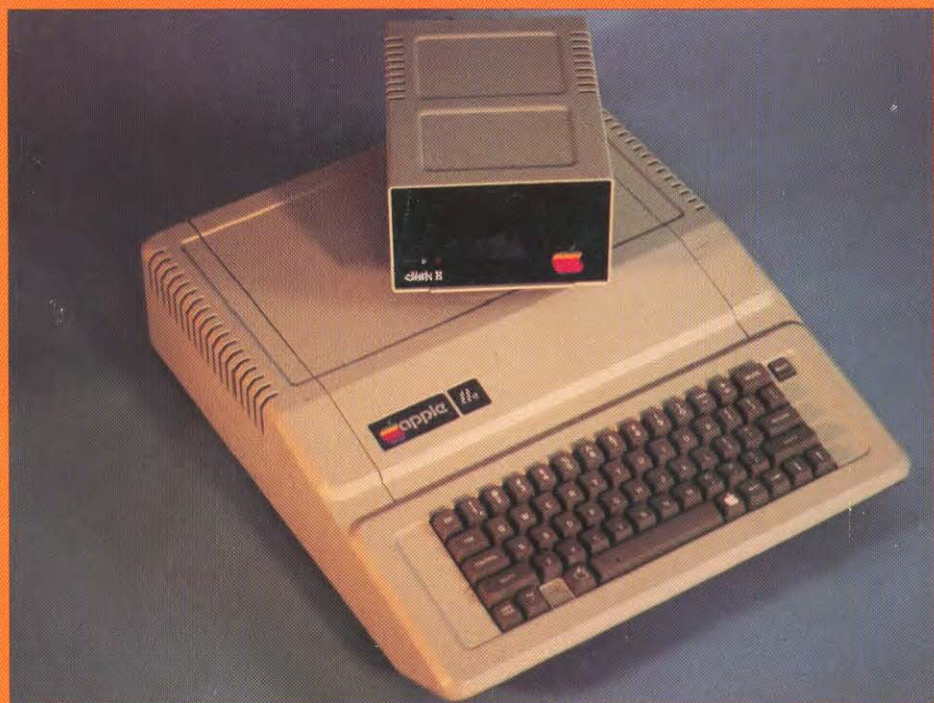


# APPLE IIe<sup>®</sup>

## 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



**Apple IIe®  
User's Handbook**



**Apple IIe®  
USER'S HANDBOOK**

**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

## **Apple II® User's Handbook**

Copyright © 1983 by Weber Systems, Inc.

All rights reserved under International and Pan-American Copyright Conventions. Published in the United States by Ballantine Books, a division of Random House, Inc., New York, and simultaneously in Canada by Random House of Canada Limited, Toronto. This is a fully revised edition of **User's Handbook to the Apple II Personal Computer®**, originally published by Weber Systems, Inc.

Apple IIe is a trademark of Apple Computer Corporation. This book has been neither authorized nor endorsed by Apple Computer Corporation.

Apple IIe®, Apple DOS®, Applesoft BASIC®, and Integer BASIC® are trademarks of Apple Computer Corporation.

Library of Congress Catalog Card Number: 83-91223  
ISBN 345-31594-4

Manufactured in the United States of America

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

# CONTENTS

- 1. INTRODUCTION TO THE APPLE IIe** 11
- History of the Apple II 11. History of the Apple IIe 11. Typical Apple IIe System 12. Inside the Apple IIe 13. Main Board 13. Expansion Slots 14. Auxiliary Slot 14. Rear Panel 15. Bits & Bytes 16. ROM & RAM 16. Dynamic & Static RAM 17. Apple IIe Power Supply 17. Apple IIe Speaker 17. Apple IIe Video Display 18. Monitor/TV Set Connection 18. Apple IIe Video Display Formats 20. Cassette Recorder 22. Apple IIe Disk Drive 23. Hand Controls 24. Apple IIe Controller Cards 24. Communications 25. Software 25. Operating Systems 26. Language Translators 26. Applesoft BASIC 27. Integer BASIC 27. Other Languages 27. Applications Programs 27.
- 2. APPLE IIe INSTALLATION, TROUBLESHOOTING, AND OPERATION** 29
- Introduction 29. Installation 29. Troubleshooting 29. Built-in Self Tests 31. Apple IIe Operation 32. Apple IIe Keyboard 33. Space Bar 35. Shift 35. Caps Lock 35. Cursor Control Keys 35. ESC Key 36. Open Apple Key 36. Solid Apple Key 36. Reset Key 37.
- 3. APPLE BASIC PROGRAMMING** 39
- Introduction 39. Switching from Applesoft to Integer 39. Compiled & Interpreted Languages 40. Immediate & Program Modes 40. Line Numbers 41. NEW Command 42. END Statment 42. Executing a Program 43. Program Lines & Display Lines 43. Multiple Statement Program Lines 43. Listing a Program 44. Error Message 45. BASIC Program Editing 45. Applesoft BASIC Data Types 46. Strings 46. Numeric Data 47. Floating Decimal Point 47. Floating Point Numbers 48. Integer 48. Scientific Notation 49. Variables 50. BASIC Variables 50. BASIC Variable Names 51. Tables & Arrays 52. Expressions and Operators 54. Compound Expressions and Order of Evaluation 55. Arithmetic Operations 56. Logical Operators 59. Applesoft BASIC Statements 61. Remark Statements 62. Assignments 63. Outputting Data 65. INPUT Statements 67. GET 69. FOR, NEXT Loops 69. Nested Loops 71. Conditional Statements 71. Branching Statements 72. ON, GOTO

Statement 73. Subroutines & GOSUB Statements 73. ON, GOSUB Statements 75. Applesoft BASIC Functions 75. String Concatenation 76. ASCII 77. CHR\$ & ASC Functions 77. PEEK & POKE 78. Stopping Program Execution 79. Control-C 79. END 79. STOP 80. RESET 80.

#### **4. APPLE BASIC REFERENCE GUIDE**

**81**

Introduction 81. ABS 82. AND 82. ASC 84. ATN 84. AUTO 85. CALL 86. CHR\$ 87. CLEAR 88. CLR 88. COLOR 89. CON 90. CONT 91. COS 92. DATA 92. DEF FN 93. DEL 94. DIM 95 DRAW 97. DSP 98. END 99. EXP 100. FLASH 100. FOR...NEXT 101. FRE 103. GET 104. GOSUB, RETURN 105. GOTO 106. GR 107. HCOLOR 107. HGR 108. HGR2 109. HIMEM 110. HLIN 111. HOME 112. HLOT 112. HTAB 113. IF...THEN 114. IN# 115. INT 116. INVERSE 116. INPUT 117. LEFT\$ 118. LEN 119. LET 120. LIST 120. LOAD 121. LOG 122. LOMEM 123. MAN 124. MID\$ 124. NEW 125. NORMAL 126. NOT 126. NO TRACE 127. ON 128. ONERR GOTO 129. OR 131. PDL 133. PEEK 133. PLOT 134. POKE 135. POP 136. POS 138. PRINT 139. READ 140. RECALL 141. REM 144. RESTORE 144. RESUME 145. RETURN 146. RIGHT\$ 146. RND 147. ROT 149. RUN 150. SAVE 150. SCALE 151. SCRIN 152. SGN 154. SHLOAD 155. SIN 155. SPC 156. SPEED 156. SQR 157. STOP 157. STORE 158. STR\$ 158. TAB 159. TAN 161. TEXT 161. TRACE 161. USR 162. VAL 163. VLIN 163. VTAB 164. WAIT 165. XDRAW 166.

#### **5. CASSETTE & DISK STORAGE WITH THE APPLE IIe**

**169**

Introduction 169. Cassette Installation & Operation 169. Saving and Loading a Program on Cassette 170. Storing and Loading Data on Cassette 171. Apple IIe Disk Storage 171. Types of Disks 171. Hard Disks 171. Floppy Diskettes 173. Tracks and Sectors 174. Hard and Soft Sectors 176. Single and Double Sided Diskettes 178. Single, Double, and Quad Density Diskettes 178. Diskette Write Protection 178. Diskette Handling Rules 179. Inserting and Removing a Diskette 180. Disk Operating System 181. Disk II System 181. Installing the Disk II System 181. Booting DOS 185. Auto Start Boot 185. Booting from Integer or Applesoft BASIC 186. Booting from the Monitor 186. Restoring DOS 187. Using 13-Sector Diskettes with the IIe 187. Prompts 180. Error Message Format DOS Commands 189. Filenames 190. Drive Specification 190. Slot Specification 191. Volume Specification 192. CATALOG 194. Track/Sector List 196. INIT 197. Master and Slave Diskettes 199. LOAD 202. SAVE 202. DELETE 203. RENAME 203. LOCK 204. VERIFY 204. MON and NOMON 205. MAXFILES 207. EXEC 208. Creating an EXEC File 209. BSAVE 210. BLOAD 210. BRUN 211. Sequential and Random File Access 211. Opening Sequential Files 214. Writing to Sequential Files 214. Reading Sequential Files 216. Closing a Sequential File APPEND 218. POSITION 219. Storing Data in Disk Files 221. Opening and Closing a Random Access File 223. Reading and Writing to Random Files 223. Byte Parameter 224.

