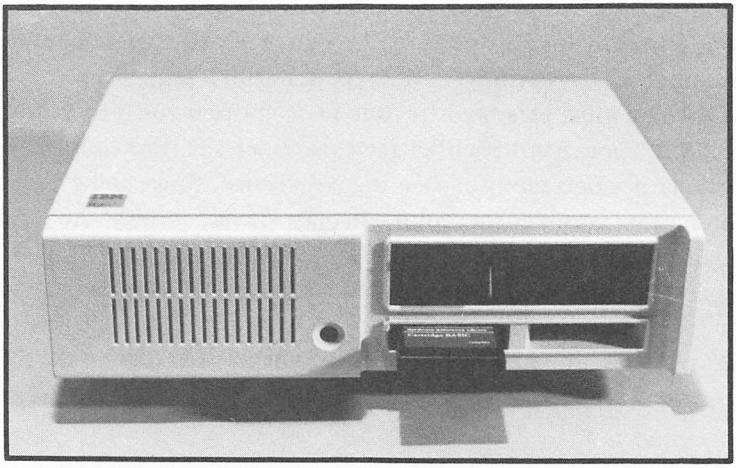


FIGURE 1-4. IBM PC*jr* System Unit.

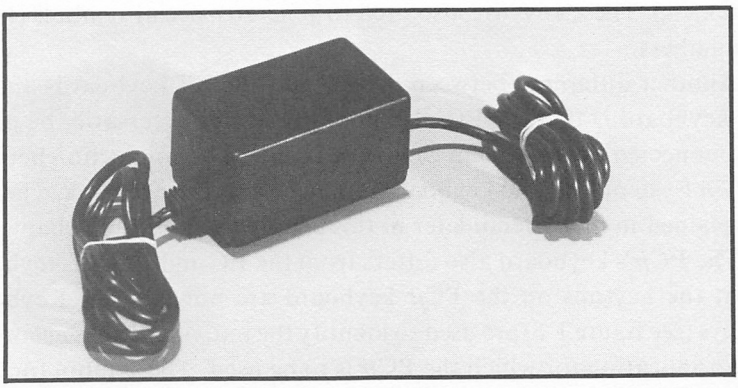


FIGURE 1-5. IBM PC*jr* Power Transformer

DIFFERENCES BETWEEN THE IBM PC and PC XT AND THE PC*jr*

Since many of our readers are likely to already be familiar with IBM's PC and PC XT models, it may be helpful to begin our discussion of the PC*jr* with a summary of the differences between it and the PC and PC XT.

First of all, while the PC comes with 128K of RAM on the System Board and the PC XT includes 256K of RAM, the PC*jr* includes only 64K of RAM. Expansion up to 128K is available on the PC*jr* by incorporating the optional Memory and Expansion board. The PC and PC XT models allow for RAM Expansion up to 896K.

The keyboard on the PC and PC XT models includes 83 separate keys, while the PC*jr* includes only 62 keys. Although the PC*jr*'s keyboard contains fewer keys than the PC and PC XT keyboards, nearly all of the characters available on these two can be generated by the PC*jr*'s keyboard. This is accomplished by having a number of keys serve a dual purpose. For example, the function keys are now located along the top of the keyboard rather than along the left hand side (as on the PC and PC

XT models). These keys are also used to generate special symbols (! @ #) and numbers.

Another difference between the PC and PC XT keyboards and the PC*jr* keyboard is that the PC*jr* keyboard need not necessarily be physically connected to the System Unit with a cable. The connection between the PC*jr* System Unit and keyboard is made via an infrared link. This will be explained in more detail later in this chapter as well as in chapter 2.

The PC*jr*'s keyboard also differs from the PC and PC XT keyboard in that the keytops on the PC*jr* keyboard are not labeled. Keyboard overlays (see figure 1-6) are used to identify the individual keys according to the application for which the PC*jr* is to be used. The symbol for each key is printed on the overlay above it.

Another difference between the PC*jr* and PC and PC XT models is the availability of disk storage. The PC*jr* allows for the addition of one optional floppy disk drive. The PC allows the addition of two drives. Fixed disk storage can also be added to the PC by adding an Expansion Unit. The PC XT includes one floppy disk drive as well as a 10MB* fixed drive as part of its standard configuration.

The PC and PC XT will support the IBM Graphics, Compact, or Color Printers. The PC*jr* will only support the Graphics and Compact Printers.

Finally, the 8087 comprocessor which is available for the IBM PC and PC XT is not available for the PC*jr*.

Technical Data

As was mentioned in the preceding section, the System Unit contains the fundamental components of the PC*jr*. A system board or mother-

* MB is an abbreviation for megabyte (1 million bytes).

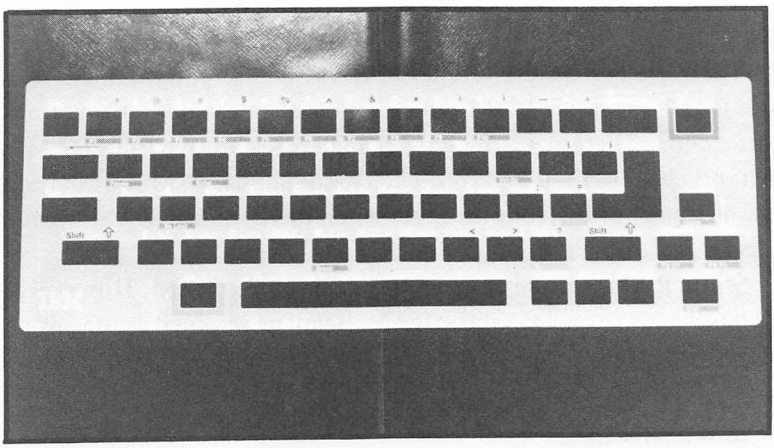


FIGURE 1-6. IBM PC*jr* Keyboard Overlay

board is housed within the System Unit. Most of the circuitry for the PC*jr* is located on the system board including the following:

- 8088 microprocessor
- 64K RAM
- 64K ROM
- RS-232 Interface
- Speaker
- Connectors for optional devices

ROM and RAM

ROM stands for Read Only Memory. ROM will hold the data stored in it permanently. If the power to the PC*jr* is shut off, the information stored in ROM will remain there. As previously mentioned, the Microsoft BASIC language interpreter is stored in ROM.

RAM stands for Random Access Memory*. Any data stored in RAM will be lost when the PC*jr*'s power is shut off. When data is loaded from a tape cassette, a disk drive, or the keyboard, it is stored in RAM.

---

* Random Access Memory is a somewhat misleading term to describe RAM, as most memory (including ROM) is randomly accessed.

DYNAMIC AND STATIC RAM

There are two different types of RAM memory; **dynamic RAM** and **static RAM.** Dynamic RAM can only hold the data it is storing for a few milliseconds. Therefore, any data being stored in dynamic RAM must constantly be rewritten or refreshed. This dynamic RAM refresh function must be a part of the support logic when the dynamic RAM memory is designed.

Static RAM is more expensive than dynamic RAM. However, once data has been written into static RAM, it will be retained as long as power is supplied.

IBM PC*jr*'s CPU

The central processing unit or **CPU** is the heart of any computer. The CPU controls all of the other components for the computer.

In larger computers, the CPU and the **ALU** (arithmetic logic unit) consist of a group of IC chips each dedicated to its own task. In smaller computers, the CPU and ALU are generally combined on a single chip which is known as a **microprocessor.**

A microprocessor can be defined as a single chip which contains the logic of a central processing unit as well as any additional logic that must complement the CPU.

The microprocessor used in the PC*jr* is the Intel 8088, which is manufactured by Intel Corporation of Sunnyvale, California. Intel is a pioneer in the development of microprocessors. Intel developed the predecessor of the 8088, the 4004 calculator chip, in the early 1970's. Intel subsequently developed the 8008, 8080, 8085, and 8088/8086 microprocessors.

Microprocessor logic is based upon the **bit.** A bit is the basis of all information storage within the computer. A bit is a simple switch that can consist of either of the two binary states, on or off.

Bits are often separated into groups of eight. These groups of 8 bits are known as a **byte.** A byte is required to represent a single character (i.e. letter, number, or symbol). Generally, bytes are processed by the computer in groups of 2.

Most 8-bit microprocessors can only **address** (or work directly with)

65,536 (64K) bytes at any one time. Even though this number appears large, a 30 page document would fill this memory area.

Most 16-bit microprocessors can address from 65,536 to 16 million bytes of memory. Moreover, 16 bit microprocessors process data at a speed from 2 to 10 times faster than 8-bit microprocessors.

One of the main advantages the IBM PC*jr* has over other home computers is the power provided by its use of the 8088 CPU.

The 8088 can address up to one million bytes of memory (or 1 megabyte, abbreviated as 1MB). The 8088 has 20 different address lines, which allows 2 to exponent 20 (2^{20}) different address combinations. This is the equivalent of 1,048,576 different addresses.

The speed of the 8088 is about .65 mps (or million operations per second). In other words, the 8088 will process 650,000 arithmetic operations or data transfers per second. When compared with the Intel 8080A, the 8088 is approximately six times faster. The characteristics of the 8088 are outlined in table 1-1.

TABLE 1-1. Intel 8088 Characteristics

	Intel 8088
Construction:	Chip technology -- HMOS 40 pin plastic package.
Manufacturer:	Intel Corporation
Introduction Date:	1979
Features:	1. 20 address lines 2. 1 MB memory addressing 3. 650,000 operations per second 4. 8 data lines 5. 4.77 Mhz clock speed 6. Separate bus interface and execution units. 7. 99 basic machine language instructions. 8. Software compatible with Intel 8086.

Software

Software can be defined as the set of instructions or programs that cause the computer to operate. Software can be divided among three general classifications:

- Operating System Software
- Language Software
- Applications Software

Each of these classes of software will be defined and discussed briefly in the context of the PC*jr* in the following sections. PC*jr* software can be stored on cassette tape, floppy diskettes, or cartridges.

OPERATING SYSTEM SOFTWARE

An operating system can be defined as a group of programs which manage the overall operation of the computer. The operating system performs system operations such as controlling data input/output, memory assignments etc. The PC*jr*'s operating system is stored permanently in ROM.

The standard operating system available for the PC*jr* is DOS version 2.1. DOS 2.1 was developed by Microsoft Corporation and is a revised version of the earlier versions of Microsoft's DOS used on the IBM PC (versions 1.0, 1.1, and 2.0). DOS version 2.1 can also be used on the IBM PC and PC XT. DOS version 2.1 usage on the PC*jr* is discussed in chapter 9.

LANGUAGE SOFTWARE

A **language** can be defined as a group of characters and/or symbols which can be combined using a set of syntax rules to represent information. Examples of languages include English, Spanish, French, as well as computer programming languages such as BASIC, ADA, PASCAL, and COBOL.

The BASIC language is supplied with the PC*jr*. The version of BASIC used is that developed by the Microsoft Corporation -- Microsoft BASIC version C1.20. Microsoft BASIC is located in memory on the PC*jr*'s system board. Microsoft BASIC is also supplied with the IBM PC

and PC XT computers. Microsoft BASIC version 2.1 is the latest version used on IBM personal computers. Earlier versions includes versions 1.0, 1.1, and 2.0.

Computer languages are often distinguished as being either **compiled** or **interpreted** languages. Microsoft BASIC is an interpreted language.

A compiled language program consists of the **source code** and the **compiled code.** The source code consists of the program statements in their original form. For example, the following is a line of source code from a program written in the CBASIC compiled language.

```
100 INPUT "ENTER TODAY'S DATE:";DATE.1
```

The source code is processed by a program known as a **compiler** into the compiled code. The compiled code is very similar to the machine language used by the microprocessor. The compiled code is the code actually used when a compiled program is run. A program known as a **run-time monitor** is used to run the compiled program.

An interpreted language consists of only the source code. The source code is translated line-by-line directly into machine language instructions. The Microsoft BASIC language that is standard on the IBM PC is an interpreted language.

One advantage of interpreted languages over compiled languages is that interpreted language programs are more easily developed. When working with interpreted languages, a programmer need only write a program, enter it, run it, and alter it at his leisure. When working with a compiled language, the source code must be recompiled every time it is edited. This can be frustrating during the program debugging process.

One advantage of compiled languages over interpreted languages is that execution time is much faster. The compiled code is much closer to the machine language than the source code. Since interpretation is not necessary, execution of compiled code is much faster.

The IBM PC and PC XT use three separate versions of BASIC; Cassette BASIC, Disk BASIC, and Advanced BASIC.

Cassette BASIC is the version of BASIC used when a cassette recorder is being used to store information. Cassette BASIC cannot be used when data is being stored on diskettes.

Disk BASIC contains all of the capabilities of Cassette BASIC and can also be used when data is being stored on diskettes. In other words, Disk BASIC can be used when data is being stored on diskettes or when it is being stored on cassette tape.

Advanced BASIC has all of the capabilities of Cassette BASIC and Disk BASIC as well as additional features not found on these other two versions of BASIC, Advanced BASIC includes several graphics commands such as CIRCLE, DRAW, and PAINT that allow the user to fully utilize the color graphics capabilities of the IBM PC when it is equipped with the Color/Graphics Monitor Adapter and a color monitor or color TV set.

Both the Advanced BASIC and the Disk BASIC versions require DOS before they can be run, or before any programs written in Advanced or Disk BASIC can be run.

The PC*jr* uses the same Cassette BASIC used by the PC and PC XT. The Cassette BASIC portion of the interpreter is resident in 32K of ROM. This is also the case with the PC and PC XT.

For the PC*jr*, the features of Disk and Advanced BASIC have been combined (along with additional new features) into Cartridge BASIC. Cartridge BASIC is called from a ROM cartridge which must be inserted in one of the PC*jr*'s two available cartridge slots. Cartridge BASIC has a length of approximately 32K, and requires 6K of RAM for operation.

As we mentioned in the last paragraph, Cartridge BASIC incorporates a number of new features not found in previous versions of BASIC available for the PC and PC XT. The majority of these features consist of commands which allow for more effective usage of the PC*jr*'s color graphics options and generation of sound through an extended feature. A few of the more useful commands included in Cartridge BASIC are described briefly below.

NOISE

This command allows the programmer to generate complex sound effects through an external speaker connected to the PC*jr*.

PALETTE

This command allows the programmer to select the palette in use.

PALETTE USING

This command allows the programmer a means of efficiently setting all palette entries.

PCOPY

This command allows the programmer to copy one page of screen memory to another page.

TERM

The TERM command causes a Terminal Emulator program, which is present in the ROM of the BASIC cartridge, to be loaded into the PC*jr*'s RAM. This Terminal Emulator program can be used to allow the PC*jr* to communicate with a host computer. The Terminal Emulator program will be discussed in detail in chapter 10.

Cartridge BASIC can be used either when DOS 2.1 has been loaded from a diskette into RAM or when DOS has not been loaded. If DOS has not been loaded, any disk related BASIC commands will not be functional.

BASIC programming on the PC*jr* will be discussed in more detail in chapters 3 through 7.

Since the PC*jr* has a great deal of appeal in educational settings, another language, Logo, is also widely used. Logo is a language which begins by teaching computer fundamentals to grade school age children and progressively exposes the reader to more advanced concepts such as mathematical and logical operators, file handling, graphics, and assembly language subroutines. The use of Logo on the PC*jr* will be discussed in more detail in chapter 8.

APPLICATIONS SOFTWARE

Applications software can be defined as a set of instructions designed to accomplish a specific task that is of some value to the user. Examples of applications programs include games, word processing programs, spreadsheets, and database software. Generally, applications programs are stored on cassette or diskette and are transferred into RAM, where the program is available to the computer. Applications programs can also be stored in a permanent form on a ROM cartridge. This ROM cartridge can be plugged into one of the PC*jr*'s cartridge slots.

A large variety of applications software is available for use with the PC*jr*. These include programs which can be used in the home such as Home Budget; programs which can be used at work such as Multiplan and Visicalc; programs with educational applications such as Monster Math and Turtle Power, and finally games such as Adventure.

PC*jr* CARTRIDGES

As was mentioned in the preceding section, cartridges are often used to store PC*jr* programs including the Cartridge BASIC interpreter. A PC*jr* ROM cartridge is pictured in figure 1-7. The cartridge consists of 32K of ROM enclosed in a plastic case.

The ROM cartridge should generally be inserted into the PC*jr*'s left cartridge slot (see figure 1-8).

Peripheral and Add-On Devices

A **peripheral** can be defined as an auxiliary device which can be connected to a computer to perform some additional function.

A number of peripherals and add-on devices can be added to the PC*jr* to expand it into a total computer system. These include a cassette player/recorder, a diskette drive, printers, joysticks, a modem, and additional RAM. A number of these peripheral components will be described in the following sections.

FIGURE 1-7. PC*jr* Cartridge

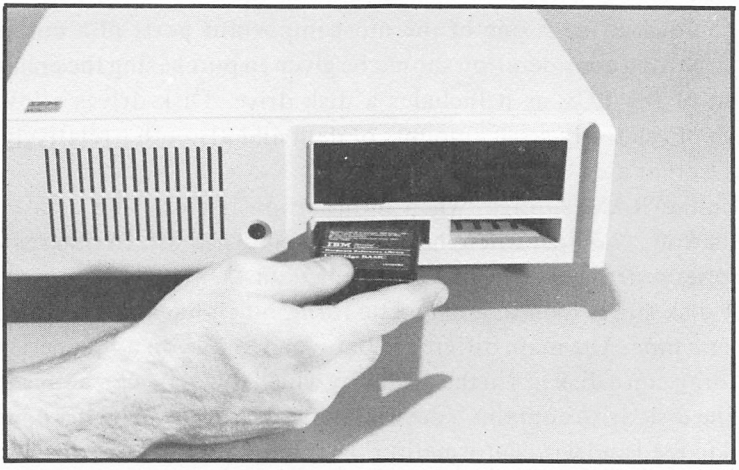


FIGURE 1-8. Inserting a PC*jr* Cartridge

Disk Drives

The optional disk drive for the PC*jr* is a **soft sectored, double density,** and **double sided** drive. The floppy disk drive uses 40 tracks per diskette side. Each track has nine **sectors*** of 512 bytes. This gives a total of 180K of data storage per side or 360K of storage for the drive.

IBM refers to the disk drive used with the PC*jr* as a slimline drive. This drive is approximately half the height of the standard size disk drives used with the IBM PC. However, the slimline drive uses the same 5¼ inch diskettes as do the standard size drives.

The PC*jr* slimline diskette drive is manufactured by Qume Corporation of San Jose, California. Qume is a subsidiary of IT&T. The majority of diskette drives used in the IBM PC were manufactured by Tandon Corporation of Chatsworth, California.

As was mentioned previously, a diskette drive is standard with the PC*jr* enhanced model. A drive can, however, be added to the basic PC*jr* model. IBM provides comprehensive installation instructions with the diskette drive for users who wish to add this option to their PC*jr*.

The disk drive is one of the most important parts of a computer system. Strong consideration should be given to purchasing the enhanced version of the PC*jr* as it includes a disk drive. Disk drives allow the storage of relatively large amounts of data and also offer relatively fast access to that data.

Unlike RAM storage, when information is stored on a disk, the information is not lost when the computer is turned off. In other words, disks offer a permanent means of storing data.

A disk stores data in a magnetic form, much like data is stored on magnetic tape. The main difference between storage on a magnetic tape and storage on a disk lies in the means by which that data can be accessed.

The disk drive contains a device known as a **read/write head,** which is used to read and write information. The computer can move the head to any position desired on the disk surface. This is in contrast to magnetic tape, where data is read from or written onto the tape in consecutive order.

---

* A definition of sectors is provided on page 28.

This capacity to read or write data at a particular position is known as **random access.** Disk drives are known as random access storage devices. On the other hand, in cases where data must be read or written in a consecutive order, the accessing is known as **sequential access.** A cassette tape recorder is known as a sequential access device.

FLOPPY DISKETTES

Disk drives store data on floppy **diskettes.** A floppy diskette consists of a round vinyl disk which is enclosed within a plastic cover. The diskette is generally stored in a diskette envelope.

This cover protects the diskette from damage while it is being handled by the operator. The diskette should never be removed from its cover. A 5¼ inch diskette with its protective envelope is shown in figure 1-9.

The diskette is allowed to rotate within the protective envelope. The round hole in the middle of the diskette allows the disk drive to hold the diskette and spin it. The oblong shaped opening on the protective envelope provides an area where the head can read from or write to the diskette surface.

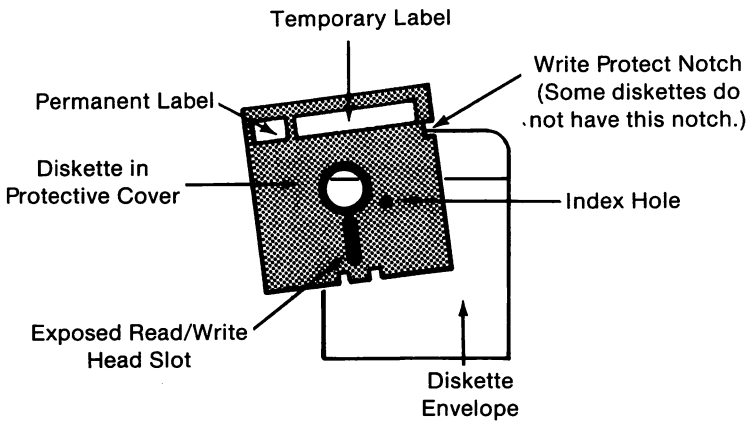


FIGURE 1-9. 5¼ Inch Floppy Diskette

TRACKS & SECTORS

To facilitate the process of searching for data on the diskette surface, that surface is divided into tracks and sectors. Tracks may be visualized as a series of concentric circles on the diskette surface, as shown in figure 1-10. DOS 2.1 divides each side of a diskette into 40 tracks.

To further reduce the time necessary to search for a particular data item, DOS 2.1 divides each track into 9 sectors, which are also shown in figure 1-10.

Each individual sector holds 512 bytes of data. When DOS has access to the track and sector where a particular data item is being stored, it will only have to search 512 bytes to find that item. The result of dividing the diskette surface into tracks and sectors is that access time is greatly decreased.

HARD AND SOFT SECTORS

Locating a particular track on the disk surface is a relatively uncomplicated matter. The drive merely moves the head to the position on the diskette where the specified track is located, much like the needle on a phonograph is positioned to the location of a specific song on a record album.

However, locating a particular sector is a more difficult process. Two different methods are used to locate sectors on a disk; hard sectoring and soft sectoring.

Both the hard and soft sector methods involve the use of an index hole. The index hole is shown in figure 1-9. It is located just to the right of the large hole in the middle of the 5¼ inch diskette.

The index hole as shown in figure 1-9 is a hole only in the diskette's protective covering. Another index hole is located on the actual diskette surface inside the envelope. As the diskette spins, the index hole (or holes) on the diskette surface passes underneath the hole in the protective envelope.

A light source inside the disk drive shines light onto the area of the diskette containing the index hole. When an index hole on the disk surface is aligned with the index hole on the protective envelope, the light will shine through to a sensor. The sensor will relay information on the

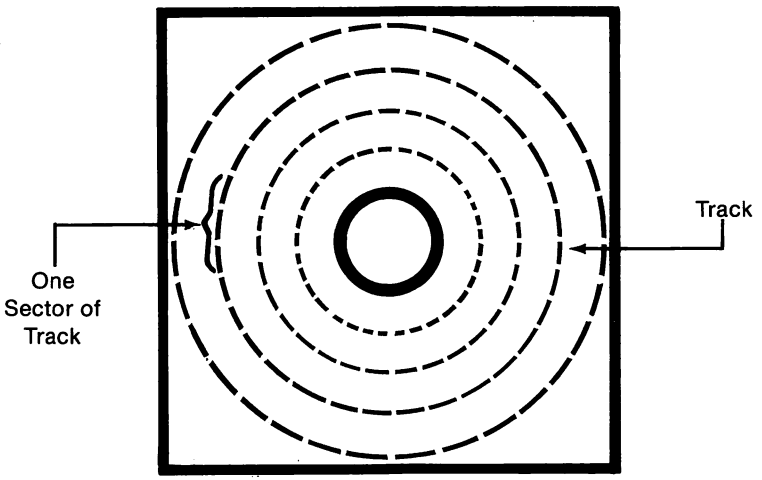


FIGURE 1-10. Tracks and Sectors

location of the index holes, which can be used to calculate the various sector locations.

Now that we have discussed the concepts of locating sectors, we will discuss the difference between hard and soft sectored diskettes. A hard sectored diskette contains a number of holes, each of which indicates the location of a sector. An extra hole is used to indicate the location of the first sector. The location of the various sectors is determined by counting the number of holes occuring after the first sector. A hard sectored diskette is depicted in figure 1-11.

Soft sectored diskettes have only one index hole as shown in figure 1-12. This solitary index hole marks the location of the first sector. By timing the rotation speed of the floppy diskette, the location of the other sectors can be determined. The IBM PC*jr* uses soft sectored diskettes.

DISKETTE CAPACITY

Two factors affect the amount of data that can be stored on a floppy diskette -- density and the number of sides to which data can be written.

Density refers to a diskette's recording format, which in turn affects its capacity. Single density (SD) 5¼ inch diskettes have roughly 90K of capacity per side. Double density (DD) 5¼ inch diskettes have a capacity

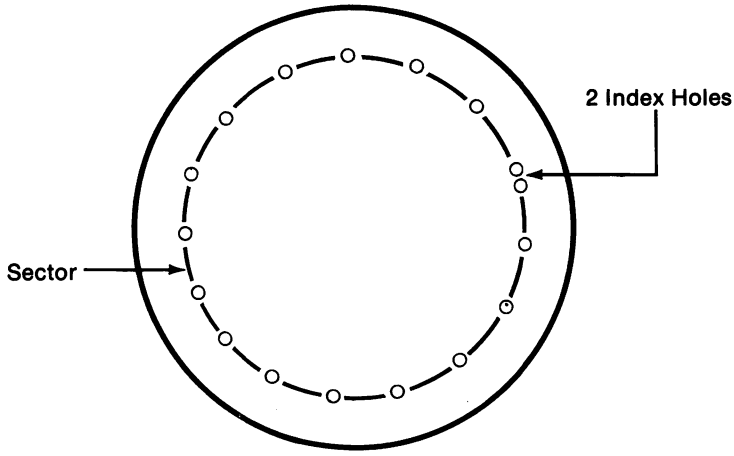


FIGURE 1-11. Hard Sectored Diskette

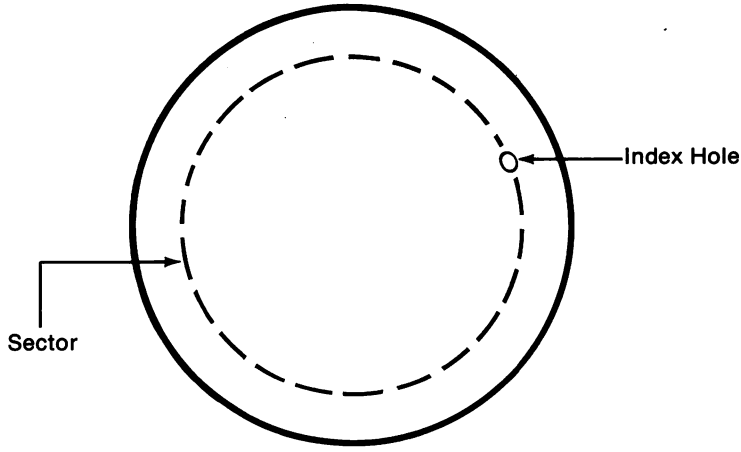


FIGURE 1-12. Soft Sectored Diskette

of approximately 180K per side.

Some floppy diskettes are designed to be written on only one side. These are known as single sided (SS) diskettes. Diskettes which are designed to be written on both sides are known as double sided (DS) diskettes. The PC*jr* uses double sided, double density diskettes (DS, DD) with a capacity of 360K.

DISKETTE WRITE PROTECTION

Diskettes have a notch on the side of their protective envelope which determines whether or not data can be written onto that diskette. On 5¼ inch diskettes this notch is known as a write-enable notch.

Information cannot be written onto a 5¼ inch diskette unless the write-enable notch has been left uncovered.

Some 5¼ inch diskettes (especially system diskettes) may be permanently write protected if their protective envelope does not contain a notch. Any 5¼ inch diskette with a notch can be write protected by merely covering the notch with a piece of tape as shown in figure 1-13.

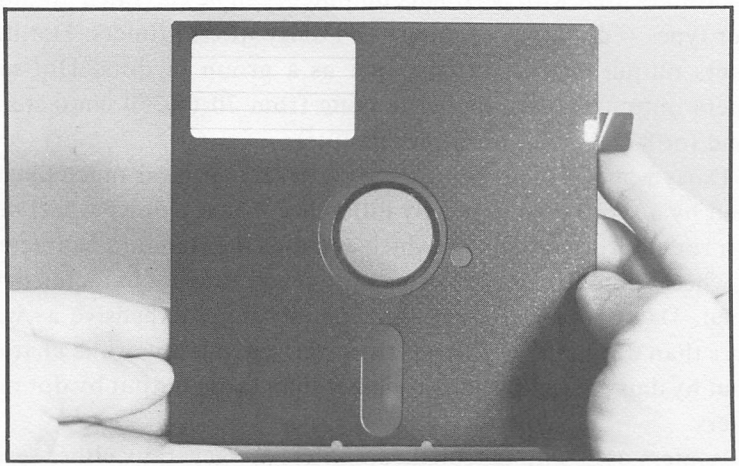


FIGURE 1-13. Write Protecting a 5¼ Inch Diskette

PC*jr* DISK DRIVE OPERATION

PC*jr* disk drive operation is a relatively simple matter. When there is no diskette in the disk drive, the disk slot handle should be in the horizontal or open position (see figure 1-14).

When inserting a diskette, the diskette's label should be facing up. The side of the diskette containing the oval-shaped opening in the cover should be inserted into the drive (see figure 1-15).

Slide the diskette into the diskette slot. Once the diskette has been fully inserted, rotate the diskette slot handle to the vertical or closed position. To remove a diskette from the drive, merely reverse this procedure.

Your PC*jr* disk drive has a small red lamp on its front cover. This lamp will light whenever data is being read from or written to a diskette. Do not remove a diskette when this lamp is on.

Printers

Printers used with personal computers can be classified among two major types -- **dot matrix** printers and **daisy wheel** printers. Dot matrix printers output characters on paper as a group of dots. Dot matrix printers output data at speeds ranging from 70 to 350 characters per second (or 800 to 4000 words per minute).

Daisy wheel printers output characters that appear much like those output by a typewriter. The only difference is that a daisy wheel printer uses a round printing element which contains the standard character set. The wheel spins to the correct position each time a character is to be printed. Daisy wheel printers are generally more expensive as well as slower than dot matrix printers. However, the quality of the characters output by daisy wheel printers is higher than those output by dot matrix printers.

Printers used with personal computers are generally either serial or parallel devices. In **serial** communications, data is transferred one bit at a time from the source device to the receiving device. In **parallel** communications, data is transferred eight bits (or one byte) at a time.

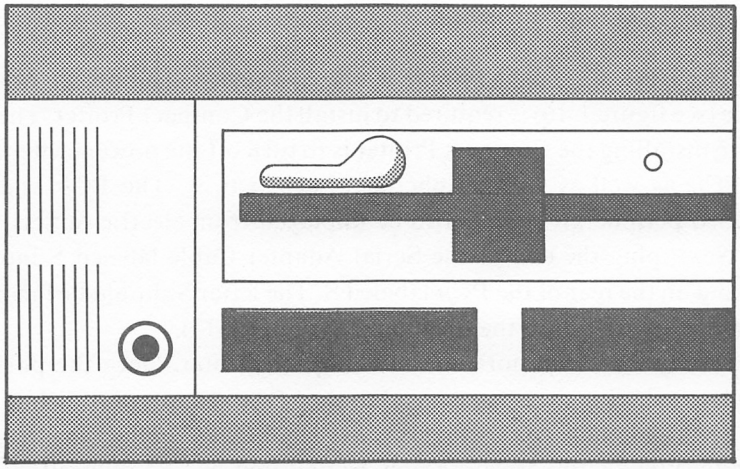


FIGURE 1-14. Diskette Slot Handle in Open Position

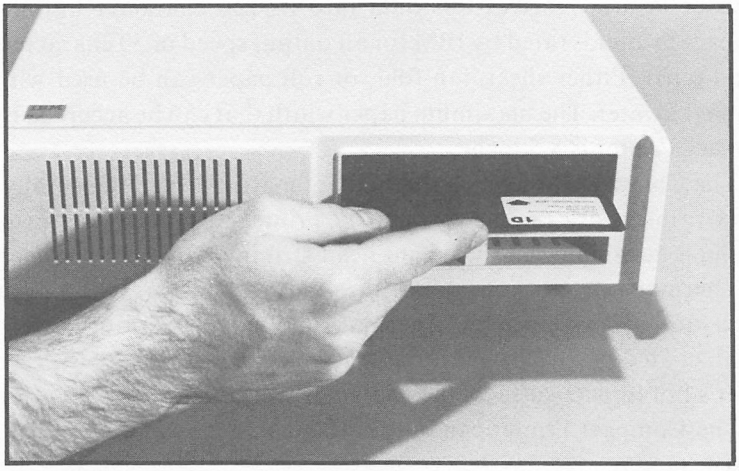


FIGURE 1-15. Inserting a Diskette into the PC*jr* Drive

IBM COMPACT PRINTER

The IBM Compact Printer is a serial device. The Serial Adapter Cable (see figure 1-16) is required to install the Compact Printer. The first step in installing the Compact Printer is to turn off the power switches on the PC*jr* as well as any peripherals attached to it. The PC*jr* and any attached peripherals should also be unpluged from electric outlets.

Next, plug the end of the Serial Adapter Cable labeled S into the opening on the rear of the PC*jr* labeled S. The letter S should be facing up when it is inserted into the opening (see figure 1-17).

Locate the serial port on the Compact Printer. Once the port has been located, plug the opposite end of the Serial Adapter Cable into that port and tighten the connecting screws. The PC*jr*, Compact Printer, and any additional peripherals can now be powered on and plugged in.

The Compact Printer can output either text or graphics characters. Text characters are output in a 5 by 7 dot matrix. Graphics characters are output with 560 dots per horizontal line with a character height of 8 dots.

Data is output from the PC*jr* at 1200 bits per second (bps). The Compact Printer receives this data into its 256 character buffer. The Compact Printer is rated by IBM for an output speed of 50 characters per second (cps). Either sheet, fan-fold, or roll paper can be used with the Compact Printer. The maximum paper width that can be accepted by the Compact Printer is 8.5 inches.

The Compact Printer is a **thermal** dot matrix printer. Generally, dot matrix printers are **impact** dot matrix printers. With impact printers, printing is accomplished by a print head striking the paper surface.

Thermal dot matrix printers utilize a less expensive technology than impact dot matrix printers. Thermal dot matrix printing requires a special, coated paper. When the paper's coating is struck by the thermal printer's hot impact surface, the character will be formed.

The Compact Printer can output text characters in any one of four different modes:

- Standard Mode
- Double-Width Mode
- Compressed Mode
- Compressed Double-Width Mode

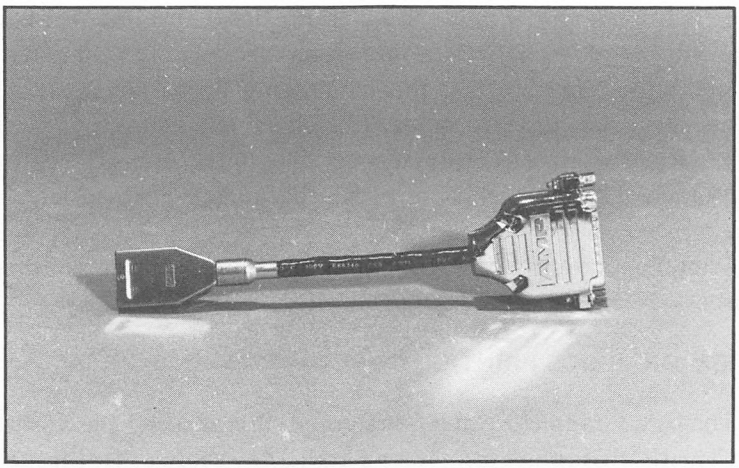


FIGURE 1-16. Serial Adapter Cable

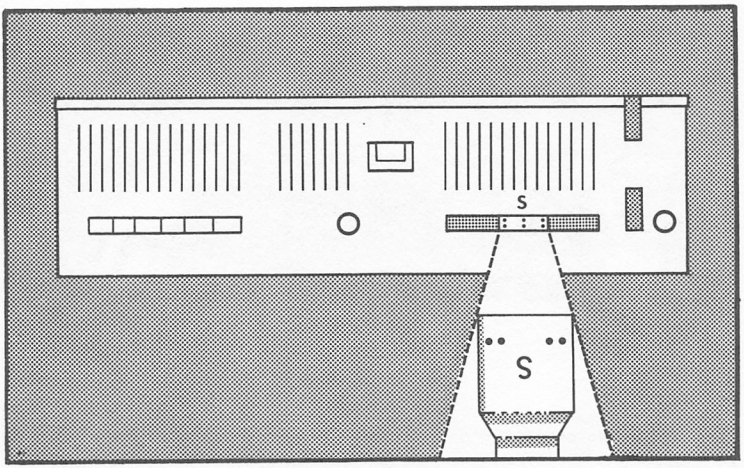


FIGURE 1-17. Serial Adapter Installation

In the standard mode, 80 characters are output per line. Ten characters are output per inch. In the double-width mode, only 5 characters per inch are output. Therefore, in this mode, only 40 characters are output per line. In the compressed mode, 17.5 characters are output per inch. This mode allows for 136 characters per line. In the compressed double-width mode, 8.75 characters are output per inch. This translates to 68 characters per line.

Printer operation and programming in BASIC for printer output will be discussed in more detail in appendix D.

IBM GRAPHICS PRINTER

The IBM Graphics Printer (see figure 1-18) can also be used with the PC*jr*. The IBM Graphics printer is a parallel device. In order for the IBM Graphics Printer to be used with the PC*jr*, a device known as the Parallel Printer Attachment must be installed on the PC*jr*. This device will also allow the connection of many parallel printers not manufactured by IBM to the PC*jr*.

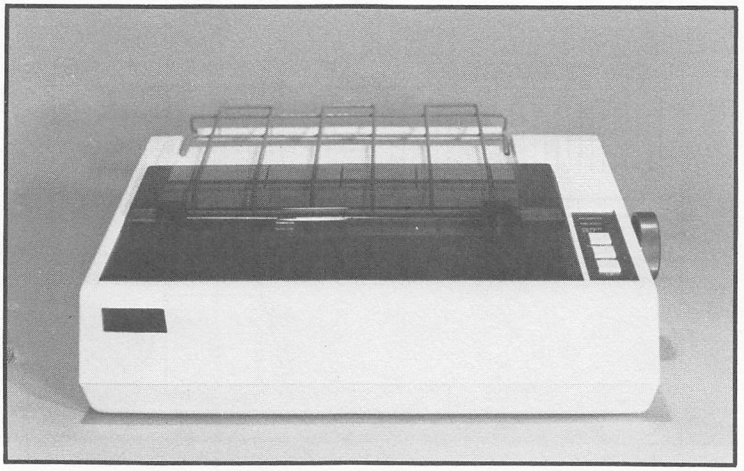


FIGURE 1-18. IBM Graphics Printer

The IBM Graphics Printer is actually a version of the Epson MX-80 printer. This printer has an output speed of 80 characters per second in any of several styles of dot matrix print. The IBM Graphics Printer will accept forms with a minimum width of 4 inches and a maximum width of 10 inches. The maximum paper thickness accepted by the IBM printer is 3 plies or .012 inches.

Paper is fed into the IBM Graphics Printer from the rear of the unit. The unit's ribbon is enclosed in a removable cartridge. Each ribbon has a print capacity of 3 million characters which translates to approximately 1150 typed pages. The IBM Graphics Printer uses a bidirectional, 9-wire printhead. The printhead has a life expectancy of 30 million impressions. The printhead can easily be replaced by the operator.

A 9 by 9 dot matrix pattern is used to form characters. Character sizes may range from 5 to 16.5 characters per inch (CPI). On an 8 inch line width, these pitch sizes (5, 8.25, 10, and 16.5) will produce a maximum of 40, 66, 80, and 132 characters respectively.

The IBM Graphics Printer offers 3 different printing styles; normal, double, and emphasized. Normal printing involves one strike of the printhead per character. Double printing involves a double strike of the printhead for each character. Emphasized printing involves a single strike for the character, after which the paper is stepped up by a fraction of an inch, and a single strike is again made. Emphasized printing gives the appearance of a bold character, which adds to the versatility of the printer.

JOYSTICKS

A joystick is a controller device used to position an object on the display. Generally, joysticks are used with game software. The IBM PC*jr* Attachable Joystick is pictured in figure 1-19.

Once the joysticks have been installed, it maybe necessary to use the centering levers (see figure 1-19) located at the top of the PC*jr* joystick to properly position the object to be controlled. Centering lever 1 controls the horizontal positioning, while centering lever 2 controls the vertical positioning.

The PC*jr* joystick can be operated in either the free mode or in the center return mode. In the free mode, the control stick will remain in

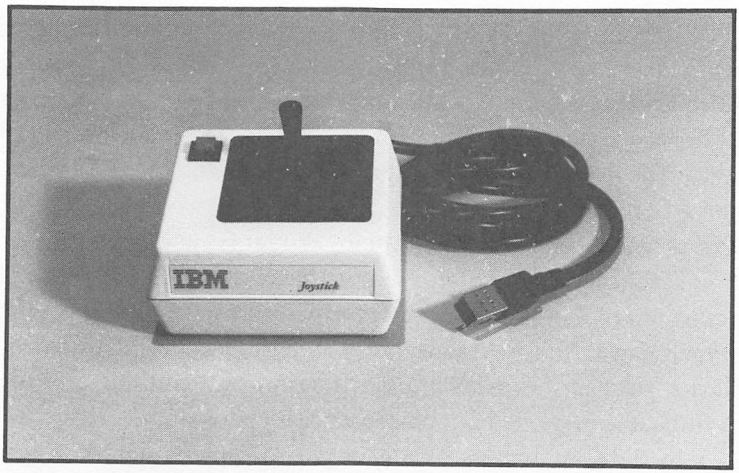


FIGURE 1-19. IBM PC*jr* Attachable Joystick

whatever position it is placed. The PC*jr* joystick can be placed in the free mode by moving the control stick to either corner and then moving either one or both of the switches located on the underside of the joystick to the free position.

In the center return mode, the controller stick always returns to the center position. The joystick can be operated in this mode by moving the switches located on the underside of the joystick to the position opposite "Free". The controller stick may be in any position when this adjustment is made.

KEYBOARD CORD

IBM's optional Keyboard Cord (see figure 1-20) must be used whenever more than one PC*jr* is being used in the same room. To install the Keyboard Cord, first of all, be certain that the PC*jr* and all attached peripherals are powered off and unplugged. Attach the end of the Keyboard Cord labelled K into the connector labelled K on the rear of the PC*jr*.

The connector on the other end of the Keyboard Cord should be plugged into the port on the rear of the PC*jr*'s keyboard.

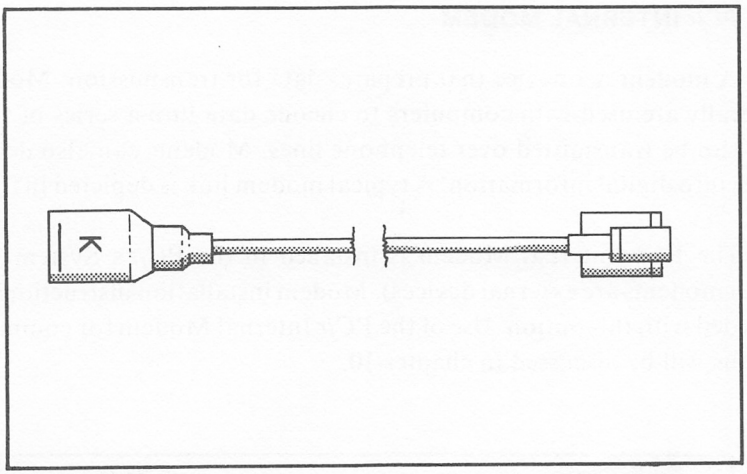


FIGURE 1-20. IBM Keyboard Cord

PC*jr* MEMORY AND DISPLAY EXPANSION

The PC*jr*'s Memory and Display Expansion device (see figure 1-21) expands the PC*jr*'s (basic model) memory from 64K to 128K. This device also supports 80 column video output. Installation of the Memory and Display Expansion is relatively simple. Instructions are included with the device.

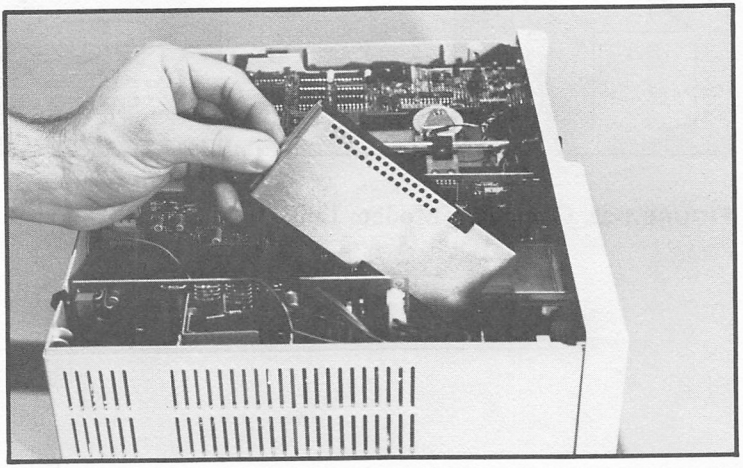


FIGURE 1-21. PC*jr* Memory and Display Expansion

PC*jr* INTERNAL MODEM

A **modem** is a device that prepares data for transmission. Modems generally are used with computers to encode data into a series of tones that can be transmitted over telephone lines. Modems can also decode tones into digital information. A typical modem link is depicted in figure 1-22.

The PC*jr* Internal Modem is installed in the PC*jr*'s System Unit (most modems are external devices). Modem installation instructions are included with this option. Use of the PC*jr* Internal Modem for communications will be discussed in chapter 10.

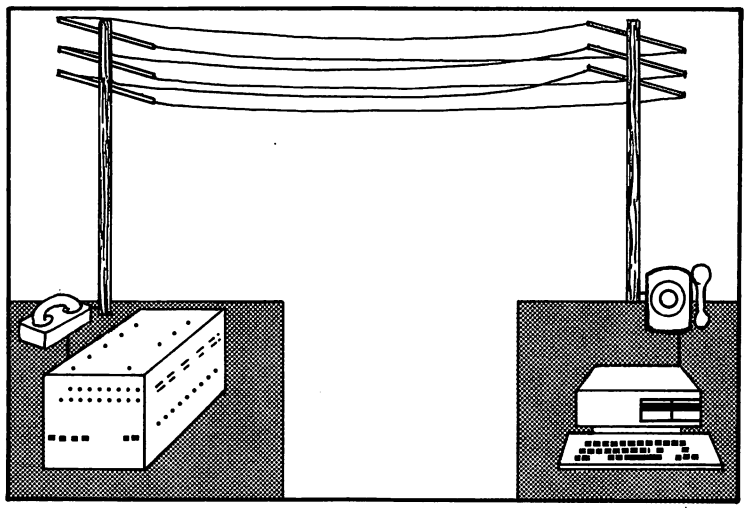


FIGURE 1-22. Computer/ Modem Link

CASSETTE RECORDER/PLAYER

The IBM PC*jr* can utilize a cassette recorder for program and data storage. Cassette recorders are the most inexpensive data storage devices available for home computers. They provide a low cost and reliable means of data storage for the budget-minded home computer consumer.

A cassette recorder uses standard cassette tapes to store data. It is good practice to use only high quality cassette tapes to save programs and data. Using lesser quality cassette tapes could result in the loss of programs and data.

The PC*jr* can use nearly any type of cassette recorder. However, a special adapter cable must be purchased in order to connect the cassette recorder to the PC*jr*. This PC*jr* adapter cable is pictured in figure 1-23.

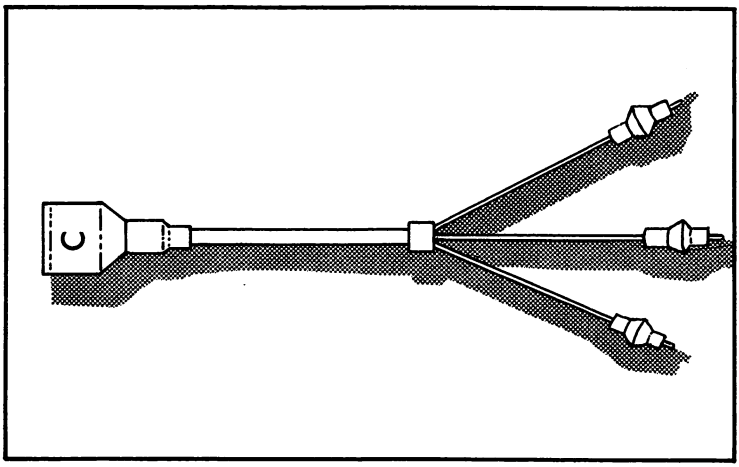


FIGURE 1-23. PC*jr* Adapter Cable for Cassette

A cassette recorder can be attached to the PC*jr* as follows:

1. Disconnect the power for the PC*jr* and all peripherals. Installing peripherals while the power is connected can be hazardous not only to the PC*jr* but to the user as well.

2. Find the end of the adapter cable with the letter "c". Insert this end into the back of the computer in the port marked "c". The "c" on the cable should be facing up.
3. The other end of the adpater cable contains three colored wires. These wires are to be connected to the cassette recorder. Plug the black cable into the cassette jack marked EAR or EARPHONE. Plug the red cable into the cassette jack marked AUX or AUXILIARY. Plug the grey cable into the cassette jack marked REM or REMOTE.
4. Plug in the power cord of the PC*jr* and all peripherals. The cassette player is now ready for operation.

2

Installation, Operation & Keyboard Usage

Introduction

In this chapter, we will explain the steps necessary for setting up the IBM PC*jr*. IBM has simplified the installation procedures for the PC*jr* to the point where almost anyone can set the unit up. Set up involves unpacking the various system components and peripherials and attaching the necessary cables and connectors.

This chapter will also explain PC*jr* keyboard usage. IBM has included a number of advanced features in the PC*jr*'s keyboard. These will be described in detail in this chapter.

Installation

In the following section, we will discuss the installation of the PC*jr* and related peripheral devices.

SYSTEM UNIT INSTALLATION

The system unit is the PC*jr*'s brain. The system unit houses all the PC*jr*'s standard and optional hardware. For example, the optional disk drive is housed within the system unit. The system unit is pictured in figure 2-1.

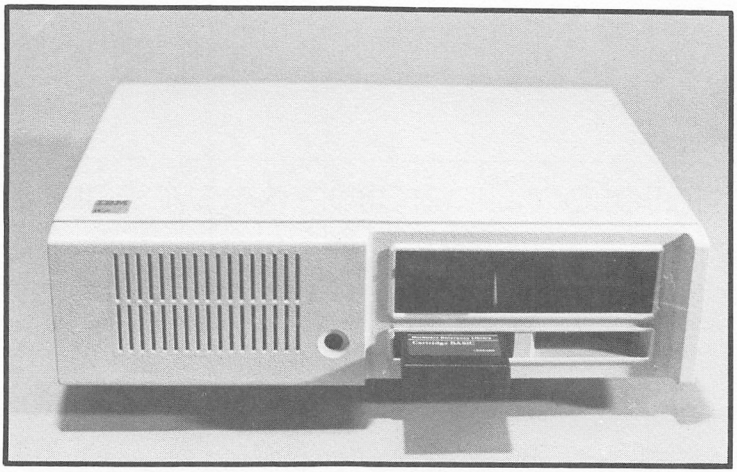


FIGURE 2-1. System Unit

The system unit must be attached to either a television set or a monitor. If a television set is used, refer to the section entitled "Attaching a Television Set." If a monitor is to be used, refer to the section entitled "Attaching a Monitor".

ATTACHING A TELEVISION SET

In order to connect a television to the PC*jr*, an optional RF modulator (T.V. connector cable) must be purchased. An RF modulator is shown in figure 2-2.

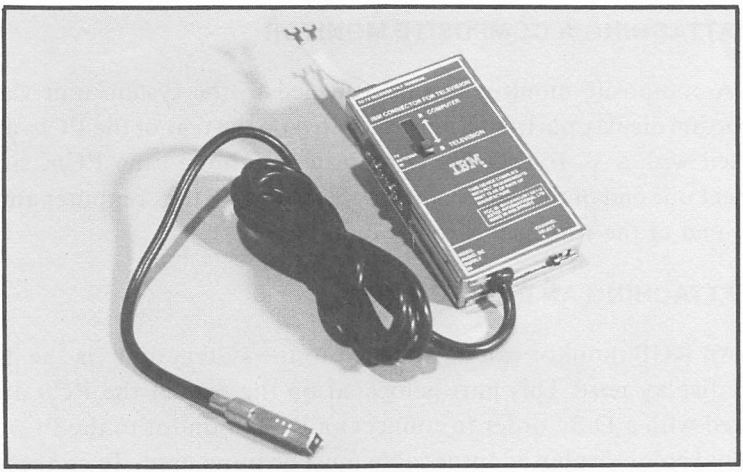


FIGURE 2-2. RF Modulator

To install the RF modulator, first disconnect the television antenna from the VHF terminals at the back of the television. Then, connect the two short wires leading from the RF modulator to the twin VHF terminals and tighten the screws. Reconnect the television antenna to the two terminals on the RF modulator and tighten the screws. Finally, connect the end of the long video cable to the port in the rear of the PC*jr* marked with a T. Be sure the T marked on the end of the video cable is facing up.

The RF modulator contains a switch marked Computer/Television. When this switch is at the computer position, the television set receives its signal from the PC*jr*. When the switch is in the television position, the television set receives its signal from the television antenna.

ATTACHING A MONITOR

Two types of monitors can be used with the PC*jr*, a composite monitor or an RGB monitor. If a composite monitor is being used, refer to the section "Attaching a Composite Monitor." If an RGB monitor is used, refer to the section "Attaching an RGB Monitor."

ATTACHING A COMPOSITE MONITOR

A composite monitor can be attached to the system unit via the composite display port. This port is located on the rear of the PC*jr*. and is marked with a V. To attach a composite montior to the PC*jr*, simply connect one end of the video cable to the V port on the computer and the other end of the video cable to the monitor.

ATTACHING AN RGB MONITOR

An RGB monitor can be attached to the system unit via the direct drive display port. This port is located on the rear of the PC*jr* and is marked with a D. In order to connect an RGB monitor to the PC*jr*, the optional color display adaptor cable must be purchased. To connect the cable, simply insert the end of the cable marked with a D into the port in the rear of the PC*jr*. Be sure the D on the cable is facing up. Connect the other end of the cable to the RGB monitor.

ATTACHING THE POWER TRANSFORMER

The power transformer must be attached to the PC*jr* via the port marked P on the rear of the system unit. Notice the two cables which originate from the power transformer. The end of the cable marked with a P must be inserted into the port marked P. The other cable must be inserted into a standard three prong wall outlet. The power transformer is pictured in figure 2-3.

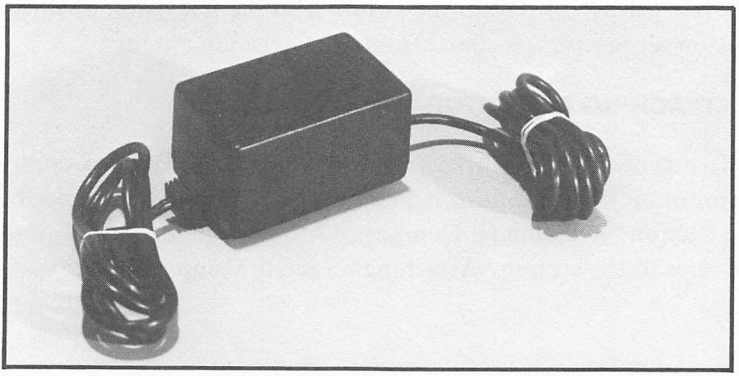


FIGURE 2-3. PC*jr* Power Transformer

KEYBOARD INSTALLATION

The IBM PC*jr* is equipped with a cordless keyboard. The cordless keyboard has the advantage of allowing the user more freedom of movement. The keyboard has a maximum operating range of 20 feet.

An infrared link allows communication between the keyboard and the system unit. The keyboard transmits an infrared light signal to the system unit each time a key is pressed. In order for a proper link to be maintained, the keyboard must be within a 30° angle of the system unit.

The keyboard is powered by four AA batteries. The batteries must be installed before the PC*jr* can be used. In order to install the batteries, follow the steps outlined below.

1. Turn off the system unit and all peripherals.
2. The battery compartment is located on the underside of the keyboard in the upper left corner. Open the cover of the battery compartment as illustrated in figure 2-4.
3. Insert the batteries into the battery compartment as illustrated in figure 2-5.
4. Close the cover of the battery compartment. The keyboard is now ready for operation.

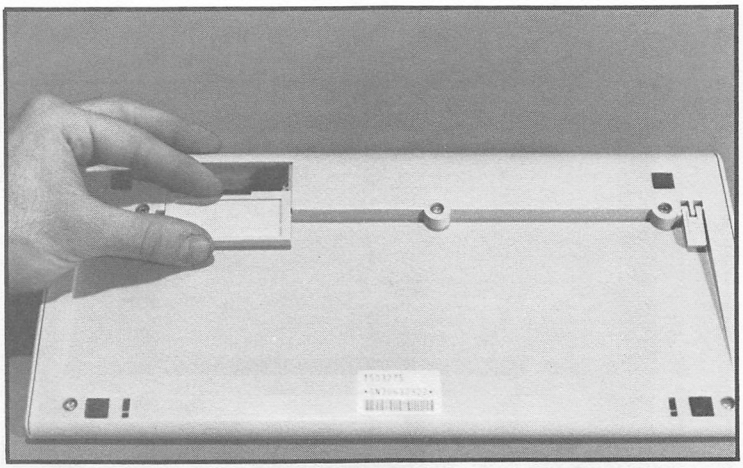


FIGURE 2-4. Opening Battery Compartment

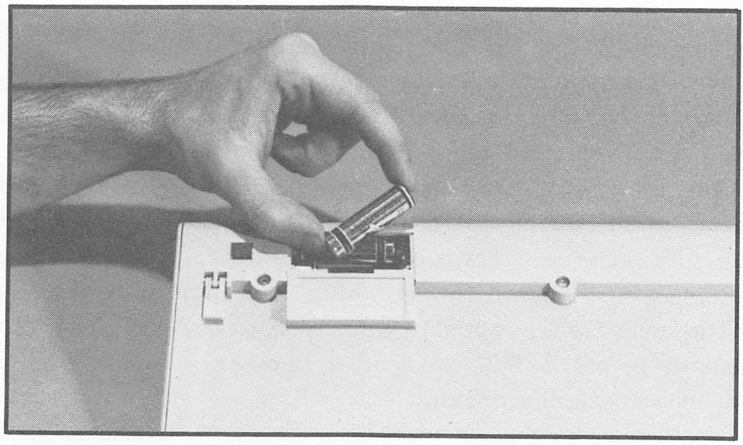


FIGURE 2-5. Inserting Batteries

Start Up Procedure

Be sure that the PC*jr* and all optional peripherals have been properly installed. The PC*jr* may now be turned on. The power switch is located in the rear of the system unit. Move the switch to the 1 (or on) position. A few seconds will elapse before a screen similar to that pictured in figure 2-6, will appear. At this point, the PC*jr* is performing a self-test. The number in the lower right of the screen indicates the amount of RAM which has been tested.

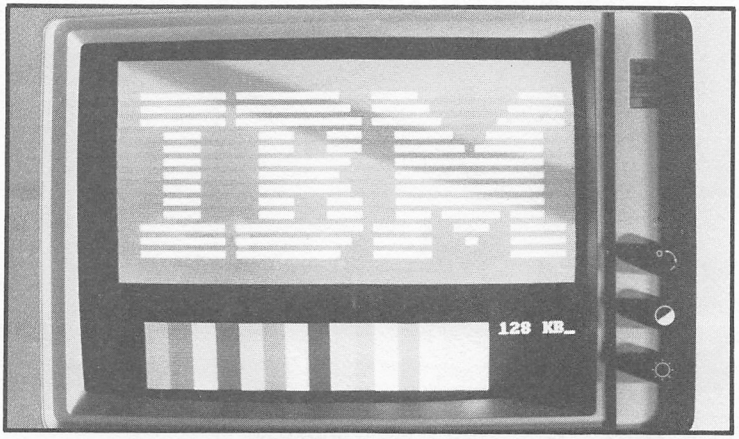


FIGURE 2-6. Start Up Display

One of the four following phenomenon will occur following the self-test:

1. If the optional disk drive is installed, and the DOS diskette (Disk Operating System diskette) is inserted into the diskette drive, the PC*jr* will boot DOS. (See chapter 9 for a more detailed explanation on booting DOS).
2. If the BASIC cartridge is inserted into one of the two cartridge ports, the PC*jr* will load BASIC and display the BASIC prompt.
3. If the DOS diskette is inserted into the disk drive and the BASIC cartridge is inserted into one of the two cartridge ports, the PC*jr* will boot DOS. BASIC can be loaded by typing the command, BASIC, while in DOS.
4. If neither a disk nor cartridge is present, the PC*jr* will start cassette BASIC. Cassette BASIC is built into the PC*jr*'s ROM.

KEYBOARD USAGE

The PC*jr*'s keyboard is pictured in figure 2-7. Most its keys are similar to those of a standard typewriter keyboard. The alpha-numeric keys and shift key are used as they are on a standard typewriter.

Many keys are marked with two characters, one in black and one in white. Pressing one of those keys will cause the white character to appear. The character in black can be accessed by holding down the shift key and then pressing the key with the desired character.

The alternate (Alt) and function (Fn) keys have special purposes depending on whether DOS or BASIC is being used. DOS uses the Fn key, in conjunction with the numeric keys, for editing. For more information on DOS editing, see chapter 9.

BASIC uses the Fn and Alt keys to access commands which have been preprogrammed into keys. For example, holding down the Fn key and pressing the 1 key causes the LIST command to be displayed on the screen. The LIST command can now be executed by simply pressing the enter key.

Preprogramming keys with BASIC commands saves program entry time. A list of the BASIC commands available using Fn and Alt is provided in table 2-1.

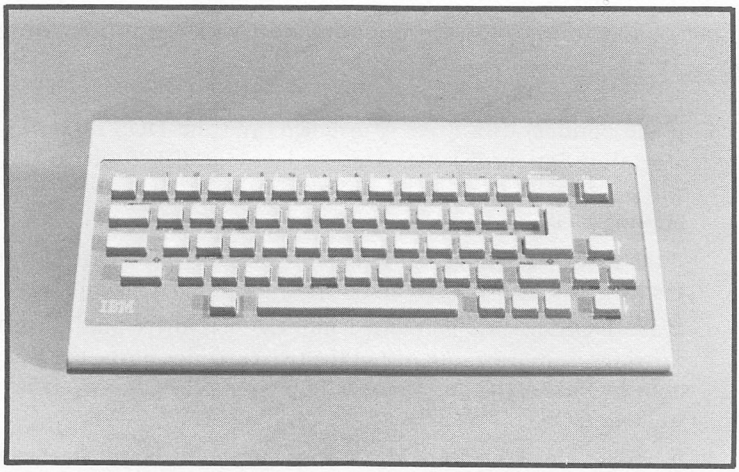


FIGURE 2-7. PC*jr* Keyboard

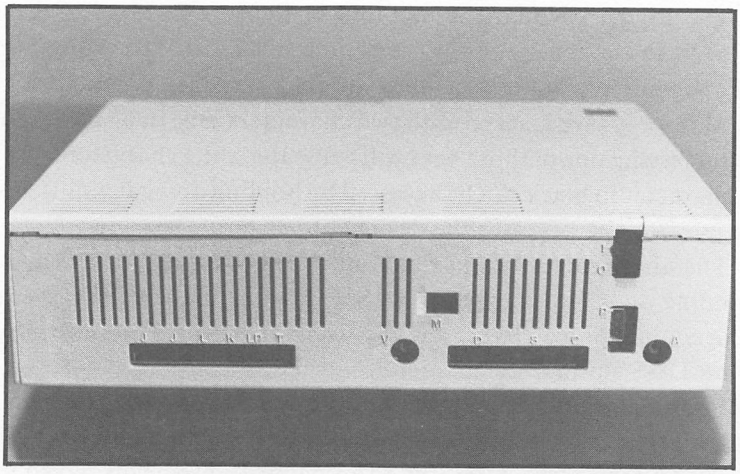


FIGURE 2-8. Rear View of PC*jr*

TABLE 2-1. Preprogrammed Keys in BASIC

Hold Down	Press	Result
Alt	A	AUTO
Alt	B	BSAVE
Alt	C	COLOR
Alt	D	DELETE
Alt	E	ELSE
Alt	F	FOR
Alt	G	GOTO
Alt	H	HEX$
Alt	I	INPUT
Alt	K	KEY
Alt	L	LOCATE
Alt	M	MOTOR
Alt	N	NEXT
Alt	O	OPEN
Alt	P	PRINT
Alt	R	RUN
Alt	S	SCREEN
Alt	T	THEN
Alt	U	USING
Alt	V	VAL
Alt	W	WIDTH
Alt	X	XOR
Fn	1	LIST
Fn	2	RUN [Ret]
Fn	3	LOAD"
Fn	4	SAVE"
Fn	5	CONT [Ret]
Fn	6	"LPT1:" [Ret]
Fn	7	TRON [Ret]
Fn	8	TROFF [Ret]
Fn	9	KEY
Fn	0	SCREEN 0,0,0 [Ret]

PERIPHERAL PORTS

The IBM PC*jr* has a number of ports designed to accommodate specific peripherals. The majority of these ports are located on the rear of the system unit (see figure 2-8). Each port is marked with a letter which represents the type of peripheral that can be attached to that port. Table 2-2 describes the peripheral ports on the PC*jr* as they appear from left to right.

When attaching peripherals to the PC*jr*, it is important to remember to first disconnect the power from the computer. Merely turning off the computer is an insufficient precaution when installing peripherals. Always disconnect the computer power cord. Installing peripherals without first disconnecting the power is dangerous to both the user and the computer.

TABLE 2-2. Peripheral Ports

Port	Name	Function
J	JOYSTICK1 PORT	Allows a joystick to be connected to the PC*jr*.
j	JOYSTICK2 PORT	Allows a second joystick to be connected to the PC*jr*.
L	no name	Reserved for possible future use.
K	KEYBOARD PORT	Allows the optional cable to be attached between the keyboard and the main console.
LP	LIGHT PEN PORT	Allows the optional light pen to be attached to the PC*jr*.
T	TELEVISION PORT	Allows a television to be connected to the PC*jr* via an RF modulator.

TABLE 2-2. (cont.) Peripheral Ports

Port	Name	Function
V	COMPOSITE DISPLAY PORT	Allows a composite display to be attached to the PC*jr*.
M	MODEM PORT	Allows connection between PC*jr* and a telephone line. The optional internal modem board must have been previously installed in the PC*jr*.
D	DIRECT DRIVE DISPLAY PORT	Allows the IBM color graphics display to be connected to the PC*jr*.
S	SERIAL PORT	Allows serial devices to be connected to the PC*jr* via the optional serial adaptor cable.
C	CASSETTE PORT	Allows a cassette recorder to be attached to the PC*jr* via the optional cassette adaptor cable.
P	POWER PORT	Allows the power cord and transformer to be attached to the PC*jr*.
A	AUDIO PORT	Allows external speakers to be connected to the PC*jr* via the audio adaptor cable.

3

Introduction to PCjr BASIC Programming

Introduction

In this chapter, the operating details necessary to begin using PC*jr* BASIC will be provided. These include start-up, program entry, statement structure, program editing, program saving, and program loading.

Getting Started with PC*jr* BASIC

As mentioned in chapter 1, BASIC is a high-level language that must be interpreted into the microprocessor's native language. This is accomplished with a program known as an **interpreter.**

The PC*jr* has a resident interpreter (Cassette BASIC) and an optional cartridge interpreter (Cartridge BASIC). These two versions of BASIC are upward-compatible. That is, anything that can be done with Cassette BASIC can also be done with Cartridge BASIC. Also, Cartridge BASIC includes features not available in Cassette BASIC.

The correct BASIC start-up procedure depends on whether or not the system contains a disk drive. Both methods will be detailed in the following sections.

START-UP WITHOUT A DISK DRIVE

This section describes how to start-up BASIC without a disk drive. Therefore, if the system includes a disk drive, skip this section and proceed to the next.

To get started, remove any cartridges from the cartridge slots on the front of the system unit. If Cartridge BASIC is to be implemented, the Cartridge BASIC cassette should now be inserted into the left-hand cartridge slot.

If the PC*jr* is not yet powered up, do so. The IBM logo will appear while the PC*jr* performs a series of self-tests. (If the ERROR B message is displayed on the screen, press Enter and proceed*.) About ten seconds later, several words will appear at the bottom of the screen. A message similar to the following will be displayed atop the screen:

```
The IBM PC jr Basic
Version C1.20
Copyright IBM Corp. 1981
62940 Bytes free
Ok
```

Cassette BASIC is known as "Version C". If the BASIC cartridge had been inserted, "Version J" would have replaced "Version C" at the beginning of line two. Cartridge BASIC is known as "Version J".

If the system has already been powered up, BASIC can still be restarted. To do so, press and hold down the Ctrl and Alt keys, then press the Del key. The Ctrl-Alt-Del sequence can be used at any time to restart the PC*jr*.

* If pressing Enter does not clear the error, the keyboard may not be working properly.

START-UP WITH A DISK DRIVE

The resident BASIC interpreter, Cassette BASIC, does not support disk access. However, disk access is supported by Cartridge BASIC. Therefore, PC*jr*'s with a disk drive should be started with the Cartridge BASIC cartridge inserted in the lefthand slot on the front of the System Unit. Also, a DOS 2.10 system diskette should be inserted into the disk drive. Remember, never use the original copy of DOS. Make a copy, and then store the original in a safe place. Always use a copy for everyday use. Chapter 9, "DOS 2.10 Usage", explains how to make back-ups.

If the system is not yet powered up, do so. The IBM logo will appear while the PC*jr* performs its self tests. (If the ERROR B message is displayed on the screen, press Enter and proceed*.) A message similar to the following should appear on the screen:

```
Current date is Tues 1-01-1980
Enter new date:
```

Either enter the date or merely press Enter to proceed. The next message to appear will be similar to the first:

```
Current time is 0:00:34.49
Enter new time:
```

Again, either enter the time or merely press Enter to proceed. The final DOS message will appear as follows:

```
The IBM Personal Computer DOS
Version 2.10 (C) Copyright 1981,
1982, 1983
A>
```

The "A>" is the DOS prompt. DOS commands are entered in response to this prompt. The user should now type the word BASIC, and then press Enter. The BASIC start-up display should now be visible.

---

* If pressing Enter does not clear the error, the keyboard may not be working properly.

```
The IBM PC jr Basic
Version J1.00
Copyright IBM Corp 1981, 1982, 1983
59694 Bytes free
Ok
```

If the system has already been powered up, DOS can still be rebooted. To do so, press and hold down the Ctrl and Alt keys, then press the Del key. The Ctrl-Alt-Del sequence can be used at any time to perform a System Reset.

BASIC COMMAND ENTRY — DOS

This section may seem a bit confusing for beginning programmers. If this section is overly confusing, skip it and continue with the rest of chapter 3.

When activating BASIC, additional parameters can be included with the BASIC entry. These parameters can be used to:

- Automatically load and run a program
- Set aside storage areas for programs, data and buffers
- Change the standard input or output device

The configuration for the BASIC command with all of its optional parameters is as follows:

BASIC [A][*filespec*][<*input*][>][>*output*][/F:*files*][/S:*bufsize*] [/C:*bufcom*] [/M:[*workarea*][,*blockspace*]]

In order to be compatible with the PC, Cartridge BASIC allows either "BASIC" or "BASICA" to be used as the BASIC command entry. The two command entries have identical results.

The optional parameter, *filespec,* indicates the file specification of a program that is to be automatically loaded and executed by BASIC. If no filename extension is specified, the extension .BAS will be used. *filespec* can also include an optional directory path specification.

Generally, data is input into a BASIC program via the keyboard. <*input* allows a BASIC program to receive data from a file specified by the user. For example, the following command line causes INPUT, LINE INPUT, INPUT$, and INKEY$ to read data from the file, TEXTA.

```
BASIC PROGRAMA < TEXTA
```

Generally, a BASIC program outputs data to the screen. *>output* allows a BASIC program to output data to a file specified by the user. For example, the following command causes the PRINT statement to output data to the file named TEXTB:

```
BASIC PROGRAMB > TEXTB
```

When *>>output* is used in place of *>output*, any data output by the program will be appended to the file indicated by *output*. For example, the following command line causes the PRINT statement to append data to the file named TEXTC:

```
BASIC PROGRAMC >> TEXTC
```

Any BASIC command line option preceded by a slash (/) is known as a **switch.** Switches are used to specify optional parameters on the BASIC command line. These control memory allocation and the number of allowable files.

The optional parameter /F:*files* indicates the maximum number of files that can be open at one time while a BASIC program is being executed. The maximum number is 15. If this parameter is omitted, the number of files open will default to 3.

The number of files that can be specified with /F depends upon the value indicated for the FILES parameter in CONFIG.SYS, the DOS configuration file. The default for FILES is 8, of which BASIC uses 4 files by default. This leaves 4 files available for BASIC I/O operations. Therefore, the maximum value for /F when FILES = 8 is /F:4.

The optional parameter /S:*bufsize* assigns the buffer size to be used with random files. The default value for the buffer size is 128. The maximum size that can be used is 32767. For maximum performance, IBM recommends that a buffer size of 512 bytes be specified. Note that the record length parameter used with the OPEN statement may not be greater than the number of bytes specified in /S:*bufsize*.

The optional parameter /C:*bufcom* is used to initialize the area to be reserved for the communications receive buffer. This parameter will not be used unless the Internal Modem has been installed on the system. The

default value for this parameter is 256 bytes for the receive buffer and 128 bytes for the transmit buffer. IBM recommends that the value /C:1024 be used for high speed lines. If a value of /C:0 is specified, no buffer space will be reserved for communications. In this case, communications support will be disabled when BASIC is loaded.

The optional parameter /M:*workarea* can be used to initialize the maximum amount of memory in bytes that can be used as a work area under BASIC. The maximum value that can be used is 64K. The default value is 64K. This parameter is often used to reserve work areas for machine language subroutines.

/M:*blockspace* allows the user to reserve a working area above the BASIC work area in memory for loading BASIC programs. *blockspace* is specified in multiples of 16. When *blockspace* is omitted, the default multiplier (4096) will be used. Therefore, since 4096 × 16 = 65536, 65536 bytes are allocated by default for BASIC's work area. The following parameter,

```
/M:,2048
```

would reserve a maximum of 32,768 bytes for BASIC working area (as 2048 × 16 = 32,768). The following command,

```
/M:32512,2048
```

would reserve 32,768 bytes for BASIC but would only allow the usage of the lower 32,512 bytes. The remaining 256 bytes would be reserved for program storage.

IMMEDIATE AND PROGRAM MODES

The immediate mode is also known as the direct or calculator mode. In the immediate mode, most BASIC command entries result in the instructions being executed without delay. For example, if the following immediate mode line was typed, and the Enter key pressed,

```
PRINT "Walter A. Haupt"
```

the following would be displayed on the video screen:

```
Walter A. Haupt
```

In the program or indirect mode, the computer accepts program lines into memory, where they are stored for later execution. This stored program will be executed when the appropriate command (generally RUN) is entered.

Figure 3-1 contains an example of the entry of a program in the program mode and its execution. Notice that in the program mode, each BASIC program line must be preceded with a line number. Line numbers will be discussed in more detail later in this chapter.

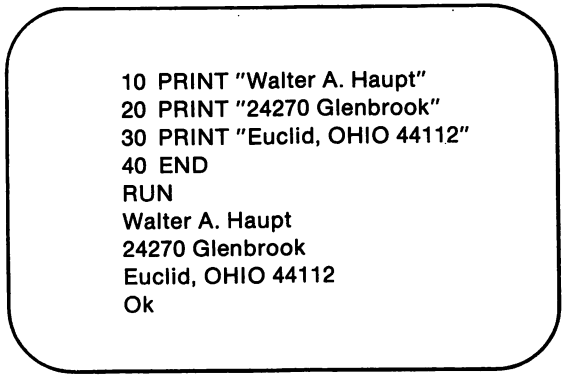


FIGURE 3-1. Program Mode Entry & Execution

COMMAND AND STATEMENT STRUCTURE

In PC*jr* BASIC, instructions being relayed to the interpreter are known as **commands** in the immediate mode, and **statements** in the program mode. In practice, the difference between a command and a statement is primarily one of semantics, as both generally use the same structure and keywords.

Both commands and statements begin with a BASIC **keyword** or **reserved word.** The keyword identifies the operation to be undertaken by the BASIC interpreter. For example, in the preceding section, the PRINT command was used to instruct the PC*jr* to display information on the screen.

In PC*jr* BASIC, keywords may be entered in either uppercase or lowercase letters. In the examples in this book, keywords will be dis-

played as uppercase letters.

A BASIC command or statement generally includes one or more **arguments** or **parameters** following the keyword. In our example, "Walter A. Haupt" is the PRINT statement parameter.

```
PRINT "Walter A. Haupt"
```

ENTERING A PROGRAM

In the preceding section, the fundamentals of entering and running a PC*jr* BASIC program have been touched upon. In this section, that discussion will be expanded upon by using the example in figure 3-2.

BASIC programs are entered as **program lines.** Any text preceded with a number (**line number**) and ended by pressing the Enter key will be regarded as a program line. The maximum number of characters that may be included in any one line is 255. Line numbers must be integers in the range 0 to 65529. If a line either exceeds its character limit, or a line number is not valid, an error condition will result.

Note that in the first 7 lines of figure 3-2, a program was entered in the command mode and run in the execute mode. After the answer, 5, had been displayed, the "Ok" prompt appeared.

At this point, the original program will be stored in memory, and can be added to or changed. That is what was done in line number 150 of figure 3-2. An additional statement was inserted between statements 100 and 200 in the program being stored in memory. This revised program can be executed by again entering RUN.

The computer memory can only hold one program at a time. The NEW command is used to erase the program in memory so as to allow a new program to be entered. Note the use of NEW in figure 3-2.

Note in our examples the following features common to BASIC programs:

1. Each program line must begin with a line number. The computer executes program lines in order from lowest line number to highest line number.
2. The END statement signals the end of a program. When END is executed, the program run will stop.

It is recommended that consecutive line numbers (10, 11, 12, 13, etc.) not be used in programs. By using numbers that are a fixed distance apart (100, 110, 120, 130, etc.), additional lines can be inserted between existing lines without renumbering the lines.

Line numbers need not be entered in any particular order. For example, the user could enter lines 100 and 200 and then enter line 150. The computer will automatically rearrange the lines according to their line numbers.

If two lines are entered with the same line number, the original line will be erased, then replaced with the new line. This feature allows the user to replace an entire line by merely entering a new line with the same line number.

A new line can be added to a BASIC program by merely entering a line number followed by the desired text and Enter. When Enter is pressed, the line will be saved as part of the BASIC program.

To delete a line in an existing program, merely enter the line number of the line to be deleted followed by Enter. A group of lines can be deleted via the DELETE command, and an entire program can be deleted with the NEW command.

```
NEW
Ok
100 PRINT 5
200 END
RUN
 5
Ok
150 PRINT -5
RUN
 5
-5
Ok
NEW
Ok
100 PRINT 50
200 END
RUN
 50
Ok
```

FIGURE 3-2. Entering and Running a Program

ERROR AND WARNING MESSAGES

When a statement with an incorrect format has been entered, an error message will be displayed. The error message describes the type of problem that occurred.

If a problem develops while a program is being executed, an error message will also be displayed. An error that occurs during the execution of a program will generate an error message that includes a description of the problem as well as the line number of the statement that caused the problem.

When an error occurs in a program, an error message will be displayed and the execution of the program will stop. If a problem occurs in a program that is not serious enough to stop the execution, a warning message will be displayed. Warning messages describe the nature of the problem as well as the line number where the problem occurred.

LISTING A PROGRAM

LIST is used to display the program stored in memory on the screen or printer. This display is often referred to as a **program listing.** An example of the use of LIST is given in figure 3-3.

When the LIST command is executed, the program in the computer's memory will be displayed on the screen. Each line of the program appears initially at the bottom of the display. In order for each subsequent line of the program to appear on the last line of the display, each line of the display must be moved one line toward the top.

As a result, if a program occupies more than 24 display lines, the first lines of the program will be moved off the top of the display in order to accommodate the last lines. This process is called **scrolling.**

When a lengthy program is listed on the display, the information will only be displayed briefly. As a result, it is often necessary to temporarily halt the listing of a program. If this is the case, simply hold down the Fn key and type Q. This key combination, Fn-Q, is known as the Pause function. The pause continues until another key is pressed.

LIST can be used with optional parameters to display only a portion of the program. For example, LIST can be used with a single line number parameter to list only that line. This is shown in figure 3-3.

LIST can also be used with a range of line numbers. For example, the command LIST 10-30 would list all line numbers within the range 10 to 30. This is again shown in figure 3-3.

LIST can be used to display all line numbers from the beginning of the program to a specified line by prefixing that line number with a hyphen and using it as the LIST parameter. If the line number is followed by a hyphen, all program lines after and including the specified line will be listed. These are also demonstrated in figure 3-3.

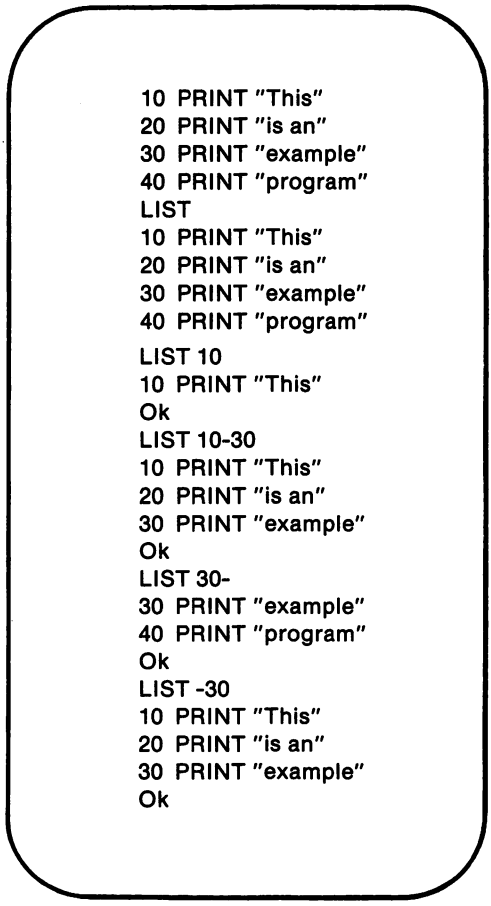


FIGURE 3-3. Listing a Program

EDITING A PROGRAM

If a program line is entered incorrectly, it can be changed in one of two ways. The first method is to simply re-enter the program line. This is accomplished by retyping the line number, followed by one or more appropriate statements.

The second method uses the PC*jr*'s full screen edit feature to alter a program line. This feature allows the cursor to be moved to any location on the screen. Once the cursor has been positioned over the incorrect entry, the correct character or characters can be typed in place of the error.

The cursor can be moved by the 4 keys on the lower right portion of the keyboard. These keys are labelled ↑, ↓, ←, and →. The "arrow" keys move the **cursor** in the direction of the arrow. The cursor is a blinking underline (_), which indicates the position where the next character entered via the keyboard will appear.

Program lines can be manipulated by the Ins and Del keys. Ins sets the insert mode. Del deletes the character at the current cursor position. The use of the full-screen editor is best explained using a simple example. Begin by entering the following program:

```
10 FOR S = 100 TO 200
20 PRINT S
30 NEXT R
```

When the LIST command is issued, the program will appear on the display as follows:

```
LIST
10 FOR S = 100 TO 200
20 PRINT S
30 NEXT R
Ok
```

Suppose that line number 30 was incorrect and was intended to appear as follows:

```
30 NEXT S
```

The correction can be made by using the ↑ key to move the cursor up to line 30. Proceed by using the → key to move the cursor beneath the "R". Correct the error by typing the correct letter, "S", then press Enter.

Suppose that line number 10 was intended to read as follows:

```
10 FOR S = 1 TO 200
```

Again, use the arrow keys to position the cursor beneath the first offending "0". Pressing the Del key will remove the first "0" from the line. The cursor should now be positioned beneath the other "0". Pressing the Del key a second time will delete the second "0". Now, press Enter. The PC*jr* does not record any corrections until the Enter key has been pressed. Therefore, once the line has been edited, always press Enter to register the line.

Finally, suppose line number 20 is also incorrect, and was intended to appear as follows:

```
20 PRINT "THE NUMBER IS ";S
```

Use the arrow keys to position the cursor beneath the "S" in line 20. This is the position where the additional characters are to be inserted. Press the Ins key once to enter the insert mode. The cursor will change from the standard underline to a rectangle. Now, type the text to be inserted, "THE NUMBER IS ";. Notice that in the insert mode, any characters to the right of the cursor will be moved over to make room for the additional characters. To exit the insert mode, either repress Ins or use one of the arrow keys. Remember to press Enter after editing line 20, so that the changes will be stored.

RUNNING A PROGRAM

Once a program is present in memory, the operator can execute it. As mentioned previously, a program can be entered into memory via the keyboard or loaded into memory from a storage device — cassette or disk. The procedure for loading a program will be discussed later in the chapter.

The RUN command is used to begin program execution. RUN can be used with or without an optional line number or file specification as its

parameter. Because RUN is generally executed without an optional parameter, we will limit our discussion of RUN, in this section, to its execution without parameters. The usage of RUN with parameters will be discussed in chapter 6.

When the RUN command has been entered and the Enter key pressed, program statements entered in the indirect mode (with line numbers) will be executed in order, beginning with the lowest line. An example of the usage of RUN is shown in figure 3-4. The execution of a program can be stopped at any time by holding down the Fn key and then typing B. This key combination, Fn-B, is known as the Break function.

```
100 PRINT "THIS IS LINE 1"
200 PRINT "LINE 2 IS BEING EXECUTED"
300 PRINT "LINE 3 IS BEING EXECUTED"
400 PRINT "LINE 4 IS THE FINAL LINE"
500 END
RUN
THIS IS LINE 1
LINE 2 IS BEING EXECUTED
LINE 3 IS BEING EXECUTED
LINE 4 IS THE FINAL LINE
Ok
```

FIGURE 3-4. RUN Command

SAVING A PROGRAM

As you may recall from our discussion of program entry, only one BASIC program may be stored in memory at any one time. When the PC*jr*'s power is turned off, the contents of memory will be erased and any program stored there will be lost unless it is first stored on a permanent medium such as a diskette or a cassette tape.

Before a program can be saved, it must first be assigned a name from one to eight characters in length. This is known as a **filename.** Once a filename has been selected for a program, it can be saved using the SAVE

command.

For example, if a program was presently residing in memory, it could be saved on a diskette with the following command:

```
SAVE "PROGRAM"
```

Notice that quotation marks are required around the filename, PROGRAM.

When SAVE is executed, the program remains in memory where it can be added to, edited, or run if desired.

Both cassettes and disks can be an effective means of retaining programs and data when the computer is turned off. The details of the procedures used to save programs and data will be presented in chapter 6.

LOADING A PROGRAM

Once a program has been saved on cassette tape or floppy disk, it can be loaded back into memory using the LOAD command. An example of a LOAD command is given below.

```
LOAD "PROGRAM"
```

Again, the quotation marks are required around the filename, PROGRAM.

If the file, PROGRAM, is not found on either the disk or cassette, a "File Not Found" error will be generated. Whether the PC*jr* searches the disk or cassette for the file depends on your specific system configuration. See chapter 6 for details.

MULTIPLE STATEMENTS

In our examples thus far, only one BASIC statement has been included in each program line. In PC*jr* BASIC, multiple statements may be included in a single program line as long as each statement is separated with a colon. The following program uses multiple statements in line 10:

```
10 PRINT "JOHN":PRINT "NELSON"
20 PRINT "ATLANTA"
30 PRINT "GEORGIA"
40 END
RUN
JOHN
NELSON
ATLANTA
GEORGIA
Ok
```

4

Data Types, Variables, and Operators

Introduction

In chapter 3, a few of BASIC's fundamental operating details were discussed. In chapter 4, the basic concepts necessary to master PC*jr* BASIC will be examined. In particular, the various **data** types used in PC*jr* BASIC as well as the **operations** that can be performed on that data will be discussed.

Data Types

The data processed in PC*jr* BASIC can be classified under two special headings: string and numeric. String and numeric data are stored differently in memory by the PC*jr*. Also, the various **operators** in BASIC affect string and numeric data in different manners.

STRINGS

A **string** can be defined as one or more **ASCII** characters. The

various ASCII characters are listed in appendix C and consist of the digits (0-9), letters of the alphabet, and a number of special symbols.

BASIC also allows a string of zero characters. This is also known as the empty or null string and is used much as a zero is in mathematics.

As may have already been noted from our examples in chapter 3, when a string is used in a BASIC statement, it must be enclosed within quotation marks. The quotation marks serve to identify the beginning and ending points of the string. They are not a part of the string.

A string enclosed within quotation marks is known as a **string constant.** A **constant** is an actual value used by BASIC during execution. The following are examples of string constants:

```
"JOHN SMITH"
"12197"
"E97432"
"BOSTON, MA 01270"
"213-729-4234"
```

Notice that numbers can be used within a string constant. Remember, however, that the numbers within a string constant are string rather than numeric data.

One final point that should be kept in mind regarding string constants is that they cannot contain quotation marks. For example, the following string constant,

```
"John said, "Goodbye." as he walked away."
```

would be illegal. Since quotation marks are used to denote the beginning and ending points of a string constant, their inclusion within the string itself would cause difficulties and, therefore, their inclusion is not allowed. The CHR$ function can be used to place the ASCII code for quotation marks within a string constant. This will be discussed in chapter 6.

NUMERIC DATA

Numeric data can be defined as information denoted with numbers. Numeric data is stored and operated on in a different manner than is string data.

Numeric constants consist of positive and negative numbers. Numeric constants cannot include commas. For example, 10,000 would be an illegal numeric constant.