#### **6. APPLE IIe GRAPHICS**

**225**

Low Resolution Graphics 225. Commands 225. Uses of Low Resolution Graphics 228. Charts 228. Writing a Game Program 229. High Resolution Graphics 232. Commands 232. Shape Table 235. Shape



Table Directory 240. Saving a Shape Table 243. Using the Shape Table 244. SCALE 244. ROT 244. DRAW 245. XDRAW 245. Programming with Shape Tables 246.

## **7. THE SYSTEM MONITOR**

**249**

Introduction 249. Activating and De-activating the Monitor 249. Commanding the Monitor 251. Memory Examine 251. Memory Dump 252. Register Examine 253. Changing Memory 253. Changing Registers 255. Move Data 255. Comparing Blocks of Memory 256. Saving and Retrieving Data with the Cassette 259. Saving and Retrieving Data from Disk 260. Other Input/Output Commands 261. Machine Language Programming 263. Mini-Assembler 263. Activating the Mini-Assembler 264. Entering the First Program Line 264. Entering Subsequent Program Lines 265. Returning to the Monitor 265. Converting Assembly Language Hex Codes 265. Executing a Machine Language Program 267. Creating a Custom Monitor Command 268. Look Up Section 269.

## **8. THE 80-COLUMN BOARD**

**273**

Activating the 80-Column Board in BASIC 274. Deactivating the 80-Column Board 275. Selecting 40 or 80 Columns While the Board is Active 276. Moving the Cursor 277. Editing Functions that Clear Parts of the Display 278. Scrolling the Display 279. Use of Control Codes 279. BASIC Support of the 80-Column Board 281. Tabbing 281. Use of INVERSE, FLASH, and HOME 282. Uppercase — Restrict Mode 282.

<b>Appendix A. Applesoft BASIC Reserved Words &amp; Tokens</b>	<b>285</b>
<b>Appendix B. Integer BASIC Reserved Words</b>	<b>286</b>
<b>Appendix C. DOS Reserved Words</b>	<b>286</b>
<b>Appendix D. Applesoft BASIC Error Messages</b>	<b>287</b>
<b>Appendix E. Integer BASIC Error Messages</b>	<b>290</b>
<b>Appendix F. DOS Error Messages</b>	<b>292</b>
<b>Appendix G. ASCII Character Set</b>	<b>295</b>
<b>Appendix H. Apple IIe Printer Usage</b>	<b>299</b>
<b>Appendix I. Monitor Subroutines</b>	<b>301</b>
<b>Appendix J. System Monitor Files</b>	<b>308</b>
<b>Index</b>	<b>311</b>

# INTRODUCTION

The *Apple IIe User's Handbook* is meant to serve as a tutorial as well as an ongoing reference guide to the Apple IIe personal computer. The latest features of the Apple IIe are discussed in detail including the 80 column card and the new IIe keyboard.

Chapter 1 of this book is intended to serve as an introduction to the IIe. The system board, expansion slots, speaker, video display, cassette recorder, disk drive, and BASIC interpreter are all discussed in this chapter. Terms basic to computing such as RAM, ROM, byte, bit, software, modem, operating system, interpreter, compiler, and assembler are all defined in Chapter 1.

Chapter 2 describes Apple IIe installation, operation, and troubleshooting. Keyboard usage is discussed in detail in this chapter.

Chapter 3 is meant to serve as tutorial on Applesoft and Integer BASIC programming on the IIe. Topics such as BASIC start-up, switching from Applesoft to Integer, constants, variables, strings, arrays, operators, loops, functions, conditional statements, and branching statements are discussed in detail. Chapter 3 assumes the user has some familiarity with BASIC programming and is not meant to serve as an introductory guide to programming for the first-time user.

Chapter 4 serves as a detailed reference guide to each of the various commands and functions available in both Integer and Applesoft BASIC.

Chapter 5 includes a detailed discussion of data storage on cassette or diskette with the Apple IIe. Each of the DOS 3.3 commands are discussed in detail. Files and file handling are also discussed.

Chapter 6 includes a detailed discussion of programming graphics on the Apple IIe.

Chapter 7 describes the usage of the IIe's system monitor and mini-assembler.

Chapter 8 discusses the usage of the IIe's optional 80-column board.

The *Apple IIe User's Handbook* includes nine useful appendices. These detail the various Applesoft, Integer, and DOS reserved words and error messages, the ASCII code set, printer usage with the IIe, and the various monitor subroutines.

# CHAPTER 1. INTRODUCTION TO THE APPLE IIe.

---

## HISTORY OF THE APPLE II

When the Apple II computer was first introduced in the summer of 1977, it was one of the first fully assembled microcomputers available. The Apple II was designed by Steven Jobs and Steven Wosniak in a garage in Los Altos, California.

From this humble beginning, the Apple II has evolved into a complete line of microcomputers, peripherals, and software. Apple computers can be found in homes, offices, small businesses, and factories throughout the world. Apple Computer, Inc. has grown into a multi-million dollar, multi-national firm in just a few short years.

## HISTORY OF THE APPLE IIe

The Apple IIe was introduced in the spring of 1983. The IIe is a redesigned version of the Apple II. The IIe contains 64K of RAM, an expanded keyboard with 63 keys that can output both upper and lower case characters, an optional 80-column display and a 6502B CPU (a high-speed version of the original 6502).

The Apple IIe includes both Applesoft and Integer BASIC. Applesoft BASIC is included in the ROM, and Integer is loaded from the System Master diskette.

The IIe uses the same Disk II drive used with the earlier Apple II models. DOS version 3.3 is used with the IIe.

---

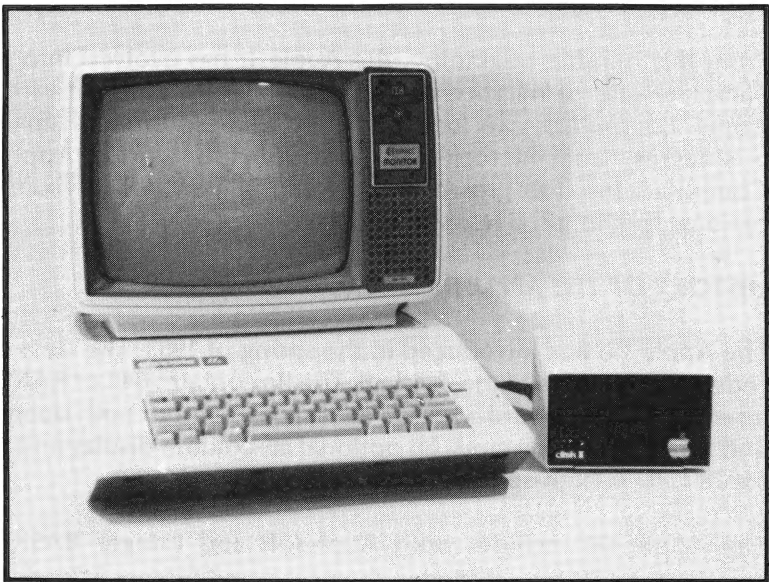
\*Although this book is dedicated to the Apple IIe, the majority of the concepts apply to the Apple II as well.

## TYPICAL APPLE IIe SYSTEM

A typical Apple IIe system is depicted in Illustration 1-1. Your Apple IIe system may not include the exact components pictured in Illustration 1-1. However, every Apple IIe system must include at least two of the components shown in Illustration 1-1--the Apple IIe computer and a monitor or television set.

We will discuss the Apple IIe first, followed by the various peripherals that can be connected to this basic Apple IIe system. These peripherals include a monitor or TV set, cassette recorder, Apple IIe Disk Drive, printers, and the various Apple controllers or cards.

**Illustration 1-1. Typical Apple IIe System**



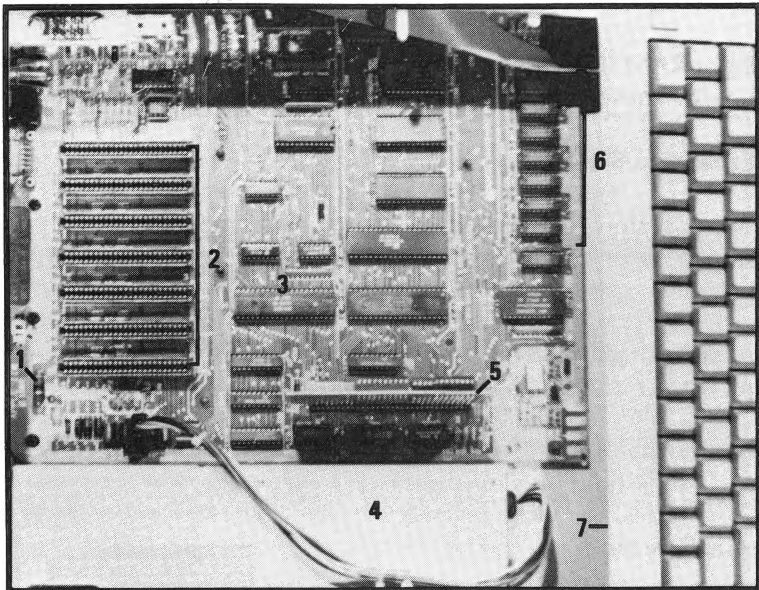
## Inside the Apple IIe

The Apple IIe consists of the 6502B microprocessor, RAM memory, ROM memory, slots for the connection of controller cards, a power supply, a speaker, game I/O connectors, a video connector, a cassette interface, and a keyboard all enclosed in a protective case. The inside of the Apple IIe is shown in Illustration 1-2.