BASIC further classifies numeric constants as **integers, fixed point** numbers, and **floating point** numbers.

Integers can be defined as the whole numbers in the range between -32767 and 32767, inclusive. Integer numbers do not have a decimal portion.

Fixed point numbers can be defined as the set of positive and negative real numbers. Fixed point numbers contain a decimal portion.

Floating point numbers are represented in scientific notation. A number in scientific notation takes the following format:

$$\pm x \text{ E} \pm yy$$

$\pm$ is an optional plus or minus sign.

x can either be an integer or fixed point number. This position of the number is known as the coefficient or mantissa.

E stands for exponent.

yy is a two digit exponent. The exponent gives the number of places that the decimal point must be moved to give its true location. The decimal point is moved to the right with positive exponents, and to the left with negative exponents.

The following are examples of floating point numbers and their equivalent notation in fixed point:

Floating Point	Fixed Point
3.87E+05	387000
4.064E-04	.0004064
-1E+06	-1000000
7.87642E+03	7876.42

BASIC can only handle floating point numbers in the range between 1.70E+38 and -1.70E +38. Any decimal numbers in the range between -2.93E-39 and 2.93E-39 will be converted to zero.

Floating point notation is used as a more efficient means for the computer to manipulate exceedingly large or exceedingly small values.

As a result, some values that are entered in fixed point notation may automatically be converted to floating point notation by the computer.

Numbers represented in floating point or fixed point notation can be stored in one of two degrees of **precision,** or accuracy. Single precision numbers are stored with 7 significant digits. Double precision numbers are stored with 17 significant digits. Any additional digits will be rounded.

A single precision constant will be specified if the constant is written using any of the following formats:

- Number contains seven or fewer digits
- Number is written in exponential form (E)
- Number contains a trailing exclamation point (!)

Double precision can be specified using any of the following formats:

- Number contains eight or more characters
- Number is written in exponential form (D)
- Number contains a trailing number sign (#)

Single Precision	Double Precision
-48.7	123456789
2.56E03	2.56D03
900!	900#

Integers differ from floating point and fixed point values in that integers cannot contain digits to the right of the decimal point. This condition allows integers to be stored in a smaller area of the computer's memory. Also, integers can be handled more quickly than other types of values.

The following are examples of integers, floating point, and fixed point numeric values:

Integers	Floating Point	Fixed Point
-7978	-387E+04	47988
32600	4.015E+07	37.0
37	6.870E-27	45.874
192	1D+06	3.1415927
-687	1.414D+00	-238.5

Note that 47988 cannot be considered an integer since it lies outside the allowable range of values (-32767 to 32767). Also, note that 37.0 is not an integer because it contains a digit to the right of the decimal point.

Variables -- An Overview

In the preceding section, we discussed BASIC's different types of data -- string and numeric. So far, data has only been represented as a constant. The value of a string or numeric constant such as "MICHELLE" or 382.436 always remains the same.

Data can also be represented by using a **variable.** A variable can be defined as an area of memory that is represented with a name. That name is known as the **variable name.** The information stored in the memory area defined by a variable name can vary as BASIC commands or statements are executed (hence the name variable). The data currently stored in the memory area defined by a variable is known as the variable's **value.**

VARIABLE NAMES

BASIC allows variable names of up to 40 characters in length. A variable name must begin with a letter of the alphabet followed by additional alphanumeric characters. Blank spaces are not allowed within a variable name. Letters entered in lowercase will be automatically converted to uppercase by the computer. The following are examples of valid BASIC variable names:

```
ADDRESS     X9
JOHN        PHONE23
```

A variable name may not duplicate a BASIC reserved word (see appendix A). However, a variable name may incorporate a reserved word as part of its name.* Therefore, although the following would be invalid variable names,

```
NEW         AND         PRINT
```

* The exception to this rule is FN. A variable name cannot begin with FN.

the following variable names would be valid:

NEWPHONE ANDY PRINTNAME

Variables, like constants, can either be string or numeric. Numeric variables can be integer, single precision, or double precision. A variable type can be declared using a type identification character. The type identification characters are as follows:

% = integer
! = single precision
= double precision
$ = string

For example, the following variable names would be declared as string and integer respectively. If a variable type character is not specified, the variable is assumed to be a single precision value.

LOUISE$ REBEL%

ASSIGNMENT STATEMENTS

Numeric variables are initially assumed to have a value of zero. String variables are initially assumed to be null. Values may be assigned to a variable as the result of a calculation or as the result of an **assignment** statement. The LET statement is used to assign a value to a variable. LET is used with the following configuration:

LET *variable* = *expression**

Whenever a LET statement is used in a program, the value of the variable on the left side of the equation will be replaced with the value appearing on the right.

---

* In our configuration examples, BASIC reserved words will be depicted in uppercase, regular face type. Parameters to be entered by the programmer will be depicted in lowercase italics.

The reserved word, LET need not actually be included in a LET statement. Both of the following commands have the same meaning:

```
LET A = 5
  A = 5
```

The value assigned to a variable can either be a constant, a variable, or the result of an operation. In the following example, A$ is assigned the string constant "JOHN", B is assigned the numeric constant 27.9, C is assigned the value of B, and D is assigned the value of B multiplied by 2:

```
10 A$ = "JOHN"
20 B = 27.9
30 C = B
40 D = B * 2
50 PRINT A$
60 PRINT B
70 PRINT C
80 PRINT D
RUN
JOHN
 27.9
 27.9
 55.8
Ok
```

Expressions & Operators

The values of variables and constants are combined to form a new value through the use of **expressions.** The following are examples of expressions:

```
4 + 7
A$ + B$
3 * 1
14 < 21
X AND Y
```

BASIC includes several types of expressions including **arithmetic, relational,** and **Boolean.** In our previous examples, the first three examples were arithmetic expressions, while the fourth and fifth were examples of relational and Boolean expressions, respectively. Each of these types of expressions will be discussed in detail in the following sections.

The sign or word describing the operation to be undertaken is known as the **operator.** An operator is a symbol or word which represents an action which is to be undertaken on one or more values specified with the operator. These values are known as operands.

The operators in our previous examples were as follows:

```
+
+
*
<
AND
```

ARITHMETIC OPERATORS

Arithmetic operators are used to perform mathematical operations on numeric variables and constants. The various arithmetic operators are listed in table 4-1.

A number of these operations should already be familiar. The symbols + and - are used for addition and subtraction, respectively. The asterisk (*) is used to indicate multiplication, while the slash (/) is used to indicate division.

```
PRINT 5 + 3
 8
Ok
PRINT 24/8
 3
Ok
```

The first arithmetic operation specified in table 4-1 is **exponentiation.** Exponentiation (carat ^) is the process of raising a number to a specified power. For example, the following two expressions would be evaluated identically as 25. The exponent, 2, indicates the number of times that the base, 5, is to be multiplied by itself.

```
5^2    = 25
5 * 5  = 25
```

When the symbol - precedes a number, it changes that number's sign. This usage is known as **negation.**

The backslash (\) represents integer division. Integer division is identical to standard division except that no decimal places are returned. The MOD operator returns the remainder of an integer division.

```
PRINT 62\8
 7
Ok
PRINT 62 MOD 8
 6
Ok
```

TABLE 4-1. Arithmetic Operations

	Symbol	Operation	Example
	^	Exponentiation	A^B
	-	Negation	-A
same priority	*	Multiplication	A * B
same priority	/	Division	A/B
	\	Integer Division	A\B
	MOD	Remainder	A MOD B
same priority	+	Addition	A + B
same priority	-	Subtraction	A - B

ORDER OF EVALUATION (ARITHMETIC EXPRESSIONS)

All of our preceding examples were **simple expressions.** A simple expression is one which contains just one operator and one or two operands. Simple expressions can be combined to form **compound** expressions. The following are examples of compound expressions:

```
(-A) + 3 MOD 2
A + B * A / (C + D)
27 + 47 \ A^B
```

With compound expressions, it is necessary that the computer knows which operations should be undertaken first. BASIC follows a standard order of evaluation within compound expressions.

In this section, the order of evaluation of compound arithmetic expressions will be discussed. Later in this chapter, the order of evaluation of relational and logical operators will be discussed. Also, the relative evaluation priorities of these three groups will be outlined.

In an expression with more than one arithmetic operator, the operators with higher priority are evaluated first followed by those with lower priority. If two operators have the same priority, evaluation is performed from left to right in the expression. The operators in table 4-1 are listed in descending priority. For example, exponentiation is listed before multiplication, because exponentiation has a higher priority. Multiplication and division have the same priority. Also, addition and subtraction have the same priority. The following is an example of the evaluation of the arithmetic operators in an expression:

```
A = 37.1 + 12.9 * 2.1 - 7 + 4 ^ 2
  = 37.1 + 12.9 * 2.1 - 7 + 16
  = 37.1 + 27.09 - 7 + 16
  = 64.19 - 7 + 16
  = 57.19 + 16
  = 73.19
```

Parentheses can be used to alter the order of evaluation in arithmetic expressions. Expressions appearing within parentheses have the highest priority in the order of evaluation. For example, the use of parentheses with our preceding example could change the value of the expression.

```
A = (37.1 + 12.9) * 2.1 - (7 + 4^2)
  = 50 * 2.1 - (7 + 16)
  = 50 * 2.1 - 23
  = 105.0 - 23
  = 82.0
```

MIXING VARIABLE TYPES

Although certain variable types may be mixed in a BASIC expression, it is preferable to use a single variable type throughout each expression. By doing so, execution time will be decreased, memory require-

ments will be reduced, and the probability for program errors will be lessened.

BASIC does not allow a string value to be assigned to a numeric variable or vice versa. However, a numeric value in one precision can be assigned to a numeric variable with another precision.

When a numeric value is assigned to a variable of lower precision, the value is rounded to fit that variable. A single precision value is assigned to an integer in the following example. Notice that the value 1.75 is rounded to 2 in order to fit the integer variable.

```
10 X = 1.75
20 X% = X
30 PRINT X%
40 END
RUN
 2
Ok
```

Likewise, a numeric value can be assigned to a variable of higher precision. The following example demonstrates the result. Notice that although the single precision value, 2.04, was assigned to a double precision variable, the result was only accurate to 7 digits (single precision). This is because the original value 2.04 was only accurate to single precision.

```
10 X = 2.04
20 X# = X
30 PRINT X#
40 END
RUN
 2.039999961853027
Ok
```

RELATIONAL OPERATORS

Relational operators are used to make a comparison using two operands. The following relational operators are used in BASIC:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
=	equal to
<>	not equal

A relational operation evaluates to either true or false. For example, if the constant 1.5 was compared to the constant 2.4 to see whether they were equal, the expression would evaluate to false. In BASIC, a value of -1 represents a condition of true, while a value of 0 represents false.

The only values returned by a comparison in BASIC are -1 (true) or 0 (false). These values can be used as any other integer would be used. The following relational expressions and their results demonstrate comparisons:

```
5>7      0 (false)
5>3     -1 (true)
7=7     -1 (true)
```

Relational operations are evaluated after the arithmetic operations. Relational operations are performed from left to right in an expression.

Relational operations using numeric operations are fairly straightforward. However, relational operations using string values may prove confusing to the first time user. Strings are compared by taking the ASCII value for each character in the string one at a time and comparing the codes.

For example, consider the two string values "BONNIE" and "BECK". In a relational expression, the initial characters of the strings will be compared first. Since both strings begin with "B", the comparison will continue with the second character. Since the ASCII code for "E" (69) is less than the ASCII code for "O" (79), "BECK" is considered less than "BONNIE".

If the end of a string is encountered during a string comparison, the string with the fewer number of characters will be considered to be less than the longer string. For example, "ABC" would be evaluated as less than "ABCD". The relational operators can be used in this manner to indicate the relative location of strings in alphabetical order.

The following examples demonstrate the use of relational operators with string values. All of the following expressions are true:

```
"ABC" = "ABC"
"ABC" > "AAA"
"ALFRED" < "ALFREDO"
A$ < Z$ where A$ = "ALFRED" and Z$ = "ALFREDO"
```

Note that all string constants must be enclosed in quotation marks.

LOGICAL OPERATORS

Logical or Boolean operators are generally used in BASIC to compare the outcomes of two relational operations. Logical operations themselves return a true or false value which will be used to determine program flow.

The logical operators are NOT (logical complement), AND (conjunction), OR (disjunction), XOR (exclusive-or), IMP (implication), and EQV (equivalence). These return results as shown in figure 4-1.

A logical operator evaluates an input of one or more operands with true or false values. The logical operator evaluates these true or false values and returns a value of true or false itself. An operand of a logical operator is evaluated as true if it has a non-zero value. (Remember, relational operators return a value of –1 for a true value.) An operand of a logical operator is evaluated as false if it is equal to zero.

The result of a logical operation is also a number which if non-zero is considered true, and false if it is zero.

The following are examples of the use of logical operators in combination with relational operators in decision-making:

```
IF X > 10 OR Y < 0 THEN 900
IF A > 0 AND B > 0 THEN 200
FLAG% = NOT FLAG%
```

In the first example, the result of the logical operation will be true if the variable X is greater than 10 or if the variable Y is less than 0. Otherwise, it will be false. If the result of the logical operation is true, the program will branch to line 900. Otherwise, it will continue to the next statement. In the second example, the result of the logical operation will be true only if the value of both variables A and B are greater than zero. If the result of the logical operation is true, program control will branch to line 200. In the final example, the value of FLAG% is switched from true to false or vice versa.

FIGURE 4-1. Logical Operators

NOT Operation

T	F	A Operand
F	T	NOT A

OR Operation

T	T	F	F	A Operand
T	F	T	F	B Operand
T	T	T	F	A OR B

AND Operation

T	T	F	F	A Operand
T	F	T	F	B Operand
T	F	F	F	A AND B

XOR Operation

T	T	F	F	A Operand
T	F	T	F	B Operand
F	T	T	F	A XOR B

IMP Operation

T	T	F	F	A Operand
T	F	T	F	B Operand
T	F	T	T	A IMP B

FIGURE 4-1. (cont.) Logical Operators

EQV Operation

T	T	F	F	A Operand
T	F	T	F	B Operand
T	F	F	T	A EQV B

Logical operators actually convert their integer operands into their 16 bit binary equivalents before evaluating them. If an operand is negative, the two's complement is used to form that operand's 16 bit equivalent.

The binary equivalent of -1 may be found as follows. First, convert 1 into binary.

0000 0000 0000 0001

Then, complement every bit in the binary equivalent. This is known as the one's complement.

1111 1111 1111 1110_2

Finally, increment the one's complement to obtain the two's complement.

-1_{10} =1111 1111 1111 1111_2

If the 16 bit equivalent is used to evaluate a logical expression, true is represented by a bit value of one, while false is represented by a bit value of zero. This is consistent with our earlier assignment of true = -1, because the 16 bit representation of -1 consists of 16 one's (1111 1111 1111 $1111_2 = -1_{10}$).

The following example illustrates the use of the logical operator, XOR, with one positive and one negative argument.

```
PRINT -15 XOR 31
-18
Ok
```

-15_{10} = 1111 1111 1111 0001
31_{10} = 0000 0000 0001 1111
-15 XOR 31 = 1111 1111 1110 1110 = -18_{10}

Overall Order of Evaluation

In this chapter, we have outlined the use of the arithmetic, relational and logical operations. Table 4-2 outlines the overall order of evaluation of the operators.

TABLE 4-2. Order of Evaluation

Type	Operation	Symbol	Priority
Arithmetic	Exponentiation	^	1
	Negation	-	2
	Multiplication Division	* /	3
	Integer Division	\	4
	Remainder	MOD	5
	Addition Subtraction	+ -	6
Relational	Equality Inequality Less Than Greater Than Less Than or Equal Greater Than or Equal	= <> < > <= >=	7
Logical	Complement	NOT	8
	Conjunction	AND	9
	Disjunction	OR	10
	Exclusive-or	XOR	11
	Implication	IMP	12
	Equivalence	EQV	13

5

BASIC Programming Concepts

Introduction

In chapters 3 and 4, BASIC programming fundamentals were discussed. In this chapter, we will discuss some additional fundamental programming concepts. These include:

- data input and output
- conditionals, branching and loops
- tables and arrays
- functions and string handling
- program concatenation

Inputting and Outputting Data

Thus far, we have briefly described the usage of the PRINT statement to output data. Now, we will discuss the usage of PRINT to format

the outputted data. After we have discussed the methods used to output data, we will discuss the statements used to input data into variables. These include INPUT and INKEY$.

PRINT

To this point, we have only used the PRINT statement to output a single constant or variable value to the screen. The PRINT statement can also be used to output more than one item to the screen. When PRINT is used in this manner, the spacing between the items to be printed can be controlled by separating them with a comma or semicolon. For example, compare the results of the following PRINT statements:

```
PRINT "PAT" "MIKE" "KEN"
PATMIKEKEN
Ok

PRINT "PAT", "MIKE"
PAT           MIKE
Ok

PRINT "PAT";"MIKE";"KEN"
PATMIKEKEN
Ok
```

Notice that in our first example, no delimiter was used to separate the three string constants. These were output as one continuous string.

In the second example, the comma was used to delimit the string constants. When a comma appears in a PRINT statement, the computer is instructed to begin printing the next parameter in the PRINT statement at the beginning of the next print zone.

When the PC*jr* is configured in the 40 column mode (WIDTH 40), the screen is divided into two print zones. The first print zone extends from column 1 to column 14. The second extends from column 15 to column 40. When the PC*jr* is configured in the 80 column mode (WIDTH 80), the screen is divided into five print zones. These begin at columns 1, 15, 29, 43, 57.

Commas are very useful when data is to be output in tabular form. This is illustrated in the following example program:

```
10 PRINT "Name", "ID No."
20 PRINT "Jack Williams",3749
30 PRINT "Ann Timmons",3622
40 PRINT "Jay Randolph",2511
50 END
RUN
Name                    ID No.
Jack Williams           3749
Ann Timmons             3622
Jay Randolph            2511
Ok
```

In the third example, on page 90, the semicolon was used as the delimiter. The semicolon causes each string data item in the PRINT statement to be output immediately adjacent to the preceding item.

When semicolons are used to separate data items in a PRINT statement, the output will be displayed without the insertion of any additional spaces between data items. As a result, spaces must be inserted in PRINT statements between any data items that need to be separated. The most common technique used to insert spaces is to include a space (enclosed in quotation marks) in a PRINT statement. The following example program demonstrates this technique:

```
10 A$ = "IBM"
20 B$ = "PCjr"
30 PRINT A$;B$
40 PRINT A$; " ";B$
50 END
RUN
IBMPCjr
IBM  PCjr
Ok
```

Generally, when a PRINT statement has been executed, the cursor or print head will advance to the leftmost position on the next output line. This is known as a carriage return/line feed, which can be abbreviated as CR/LF.

A CR/LF can be suppressed by ending a PRINT statement with either a comma or a semicolon. When a semicolon is used to end a PRINT statement, the output from the next PRINT statement will be positioned immediately after the data output by its predecessor. This is illustrated in the following example:

```
10 PRINT "Data1";
20 PRINT "Data2";
30 PRINT "Data3"
40 END
RUN
Data1Data2Data3
Ok
```

When a PRINT statement ends with a comma, subsequent data will be output at the next zone on the same display line. This is shown in the following example:

```
10 PRINT "Data1",
20 PRINT "Data2"
30 END
RUN
Data1           Data2
Ok
```

HORIZONTAL FORMATTING

BASIC includes several functions that allow the programmer to control the horizontal format of output. These include TAB and SPACE$.

BASIC allows an item to be printed in any position on the screen or printer with the TAB command. The print position can range from 1 to 255. Notice that the following example causes output to be positioned at columns 10 and 20:

```
10 PRINT TAB(10) "10";TAB(20)"20"
RUN
          10        20
Ok
```

There are 255 print positions available because of the fact that a logical line in BASIC can consist of up to 255 characters. If the argument of a TAB function exceeds the length of the display line, the output will appear on a subsequent line.

The SPACE$ function causes the number of spaces specified as its argument to be output. In the following example, SPACE$(10) causes the cursor to move 10 positions to the right once BOB has been output:

```
10 PRINT "BOB";SPACE$(10);"LIGGETT"
RUN
BOB             LIGGETT
Ok
```

VERTICAL AND HORIZONTAL TABS

The output of a program can be tabulated by either rows or columns on the display. The LOCATE statement causes the cursor to move to a specified row and column. The following example program uses the tabulation feature to output data in the four corners of the screen:

```
10 CLS
20 PRINT "UPPER LEFT"
30 LOCATE 1,28
40 PRINT "UPPER RIGHT"
50 LOCATE 22,1
60 PRINT "LOWER LEFT"
70 LOCATE 22,28
80 PRINT "LOWER RIGHT"
90 END
```

The CLS statement at line 10 causes the screen to be cleared. As a result, the first output will occur in the upper-left corner of the display. The LOCATE statement at line 30 moves the cursor to column 28 of row 1. The second output will then occur in the upper right of the screen. Likewise, the LOCATE statements at lines 50 and 70 position the cursor to the lower right and lower left of the display. The row numbers extend from 1 (top) to 25 (bottom), while the column numbers extend from 1 (left) to 20, 40 or 80 (right). The correct range of column numbers depends on the selected screen width.

FORMATTING OUTPUT

BASIC allows the user to format numeric data when it is output to the screen or to the printer. Formatting is accomplished through the use of the PRINT USING statement and a **format string**. The format string contains the formatting characters enclosed in quotation marks. These characters determine the appearance of the numeric data when it is output.

The following statement,

```
100 PRINT USING "####.##";X
```

will result in the numeric value in variable X being output in decimal form with four digits to the left of the decimal point, and two digits to the right. The # sign in the format string is used to represent a digit, while the period is used to represent a decimal point.

If the following numeric data items were printed with the preceding statement's format string,

```
     7
8971.17
   18.5782
 999.77
1000.01
85712.11
```

the following would be output:

```
   7.00
8971.17
  18.58
 999.77
1000.01
%85712.11
```

Note that the final data item has a value greater than that allowed for by the format string. In these situations, the number will be displayed with a leading percent character (%).

A second formatting character, the dollar sign ($), is often used with the digit character (#) to display monetary figures. A single dollar sign in a format string causes the $ to be printed in that position in the output. If we assume the value of X to be 17.98, the following statement,

```
100 PRINT USING "$####.##";X
```

would result in the following output:

```
$ 17.98
```

The inclusion of two dollar signs ($$) in the format string is known as the floating dollar sign. The floating dollar sign results in a single dollar sign character being printed to the immediate left of the numeric value being output. Again, if we assume the value of X to be 17.98, the following statement,

```
100 PRINT USING "$$####.##";X
```

would result in the following output:

```
$17.98
```

The format string can be used to include commas when printing large numbers. If we assume the value of X to be 999999, the following statement,

```
100 PRINT USING "###,###.##";X
```

would result in the following output:

```
999,999.00
```

The format string can also be used to include a numeric value's sign in its output. By including the addition sign (+) at the beginning or end of the format string, the number's sign (+ or -) will be printed in the position specified. For example, if we assume the value of X to be -77.1, the following statement,

```
100 PRINT USING "+###.##";X
```

would result in the following output:

```
-77.10
```

The output could be altered by using the following format string:

```
100 PRINT USING "###.##+";X
```

The result would be 77.10-.

Another formatting character, leading asterisks (**), will cause all blank digit positions in a format string to be filled with asterisks. If we assume that X has a value of 58.29, the following statement,

```
100 PRINT USING "**####.##";X
```

would result in the following output:

```
****58.29
```

Another formatting character, the trailing minus sign (-), causes the number to be output with a trailing minus sign if that number has a negative value and no sign if it is positive. If we assume X's value to be -79, the following statement,

```
100 PRINT USING "####-";X
```

would output the following:

```
79-
```

OUTPUTTING DATA TO THE PRINTER

The printer can be accessed as the output device by using LPRINT or LPRINT USING. The LPRINT statement functions as does the PRINT statement, except that LPRINT sends data to the printer rather than the screen. A program listing can be sent to the printer by using LLIST.

LINE WIDTH

The WIDTH command allows the user to change the line width of the video display or printer.

The width of the video display can be set to 20, 40, or 80 characters. For example, to set the screen width to 20 columns, the following statement would be used:

```
WIDTH 20
```

Likewise, WIDTH 40 and WIDTH 80 would set screen width to 40 columns and 80 columns, respectively.

The LPRINT and LPRINT USING statements assume a printer width of 80 characters. However, this width can be changed by executing a WIDTH "LPT1:" statement. LPT1: is the device name for the printer. The following statement sets the printer width to 130 characters:

```
WIDTH "LPT1:",130
```

If a line being sent to the video display consists of a greater number of characters than the width of the screen, the extra characters will "wrap around" to the next line. Likewise, if a line being output to the printer contains a greater number of characters than allowed for, the excess characters will be printed on successive lines.

INPUT

When an INPUT statement is executed, the computer will display a question mark and wait for the operator to enter a response. That entry will be assigned to the variable indicated. The entry must be ended by pressing the Enter key. Program execution will then resume.

The values of several variables can be input with a single INPUT statement. These variables may either be numeric or string as shown in the following example:

```
100 INPUT A$,B$,C
```

When the preceding INPUT statement is executed, the INPUT prompt (?) will be displayed. The operator should then input the data items for variables A$, B$, and C. Each input should be separated by a comma. The Enter key should be pressed after all input entries have been made. An example of a valid entry for the preceding INPUT statement is given below:

```
JOHN,SMITH,281
```

These entries will be assigned to the variables as follows:

```
A$ = "JOHN"
B$ = "SMITH"
 C = 281
```

If an incorrect number of entries were made or if a string constant were input for a numeric variable or vice versa, the following message would be displayed,

```
? Redo from start
?
```

and the computer would wait for a valid entry.

It is a good programming practice to include a prompt message with the INPUT statement to let the operator know what data the computer is expecting. The prompt message is a string constant that will be displayed when the computer expects an input. For example, the following INPUT statement,

```
100 INPUT "ENTER NAME, PHONE";A$,B$
```

will result in the prompt being displayed. The PC*jr* will then wait for a keyboard entry. The operator enters the underlined output in the following example:

```
ENTER NAME,PHONE? MARY, 845-2956 (←┘)
```

These entries will be assigned to the variables A$ and B$. A$ will be assigned a value of "MARY", and B$ will be assigned a value of "845-2956". Notice that the string entries did not need to be enclosed in quotation marks. When a string is entered in response to an INPUT prompt, the quotation marks can be excluded, unless the entry includes commas or begins with spaces.

LINE INPUT

The LINE INPUT statement is a variation of the INPUT statement. LINE INPUT allows an entire line (maximum of 254 characters) to be input to a string variable. Unlike INPUT, LINE INPUT does not recognize spaces and commas as delimiters. Therefore, it is more simple to include these characters in an inputted string. Pressing the Enter key ends the input process.

LINE INPUT [;]["*prompt*";]*variable$*

If the optional semi-colon is included after LINE INPUT, a CR/LF to the screen will not occur when the Enter key is pressed.

INKEY$

An INKEY$ statement is an alternate means of inputting data from the keyboard. INKEY$ can be used to accept a single character, and assign that character to a string variable. However, unlike INPUT or LINE INPUT, if no character was typed when INKEY$ was executed, the PC*jr* will not wait for a keyboard response. The following program illustrates INKEY$:

```
10 A$ = INKEY$
20 PRINT A$
30 GOTO 10
```

Line 10 assigns the string value of the currently pressed key to A$. If no keys have been pressed, the null string will be assigned to A$. Line 20 prints the string, while line 30 transfers program control back to line 10. The GOTO statement will be discussed in the next section of this chapter.

Conditionals, Branching and Looping

Thus far in our discussion of PC*jr* BASIC, program statements have been executed in sequential order. Several BASIC statements are available that can be used to alter program control. These include:

IF-THEN-ELSE	ON-GOTO
GOTO	ON-GOSUB
GOSUB	ON ERROR GOTO
FOR-NEXT	WHILE-WEND

These statements will be discussed in the following sections.

CONDITIONALS

One of the most important features of a computer is its ability to make a decision. BASIC uses the IF-THEN statement to take advantage

of the computer's decision making ability. The IF-THEN statement takes the following form:

IF *expression* THEN *statement*

The IF statement sets up a decision. If *expression* evaluates to true, then *statement* will be executed. If *expression* evaluates to false, the subsequent program statement will be executed. In the following example, if AGE is greater than or equal to 21, "LEGAL" will be printed.

```
IF AGE >= 21 THEN PRINT "LEGAL"
```

An ELSE statement may be added to an IF-THEN statement to achieve an IF-THEN-ELSE structure. IF-THEN-ELSE uses the following configuration:

IF *expression* THEN $statement_1$ ELSE $statement_2$

If *expression* evaluates to true, then $statement_1$ will be executed. Otherwise $statement_2$ will be executed. In other words, a true *expression* causes $statement_1$ to be executed, while a false *expression* causes $statement_2$ to be executed. In the following example, if SCORE is greater than 90000, then the PC*jr* will print "NICE GAME". Otherwise, it will print "NOVICE".

```
IF SCORE > 90000 THEN PRINT "NICE GAME"
                 ELSE PRINT "NOVICE"
```

BRANCHING STATEMENTS

Branching statements change the execution pattern of programs from their usual line by line execution. A branching statement allows program control to be altered to any line number desired. The most commonly used branching statements in BASIC are GOTO and GOSUB.

GOTO takes the following format:

GOTO *line number*

For example, the following program statement,

```
500 GOTO 999
 .
 .
 .
999 END
```

would branch program control from line 500 to 999.

SUBROUTINES AND GOSUB

Many times you will find that the same set of program instructions are used more than once in a program. Re-entering these instructions throughout the program can be very time consuming. By using **subroutines,** these additional entries will be unnecessary.

A subroutine can be defined as a program which appears within another larger program. The subroutine may be executed as many times as desired.

The execution of subroutines is controlled by the GOSUB and RETURN statements. The format for the GOSUB statement is as follows:

GOSUB *line number*

The computer will begin execution of the subroutine beginning at the *line number* indicated. Statements will continue to be executed in order, until a RETURN statement is encountered. Upon execution of the RETURN statement, the computer will branch out of the subroutine back to the first line following the original GOSUB statement. This is illustrated in the following example:

```
10  GOSUB 100
20  GOSUB 200
30  END
100 PRINT "subroutine #1"
110 RETURN
200 PRINT "subroutine #2"
210 RETURN
RUN
subroutine #1
subroutine #2
Ok
```

Subroutines can help the programmer organize his program more efficiently. Subroutines also can make writing a program easier. By dividing a lengthy program into a number of smaller subroutines, the complexity of the program will be reduced. Individual subroutines are smaller and therefore, more easily written. Subroutines are also more easily debugged than a longer program.

CONDITIONAL STATEMENTS WITH BRANCHING

Branching statements are often used in conjunction with conditional statements. In such a situation, the normal execution of the program will be altered depending upon the outcome of the condition set up in an IF or an ON statement. This is shown in the following example:

```
100 INPUT "Enter the amount";A
200 IF A = 0 THEN GOTO 500
300 PRINT A
400 GOTO 100
500 INPUT "Are you finished?";A$
600 IF A$ <> "y" THEN 100
700 END
```

In our preceding example, if the value input for A has a zero value, then the program will branch to line 500 where the operator will be asked whether he has finished entering data. In line 600, the program will set up a condition where if the input was anything other than the letter "y", the program will branch to line 100. If the entry was equal to y, the program will end at line 700.

Note in line 600 that a GOTO statement is not used to precede the line number being branched to. When a line number is indicated following a THEN statement, the computer assumes the presence of GOTO.

The ON-GOTO and ON-GOSUB statements are also combinations of a conditional statement and a branching statement. The use of the ON-GOTO statement is illustrated in the following program:

```
100 INPUT A
200 ON A GOTO 400, 500
300 GOTO 999
400 PRINT "A = 1":GOTO 999
500 PRINT "A = 2"
999 END
```

If the variable or expression following ON evaluates to 1, program control will branch to the first line number specified after GOTO; if 2, to the second, etc.

If the variable or expression evaluates to a number greater than the number of line numbers following GOTO, program control will branch to the statement immediately following the ON-GOTO statement. This is also the case if the variable or expression following ON evaluates to zero. Negative values for the control expression are not allowed.

The ON-GOSUB statement is very similar in nature to the ON-GOTO statement. The following statement is an example of an ON-GOSUB statement:

```
100 ON X GOSUB 1000,2000,3000
```

If the value of X is 1, the subroutine at line 1000 will be executed. If X is 2, the subroutine at line 2000 will be executed. If X is 3, the subroutine at line 3000 will be executed. If X evaluates to 0 or to a number greater than 3, the statement immediately following the ON-GOSUB will be executed.

If ON-GOSUB causes a branch to a subroutine, program control will revert to the line immediately following the ON-GOSUB statement, once the subroutine has been executed.

LOOPING STATEMENTS

Suppose that you needed to compute the square of the integers from 1 to 20. One way of doing this is by calculating the square for each individual integer as shown below:

```
100 A = 1^2
200 PRINT A
300 B = 2^2
400 PRINT B
500 C = 3^2
600 PRINT C
    .
    .
    .
```

This method is very cumbersome. The problem could be solved

much more efficiently through the use of a FOR-NEXT loop as shown below:

```
100 FOR A = 1 TO 20
200   X = A^2
300   PRINT X
400 NEXT A
500 END
```

The sequence of statements from line 100 to 400 is known as a **loop.** When the computer encounters the FOR statement in line 100, the variable A will be set to 1. X will then be calculated and displayed in lines 200 and 300.

The NEXT statement in line 400 will request the next value for A. Execution returns to line 100 where the value of A will be incremented (to 2) and then compared to the value appearing after TO. Since the value of A is less than that value, the loop will be executed again with the value of A set at 2. The loop will continue to be executed until A attains a value greater than 20. When this occurs, the statement following the NEXT statement will be executed.

In our preceding example, A is known as an **index variable.** If the optional keyword STEP is not included with the FOR statement, the index variable will be increased by 1 every time the NEXT statement is executed.

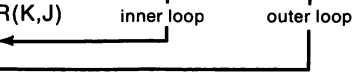
STEP can be included at the end of a FOR statement to change the value by which the index variable is increased. The integer appearing after STEP is the new increment. For example, if our preceding example were changed as follows,

```
100 FOR A = 1 TO 20 STEP 2
200   X = A^2
300   PRINT X
400 NEXT A
500 END
```

the index variable, A, would be increased by 2 every time the NEXT statement was executed.

One loop can be placed inside another loop. The innermost loop is known as a **nested** loop. The following program contains a nested loop:

```
100 DIM R(2,3)
200 DATA 10,20,30,40,50,60
300 FOR K = 1 TO 2
400   FOR J = 1 TO 3
500     READ R(K,J)
600   NEXT J
700 NEXT K
```



Our preceding example is used to read data into the numeric array R. Arrays, as well as the READ and DATA statements, will be discussed in detail later in this chapter.

Be certain that any integer loop is ended prior to ending its outer loop. Also, be certain that every NEXT statement has a matching FOR statement. If the BASIC interpreter cannot match every NEXT statement with a preceding FOR statement, an error will result.

CONDITIONAL LOOPING STATEMENTS

Suppose that you wanted to execute a series of statements in a loop as long as a given condition remained true. An example would be reading from a data file, until a desired item was reached. Another example would be doing an iterative approximation of a mathematical function.

The WHILE-WEND statement pair forms a loop similar to the one formed by the FOR-NEXT statements. As long as the expression following the keyword WHILE evaluates to a logical value of true (-1), the statements between WHILE and WEND will be continuously reexecuted.

WHILE-WEND loops also may be nested just as with FOR-NEXT loops. Again, be careful not to cross nest WHILE-WEND loops. Since it is not always clear which WEND statement is associated with a corresponding WHILE statement, try to clarify this by using the same indentation for associated WHILE-WEND statements. REM statements may also be used to indicate which WHILE-WEND statements are associated.

```
10 I =0
20 WHILE I < 10
30   PRINT I
40   I = I + 1
50 WEND
60 END
RUN
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
Ok
```

In the previous example, the value of I is continually incremented in the WHILE-WEND loop, until the value of I is no longer less than 10.

The WHILE-WEND loop in the previous example did not offer any gains over the FOR-NEXT loop structure. In fact, a FOR-NEXT loop with three statements could have duplicated the results of the previous example.

```
10 FOR I = 0 TO 9
20   PRINT I
30 NEXT I
40 END
```

The real beauty of the WHILE-WEND loop structure is evidenced when the programmer is uncertain as to how many times the loop must be executed. Suppose that one wished to obtain a root of the following polynomial. A root is another name for the solution.

$$X^3 - 3X^2 + 1 = 0$$

The algebraic expression can manipulated into the following form:

$$X = \frac{-1}{X^2-3X}$$

We will now instruct the computer how to numerically approximate the solution by **iterations.** If a value is plugged into the right-hand side of the previous equation, a value will be obtained that is closer to the root than the original guess. If this new value is then plugged into the right-hand side of the equation, a value even closer to the answer will be obtained. If this process is repeated until the last two obtained values are very close to each other, these two values will be very close to the root.

```
100 XNEW = 2
110 WHILE ABS(X-XNEW) > 0.0000001
120     X = XNEW
130     XNEW = (-1)/(X^ 2-3*X)
140 WEND
150 PRINT X;"is the ROOT"
160 END
```

The condition for the loop to continue is that the last two values are farther apart than 0.0000001. ABS(XNEW-X) is merely their separation. ABS is a function that computes the absolute value. Functions will be discussed later in the chapter. Line 130 is the computer's representation of the second algebraic expression.

ERROR HANDLING

In some situations, it is easier to correct problems as they occur in a program, rather than to avoid them. This technique is called **error handling**. BASIC allows the use of an ON ERROR GOTO statement to specify a line number where the program should proceed if an error occurs. This feature allows a portion of the program to be set aside as an error handling routine.

Error handling routines are commonly used to correct small problems that occur infrequently in a program. When appropriate corrections have been performed, a RESUME statement can be used to branch the program back to the location where the error occurred. The function of a RESUME statement is analogous to the use of a RETURN statement at the end of a subroutine.

The following program demonstrates the technique used to branch a program in event of an error:

```
10  ON ERROR GOTO 100
20  INPUT "INPUT X:";X
30  Y = X ^ .5
40  PRINT "The square root of ";X;"is";Y
50  END
100 Y = (-X) ^ .5
110 PRINT "The square root of ";X";is";Y"i"
120 END
```

The preceding example program contains an ON ERROR GOTO statement at line 10. This statement indicates that the program control will branch to line 100 in the event of an error. An ON ERROR GOTO statement must be executed in a program before an error actually occurs.

The program calculates the square root of a value input for the variable X. However, BASIC does not allow the square root of negative numbers. These values can only be defined in the context of complex numbers, where the symbol "i" is used to represent the square root of -1. As a result, the square root of -4 could be represented by the value 2i since the following expression is true:

$$\sqrt{-4} = \sqrt{4}\,\sqrt{-1} = \sqrt{4i} = 2i$$

It is not necessary to understand the use of "complex" numbers to comprehend the example. The main concept of the program is:

> The statement at line 30 would normally have caused an error if a negative value had been input for the variable X. However, in this case, the ON ERROR GOTO statement causes the program to branch to line 100 whenever an error occurs.

Lines 100 and 110 perform an alternate set of operations whenever a negative value is input for X. Some typical applications of the sample program would appear as follows:

```
RUN
INPUT X: 4 <------ user's response
The square root of 4 is 2
Ok
RUN
INPUT X: -16 <------ user's response
The square root of -16 is 4i
Ok
```

ERR and ERL are used to return the error code and the line number associated with an error. ERR returns the error code for the last error. ERL returns the line number of the line where the error was discovered. The ERL and ERR variables are often used in conjunction with an IF-THEN statement to test for an error as shown in the following example:

```
100 ON ERROR GOTO 999
 .
 .
 .
999 IF ERR = 27 THEN LOCATE 1,1:
    PRINT "Please Turn on Printer";:RESUME
```

Appendix F contains the BASIC error messages along with their corresponding error numbers and description of the errors.

Tables and Arrays

In chapter 4 we introduced the concept of variables. A variable is designed to hold a single data item — either string of numeric. However, some programs require that hundreds or even thousands of variable names be used.

The processing of large quantities of data can be greatly facilitated through the use of arrays and tables in a program.

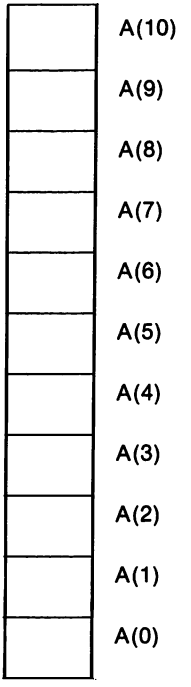
SUBSCRIPTED VARIABLES

Obviously, the use of thousands of individual names could prove extremely cumbersome. To overcome this problem, BASIC allows the use of **subscripted variables**. Subscripted variables are identified with a **subscript**, a number appearing within parentheses immediately after the variable name. An example of a group of subscripted variables is given below:

```
A(0),A(1),A(2),A(3),A(4),etc.
```

Note that each subscripted variable is a unique variable. In other words, A(0) differs from A(1), A(2), A(3), etc...

Subscripted variables may be visualized as an **array** (or **table**). In our previous example, the data contained in the array defined by A would consists of a one-dimensional array with 11 elements.



Arrays can also have two or more dimensions. Two-dimensional arrays are also known as tables. A table containing 6 rows and 8 columns is depicted below:

		Columns							
		0	1	2	3	4	5	6	7
	0	A(0,0)	A(0,1)	A(0,2)	A(0,3)	A(0,4)	A(0,5)	A(0,6)	A(0,7)
	1	A(1,0)	A(1,1)	A(1,2)	A(1,3)	A(1,4)	A(1,5)	A(1,6)	A(1,7)
Rows	2	A(2,0)	A(2,1)	A(2,2)	A(2,3)	A(2,4)	A(2,5)	A(2,6)	A(2,7)
	3	A(3,0)	A(3,1)	A(3,2)	A(3,3)	A(3,4)	A(3,5)	A(3,6)	A(3,7)
	4	A(4,0)	A(4,1)	A(4,2)	A(4,3)	A(4,4)	A(4,5)	A(4,6)	A(4,7)
	5	A(5,0)	A(5,1)	A(5,2)	A(5,3)	A(5,4)	A(5,5)	A(5,6)	A(5,7)

Notice from our illustration that a position within the table is identified with a subscripted variable. The subscript contains two numbers. The first number identifies the row number and the second identifies the column. For instance, A(1,2) identifies the element located in column two of row one.

Array variables can be assigned values and used with operators as can any other variable. This is illustrated in the following example:

```
10 A(0) = 5
20 A(1) = 6
30 A(2) = 7
40 A(3) = 8
50 A(4) = 9
60 PRINT A(0) * A(1)
70 A(5) = A(2) + A(4)
80 PRINT A(5)
90 END
RUN
 30
 16
Ok
```

DIMENSIONING AN ARRAY

Before an array variable can be used in a program, an area in memory must be reserved to store its elements. This is known as dimensioning the array and is accomplished with the DIM statement.

The DIM statement defines the maximum subscript value that can be used for an array. For example, the following DIM statement:

```
DIM A(20)
```

would define a one-dimensional array consisting of twenty-one elements ranging from A(0) to A(20) inclusive.

Two dimensional arrays are dimensioned as follows:

```
DIM A(4,7)
```

The preceding DIM statement would dimension an array consisting of five rows with eight columns each.

Notice that a DIM statement was not included in our first example.

When a subscripted variable which has not been previously dimensioned is referenced in a program, the array variable is automatically dimensioned with a maximum subscript value of 10. If we added the following program line to our example:

```
85 A(11) = 24:PRINT A(11) * A(1)
```

the following error message would be displayed:

```
Subscript out of range in 85
```

This error is generated because an array variable was referenced with a subscript greater than originally stated.

If the following DIM statement was inserted in our example program,

```
5 DIM A(11)
```

it would execute properly, because A(11) would have been defined by the DIM statement.

Generally, it is good programming practice to dimension all array variables and to group all DIM statements at the beginning of the program. This prevents an array variable from inadvertently being referenced before it has been dimensioned.

When an array is no longer needed in a program, the DIM statement can be reversed with a CLEAR statement. This will free the memory area previously reserved for the array. This is illustrated in the following program:

```
10 PRINT FRE(0)
20 DIM A(50,50)
30 PRINT FRE(0)
40 CLEAR
50 PRINT FRE(0)
60 END
RUN
 59631
 49216
 59631
Ok
```

In line 10, the number of available bytes in memory is displayed. FRE is a function which displays the available free bytes in memory. FRE is explained in more detail, later in this chapter.

In line 20, the DIM statement reserves an area in memory for a table consisting of 2601 elements. From line 30, it is evident that the number of free bytes has decreased substantially. This is due to the fact that an area of memory has been reserved for the elements in table A.

In line 40, the CLEAR statement reverses the DIM statement and frees the memory previously required for the elements in table A. The CLEAR statement is also a memory management command. CLEAR has three optional parameters.

CLEAR [*memory*] [,[*stack*] [,*screen*]]

memory is a byte count which sets the maximum BASIC workspace. *memory* can be set to reserve memory for machine language subroutines. *stack* sets aside the specified number of bytes for stack usage (default = 512 bytes). It is a good practice to increase the size of the stack if a program includes a large number of FOR-NEXT loops or subroutines.

screen specifies the total memory that is reserved for screen usage (default = 16384 bytes). In order to execute a SCREEN 5 or SCREEN 6 statement, 32768 bytes of memory must be set aside for the screen.

DATA & READ STATEMENTS

Earlier, we discussed how data could be assigned to a variable with a LET statement as well as how data could be input directly from the keyboard and assigned to a variable with an INPUT or INKEY$ statement. However, none of these statements are practical for assigning data values to the individual variables in a large array or table. DATA and READ statements are much more practical for assigning values to variables in an array. DATA and READ statements can be used for assigning values to any variable — not just array variables.

A typical DATA statement is shown below:

```
100 DATA "WILLIAMS",27,"ST.LOUIS","314-727-1141"
```

Notice that this DATA statement contains four data items, three of

which are string, and one of which is numeric. In our example, we have enclosed the string data items in quotation marks. However, this was not actually required. In a DATA statement, a string only needs to be enclosed in quotation marks if it contains a comma, a colon, or if its first character is a blank space.

DATA statements are used in conjunction with READ statements to assign data values to variables. An example of a READ statement is given below:

```
200 READ NAME$,AGE,CITY$,PHONE$
```

When a READ statement is executed, the computer will first search for a DATA statement. When a DATA statement is found, the values in the DATA statement will be assigned one-by-one to the variables in the READ statement.

If the first DATA statement encountered does not have enough data items to be assigned to all the variables in the READ statement, the next DATA statement will be searched for. The values from this and succeeding DATA statements will continue to be assigned to the variables in the READ statement until all of the variables in the READ statement have been assigned a value.

The computer keeps track of the next DATA statement data item to be used via an internal pointer. When any future READ statements are executed, this pointer will determine which is the next data item to be read into the READ variable.

BASIC includes a statement known as RESTORE, which when executed, sets the DATA item pointer back to the beginning of the DATA statement list. The use of the DATA item pointer and the effect of RESTORE on it is depicted in figure 5-1.

The RESTORE statement may be used with a line number following the reserved word. When RESTORE is used in this manner, the DATA item pointer is set to the first item of the DATA statement in that line. For example, if line 400 in our example had been the following,

```
400 RESTORE 110
```

the READ statement in line 500 would have assigned the value 27 to the variable X.

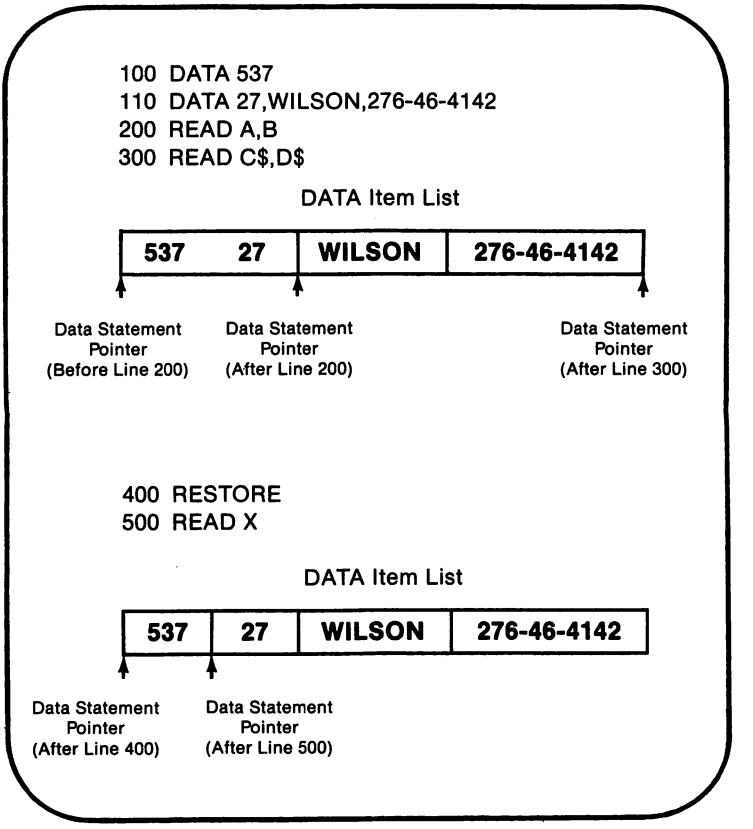


Figure 5-1. DATA Statement Pointer

When not properly used, DATA and READ statements can be the source of program errors. One potential error source occurs when the program attempts to READ more data items than were given in the DATA statements. Such an error would occur in the following program:

```
100 DATA 7,8,11,13,15
200 FOR K = 1 TO 7
300   READ X(K)
400   PRINT X(K)
500 NEXT K
600 END
```

In the preceding example, the program would attempt to read 7 data items. However, since the DATA statement only contained 5 data items, the following error message would appear:

```
Out of Data in 300
```

Another potential source of error when executing DATA and READ statements are situations where the program attempts to read a numeric data item into a string variable or vice versa. If such an error is encountered, the following message will be displayed:

```
Syntax error
```

DATA and READ statements are often used in conjunction with FOR-NEXT loops to read large amounts of data into arrays. An example of this use of FOR-NEXT is given below:

```
10  FOR K = 0 TO 5
20    READ NAMES$(K)
30    READ AGE(K)
40  NEXT K
50  FOR J = 0 TO 5
60    PRINT NAMES$(J),AGE(J)
70  NEXT J
80  END
90  DATA Jim,10
100 DATA Tom,11
110 DATA Matt,9
120 DATA Eric,10
130 DATA Steve,10
140 DATA Joe,9
Run
Jim                    10
Tom                    11
Matt                   9
Eric                   10
Steve                  10
Joe                    9
Ok
```

An example of the use of the READ and DATA statements in conjunction with a FOR-NEXT loop for the purpose of reading data into a two-dimensional array is given in the following program:

```
10  DATA 10,20,30,40
20  DATA 50,60,70,80
30  DATA 90,10,20,30
40  FOR J = 0 TO 2
50    FOR K = 0 TO 3
60      READ A(J,K)
70      PRINT A(J,K);" ";
80    NEXT K
90    PRINT
100 NEXT J
110 END
RUN
 10   20   30   40
 50   60   70   80
 90   10   20   30
Ok
```

The preceding program would read data items into table A( ) as shown in the following illustration:

		columns			
		0	1	2	3
	0	10	20	30	40
Rows	1	50	60	70	80
	2	90	10	20	30

Functions and String Handling

In mathematics, a function is generally defined as a quantity whose value will vary as a result of another quantity. In computing, functions define operations that are performed on strings or numeric values.

In BASIC, a number of functions are already defined by reserved words and are a part of the BASIC interpreter. These are known as **built-in** functions (see table 5-1). Built-in functions cover a wide range of standard math operations such as absolute value, square root, logarithms, etc. Built-in functions are also available for working with strings,

as well as a variety of other operations.

BASIC also allows the programmer to define his or her own functions. These are known as **user-defined** functions. Both built-in and user-defined functions will be discussed in this section.

BUILT-IN MATHEMATICAL FUNCTIONS

The majority of BASIC functions are used in mathematical applications. We provide an overview of BASIC's math functions in this section. Each individual function will be described at the end of the chapter.

Table 5-1. BASIC Built-in Functions

ABS	ERR	LPOS	SPC
ASC	EXP	MID$	SQR
ATN	FIX	OCT$	STICK
CDBL	FRE	PEEK	STR$
CHR$	HEX$	PEN	STRIG
CINT	INKEY$	PLAY	STRING$
COS	INP	PMAP	USR
CSNG	INPUT$	POINT	TAB
CSRLIN	INSTR	POS	TAN
CVD	INT	RIGHT$	TIME$
CVI	LEFT$	RND	TIMER
CVS	LEN	SCREEN	VAL
DATE$	LOC	SGN	VARPTER
EOF	LOF	SIN	VARPTER$
ERL	LOG	SPACE$	

All of the BASIC mathematical functions operate in much the same manner. Each function is defined by a reserved word (ex. SIN for Sine, COS for Cosine, LOG for Logarithm etc.).

A numeric constant, variable, or expression may appear in parentheses following the reserved word which identifies the function. The function for that numeric value will then be calculated by the computer. The use of several mathematical functions is shown in figure 5-2.

BASIC includes the following three trigonometric functions:

```
SIN(N) = sine of the angle N.
COS(N) = cosine of the angle N.
TAN(N) = tangent of the angle N.
```

The angle N must be given in terms of radians. One radian is the equivalent of 57.29578 degrees. One degree equals .017453 radians. Therefore, the following can be used to calculate a trigonometric function with its argument (X) given in degrees:

```
SIN(.017453 * X)
COS(.017453 * X)
TAN(.017453 * X)
```

```
 100 PRINT SIN(.47)
 200 PRINT COS(.98)
 300 PRINT TAN(.37)
 400 PRINT SQR(49)
 500 PRINT INT(5.79)
 600 PRINT INT(-5.79)
 700 PRINT ABS(-4.7)
 800 PRINT SGN(2.7)
 900 PRINT SGN(-2.7)
1000 END
RUN
 .4528863
 .5570226
 .3878632
 7
 5
 -6
 4.7
 1
 -1
Ok
```

FIGURE 5-2. Mathematical Functions

The other three principle trigonometric functions: secant, cosecant, and cotangent can be computed by using SIN, COS, and TAN as shown in the following identities:

```
SEC(X) = 1/COS(X)
CSC(X) = 1/SIN(X)
COT(X) = 1/TAN(X)
```

BASIC also includes the arctangent function ATN. This function returns the angle (expressed in radians) whose tangent is given in its argument.

```
ATN(X) = angle in radians whose tangent equals X
```

The following formula can be used to calculate the angle expressed in degrees (rather than radians) whose tangent is given in X:

```
57.29578 * ATN(X)
```

BASIC also contains functions for calculating natural logarithms and exponentials. The exponential formula takes the following form:

```
A = EXP(B)
```

The preceding EXP function is calculated by computing the value of **e** raised to the B power. **e** is known as the base of natural logarithms. The value **e** in BASIC is 2.718281828459.

The natural logarithm of a number may be calculated with the LOG function.

```
LOG(X) = natural logarithm of X
```

Logarithms with a base other than **e** may be calculated using the following formula:

$$LOG_b(X) = LOG(X)/LOG(b)$$

where b is the base of the logarithm.

BASIC includes the SQR function for determining the positive square root of its argument.

SQR(X) = positive square root of X

The square root of a number can also be calculated with the exponential arithmetic operator. The following expression,

X ^ (1/2)

will calculate the square root of X. The arithmetic exponential operator can also be used to calculate a root other than the square root (ex. cube root) as shown below:

X ^ (1/3)

BASIC also includes several functions that can be used in working with numeric values. These include INT, ABS, and SGN. The INT function returns the integer with the greatest value which is less than or equal to its argument. INT takes the following form:

INT(X) = highest integer whose value
is less than or equal to X

Figure 5-2 contains examples of the usage of the INT function.

The ABS function returns the absolute value of its argument. ABS takes the following form:

ABS(X) = | x |

An example of the use of ABS appears in figure 5-2.

The SGN function returns the sign of its argument. An example of the use of SGN appears in figure 5-2.

USER-DEFINED FUNCTIONS

In the preceding section, we discussed a number of predefined BASIC functions. BASIC also allows the user to define his own functions. These are known as **user-defined** functions. A user-defined function must be defined with the DEF FN statement before it can be used in the program.

For example, the following DEF FN statement would define a

function in which the argument was squared, and 1 was then subtracted from that calculation:

```
100 DEF FN A(X) = X ^ 2-1
```

The name of the function (FN A) appears immediately following the DEF statement. Any valid variable name may be used as a user-defined function name. The following would be a valid function name:

```
FN TANH
```

In our first example, notice the X in parentheses following the function name. This is known as a **dummy argument**. Any valid variable name can be substituted for X as the dummy argument.

When the user-defined function is called in the program, the argument supplied with the function when it is called will be substituted for the dummy argument whenever it appears on the right-hand side of the DEF FN statement. The expression will then be evaluated, and the value returned as the value of the function.

```
10 M = 3.9878
20 DEF FN S(X) = COS(X) + SIN(X)
30 PRINT FN S(M)
RUN
-1.4116
Ok
```

The previous example contains a program that has a DEF FN statement at line 20. The function is assigned the name S, and the dummy argument X is used in the function. The operations in the function (COS(X) + SIN(X)) can be as complicated as necessary. At line 30, the S function is evaluated as the value of the variable M. The function substitutes 3.9878 for the dummy argument X and returns a numeric value that is displayed by the PRINT statement.

STRINGS & STRING HANDLING

As a programmer, you will encounter a number of situations where you may need to work with string data. For example, you might want to combine several strings, compare two strings, separate portions of a

string, or even convert string data to its numeric equivalent. BASIC allows for all of these.

STRING CONCATENATION

The process of joining together one or more strings is known as **concatenation**. The arithmetic operator for addition (+) is used for string concatenation. However, concatenation is very different from addition. In concatentation, the strings being concatenated are joined to form a new string as shown below:

```
100 A$ = "JOHN"
200 B$ = "SON"
300 C$ = A$ + B$
400 PRINT C$
500 END
RUN
JOHNSON
Ok
```

Either string constants or variables may be concatenated. Any number of strings may be concatenated as long as the resulting string contains 255 or fewer characters.

STRING HANDLING FUNCTIONS

BASIC contains a number of strings handling functions which allow the user to extract a part of a string. These functions are LEFT$, MID$ and RIGHT$.

The LEFT$ function takes the following format:

```
LEFT$(A$,X)
```

A$ represents a string constant or string expression on which the operation is to be performed, and X represents the number of characters to be extracted. The LEFT$ function will extract the number of characters given in X from the left-hand side of the string given in A$. Figure 5-3 contains an example of the use of LEFT$.

```
100 A$ = "JOHNSON"
200 B$ = LEFT$(A$,4)
300 PRINT B$
400 END
RUN
JOHN
Ok
```

FIGURE 5-3. LEFT$

The RIGHT$ function works exactly like the LEFT$ function except that the number of characters specified are returned from the string's right-hand side. Figure 5-4 contains an example of the use of RIGHT$.

```
100 A$ = "JOHNSON"
200 B$ = RIGHT$(A$,3)
300 PRINT B$
400 END
RUN
SON
Ok
```

FIGURE 5-4. RIGHT$

The MID$ function can be used to return a portion of a string. MID$ takes the following format:

$$a\$ = \text{MID}\$(b\$,x[,y])$$

The string being returned is *a*$. *a*$ is being returned from *b*$. The string being returned will begin with the *x*th character in *b*$. The number of characters returned from *b*$ is specified in *y*. *y* is an optional parameter. If *y* is omitted, all rightmost characters in *b*$ will be returned in a$. An example of the use of MID$ to return a portion of a string is given in figure 5-5.

```
100 X$ = "NEW CASTLE"
200 Y$ = MID$(X$,5,4)
300 PRINT Y$
400 END
RUN
CAST
Ok
```

FIGURE 5-5. MID$

STRING/NUMERIC DATA CONVERSION

Programmers often encounter situations where numeric data must be converted into string data and vice versa. This is often the case where a function is being used which will accept only string or numeric data as its arguments.

The STR$ and VAL functions are used to convert numeric data to its string equivalent and strings to their numeric equivalent respectively. The ASC function is used to convert a single character to its ASCII numeric equivalent. If ASC is given a string, it will return the ASCII equivalent of the first character in that string. The CHR$ function converts an ASCII numeric code to an equivalent text character.

Examples of the use of STR$, VAL, CHR$, and ASC are given in figure 5-6 and figure 5-7.

```
100 W = 33578
200 W$ = STR$(W):REM W$ = "33578"
300 X = 33579
400 X$ = STR$(X)
500 Y$ = W$ + X$:REM Y$ = "3357833579"
600 Y = VAL(Y$):REM Y = 3357833579
700 Z = INT(Y/10000)
800 PRINT Z
900 END
RUN
 335783
Ok
```

FIGURE 5-6. STR$ and VAL Examples

```
100 A$ = "GILBERT"
200 A = ASC(A$)
300 PRINT A
400 X = 90
500 X$ = CHR$(X)
600 PRINT X$
700 END
RUN
 71
Z
Ok
```

FIGURE 5-7. CHR$ and ASC Examples

VARIABLE TABLE AND STRING STORAGE

BASIC maintains an area in memory in which an entry is maintained for every variable (including array variables) referenced either in a program or in the direct mode. This memory area is known as the **variable table**.

For numeric variables, the value currently assigned to that variable is also stored in the variable table. When that variable's value is changed, the value stored in the variable table will also be changed.

In BASIC, the amount of memory required to store a numeric value in the variable table remains constant. On the other hand, the amount of memory required to store a string variable's value can vary depending upon that value.

Since the memory space required to store a string value can vary, it would be difficult to store these values in the variable table, as that table would have to be continually revised as different values were assigned to string variables. For this reason, BASIC stores string values in a separate memory area known as a **string space**.

BASIC stores a value in the variable table which associates the string variable name (in the variable table) with its associated value in string space. This is known as the **descriptor**. The descriptor describes the number of characters currently assigned to the string variable as well as its location in string space.

Descriptors are not limited to referencing string values stored in the string space. Descriptors can reference strings stored anywhere within BASIC's working area — including file buffers and the program itself. When a string constant is assigned to a string variable in a BASIC program (ex. 10 A$ = "TINA"), that constant need not be stored in the string space as the descriptor can reference it in the program storage area.

HOUSEKEEPING AND FRE

Areas assigned to strings can become unused, because strings in BASIC can have variable lengths. Every time a different value is assigned to a string variable, its length may change. This may cause the space assigned to a string to become partially unused. If the string space is in need of a housekeeping, BASIC will automatically halt program execution and perform one. This operation may be time consuming.

The FRE function allows the programmer to see how much space is available and perform a housekeeping operation if desired. FRE can take either a string or numeric argument. When given a numeric argument, FRE (1) for example, FRE will return the number of bytes of memory remaining. If FRE is given a string argument. FRE(" ") for example, it will first perform a housekeeping operation, and then report the number of free bytes remaining.

FUNCTION SUMMARY

The various built-in functions available in PC*jr* BASIC are summarized in table 5-2.

Table 5-2. PC*jr* BASIC Functions

Function	Description
ABS(*x*)	Absolute value of *x*
ASC(*x*$)	ASCII code of the first character in *x*$
ATN(*x*)	Arctangent of *x* (in radians)
CDBL(*x*)	*x* in double precision
CHR$(*x*)	Character with ASCII code *x*
CINT(*x*)	*x* in integer precision
COS(*x*)	Cosine of *x* (in radians)
CSNG(*x*)	*x* in single precision
CSRLIN	Vertical line position of cursor
CVD(*x*$)	Double precision value for *x*$
CVI(*x*$)	Integer value for *x*$
CVS(*x*$)	Single precision value for *x*$
DATE$	System date
EOF(f)	True or false, indicating End of File of file f
ERL	Line number where last error occurred
ERR	Error code of last error
EXP(*x*)	e raised to the *x* power.
FIX(*x*)	Integer obtained by truncating *x*
FRE(*x*$)	Amount of available memory
HEX$(*x*)	Hexadecimal string equivalent of *x*
INKEY$	Character from keyboard
INP(*x*)	Byte value from port *x*
INPUT$(*x*,#f)	*x* characters from file #f
INSTR(*x*,*x*$,*y*$)	Position of the first occurrence of *y*$ in *x*$ starting at location *x*.
INT(*x*)	Largest integer less than or equal to *x*
LEFT$(*x*$,*x*)	Leftmost *x* characters of *x*$
LEN(*x*$)	Length of *x*$
LOC(*f*)	Location of file pointer in file *f*
LOF(*f*)	Length of file *f*
LOG(*x*)	Natural logarithm of *x*
LPOS(*x*)	Carriage position of printer

Table 5-2. (cont.) PC*jr* BASIC Functions

Function	Description
MID$(*x*$,*x*,*y*)	*y* characters from *x*$, starting at the *x*th character.
OCT$(*x*)	Octal equivalent of *x*
PEEK(*x*)	Byte value in memory location *x*
PEN(*x*)	Light pen value
PLAY(*x*)	Number of notes in the music background buffer
PMAP	Either actual or relative coordinates
POINT(*x*,*y*)	Color of the point (*x*,*y*)
POS(*x*)	Cursor column position
RIGHT$(*x*$,*x*)	Rightmost *x* characters from *x*$
RND(*x*)	Random number
SCREEN(*x*,*y*,*z*)	Character or color at the point (*x*,*y*)
SGN(*x*)	Sign of *x*
SIN(*x*)	Sine of *x* (in radians)
SPACE$(*x*)	String of *x* spaces
SPC(*x*)	String of *x* spaces
SQR(*x*)	Square root of *x*
STICK(*x*)	Coordinates of joystick
STR$(*x*)	String value corresponding to *x*
STRIG(*x*)	State of joystick button
STRING$(*x*,*y*)	Character with ASCII code *y*, *x* times
STRING$(*x*,*x*$)	First character of *x*$, *x* times
USR*n*(*x*)	Calls machine language routine with argument *x*
TAB(*x*)	Enough spaces to move cursor to position *x*
TAN(*x*)	Tangent of *x* (in radians)
TIME$	System time
TIMER	Number of seconds since midnight or reset
VAL(*x*$)	Numeric value of *x*$
VARPTER(*variable*)	Address of *variable* in memory
VARPTER(#f)	Address of file control block for file *f*
VARPTER$ (*variable*)	Three byte string containing the type of *variable*, and its address in memory

Program Concatenation

The final topic covered in this chapter is program concatenation, or program chaining. A complex program may overrun the PC*jr*'s memory limitations. When this is the case, the program can be separated into two or more self-sufficient parts. If a portion of the program is needed that is not currently in memory, it can be loaded without destroying any of the work that the previous portion had accomplished. Program concatentation is facilitated by BASIC's CHAIN and COMMON statements.

CHAIN

CHAIN transfers control to another program and passes variables and files to it from the current program. CHAIN has the following configuration:

CHAIN [MERGE] *file* [,[*linenumber*][,[ALL][,DELETE *range*]]]

file is the name of the program that is to be loaded and run. *file* follows the same rules as would any other program file. The following example would load and run the program, PJK:

```
CHAIN "PJK"
```

linenumber specifies the line to which control will be given, following the loading of the new program. If *linenumber* is omitted, execution will begin with the first line in the new program.

ALL keeps all variables intact. Therefore, the new program can use all the variables of the old program as if the new program had created them itself. The CHAIN command will not, however, preserve the current variable types or currently defined functions, unless the MERGE parameter is also used.

MERGE is used to combine two programs. For example, suppose a program has a main menu screen which selects the different options available to the operator. It would be convenient, if the main menu was always in memory. The various options could then be loaded into or out of memory, as needed. The main menu portion of the program could be merged with the options as they were called for. The following example

merges the program named SORT into the main program, and then starts execution at line 100. A program must be stored in the ASCII format, in order to be merged.

```
CHAIN MERGE "SORT",100
```

After one of the options, or **overlays**, has been used, it would be desirous to delete it from memory. In this way, room could be made to accommodate another section of the program. The optional DELETE parameter will erase the program lines specified in *range*.

```
CHAIN MERGE "WILBUR",3000,DELETE 3000-3999
```

The previous example will merge the program named WILBUR into memory after deleting lines 3000 through 3999 of the program currently in memory. The DELETE parameter is rather picky in that it requires that the line numbers specified as the first and last in the range. actually exist.

COMMON

If all the variables of the old program are required by the new program, the programmer may choose which variables are passed to the new program. The COMMON statement declares a variable as a common varible to both programs. The COMMON statement must appear in the old program, before a CHAIN command can be executed.

```
COMMON variable [,variable]...
```

It is recommended that CHAIN's ALL parameter be used, if a number of variables are to be passed to the main program.

```
100 COMMON PAT,PHONE$,PRO( )
```

Notice that arrays may be specified in a COMMON statement by placing a pair of parentheses "( )" after the array's name. Of course, the array must have previously been dimensioned, using a DIM statement.

6

Files & File Handling with PCjr BASIC

Introduction

In the preceding chapters, we did not discuss the concepts and programming techniques related to storage of data on cassette tape or diskette. In this chapter, these concepts will be discussed. The writing of programs which make use of these devices will also be discussed.

FILES, RECORDS, AND FIELDS

Before learning specific concepts which relate to the cassette tape unit and diskette drives, it is essential that the user understand the concepts of **files, records,** and **fields.**

A file can be defined as a collection of related data. Files can be distinguished as being either **program files** or **data files.** A program file consists of a program which has been saved on diskette or cassette tape.

A data file consists of a collection of related information which has been saved on a diskette or cassette tape. Generally, a data file is read from or written to storage by a program. Data files are divided into

smaller segments known as records and fields. A field is a single piece of data. Fields are grouped together as a record. These records, in turn, make up the file.

A simple illustration may help clarify the concepts of a data file, record, and field. Take an address book as an example of a data file. This file would contain name, address, and telephone number data for the individuals appearing in the address book. Each individual's name, address, and phone number would represent one record. For example, the following data would make up one record:

Jay Gatsby
1 Shore Lane
West Egg, NY 10565
516-787-2122

Each individual data item within the record (i.e. name, street address, city, state, zip code, telephone number) could be thought of as a field.

A data file is written or read as a series of constants. For example, our address book example might be read as follows:

```
"Jay Gatsby","1 Shore Lane","West Egg","NY",10565,"516-787-2122"
"Nick Carraway","7 Shore Lane","West Egg","NY",10565,"516-787-2736"
```

When these data items are read or written, the first field will have been defined as the name, the second as the street address, the third as the city, the fourth as the state, the fifth as the zip code, and the sixth as the telephone number.

Note that the fifth field is numeric, while the others contain string data. Notice that the string data is enclosed in quotation marks. Finally, note that each data item is separated by a comma. For the computer to be able to distinguish where one data item ends and another begins, these items must be separated with a character known as a **delimiter.** A delimiter might consist of a comma (as in our example), a blank space, a line return character, or a form feed character.

The advantages of using data files with programs is obvious. Data files allow the user to save, alter, and redisplay data as is necessary. For example, using our address book as an example, programs could be

written to do the following:

1. Enter changes in an individual's record by reading the file from storage until the desired record is found, inputting the required changes, and rewriting the file back into storage.
2. Displaying an individual's name, address, and telephone information by reading the file from storage until the desired record has been found, outputting the field data to the screen, and rewriting the file back into storage.

The use of a data file with a mass storage device is analogous to the use of a file cabinet for storing information in an office.

FILE SPECIFICATIONS

Every file is identified with a **file specification,** which consists of a **filename** and a **device name.*** The filename identifies the file to be searched for, and the device identifies where the file is to be searched for. Some examples of file specifications are given below.

```
CAS1:TEXT
A:INVOICE
A:LETTER2.TXT
```

Every file is identified by a filename that can include up to eight characters. These characters can contain the letters A to Z, the numbers 0 to 9, or any of the following special characters:

( ) { } @ # $ % & ! - ' ' / ~ _ ¦

Filenames for files being stored on diskette may also include a **filename extension.** A filename extension consists of a period and three letters which appear immediately after the primary filename.

A filename can be entered with upper or lowercase letters. However, the computer will interpret all lowercase entries as capitals. The following filenames all refer to the same file:

---

* In certain instances in Cartridge BASIC, a directory path must also be specified.

Vendor.TXT
VENDOR.txt
VENDOR.TXT
vendor.txt

The file specification prefixes the filename with a device name. The device name is the name of the storage device which is to hold the file.

A device name can consist of 1 to 4 characters followed by a colon. Table 6-1 contains the device names that are recognized by the PC*jr*. Each of the devices in the table can accept files. Although the disk drive and cassette unit will be used throughout the chapter as input and output devices, any of the devices listed in table 6-1 could easily be substituted. Some of the devices can only be used for input or output. For example, SCRN: can only be used as an output file, and KYBD: can only be used as an input file.

The device name need not be specified in the file specification if that device is the default device. In Cassette BASIC, the default device is CAS1:. In Cartridge BASIC, the default device is A:.

TABLE 6-1. Filename Devices

Device Name	Reference	Input	Ouput
CAS1:	Cassette Recorder	x	x
A:	Diskette Drive	x	x
COM1:	Modem or Asynchronous Communications Adapter	x	x
COM2:	Asynchronous Comm. Adapter	x	x
LPT1:	Line Printer		x
KYBD:	Keyboard	x	
SCRN:	Screen		x

File Access

File access refers to the processes of reading data from a file or

writing data to it. In BASIC, data is organized in a file in either a sequential or a random manner. The mode in which a file's data is organized determines how that data will be accessed. Random access does not mean that the file is stored in a haphazard manner. Random access denotes that any part of the file can be accessed directly. Sequential access denotes that the file's data must be read or written in a specific order.

SEQUENTIAL AND RANDOM FILES

Two types of data files are used in PC*jr* BASIC, sequential data files and random access data files. Cassette BASIC allows only sequential files, while Cartridge BASIC allows either. Random access is not available when a cassette tape is being used for data storage.

Each record of a sequential disk file is assigned exactly as much storage space as it requires. There are no blank spaces between records in a sequential file. In random data files, a constant space is assigned to every record in the file. If the record does not occupy the entire space assigned to it, the remaining space is left blank.

The concepts of sequential and random files are pictured in figure 6-1. Notice that the length of each record in the random file is constant at 400 bytes. The record length of a sequential file is variable.

The important difference between random and sequential files lies in how each file is accessed. **Direct access** of any record in a random file is possible regardless of where that record is located on the file. By direct access, we mean that any record in the file may be retrieved regardless of its position, without having to search through the entire file to find it. With random files, PC*jr* BASIC knows the length of each record and can easily calculate the location of any record in the file.

Records in a sequential file can only be retrieved by **sequential access.** In sequential access, the record search begins with the first record in the file and must continue until the desired record is found. In other words, in a sequential file, to find record 17, PC*jr* BASIC would first have to read the preceding 16 records, one by one. Since BASIC does not know the record length of sequential files, it has no way of determining the location of record 17, other than by reading the first 16 records.

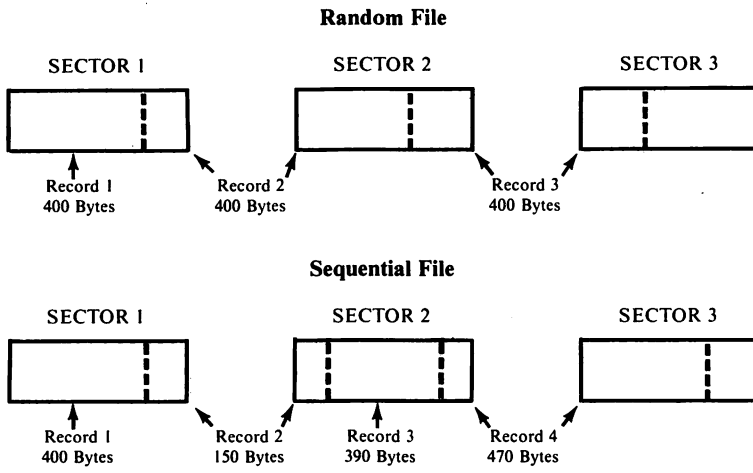


FIGURE 6-1. Random and Sequential Data Files

Random and sequential files each have advantages and disadvantages. Sequential files use less disk space than random files. Since each record in a sequential file is assigned only the disk space it needs, no diskette space is wasted by sequential files. Random files require every record to be assigned the same amount of disk space required by the longest record in that file. This generally results in wasted space. Random files have an advantage over sequential files in that a record from a random file may be read into memory, changed, and then written back to the disk. A record from a sequential file cannot be read, modified, and then rewritten, as any change that might affect a single record's length could adversely affect the entire file.

OPENING A SEQUENTIAL FILE

Before a file can be read from or written to in PC*jr* BASIC, it must be opened. When a file is opened in BASIC, first, the operating system is called upon to read the disk to find information regarding that file in the disk directory. Once this information has been obtained, BASIC will initialize buffer areas in memory through which data will pass as it is read from and written onto the disk.

Once a file has been opened, the BASIC program can read data from

that file one sector at a time. This data is passed to the memory buffer that had been set up when the file was opened. BASIC may then read this data from the buffer area in the same manner that it would read and use any other data stored in memory.

When BASIC writes data to an open file, the data is first written to that file's memory buffer. Data is not actually written to the diskette or cassette until the memory buffer has become filled. When the buffer is full, the data is written to the diskette or cassette one sector at a time.

Once a file has been opened and file access operations have been completed, that file should be closed. This is especially important whenever data is written to a file.

When a BASIC file is closed, any data remaining in that file's memory buffer will be written to the diskette. This occurs even if the memory buffer is not full. Next, the operating system adds the necessary directory information for that file.

The BASIC statement OPEN is used to open a file. OPEN can be used with either of two configurations with sequential files. The first is as follows:

OPEN *"file specification"* FOR *mode* AS [#] *filenumber*

The abbreviations in the preceding format can be interpreted as follows:

file specification Filename and optional drive identifier

mode can be any of the following:

INPUT -- for sequential input. Data can only be read from the file. Data cannot be written to it. A "File not found" error will occur if an attempt is made to open a file for input that does not already exist.

OUTPUT -- for sequential output. OUTPUT always causes a new file to be created. If a file already exists with the same filename as that specified in an OPEN statement with the OUTPUT mode indicated, existing data in the file will be erased. Data will be written to that file from its beginning point.

APPEND -- for sequential output mode for diskette files only. APPEND is specified when data is to be added to the end of an

existing file. If the file being opened for an APPEND already exists, new data will be written to the end of that file. If that file does not exist, a new output file will be created.

filenumber is a required number which is used to refer to a file while it is open. It is much easier to refer to a file as #1 than by its file specification.

The following are examples of valid OPEN statements:

```
OPEN "A:TRANS.DAT" FOR INPUT AS #1
OPEN "B:TEXT.FLE" FOR APPEND AS #2
```

The first example opens TRANS.DAT on drive A for input as file #1. The second example opens TEXT.FLE on drive B for an append as file #2.

The OPEN statement can also be used with the following alternate format for sequential files:

OPEN *altmode,* [#] *filenumber, filespec*

The abbreviations in the preceding format can be interpreted as follows:

altmode is a string whose first character is one of the following:

O Indicates sequential output mode.
I Indicates sequential input mode.
(APPEND is not available in this configuration.)

filenumber and *filespec* retain the same meanings as in the first format.

The following are examples of the use of the alternate format of the OPEN statement:

```
OPEN "O",#1,"LPT1:"
OPEN "I",#2,"A:VENDOR.DAT"
OPEN "O",#1,"CAS1:CHECKS"
```

The first example opens the printer as #1 for output. The second example opens VENDOR.DAT as #2 for input on the disk drive. The third example opens CHECKS as #1 for output on the cassette device.

In Cassette BASIC and Cartridge BASIC without DOS, a maximum of four files can be open at any one time. However, none of these open

files can be disk files. Although Cartridge BASIC with DOS does allow disk files, a limit exists on the number of disk files that may be open at any one time. The maximum number of open disk files is specified in the BASIC command entry (as explained in chapter 3). From 1 to 15 files may be specified as that maximum. If no entry is made in the BASIC command, a maximum of three disk files can be open at any one time. Notice that the use of a filenumber other than 1, 2, or 3 in such a case will cause the "Bad file number" error to occur.

It is possible to open a single file under two separate file numbers. A sequential file can be opened for both input and output, provided that the file is first opened for output and then opened for input. PC*jr* BASIC will not allow a file to be opened first for input, then for output, because of the problems that this situation presents to the operating system.

To change a file from input to output, the file must first be closed. This could be accomplished by using a CLOSE statement. Once a file has been closed as an input file, it may be reopened as an output file.

It is good programming practice to close a file, once the program has finished accessing it. More than one file can be closed with a single CLOSE statement as illustrated below:

```
900 CLOSE #2,#3
```

If the CLOSE statement is executed without an argument as shown below, all open files will be closed:

```
950 CLOSE
```

WRITING TO A SEQUENTIAL FILE

Once a sequential file has been opened, any one of the following statements can be used to output data to it:

```
PRINT#
PRINT# USING
WRITE#
```

PRINT# and PRINT# USING function almost exactly as do PRINT and PRINT USING. The difference lies in the fact that PRINT# and

PRINT# USING require that a file number be specified. Data is written to that file rather than to the display. An example of PRINT# and PRINT# USING is given below:

```
10 OPEN "FILE.DAT" FOR OUTPUT AS #1
20 A = 27.932:B$ = "DON"
30 C = 5.72:D = 9.84
40 PRINT#1,A;B$
50 PRINT#1,USING "**$##.## ";C,D
```

The following will be saved by the PRINT# and PRINT# USING statements in lines 40 and 50:

```
 27.932   DON
***$5.72 ***$9.84          CR and LF characters
```

We can actually check this output by substituting "SCRN:" for "FILE.DAT" in line 10. This causes the output to appear on the screen.

When WRITE# is used to output data to a sequential file, individual data items will be separated with commas, and strings will be surrounded by quotation marks.

```
10 OPEN "SCRN:" FOR OUTPUT AS #1
20 A = 27.932:B$ = "DON"
30 C = 5.72:D = 9.84
40 WRITE#1,A,B$,C,D
50 END
RUN
27.932,"DON",5.72,9.84
Ok
```

One advantage of the WRITE# statement is that it outputs data in the same format in which it is read by the INPUT# statement.

READING FROM A SEQUENTIAL FILE

The following commands are used in PC*jr* BASIC to input data from a sequential file:

```
INPUT#
LINE INPUT#
INPUT$
```

INPUT# and LINE INPUT# function with sequential files much like INPUT and LINE INPUT do with the keyboard. INPUT# will read the data at the current position in the sequential file and assign that data to the variable indicated as its argument. The data and variable must be of the same type. If they are not, an incorrect data value may be assigned to a variable.

When INPUT# is reading numeric data, any leading blanks will be ignored. Any non-numeric characters assign the value zero to the variable in the INPUT# statement. As is the case with INPUT, CR/LF characters and commas may be used as delimiters. Spaces may also be used with the INPUT# statement.

When INPUT# is reading string data, any leading blanks will again be ignored. If the first character read by INPUT# is a double quotation mark, every subsequent character will be assigned to the string variable until the next double quotation mark is encountered. If the first character read is not a double quotation mark, every subsequent character will be assigned to the string variable until a comma, carriage return, or line feed character is encountered. A maximum of 255 characters can be assigned to a string variable.

```
10 A$ = "Jerry Smith":B$ = "Williams,Dave"
20 OPEN "TEXT.DAT" FOR OUTPUT AS #2
30 PRINT#2,A$;B$
40 CLOSE#2
50 OPEN "TEXT.DAT" FOR INPUT AS #2
60 INPUT#2,A$,B$
70 PRINT A$
80 PRINT B$
90 END
RUN
Jerry SmithWilliams
Dave
Ok
```

In our preceding example, A$ and B$ were output as follows:

```
Jerry SmithWilliams,Dave*
```

* The PRINT# statement does not output the quotation marks specified in the assignment statement in line 10.

When INPUT# caused these to be read and assigned to variables, "Jerry SmithWilliams" was assigned to A$ and "Dave" was assigned to B$. Note that the data was input into A$ until the comma was encountered.

By using quotation marks to delimit the strings when they are output in line 30, we can re-input the string properly. We must use the CHR$ function with the ASCII code for the quotation mark (34) to represent this character. If line 30 of our preceding example was edited as follows,

```
30 PRINT#2,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

and the program was executed, the following data would be displayed:

```
Jerry Smith
Williams, Dave
```

LINE INPUT# is used with the following configuration:

LINE INPUT# *filenumber, variable*

All characters will be assigned to the specified variable until a CR/LF character is encountered.

Note our revised version of the program described on the previous page using LINE INPUT#. LINE INPUT# accepted every character including the double quotations that served as delimiters in INPUT#.

```
10 A$ = "Jerry Smith":B$ = "Williams, Dave"
20 OPEN "TEXT.DAT" FOR OUTPUT AS #2
30 PRINT#2,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
40 CLOSE#2
50 OPEN "TEXT.DAT" FOR INPUT AS #2
60 LINE INPUT#2,A$
70 PRINT A$
80 PRINT B$
90 END
RUN
"Jerry Smith""Williams, Dave"
Williams, Dave
Ok
```

INPUT$ is used to retrieve a specific number of characters from a

file. INPUT$ is used with the following configuration:

$$z\$ = \text{INPUT\$}(x\,[,\#y])$$

If x and y are both specified, the next x characters from the file indicated by y will be read into $z\$$. If a file is not specified, the designated number of characters will be read from the keyboard. The use of INPUT$ is shown in the following example:

```
10 A$ = "Jerry Smith":B$ = "Williams, Dave"
20 OPEN "TEXT.DAT" FOR OUTPUT AS #2
30 PRINT#2,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
40 CLOSE#2
50 OPEN "TEXT.DAT" FOR INPUT AS #2
60 A$ = INPUT$(13,#2)
70 PRINT A$
90 END
RUN
"Jerry Smith"
Ok
```

EOF, LOC, AND LOF WITH SEQUENTIAL FILES

The EOF (End of File) function can be used with sequential files to determine if the end of the file has been reached. The configuration for EOF is given below:

A = EOF(*filenumber*)

filenumber indicates the file for which the end-of-file condition is to be checked. This function returns a value of true (–1) if the end-of-file has been reached, and a value of false (0) if it has not. EOF is often used to test for end-of-file while inputting data to avoid an input attempt when the end-of-file has been reached. The following program illustrates this use of EOF:

```
10 OPEN "TEXT.DAT" FOR OUTPUT AS #1
20 PRINT#1, "WHAT'S THE DIFFERENCE BETWEEN AN APPLE?"
30 PRINT#1, "AN ORANGE;"
40 PRINT#1, "BECAUSE A VEST HAS NO SLEEVES."
50 CLOSE:OPEN"TEXT.DAT" FOR INPUT AS #1
60 WHILE 1 = 1
```

```
 70   INPUT#1,A$
 80   PRINT A$
 90 WEND
100 END
RUN
WHAT'S THE DIFFERENCE BETWEEN AN APPLE?
AN ORANGE;
BECAUSE A VEST HAS NO SLEEVES
INPUT PAST END IN 70
Ok
```

If line 60 is edited as follows, the error would no longer occur:

```
60 WHILE NOT EOF(1)
RUN
WHAT'S THE DIFFERENCE BETWEEN AN APPLE?
AN ORANGE;
BECAUSE A VEST HAS NO SLEEVES.
Ok
```

The LOF (Length of File) function returns the specified file in bytes. LOF is used with the following configuration:

A = LOF(*filenumber*)

LOC returns the location within the file of the last record read or written. LOC is used with the following configuration:

A = LOC(*filenumber*)

LOC is calculated in terms of 128 byte blocks. In other words, LOC will return the number of records (assuming each record consists of 128 bytes) that have been input or output to the file since it was opened. For instance, if 256 records of 16 bytes each (4096 bytes) were written to a sequential file, LOC would return a value of 32.

Random File Access

Before a random file can be used, the programmer must determine the length of each of its records. The length of each record in the random file must be the same, and can be calculated by adding the length of each

of the individual fields in that record.

When a random file is opened, a random file buffer is set up in memory. Data to be written to the random file is first placed in that file's buffer. When all the data for a record has been written to the buffer, the programmer can instruct the computer to store the record on diskette. When data is to be read from a random file, a record is first read from disk into the random file buffer. The individual fields within that record can then be read by instructing the program to access the buffer.

Once a random file has finished being accessed, it should be closed. When a random file is closed, any existing data in the random file buffer will be written to the random file on disk, and the memory space reserved for the buffer will be freed.

OPENING AND CLOSING A RANDOM FILE

The following configurations can be used to open a file.

OPEN *filespec* AS [#] *filenumber* [LEN = *rcdlen*]
OPEN "R", [#] *filenumber, filespec* [,*rcdlen*]

filespec refers to the file specification. *filenumber* refers to the number that will be assigned to the random file. *rcdlen* specifies the length of each record in the random file in bytes. If this parameter is omitted, a record length of 128 bytes will be assumed. The maximum record length is 32,767 bytes.

The random file's record length may not be greater than that indicated by the /S: option when BASIC was started up. If /S: was not specified in the BASIC command line, a record of up to 128 bytes will be allowed. CLOSE works with random files exactly as it does with sequential files.

FIELD VARIABLES

As discussed earlier, random file records are read into a random buffer. Each field in the record will correspond to a certain area in the buffer. In order to access a particular field in the random file record, the corresponding field in the buffer must actually be accessed. In order to do so, these buffer fields must be identified with a **buffer variable** or **field variable.**

A field variable must always be a string variable. A string variable via its descriptor can refer to a string value in the string storage space, in the BASIC program itself, or in fact anywhere in the BASIC working area. A string variable can use its descriptor to reference a string constant in the file buffer.

A numeric variable could not be used for a field variable because the value in the field buffer would also have to be stored in the variable table.

The FIELD statement is used to define field variables. FIELD is used with the following configuration:

FIELD # *filenumber, size* AS *fieldvariable,...*

size indicates the length of the field in bytes, and *fieldvariable* indicates the string variable used to reference that field.

A FIELD statement cannot be executed until a file has been opened. The sum of the lengths of the field variables specified by FIELD cannot exceed the record length indicated in the OPEN statement. If the combined length of the field variables exceeds that of the record length, the following error message will appear:

```
FIELD overflow
```

More than one FIELD statement can be executed concurrently for the same random file. This allows the programmer to create several different types of records within the same random file depending upon how the field variables are defined.