### Main Board

Once you have opened your Apple IIe you will notice a large green printed circuit board at the bottom of the open compartment. This board is known as the main board. The main board contains the various IC chips and components that control the Apple IIe.

**Illustration 1-2. Inside of the Apple IIe**



1. Internal Power-On Light
2. Expansion Slots
3. 6502B Microprocessor
4. Power Supply
5. Auxiliary Slot with 80 Column Board
6. Memory
7. Speaker

Perhaps the most important component on the main board is the 6502B microprocessor. As shown in Illustration 1-2, the 6502B microprocessor is located in the center of the main board just below the seven slots. The 6502B is a high speed version of its predecessor, the 6502. The 6502 was used in the Apple II and II Plus. The 6502B can address 64K of RAM. However, the IIe has incorporated special memory banking and switching techniques to allow more than 64K to be addressed.

Several other important components of the main board are depicted in Illustration 1-2. These include an AY-3600 integrated circuit and a ROM chip that are used to encode the keyboard characters. The main board also includes two ROM chips that contain the Applesoft BASIC interpreter.

The main board contains an MMU integrated circuit which controls memory addresses within the IIe, and an IOU integrated circuit which controls the built-in input/output features of the IIe.

Eight RAM IC chips are located at the bottom right of the main board. These supply the IIe with 64K of RAM.

### **Expansion Slots**

Towards the rear of the main board are located seven expansion slots. These slots allow additional hardware devices to be installed with the Apple IIe. For example, an expansion card is available that enables the IIe to run the CP/M operating system. Also, controller cards must be installed in these slots to enable a printer or disk drive to be used with the IIe.

Notice that these seven slots are numbered from 1 through 7. In the Apple II, eight slots (numbered 0 through 7) were available.

### **Auxiliary Slot**

Notice the large expansion slot located on the left side of the main board. This is known as the auxiliary slot. If your IIe contains the 80-column text option, the 80 column text card will be installed in this slot.

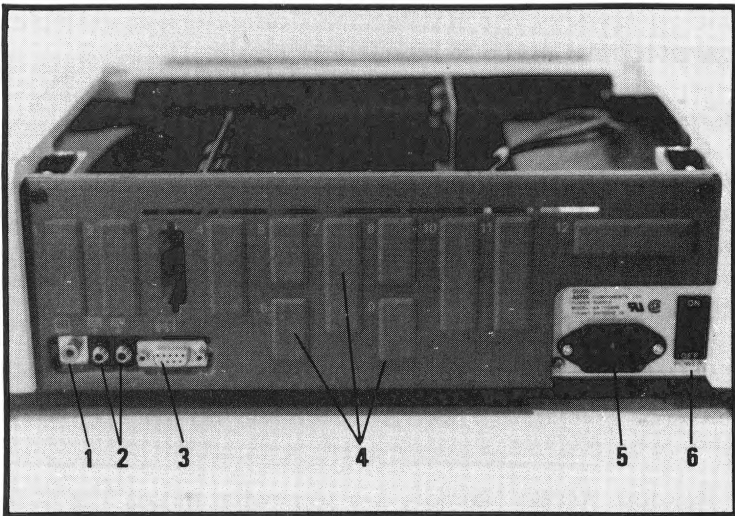
## Rear Panel

The bottom left-side of the rear panel of the Apple IIe (see Illustration 1-3) contains a 9-pin D-connector for the installation of hand controls, two phone jacks for the installation of a cassette recorder, and an RCA type jack for the installation of a video monitor.

The bottom right-hand side of the IIe's rear panel contains the AC cord socket and an on/off switch. Both of these are connected to the power supply.

Finally, note the rectangular openings along the IIe's rear panel. These openings are numbered from 1-12. When the IIe is connected to one or more peripheral devices using an interface card, that card's connector is attached to the proper opening. Openings 1 through 4 are used for 19-pin connections; 5, 6, 8 and 9 are used for 9-pin connections; and 7, 10, 11, and 12 are used for 25-pin connections.

**Illustration 1-3. Apple IIe Rear Panel**



1. Video Monitor Jack
2. Cassette Recorder Input/Output Jacks
3. 9-pin D-Connector
4. Rectangular Openings
5. AC Cord Socket
6. On/Off Switch.



## Bits & Bytes

Microprocessor logic is based upon the **bit**. A bit is the basis of all information storage within the computer. A bit consists of 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 of the 8-bit microprocessors can only address (or work directly with) 65,535 (64K) bytes at any one time. Even though this number appears large, a 30 page document would fill this memory area. The Apple IIe uses an 8 bit microprocessor. However, due to the usage of special memory banking and switching techniques, the IIe can address over 64K of memory.

Most 16-bit microprocessors can address from 65535 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. The IBM Personal Computer is an example of a computer that uses a 16 bit microprocessor.

## ROM and RAM

ROM stands for Read-Only Memory. ROM will hold the data stored in it permanently. If the power to the Apple is shut off, the information stored in ROM will remain there. As previously mentioned, the Apple's BASIC language interpreter is stored in ROM.

RAM stands for Random Access Memory\*. Any data stored in RAM is lost when the Apple's power is shut off. When data is

---

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

loaded from a tape cassette, a disk drive, or the keyboard, it is stored in RAM. For example, when a program is loaded from the Apple Disk II, it will be stored in RAM.

### **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.

### **Apple IIe Power Supply**

The Apple IIe's power supply is located on the left side of the inside of the unit as shown in Illustration 1-2. The power supply will supply four voltages: +5v, -5v, +12.0v, and -12v.

The Apple's main power cord plugs into the power supply on the back of the Apple. The Apple's power-on switch is also located on the back of the power supply. This is pictured in Illustration 1-3.

Some Apple's have a power supply with a switch which allows the user to select either 110 or 220 volts.

### **Apple IIe Speaker**

As pictured in Illustration 1-2, the Apple IIe's speaker is located inside the case on the lower left hand side. The speaker is connected to the Apple so that a program can be used to create sounds on it.

The Apple IIe's speaker is controlled by a **soft switch**. Soft switches have two states (ex. in/out; on/off; text/graphics). By addressing a special memory location associated with the soft switch, a program can change the state of the switch.

It is unimportant what data values are actually read from or written into the memory address associated with a soft switch. It is the reference to that address that throws the switch. The data written to or read from the location has no effect.

Machine language programs should reference the hexadecimal value for the memory address associated with the soft switch. BASIC programs should use a read operation to the decimal value for the memory address associated with the soft switch. A write operation causes such a short pulse that the speaker will not emit a sound.

The memory address for the soft switch associated with the speaker is 49200 or hexadecimal C030H. Whenever this address is referenced in a program, the speaker will emit a small click. By continually referencing this address, the speaker will generate a continuous tone.

### **APPLE IIe VIDEO DISPLAY Monitor/TV Set Connection**

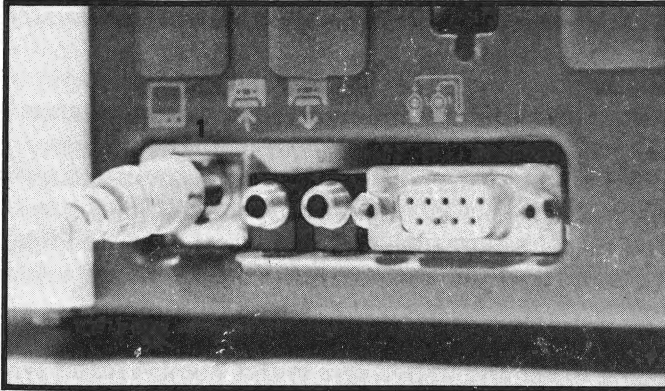
The Apple IIe can either use a monitor or a regular TV set for video output. The connection between the IIe and a monitor is simple. Merely use a video cable to connect the video port on the rear of the IIe to the video input port on the monitor. This connection is depicted in Illustration 1-4.

The connection between the IIe and a TV set is somewhat more complicated due to the fact that the IIe's video signal must be converted to a signal that can be comprehended by the TV set. A device known as an RF modulator must be installed in the IIe in order to convert the IIe's video signal. The RF modulator is generally connected to a group of four Molex-type pins on the rear right-hand side of the main board.

A switch box is also generally connected to the VHF terminals of the TV set's antenna box. This switch box enables the TV set to be operated normally in one setting; while in the second setting, it serves as the IIe's video output device. The installation of the TV switch box is depicted in Illustration 1-5.

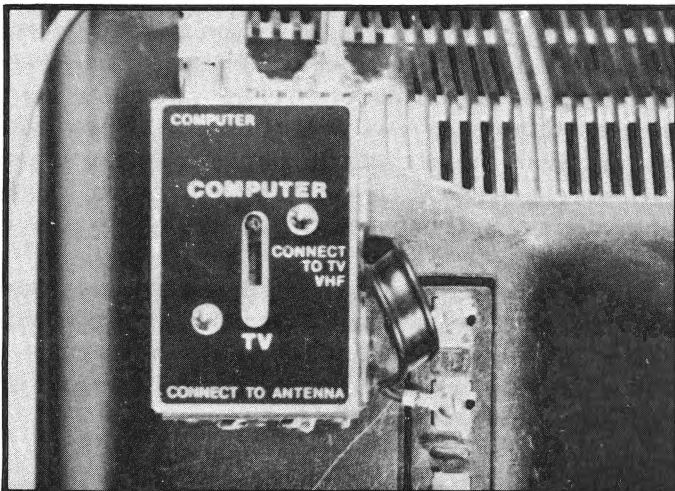
Modulators are also available which can be connected directly to the VHF terminals in the TV set's antenna. If this type of modulator is directly connected to the TV set, a cable can be run from the video out port on the rear of the IIe to the modulator to make the proper connection.