```
FIELD 1, 10 AS A$, 20 AS B$, 30 AS C$
```

In the preceding example, the FIELD statement allocates the first 10 positions of the random file buffer #1 for the string variable A$. The next 20 positions are reserved for B$, and the following 30 positions are reserved for C$.

WRITING DATA TO A RANDOM FILE BUFFER

The LSET and RSET statements assign values to field variables. In effect, LSET and RSET place data into the buffer field designated by the

field variable. LSET and RSET are used with the following configurations:

LSET *fieldvariable* = *x$*
RSET *fieldvariable* = *x$*

fieldvariable refers to a string variable referenced in a FIELD statement. *x$* can refer to either a string constant or a string variable. If *x$* contains fewer bytes than those allowed for in *fieldvariable,* LSET will cause *x$* to be left-justified in the field buffer. In other words, the first character in *x$* will be placed in the farthest left byte in the field buffer. If *x$* does not occupy the entire field buffer, blank spaces will be added to the right of *x$*'s last character. RSET causes *x$* to be right-justified in the field buffer. If necessary, blank spaces will be added to the left of *x$*'s first character.

If the string to be placed in the field buffer is larger than that allowed for, extra characters will be truncated from the right of the string. This is true regardless of whether RSET or LSET is being executed.

Although only string variables can be used to define field buffers, numeric values can still be stored in a random file. However, to store numeric values in a random file, it is necessary to convert the numeric data to a string. The STR$ function could be used to convert a numeric value to a string. The difficulty in using STR$ is that a string value returned by STR$ consumes more memory storage than a numeric constant would.

PC*jr* BASIC includes three functions, MKI$, MKS$, and MKD$ which allow a string to be converted to a numeric value. The resultant string will occupy the same number of bytes on the disk as did the original variable in memory. These functions are used as follows:

A$ = MKI$(*integer*)
A$ = MKS$(*single precision*)
A$ = MKD$(*double precision*)

integer, single precision, and *double precision* refer to integer, single precision, and double precision constants or variables respectively. MKI$ converts an integer value into a 2-byte string. MKS$ converts a single precision value into a 4-byte string, and MKD$ converts a double preci-

sion value into an 8-byte string. MKI$, MKS$, and MKD$ are only meant to be used to convert data for storage in a random file.

To store numeric data in a random file, the field variables must have a sufficient number of bytes reserved in the FIELD statement. For example, the field variables defined by the following FIELD statement could allow for the storage of 4 numbers in a record. Two of these numbers could be integers, one could be single precision, and one could be double precision.

```
1000 FIELD #2, 2 AS A$, 2 AS B$, 4 AS C$, 8 AS D$
```

WRITING A RECORD FROM THE RANDOM FILE BUFFER TO THE FILE

Once the field buffers have been assigned values via LSET and RSET, the PUT statement can be used to write the random buffer to the random file as a record. PUT is used with the following configuration:

PUT # *filenumber* [,*rcdnumber*]

filenumber refers to the number assigned to the random file when it was opened. If the optional *rcdnumber* parameter is omitted, the contents of the file buffer will be written in the next available position in the random file. If *rcdnumber* is specified, the file buffer's contents will be written at the indicated record number. The record number can range from 1 to 32,767.

In the following example, PUT will cause the contents of the file buffer to be written at the fifth record location in the random file opened as #3.

```
100 PUT #3,5
```

READING A RECORD FROM THE RANDOM FILE INTO THE BUFFER

The GET statement is used to read a record from the random file into the buffer. The configuration for GET is very similar to that for PUT.

GET # *filenumber* [,*rcdnumber*]

If *rcdnumber* is not specified, the record following the last one read will be retrieved. The following GET statement would cause the tenth record of the random file opened as number 5 to be read into the buffer:

```
100 GET #5,10
```

READING DATA FROM THE RANDOM FILE BUFFER

Once data has been read into the random file buffer, that data can be read from the buffer by using that buffer's field variables in an assignment statement. If a string value in the buffer represents a numeric value, the string should be converted back to its numeric equivalent. PC*jr* BASIC's CVI, CVS, and CVD functions allow string data to be converted back into numeric data. These function as the exact opposites of MKI$, MKS$, and MKD$, respectively. CVI converts a 2-byte string to an integer, CVS converts a 4-byte string to a single precision numeric value, and CVD converts an 8-byte string to a double precision numeric value.

LOC & LOF WITH RANDOM FILES

When used with random files, LOC will return the number of the record last accessed by BASIC. Assuming that file buffers #1 and #2 are opened to the same file, the following line could be used to load file buffer #2 with the record last accessed by file buffer #1:

```
GET #2,LOC(1)
```

LOF operates with random files the same as it does with sequential file. EOF cannot be used with random files.

USING RANDOM ACCESS FILES

Random access files are much easier to use than sequential access files. The examples in figures 6-2 and 6-3 demonstrate the use of random files. Figure 6-3 uses the economical MKS$ function in order to reduce disk space used. Figure 6-2 uses the more readable STR$ function in order to make the data file more understandable.

```
 10 INPUT "ENTER PAYROLL NUMBER ";Z
 20 Z$ = STR$(Z)
 30 Y$ = "PAY" + Z$ + ".DAT"
 40 OPEN Y$ AS #2 LEN = 53
 50 FIELD#2,2 AS RCD.NO$,4 AS EMPL.NO$,15 AS LAST.NAME$,
    15 AS FIRST.NAME$,5 AS NO.HOUR$,5 AS RATE$,7 AS PAY$
 60 X% = 1
 70 A$ = " ":B$ = " ":C$ = " ":D = 0:E = 0
100 INPUT "ENTER EMPLOYEE'S PERSONNEL NO. ";A$
110 INPUT "ENTER EMPLOYEE'S LAST NAME ";B$
120 INPUT "ENTER EMPLOYEE'S FIRST NAME ";C$
130 INPUT "ENTER NUMBER OF HOURS WORKED ";D
140 INPUT "ENTER PAY RATE ";E
145 LSET RCD.NO$ = STR$(X%)
150 LSET EMPL.NO$ = A$
160 LSET LAST.NAME$ = B$
170 LSET FIRST.NAME$ = C$
180 LSET NO.HOUR$ = STR$(D)
190 LSET RATE$ = STR$(E)
200 LSET PAY$ = STR$(D*E)
210 PUT#2,X%
220 INPUT "DO YOU HAVE ADD'L ENTRIES (Y = YES:N = NO) ";FLAG$
230 IF FLAG$ = "Y" THEN X% = X% + 1:GOTO 70
240 IF FLAG$ = "N" GOTO 300 ELSE 220
300 INPUT "DO YOU WISH TO DISPLAY A RECORD (Y = YES:N =
    NO) ";BFLAG$
310 IF BFLAG$ = "N" THEN 999
320 IF BFLAG$ = "Y" THEN 330 ELSE 300
330 INPUT "WHICH RECORD WOULD YOU LIKE TO DISPLAY (0 =
    END) ";Y%
340 IF Y% = 0 THEN 999
350 IF Y% <= X% THEN GET #2,Y% ELSE 330
360 PRINT RCD.NO$;EMPL.NO$;LAST.NAME$;FIRST.NAME$;
    NO.HOUR$;RATE$;PAY$
365 GOTO 330
370 CLOSE 2
999 END
```

FIGURE 6-2. Random File Access Using STR$

```
 10 INPUT "ENTER PAYROLL NUMBER ";Z
 20 Z$ = STR$(Z)
 30 Y$ = "PAY" + Z$ + ".DAT"
 40 OPEN Y$ AS #2 LEN = 50
 50 FIELD#2,2 AS RCD.NO$,4 AS EMPL.NO$,15 AS LAST.NAME$,15 AS
    FIRST.NAME$,4 AS NO.HOUR$,4 AS RATE$,4 AS PAY$
 60 X% = 1
 70 A$ = " ":B$ = " ":C$ = " ":D! = 0:E = 0
100 INPUT "ENTER EMPLOYEE'S PERSONNEL NO. ";A$
110 INPUT "ENTER EMPLOYEE'S LAST NAME ";B$
120 INPUT "ENTER EMPLOYEE'S FIRST NAME ";C$
130 INPUT "ENTER NUMBER OF HOURS WORKED ";D
140 INPUT "ENTER PAY RATE ";E
142 F = D*E
145 LSET RCD.NO$ = MKI$(X%)
150 LSET EMPL.NO$ = A$
160 LSET LAST.NAME$ = B$
170 LSET FIRST.NAME$ = C$
180 LSET NO.HOUR$ = MKS$(D)
190 LSET RATE$ = MKS$(E)
200 LSET PAY$ = MKS$(F)
210 PUT#2,X%
220 INPUT "DO YOU HAVE ADD'L ENTRIES(Y = YES:N = NO) ";FLAG$
230 IF FLAG$ = "Y" THEN X% = X% + 1:GOTO 70
240 IF FLAG$ = "N" GOTO 300 ELSE 220
300 INPUT "DO YOU WISH TO DISPLAY A RECORD(Y = YES:N =
    NO) ";BFLAG$
310 IF BFLAG$ = "N" THEN 999
320 IF BFLAG$ = "Y" THEN 330 ELSE 300
330 INPUT "WHICH RECORD WOULD YOU LIKE TO DISPLAY (0 =
    END ";Y%
340 IF Y% = 0 THEN 999
350 IF Y%<=X% THEN GET #2,Y% GOTO 330
355 A% = CVI(RCD.NO$):B! = CVS(NO.HOUR$):C! =CVS(RATE$):
    D! = CVS(PAY$)
360 PRINT A%;EMPL.NO$;LAST.NAME$;FIRST.NAME$;B!;C!;D!
365 GOTO 330
370 CLOSE 2
999 END
```

FIGURE 6-3. Random File Access Using MKI$, MKS$, & MKD$

File Commands

PC*jr* BASIC includes eight commands designed to allow the user to perform file handling operations while the BASIC interpreter is active. These include SAVE, LOAD, RUN, KILL, NAME, MERGE, CHAIN, and FILES.

SAVE

SAVE generally is used to store a program on a cassette or a disk file. SAVE is used with the following configuration:

SAVE *"filespecification"* [,P / ,A]

filespecification indicates the device where the program in RAM is to be saved as well as the filename to be assigned to that file. Note that *filespecification* must be enclosed within double quotation marks.

The optional parameter A indicates that the file is to be saved in ASCII format. Only files saved in this format can be loaded with the MERGE command.

The optional parameter P causes the program file to be saved in encoded binary format. In effect, P results in the program file being protected. If an attempt is made to LIST or EDIT a protected program file, the following error will result:

```
Illegal function call
```

When SAVE is used to save a file on disk and no filename extension is indicated in *filespecification,* an extension of .BAS will automatically be assigned. If Cassette BASIC is active, the default device will be CAS1:. If Cartridge BASIC is active, the default device will be A:.

LOAD

The LOAD command is generally used to load a program file into memory from cassette or diskette. LOAD is used with the following configuration:

LOAD *"filespecification"* [,R]

LOAD erases any program lines and variables in memory before the specified program is loaded. When LOAD is used without the optional R parameter, any open files will be closed.

When LOAD is executed with R, the program will be automatically run after it has been loaded. Also, if R is specified, all data files will remain open. LOAD is generally executed with the R option in order to chain two or more programs together. Since existing data files remain open, this information can be shared by the programs being chained.

If *filespecification* does not include a filename extension, the .BAS extension will be automatically provided.

When LOAD is used with the cassette device (CAS1:), the filenames present in the cassette will be displayed one by one on the screen. Each filename will be followed by a period and a letter which specifies the file type. These letters are listed with the types of files they reference in table 6-2.

If the filename displayed matches that indicated by *filespecification,* that file will be loaded into memory and the following message will be displayed:

```
Found
```

If these do not match, the message "Skipped" will be displayed, and the next filename on the cassette will be displayed.

The search for the indicated filespecification on the cassette tape can be ended at any time by pressing Ctrl-Break. If *filespecification* is not indicated when LOAD is executed for the cassette unit, the next program file on the cassette will be loaded.

TABLE 6-2. File Type Abbreviations During a Cassette LOAD

File Type Abbrev.	Reference
.B	BASIC programs stored in internal format (with SAVE).
.A	BASIC programs stored in ASCII format (with SAVE,A).
.P	Protected BASIC programs stored in encoded binary format (with SAVE,P).

.D Data files created by OPEN.

.M Memory image files created by BSAVE.

RUN

The RUN command is used to begin execution of a program. RUN is used with the following configuration:

RUN ["*filespecification*"][,R]

If RUN is executed with *filespecification,* the indicated file will be loaded into memory and executed. If RUN is executed without the *filespecification* parameter, the program currently stored in RAM will be executed. If *filespecification* is indicated, it must be enclosed within double quotation marks.

If the optional parameter R is specified, any open files will remain open. Otherwise, they will be closed.

KILL

KILL is used with the following configuration to delete the indicated diskette file. If the filename includes the .BAS extension, that extension must be included in *filespecification.*

KILL "*filespecification*"

NAME

NAME is used with the following configuration to change a filename.

NAME "*filespecification*" AS "*newfilename*"

The filename included in *filespecification* will be changed to that specified in *newfilename.* If the filename includes the .BAS extension, that extension must be included in the command line.

MERGE

The MERGE command loads the specified program file into memory and combines it with existing program lines in memory.

MERGE *"filespecification"*

If the file being loaded contains a program line with the same line number as one of the program lines already present in memory, the program line being loaded will replace that line.

For a program to be loaded with MERGE, it must have been saved in ASCII format using the A option with the SAVE command.

CHAIN

The CHAIN statement allows one BASIC program to load and run another BASIC program. A COMMON statement can be included in the program containing the CHAIN statement which allows that program to pass the values of some or all of its variables to the program being chained.

FILES

The FILES command can be used to display the filename of one, several, or all of the files on the indicated drive.

FILES [*"filespecification"*]

If FILES is executed without the optional parameter, all diskette files on the disk drive will be displayed. The use of wild cards in *filespecification* is analogous to their usage with the DOS command, DIR. Please refer to chapter 9 for an explanation of wild cards.

7

PCjr BASIC Graphics and Sound

Introduction

The PC*jr* has six available graphics modes, four graphics modes more than its predecessors, the IBM PC and IBM PC XT. Since the PC*jr*'s display was designed to mimic the IBM PC's color graphics board, all of the original graphics modes operate the same on the PC*jr* as they do on the PC. The new modes give the PC*jr* some of the best color graphics available on a home computer.

All of the graphics modes are color capable, although the maximum number of concurrently displayable colors is limited by the selected graphics mode. In addition to its color capability, the PC*jr* supports three distinct screen resolutions — low (160 x 200 pixels*), medium (320 x 200 pixels) and high (640 x 200 pixels). Table 7-1 illustrates the available combinations of colors and screen resolutions.

* A pixel can be defined as a single screen coordinate.

Table 7-1. # of Colors vs. Resolution

Colors	Screen Resolution
16	low
4	medium*
16	medium
2	high
4	high

Besides its additional graphics modes, the PC*jr* supports sophisticated sound capabilities, including three channel music and sound effects. These are generated by the Texas Instruments programmable tone generator, the SN76489A. The use of this IC to generate complex sounds will be discussed later in this chapter.

Pixels

In low resolution graphics, the display can be divided into a grid of 200 rows of 160 columns each. Every point on the screen can be uniquely identified by using its row and column numbers. In low resolution, there are 32000 uniquely addressable screen elements. Each specific screen element is called a **pixel**.

The farthest left column of the screen has been defined as column 0. The farthest right column has been defined as column 159. Likewise, the row numbers extend from 0 (top) to 199 (bottom). This arrangement may seem upside down to persons familiar with a cartesian coordinate system.

The medium and high resolution screens can likewise be divided into grids of 320 x 200 and 640 x 200, respectively. In the case of medium resolution graphics, the column numbers now extend from 0 (left) to 319 (right). In the case of high resolution graphics, they extend from 0 (left) to 639 (right). The row numbers extend from 0 (top) to 199 (bottom) in both medium and high resolution.

* There are two medium resolution modes that can support four colors. The color selection process differs between these modes.

REDEFINING THE SCREEN COORDINATES

As previously mentioned, the default screen coordinates are "upside-down" when compared to standard cartesian coordinates. Figure 7-1 shows the difference between the default coordinates in low resolution and the more natural cartesian coordinates.

Cartridge BASIC allows the programmer to redefine the default screen coordinates. Using the WINDOW statement, cartesian coordinates may be selected. In fact, any rectangular coordinate system may be chosen. The correct syntax of the WINDOW command is as follows:

WINDOW [[SCREEN] (x_1,y_1) - (x_2,y_2)]

x_1, y_1, x_2, and x_2 are called **world coordinates**. World coordinates are single precision floating point numbers. They redefine the coordinates of the screen block, defined by the VIEW statement (discussed later). If no VIEW statements have been executed, the coordinates of the entire screen will be redefined.

```
WINDOW (-1, -1) - (1, 1)
```

For example, the previous WINDOW statement will set the screen coordinates to the cartesian coordinates shown in figure 7-1.

The optional SCREEN parameter allows the screen to be left upside down while still redefining the screen coordinates. The statement,

```
WINDOW SCREEN (-1, -1) - (1, 1)
```

would define the screen coordinates as shown in figure 7-2.

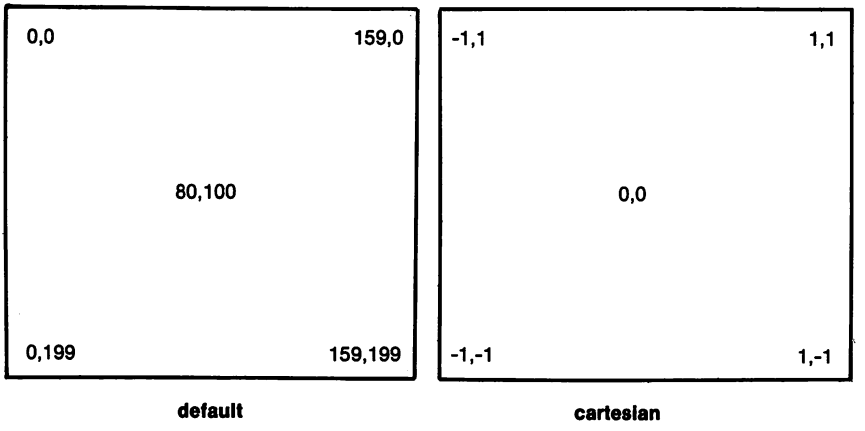


FIGURE 7-1. Coordinate System Comparison

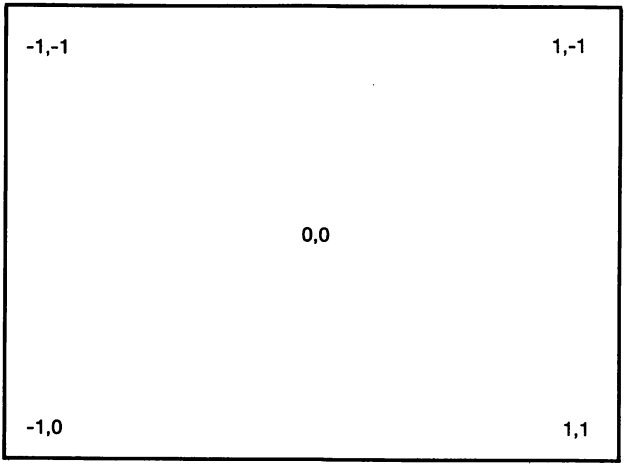


FIGURE 7-2. Effect of WINDOW SCREEN (-1, -1) - (1, 1)

Suppose that a screen, similar to figure 7-3, is to be replicated using the PC*jr*. Perhaps the screen is one from an educational program for preschoolers. At any one time, the programmer would like to devote his attention to a particular quadrant of the screen. VIEW allows the definition of subsets of the screen called **viewports.** Once a viewport has been defined, all screen input and output is accomplished through that part of the screen. VIEW uses the following configuration:

VIEW [[SCREEN] [(x_1,y_1) - (x_2,y_2)[,[*fill*] [,[*boundary*]]]]]

x_1, y_1, x_2, and y_2 are the coordinates of the defined viewport. In low resolution graphics, the following statement would establish the viewport illustrated in figure 7-4:

VIEW (10, 15) - (75, 90)

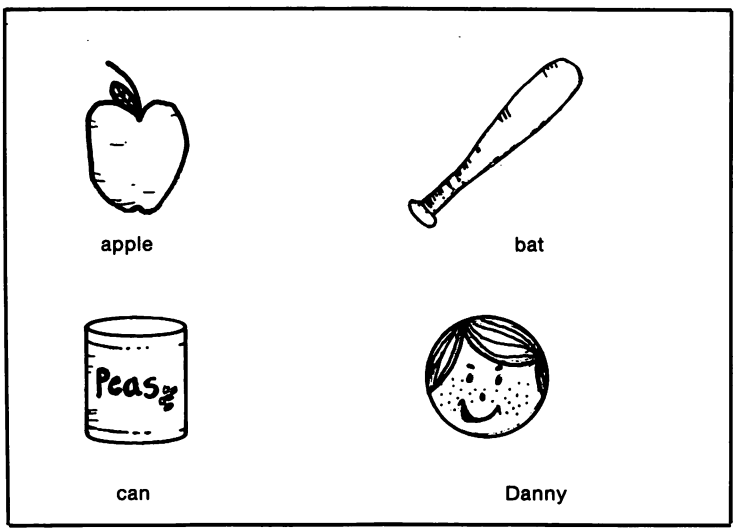


FIGURE 7-3. Screen to be Replicated

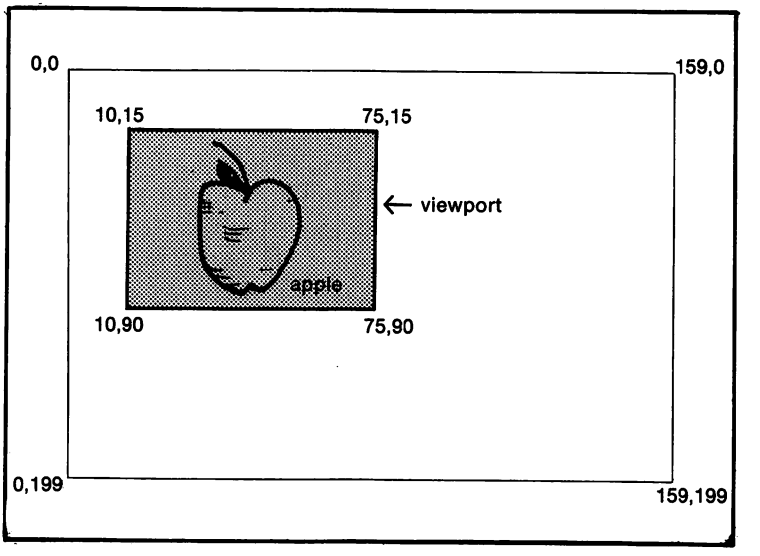


FIGURE 7-4. Viewport

If the previous statement had been executed in a program, any subsequent references to the point (0, 0) would actually affect pixel (10, 15). Likewise, a reference to the point (159, 199) would affect pixel (75, 90). VIEW compresses the old screen coordinates (defined by WINDOW) to fit into the new viewport. Therefore, scaling is easily accomplished.

```
10  SCREEN 2: CLS
20  WINDOW SCREEN (0,0) - (639,199)
30  GOSUB 100
40  FOR J = 1 TO 1000: NEXT J: CLS
50  VIEW (0,0) - (320,100)
60  GOSUB 100
70  END
100 'DRAW ELLIPSES
110 CIRCLE (320,100), 80... .3
120 CIRCLE (300,100), 70... .8
130 RETURN
```

The optional parameter, *fill*, will cause the newly defined viewport to be filled with the color assigned to the attribute specified by *fill*.

Attributes will be discussed later in this chapter. The optional parameter, *boundary*, will cause a border to be drawn around the newly defined viewport, using the color assigned to the attribute specified by *boundary*. In the 16 color modes, both *fill* and *boundary* may range from 0 to 15. In the 4 color modes, they may range from 0 to 3. And in the 2 color mode, they may be assigned either 0 or 1.

SCREEN is the final optional parameter. When included, the SCREEN's coordinate system will no longer fit into the new viewport. In fact, the coordinate system is not changed. However, only points plotted within the viewport will appear on the SCREEN. For example, if the point (10,10) is plotted after "VIEW SCREEN (20,20) - (30,30)" has been executed, nothing will appear as the screen, because the point (10,10) was not present in the viewport.

WINDOW and VIEW will sort both the *x* and *y* pairs, placing the smallest values first. For example, the following four WINDOW commands produce the same coordinate definition, as shown in figure 7-5:

```
WINDOW (0,0) - (10,20)
WINDOW (0,20) - (10,0)
WINDOW (10,0) - (0,20)
WINDOW (10,20) - (0,0)
```

If VIEW or WINDOW is used without an argument, the default conditions for the currently active graphics mode will be restored.

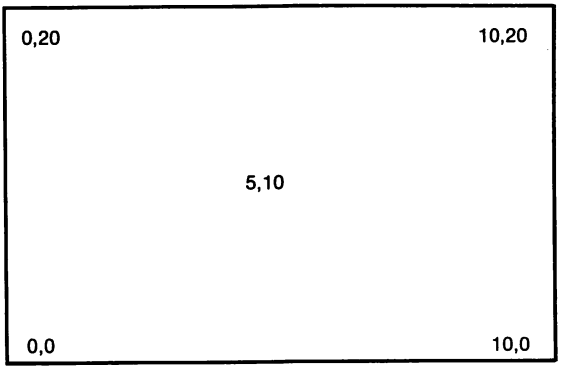


FIGURE 7-5. Coordinate Definition Produced by Example WINDOW Commands

SELECTING A GRAPHICS MODE — SCREEN

As mentioned earlier, six graphics modes are available. These are the low resolution graphics modes and two high resolution graphics modes. The SCREEN statement allows the programmer to select between these modes and the text mode. SCREEN uses the following configuration.

SCREEN [*mode*][,[*burst*][,[*active pg*][*visual pg*]][,*erase*]]

mode indicates the display mode. SCREEN 0 sets the display to the text mode. SCREEN 1 through SCREEN 6 select the graphics modes. This is summarized in table 7-2. SCREEN 0 through SCREEN 2 are also supported on the PC with a Color/Graphics Monitor Adapter installed. SCREEN 3 through SCREEN 6 are only supported in Cartridge BASIC.

Table 7-2. Graphics Modes & Memory Requirements

Screen	Display Mode	Memory Requirement
0	text	40 Column 2K 80 Column 4K
1	medium resolution; 4 colors	16K
2	high resolution; 2 colors	16K
3	low resolution; 16 colors	16K
4	medium resolution; 4 colors	16K
5	medium resolution; 16 colors	32K
6	high resolution; 4 colors	32K

SCREEN can also include an optional second parameter which enables or disables the **color burst signal**. A TV signal contains a component known as the color burst. If the color burst is present, a color picture will be displayed. If it is absent, a black and white picture will be displayed. In the text mode, if *burst* evaluates to 0 (false), the color burst will be disabled. A value of 1 (true) in the text mode enables the color burst. In medium resolution graphics (*mode* = 1 or *mode* = 4), a true value for *burst* will disable color, while a false value will enable color. *burst* has no effect when mode = 2, 3, 5 or 6.

Using the CLEAR statement, more memory may be allotted to the screen than is needed to hold a single page of graphics or text. This allows more than one page of screen memory to be stored. Each of the display modes uses a specific amount of memory (see table 7-2). For example, suppose that 20K was allotted to the screen. Five pages of 80 column text could be stored (5 x 4K = 20K). These would be numbered 0 to 4, inclusive. If 40 column text had been displayed, 10 pages could have been stored (10 x 2K = 20K). These would have been numbered 0 to 9, inclusive.

The two optional parameters for SCREEN, *activepg* and *visualpg*, allow the programmer to select the page to which output will be sent (*activepg*), as well as the page which is displayed on the screen (*visualpg*). This allows the user to write to one page while another is being displayed. *activepg* and *visualpg* can be used in any screen mode. However, sufficient memory must be allotted for the screen.

The final optional parameter, *erase*, indicates how much display memory is to be cleared. *erase* should evaluate to an integer in the range 0-2. 0 indicates not to erase video memory, even if the screen mode changes. 1 indicates to erase the union of the new page and the old page, provided that the *mode* or burst changes. And 2 indicates to erase all video memory if *mode* or *burst* changes. If no value is specified, a 1 will be assumed.

ATTRIBUTES & PALETTES

An **attribute** may be defined as one of the sixteen possible color numbers. Any color may be assigned to each of the attributes. The notion of an attribute is analogous to that of painting by number. Using a graphics command, such as LINE, CIRCLE or PSET, corresponds to drawing the boundaries of a paint by number canvas.

```
LINE (0,0) - (4,4),2,B
```

When LINE is used in the previous configuration, a box will be drawn with corners at pixels (0,0), (0,4), (4,4), and (4,0). The square will be drawn using attribute number two. The following will be the result:

Row	0	1	2	3	4	5 ← Column
0	2	2	2	2	2	0
1	2	0	0	0	2	0
2	2	0	0	0	2	0
3	2	0	0	0	2	0
4	2	2	2	2	2	0
5	0	0	0	0	0	0

Now that the paint by number canvas has been constructed, the canvas must be given colors. Colors are assigned to attributes by the PALETTE statement.

PALETTE [*attribute*][,*color*]

attribute is an expression that evaluates to an integer. It determines which attribute will be effected by the PALETTE statement. *attribute* may range from 0 to either 1, 3 or 15, depending on the graphics mode.

color is an integer that selects from the sixteen displayable colors. Table 7-3 contains a list of these colors. Colors 8 through 15 may be thought of as the "high intensity" values of colors 0 through 7.

```
PALETTE 2, 10
PALETTE 0, 6
```

The previous two statements will cause our square to be colored light green, with a brown background. The square could have just as easily been colored brown with a light green background.

```
PALETTE 2, 6
PALETTE 0, 10
```

Also, the square and its background could have been assigned the same color. In this case, the square would have been invisible to the viewer.

```
PALETTE 2, 6
PALETTE 0, 6
```

Table 7-3. Displayable Colors

0 — Black	8 — Gray
1 — Blue	9 — Light Blue
2 — Green	10 — Light Green
3 — Cyan	11 — Light Cyan
4 — Red	12 — Light Red
5 — Magenta	13 — Light Magenta
6 — Brown	14 — Yellow
7 — White	15 — High-Intensity White

Often, many attributes must be changed simultaneously. PALETTE USING allows the setting of all the attributes with one statement.

PALETTE USING [*array* (*start*)]

array is an integer array that must have 16 elements beginning at the starting index. *start* is an integer indicating the first position in the array to be used in assigning palette colors.

The values in *array* are directly assigned to the attributes. If a value of -1 is in the array, the attribute corresponding to that position in the array will not be changed. Figure 7-6 demonstrates this process.

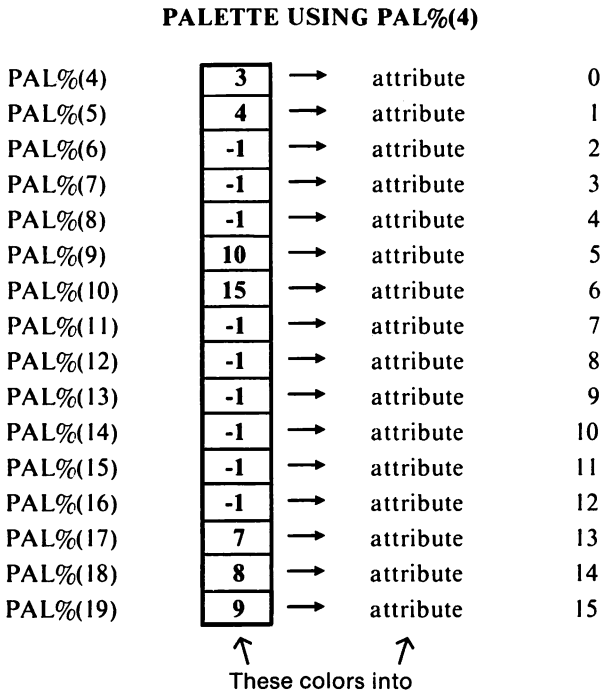


FIGURE 7-6. PALETTE USING Example

SELECTING A COLOR

Before any graphics information can be plotted to the screen, an attribute must be selected. This is accomplished through the use of the COLOR command. The correct syntax of this command is as follows:

FOR SCREENS 3-6 → COLOR [*foreground*],[[*background*]]
FOR SCREENS 1 → COLOR [*background*][,[*palette*]]

The two different configurations can be understood by realizing that screen 1 was originally designed for the PC, while screens 3 through 6 were designed especially for the PC*jr*'s advanced hardware.

In both configurations, *background* represents the color to be assigned to attribute 0.

In the configuration for screens 3-6, whenever a graphics command such as PSET or LINE is issued, the configuration of the command will allow an *attribute* to be specified. This *attribute* is then used to draw the graphics object. If no attribute is specified by the programmer, the current default attribute will be used. *foreground* determines the default drawing attribute.

The use of COLOR with the latter configuration is somewhat more complex. SCREEN 1 does not support the flexible palette of screens 3-6, but instead supports only two fixed palettes. These are given in table 7-4. For example, if *palette* = 0, then green will automatically be assigned to attribute 1. SCREEN 1 is obviously a poor choice for a graphics mode, because SCREEN 4 could be used in its place without sacrificing the number of available colors (4) or changing the memory requirements, (16K). SCREEN 1 was merely included on the PC*jr* to make it compatible with the PC and the PCXT.

Table 7-4. Screen 1 Palettes

Attribute	Palette 0	Palette 1
1	Green	Cyan
2	Red	Magenta
3	Brown	White

PLOTTING — PSET

After a COLOR has been selected, information can be plotted to the screen. This is accomplished by using the PSET command. The correct syntax of this command is as follows:

PSET (*col, row*) [,*attribute*]

col and *row* specify the world coordinates of the pixel to be illuminated. The optional parameter *attribute* determines the color of the plotted pixel. If this parameter is not included, then the current default *attribute* will be used.

RELATIVE COORDINATES

PSET, as well as the rest of the graphics commands, requires the programmer to specify the position on the screen to which graphics data is to be plotted. This information is given in the form of coordinates. As was the case with PSET, coordinates can be of the form (*col, row*), where *col* is the column number and *row* is the row number. This is known as **absolute** addressing. *row* and *col* actually specify screen coordinates, without regard to the last point referenced (LPR). The LPR is analogous to the cursor position in the text mode.

Another means of indicating coordinates is **relative** addressing. With relative addressing, BASIC must be told where the point is located relative to the LPR. This form of addressing uses the following syntax:

STEP (*coloff, rowoff*)

coloff and *rowoff* indicate the column offset and row offset, respectively. The numbers specify the horizontal and vertical direction of the new point from the LPR.

```
100 SCREEN 3
110 PSET (100,100), 15
120 PSET STEP (20,-10), 15
```

The previous program sets two points on the screen. Their actual coordinates are (100,100) and (120,90).

ADVANCED GRAPHICS COMMANDS

With the right combination of PSET and elbow grease, any graphics screen can be drawn. In other words, although PSET gets the job done, it does not accomplish it with a great deal of efficiency. BASIC includes three commands that can simplify the creation of graphics displays. These are LINE, CIRCLE, and PAINT. LINE and CIRCLE have already been illustrated in several of our examples.

LINE can be used to plot consecutive pixels. For example, the following statement will connect the pixel (100,100) to pixel (150,150) with a white line:

LINE (100,100) - (150,150), 15

If the first coordinate is omitted, the line will be drawn from the LPR to the second coordinate. The following statement will connect the LPR (150,150) to pixel (20,150) with a blue line.

```
LINE -(20,150), 1
```

When the optional parameter ",B" is included in a LINE statement, the two pixels are not connected. Instead a box is drawn, using the two coordinates as opposite corners.

```
LINE (20,20) - (100,150), 2, B
```

If ",B" is replaced with ",BF", the box will be drawn. However, it will be filled with the specified attribute. As in the following example, if the attribute is omitted, the current default attribute will be used. Notice that the double comma is used to indicate the omitted attribute.

```
LINE (0,0) - (50,30),, BF
```

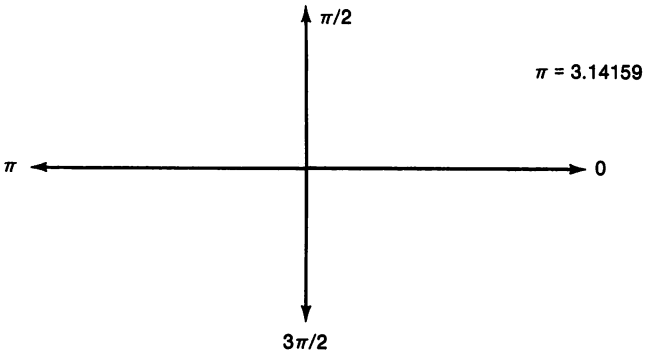
CIRCLE is used to draw an ellipse on the screen. Of cource, CIRCLE can also draw a circle, because a circle is merely a special case of an ellipse.

```
CIRCLE (80,100), 40, 2
```

The previous example will draw a circle centered at (80,100) with a radius of 40. The circle will be drawn using attribute 2. If only part of a circle (arc) is to be drawn, the optional *start* and *end* parameters should be used.

```
CIRCLE (100,100), 40,, 0, 3.14159
```

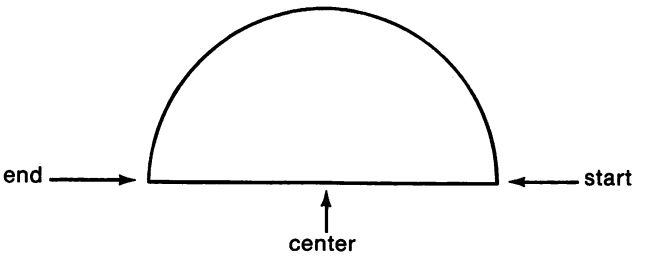
This example draws a semi-circle, centered at (100,100) with a radius of 40. The semi-circle will be drawn with the default attribute since the attribute parameter has been omitted. The two numbers, 0 and 3.14159 are the starting and ending angles in radians.



If the start or end angles are negative, the PC*jr* will treat them as positive values and will connect the ellipse to its center.

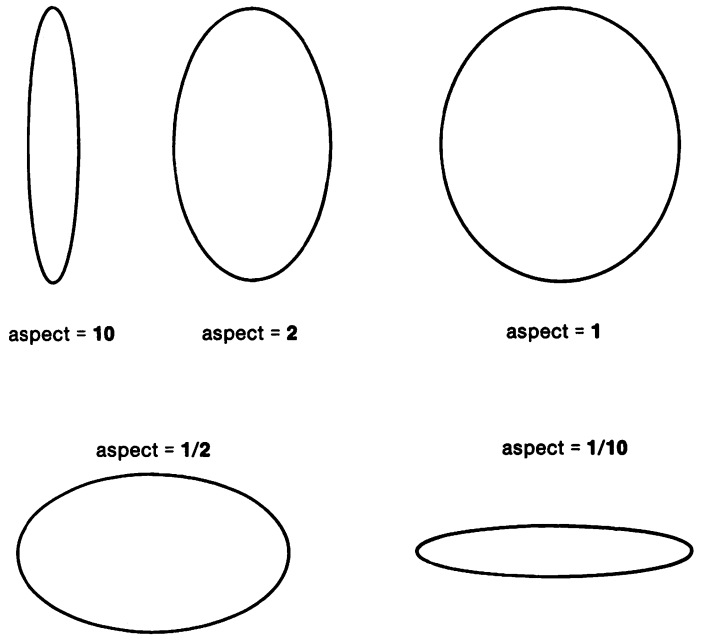
```
CIRCLE (160,100), 80,, -6.28, -3.14
```

The previous statements will draw a semi-circle with its starting and ending points connected to its center.



The final optional parameter is called **aspect**. The aspect determines the ellipse's elongation. When aspect =1, a nearly perfect circle will be drawn in screen 4. For values less than one, a flat ellipse will be drawn. For values greater than one, a narrow ellipse will be drawn. For example, the following statement will draw a long narrow ellipse:

```
CIRCLE (160,100), 80,,,,10
```



The third advanced graphics command is PAINT. This command will fill an enclosed section of the screen with a user determined attribute. The correct syntax of this command is as follows:

PAINT (*col, row*) [,*paint*][,*boundary*]

col and *row* determine where the filling process will begin. *paint* determines the attribute used for PAINT. If *paint* is omitted, the maximum attribute for the current graphics screen will be assumed. *boundary* is the attribute where painting is to be halted. For example, if a circle is drawn using attribute two, it may be filled with attribute ten in the following manner. If *boundary* is omitted, the *paint* attribute will be assumed.

```
CIRCLE (80,100), 40, 2
PAINT (80,100), 10, 2
```

The following example program uses the majority of the graphics commands learned thus far:

```
100 CLEAR ,,,32768
110 DIM PAL%(30)
120 SCREEN 5
130 KEY OFF
140 WINDOW (-10,-10)-(10,10)
150 FOR RADIUS = 13 TO 1 STEP -1
160       CIRCLE (0,0),RADIUS,RADIUS
170       PAINT (0,0),RADIUS
180 NEXT RADIUS
190 FOR K = 0 TO 30
200          PAL%(K) = K MOD 16
210 NEXT K
220 FOR K = 15 TO 0 STEP -1
230       PALETTE USING PAL%(K)
240 NEXT K
250 GOTO 220
```

The CLEAR statement at line 100 reserves 32768 bytes for screen memory. (The amount needed for graphics screen five.) The DIM statement in the next line dimensions an integer array to be used in the PALETTE USING statement at line 230. KEY OFF merely turns off the function key labels on line 25 of the display.

Lines 140-180 draw and paint the concentric circles. Lines 190-210 initialize the array which will be used in the loop at lines 220-240. This loop strobes the colors, giving the illusion of movement. Line 250 endlessly repeats the motion loop.

RETRIEVING INFORMATION FROM THE SCREEN

After information has been output to the screen, it may become necessary to determine which color is displayed at a certain screen position. The function POINT takes as its arguments the row and column numbers, and returns the attribute number. The correct syntax of the POINT function is as follows:

X = POINT (*col, row*)

col represents the column number. *row* represents the row number. Upon

execution of the preceding command, X will be assigned the value of the attribute at screen location column = *col*, row = *row*.

SHAPE DEFINITION

Often, a particular shape must be drawn a number of times in the same graphics screen. For example, 50 stars must be drawn in order to create an American flag. The DRAW statement is used to draw an object defined by a string containing drawing commands. These commands are collectively known as a graphics definition language (GDL™). The configuration for DRAW is given below:

DRAW *string*

string is string expression containing GDL commands. When DRAW is executed, each GDL command in the string will be interpreted and executed one by one.

DRAW — MOVEMENT COMMANDS

The GDL commands can be categorized as movement commands, set angle commands, set color commands, set scale factor commands, and execute substring commands. We will discuss the movement commands first. These commands are described as follows:

U*n*	Move up
D*n*	Move down
L*n*	Move left
R*n*	Move right
E*n*	Move diagonally up and right
F*n*	Move diagonally down and right
G*n*	Move diagonally down and left.
H*n*	Move diagonally up and left

Each of the movement commands is executed in a similar manner. Each indicates a direction in which the plotting will take place. A numeric argument (*n*) follows the commands which indicate the distance to be moved.

n does not specify the actual number of points to be moved. This can be calculated by multiplying *n* times the scaling factor. The scaling factor

is set with the scale command using the following configuration:

S*n*

n can range from 1 to 255. The scale factor is calculated by dividing *n* by 4. Therefore, the effective range for the scale factor is .25 to 63.75.

The following program illustrates the use of the movement commands and the scale factor:

```
10 SCREEN 4
20 COLOR 3,1
30 CLS
40 A$ = "U30  R40  D30  L40"
50 FOR X = 1 TO 10
60      B$ = "S" + STR$(X) + A$
70      DRAW B$
80 NEXT X
90 END
```

Line 40 defines a square as A$. This square is drawn at 10 different scales by the loop at lines 50-80. Notice that although 40 points were plotted along the x-axis and only 30 points were plotted along the y-axis, that our figure was drawn as a square. This is due to the fact that the screen's aspect ratio in medium resolution graphics is 4:3, which means that four horizontal points occupy the same amount of space as do 3 vertical points.

In high resolution, the aspect ratio is 16:7, while the aspect ratio in low resolution is 2:3. Aspect ratio will not affect the diagonal move commands. This is shown in the following example program:

```
10 SCREEN 4
20 CLS
30 A$ = "E40  F40  G40  H40"
40 DRAW A$
50 END
```

DRAW-M COMMAND

Although the U, D, L, R, E, F, G, and H commands are useful for drawing a figure, they are less useful in situations where a line is to be drawn from the LPR to a specific screen location. The M (movement)

command is provided for this purpose. M is used with the following configuration:

M*x,y*

x,y specify the coordinates to which a line is to be drawn from the LPR. If a plus (+) or minus (-) sign prefix *x* and *y*, the move will be relative. Otherwise, the move will be absolute. This is illustrated in the following example:

```
10 SCREEN 4
20 CLS
30 A$ = "U30  R40  D30  L40"
40 DRAW A$
50 M$ = "M +20,+30
60 DRAW M$
70 N$ = "M200,100"
80 DRAW N$
```

DRAW-B COMMAND

If the B command is placed in front of any GDL movement command, the move will be executed, but no points will be plotted. By editing line 70 of the previous program as follows:

```
70 N$ = "BM 200,100"
```

the effect of B will be illustrated.

DRAW-N COMMAND

If GDL's N command is placed in front of a movement command, the line will be drawn but the LPR will remain unchanged. The effect of N can be shown by editing line 30 as follows:

```
30 A$ = "U30  R40  D30  NL40"
```

DRAW-C COMMAND

The C command can be used to set the color for the shape being drawn. C is used with the following configuration:

C*n*

n specifies the attribute number. Its allowable range depends on the current graphics mode. The following program illustrates the use of the C command:

```
10 SCREEN 4
20 CLS
30 A$ = "C1  U30  R40  C2  D30  R40"
40 DRAW A$
50 END
```

DRAW-A COMMAND

The A command is used to set a rotational angle for any subsequent GDL commands. The format for A is as follows:

A*n*

n can be any value from 0 to 3, where 0 indicates 0 degrees, 1 indicates 90 degrees, 2 indicates 180 degrees, and 3 indicates 270 degrees.

DRAW-TA COMMAND

The TA command can be used to turn the figure at the angle indicated by *n*. *n* can range from +360 to -360. If *n* is negative, the angle turns clockwise. The use of TA is illustrated in the following program:

```
10 SCREEN 4
20 CLS
30 FOR X = 0 TO 360 STEP 45
40      DRAW "TA" + STR$(X)+"U30"
50 NEXT X
```

DRAW-P COMMAND

The P command is the GDL equivalent of the PAINT command. P specifies the attribute that the figure is to be filled with, as well as the border attribute. P is used with the following configuration:

P *paint, boundary*

paint can range from 0-15 and specifies the attribute of the figure. *boundary* indicates the border attributes of the figure to be filled in. *boundary* can also range from 0-15. Both parameters must be specified.

```
10 SCREEN 4
20 CLS
30 A$ = "U30  R40  D30  L40"
40 B$ = "BE10"
50 C$ = "P1,3"
60 DRAW A$ + B$ + C$
```

DRAW-X COMMAND

The X command allows a DRAW statement to execute a substring from within another command string. The substring is generally specified with a variable name. It must be preceded by X and followed by a semicolon.

The X command allows the user to define a part of a drawing as a separate part from the definition of the entire object. This command also facilitates the defining of objects which need more than 255 characters for their definition.

```
10 SCREEN 4
20 CLS
30 A$ = "E50  D50  L50"
40 B$ = "A1" + A$
50 C$ = "A2" + A$
60 D$ = "A3" + A$
70 DRAW "XA$; XB$; XC$; XD$;"
80 END
```

With any of the commands in the GDL, the argument *n* can either be a constant or it can be a variable. If *n* is a variable, it must be preceded by an equal sign (=). Also, a semicolon must be used to delimit commands where a variable is used for *n*. The various GDL commands are summarized in table 7-5.

Table 7-5. DRAW's GDL Commands

Command Format	Reference
U*n*	Move up.
D*n*	Move down.
L*n*	Move left.
R*n*	Move right.
E*n*	Move diagonally up and right.
F*n*	Move diagonally down and right.
Gn	Move diagonally down and left.
H*n*	Move diagonally up and left.
M*xy*	Move absolute or relative. If *x* is preceded by a plus (+) or minus (-) sign, it is relative. If not, it is absolute.
B	Moves as indicated without the plotting of any points.
N	Moves per the command but returns to the original position when the movement has been completed.
A*n*	Sets an angle at the value indicated by *n*. *n* can be any value from 0 to 3, where 0 indicates 0 degrees, 1 indicates 90 degrees, 2 indicates 180 degrees, and 3 indicates 270 degrees.
C*n*	Sets the color to the value indicated by *n*. *n* can range from 0 to 3 in medium resolution or from 0 to 1 in high resolution.

Command Format	Reference
P *paint, boundary*	Sets the color of the figure to the color specified by *paint* (0-3). *boundary* indicates the border color of the figure to be filled.
S*n*	Sets the scale factor. *n* can range from 1 to 255. To calculate the scale factor, divide *n* by 4.
TA*n*	Turns at the angle indicated by *n*. *n* can range from +360 to -360. If *n* is positive, the angle turns counterclockwise. If *n* is negative, the angle turns clockwise.
X*string*	Executes the substring. This allows the user to execute a second string from within the original string.

Sound

As mentioned earlier, the PC*jr* contains a programmable tone generator. The PC*jr* is also equipped with the audio hardware of the PC, a 8253 timer. The tone generator can produce top quality music and sound effects. The less capable 8253 timer was included so that the PC*jr* could emulate the PC. Either set of hardware may be software selected. However, both may not be active simultaneously. The following two commands select which set of hardware will be activated:

SOUND ON	activate programmable tone generator
SOUND OFF	set PC emulation mode

The PC emulation mode is the default setting.

In addition to its two means of sound generation, the PC*jr* has three ways of outputting that sound. These include the internal speaker, an audio jack, and a modulated output to be used by a television set.

Although the sound of the 8253 timer can be directed to any of these outputs, the programmable tone generator may not use the internal speaker. Therefore, whenever the tone generator is activated (SOUND ON), the internal speaker will be deactivated. Likewise, the internal speaker will be activated whenever the 8253 is being used.

PC*jr* BASIC contains another pair of commands that activate the external audio outputs. The external outputs are the audio jack and the TV modulated output. These are controlled by the BEEP commands.

BEEP ON	activate external connections
BEEP OFF	deactivate external connections

By using the aforementioned rules, if BEEP is OFF and SOUND is ON, it would seem that all audio outputs would be disabled. SOUND ON would turn off the internal speaker, and BEEP OFF would deactivate the external audio connections. In this combination, PC*jr* BASIC will activate the external audio connections, to assure an active output channel. Table 7-6 illustrates the possible combinations of BEEP and SOUND.

Table 7-6. BEEP & SOUND

SOUND	BEEP	INTERNAL SPEAKER	EXTERNAL CONNECTIONS
ON	ON	OFF	ON
ON	OFF	OFF	ON
*OFF	*ON	ON	ON
OFF	OFF	ON	OFF

When activated, the 8253 timer can be programmed to play any single note. The duration of this note may also be programmed. However, when compared to the capabilities of the SN76489A, the 8253 is dwarfed. The programmable tone generator can play any three notes, can independently control the volume on each of these notes, and can pro-

* default setting

duce sound effects. The duration of these notes may again be programmed. Therefore, any sound produced on the 8253 can also be produced on the SN76489A. Henceforth, all PC*jr* BASIC commands will be discussed in reference to the SN76489A.

PRODUCING SOUNDS

Besides being used to enable the programmable tone generator, the SOUND statement has an alternate configuration that can be used to directly control the frequency, volume and duration of a note, played on any of the **voices**. A voice can be defined as an oscillator that can produce any one note. The SN78498A has 3 tonal voices and 1 noise voice. The SOUND command may be configured as follows:

SOUND *pitch, duration* [,[*volume*][,[*voice*]]]

pitch is the frequency in Hertz of the note to be played. *pitch* must lie between 37 and 32767. Any value from 37 to 110 will produce a 110 Hz note, the lowest attainable by the tone generator. The value 32767 will produce a period of silence. Table 7-7 contains a listing of musical notes and their corresponding frequencies. Notes in a higher octave may be obtained by doubling the frequency of the corresponding note in the previous octave. Notes in a lower octave may be obtained by halving the frequency of the corresponding note in the next octave.

Table 7-7. The Frequencies of Common Musical Notes

Note	Frequency (Hz)	Note	Frequency (Hz)
C	261.63	middle C	523.25
C#	277.18	C#	554.37
D	293.66	D	587.33
D#	311.13	D#	622.25
E	329.63	E	659.26
F	349.23	F	698.46
F#	369.99	F#	739.99
G	392.00	G	783.99
G#	415.30	G#	830.61
A	440.00	A	880.00
A#	466.16	A#	932.33
B	493.88	B	987.77

duration is the desired length of time that the note or silence is to be sustained. *duration* can be calculated by multiplying the desired delay time in seconds by 18.2. For example, to facilitate a 2 second delay, *duration* must equal 36.4 (2 x 18.2).

volume can range from 0 to 15, with 15 being loudest and 0 being silent. *voice* can range from 0 to 2. If this parameter is omitted, 0 will be assumed.

PC*jr* executes the SOUND command as follows — first, the tone generator will be instructed to produce the desired note. Then, the program will continue executing program lines until another SOUND statement is encountered. When encountered, the computer will wait until the first statement is completed (duration runs out). Then, it will execute the new SOUND statement in a manner similar to the first. However, if the new statement has a duration of zero, that voice will immediately be turned off.

When using Cartridge BASIC, the programmer has the option to **buffer** the SOUND statements. To buffer means to continue program execution even when a new SOUND statement is encountered. The program merely stores the musical information of the SOUND statement until the computer is able to execute it. Up to 32 unplayed SOUND

statements can be stored this way. When SOUND statements are being buffered, a zero value for duration will result in an error.

ADVANCED MUSIC

Activating all three voices requires three SOUND statements. While SOUND gets the job done, it is not the most efficient means of programming sound. A single PLAY statement can control any or all of the voices. PLAY is similar to DRAW in that it employs a "tune definition language." This tune definition language in turn controls the voices. The configuration for PLAY is as follows:

PLAY *music0$*[,[*music1$*][,*music2$*]]

music0$, music1$ and *music2$* are string constants or string expressions consisting of music commands. *music0$* controls voice #0; *music1$* controls voice #1; and *music2$* controls voice #2. If a string is left out, then that voice is not affected by the PLAY statement. For example, a statement like the following would skip voices 0 and 1, and use voice #2:

```
PLAY,,TUNE$
```

The commands in the "tune definition language" are listed in table 7-8.

Table 7-8. The Tune Definition Language

Command	Result
A-G # + -	The note given is played. # or + after the note means sharp, while - means flat.
L*x*	This is used to set the length of each note. L1 indicates a whole note; L2 a half note; L4 a quarter note; L16 a sixteenth note, etc... *x* may range from 1 to 64. If you wish to change the length for only one note, you can do so by placing the length directly after the letter for the note (ex. A4 would be a quarter note A).
MB	This causes music to play in the background. Each note or sound is placed in a buffer. This allows the BASIC program to continue execution as music plays in the background. As many as 32 notes can be played in the background at any one time. MB (music background) is the default value of MB and MF.
MF	This causes music to run in the foreground. Each note or sound will not begin to play until the preceding note or sound has finished playing.
ML	Music legato causes each note to be played at the entire length specified in L.
MN	Music normal, results in every note being played at 7/8th of the value given in L (length). This is the default setting of ML, MN and MS.
MS	Music staccato causes each note to be played at 3/4 of the time specified in L.
N*a*	This is used to play the note specified in *a*, where *a* can range from 0 to 84. Since there are 7 octaves available, there are 84 notes available. If *a* is 0, a rest is indicated.

Table 7-8. (cont.) Definition Language

Command	Result
O*b*	This is used to set the current octave. Seven octaves are available. These are numbered from 0 to 6. Each octave extends from C to B. Octave 3 begins with middle C.
P*c*	This is used to indicate a pause. The length of the pause (as given in *c*), may range from 1 to 64 and is calculated as in L (length).
T*d*	This is used to indicate the tempo. Tempo is defined by the number of quarter notes per second. *d* may range from 32 to 255. The default value for *d* is 120.
V*e*	V*e* adjusts the volume. The volume must lie between 0 and 15. This command is only valid when SOUND is ON.
	A period following a note causes it to be played as a dotted note (the note's length is multiplied by 1.5). More than one period may follow a note. If more than one period does follow a note, its length will be adjusted as indicated. For instance, "A..." will play 3.37 times as long as "A." Periods may also follow a pause to lengthen it in the same manner.
>*f*	This is used to play the note given in *f* in the next higher octave. Each time the note specified is played, the next higher octave is used, until octave 6 is attained.
<*g*	This functions as does >*f*. Each note is played in the next lower octave.
X *string$*	This causes the execution of the specified string $.

The following example program will play "Row, Row, Row Your Boat" in rounds. The three part harmony is replicated quite well by the PC*jr*.

```
100 'write the song
110   ROW1$ = "L4   CC   C8   E"
120   ROW2$ = "L8   EDEF   G2"
130   ROW3$ = "L4 >C<GEC"
140   ROW4$ = "L8   GFED   C2"
150   SONG$ = ROW1$ + ROW2$ + ROW3$ + ROW4$
160   SONG$ = SONG$ + SONG$
170 'place the song into rounds
180   V0$ = "O3" + SONG$
190   V1$ = "O4 P1" + SONG$
200   V2$ = "O5 P1 P1" + SONG$
210 'play the song
220   SOUND ON
230   PLAY V0$,V1$,V2$
```

SOUND EFFECTS

In addition to its three tonal voices, the SN76489A has a "noise voice." The noise voice, as used in sound effects, is produced by an irregular sequence of electrical impulses. These are generated using a shift register with exclusive — or feedback. Eight types of noise wave forms can be generated. Each produces a different sound. The programmer can select among these by using the NOISE command. NOISE is configured as follows:

NOISE *source, volume, duration*

source is a numeric constant or a numeric expression that evaluates in the range from 0 to 7. A range of 0-3 indicates a periodic noise, while a range of 4-7 indicates a white noise. Each type has a particular use in game programs. White noise sounds somewhat like hissing. It can be used to produce the sound of rocket engines or, depending on the frequency, the sound of a tire blowout. Periodic noise generates a more ragged sound that could be used to simulate a chain saw. The following table summarizes the available noises:

Periodic	White	Source	Sound
0	4	2330Hz	high option; less coarse hiss
1	5	1165Hz	medium pitch
2	6	582Hz	low pitch; more coarse hiss
3	7	freq. of voice #2	depends on voice #2

Notice that for *source* = 3 and *source* = 7, the frequency of voice #2 is used instead of a fixed frequency. By varying voice #2, even more complex sounds can be generated. Voice #2 can be controlled using either SOUND or PLAY. For example, the following program uses SOUND to vary voice #2 which in turn controls the NOISE output. Notice that although the volume of voice #2 has been set very low, the voice still controls the NOISE output.

```
100 SOUND ON
110 WHILE I>100 AND I>32000
120         NOISE 3,15,3
130         SOUND I,.00001,1,2
140         A$=INKEY$
150         IF A$="." THEN I=I*1.1
160         IF A$="," THEN I=I/1.1
170         PRINT I
180 WEND
190 I=4000
200 GOTO 110
```

By using the period (.) and comma (,) keys, the operator can vary the frequency of voice 2. This, in turn, controls the frequency of the noise output.

Writing a Game Program

In this section, the game, "BARRICADE" will be designed. The object of the game is to avoid the barricades as well as your own trail. The game will be written in BASIC so that it may be easily modified.

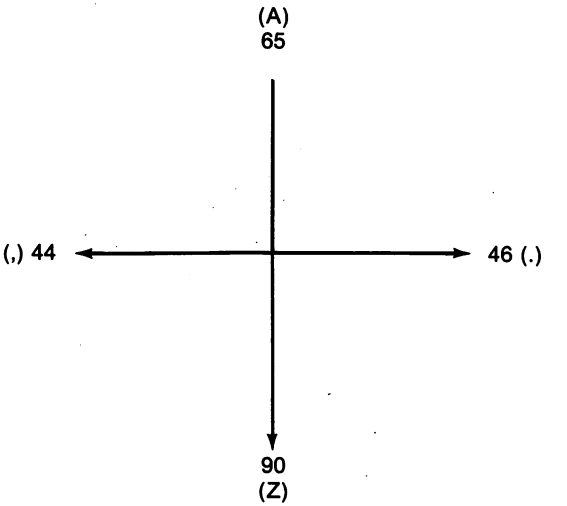
If the reader does not wish to follow the step-by-step designing of BARRICADE, he may page through the chapter. All program lines may easily be distinguished from the rest of the text. To play BARRICADE,

merely enter every line belonging to the program.

The first step in designing "BARRICADE" is to program the computer to draw a trail. The following statements accomplish this:

```
100 SCREEN 3
110 CLS
120 KEY OFF
150 COLOR 3
170 X = 80:Y = 100:A = 65
180 PSET (X,Y)
190 '********MAIN LOOP**********
200     A$ = INKEY$
210     IF A$<>"" THEN A = ASC(A$)
220       IF A = 44 THEN X = X-3
230       IF A = 46 THEN X = X+3
240       IF A = 65 THEN Y = Y-3
250       IF A = 90 THEN Y = Y+3
270     LINE -(X,Y)
310 GOTO 190
```

Line 100 enables low resolution graphics, while lines 110 and 120 clear the screen. Line 170 sets the initial position at screen location (80,100). This is the center of the screen. The A = 65 statement that also appears in line 170 selects "up" as the initial direction of movement.



These values represent the ASCII coded values for the controller keys: A, Z, period and comma.

Lines 190-310 set up a loop that will monitor the keyboard and act accordingly. Line 200 accepts a keypress. Line 210 sets the direction variable, A. Lines 220 through 250 recalculate the position variables based on the current direction. Executing the program is the best way to see how it operates. By the way, the program will not operate correctly unless the CapsLock key is active. If the program does seem to be operating correctly, try depressing the CapsLock key once.

Recall from our description of BARRICADE, that one of the rules was that the player was not allowed to collide with the trail. The POINT function will be used to check for collision. If the following line is added to the program collisions will be detected:

```
260 IF POINT (X,Y) > THEN 20
```

The program has not yet been completed. When it is run, an error occurs after every collision. This is because the computer does not know where to jump when there is a collision. Let's tell it, by adding the following lines to the program:

```
130 SOUND ON
320 '*******COLLISION ROUTINE********
330     LOCATE 12,7
340     PRINT "COLLISION"
350     NOISE 4,15,20
360     NOISE 5,12,20
370     NOISE 6,9,40
380     FOR J = 1 TO 15
390         PALETTE 0,J
400     NEXT
410     IF INKEY$<>CHR$(13) THEN 410
420 GOTO 100
```

Lines 330 and 340 print the word, "COLLISION" in the center of the screen. Lines 350-370 produce a noise that sounds something like an explosion. Lines 380-400 flash the screen with a number of colors. Finally, line 410 delays a new game from starting until Enter has been pressed.

BARRICADE's score can be easily kept track of by merely incrementing a variable each time a move has been completed.

```
160 I = 0
280         I = I+1
290         LOCATE 1,1
300         PRINT I;
```

A final technicality remains in the creation of a playing field. A square field with scattered circular barriers was chosen.

```
140 GOSUB 430:'DRAW PLAYING FIELD
430 '******DRAW PLAYING FIELD******
440     COLOR 9
450     LINE (  2,10)-(  4,199),,BF
460     LINE -(158,197),,BF
470     LINE -(156,  12),,BF
480     LINE (158,12)-(  2,  10),,BF
490     FOR J = 1 TO 6
500         X = 120*RND+20
510         Y = 150*RND+30
520         CIRCLE (X,Y),10*RND+10
530         PAINT (X,Y)
540     NEXT
550 RETURN
```

The ideas in this section by no means exhaust the possibilities that could be added to "BARRICADE". Other upgrades might include: keeping track of the high score, or adding another player. The only two limiting factors are execution speed and one's imagination.

8

Logo on the PCjr

Introduction to Logo

Logo is a language which was originally designed to teach children about programming computers. However, due to the simplicity and power of Logo, it has become popular for many different age groups for a number of different applications.

Logo is best known for its excellent graphics capabilities. Simple commands can be used to create graphics. This chapter will focus specifically on Logo's graphics capabilities.

The version of Logo which will be discussed in this chapter is the disk based IBM Logo version 1.00. This chapter is not meant to be a reference chapter for Logo, but rather an introductory tutorial for the inexperienced Logo programmer.

Loading the Logo Diskette

Logo can be loaded in two different ways depending on the present state of the computer. The loading procedure is detailed below. If the PC*jr* is presently off, begin at step 1. If the PC*jr* is presently powered on, begin at step 5.

1. Insert the Logo diskette into the PC*jr*'s disk drive.
2. Turn on the monitor, printer and any other peripheral device. Turn on the computer after all other peripherals have been turned on.
3. A pause of 3 to 45 seconds will occur while the system check takes place. Following the system check, the disk drive will engage, and the Logo diskette's contents will begin to load into memory.
4. Proceed to step 8.
5. Insert the Logo diskette into the disk drive of the PC*jr*.
6. Press the CTRL and Alt keys, and while holding them, press the Del key. Then, release all three.
7. The computer's memory will now be cleared. The disk drive will engage and the contents of the Logo diskette will begin to load into memory.
8. A message will appear prompting the user the enter a new date. The date must be entered in the form *month-day-year*. The following shows the prompt which will be displayed along with the user response. In this example, the date 6-23-83 was entered:

```
A>DATE
Current date is Tue 1-01-1980
Enter new date: 6-23-83
```

9. Following the date prompt, a message will appear prompting the user to enter a new time. The time must be entered in the form *hours:minutes:seconds.hundredths of seconds**. The following shows the prompt which will be displayed along with the user response. In this example, the time 16:34 (4:34 PM) was entered.

```
A>TIME
Current time is 0:00:55.360
Enter new time: 16:34
```

---

* The *seconds* and *hundredths of seconds* may be omitted. These values will simply default to zero.

10. Following the time prompt, the disk drive will engage and Logo will begin to load into the computer's memory. Once Logo has been completely loaded into the computer's memory, the Logo prompt will appear on the display as follows:

```
IBM Personal Computer Logo Version 1.00
(C) Copyright IBM Corp. 1983
(C) Copyright LCSI 1983
Serial Number XXXXXXXX

WELCOME TO LOGO
?
```

At this point, Logo is ready and waiting for a command.

Exploring Turtle Graphics

"Turtle Graphics" is the term used to describe the type of graphics which Logo uses. Logo uses a small character, which is called a turtle, to create graphics. The turtle can be moved around the screen while leaving a trail. The turtle can be thought of as a pen. By moving the turtle, lines can be drawn.

FINDING THE TURTLE

When Logo is first loaded, the turtle cannot be seen. The command SHOWTURTLE is used to make the turtle visible. The SHOWTURTLE command can be abbreviated as ST.

EXAMPLE: ST
RESULT: The turtle becomes visible. ST has no affect if the turtle is already visible.

The turtle can be made to disappear by using the command HIDETURTLE. The HIDETURTLE command can be abbreviated as HT.

EXAMPLE: HT
RESULT: The turtle disappears. HT will have no affect if the turtle is not presently visible.

MOVING THE TURTLE

As stated earlier, the Logo turtle can be thought of as a pen. By moving the turtle, lines can be drawn on the screen. There are several commands which can be used to move the turtle. The FORWARD command moves the turtle forward a specified numbers of pixels.* Forward is defined as the direction in which the turtle is pointing. The FORWARD command can be abbreviated FD.

EXAMPLE: FD 60
RESULT: The turtle moves forward 60 pixels. A line is drawn from the turtle's starting point to its ending point.

The BACK command moves the turtle backwards a specified number of pixels. Backwards is defined as the opposite direction from which the turtle is pointing.The BACK command can be abbreviated BK.

EXAMPLE: BK 70
RESULT: The turtle moves backwards 60 pixels. A line is drawn from the turtle's starting point to its ending point.

When Logo is initially loaded, the turtle is located in the center of the screen and is pointing upwards. The direction in which the turtle is pointing can be changed by using either the RIGHT or LEFT commands.

The RIGHT command is used to turn the turtle a specified number of degrees. There are 360 degrees in a complete circle. Therefore, 90 degrees is 1/4 of a complete turn, 180 degrees is 1/2 of a complete turn, and so on. The RIGHT command will turn the turtle to the right the specified number of degrees, while the LEFT command will turn the turtle to the left the specified number of degrees. The RIGHT and LEFT commands can be abbreviated RT and LT respectively.

* A pixel is one screen element. A pixel can be thought of as the smallest dot which the computer can generate on the screen. There are 250 vertical pixels and 320 horizontal pixels.

EXAMPLE: RT 70
RESULT: The turtle will turn 70 degrees to the right.

EXAMPLE: LT 45
RESULT: The turtle will turn 45 degrees to the left.

We have now learned several commands which can be used to manipulate the turtle. It is time to use what we have learned to create a graphics display. First of all, clear the screen using the CLEAR SCREEN command. To do this, simply type CS and press enter (CS is the abbreviation for CLEAR SCREEN). Now, enter the following lines one at a time. Be sure to press the ENTER key after each line is entered. Observe the movements of the turtle as each line is entered.

```
FD 50
RT 90
FD 50
RT 90
FD 50
RT 90
FD 50
```

The turtle executed each command as it was entered. The result of the preceding lines is a square in the middle of the screen.

THE LOGO EDITOR

In the previous section, the graphics commands FORWARD (FD) and RIGHT (RT) were used to create a square on the screen. It took a total of seven lines to create a square on the screen. It would be a tedious task to type in the same seven commands more than once. Fortunately, procedures can be defined in Logo.

A **procedure** is a set of instructions which can be executed by using one command. For example, a procedure would contain all the instructions which were previously used to create the square.

Procedures are defined using the Logo Editor. To enter the Logo Editor, simply type the command EDIT followed by the name of the procedure. For example, to create a procedure called SQUARE, type the following command:

```
EDIT "SQUARE
```

The following should now be displayed on the screen:

```
TO SQUARE
```

At this point, the Logo Editor is waiting for the instructions that will define the procedure SQUARE. To define the procedure SQUARE, press the Enter key and type the following lines. The four arrow keys may be used to position the cursor. This feature is convenient for correcting typing errors.

```
FD  50
RT  90
FD  50
RT  90
FD  50
RT  90
FD  50
END
```

The last command entered is the END command. The END command must always be the last command entered when defining a procedure. When all the lines in a procedure have been entered, press the ESC (escape) key to complete the process.

Let's check the procedure we have just defined to see if it functions as desired. First of all, clear the screen by entering the CS command. Now, enter the command SQUARE. The turtle did what was expected.

```
EXAMPLE:  CS
          SQUARE
  RESULT: The screen is cleared and the instructions in the procedure
          SQUARE are executed.
```

We have now learned several Logo commands. We have also learned how to combine a number of commands into a procedure. The instructions in a procedure can be executed simply by entering the name of the procedure. Now, let's go on and discover more Logo commands.

LIFTING THE PEN

As stated earlier, the Logo turtle may be thought of as a pen. Whenever the turtle moves, a line is drawn from its starting point to its

ending point. However, at times it may be desirable to move the turtle without drawing a line. This can be accomplished by using the PENUP command (abbreviated PU). Executing the PU command is similar to lifting the pen off the paper. The turtle can now be moved around the screen without leaving a trail.

Similarly, the PENDOWN command (abbreviated PD), is used to put the pen back down on the paper. Executing a PD command will cause the turtle to leave a trail when it is moved around the screen.

EXAMPLE: PU
RESULT: The turtle will no longer leave a trail when it moves around the screen. A PD command must be executed before the turtle will leave a trail.

EXAMPLE: PD
RESULT: The turtle will leave a trail whenever it is moved around the screen.

REPEATING COMMANDS

In many instances, it may be desirable to repeat a command or commands. Logo's REPEAT command facilitates command repetition. The REPEAT command can be used to repeat a command a specified number of times. The following example illustrates the format of the REPEAT command:

EXAMPLE: REPEAT 20 [FD 5]
RESULT: The command FD 5 will be executed 20 times.
The previous example is identical to entering the command FD 5 20 times

The REPEAT command has many useful applications. For example, a circle can be drawn by executing one REPEAT command. A circle can be drawn by repeating the two commands FD1 and RT1 360 times. Let's draw a circle using the REPEAT command. First of all, clear the screen using the CS command. Next, enter the following line:

```
REPEAT 360 [FD 1 RT 1]
```

It will take the computer a few moments to complete the circle. This is because it has to repeat two instructions 360 times.

The size of the circle can be increased or decreased simply by changing the number of pixels moved for every degree turned. For example, a larger circle can be drawn using the following command:

```
REPEAT 360 [FD 2 RT 1]
```

A smaller circle can be drawn using the following command:

```
REPEAT 180 [FD 1 RT 2]
```

The REPEAT command could have been used to draw the square which we created earlier in this chapter. If you recall, the square consisted of two commands, which were repeated four times. Use the CS command to clear the screen and enter the following line:

```
REPEAT 4 [FD 50 RT 90]
```

The preceding command created a square. It can be seen that the REPEAT command is very useful when commands are to be executed more than once.

FILLING AN AREA OF THE SCREEN

In the preceding sections, we learned the process for drawing shapes such as circles and squares. Previously, however, only the outline of these shapes were drawn. It is possible to fill a closed shape by using the FILL command. The FILL command causes the closed shape which contains the turtle to be filled in.

Let's try the FILL command. First of all, clear the screen using the CS command (you have probably already noticed that the CS command also causes the turtle to return to the center of the SCREEN.) Now, create a circle using the following command:

```
REPEAT 360 [FD 1 RT 1]
```

Now, the turtle must be positioned inside the circle. First of all, lift the pen of the turtle using the PU command. Next, turn the turtle towards the center of the circle using the RT 90 command. Move the turtle into the circle by entering FD 20. In order to use the FILL command, the turtle's pen must be down. Therefore, enter the PD command. Finally,

enter the FILL command. The circle will now be filled in.

Let's create a procedure which will draw a square and fill it in. First, we must enter the Logo Editor by entering the EDIT command. The name of this procedure will be FILLEDSQUARE.

```
EDIT "FILLEDSQUARE
```

Now, enter the following lines. Remember to press Esc when all the lines have been entered.

```
CS
PD
REPEAT  4 [FD 50 RT 90]
PU
RT      45
FD      10
PD
FILL
END
```

The following steps should be followed to fill a particular area.

1. Raise the turtle's pen by using the PU command.
2. Move the turtle into the closed shape which is to be filled.
3. Put down the turtle's pen by using the PD command.
4. Enter the FILL command.

TURTLE COORDINATES

As stated earlier in this chapter, the display screen is made up of screen elements known as pixels. These pixels can be divided into a coordinate system with the center pixel being labeled 0,0. With this type of coordinate system, the X or horizontal coordinate can have a value ranging from -159 to 160. The Y or vertical coordinate can have a value ranging from -124 to 125.

This type of coordinate system allows the turtle to be relocated to a specific point. A simple command, the SETPOS command, can be used to relocate the turtle to any point on the screen.

EXAMPLE: SETPOS [20 40]
RESULT: The turtle will move from its present position to the coordinate position (20,40). A line will be drawn from its previous position to its new position if the pen is down.

The turtle's present position can be determined by using the command PRINT POS. PRINT POS will print the coordinates of the turtle's present position.

EXAMPLE: PRINT POS
RESULT: Two numbers will be displayed on the screen. The first number is the turtle's X coordinate and the second number is its Y coordinate.

If only one of the two coordinates needs to be changed, a specialized command can be used. For example, if only the turtle's X coordinate need be changed, the command SETX can be used. If only the turtle's Y coordinate need be changed, the command SETY can be used.

EXAMPLE: SETX 30
RESULT: The turtle will move to the X coordinate 30. The Y coordinate will not be changed. A line will be drawn from the old X coordinate to the new X coordinate.

EXAMPLE: SETY 70
RESULT: The turtle will move to the Y coordinate 70. The X coordinate will not be changed. A line will be drawn from the old Y coordinate to the new Y coordinate.

The turtle's home or starting position is (0,0). The HOME command will cause the turtle to return to its home position.

EXAMPLE: HOME
RESULT: The turtle will move to its home position (0,0). A line will be drawn from the turtle's old position to the point (0,0).

Color

Up to this point, we have only used a black background and a white turtle. We will now show how to change the color of both the background and the turtle.

BACKGROUND COLOR

The background color can be changed by using the SETBG command. The background can be changed to any one of 16 colors. The 16 available background colors are listed in table 8-1.

TABLE 8-1. Background Colors

Number	Color	Number	Color
0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	High-Intensity White

EXAMPLE: SETBG 8
RESULT: The background color becomes gray.

EXAMPLE: SETBG 12
RESULT: The background color becomes light red.

The color code of the background color presently being used can be displayed using the following command:

PRINT BG

EXAMPLE: PRINT BG
RESULT: The color code of the present background color being used is displayed on the screen.

PEN COLOR

The color of the turtle, or the pen color, can be changed using the SETPC command. The SETPC command is used in conjunction with the SETPAL command to determine the pen color. Logo includes three different pen color numbers and two different palettes. Therefore, a total of six colors are possible. These colors, along with there respective color code combinations, are given in table 8-2.

TABLE 8-2. Pen Colors

SETPC	SETPAL	Resulting Color
1	0	Green
2	0	Red
3	0	Brown
1	1	Cyan
2	1	Magenta
3	1	White

EXAMPLE: SETPAL 0
SETPC 2
RESULT: The turtle's pen will turn red, and any lines drawn by the turtle will be red.

EXAMPLE: SETPAL 1
SETPC 3
RESULT: The turtle's pen will turn white, and any lines drawn by the turtle will be white.

The present value of the pen color and the palette can be displayed using the PRINT command.

EXAMPLE: PRINT PC
RESULT: Displays the present value of the pen color.

EXAMPLE: PRINT PAL
RESULT: Displays the present value of the palette.

CLEAN-UP

We have already encountered the CLEARSCREEN command (CS). The CS command clears the graphics screen and returns the turtle to its home position. The CLEAN command also clears the graphics screen. However, the CLEAN command does not cause the turtle to return to its home position. Instead, the turtle will remain at its present position.

Neither of the preceding commands will clear the text at the bottom of the screen. The CLEARTEXT (or CT) command is used to clear the text at the bottom of the screen. The CT command will clear only the text and will not affect the graphics screen.

SOUND

Logo has the capability of generating a wide range of musical sounds. All sound is controlled by one command, the TONE command. The TONE command is followed by two parameters. The first parameter is the frequency of the desired note. The frequency can range from 37 to 19723. Table 8-3 includes a list of the more popular musical notes and their corresponding frequencies.

The second parameter of the TONE command is the duration for which the specified frequency is to be generated. The duration can range from 0 to 9999. A duration of 18 is approximately one second.

TABLE 8-3. Musical Notes and Frequencies

Note	Frequency	Note	Frequency	Note	Frequency
C	130.810	F	349.230	B	987.770
D	146.830	G	392.000	C	1046.800
E	164.810	A	440.000	D	1174.700
F	174.610	B	493.880	E	1318.500
G	196.000	C*	523.250	F	1396.900
A	220.000	D	587.330	G	1568.000
B	246.940	E	659.260	A	1760.000
C	261.630	F	698.460	B	1975.500
D	293.660	G	783.990		
E	329.630	A	880.000		

EXAMPLE: TONE 392 32
RESULT: A G note will be generated for approximately two seconds.

EXAMPLE: TONE 880 128
RESULT: An A note will be generated for approximately eight seconds.

* Middle C

SAMPLE PROCEDURE

The following are sample procedures. Only commands which have been discussed in this chapter will be used.

PROCEDURE 1:

```
EDIT "CIRCLE
CS
REPEAT 360 [FD 1 RT 1]
PU
RT 90
FD 20
PD
FILL
END
```

PROCEDURE 2:

```
EDIT "SINEWAVE
CS
REPEAT 3 [REPEAT 90 [FD .93 RT 2]
          REPEAT 90 [FD .93 LT 2]]
RT 90
FD 320
END
```

PROCEDURE 3:

```
EDIT "FIGURE8
CS
REPEAT 180 [FD 1 RT 2]
REPEAT 180 [FD 1 LT 2]
END
```

PROCEDURE 4:

```
EDIT "3DBOX
CS
REPEAT 4 [FD 60 RT 90]
FD 60
RT 45
FD 60
RT 45
FD 60
RT 135
FD 60
LT 45
FD 60
LT 135
FD 60
LT 45
FD 60
HT
END
```

PROCEDURE HANDLING

At this point, we have defined several procedures. Also, several options are available. If you wish to review all the procedures which are currently defined, enter the command POPS.

EXAMPLE POPS
RESULT: All procedures along with their instructions will be displayed on the screeen.

If only one procedure is to be displayed, the PO command can be used. Entering PO followed by the name of the procedure enclosed in quotation marks will cause that procedure to be displayed.

EXAMPLE: PO "SQUARE
RESULT: Displays the procedure SQUARE on the screen.

Procedures currently defined can be saved on disk for future use. The SAVE command will save all procedures under the specified file name.

EXAMPLE: SAVE "FILE1
RESULT: All procedures which are presently defined will be saved on disk under the name FILE1.

The LOAD command is used to retrieve procedures which have been previously stored on disk.

EXAMPLE: LOAD "FILE1
RESULT: Loads into memory all the procedures stored under the filename FILE1.

CONCLUSION

This chapter has explained only a few of the many capabilities of the Logo language. The user should now have a strong foundation upon which he or she can continue on to learn of all the capabilities of Logo.

9

DOS 2.1 Features

Introduction

DOS is an abbreviation for Disk Operating System. IBM DOS consists of a group of programs which allow the user to manage information on diskette. The DOS programs (commands) are themselves provided on a diskette known as the DOS diskette. The original DOS diskette should not be used in day to day operations -- copies should be used instead. Procedures for copying the original DOS diskette will be detailed in this chapter. DOS start up as well as DOS command usage will also be explained.

There are several versions of DOS that can be used with IBM computers. This chapter will focus specifically on version 2.1. This is the version that is used with the PC*jr*. The label on the DOS diskette should specify the version of DOS.

Notice that the DOS diskette does not have a write protect notch. If a diskette does not have a write protect notch, it is permanently write protected and no data can be written to it by the computer. This will

prevent the user from accidentally erasing data by overwriting it with new data.

NOTATION

DOS allows the user to enter data with uppercase letters, lowercase letters, or a combination of both. To prevent confusion, our examples will use upper and lowercase letters except for the names of DOS commands, program names, and filenames. These will be displayed in uppercase letters in our examples.

TYPES OF COMMANDS

There are two types of DOS commands on the DOS diskette. These are **internal** commands and **external** commands. Internal commands are those which are loaded into the memory when DOS is loaded. External commands are not loaded with the DOS, but remain on the DOS diskette. In order to access an external command, the DOS diskette must be inserted in the drive before the command is given. Table 9-1 shows which commands are internal and which are external.

TABLE 9-1. Internal and External Commands

Internal			External	
BREAK	GOTO	TIME	ASSIGN	MODE
CHDIR	IF	TYPE	BACKUP	MORE
CLS	MKDIR	VER	CHKDSK	PRINT
COPY	PATH	VERIFY	COMP	PROMPT
CTTY	PAUSE	VOL	DISKCOMP	RECOVER
DATE	REM		DISKCOPY	RESTORE
DIR	RENAME		EXE2BIN	SORT
ECHO	RMDIR		FIND	SYS
ERASE	SET		FORMAT	TREE
FOR	SHIFT		GRAPHICS	

STARTING DOS

Generally, DOS will be started whenever the computer is started. When starting (or loading) DOS, a copy of the DOS programs will be read from the DOS diskette and loaded into the computer's memory. The diskette drive will emit a series of noises while this is taking place. The steps involved in loading DOS with the computer off are as follows:

1. Insert DOS diskette in drive A and close the drive door.
2. Turn on printer, monitor, and then computer.
3. A pause of 3 to 45 seconds will occur while the system check takes place. The disk drive will emit a series of noises while DOS is being read into memory.

If the computer is already on, DOS can be loaded by performing a system reset. A system reset first causes the current contents in memory to be erased. DOS is then loaded into memory. A system reset is similar to turning the computer off and then on. The following steps should be taken to reset the system and load DOS:

1. Insert the DOS diskette in drive A and close the drive door.
2. Press the CTRL and ALT keys, and while holding them press the DEL key. Then, release all three.
3. The disk drive's in-use light will come on while DOS is being read into memory.

ENTERING THE DATE

When DOS has been loaded and is ready, a message similar to that shown below will be displayed on the screen:

```
Current date is Mon 4-01-1983
Enter new date:
```

The new date should be entered as follows:

1. Enter one or two numbers from 1 to 12 for the month.

2. Enter a dash (-) or a slash (/).
3. Enter one or two numbers from 1 to 31 for the day.
4. Enter a second dash or slash.
5. Enter two numbers between 80 and 99 for the year.
6. Press the Enter key.

DOS will check the date entry to be certain that it is valid. If not, the following will be displayed:

```
Invalid Date
Enter new date:
```

The following would all be valid entries for June 10, 1983:

```
6-10-83
6/10/83
06/10/83
```

ENTERING THE TIME

Earlier versions of DOS may not require a time entry. However, with version 2.1 of DOS, once the date has been entered, the following prompt will appear:

```
Current time is 0:01:21.53
Enter new time: _
```

Time is displayed in the following format:

Hour:Minutes:Seconds.Hundredths of Seconds

The numeric keys at the top of the keyboard will be used to set the time. The steps for setting the time are as follows:

1. Enter one or two numbers in the range (0-23) for the hours.
2. Enter a colon (:).
3. Enter one or two numbers in the range (0-59) for the minutes. To enter seconds and hundredths of seconds, continue with steps 4 through 7. Otherwise, skip to step 8, and DOS will set these values to zero.

4. Enter another colon (:).
5. Enter one or two numbers in the range (0-59) for seconds.
6. Enter a period (not a colon) to separate seconds from hundredths of seconds.
7. Enter two numbers in the range 00 to 99 for the hundredths of seconds.
8. Press the Enter key.

The following would all be valid entries for 10 A.M.:

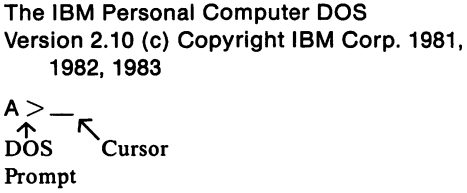
```
10:00
10:00:0.00
10:00:0
```

If an invalid entry was made for time, the following message will be displayed:

```
Invalid Time
Enter new time: _
```

DOS PROMPT

Once a valid time has been entered, the following will be displayed on the screen:



A> is known as the DOS **prompt.** A prompt is an indication to the user that he or she is to enter data. Note that the DOS prompt is a letter (A or B). This letter identifies which drive is the default drive (or current drive). The default drive is the drive that DOS assumes will contain the diskette or files specified by the commands. The default drive can be changed from A to B by entering the following:

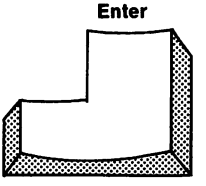
```
A > b:
B > _
```

Although the PC*jr* has only one disk drive, the drive identifier B can still be used. The same drive will now be identified as drive B. The utility of this feature will become evident later in this chapter and in more advanced applications.

DOS Keyboard Usage

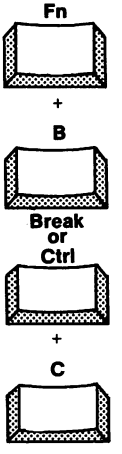
In the next few pages, we will discuss how some of the keys on the IBM keyboard are used with DOS.

ENTERING A COMMAND



The Enter key is used to execute or start a command after it has been typed in.

STOPPING A COMMAND



A command can be stopped in one of two ways. One way is to hold down the Function key while pressing the Break key. When both keys are released, the command will stop. The DOS prompt will appear, and the next command can be entered.

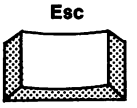
The second way to stop a command is to hold down the Control key while pressing the C key.

CORRECTING TYPING ERRORS

There are several ways of correcting errors in a DOS command. We will discuss two of the most commonly used here, the Backspace and the

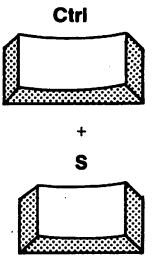
Escape keys. Note that mistakes can only be corrected before the Enter key has been pressed to execute the command.

The Backspace key moves the cursor one character to the left. The Backspace key deletes characters as the cursor is moved to the left. Once the Backspace key has deleted characters, the correct characters can be entered.



The Escape key can be used to erase an entire line. A backslash will appear, and the cursor will move down one line on the screen where the command can be re-entered.

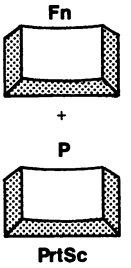
SLOW SCREEN SCROLLING



Scrolling is a term which is used to describe the process where information continues to move upward on the screen as new information replaces it on the bottom of the screen.

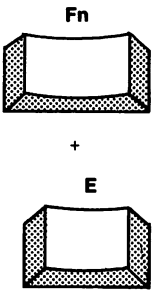
If scrolling is taking place too fast to read the data on the screen, press the Control and S keys together and then release them. To display new data, press any key.

SEND DATA ON THE SCREEN TO THE PRINTER



Data appearing on the screen can be sent to the printer as well. First of all, be sure that the printer is turned on and loaded with paper. Then, press and hold the Function key while pressing the PrtSc (Print Screen) key.

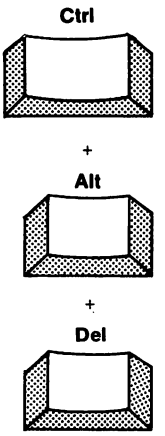
SEND KEYBOARD ENTRIES TO THE PRINTER



If a printer is connected to the system, keyboard entries can be sent to the printer as well as to the screen by pressing the Function key simultaneously with the Echo key. Then, release both keys.

All subsequent keyboard entries will be displayed on the screen and also simultaneously transmitted or **echoed** to the printer. To end this echoing, again press the Control and Print Screen keys simultaneously, and then release same.

SYSTEM RESET



As previously mentioned, to begin DOS over again, be sure a DOS diskette is in drive A. Then, press and hold the Ctrl and Alt keys and press the Del key. Then, release all three keys. DOS will be reloaded and the DOS start up message will appear.

DOS EDITING KEYS

The DOS editing keys can be used to correct DOS commands. The DOS editing keys should be used to edit within a line. To operate on one or more complete lines within a file, use the Line Editor program (EDLIN). EDLIN is discussed later in this chapter. However, if EDLIN is active and editing is to take place within a line, the DOS editing keys can be used for that purpose.

One principle to keep in mind when entering DOS command lines is that the line entered via the keyboard is retained in an input buffer when Enter is pressed. Since this line remains in the input buffer, it can be used as a **template** which can be edited as desired by the DOS editing keys. The user has the option of repeating the template line, changing it with the DOS editing keys, or entering an entirely new line. The DOS editing keys are summarized in Table 9-2.

TABLE 9-2. DOS Editing Keys

Key	Function
Del	Skips over one character in the template without moving the cursor
Esc	Cancels the line currently being displayed without affecting the template
F1	Copies and displays one character from the template
F2	Copies all characters up to the character specified
F3	Copies all characters remaining in the template to the screen
F4	Skips over all characters prior to the character specified
F5	Causes the line currently displayed to become the new template without sending it to the receiving program
Ins	Allows characters to be inserted within a line

DOS Usage

In the remainder of this chapter, we will discuss the application of DOS commands to everyday disk operating tasks. To clarify our examples, commands and user's responses will be underlined.

COPYING YOUR DOS DISKETTE

As mentioned previously, a copy of the DOS diskette should be made. This original DOS diskette is also known as the **master diskette.**

Copies, or **back-ups,** should be used in day to day operation. The master diskette should be stored in a safe place. That way, if the back up DOS diskette becomes damaged or misplaced, additional copies can be made from the master. The procedure for copying the DOS diskette is as follows:

Step 1. Start DOS and be certain that the DOS prompt (A>) is displayed. Do not remove the DOS diskette from the disk drive.

Step 2. Enter the following:

A > DISKCOPY ↵ ← denotes pressing Enter

The following prompts will be displayed:

```
Insert source diskette in drive A:
Strike any key when ready
```

The DOS diskette is the source diskette. Therefore, at this point, simply press any key.

Step 3. Swapping procedure. A prompt will now appear requesting insertion of the target diskette. The target diskette is the diskette which is to contain the copy of the DOS. This diskette can be a new diskette and does not have to be formatted. Insert the new diskette and press any key.

Prompts will be displayed describing which disk is to be inserted in the drive. Each time a prompt is displayed, insert the specified diskette and press any key. Remember, the source diskette is the DOS diskette and the target diskette is the new diskette. When the computer has completed making the DOS copy, the following prompt will be displayed:

```
Copy complete
Copy another (Y/N)?
```

At this point, press "N", and the DOS prompt will appear. The original DOS diskette should now be stored in a safe place. Use the copy for daily use.

FILES, FILENAMES, AND FILENAME MATCH CHARACTERS

At this point, you have gained some knowledge as to the organization of diskettes (chapter 1). You have gained an introductory knowledge of DOS and have learned how to copy your DOS diskette master. Before continuing our discussion of DOS, we will explain the concepts of files, filenames, and filename extensions.

A file is a group of related information. For example, a file might consist of a list of all the names, addresses, and phone numbers of your customers. A file might also contain the text of a standard form letter being sent to sales prospects. A file could also contain a program to edit and print the form letters. The advantage of grouping information in a file is that it can be easily retrieved by DOS, and can be stored on diskette when it is not being used.

A number of files can be stored on a single diskette. IBM DOS 2.1 allows up to 64 files per single sided diskette (112 for double-sided). Every file that is stored on the same diskette must have a unique filename. In other words, no two files on the same diskette can have the same filename. A filename consists of a primary filename and an optional filename extension. Examples of filenames are given below. Note that the second example does not contain a filename extension.