**Illustration 1-4. Apple IIe/Monitor Connection**



1. Video Cable Connector

**Illustration 1-5. Apple IIe/TV Set Connection**



## Apple IIe Video Display Formats

The IIe can display the following four types of video output:

- 40-column text mode.

- 80-column text mode (with optional 80-column card).

- Low-resolution graphics (40 x 48 with 16 colors).

- High-resolution graphics (280 x 192 with 6 colors).

In the text mode, the IIe has the capability to display 24 lines with 40 or 80 columns per line. The characters in the 80 column mode are only half as wide as those in the 40 column mode. For this reason, the 80 column text mode generally cannot be used with a regular color or B&W television set as the resultant output is so blurred that it is unreadable. A high-resolution video monitor must be used to obtain clear output in the 80-column text mode.

In the text mode, any of the 96 ASCII characters can be displayed including lower and upper-case letters numbers, and symbols. The text characters are generally displayed as white dots on a black background. However, text characters can also be displayed as black characters on a white background. This type of display is known as the **inverse format**.

Each individual character is displayed on a matrix seven dots wide by eight dots high. The character itself is only five dots wide. This leaves one blank dot on either side of the character. Therefore, a total of two rows of blank dots are allowed between characters. With the exception of lowercase characters with descenders characters are created seven dots high. This leaves one blank line of dots between characters.

Two different character sets are available in the text mode: the primary character set and the alternative character set. The characters themselves are identical in each character set. However, the format in which the characters are displayed differs. The following formats are available:

- normal* - white dots on black background.

- inverse* - black dots on a white background.

- flashing* - alternating between normal and inverse.

In the primary character set, the IIe can display uppercase characters in either the normal, inverse or flashing formats. Lowercase letters can only be displayed in the normal format.

In the alternative character set, the flashing format for uppercase and lowercase letters is unavailable. However, the normal and inverse formats for both upper and lowercase letters is available. The alternative character set is used when the 80-column card is active.

In low resolution graphics, the IIe can display 1920 blocks of data within an array that measures 48 blocks high by 40 blocks wide. Each block can be assigned any one of the following 16 different colors available with low resolution graphics.

Black	Brown
Magenta	Orange
Dark Blue	Grey 2
Purple	Pink
Dark Green	Light Green
Grey 1	Yellow
Medium Blue	Aquamarine
Light Blue	White

No empty space exists between blocks. Therefore, if a group of several adjacent blocks are assigned the same color, they will appear as a single mass.

High resolution graphics consists of 53,760 dots in 280 dot wide by 192 dot high array. The dots used in high resolution graphics are the same size as the dots used to make up characters in the text mode.

The following six colors are available in the high resolution graphics mode:

black	blue
white	green
orange	purple

Every dot in high resolution graphics can either be black, white, or one of the colors. However, every color is not available for every dot in high resolution graphics.

In either low or high resolution graphics, the user can include 4 lines of text at the bottom of the display. In low resolution graphics, these 4 lines of text replace the final 8 rows of blocks. In high resolution graphics, they replace the lower 32 rows of dots. These display modes which contain both text and graphics characters are known as **mixed modes**.

### **Cassette Recorder**

Your Apple IIe can be connected to a cassette recorder via the cassette interface. The cassette recorder can be used to store programs or data transferred from RAM. These programs or data later can be transferred back into RAM.

The cassette interface jacks are located on the rear of the IIe. The cassette interface jacks can be used to connect the Apple to a standard cassette tape recorder. The tape recorder can be used as a data storage device for the Apple.

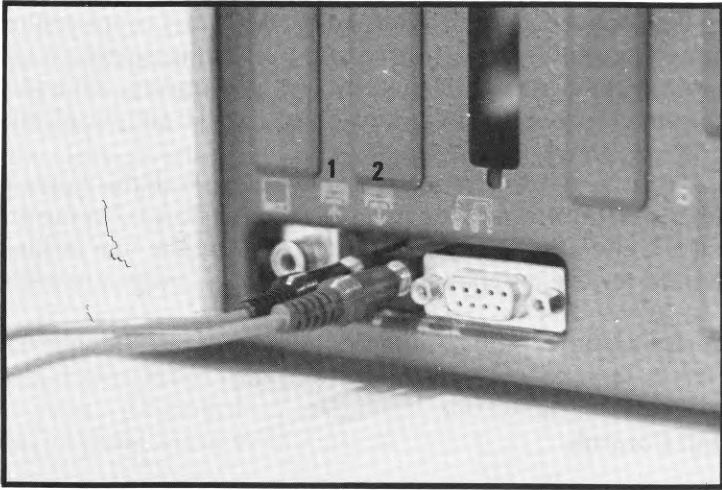
The two cassette interface jacks are labeled with illustrations. The output jack is labeled with a picture of an arrow pointing towards a cassette. The input jack is labeled with a picture of an arrow coming from a cassette.

The cassette input jack should be connected to the cassette recorder's earphone or monitor output jacks. The input jack will listen to the tones on the cassette tape, translate those tones into data, and then store them in memory.

The OUT or output jack should be connected to the cassette recorder's microphone input jack. The output jack is connected to a soft switch just as the Apple speaker is. The cassette's output jack soft switch memory address is 49184 (-16352 Integer BASIC). By referencing this address, the voltage at the output jack will be varied, causing a tone to be produced on the tape. By altering the pitch and duration of this tone, data can be recorded on the cassette tape.

An Apple IIe/cassette recorder hook up is pictured in Illustration 1-6. The cassette recorder will be covered in more detail in Chapter 5.

### Illustration 1-6. Apple IIe Hookup to Cassette Recorder



1. Output Jack 2. Input Jack

### Apple II Disk Drive

A floppy disk system is a much more efficient means of data storage than is a cassette recorder system. The name of the floppy disk system used by the Apple IIe is the Disk II. The Disk II system includes a disk drive, disk controller card, and a cable used to connect the disk controller card to the disk drive. (see Illustrations 5-8 and 5-9).

The Disk II's controller card is installed in one of the expansion slots in the IIe's main board. The Disk II's operation and programming will be discussed in detail in Chapter 5.

### Printers

A printer can be connected to the Apple IIe either via a serial interface card or a parallel interface card. The type of card used depends on whether the printer being connected is a serial or parallel device.



Data may be sent from the computer to the receiving device (printer or video display) in two different manners; serial and parallel. In parallel communications, the 8 bits representing a character are all sent at one time to the receiving device. In serial communications, each of the 8 bits are sent one at a time to the receiving device. Generally, printers use parallel communication.

A wide range of printers are available for use with the Apple IIe. The printer interface card can be installed in slots 1-7 in the main board. Generally, openings 7, 10, 11, or 12 on the rear of the IIe are used for the installation.

The use a printer with the Apple IIe will be covered in greater detail in Appendix H.

### **Hand Controls**

The 9-pin connector in the IIe's rear panel is generally used for the installation of hand controls such as game paddles, a joystick, etc.

A Game I/O Connector is also available on the main board (see Illustration 1-2). Controllers using a 16-pin connection can be attached via the connector.

### **Apple IIe Controller Cards**

Earlier in this chapter, we mentioned the seven plug-in slots on the Apple's main board. The slots were designed to accept special circuit boards known as controllers or cards.

Cards are available for a number of different applications. For example as we mentioned previously, a parallel or serial communications card is required to install a printer on your Apple IIe. Controller cards are required for installing a disk drive with you Apple IIe.

A number of other cards are also available for the Apple IIe. These include communications cards which allow you to connect you Apple IIe to a modem, which in turn allows your Apple to accept data transmitted over telephone lines. Cards are also available which convert digital signals to analog and vice versa. Finally, cards are also available that allow the Apple IIe to run under an operating system completely different from Apple DOS known as CP/M

### **Communications**

The Apple IIe can communicate with another Apple computer located a few miles away or a few thousand miles away. A device known as a **modem**, your telephone, and telephone lines allow this communication to take place.

A modem converts signals generated by the IIe into signals which can be transmitted over telephone lines. A modem will also reconvert the signals transmitted over phone lines back into a signal which can be understood by the IIe.

Generally, a modem is connected to the IIe using a Super Serial Card. Once the Super Serial Card has been installed (in slots 1-7), connect the card to the modem. Then, connect the modem to your telephone.

A modem allows the user to access one of the many available information services. Examples of these information services include CompuServe, The Source, and the Dow Jones News and Quotes service. A modem also allows communications with other Apples.

### **SOFTWARE**

Software can be described as the instructions or programs that cause the computer to operate. Several different classifications of software exist for the performance of different functions. These can be classified as operating systems, languages, and applications programs.

## Operating Systems

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 standard operating system supplied with the IIe with a disk drive is DOS 3.3. DOS stands for disk operating system.