```
TEXT.DAT
PROGRAM1
NEW.TXT
ACCOUNT.BAS
```

IBM DOS allows primary filenames of up to eight characters in length. These characters can be the letters of the alphabet, integer numbers (0-9), or the following symbols:

```
$ # & @ ! % ( ) - { } ' _ '
```

The filename extension is an optional name that can appear after the primary filename. The filename extension begins with a period and either one, two, or three characters. When a filename extension is included in a filename, both the primary filename and the extension must be used to identify the file in DOS.

When inventing filenames, it is a good idea to make up a filename

that somehow identifies the information contained in that program. For example, the following would be a good name for a BASIC program file that printed a list of a firm's customers:

CUSTLIST.BAS

In situations where the same DOS operations are to be performed with several files, it may be advantageous to copy all of the data files on one diskette to another diskette. DOS includes two **filename match** characters. These allow the user to specify a number of files with just one specification. These characters are the asterisk (*) and the question mark (?). For example, if the current drive (A) contained the following files,

TEXT1.DAT
TEXT2.DAT
TEXT3.DAT
TEXT4.DAT
TEXT5.BAK

and the following DOS command was entered,

```
DIR TEXT?.DAT
```

all files except the TEXT5.BAK would be listed.

If the following DOS command was entered,

```
DIR TEXT?.???
```

all of the above files would be listed.

The asterisk (*) in a primary filename or filename extension is used to indicate that any character can be placed in that particular position in the rest of the filename. Using our previous example's files, if the following DOS command were entered,

```
DIR *.BAK
```

the following file would be listed:

TEXT5.BAK

If the following DOS command was entered,

```
DIR TEXT?.*
```

all of the files would be listed.

FORMATTING A DISKETTE -- FORMAT

Before a new diskette is used, it must first be formatted using the FORMAT command. The FORMAT command writes on every sector of a diskette, initializes the **directory,** initializes the **File Allocation Table,** and places a program known as the **boot record** at the beginning of the diskette.

The directory is used to store the names of the files on the diskette along with other data regarding the files such as size, location, and data created. The File Allocation Table is located adjacent to the directory on the diskette. The File Allocation Table keeps track of which sectors belong to which file, as well as available unused disk space. Each diskette contains one directory and two copies of the File Allocation Table. The second copy is used as insurance in case the first is unreadable for some reason.

FORMAT is an external command. Therefore, the DOS diskette must be inserted before the command is given. When the FORMAT command is given, directions as to how to proceed will appear on the screen. The following messages will be displayed when the FORMAT command is executed. Those parts which are underlined are the user's response.

```
A > FORMAT/V ↵
Insert new diskette for drive A:
and strike any key when ready

Formatting...Format complete

Volume label (11 characters, ENTER for none)? DISK1

  xxxxxx bytes total disk space
  xxxxxx bytes available on disk

Format another (Y/N)? N
A >
```

The preceding example shows the FORMAT command being issued along with the V or **volume** option. Including the /V with the FORMAT command allows the user to assign a name to the new diskette. The name of the diskette is referred to as its volume label. It is a good practice to include a volume label on all new diskettes. The volume label can become the identifying factor for all diskettes. In the preceding example, the volume label was given as DISK1.

When the formatting procedure has been completed, a prompt will appear which asks whether another diskette is to be formatted. At this point, simply press "N", and the usual DOS prompt will appear. The formatting process is now complete.

BACKING UP DISKETTES

Back up copies should always be made of important diskettes, so if a diskette becomes damaged or if a file is accidentally erased, the user will still have access to that information. IBM DOS includes two different commands for creating back up; DISKCOPY and COPY.

The DISKCOPY command is used to copy an entire diskette to another. DISKCOPY copies every file on the diskette including DOS in one operation. DISKCOPY is the fastest method of copying a diskette. DISKCOPY is only recommended when copying to new diskettes (formatted or unformatted). If the diskette being copied to either has defective tracks or has had a large amount of data written and then erased, the use of the COPY command is recommended.

The COPY command can also be used to copy one or more of a diskette's files to a different diskette. The COPY command is slower than the DISKCOPY command. However, it is the preferred method of copying to a diskette where a substantial amount of data writing and erasure has taken place. This is because unlike the DISKCOPY command which creates an exact duplicate of the original, the COPY command writes files sequentially -- or one right after the other. This sequential writing of files results in better performance, as the files are all grouped together. When files are copied with DISKCOPY, they may be spread throughout the diskette.

DISKCOPY

Since the PC*jr* contains only one diskette drive, the original diskette must be removed and replaced with the diskette on which the copy is to be made. This process is known as **swapping.** Once DISKCOPY has been executed, diskettes may have to be swapped as many as 4 or 5 times before the entire copying operation has been completed. To begin DISKCOPY, enter the following:

```
A > DISKCOPY ↵
```

DOS will display the following prompt when it is necessary to swap diskettes:

```
Insert source diskette in drive A:
Insert target diskette in drive A:
```

The source diskette is the original diskette. The target diskette is the back up. When the disk swapping process has been completed, the following message will appear:

```
Copy complete
Copy another (Y/N)?
```

If N is entered, DISKCOPY will end, and the DOS prompt will appear.

Once a disk has been copied, it is good practice to check for any errors on the newly copied disk. The DISKCOMP command can be used to compare two diskettes. If any differences are discovered between the two diskettes, an error will be displayed. To begin DISKCOMP, enter the following:

```
A > DISKCOMP ↵
```

As with DISKCOPY, DOS will prompt the user when it is necessary to swap diskettes. If a discrepancy occurs during the comparison, an error message will be displayed. When the comparison procedure has been completed, a message will be displayed stating so.

COPY -- COPYING A FILE TO THE SAME DISKETTE

As discussed previously, COPY can be used to make back up copies of one or more diskette files. Files can be copied to the same diskette (as long as a new filename is used), or to a different diskette The following configuration is used to copy a file to the same diskette with a different filename:

COPY *original backup*

original is the name of the file being copied, and *backup* is the name of the copy. Remember to always use different filenames when copying files to the same diskette. A diskette cannot contain two files with identical filenames.

COPY -- COPYING A FILE TO A DIFFERENT DISKETTE

Just as with DISKCOPY, since the PC*jr* contains only one drive, disk swapping will be necessary. The prompt messages used by COPY give the impression that the system contains two drives. To prevent confusion, think of COPY's prompt messages as referencing diskettes rather than drives.

Assuming a system drive contained a diskette containing DOS and a file named TEXTA, the following entry should be made to copy TEXTA:

```
A > COPY TEXTA B: ↵
```

The following prompt would then be displayed:

```
Insert diskette for drive B: and
strike any key when ready
```

The original diskette should then be replaced with the diskette which is to contain the copy of TEXTA. This diskette should have already been formatted. If the file being copied is large, it may be necessary to swap diskettes several times. If this is the case, DOS will prompt for the diskette swaps. When the copying process has been completed, the DOS prompt will appear.

Once a file has been copied, it is a good practice to check that it had been copied correctly. The COMP command can be used to accomplish

this. The COMP command will cause the computer to check for any discrepancies between the two specified files. The following is an example of COMP. In this case, the files TEXT1 and TEXT2 are compared. We have assumed that both files are on the same diskette and that DOS is resident on the diskette.

```
A > COMP TEXT1 TEXT 2 ↵
```

The previous example will compare two files residing on the same diskette. In order to compare files which reside on different diskettes, a drive specification must be given. The following example shows how files on different diskettes can be compared. The prompts will instruct the user which diskette is to be inserted in the drive. When the prompt asks for drive A, insert the first diskette. When the prompt asks for drive B, insert the second diskette.

```
A > COMP A:TEXT1  B:TEXT1 ↵

A:TEXT1
Insert diskette for drive B: and strike
any key when ready

and B:TEXT1

Eof mark not found

Files compare ok

Insert diskette for drive A: and strike
any key when ready

Compare more files (Y/N)? n
A >
```

DISKETTE DIRECTORY -- DIR

The Directory command (DIR) can be used to list one, several, or all of the files on a diskette. The following entry will list all files on the diskette that is inserted in drive A:

```
A > DIR ↵
```

The following information pertaining to each file would also be displayed:

Filename
Filename extension
File size in bytes
Date and time when data was last written to the file

After all of the files have been listed, the amount of free diskette space will be displayed in bytes.

In certain instances, it may be necessary to check a diskette to see if it includes a certain file. The following entry would check the diskette in drive A for the existence of TEXTA:

```
A > DIR TEXTA ↵
```

Filename match characters can also be used with DIR to display several files. The following entry would display all files on the diskette in drive A with the filename extension .BAS:

```
A > DIR *.BAS ↵
```

DISPLAYING A FILE'S CONTENTS -- TYPE

The TYPE command allows the user to display a file's contents on the screen. The following entry would display the contents of TEXTA on the screen:

```
A > TYPE TEXTA ↵
```

By pressing the Function and Echo keys simultaneously prior to the entry of the TYPE command, the file's contents will be sent to the printer as well as displayed on the screen.

RENAMING A FILE -- RENAME

The RENAME command is used to change a file's name. The following RENAME command will change the name of TEXTA to FILEA. Note that the "old" name precedes the "new" name in the command.

```
A > RENAME TEXTA FILEA ↵
```

ERASING A FILE -- ERASE

The ERASE command allows the user to remove files from a diskette that no longer are needed. The following commands will erase the file-name TEXTA from the diskette in drive A:

```
A > ERASE TEXTA ↵
```

The DEL command can be used interchangeably with the ERASE command. Both commands give the same results.

CLEARING THE SCREEN -- CLS

At times, the display screen may become cluttered with information. This would make it desirable to have some means of clearing the screen. Fortunately, DOS has a command that will clear the screen and return the cursor to the upper left-hand corner of the display screen. This command is the CLS command. Upon execution, the CLS command will clear the entire screen and return the cursor to its upper left-hand side. The following example illustrates the correct syntax for the CLS command:

```
A > CLS ↵
```

CHANGING THE DOS PROMPT -- PROMPT

By this point, the DOS prompt (A>) should be familiar. This prompt can be changed to any group of characters desired using DOS's PROMPT command. Any text entered following the PROMPT command will become the new prompt. The following example shows the prompt being changed to "ENTER COMMAND":

```
operator types → A > PROMPT ENTER COMMAND ↵
                 ENTER COMMAND _      ← cursor
```

Several special characters can be used to create variable prompts. These special characters will generate the prompts listed in table 9-3. The "$" character must precede the special character to display the variable

prompt. The following example illustrates the use of the special character "T" to set the time as the new DOS prompt:

```
A > PROMPT $T
3:37:22.18 _
           ↑cursor
```

TABLE 9-3. Special Prompt Characters

Character	Prompt
$	Displays "$" character
T	Displays current time
D	Displays current date
P	Displays the current directory of the default drive
V	Displays the version of DOS (version number)
N	Displays the default drive letter
h	Causes a backspace and erases last character
E	Displays the escape character
_	Causes a carriage return and line feed

CHANGING THE DATE -- DATE

Earlier in this chapter, we learned how to enter the date when DOS is loaded. Now, we will learn how to change the date using the DATE command. The correct syntax for the DATE command is as follows:

```
A > DATE ↵
```

Once the DATE command has been entered, the current date will be displayed. (The current date is the date which was entered when DOS was loaded.) Following the current date, a prompt will be displayed requesting the new date. A new date may be entered at this point. Use the same format for entering the data as was used when DOS was loaded. The following is a sample of the prompts displayed when the DATE command is entered:

```
A > DATE ↵
Current date is Tue  1-24-1984
Enter new date:
```

CHANGING THE TIME -- TIME

Earlier in this chapter, we learned how to enter the time when DOS is loaded. Now, we will learn how to change the time using the TIME command. The correct syntax for the TIME command is as follows:

```
A > TIME ↵
```

Once the TIME command has been entered, the current time will be displayed. (The current time was initiated when DOS was loaded.) Following the current time, a prompt will be displayed requesting the new time. A new time may be entered at this point. Use the same format for entering the time as we did when DOS was loaded. The following is a sample of the prompts displayed when the TIME command is entered:

```
A > TIME ↵
Current time is 12:34:29.31
Enter new time:
```

CHANGING THE STANDARD I/O DEVICE -- CTTY

When DOS is loaded, the standard input and output device is set to the console (keyboard and display). The standard input and output device can be changed to any device which can be used for input and output. A modem, for example, can be used as an input and output device. A printer, however, cannot be used as an input and output device.

The CTTY command can be used to change the standard input and output device. The CTTY command must be followed by the device name

of the new input and output device. Some legal device names are AUX, COM1, and COM2. The following is a typical CTTY command that will change the input and output device to COM2:

```
A>CTTY COM2 ↵
```

DETERMINING THE VERSION OF DOS -- VER

When DOS was first loaded, a message appeared stating the version of DOS being used. The version of DOS being used can be verified by entering the VER command. VER results in a message being displayed which states the version of DOS being used. The following illustrates the usage of VER:

```
A>VER ↵
IBM Personal Computer DOS Version 2.10
A>
```

ANALYZING A DISKETTE -- CHKDSK

Frequently used diskettes should be regularly checked for errors. DOS's CHKDSK command will display the status of the diskette and optionally correct any errors. CHKDSK (also known as the Check Disk command) is used to analyze the directories and File Allocation Table and to produce a disk and memory status report. The following configuration should be used with the CHKDSK command. Information within the brackets are optional.

CHKDSK [*filename*][/F][/V]

If a *filename* is specified, CHKDSK will display the number of non-continuous areas that are occupied by the specified file. /F causes CHKDSK to automatically correct errors found in the directory and File Allocation Table. /V causes CHKDSK to display a set of messages which provide information on the command's progress and the errors it discovers. After the diskette has been checked, any pertinent error messages will be displayed along with the following status report:

```
Volume DSKAA      Created MAY 12, 1984, 11.00

      xxxxxx bytes total disk space
      xxxxxx bytes in x hidden files
         xxx bytes in x directories
       xxxxx bytes of x user files
      xxxxxx bytes available on disk

      xxxxxx bytes total memory
      xxxxxx bytes free
```

The x's will be replaced by integers in an actual display. It is recommended that CHKDSK be periodically run on frequently used diskettes to check for errors and determine available free space.

SCREEN DUMPING -- GRAPHICS

The act of copying an entire screen of graphics onto the printer is known as a **screen dump.** Screen dumping is supported by DOS 2.1. DOS's GRAPHICS command can be used to prepare the PC*jr* for a screen dump. The GRAPHICS command has no arguments or parameters. Simply enter the command, GRAPHICS, while in DOS. The computer is now ready to copy a graphics screen onto the printer.

The following example illustrates the procedure involved in a screen dump:

Step 1. Load DOS and enter the GRAPHICS command.

A > GRAPHICS ↵

The computer will respond with the DOS prompt.

Step 2. A graphics screen will now be created. This example will create a graphics screen in BASIC. Type BASICA to enter BASIC*. Once BASIC has been loaded, enter the following program:

* The BASIC cartridge must be inserted in the proper slot.

```
10 LPRINT CHR$(27);"A";CHR$(8)
20 CLS
30 SCREEN 2
40 FOR I = 1 TO 100
50 LINE -(640*RND, 200*RND)
60 NEXT I
```

This program will create a random graphics screen. Line 10 is an escape character which instructs the printer to print nine lines per inch.* Lines 20 through 60 create the graphic screen.**

Step 3. Once the program has been loaded, enter the RUN command. Lines will be drawn randomly on the screen.

Step 4. When the drawing has been completed, hold down the Function and PrtSc keys simultaneously. The entire graphics screen will be printed on the printer. This process may take a few minutes.

CHECKING DATA WRITTEN TO A FILE -- VERIFY

When the computer writes data, such as a file, onto the disk, it normally does not check whether or not that data was written properly. The VERIFY command can be used to cause the computer to automatically check whether data was correctly written to the disk. The following examples illustrate the three possible syntaxes for the VERIFY command:

```
A > VERIFY ON ↵
A > VERIFY OFF ↵
A > VERIFY ↵
```

The first example turns on VERIFY. This means that every time the computer writes to the disk, it will automatically check to be sure that the data was properly written.

---

* Line 10 is a specific command for the IBM parallel printers. If a different printer is being used, line 10 may have to be changed.

** Refer to chapter 7 for an explanation of the graphics commands.

The second example turns off VERIFY. The computer will not check whether data was properly written onto the disk.

When VERIFY is entered without any parameters (example 3), the current status of VERIFY (ON or OFF) will be displayed.

SAVING A DAMAGED DISKETTE -- RECOVER

Diskettes that are used frequently or mishandled may become damaged. If any sectors containing data are damaged, all of the data on that sector will be lost. If a sector containing the directory is damaged, all of the files in that portion of the directory can no longer be accessed as usual.

Fortunately, DOS 2.1 provides a method for recovering data from all sectors of the diskette not directly damaged. The RECOVER command causes the computer to recover any or all files on a diskette.

Specifying a filename after RECOVER will cause the computer to recover only the specified file (minus any data lost because of a bad sector). If the RECOVER command is entered without a filename, the entire diskette will be recovered (minus any data lost in bad sectors).

An important note to keep in mind is that when an entire diskette is recovered, the computer will automatically rename all the files that have been recovered. The filenames will be in the form FILEnnnn.REC*.

The following two examples provide the correct syntax for the RECOVER command. The first example will recover the file TEXT5. The second example will recover the entire diskette in drive A.

```
A > RECOVER TEXT5 ↵
A > RECOVER A: ↵
```

* Files will be numbered in the order in which they were recovered. The number of the recovered file will be displayed in place of the nnnn in the filename. A typical directory of a recovered disk might be as follows:

```
FILE0001  REC      5120   1-01-80   12:12a
FILE0002  REC     17408   1-01-80   12:12a
FILE0003  REC     18432   1-01-80   12:12a
FILE0004  REC      2048   1-01-80   12:12a
FILE0005  REC      6144   1-01-80   12:12a
FILE0006  REC      7168   1-01-80   12:12a
```

CHECKING THE VOLUME LABEL OF A DISKETTE -- VOL

Earlier in this chapter, we learned how to format a diskette as well as how to assign a volume label to a diskette. The volume label is used strictly for identification and organizational purposes.

The VOL command can be used to read the volume label of a particular diskette. The correct syntax for the VOL command is shown below:

```
A > VOL ↵

Volume in drive A is REPFILES
```

TRANSFERRING THE OPERATING SYSTEM -- SYS

The SYS command is used to transfer the operating system from the default drive to the drive specified in the command. The following illustrates the correct syntax for the SYS command. The letter following the SYS command specifies the drive to which the operating system is to be transferred.

```
A > SYS A: ↵
```

The following files will be copied to the drive specified when SYS is executed:

IBMBIO.COM
IBMDOS.COM

When using SYS, the diskette in the drive specified by the command must previously have been formatted with the FORMAT command and /S or /B options so as to allocate the directory entries for IBMBIO.COM and IBMDOS.COM. If this is not done, SYS will not be able to transfer the operating system files.

The SYS command was designed to be used to transfer a copy of the DOS files to an applications program diskette formatted to use DOS, but sold without DOS.

CONTROLLING THE BREAK KEY -- BREAK

The BREAK command can be used to cause DOS to check for a control break whenever the program requests a DOS operation (ex. disk access). The BREAK command can appear in one of three forms. If the keyword ON is entered following BREAK, DOS will check for a control break whenever a program requests any DOS function. If the keyword OFF is entered following BREAK, DOS will only check for a control break during screen, keyboard, printer, or asynchronous communications operations. When BREAK is encountered without parameters, the current status of BREAK (ON or OFF) will be displayed.

The following examples illustrate the correct syntax for BREAK:

```
A > BREAK ON ↵
A > BREAK OFF ↵
A > BREAK ↵
```

PRINTING A LIST OF DATA FILES -- PRINT

The PRINT command allows a queue (list) of data files to be output on the printer while the computer is undertaking other processing tasks. As many as ten *filenames* can be specified with PRINT. Only files in the current directory can be opened for printing. The following configuration is used with PRINT:

PRINT [[*d:*][*filename*][/T][/C][/P]...]

/T sets the terminate mode. In this mode, all queued files will be cancelled from the print operation. If a file is in the process of being printed, the printing will stop, a cancellation message will be generated, the paper will advance to the top of the page, and the printer's alarm will sound.

/C sets the cancel mode which allows the user to select one or more files which are to be cancelled from the print queue. The filename preceding /C and all subsequent filenames until a /P has been specified will be cancelled.

/P sets the print mode. The filename preceding /P and all subsequent filenames until a /C has been specified will be added to the printer queue.

If no parameters are given in the command line, a list of all the files in the print queue will be displayed.

When PRINT is executed for the first time, the following message will be displayed:

```
Name of list device [PRN]:
```

The user should then specify the output list device. This can be LPT1, LPT2, LPT3, PRN, COM1, COM2, or AUX. The default is PRN. PRN can be selected by just pressing the Enter key.

After PRINT has been executed, the files will be queued for printing in the order specified in the command line. After each file has been printed, the printer will advance to the top of the next page.

```
A > PRINT a:texta ↵
A > PRINT /T ↵
A > PRINT text?/C ↵
```

In the first example, texta is added to the print queue. In the second example, the printer queue is emptied. In the final example, any files in the print queue that match text? will be removed from the queue.

Filters

Filters are special DOS commands which can be used to manipulate or change files in some way. Filters can be used to examine or completely change a file. Filters normally use the standard output device for display purposes. However, the filter command's output can also be sent to a file.

Special characters can be used to redirect a filter's input and output. The less than symbol (<) can be used to direct a particular file as input to a filter. The greater than symbol (>) can be used to direct the output of a filter to a particular file. The double vertical slash (¦) can be used to cause the output of one command to be the input of a filter.

```
A > SORT < LIST ↵
A > SORT < LIST > OUTCOME ↵
A > DIR ¦ SORT ↵
```

The previous three examples illustrate several of the possible uses of the special redirectional characters (< > |). The first example will cause the file named LIST to be the input to the SORT filter. The output will be displayed on the default output device (usually the display screen unless changed using the CTTY command). The second example will use the file named LIST as the input to the SORT filter and the output will now go to a file named OUTCOME. The third example will direct the output of the DIR command as input to the SORT filter.

There are three filters in DOS version 2.1. These are FIND, MORE, and SORT. These three filters will be explained individually in the following three sections.

LOCATING A STRING IN A FILE -- FIND

At times, it may be desirable to locate only part of a file. The FIND filter can be used to do so. Any and all lines containing a specified string can be located using the FIND filter. The FIND filter must be followed by the string which is to be searched for. Following this string, a filename must be specified. The following is a typical FIND filter:

```
A>FIND "BUSINESS" TEXT1 ↵
```

The previous example will look for the string BUSINESS in the file named TEXT1. The computer will then display all lines containing the string BUSINESS. The entire line will be displayed.

More than one filename can be specified after the string specification in FILE. In this case, every occurrence of the specified string in each file will be displayed. These will be displayed in the same order in which they appeared in the files. If more than one file was specified, the occurrences in the first file will be displayed first, followed by the second, and so on.

Several options can be used with the FIND filter. These options are designated by single characters and must follow the keyword FIND. Each character which represents an option must be preceded by a slash (/) when used with the FIND filter. Table 9-4 describes these options and shows their corresponding character representations.

TABLE 9-4. FIND Options

Character Representation	Option Effect
V	Causes the FIND command to display all the lines that do not contain the specified string.
C	Causes the FIND command to display only the number of occurrences of the specified string in the specified file.
N	Causes the FIND command to display the line number along with the line containing the specified string.

```
A>FIND/V "BUSINESS" TEXT1 ↵
A>FIND/C "BUSINESS" TEXT1 ↵
A>FIND/N "BUSINESS" TEXT1 ↵
```

The previous three examples all designate the same file but perform different tasks. The first example will display all the lines that do not contain the word BUSINESS. The second example will display a number that represents the number of times the word BUSINESS appears in the file TEXT1. The third example will display all the lines containing the word BUSINESS along with their respective line numbers.

SCREEN FILLING -- MORE

Earlier in this chapter, we discussed the TYPE command and its usage for displaying a file's contents on the screen. This is fine for short files, but what would happen if the file contained more information than the screen could hold. Using the TYPE command with such a file would cause the contents of that file to scroll off the screen before it could be read. The only information visible would be the last page. Fortunately, a DOS filter command is available which solves this problem. The MORE

filter command can be used to display a file's contents one page at a time.

The MORE filter will display one page of a file and pause. The prompt "--MORE--" will appear at the bottom of the screen. At this point, pressing any key will cause the next page of the file to be displayed. Once again, the prompt "--MORE--" will appear at the bottom of the screen. Press any key to display the next page. This process of displaying one page at a time will continue until the file's entire contents has been displayed. If MORE is to be exited before the entire file has been displayed, press CTRL-C or Fn-BREAK. The DOS prompt will then appear.

The following example uses the MORE filter. In this case, the file named TEXT1 is piped into the MORE filter. The file TEXT1 will be displayed one page at a time.

```
A >MORE <TEXT1 ↵
```

SORTING -- SORT

DOS version 2.1's SORT filter can be used to sort a file's lines by ASCII code either in ascending or descending order. The following examples describe usage of SORT along with all of its options.

Example 1: Sorting a File in Ascending Order.

The following example will sort the file CUSTLIST and store the newly sorted file into the file called NEWLIST. If ">NEWLIST" had not been included in this example, the sorted file would have been displayed on the screen. It would not have been saved as a file.

```
A >SORT <CUSTLIST >NEWLIST ↵
```

Example 2: Sorting a File in Descending Order

By including a /R after the SORT command, a reverse sort can be accomplished. The following example will reverse sort the file named CUSTLIST and store that newly sorted file as NEWLIST. If ">NEWLIST" had not been included in this example, the reverse sorted file would have merely been displayed on the screen. It would not have been saved as a file.

```
A >SORT/R <CUSTLIST >NEWLIST ↵
```

Example 3: Sorting a Specified Column

The SORT filter can sort a specified column in a file. For example, the SORT filter could be used to sort a file according to the ninth column of each line. The column where the sort should take place must be specified in the following format:

$$/+n$$

where *n* is the column number.

The following example will sort the directory according to column 14. Column 14 of the directory is the size of each file. Therefore, the following example will sort the directory in ascending order according to file size:

```
A>DIR¦ SORT/+14
```

Batch Processing

Batch processing allows automatic execution of a file with a filename extension of .BAT. Naturally, this file must have been created prior to its execution as a batch file. The proper configuration for executing a batch file is given below. The items enclosed in brackets are optional.

[*drive specifier:*] *filename* [*parameters*]

Notice that only the primary filename and not the filename extension (.BAT) need be entered. The .BAT extension is assumed.

Batch processing can be temporarily stopped by pressing the FN and BREAK keys (or CTRL and C keys). When these keys are pressed, the following prompt will appear:

```
Terminate batch job (Y/N)?
```

If Y is entered, batch processing will be ended and the system prompt (A>) will appear. If N is entered, the current command only will be ended, and batch processing will continue with the next command in the file.

AUTOEXEC.BAT FILE

The AUTOEXEC.BAT file represents a special usage of batch processing. Whenever DOS is started, the command processor will search for a file with the filename AUTOEXEC.BAT. If this file is present, it will be executed automatically upon system start up.

CREATING A BATCH FILE

A batch file can be created either via the line editor (EDLIN) or by using the console as an input device. The line editor will be explained later in this chapter. We will explain the use of the console here.

Suppose that you wished to create an AUTOEXEC.BAT file that would automatically run a BASIC program called LIST.BAS. Your first entry would be as follows:

```
A >COPY CON: AUTOEXEC.BAT ↵
```

The preceding command instructs DOS to copy from the console device into AUTOEXEC.BAT. Your next entry would be as follows:

```
BASIC LIST ↵
```

This entry places a command in AUTOEXEC.BAT to load BASIC and run the LIST program.

Press the F6 key followed by Enter to mark the end of the AUTOEXEC.BAT file and end the use of CON: for input. Now, when DOS is started, BASIC will automatically be loaded and the program LIST will be run.

USING REPLACEABLE PARAMETERS WITH .BAT FILES

When you enter the commands to be included in a batch file, you can include **replaceable parameters.** These replaceable parameters will be replaced by values specified when the batch file is executed. Up to 10 replaceable parameters can be used. These are %0, %1, %2, %3, %4, %5, %6, %7, %8 and %9. The dummy parameters are replaced one by one in ascending order. The first dummy parameter, %0 is always replaced by the filename of the batch file preceded by the drive designator (if one is

specified).

The following entries illustrate the use of replaceable parameters in a batch file:

```
A > COPY CON:FILE.BAT ↵
COPY %1.TXT %2.TXT ↵
TYPE %2.TXT ↵
TYPE %0.BAT ↵
F6 ↵    *
1 File copied
A >
```

The batch file named FILE.BAT will now be placed on the diskette in the current drive, A.

EXECUTING A BATCH FILE WITH REPLACEABLE PARAMETERS

To execute a batch file with replaceable parameters, simply enter the name of the batch file followed by the parameters to be substituted for %1, %2, %3, etc.

If the following were entered,

```
A > FILE.BAT A:OLD B:NEW ↵
```

FILE would be substituted for %0, A:OLD would be substituted for %1, and B:NEW would be substituted for %2.

BATCH SUBCOMMANDS

Batch subcommands are commands that can be executed directly from a batch file. Table 9-5 gives a listing of all the batch commands along with their proper configurations. Table 9-5 also includes a short explanation of each subcommand.

---

* F6 indicates that the F6 key is to be pressed.

TABLE 9-5. Batch Subcommands

Subcommand	Configuration	Reference
ECHO	ECHO [ON/OFF/*message*]	ECHO ON causes batch commands to be displayed on the screen as they are read from the batch file. ECHO OFF stops this display. ECHO ON is the default. *message* causes a message to be displayed even if ECHO has been turned off.
FOR	FOR %% *variable* IN (*set*) DO *command*	FOR allows the repeated execution of DOS commands. %%*variable* is set sequentially to each member of *set*. The *command* is then evaluated and executed.
GOTO	GOTO *label*	GOTO causes commands to be executed beginning with the line immediately after *label*. A *label* in a batch file is a character string where the first 8 characters are significant.
IF	IF [NOT] *condition command*	The IF subcommand allows conditional execution of DOS commands. The *condition* parameter is one of the following: ERRORLEVEL *number* *string*1 == *string*2 EXIST *filespec* When the IF condition is true, the DOS command is executed. Otherwise, it is passed by and the next command is executed. ERRORLEVEL *number* is true if the previous program had an exit code of *number* or higher.

Subcommand	Configuration	Reference
IF (*cont.*)	IF[NOT] *condition command (cont.)*	*string*1 == *string*2 is true when both are identical. EXIST *filespec* is true if *filespec* is found on the indicated drive.
SHIFT	SHIFT	SHIFT allows command lines to make greater use of the replaceable parameters (%0-%9). SHIFT shifts all parameters on the command line one position to the left with %0 being replaced by %1, etc. This allows over 10 parameters on the command line.
PAUSE	PAUSE [*remark*]	This subcommand suspends system processing and causes the following message to be displayed: Strike a key when ready...
REM	REM [*remark*]	This subcommand displays remarks within a batch file.

Introduction to EDLIN

DOS's line editor (EDLIN) can be used to create, modify, and display **source** or **text** files. A source file is an assembly language program in source language format which has not been assembled.

The text of the files which are being processed by EDLIN are divided into lines. The length of these lines can vary -- with a maximum of 253 characters per line. Numbers are automatically assigned to these lines. Although these line numbers are displayed during the editing session, they are not actually present within the file being edited.

EXECUTING EDLIN

EDLIN is executed using the following configuration:

EDLIN [*d:*] *filename* [/B]

If the file specified does exist on the current drive, the file will be loaded into memory until it is 75% full. If enough memory is available to load the entire file, the following message and prompt will be displayed:

```
End of input file
*_
```

If the entire file cannot be loaded, the file will be loaded line by line until memory has become 75% full. The EDLIN prompt (*) will then be displayed.

When only a portion of a file has been loaded, only those lines which have been loaded into memory can be edited. Before you can edit the remaining lines, some of the edited lines in memory must be written to the diskette so as to free available memory for the unedited lines.

If the /B parameter is not specified, EDLIN will cease the loading when it encounters Ctrl-Z (EOF mark). When /B is specified, the entire file will be loaded, regardless of any EOF characters.

If the file specified is not present on the designated drive, a new file with that name will be created. When this occurs, the following message and prompt will appear:

```
New file
*_
```

The asterisk is the EDLIN prompt

You may now enter lines into this new file after issuing an Insert lines command (discussed later).

When the editing process is ended, both the original and the edited files will be saved on a diskette. The original file will be renamed with the same primary filename and an extension of .BAK. The new file will have the filename specified in the EDLIN command.

When EDLIN is executed, if a file exists with the same primary name as the file specified and filename extension .BAK, that file will be erased. A new file with the same primary name as the file specified and extension $$$ will be created. This file will eventually be used to hold the edited file, and its filename extension will be changed from $$$ to that specified in the EDLIN command.

A file with an extension of .BAK cannot be edited. If you wish to edit such a file, first use the RENAME command to change the filename extension from .BAK.

EDLIN Command Summary

In this section, we will discuss the various commands that can be used with EDLIN. All EDLIN commands consist of a single letter except the Edit line command. Commands generally are used with parameters. Both commands and parameters can be entered in uppercase, lowercase, or a mixture of both.

Generally, commands and their parameters are separated by delimiters (spaces or commas) to improve their readability. However, this is not required. Delimiters are only required between two adjacent line numbers.

DOS commands can be aborted by pressing Fn-Break. A DOS command will be executed when Enter is pressed at the end of the command. If a large amount of data is being displayed as a result of a DOS command, you can hold the screen by pressing Ctrl-S. To continue the scrolling, press any key.

The following parameters will be used in our discussion of DOS commands:

line Three entries can be made for *line*. An integer from 1 to 65529 can be used to identify a particular line.

A period (.) can be used to identify the **current line.** The current line is used to mark the location of the last change in the file being edited. This does not have to be the last line displayed. The current line is identified by an asterisk which appears between the line number and first character in the line.

A pound sign (#) is used to identify the line after the final line in memory.

n This specifies the number of lines to be written to the diskette or read from the diskette.

The *n* parameter is used only with the Write Lines and Append Lines commands. These commands are used when the file being edited is too large to fit in memory.

string One or more characters are to be entered in place of this parameter. These characters represent text to either be searched for, replaced, deleted, or to be used in place of other text. The *string* parameter is only used with the search text and replace text parameters.

APPEND LINES EDLIN COMMAND

The Append Lines command is used to append lines from the diskette to the file in memory being edited. These lines are added to the end of that file.

Configuration

[*n*] A

The Append Lines command is only used when the file being edited is too large to fit in memory. If no *n* is included, the number of lines to be appended are not specified in the command. In this case, lines will be appended until available memory is 75% filled. When the last line of a diskette file has been appended into memory, the following message will be displayed:

End of input file

COPY LINES EDLIN COMMAND

The Copy Lines command copies the range of lines specified in the first two parameters to the line number specified by the third parameter. This new data is inserted before the line indicated in the third parameter.

The Copy Lines operation is repeated the number of times specified in the optional parameter *count*. *count* has a default value of 1.

Configuration

[*line*],[*line*],*line*[,*count*] C

If either of the first two parameters are omitted, the line being copied defaults to the current line. In effect, this results in the current line being copied to the line specified in the third parameter.

When Copy Lines is executed, the file's lines are renumbered to reflect the repositioned lines. The first line which has been copied will become the current line.

Example

```
1, 3, 8 C
```

In the preceding example, lines 1 through 3 are copied to line 8. Line 8 becomes the current line.

DELETE LINES EDLIN COMMAND

The Delete Lines command is used to delete a specified group of lines from the file being edited.

Configuration

[*line*][,*line*] D

When lines are deleted, the line following the last line deleted will become the current line in memory. The current line and those lines following it will be renumbered.

If both parameters are included, that range of lines will be deleted. If the first parameter is deleted, the deletion will begin with the current line and continue to the line specified by the second parameter (note the comma's inclusion to denote the second parameter).

If the second parameter is omitted (*line*D or *line,*D), only the single line named will be deleted. If both parameters are omitted, only the current line will be omitted.

EDIT LINE EDLIN COMMAND

This command is used to enter the line number of the line to be edited.

Configuration

[*line*]

If no entry is made (the Enter key is pressed), this indicates that the line following the current line is to be edited. When this command is executed, the line number entered and its contents will be displayed. The line number will then be repeated on the following line.

This may then be edited using the Control and Editing keys, or the line may be completely replaced by entering new text. When editing is complete, press Enter, and the edited line will be placed in the current file where it will become the current line.

If after reviewing the line displayed by Edit Line, you decide not to edit it, you can either press FN-Break or the Enter key (if the cursor is at the first position) to abort the command.

END EDIT EDLIN COMMAND

The End Edit command is used to end EDLIN and to save the file being edited on diskette.

Configuration

E

The file being edited will be stored on a diskette with the name specified when the EDLIN command was issued. The original version of the newly edited file will be saved with the filename extension .BAK.

When editing files, be certain that there is enough available diskette space to store both the original file and its backup. If enough space is not available, that part of the file in memory for which diskette space is not available will be lost. When the EDLIN command has been ended, the DOS prompt will appear.

INSERT LINES EDLIN COMMAND

The Insert Lines command is used to insert lines of text in front of the line number specified. If no line number is specified, the insertion will occur before the current line.

Configuration

[*line*] I

If the specified line number is greater than the highest existing line number, or is specified as #, the insertion will be made after the final line in memory.

As each new line is typed in, press the Enter key. A new line number will be displayed every time Enter is pressed. To end the Insert command, press the FN-Break keys.

The line following the inserted lines will become the current line. Both the current line and all following lines will be renumbered to reflect the insertion.

LIST LINES EDLIN COMMAND

The List Lines command is used to display the lines specified in the command. The current line is not changed by this command.

Configuration

[*line*][,*line*] L

If the first parameter is omitted, the display will begin 11 lines before the current line, and 23 lines in total will be displayed.

If the second parameter is omitted, a total of 23 lines will be displayed beginning with the specified line.

If both parameters are omitted, a total of 23 lines will be displayed. These include the 11 lines immediately preceding the current line, and the 11 lines immediately following the current line. If there are not 11 lines preceding the current line, additional lines following the current line will be displayed to total 23 lines.

MOVE LINES EDLIN COMMAND

The Move Lines command moves the range of lines indicated by the first two *line* parameters ahead of the line indicated by the third *line* parameter.

If either of the first two parameters are omitted, the parameter will default to the current line. Note that the third parameter is required.

Configuration

[*line*],[*line*],*line* M

After the move has been completed, the first line moved will become the current line. The lines are renumbered following the move.

Example

```
50M
```

In the preceding example, the current line is moved just ahead of line 50.

PAGE EDLIN COMMAND

The Page command lists the block of lines indicated by its parameters.

Configuration

[*line*][,*line*] P

If the first *line* is omitted, the default is the current line plus 1. If the second *line* is omitted, 23 lines are listed. Page differs from List Lines in that it changes the current line to the last line displayed.

QUIT EDIT EDLIN COMMAND

The Quit Edit command is used to end the editing session without any of the editing changes being saved.

Configuration

Q

When the Q command is used, the following prompt will appear:

```
Abort edit (Y/N)?
```

If you wish to exit EDLIN with no changes made in the original file, enter Y. Otherwise, enter N and continue editing.

Note that when you execute EDLIN, your previous backup copy

(with the .BAK filename extension) will have been erased. Therefore, exercise caution when using the Q command as your backup file will have been erased.

REPLACE TEXT EDLIN COMMAND

The Replace Text command is used to replace existing string data with new data.

Configuration

[*line*][,*line*][?] R *string* [F6*string*]

The Replace Text command will replace the characters specified in the first string with those specified in the second string in the range of lines given preceding the R. If a second string is not specified, the characters given in the first string will be deleted. The last line changed will be the current line.

The optional parameter ? is used to display a prompt (Ok?) after each line to be modified. If a Y or Enter is pressed, the modification will be made. Otherwise, the modification will not be made.

If the first *line* parameter is omitted, the search will begin with the line after the current line. If the second *line* parameter is omitted, the second line value will default to the last line. If both *line* parameters are omitted, all lines in memory will be searched.

Note that the F6 key is used to separate the two strings. A Ctrl-Z entry can also be used to separate these.

SEARCH TEXT EDLIN COMMAND

The Search Text command is used to search a specified range of lines for a designated string.

Configuration

[*line*][,*line*][?] S *string*

Unless the ? parameter is used, the first line to contain a matching string will be displayed and the search will end. This line will become the current line.

If no match is found, the following message will be displayed, and the

search will end. The current line will remain the same.

```
Not found
```

If the optional ? parameter is included, the following prompt will be displayed when a match is encountered:

```
Ok?
```

If Y or Enter is pressed in response to this prompt, the matching line will become the current line and the Search Text command will end.

If any other entry is made, the search will continue until another match is found or until all the lines in the specified range have been searched.

If the first *line* parameter is omitted, the default value will be the first line following the current line. If the second *line* parameter is omitted, the value will be the last line. If both are omitted, all lines from the line after the current line to the last line will be searched.

TRANSFER LINES EDLIN COMMAND

The Transfer Lines command is used to merge a specified file into the file that is currently being edited.

Configuration

[*line*] T [*d:*] *filename*

The *filename* specified will be inserted prior to the line specified. If *line* is omitted, the default will be the current line.

The file being merged will be read from the current directory of the drive indicated. If a path was specified in EDLIN, then the directory specified by that path will be the current directory, and any Transfer Lines command for that drive must indicate a file from that directory.

WRITE LINES EDLIN COMMAND

The Write Lines command is used to write to the diskette file from the lines being edited in memory. The lines are written beginning with the first line.

Configuration

[*n*] W

The Write Lines command need only be used in instances where the file being edited was too large to fit into memory. In these cases, edited lines in memory must be written to the diskette file before additional lines can be appended with the Append Lines command.

DOS COMPONENTS

The core of DOS consists of three programs, the boot record, IBMBIO.COM, and IBMDOS.COM.

The boot record is a program that lies at the very beginning of your DOS diskette. The boot record is automatically loaded into memory when DOS is started (or booted). The boot record then loads the remaining DOS programs. The boot record is placed on all diskettes by the FORMAT program.

IBMBIO.COM is a program which handles the transfer of data between the computer's memory and the various input and output devices attached to it. IBMBIO.COM is also placed on the diskette by the FORMAT program. IBMBIO.COM will not be included in a directory listing of a DOS diskette's files.

IBMDOS.COM includes a file management program and a number of service programs that are used to control programs written to run under DOS. Like IBMBIO.COM, IBMDOS.COM will not be included in a directory listing of a DOS diskette's files.

10

PCjr Communications

Communications with the PC*jr*

The IBM PC*jr* can be used as a terminal to communicate with other computers via phone lines. This is accomplished through the use of the IBM internal modem and a Terminal Emulator program.

INTERNAL MODEM

The IBM internal modem contains all the hardware necessary for phone line communications. The internal modem must be installed directly into the PC*jr*'s system unit. For a detailed explanation on installation, refer to the manual that accompanies the internal modem.

Once the internal modem has been properly installed in the system unit, the PC*jr* may be connected to the telephone lines. A cable is provided with the internal modem which must be used to connect the PC*jr* to a standard modular telephone jack. Insert one end of the cable into the port in the rear of the system unit marked with an M. Insert the

other end into a standard modular telephone jack. The modem is powered by the PC*jr* and is ready for operation whenever the PC*jr* is powered up.

TERMINAL EMULATOR

The internal modem supplies all the hardware necessary for communications over phone lines. However, a program is also necessary to utilize this available hardware. A program that is used with a modem for communications purposes is known as a Terminal Emulator program. The PC*jr*'s cartridge BASIC has a built-in Terminal Emulator. This program is designed to utilize the IBM internal modem for communications purposes.

IBM TERMINAL EMULATOR

The Terminal Emulator program that is built into the BASIC cartridge can be accessed directly from BASIC. The TERM command is used to load and run the Terminal Emulator program.

EXAMPLE: TERM
RESULT: The Terminal Emulator program is loaded into memory and begins execution.

Keep in mind that when the TERM command is entered, any program that was in memory will be lost and any open files will be closed. Any program in memory should be saved on disk before the TERM command is entered.

When the Terminal Emulator program begins execution, the screen will clear and the following message will be displayed:

(TERM) — TERMINAL EMULATOR

Following this message, the Terminal Emulator program checks to see if the internal modem is present. If the internal modem is present, the baud rate will automatically be set to 300.

The terminal selection menu will now be displayed on the screen. The terminal selection menu appears as follows:

```
(TERM)—Terminal Emulator

1 Line Bit Rate          [300] (300..4800)
2 Data Bits              [7] (7 or 8)
3 Parity                 [E] (E,O,N)

4 Host Echos             [Y] (Y or N)
5 Screen Width           [80] (40 or 80)

6 Modem Command [ ]

Change <line,data>?

f1-conv f2-exit f3-nul f4-break
```

The terminal selection menu allows transmission specifications to be programmed into the Terminal Emulator. The transmission specifications depend primarily on the capabilities of the host computer. The transmission specifications presently being used are enclosed in brackets. The parentheses enclose the options which may be chosen. Each transmission specification will now be discussed individually.

LINE BIT RATE (BAUD RATE)

The baud rate is the speed at which data will be transferred between the PC*jr* and the host computer. Several different baud rates can be specified in the Terminal Emulator program. However, the internal modem can only transmit and receive at a baud rate of 300. When the Terminal Emulator program is initially executed, the baud rate is set to 300 if the internal modem is present. If a device other than the internal modem is to be used, the baud rate can be increased.

CHANGING THE BAUD RATE

All specifications are changed by entering the line where the change is to occur, followed by the change itself. In the case of the baud rate, the line number is 1. Therefore, to change the baud rate, simply type the number 1 followed by a comma and the new baud rate. Acceptable values for baud rate are 300, 600, 1200, 1800, 2400, and 4800.

EXAMPLE: 1,1200
RESULT: The baud rate is changed to 1200. The new baud rate will appear in the brackets in line 1. The numbers in the parentheses show the possible range of values for the baud rate.

DATA BITS

The number of bits transmitted for each piece of data is known as **data bits**. The number of data bits can only be 7 or 8. The value chosen for the number of data bits depends entirely on the host computer. If the host computer transmits 7 data bits, then the data bits should be set to 7. If the host computer transmits 8 data bits, then the data bits should be set to 8. Consult the specifications of the host computer to determine the number of data bits to use.

CHANGING THE NUMBER OF DATA BITS

All specifications are changed by entering the line where the change is to occur followed by the change itself. In the case of the data bits, the line number will be 2. Therefore, to change the number of data bits, type the number 2 followed by a comma and the number of data bits desired.

EXAMPLE: 2,8
RESULT: The number of data bits will be changed to 8. The number of data bits currently being used will be displayed in brackets. The values enclosed in parentheses are values that could be used for the number of data bits.

PARITY

Parity is a method used by computers to check for errors that may have occurred during transmission. In order for this error detection method to work properly, both computers must be set on the same parity type.

Parity takes advantage of the fact that data is transmitted in sets of one's and zero's. An error can be detected simply by determining if an even or odd number of one's has been received.

There are two types of parity. The first type is even parity. Even parity means that both computers will transmit data with an even number of one's. If either computer receives data with an odd number of

one's, an error has occurred in the transmission.

The second type of parity is odd parity. Odd parity means that both computers will transmit data with an odd number of one's. If either computer receives an even number of one's, an error has occurred in the transmission.

The parity type that should be chosen depends entirely on the host computer. If the host computer uses even parity, then even parity should be used on the PC*jr*. If the host computer uses odd parity, then odd parity should be used on the PC*jr*. If the host computer uses no parity check, then the N option should be chosen. Consult the specifications of the host computer to determine the type of parity that should be used.

CHANGING THE PARITY

All specifications are changed by entering the line where the change is to occur followed by the change itself. In the case of parity, the line number is 3. The possible options for parity are EVEN (E), ODD (O), or NONE (N). To change the parity, type the number 3 followed by a comma and the desired parity option.

EXAMPLE: 3,O
RESULT: Changes the parity type to odd.
EXAMPLE: 3,N
RESULT: Turns off the parity check.

ECHOING

Echoing is the process by which the host computer returns the original message to the sender to verify that the message was properly received. If yes (Y) is specified for echo, the PC*jr* will not display what has been typed until the host computer sends back the original message. If no (N) is specified for echo, the PC*jr* will display all characters as they are typed. The PC*jr* will not wait for an echo from the host computer.

Whether or not an echo is specified depends entirely on the host computer. If the host computer does echo characters, then yes (Y) should be specified for echo. If the host computer does not echo characters, then no (N) should be specified for echo.

CHANGING THE ECHO STATUS

All specifications are changed by entering the line where the change is to occur followed by the change itself. In the case of echo, the line number will be 4. The possible options for echo are yes (Y) and no (N). To change the echo status, type the number 4 followed by a comma and either Y or N.

EXAMPLE: 4,N
RESULT: The PC*jr* will not wait for echoed characters and will display all characters as they are typed.

SCREEN WIDTH

The screen width specifies the number of characters that will be displayed on one line. The two possible options for screen width are 40 and 80. In order to display 80 characters per line, the PC*jr* being used must have 128K of memory. If the PC*jr* being used has only 64K of memory, only 40 characters can be displayed per line.

CHANGING THE SCREEN WIDTH

The screen width option appears on line 5. Therefore, a 5 must be specified to change the screen width. The line number should be followed by a comma and the desired screen width (40 or 80).

EXAMPLE: 5,80
RESULT: The screen width becomes 80 characters per line.

MODEM COMMANDS

Modem commands are specialized commands used with the IBM internal modem. If the IBM internal modem is not present, this option will not appear in the terminal selection menu.

There are several modem commands that can be used with the Terminal Emulator program. These commands will be discussed individually.

1. ANSWER command (A)

   The ANSWER command is used to put the modem in the Answer mode.

EXAMPLE: A
RESULT: The modem will respond to an incoming tone.

2. BREAK command (B)

   The BREAK command is used to send a break character for a specified amount of time. The argument of the BREAK command must be in hexadecimal. The argument specifies the multiple of 100 milliseconds at which a break character should be sent.

EXAMPLE: B 4
RESULT: The modem will send a break character for a period of 400 milliseconds.

3. COUNT command (C)

   The COUNT command will cause the modem to answer an incoming call after a specified number of rings. COUNT's argument specifies the number of rings for which the modem should wait. The argument must be in hexadecimal format.

EXAMPLE: C 7
RESULT: The modem will answer an incoming call after 7 rings.

4. DIAL command (D)

   The DIAL command causes the modem to automatically dial a specified number. The number can be up to 33 digits in length.

   Several options may be used with the DIAL command. These options are as follows:

   P: Specifying a P after the DIAL command causes the modem to wait for 10 seconds for a dial tone.

   W: Specifying a W before a set of numbers causes the modem to wait 5 seconds before continuing to dial. This feature is convenient when an access code is necessary to reach an outside line.

   I: Specifying an I before a set of numbers causes the following numbers to be rotary-dialed as opposed to tone-dialed.

EXAMPLE: DIAL P 9876543
RESULT: The modem will wait 10 seconds for a dial tone and then proceed to dial the specified number.

EXAMPLE: DIAL 9 W 9876543
RESULT: The modem will dial a 9, wait 5 seconds, then dial the given number.

5. FORMAT command (F)

   The FORMAT command is used to specify several transmission specifications. However, these transmission specifications have already been set in lines 1 through 5 of the Terminal Emulator program. Therefore, the FORMAT command need not be specified when using the Terminal Emulator program. The FORMAT command is generally used when accessing the modem through BASIC.

6. HANGUP command (H)

   The HANGUP command causes the modem to hang up the phone lines. This command does not apply when using the Terminal Emulator program, since Fn 1 will cause the modem to break all connections.

7. INITIALIZE command (I)

   The INITIALIZE command causes the modem to restart. This is identical to a cold start. When the INITIALIZE command is executed, all modem default options will be assumed.

EXAMPLE: I
RESULT: The modem performs a cold start on itself.

8. LONG RESPONSE command (L)

   The LONG RESPONSE command modifies the message feedback to either long or short. If a zero is used as the LONG RESPONSE argument, the message feedback will be set long. If a one is specified as the LONG RESPONSE argument, the message feedback will be set short.

EXAMPLE: L 1
RESULT: The message feedback will be set short.

9. MODEM command (M)

   The MODEM command sets the modem into the data state. The signal carrier is placed on the line:

EXAMPLE: M
RESULT: The modem is set into the data state.

10. NEW command (N)

    The NEW command is used to change the command character. The command character is only used when entering modem commands from BASIC. When using the Terminal Emulator program, command characters are not necessary.

11. ORIGINATE command (O)

    The ORIGINATE command causes the modem to go into the originate mode. The modem will generate the originate tone.

EXAMPLE: O
RESULT: The modem assumes the originate mode.

12. PICKUP command (P)

    The PICKUP command causes the modem to go into the voice state. Generally, this command will only be used in the BASIC command mode and not while executing the Terminal Emulator program.

13. QUERY command (Q)

    The QUERY command is used to display information regarding modem operation. Table 10-1 lists the possible characters returned by the QUERY command.

EXAMPLE: Q
RESULT: A modem status report will be displayed in the format depicted in table 10-1.

14. RETRY command (R)

    The RETRY command is used in conjunction with the DIAL command. The RETRY command will cause the modem to redial a number up to ten times if a busy signal is encountered.

Table 10-1. Possible Results of QUERY Command

Character display	Meaning
H0	Modem is presently off hook.
H1	Modem is presently on hook.
S#	The number following the S specifies the present setting of COUNT.
B	The number dialed gave a busy signal.
D	There is no dial tone. The line is dead.
L	The number was dialed and there is a connection.
N	The number was dialed but the dial tone is still present.
T0	Integrity test passed.
T1	Integrity test failed.

EXAMPLE: R
RESULT: The modem will redial the present number up to ten times if a busy signal is encountered.

15. SPEED command (S)

    The SPEED command is used to set the baud rate at which the modem will send and receive data. The SPEED command is not necessary when using the Terminal Emulator program since this option is already included in the main menu.

16. TRANSPARENT command (T)

    The TRANSPARENT command causes the computer to ignore command sequences which normally are directed to the modem. The modem will send every character it receives. This command is not necessary when using the Terminal Emulator program. The Terminal Emulator program has a separate conversation mode and main menu mode.

17. VOICE command (V)

    The VOICE command causes the modem to go into the voice state. This command is generally used in the BASIC command mode and not while executing the Terminal Emulator program.

18. WAIT command (W)

    The WAIT command causes the modem to stop any actions and wait for the next command from the host computer.

EXAMPLE: W
RESULT: The computer stops all actions and waits for a command from the host computer.

19. XMIT command (X)

    The XMIT command can only be used in the voice state. This command is used to send Dual Tone Modulated Frequency tone pairs while in the voice state. This command is generally used only in the BASIC command mode and not while executing the Terminal Emulator program.

20. ZTEST command (Z)

    The ZTEST command is used to place the modem in the test mode. The Terminal Emulator program is presently set to perform a hardware test when the modem is activated. Therefore, the ZTEST command need not be used with the Terminal Emulator program.

ENTERING A MODEM COMMAND

A modem command is entered by typing the number 6 followed by the modem command. If more than one modem command is to be entered, they should be separated by commas. When the modem commands are entered via the Terminal Emulator program, they will not be sent to the modem until Fn 1 is pressed. Fn 1 initiates the conversation mode and sends all modem commands to the modem.

EXAMPLE: 6, DIAL 9876543, R
RESULT: When Fn 1 is pressed, the modem will be instructed to dial the number 987-6543 and to keep dialing if a busy signal is encountered.

EXAMPLE: 6, COUNT 3, 0
RESULT: When Fn 1 is pressed, the modem will be instructed to answer an incoming call after three rings. The modem will be in the originate mode.

ERRORS

If an error is made while entering a line, that line will be ignored. No error message will be displayed. If an error is discovered in a line before ENTER is pressed, the line can be erased by pressing ESC. No other editing keys are functional.

FUNCTION KEYS

There are several special keys available that can be used in the Terminal Emulator program. These special function keys are described in table 10-2.

Table 10-2. Function Keys

Keys	Purpose
Fn + 1	Initiates the connection between the host computer and the PC*jr*. Also, all modem commands in line 6 will be sent to the modem. The Terminal Emulator program will now be in the conversation mode. If the Terminal Emulator program is presently in the conversation mode and Fn 1 is pressed, the Terminal Emulator program will return to the terminal menu. Any connection with the host computer will be broken.
Fn + 2	Causes the Terminal Emulator program to stop. The computer returns to normal BASIC operation. The Terminal Emulator program will still reside in memory and can be executed using a RUN statement.
Fn + 3	Sends a null character to the host computer. Fn 3 is only operational in the conversation mode.
Fn + 4	Sends a BREAK signal to the host computer. Fn 4 is only operational in the conversation mode.

USING THE TERMINAL EMULATOR PROGRAM

This section will provide a step by step approach to the use of the Terminal Emulator program for communicating with a host computer. The first step for communicating with a host computer is to acquire its communications specifications. This information is generally supplied to the subscriber of the host computer. For this example, the following communications specifications will be used:

Line Bit Rate (Baud Rate)	300
Data Bits	7
Parity Type	Odd
Host does not echo characters	
Host is in orginate mode.	

In order for the IBM PC*jr* to communicate with this particular host computer, the Terminal Emulator program must be used to change some of the communications specifications. The first step is to execute the Terminal Emulator program by entering the TERM command. The Terminal Emulator program will begin execution and the Terminal selection menu will be displayed on the screen as shown in figure 10-1.

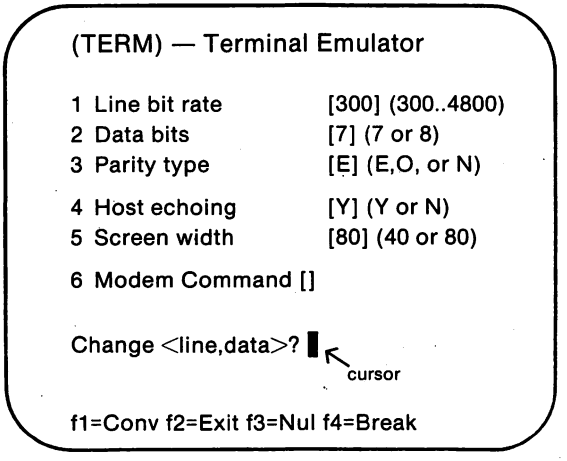


FIGURE 10-1. Terminal Selection Menu after Execution of TERM

Once the terminal selection menu appears on the screen, changes can be made in any or all of the communications specifications. For this example, the line bit rate and the number of data bits will not have to be changed. The parity type, however, must be changed from even to odd. This is accomplished by typing the line number where the change is to be made, in this case 3, followed by the desired change, in this case 0, for odd. The screen should appear as shown in figure 10-2.

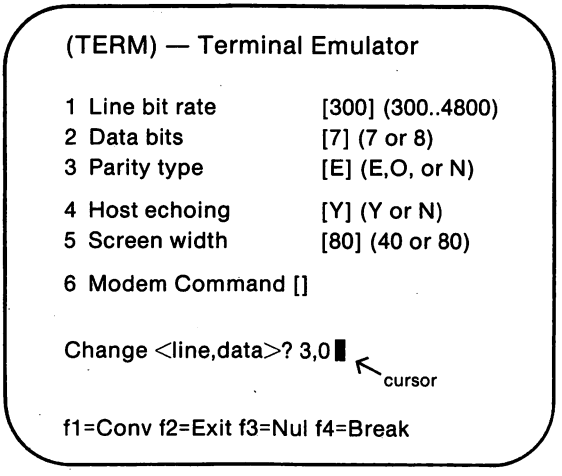


FIGURE 10-2. Terminal Selection Menu after Changes in Communications Specifications

The parity type will not be changed until the Enter key has been pressed. Once the Enter key has been pressed, the parity change will appear in line 3. The screen will resemble that depicted in figure 10-3 when the Enter key has been pressed.

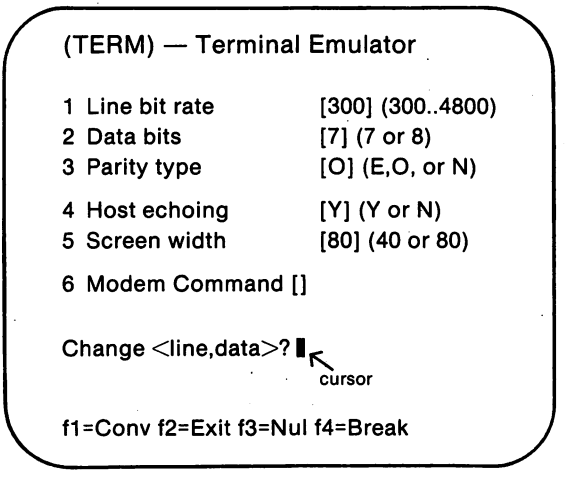


FIGURE 10-3. Terminal Emulator Selection Menu after Pressing Enter Key

The next change which must be made is in line 4. According to the communications specifications, the host computer does not echo characters. The Terminal Emulator is presently set for a host computer that does echo characters. This must be changed by typing the line number (4) followed by "N". The "N" signifies that the host computer does not echo characters.

As with the parity type, the change will not be made until the Enter key has been pressed. Pressing the Enter key at this point will cause the change in the echo status to be made. The screen should appear as shown in figure 10-4 after the Enter key was pressed.

The final changes that must be entered via the terminal emulator program are the modem commands. The modem commands that will be used in this example are DIAL, RETRY, and ANSWER. The DIAL command will be used to instruct the modem to dial the number of the host computer. The RETRY command will be used to redial the number of the host computer if a busy signal is encountered. The ANSWER command will be used to place the modem in the answer mode. This command is necessary since the host computer is the originate mode.

All three of the modem commands that will be used can be abbreviated by their first character. When the modem commands are typed in, the screen should appear as depicted in figure 10-5.

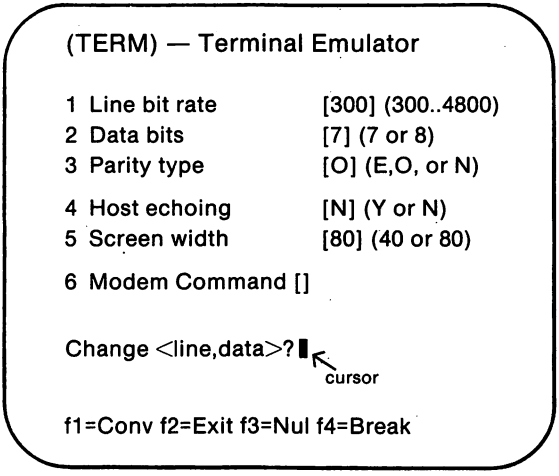


FIGURE 10-4. Terminal Emulator Menu after Adjusting Host Echoing

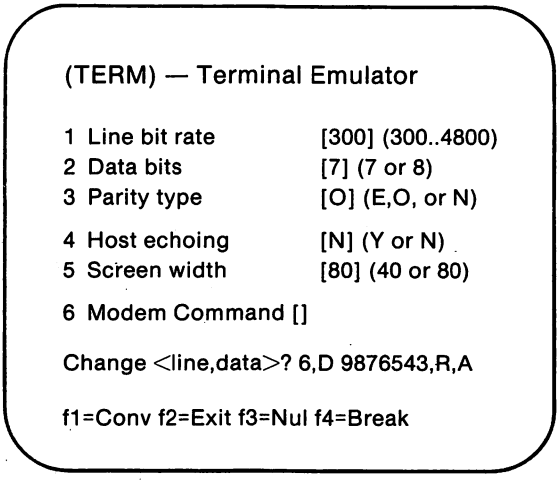


FIGURE 10-5. Terminal Emulator Menu after Modem Command Entries

The number following the DIAL command is the number of the host computer. Each command must be separated by a comma. At this point, press the Enter key and the screen will appear as shown in figure 10-6.

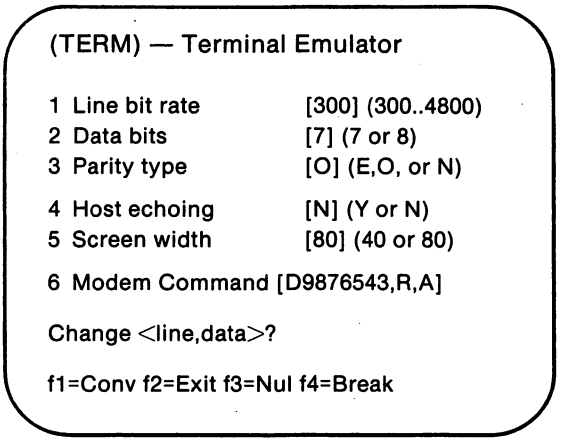


FIGURE 10-6. Terminal Emulator Menu after Pressing Enter

The Terminal Emulator is now prepared to connect the PC*jr* to the host computer. At this point, simply hold down the Fn key and press the 1 key. The modem will automatically dial the number of the host computer and establish a connection. If an error occurs in establishing a connection between the PC*jr* and the host computer, an error message will be displayed on the screen. If the connection has been properly made, the following message will be displayed:

```
CONNECTION...COMPLETE
Ok
```

The PC*jr* is now connected to the host computer. The PC*jr* can now be used as a direct terminal link to the host computer.

If at any time, the connection needs to be broken, simply hold down the Fn key and press the 1 key. This will cause the terminal emulator program to return to the terminal selection menu.

Appendix A.
PC*jr* BASIC Reserved Words

Reserved words are words which have a special meaning in BASIC. They include all BASIC commands, statements, function names, and operator names.

Reserved words are not allowed to be used as variable names in BASIC statements. Also, reserved words must be delimited in BASIC statements so that they can be recognized. Generally, referenced words can be delimited through the use of blank spaces or special characters.

PC*jr* BASIC Reserved Words

ABS	DIM	IOCTL	OR	SPACE$
AND	DRAW	IOCTL$	OUT	SPC(
ASC	EDIT	KEY	PAINT	SQR
ATN	ELSE	KILL	PALETTE	STEP
AUTO	END	LEFT$	PCOPY	STICK
BEEP	ENVIRON	LEN	PEEK	STOP
BLOAD	ENVIRON$	LET	PEN	STR$
BSAVE	EOF	LINE	PLAY	STRIG
CALL	EQV	LIST	PMAP	STRING$
CDBL	ERASE	LLIST	POINT	SWAP
CHAIN	ERDEV	LOAD	POKE	SYSTEM
CHDIR	ERDEV$	LOC	POS	TAB(
CHR$	ERL	LOCATE	PSESET	TAN
CINT	ERR	LOF	PRINT	TERM
CIRCLE	ERROR	LOG	PRINT#	THEN
CLEAR	EXP	LPOS	PSET	TIME$
CLOSE	FIELD	LPRINT	PUT	TIMER
CLS	FILES	LSET	RANDOMIZE	TO
COLOR	FIX	MERGE	READ	TROFF
COM	FN*xxxxxxxx*	MID$	REM	TRON
COMMON	FOR	MKDIR	RENUM	USING
CONT	FRE	MKD$	RESET	USR
COS	GET	MKI$	RESTORE	VAL
CSNG	GOSUB	MKS$	RESUME	VARPTR
CSRLIN	GOTO	MOD	RETURN	VARPTR$
CVD	HEX$	MOTOR	RIGHT$	VIEW
CVI	IF	NAME	RMDIR	WAIT
CVS	IMP	NEW	RND	WEND
DATA	INKEY$	NEXT	RSET	WHILE
DATE$	INP	NOISE	RUN	WIDTH
DEF	INPUT	NOT	SAVE	WINDOW
DEFDBL	INPUT#	OCT$	SCREEN	WRITE
DEFINT	INPUT$	OFF	SGN	WRITE#
DEFSNG	INSTR	ON	SHELL	XOR
DEFSTR	INT	OPEN	SIN	
DELETE	INTER$	OPTION	SOUND	

Appendix B.
DOS Reserved Words

ASSIGN
BACKUP
BREAK
CHDIR
CHKDSK
CLS
COMP
COPY
CTTY
DATE
DIR
DISKCOMP
DISKCOPY
ECHO
ERASE
EXE2BIN
FIND
FOR
FORMAT
GOTO
GRAPHICS
IF
MKDIR
MODE
MORE
PATH
PAUSE
PRINT
PROMPT
RECOVER
REM
RENAME
RMDIR
SET
SORT
SYS
TIME
TREE
TYPE
VER
VERIFY
VOL

Appendix C.
ASCII Character Codes

In the following table, the ASCII codes will be given with any associated characters and control characters (for codes 0-31).

If you wish to display these characters, you can do so by issuing the following statement:

```
PRINT CHR$(x)
```

where *x* is the ASCII code of the character being displayed.

ASCII Value*	Character	Control Character	ASCII Value	Character	Control Character
000	(null)	NUL	016	▶	DLE
001	☺	SOH	017	◀	DC1
002	☻	STX	018	↕	DC2
003	♥	ETX	019	!!	DC3
004	♦	EOT	020	π	DC4
005	♣	ENQ	021	§	NAK
006	♠	ACK	022	▬	SYN
007	(beep)	BEL	023	↨	ETB
008	◘	BS	024	↑	CAN
009	(tab)	HT	025	↓	EM
010	(line feed)	LF	026	→	SUB
011	(home)	VT	027	←	ESC
012	(form feed)	FF	028	(cursor right)	FS
013	(carriage return)	CR	029	(cursor left)	GS
014	♫	SO	030	(cursor up)	RS
015	☼	SI	031	(cursor down)	US

* Decimal

ASCII Value	Character	ASCII Value	Character	ASCII Value	Character
032	(space)	071	G	110	n
033	!	072	H	111	o
034	"	073	I	112	p
035	#	074	J	113	q
036	$	075	K	114	r
037	%	076	L	115	s
038	&	077	M	116	t
039	'	078	N	117	u
040	(	079	O	118	v
041	)	080	P	119	w
042	*	081	Q	120	x
043	+	082	R	121	y
044	,	083	S	122	z
045	-	084	T	123	{
046	.	085	U	124	|
047	/	086	V	125	}
048	0	087	W	126	~
049	1	088	X	127	⌂
050	2	089	Y	128	Ç
051	3	090	Z	129	ü
052	4	091	[	130	é
053	5	092	\	131	â
054	6	093	]	132	ä
055	7	094	^	133	à
056	8	095	_	134	å
057	9	096	`	135	ç
058	:	097	a	136	ê
059	;	098	b	137	ë
060	<	099	c	138	è
061	=	100	d	139	ï
062	>	101	e	140	î
063	?	102	f	141	ì
064	@	103	g	142	Ä
065	A	104	h	143	Å
066	B	105	i	144	É
067	C	106	j	145	Æ
068	D	107	k	146	æ
069	E	108	l	147	ô
070	F	109	m	148	ö

ASCII Value	Character	ASCII Value	Character	ASCII Value	Character
149	ò	188	╝	227	π
150	û	189	╜	228	Σ
151	ù	190	╛	229	σ
152	ÿ	191	┐	230	µ
153	Ö	192	└	231	τ
154	Ü	193	┴	232	Φ
155	¢	194	┬	233	Θ
156	£	195	├	234	Ω
157	¥	196	─	235	δ
158	Pt	197	┼	236	∞
159	ƒ	198	╞	237	∅
160	á	199	╟	238	ϵ
161	í	200	╚	239	∩
162	ó	201	╔	240	≡
163	ú	202	╩	241	±
164	ñ	203	╦	242	≥
165	Ñ	204	╠	243	≤
166	a	205	═	244	⌠
167	o	206	╬	245	⌡
168	¿	207	╧	246	÷
169	⌐	208	╨	247	≈
170	¬	209	╤	248	°
171	½	210	╥	249	∙
172	¼	211	╙	250	·
173	¡	212	╘	251	√
174	«	213	╒	252	n
175	»	214	╓	253	2
176	▒	215	╫	254	■
177	▒	216	╪	255	(blank 'FF')
178	▓	217	┘		
179	│	218	┌		
180	┤	219	■		
181	╡	220	▄		
182	╢	221	▌		
183	╖	222	▐		
184	╕	223	▀		
185	╣	224	α		
186	║	225	β		
187	╗	226	Γ		

Appendix D.
Printer Usage with the PC*jr*

A printer can be a valuable addition to a PC*jr* computer system allowing it to perform a number of useful tasks. For example, a printer enables the PC*jr* to function as a word processor. IBM provides two ways to interface a printer to the PCjr:

- IBM PC*jr* Parallel Printer Attachment
- Serial Port (built-in)

Almost any parallel printer may be connected to the PC*jr* after the Parallel Printer Attachment has been correctly installed. Specifically, IBM markets a medium-priced parallel printer — the IBM 80 CPS Matrix Printer.

As an alternative to the Parallel Printer Attachment, the consumer may elect to use the Serial Port on the back panel of the PC*jr*. This port is indicated with an "S".

The inexpensive IBM PC Compact Printer plugs directly into the Serial Port. Although the Compact Printer attaches quite easily to the Serial Port, the connector used is not the industry standard — RS232 (25 pin-D connector). A serial adapter cable may be purchased to allow any RS232 compatible printer to interface to the PC*jr* through the Serial Port.

LISTING PROGRAMS

The LIST command can output a copy of the program currently stored in the computer's memory. Since the printer is known as "LPT1:", the LIST command requires this device name to cause the output to be sent to the printer.

```
LIST "LPT1:"
```

The LLIST command is functionally equivalent to the previously described LIST command. Note that "LPT1:" is an abbreviation for Line Printer #1.

OUTPUTTING DATA

A PRINT# statement is most commonly used to output data to the printer. However, an I/O channel must be opened for the printer before any data can be output. The following statement is a typical OPEN statement that can be used to establish an I/O channel for the printer:

```
OPEN "LPT1:" FOR OUTPUT AS #3
```

The following example program demonstrates the use of the OPEN and PRINT# statements to output data to the printer:

```
100 OPEN "LPT1:" FOR OUTPUT AS #1
110 FOR J = 1 TO 15
120     PRINT #1,J,J^2
130 NEXT
140 CLOSE #1
150 END
```

If only intermittent printer output is necessary for a specific program, it is advisable to use LPRINT instead of PRINT#. The equivalent of the previous program using LPRINT is given below:

```
100 FOR J = 1 TO 15
110     LPRINT J,J^2
120 NEXT
130 END
```

Notice that no OPEN statement is required with LPRINT. Generally, PRINT# is used in place of LPRINT because PRINT# is faster than LPRINT. The speed difference is difficult to notice in a short program, but becomes apparent in more lengthy applications.

CONTROL CODES

All printers have control codes that select specific printer functions. These codes can be used to change character sets or print styles, cause a line feed or carriage return, and select other functions of the printer.

The IBM PC Compact Printer and the IBM 80 CPS Matrix Printer are by far the most popular with the PC*jr*. IBM has written its printer software to be compatible with these printers. The control codes for these

printers are given in the balance of this appendix. These codes may be sent in BASIC by using the CHR$(*n*) function.

Table D-1. IBM 80 CPS Matrix Printer (Graphics Printer) Control Codes

Control Code	Value	Function
BEL	07	Sounds buzzer for 1 second
BS	08	Backspace
HT	09	Horizontal Tabulation
LF	10	Line Feed
VT	11	Vertical Tabulation
FF	12	Form Feed
CR	13	Carriage Return
SO	14	Turns on enlarged printing mode
SI	15	Turns on condensed printing mode
DC1	17	Selects printer
DC2	18	Turns off condensed mode
DC3	19	Deselects printer
DC4	20	Turns off enlarged mode
ESC 0	27, 48	Line spacing = 8 lines per inch
ESC 2	27, 50	Line spacing = 6 lines per inch
ESC 8	27, 56	Deselects paper end detector
ESC 9	27, 57	Selects paper end detector
ESC A *n*	27, 65, *n*	Line spacing = *n*/72 inches
ESC B *n* NUL	27, 66, *n*, 0	VT = *n*
ESC C *n*	27, 67, *n*	Form length = *n* lines
ESC D *n* NUL	27, 68, *n*, 0	HT = *n*
ESC E	27, 69	Turns on emphasized mode
ESC F	27, 70	Turns off emphasized mode
ESC K	27, 75	Turns on normal density
ESC L	27, 76	Turns on dual density
ESC N	27, 78	Sets skip over perforation
ESC O	27, 79	Release skip over perforation
ESC Q *n*	27, 81, *n*	Column width = *n*
ESC R *n*	27, 82, *n*	Select character set = *n*

Table D-2. IBM PC Compact Printer Control Codes

Control Code	Value	Function
HT	09	Horizontal Tabulation
LF	10	Line Feed
VT	11	Vertical Tabulation
FF	12	Form Feed
CR	13	Carriage Return
SO	14	Set Double Width
SI	14	Set Compressed
DC2	18	Compressed Off
DC4	20	Double Width Off
CAN	24	Clears the printer buffer
ESC — 1	27, 45, 1	Set underline mode
ESC — Ø	27, 45, Ø	Clear underline mode
ESC Ø	27, 48	Line feed = 1/9 inch
ESC 1	27, 48	Identical to ESC Ø
ESC 2	27, 50	Line feed = 1/6 inch
ESC 5 1	27, 53, 1	Line feed after every CR
ESC 5 Ø	27, 53, Ø	No line feed after every CR
ESC <	27, 60	Homehead
ESC B *n* NUL	27, 66, *n*, Ø	Vertical Tab = *n*
ESC C *n*	27, 67, *n*	Lines per page = *n*
ESC D *n* NUL	27, 68, *n*, Ø	Horizontal Tab = *n*
ESC K n, n_2	27, 75, n, n_2	Sets graphics mode. n & n_2 specify the number of data points.
ESC N *n*	27, 78, *n*	Set skip perforation. # of lines = *n*
ESC O	27, 78	Cancel skip perforation
ESC R	27, 82	Clear Tabs
ESC W 1	27, 87, 1	Set double width
ESC W Ø	27, 87, Ø	Cancel double width

Appendix E.
IBM PC*jr* Device Names

Device Name	Interpretation
KYBD:	Keyboard — used in all versions of BASIC for input.
SCRN:	Screen — used in all versions of BASIC for output.
CAS1:	Cassette Tape Unit — used in all versions of BASIC for input and output.
A:	System Disk Drive — used in Cartridge BASIC and Compiler BASIC for input and output.
LPT1:	Printer — used in all versions of BASIC for output.
COM1:	Asynchronous Communications Adapter — used in Cartridge BASIC for serial input and output (if internal modem is not present). Internal Modem — used in Cartridge BASIC for serial input and output using the phone lines.
COM2:	Asynchronous Communications Adapter — used in Cartridge BASIC for serial input and output (if internal modem is present).
CON:	Console — used in Cartridge BASIC for input and output. Console is the combination of the Keyboard and Screen devices.

Appendix F.
IBM PC*jr* BASIC Error Messages

The following give all the BASIC error messages along with the related error message number and a description of the error.

1 **NEXT without FOR** The variable that follows NEXT does not match with any preceding FOR statement.

2 **Syntax error** The program line contains errors in punctuation or spelling (ex. misspelled reserved word, deleted parentheses, etc.).

3 **RETURN without GOSUB** A RETURN statement is found which does not have a corresponding GOSUB.

4 **Out of data** A READ statement is encountered where no more items are available to be read from DATA statements.

5 **Illegal function call** A parameter is sent to a system function that is out of range. Examples of the cause of this error include the following:

- A negative subscript
- A subscript that is too large
- A call to a USR function when the starting address for that function had not been given
- A negative record number used with GET and PUT
- Trying to list or edit a BASIC program that is protected
- An argument for a function or statement that is not legal

6 **Overflow** A number is larger than that allowed by BASIC. If the overflow occurs with an integer, program execution will stop. With non-integers, machine infinity will be returned with the proper sign.

If a number is smaller than that allowed by BASIC, an underflow condition will result. A zero will be returned and execution will continue.

7 **Out of memory** This error occurs when the program is too large for available memory, or if a program contains too many FOR loops or GOSUB's.

8 **Undefined line number** A reference is made in the program for a line number that does not exist.

9 **Subscript out of range** An array variable contains a subscript that is outside of the range that was given in the DIM statement.

10 **Duplicate definition** The same array was defined twice. The following may cause this error:

- Two DIM statements were included for the same array.
- A DIM statement is used for an array after a default dimension of 10 had been established previously.
- An OPTION BASE statement which sets an unacceptable array size was encountered after an array had already been dimensioned by a DIM statement or by default.

11 **Division by zero** Either division by zero was attempted, or an attempt was made to raise to a negative power.

12 **Illegal direct** An attempt was made to enter a statement in the direct mode that can only be entered in indirect.

13 **Type mismatch** The data used for a variable does not match that variable's type (ex. numeric data for a string variable).

14 **Out of string space** BASIC assigns available free memory to string variables until that memory has been depleted. When this occurs, this message will be displayed.

15 **String too long** The user attempted to create a string in excess of 255 characters.

16 **String formula too complex** The string expression is too long or too complex. Try breaking the expression into smaller, less complex expressions.

17 **Can't continue** CONT was used in one of the following situations:

- CONT was used to attempt to start a program that had stopped because of an error.
- CONT was used to attempt to start a program that had been changed during a temporary halt in execution.
- CONT was used to attempt to start a program that does not exist.

18 **Undefined user function** A function was called before it was defined with DEF FN.

19 **No RESUME** The program branched to an error trapping routine without a RESUME statement.

20 **RESUME without error** A RESUME statement was encountered before an error trapping routine was executed.

22 **Missing operand** No operand following an operator such as +, *, AND.

23 **Line buffer overflow** The user tried to enter a line containing too many characters.

24 **Device Timeout** Information was not received from an input or output device within an allotted length of time.

25 **Device Fault** A hardware error flag was returned by the interface adapter.

26 **FOR without NEXT** A FOR statement was encountered without a corresponding NEXT.

27 **Out of Paper** Either the printer has run out of paper or it is not turned on.

29 **WHILE without WEND** A WHILE statement was encountered without a corresponding WEND.

30 **WEND without WHILE** A WEND statement was encountered without a corresponding WHILE statement.

50 **FIELD overflow** The program contains a FIELD statement in which more bytes are allocated for a random file's record length than were specified for that file in its OPEN statement. Another possibility is a situation where the end of the FIELD buffer was reached while a sequential input/output was being performed to a random file (ex. PRINT#, WRITE#, INPUT#, etc.).

51 **Internal Error** Some problem is present internally in BASIC. Call IBM or your IBM dealer with a description of the circumstances under which the error occurred.

52 **Bad file number** A file number is referenced that is not currently assigned to an open file, or is not within the range of valid file numbers specified during initialization. Another possibility is when an invalid device name is used in a file specification or when the filename is invalid.

53 **File not found** A file is referenced in a LOAD, KILL, FILES, NAME, or OPEN that does not exist on the diskette on the specified drive.

54 **Bad file mode** This error occurs when the PUT or GET statement was used with one of the following:

- Sequential file
- Closed file
- To MERGE a non-ASCII file
- To execute an OPEN with a file mode other than input, output, append, or random

55 **File already open** An attempt was made to open a file that had been previously opened for sequential output or an append. This error also occurs when an attempt is made to kill an open file.

57 **Device I/O Error** An error occurred during a device I/O operation. Error recovery is not possible in DOS.

58 **File already exists** The filename with a NAME statement duplicates that of a filename already being used on that diskette.

61 **Disk full** The entire diskette space is being used. When this error occurs, all files will be closed.

62 **Input past end** This error statement indicates that an end of file error occurred. This is caused by an INPUT# statement being executed for a sequential file whose entire data has already been read or for a null file. By using the EOF function, this error can be avoided. Another cause of this error is an attempt to read from a file that had been opened for an append or for output.

63 **Bad record number** The record number used in a GET or PUT statement was either zero or was greater than the allowed maximum (32767); BASIC 2.0 (16,777,215); Cartridge BASIC (16,777,215).

64 **Bad filename** An invalid form is used for a filename.

66 **Direct statement in file** Any ASCII files loaded by LOAD or CHAIN should only contain statements with line numbers. If a direct statement is encountered in a program file being LOAD'ed or CHAIN'ed, the LOAD or CHAIN will be terminated. A common cause of this error is the inclusion of a line feed character.

67 **Too many files** An incorrect file specification was used, or an attempt was made to create a new file when the directory was full.

68 **Device unavailable** An attempt was made to open a file to a non-existent device. The device may have been disabled or the hardware may not be present.

69 **Communications buffer overflow** A communications input statement was executed with the input buffer already full. When this error condition occurs, use the ON ERROR statement to attempt input again. When ensuing inputs are attempted, the error condition will be cleared unless characters are received at a rate faster than the program can process them. In this case, try one of the following:

- Use /C: option when starting BASIC to increase the size of the communications buffer.
- Use a hand-shaking routine with the other computer to send a message to tell it to stop sending data so that the receiving computer can empty its buffer.
- Use a lower baud rate for data transmission and reception.

70 **Disk write protect** The user attempted to write to a diskette that was write-protected.

71 **Disk not ready** Either a diskette is not in place or the diskette door is open.

72 **Disk media error** Generally, this is due to a bad diskette, although the cause can be a hardware related problem. The user should copy any existing data to a new diskette and reformat the bad diskette. If the formatting fails, the diskette is unusable and should be discarded.

73 **Advanced feature** This error occurs when a program attempts to use an Advanced BASIC feature when Disk BASIC is being used. Load Advanced BASIC and run the program under it.

74 **Rename across disks** During an attempt to rename a file, the wrong disk identifier was used.

75 **Path/File Access Error** The user used an illegal path or filename during an OPEN, NAME, MKDIR, CHDIR, or RMDIR statement.

76 **Path not found** The specified path cannot be located.

- **Incorrect DOS version** The command specified requires a different version of DOS.

— **Unprintable error** The error condition does not have a corresponding error message. This message is generally the result of using an ERROR statement with an undefined error code.

- **Cartridge required** An attempt was made to access BASIC from a diskette without the BASIC Cartridge present.

Index

A

B

C

D

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Special Characters

ABOUT THE WEBER SYSTEMS, INC. STAFF

In 1982, Weber Systems, Inc. began a start-up publishing division specializing in books related to the personal computer field. They initially published three books, and within a year, expanded their list to eighteen machine-specific titles, with fourteen more scheduled for early 1984.

All Weber Systems USER'S HANDBOOKS are created by an in-house editorial staff with extensive backgrounds in computer science and technical writing. The three basic tenets of their publishing philosophy are: quality, timeliness and maintenance (frequent updating).

Weber Systems is located in Cleveland, Ohio.

Other Books in This Series
Published by Ballantine Books

IBM BASIC® USER'S HANDBOOK

IBM PC® & XT® USER'S HANDBOOK

KAYPRO® USER'S HANDBOOK

VIC-20® USER'S HANDBOOK

COMMODORE 64® USER'S HANDBOOK

APPLE IIe® USER'S HANDBOOK

IBM PCjr® USER'S HANDBOOK

The IBM PCjr USER'S HANDBOOK is a clear, concise, and practical guide to the capabilities and operations of IBM's inexpensive new home computer—destined to be the most widely used home computer yet. This handbook can be used by families, schools, and at home in conjunction with the IBM PC & XT.

This complete description of the PCjr includes precise information on set-up, operation, programming, and maintenance.

The following topics are covered in detail:

- PCjr Installation
- PCjr Operation
- Available PCjr Software
- Keyboard Usage
- PCjr Peripherals
- PCjr BASIC Programming
- PCjr Communications

No user or potential user of the IBM PCjr should be without the IBM PCjr USER'S HANDBOOK.

ISBN 0-345-31597-9

Cover printed in USA