There are several other operating systems available for the IIe. The most well-known of these is CP/M. To install CP/M, a Z80 peripheral card must be installed in slot 7 of the IIe. The Z80 card contains a Z80 microprocessor. The Z80 is necessary to run CP/M.

## Language Translators

Programs are generally written in a high-level language that is different from the instructions the computer uses. A program known as a language translator must be used to translate the high-level language into a form that the computer can comprehend.

There are three categories of language translators: **interpreters**, **compilers**, and **assemblers**.

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 Applesoft BASIC language that is standard on the IIe is an interpreted language.

An **assembler** translates program instructions one by one into instructions that the CPU can comprehend.

### **Applesoft BASIC**

The most widely used language on the IIe is Applesoft BASIC. Applesoft BASIC is resident in ROM in the IIe's main board. Applesoft BASIC is a floating point language. This allows it to process both very large and very small numbers.

### **Integer BASIC**

Integer BASIC is automatically loaded into RAM when the system is booted from the DOS 3.3 System Master diskette. Unlike Applesoft BASIC, Integer BASIC does not have the capability to process numbers with decimal portions.

### **Other Languages**

A number of other languages are available for the IIe including PASCAL, FORTRAN, Logo, PILOT, and 6502 Assembly Language.

### **Applications Programs**

Applications programs are those written to accomplish a specific task. Examples of applications programs include word processing programs, electronic spread sheets, data base systems, and accounting programs. Generally, applications programs are stored on cassette or diskette and are transferred into RAM, where the program is available to the computer.

Some of the more popular applications programs available for the IIe include:

<b>Database:</b>	PFS: The Personal Filing System DB Master Quick File II
<b>Electronic Spreadsheets:</b>	Visicalc Plan 80
<b>Word Processors:</b>	Wordstar Apple Writer

# **CHAPTER 2. APPLE IIe INSTALLATION, TROUBLESHOOTING, AND OPERATION**

---

## **INTRODUCTION**

In this chapter, we will explain in detail steps involved in installing the Apple IIe. We will then outline a few troubleshooting hints for the IIe. Finally, we will discuss IIe operation.

## **INSTALLATION**

The first step in installing the IIe is to position the console on a flat desk or table near a household AC outlet.

The next step is to connect the AC power cable to the power cord outlet on the rear of the IIe. Then, connect the other end of the AC power cable to a household outlet. Turn the power switch (on the IIe's rear panel) to the on position. If the connection was properly made, the IIe should power on.

You are now ready to install a monitor or TV set as described in Chapter 1. Be certain to turn off the IIe's power switch and unplug the power cord prior to making this connection.

## **TROUBLESHOOTING**

Table 2-1 contains a number of hints that may be able to help you isolate any problems should your IIe not function properly.

**Table 2-1. Apple IIe Troubleshooting Hints**

<b>Symptoms of Problem</b>	<b>Possible Solution</b>
<p>Apple IIe powers on, but cursor does not appear.</p>	<p>Be sure that the video display is plugged in and turned on. Check the display's contrast and brightness controls. If the preceding solutions do not solve the problem, disconnect all peripherals from the IIe except the display and run the IIe's selftests (explained later).</p>
<p>Apple IIe will not power on.</p>	<p>Check to be sure that the power cord is properly connected to both the IIe and to the wall outlet. Be sure that the wall outlet is receiving power.</p>
<p>Program will not load from cassette recorder.</p>	<p>Check the cassette tape to be sure that it has been rewound. Also, check the cable connection between the recorder and the IIe. Check the volume setting on the cassette recorder. Finally, certain recorders do not work well with the IIe. If you suspect this to be the problem, try another recorder.</p>
<p><b>SYNTAX ERROR</b> message.</p>	<p>This is generally due to the incorrect entry of an Applesoft or Integer BASIC command. Be careful not to use lower-case characters in reserved words.</p>

**Table 2-1. Apple IIe Troubleshooting Hints (con't.)**

<b>Symptoms of Problem</b>	<b>Possible Solution</b>
Must Boot From Slot 6 message.	The PASCAL operating system must be booted with the system diskette in drive 1—which must be attached via slot 6. If necessary, reinstall the disk controller in slot 6.
Disk drive's in use light remains on.	Check to be certain that the drive's door is closed. Try pressing Ctrl-Reset to stop the drive.
Drive emits weak sounds while rotating.	If a card is installed in the auxiliary slot, the disk controller card should not be installed in slot 3. Reinstall the disk controller card in a different slot.
Disk drive occasionally rattles during operation.	Be sure the diskette being used was properly formatted. Be certain that the disk was properly inserted into the drive.

**Built-In Self Tests**

The IIe contains a series of built-in self-tests designed to evaluate the operation of the unit's internal circuitry. These tests do not check the operation of any devices attached to the IIe.

To begin the self-test, press the Solid Apple and Ctrl-Reset keys simultaneously. Release Ctrl-Reset first followed by the Solid Apple key.

The self-tests require approximately 20 seconds of execution time. During the self-tests, patterns will move across the screen. Upon successful completion of the tests, the following message should appear on the screen:



## KERNEL ON

If any other message appears, the IIe requires servicing.

### **APPLE IIe OPERATION**

The IIe is started up by booting DOS (abbreviation for disk operating system) from the System Master diskette. The System Master diskette is included with your Apple IIe Disk II drive and is labeled "DOS 3.3 System Master." You can either use that diskette or a copy to start up the IIe. It is recommended that once you learn diskette copying procedures (see Chapter 5), you use a copy of the System Master diskette for everyday IIe operations.

To insert the diskette, open the drive door to drive 1. The System Master must be inserted in drive 1. This is the drive attached to the controller card in slot 6. Insert the System Master diskette into the drive with its label facing up. The label side should be nearest your hand and the side of the diskette with the oval slot should be inserted into the drive. Once the diskette is in place, close the drive door.

Next, power on the monitor connected to the IIe. Then, power on the IIe itself by pressing the toggle switch located at the rear of the unit next to where the power cord plugs into the machine. The IIe will beep, the disk drive will spin, and the drive light will blink on and off as DOS is loaded. The green power light to the left of the open Apple key will also be lighted and the display depicted on page 33 will appear on the monitor.

The symbol ] is the Applesoft BASIC prompt. When the symbol appears, the user can enter either Applesoft BASIC or DOS commands.

APPLE II  
DOS VERSION 3.3 SYSTEM MASTER  
JANUARY 1, 1983  
COPYRIGHT APPLE COMPUTER, INC 1980,1982

### Apple IIe Keyboard

The Apple IIe keyboard is depicted in Illustration 2-1. The Apple IIe keyboard is arranged differently than the keyboard on the Apple II or II Plus.

First of all, the IIe contains 12 additional keys. These new

**Illustration 2-1. Apple IIe Keyboard**



keys include:

DELETE  
TAB  
CAPS LOCK  
↑  
↓  
Open Apple Function Key  
Solid Apple Function Key  
Special Character Keys

The REPT key on the Apple II and II Plus has been replaced with an auto-repeat feature in the IIe. The auto-repeat feature causes all printing characters to automatically repeat if the key is held down longer than a second or two.

The Apple IIe keyboard allows the user to send any one of 128 ASCII characters to the computer. Of these 128 characters, 96 are printing characters. That is, they will be echoed on the screen when entered at the keyboard. The printing characters include:

- 26 lowercase letters
- 26 uppercase letters (output with caps lock or shift depressed)
- 10 numerics
- 34 special characters

The non-printing characters consist of the 32 control characters. The control characters are output by simultaneously pressing the Control key with a second key (generally a letter).

Several of the control codes can also be output via special keys on the IIe keyboard. For instance, the same control code that is output by Ctrl-M is also output by the Return key. Therefore, the ASCII return code (ASCII 13 decimal) can be produced in two separate ways.

As a general rule, the printing characters are used for outputting information, while the control characters are used to instruct the system to perform some function.

We will discuss the more important keys on the IIe keyboard in more detail in the following sections.

### **Space Bar**

The space bar (located at the bottom of the IIe keyboard) generates the space character (ASCII 32 decimal). Be certain to include the space character where specified in DOS or BASIC commands.

### **Shift**

The Shift key causes the uppercase character to be output for the key being pressed. There are two Shift keys on the IIe keyboard, one on the left and one on the right side of the keyboard.

### **Caps Lock**

The Caps Lock key is located just below the Shift key on the left-hand side of the keyboard. When the Caps Lock key is on (depressed), all alphabetic keys will be output as uppercase characters. No other keys will be affected. This allows the user to output only uppercase letters, while still being able to enter numbers.

Once Caps Lock is on (depressed), it can be turned off by pressing the key a second time. When the key is pressed, it clicks back into the up position.

### **Cursor Control Keys**

The cursor control keys can be used to move the cursor around the screen. The cursor control keys are described below:

Left-Arrow	Moves the cursor one position to the left.
Right-Arrow	Moves the cursor one position to the right.

Down-Arrow	Moves the cursor down by one line.
Up-Arrow	Moves the cursor up by one line. (ESC key must first be pressed.)
Tab	Moves the cursor to the next tab setting. Tabs generally are set after every eight characters. This key only functions with certain programs (generally word processing software).
Return	Moves the cursor to the beginning of the next line.

### **ESC Key**

The ESC key outputs the control code for ESC (ASCII 27 decimal). The ESC key is often used to either place the computer in the escape mode or to enter an escape sequence.

### **Open Apple Key**

The Open Apple key appears to the left of the space bar. The Open Apple key can be used to restart the IIe when it has already been powered on. This is accomplished by pressing the Open Apple key simultaneously with Ctrl-Reset.

Pressing the Open Apple key also has the same effect as pressing the button on game controller #0. The Open Apple key can be used as an alternative to pressing the game controller.

### **Solid Apple Key**

The Solid Apple key is located to the right of the space bar. The Solid Apple key when pressed simultaneously with Ctrl-Reset starts the Apple IIe's built-in self-test.

The Solid Apple key also has the same effect as pressing the button on game controller #1.

## **Reset Key**

The Reset key is located to the right of the Delete key on the right hand side of the IIe keyboard. Pressing Ctrl-Reset causes execution of most IIe programs to stop.



## CHAPTER 3. APPLESOFT BASIC PROGRAMMING

---

### INTRODUCTION

BASIC is the most widely used personal computer programming language with the Apple IIe being no exception. The IIe includes two different versions of BASIC; Applesoft and Integer. Applesoft BASIC is contained in ROM and will be active when the IIe is started up. Integer BASIC is loaded from the System Master diskette when DOS is booted.

Applesoft BASIC is a floating point language. This allows Applesoft to handle numbers with decimal portions, as well as extremely large and small numbers expressed in decimal notation.

Integer BASIC can only deal with integers. Obviously, Applesoft is a more practical version of BASIC than Integer. For this reason, our discussion of BASIC programming will be centered around Applesoft BASIC.

### Switching from Applesoft to Integer & Vice Versa

As mentioned in the preceding section, Applesoft BASIC will be active when the IIe is started up. If you wish to switch to Integer BASIC, enter INT via the keyboard. The Integer BASIC prompt ( > ) will appear.

If you wish to switch back to Applesoft BASIC, enter FP via the keyboard. The Applesoft BASIC prompt (|) will appear.



## **Compiled & Interpreted Languages**

The conversion from a high level language to machine language is either done with an interpreter or a compiler. A compiler is a program that converts an entire high level program to machine language. A compiler performs a complete translation of the set of instructions before the program is actually executed. An interpreter converts each instruction to machine language as the program is executed.

A compiled language such as CBASIC executes programs very quickly. However, an interpreted language such as Applesoft BASIC is easier to use, because it does not need to be compiled. Unfortunately, interpreted languages require more time to execute because each instruction must be translated into machine language as the program proceeds.

## **Immediate & Program Modes**

The immediate mode is also known as the direct or the 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 entered:

```
PRINT "JIM SMITH"
```

the following would be displayed on the video screen:

```
JIM SMITH
```

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

Illustration 3-1 contains an example of the entry of a program in the program mode and its execution.

**Illustration 3-1. Program Mode Entry & Execution**

```
] NEW  
] 10 PRINT "JIM SMITH"  
] 20 PRINT "1220 EUCLID AVE"  
] 30 PRINT "CLEVELAND, OH 44122"  
] 40 END  
] RUN  
JIM SMITH  
1220 EUCLID AVE  
CLEVELAND, OH 44122  
]
```

**Line Numbers**

In the program mode, program lines must begin with a line number. A line number is a one through five digit number entered at the beginning of a program line. The line number at the beginning of a program line is the only difference between it and an immediate mode line.

No two line numbers can be the same. If the same line number is used more than once in a program, the line most recently entered will replace the original.

The execution sequence of a BASIC program is determined by the value of its line number. The lowest line numbers will be executed first, followed by program lines with higher line numbers. Even if program lines are not arranged in sequential order, the Applesoft BASIC interpreter will place the lines in the correct order.

Adding program lines to a program stored in RAM is very easy. Just type in the line number followed by the program line. The line will be inserted in the program in the position indicated by its line number. For example, by adding the following line to the program in Illustration 3-1:

```
35 PRINT "216-777-5579"
```

the phone number for Jim Smith will be displayed on the line following his city, state, and zip.

Program lines can be deleted by typing the line number to be deleted followed by Return. For example, the following entry:

```
30
```

would result in line 30 being deleted.

Program lines can be changed by merely retyping the new line. The existing line in the IIe's memory will be replaced with the new line. For example, the following entry:

```
10 PRINT "THOMAS HILL"
```

would result in "THOMAS HILL" being output rather than "JIM SMITH" in the program Illustration 3-1.

### **NEW Command**

You may have noticed the execution of the NEW command in Illustration 3-1. The NEW command is used to erase an old program from memory before a new one is typed in.

The IIe can only store one program in RAM at any one time. If you attempt to enter a new program while another program is already stored in RAM, the new program will be merged with the existing program.

### **END Statement**

Notice the last line in the program in Illustration 3-1. That line consists only of the line number plus the BASIC reserved word END.

The END statement identifies the end of a program, and instructs BASIC to return to the immediate mode. Obviously, the END statement should be the last line in your program.

Actually, Applesoft BASIC does not require an END statement. When the program's final statement is executed, it will end. However, it is good programming practice to end a BASIC program with the END statement. Integer BASIC does require an END statement.

### Executing a Program

A program is executed in the program mode by entering the RUN command. This is shown in Illustration 3-1. Every time RUN is executed, the program will be re-executed. As previously discussed, in the immediate mode, each program line will be executed when the Return key is pressed.

### Program Lines & Display Lines

A **display line** can be defined as one row on the video display. A **program line** is regarded by the BASIC interpreter as one line, regardless of the number of display lines it occupies on the screen. The end of a program line is signalled when the Return key is pressed. Program lines are generally limited to 255 characters.

### Multiple Statement Program Lines

A **statement** can be defined as an instruction to the computer. The terms statement and command are often used interchangeably. Most programs consist of a large number of statements.

The following are examples of statements:

```
PRINT "TIM GREGORY"  
070 DIM A(15)  
100 C=2*B
```

Every statement in Applesoft BASIC must contain at least one **key** or **reserved** word. A keyword identifies the calculation, decision, input, or output function to be performed. The keywords are described individually in Chapter 4 and are listed in alphabetical order in Appendices A and B.

In addition to keywords, numeric constants, string constants, variables, and special symbols may appear in a BASIC statement. These are known as the statement **parameters**.

Applesoft BASIC allows the user to place more than one statement on a single program line. Multiple statements must be separated with a colon (:). The following is an example of a multiple statement program line:

```
10 A = B * 7:PRINT B
```

### Listing a Program

As mentioned earlier, the LIST command can be used to display program lines currently stored in RAM. Remember, if the NEW command is issued, the program in RAM will have been erased, and can no longer be displayed by LIST.

LIST is used in the following configuration:\*

```
LIST [line 1 - line 2]**
```

where *line 1* is the line number of the first line to be listed, and *line 2* is the line number of the last line to be listed.

---

\*In this book, a standard format will be used to describe BASIC keyword configurations. The keyword will be displayed in our regular type style in upper case. Parameters will be displayed in our italic type style in lower case. Optional parameters will be enclosed in brackets.

\*\* This option is only available in Applesoft. A comma may be substituted for the hyphen.

LIST can be used without any parameters to list the entire program. LIST can also be used with a single line number to list just that program line. If LIST is used with the following format:

LIST *line 1-*

*line 1* and all subsequent lines will be listed.

If LIST is used as follows:

LIST *-line 2*

*line 2* and all lines preceding it will be listed.

In certain situations (especially when a long program is being listed, you may wish to cancel or temporarily halt a program listing. With the IIe, you can cancel an Applesoft program listing by pressing Ctrl-C.

A program listing can be stopped temporarily by pressing Ctrl-S and resumed by pressing the space bar.

### **Error Messages**

When the IIe encounters a statement with an error, an **error message** will be displayed. The error message will be displayed in the following formats:

? SYNTAX ERROR ← Applesoft BASIC  
 \*\*\*SYNTAX ERROR ← Integer BASIC

The various Applesoft and Integer BASIC error messages are described in Appendices D and E.

### **BASIC PROGRAM EDITING**

In our discussion of the IIe keyboard, we mentioned several of the keys used in editing. These include:

← used to backspace the cursor.  
 → used to move the cursor forward.

↑ used to move the cursor up.  
↓ used to move the cursor down.

The following keys and key combinations can also be used in editing:

Ctrl-X	erases the current line.
Esc-@	clears the display and homes the cursor.
Esc-A or K	moves the cursor to the right.
Esc-B or J	moves the cursor to the left.
Esc-C or M	moves the cursor down one row.
Esc-D or I	moves the cursor up one row.
Esc-E	erases text from cursor to end of line.
Esc-F	erases text from cursor to end of text windows.

Once Esc I, J, K, or M has been pressed, the IIe will be in the edit mode. In the edit mode, it will no longer be necessary to press Esc with I, J, K, or M to move the cursor. The cursor can be moved by pressing I, J, K, or M. The edit mode can be ended by pressing any key except I, J, K, M, Ctrl, or Shift.

## APPLESOFT BASIC DATA TYPES

Data can be classified under two major categories: **text** and **numeric**. Text data consists of characters. These characters are generally used within strings.

Examples of numeric data include:

- Integers
- Floating Point Numbers
- Scientific Notation

Each of these data types will be discussed in the following sections.

## STRINGS

A **string** consists of one or more characters enclosed within double quotation marks. The following are examples of strings:

"F. SCOTT FITZGERALD"  
 "149 LEXINGTON AVE"  
 "NEW YORK, NY 10017"  
 "212-349-9879"

Notice that a string can contain both letters, numbers and symbols. Any string containing numbers cannot be used in a mathematical operation, unless it is first converted into numeric data. String to numeric data conversion is covered later in this chapter.

## NUMERIC DATA

The IIe can use either of two types of numeric data—integers and floating point numbers. Integers do not have a decimal portion while floating point numbers can have a decimal portion.

Applesoft BASIC can process both integers and floating point numbers while Integer BASIC can only process integers.

### Floating Decimal Point

With floating decimal point numbers, a decimal point is always assumed. Any number of digits can be placed on either side of this decimal point. Even with numbers with no decimal position, a decimal point always is assumed following the number's last digit.

Floating point numbers of up to 9 digits can be used with Applesoft BASIC. For example, the following entry of a nine digit floating point number:

```
PRINT .566666666
```

would generate a nine digit display. If a 10 digit floating point number was entered:

```
PRINT .5666666666
```

the last digit would not be displayed and the number would be rounded as follows:



.56666667

Commas may not be included within numeric data. For example, 109000 would be a valid number in Applesoft BASIC while 109,000 would be invalid.

**Floating point numbers** include both integers, as well as numbers with decimal positions. The following are examples of floating point numbers:

-0.789  
5  
77.39  
0  
+.000001  
67.98

Negative floating point numbers should be preceded with the minus sign (-). Positive floating numbers can optionally be preceded with the plus sign (+), however, a floating point number is assumed positive if it doesn't have a sign.

## Integer

An **integer** is a number without a decimal position. Integers can either be positive or negative. The following are examples of integers:

-1134  
0  
1  
-1  
17945  
+32

Integers can range from -32768 to +32767. Negative integers are preceded with the (-) sign. Positive integers can be preceded with the (+) sign, although integers without a (+) sign are assumed to be positive.

## Scientific Notation

Applesoft BASIC uses **scientific notation** to express either extremely large or extremely small numbers. A number in scientific notation takes the following format:

$$\pm x E + yy$$

Where;

- $\pm$  is an optional plus or minus sign.
- $x$  can either be an integer or floating point number. This value is known as the coefficient or mantissa.
- $E$  stands for exponent.
- $yy$  is a one or 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 the positive exponents. The decimal point is moved to the left with negative exponents.

The following examples specify a number in both standard floating point and scientific notation:

1000000  $\longrightarrow$  1 E6  
 .000001  $\longrightarrow$  1 E-6  
 57500000  $\longrightarrow$  5.75 E+07  
 -.00000479  $\longrightarrow$  -4.79 E-06

Any integers containing 10 or more digits will be expressed in scientific notation as shown in the following example.

```
PRINT 121212121212
      1.21212121E+11
```

Notice that the decimal portion of the preceding example contains 8 digits of precision. Applesoft will round any additional digits.

Applesoft can only handle numbers expressed in scientific notation in the following range:

Largest floating point number → +1.70141183E+38

Smallest floating point number → +2.93873588E-39

If a larger number is encountered, the following error message will be displayed.

### OVERFLOW ERROR

Any numbers smaller than that allowed will be assigned a value of 0.

### VARIABLES

So far, we have only discussed data **constants**. A constant can be defined as a fixed value. The following are examples of string and numeric constants.

"JACK NOVET"

"375"

27.59

0

100000

A name can be used to express data as well as a constant. **Variables** are used to express data as a name.

### BASIC Variables

A variable can be defined as a quantity that can assume any one of a group of values. Variables are represented by variable names. These consist of a letter followed optionally by additional letters and/or numbers. The value assumed by a variable is subject to change, depending upon the program statement being executed. For example, in the following:

```

100 LET A = 5.0
200 LET B = 7.0
300 LET A = A + B

```

the variable A is initially assigned a value of 5.0 and B is assigned a value of 7.0. In line 300, the variable A is assigned a new value equal to the sum of variables A and B, which is 12.0. The previous value of A is erased.

### BASIC Variable Names

Applesoft BASIC allows any group of up to 238 characters to be used as a variable name--as long as the first character of the group is a capital letter of the alphabet, and as long as the variable name does not duplicate a reserved word (see Appendices A & B). Examples of reserved words are:

LET, GOTO, IF, READ, DATA

The following are examples of valid Applesoft BASIC variable names:

A	JOHN
B23456	N4N
DOT	B%
A2	N

While the following are invalid variable names:

2BB7	END
1A	FOR
PRINT	COS

All of the preceding examples of valid variable names should be used to represent numeric data. Variable names can also be used to represent string data. These are known as **string variables**. String variable names consist of a valid variable name followed by the dollar sign (\$). The following are examples of valid string variable names.

A\$	NED\$
ZIP\$	MOP\$
A7\$	N222\$

A distinction can also be made among numeric variable names between floating point variable names and integer variable names. In the following example, A1 is used to represent a floating point number, while A% represents an integer.

```
100 A1 = 1.75:A% = A1 * 2
200 PRINT A1, A%
RUN
      1.75      3
```

Notice that only the integer value of A% is output. The decimal portion is dropped or **truncated** as A% can only accept integer values.

Keep in mind that although Applesoft allows variable names of up to 255 characters, only the first 2 alphanumeric characters are recognized by the interpreter. In other words, the following variable names would be identified as being identical by the Applesoft interpreter.

```
TELLER
TENNIS
TENNESSEE
```

However, the special symbols which identify the variable type (i.e. \$, %) differentiate among variables with identical 2 character names. The following variable names would all be identified as unique by the Applesoft interpreter.

```
TE%
TE
TE$
```

## TABLES & ARRAYS

Earlier in this chapter, we introduced the concept of variables. A variable is designed to hold a single data item--either string or

numeric. However, some programs require that hundreds or even thousands of variable names be used.

Obviously, the use of thousands of individual variable 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:

$$A(0), A(1), A(2), A(3), A(4), \dots, A(100)$$

Note that each subscripted variable is a unique variable. In other words,  $A(0)$  differs from  $A(1)$ ,  $A(2)$ ,  $A(3)$ ,  $A(4)$ , etc.

Subscripted variables should be visualized as an array (or table). In our previous example, the data contained in the array defined by  $A$  would consist of one row with 101 columns in it. Such an array is a single-dimension array.

In Applesoft BASIC, arrays of up to eleven\* elements can be used as needed in a program. Arrays which contain more than eleven elements must first be identified via the Dimension (DIM) statement. When an array is dimensioned, BASIC will reserve an area in memory for that array's elements. The following Dimension statement will dimension a numeric array of 16 elements.

$$100 \text{ DIM } B(15)$$

More than one array can be defined with a single DIM statement. This is shown in the example below:

$$100 \text{ DIM } Z(5,2), B(100), C(2,3)$$


---

\*An array of eleven elements would contain the subscripts 0 through 10 inclusive. For example, an array dimensioned as  $A(10)$  would have eleven elements  $A(0)$  through  $A(10)$  inclusive.

A DIM statement should appear in a program before the array variable it is dimensioning appears. If an array variable is used in a program before it is dimensioned, the Bad Subscript error may occur.

An array can also consist of two dimensions. Such an array is known as a two-dimensional array (or table). An example of an array of 4 rows and 3 columns is shown in Illustration 3-2.

A two-dimensional array contains two subscripts. The first subscript contains the row location, while the second subscript contains the column location. The subscripted variable A(1,0) identifies the darkened area in the array shown in Illustration 3-2.

**Illustration 3-2. Two-Dimensional Array**

		Columns		
		0	1	2
Rows	0			
	1			
	2			
	3			

### Expressions and 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 \* 2  
 14 < 21  
 X AND Y

Applesoft BASIC includes several types of expressions including **arithmetic**, **relational**, and **Boolean**. In our previous examples, the first three examples are arithmetic expressions, while the

fourth and fifth are 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 phrase describing the operation to be undertaken is known as the **operator**. The operators in our previous example were as follows:

+  
+  
\*  
<  
AND

The constants or variables which are affected by the operator are known as **operands**.

### Compound Expressions and Order of Evaluation

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 + B) \* 7-4  
(A + B) AND (C + D)  
IF A = 1 AND B = 1 THEN C = 1

With compound expressions, it is necessary that the computer knows which operation should be undertaken first. Applesoft BASIC follows a standard order of evaluation within compound expressions. This order is outlined in Table 3-1.

Note that parentheses have the highest precedence level. In other words, any expression enclosed within parentheses will be evaluated first. If more than one set of parentheses appears in an expression, these will be evaluated from left to right.

One pair of parentheses can be used to enclose an operator enclosed within another set. In such an instance, Applesoft



BASIC will evaluate the expression within the innermost set of parentheses first, followed by the next innermost set, etc.

**Table 3-1. Order of Evaluation**

	<b>Operator</b>	<b>Description</b>
<b>Parentheses</b>	( )	Used to alter order of evaluation.
<b>Arithmetic Operators</b>	^ - * / + -	Exponentiation Unary Minus Multiplication Division Addition Subtraction
<b>Relational Operators</b>	= <> < > <= >=	Equal To Not Equal To Less Than Greater Than Less Than or Equal To Greater Than or Equal To
<b>Boolean Operators</b>	NOT AND OR	Logical Complement Logical AND Logical OR

When expressions have the same order of evaluation, they will be evaluated in order from left to right within the compound expression.

### **Arithmetic Operations**

The symbols used for addition, subtraction, multiplication, division, and exponentiation are known as **arithmetic operators** in BASIC. The symbols + and - are used for addition and subtraction respectively. The asterik (\*) is used to indicate multiplication, while the slash (/) is used to indicate division.

When a + or - sign precedes a number, the symbol is used to specify that number's sign. When + or - is used to change a number's sign, that usage is known as a **unary** operation. Unary operators can be used to change the sign of a numeric constant or variable as shown below:

```
100 LET A = -A
```

When unary operators are used in the manner shown above, the unary operation is regarded as an arithmetic operation.

The term **arithmetic expression** is used to describe the use of an arithmetic operator with numeric constants and/or variables. The following are examples of arithmetic expressions.

```
X + Y + 70
100/A + B
3000 * 10 + 1
```

**Exponentiation** is the process of raising a number to a specified power. For example, in the following,

$$A^5$$

the numeric variable A would be evaluated as:

$$A * A * A * A * A$$

In Applesoft BASIC, exponentiation is indicated with the caret arrow symbol, ^ .

Exponentiation can be used in an arithmetic expression as shown below:

$$8*3+7^2$$

The preceding expression would evaluate to 73.

A relational operation evaluates to either true or false. For example, if the constant 1.0 was compared to the constant 2.0 to see whether they were equal, the expression would evaluate to false. In Applesoft BASIC, a non-zero value 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 results are generated by the following relational expressions.

$$\begin{array}{l}
 5 > 7 \longrightarrow 0 \text{ (false)} \\
 3 = 3 \longrightarrow 1 \text{ (true)} \\
 2 < > 2 \longrightarrow 0 \text{ (false)} \\
 (2 = 2) * 4 \longrightarrow 4 \\
 (1 > 7) + 7 \longrightarrow 7
 \end{array}$$

The first three examples are easy enough to understand. In the fourth example, the relational expression  $(2=2)$  is evaluated first as true or 1. This result is then multiplied by 4 with a product of 4 as the result. In the fifth example, the relational expression  $(1>7)$  evaluates as false or 0. This result is added to 7, with the result being 7.

Relational operations using numeric operations are fairly straightforward. However, relational operations using string values may prove confusing to the first-time computer user.

Strings are compared by taking the ASCII value for each character in the string one at a time and comparing the codes.

If the strings are of the same length, then the string containing the first character with a lower code number is the lesser. Blank spaces are counted in string comparisons and have the ASCII value of 32.

The following comparisons between strings would all evaluate as true.

```

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

```

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

### Logical Operators

Logical or Boolean operations 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), and OR (disjunction). These are best explained with a simple analogy. Suppose that Steve and Sherry were shopping in the produce department of their grocery store. If they decided to collectively purchase an item if either of them individually wanted that item, they would be acting under the OR logical operator.

Now, suppose that Steve and Sherry decided that they would only purchase an item if they both wanted that item. They would then be acting under the AND logical operation.

Now, suppose that Sherry was angry with Steve. If Sherry decided not to purchase the items that Steve wanted, she would be acting under the NOT logical operation. The NOT, AND, and OR logical operators are summarized in Illustration 3-3.

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
B = -1: PRINT NOT B
```

In the first example, the result of the logical operation will be true if variable X has a value greater than 10 or if variable Y has a value 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. Otherwise, program control will branch to the next line.

In the third example, B is set to a value of -1 (true). The value of NOT B is then printed. This will be 0 or false.

Illustration 3-3 contains tables that may prove of help when evaluating program statements using logical operators in combination with relational operators.

**Illustration 3-3. Logical Operators****NOT Operation**

T	F	A Operand
---	---	-----------

F	T	NOT A
---	---	-------

**AND Operation**

T	T	F	F	A Operand
---	---	---	---	-----------

T	F	T	F	B Operand
---	---	---	---	-----------

T	F	F	F	A AND B
---	---	---	---	---------

**OR Operator**

T	T	F	F	A Operand
---	---	---	---	-----------

T	F	T	F	B Operand
---	---	---	---	-----------

T	T	T	F	A OR B
---	---	---	---	--------

**APPLESOFT BASIC STATEMENTS**

In the next several sections, we will discuss many of the more commonly used statements in Applesoft BASIC. These include the following:

- Remark Statements
- Assignment Statements
- Output Statements
- Input Statements
- Loops
- Conditional Statements
- Branching Statements
- Subroutines
- Applesoft BASIC Functions

## Remark Statements

Remark statements are used to include a programmer's comments within a program. It is good programming practice to include numerous Remark statements in your programs. Not only do Remark statements make your programs easier for others to understand, they also help you remember your program's logic.

Remark statements consist of a line number, the reserved word REM, and the programmer's comment. An example of a Remark statement is given below.

```
100 REM Initialize 1 to 0
```

Remark statements are ignored by the Applesoft BASIC interpreter, but are included in program listings.

In multiple line statements, the REM statement must be the final statement. The Applesoft BASIC interpreter ignores all text following the keyword REM.

## Assignment Statements

Assignment statements were discussed briefly earlier in this chapter. Assignment statements are used to assign values to variables. The following are examples of assignment statements.

```
100 LET A = 7
200 B = 42
300 NA$ = "PHIL"
400 X=1: Y=2: Z=3
```

Notice that the keyword LET is optional. Generally, LET is assumed. Both string and numeric variables can be assigned values with an assignment statement. Also, multiple assignment statements can be included in a single line, as long as each of the individual statements is separated with a colon.

## DATA, READ Assignment Statements

Assigning values to a large number of variables with individual assignment statements could prove very cumbersome. The DATA, READ statement can be used to assign values to a large number of variables. The following is an example of a DATA, READ statement.

```
100 DATA 100, 500, 1000, "JACK"
200 READ A, B, C, D$
```

The DATA statement creates a list of constant values known as a DATA list. The items in the DATA list are assigned sequentially to the variables in the READ statement. A DATA list is depicted in Illustration 3-4.

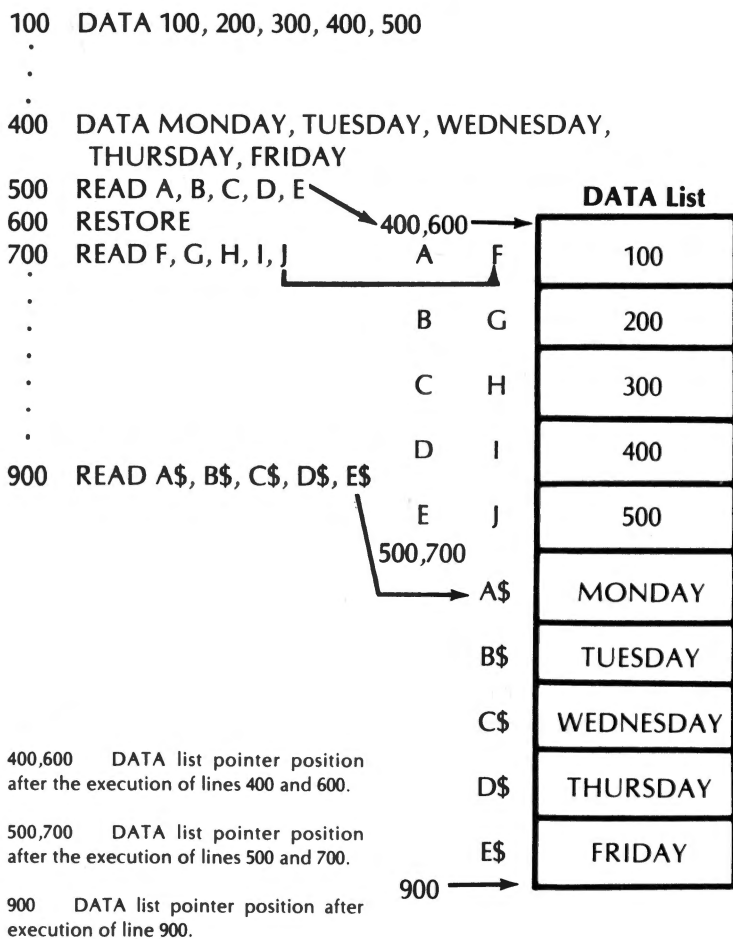
DATA statements may contain numeric or string values. These values must be separated or **delimited** with commas. DATA statements may appear at any point in the program. No other statements can appear in the same program line with a DATA statement.

The DATA list uses a pointer to indicate which value within the list is to be assigned to the next variable in a READ statement. Before the first READ statement is encountered, the DATA list pointer will point to the first value in the DATA list. As values from the DATA list are assigned to variables in the READ statement, the pointer will move sequentially to each successive item in the DATA list.

The values from the DATA list must match the type of variable to which they are assigned in the READ statement. In other words, a string value cannot be assigned to a numeric or vice versa.



### Illustration 3-4. DATA List



The RESTORE statement is used to reset the DATA list. In Illustration 3-4, note the use of the RESTORE statement. After DATA list values have been read into A, B, C, D, and E in line 500, a RESTORE statement is executed. This causes the DATA list pointer to be reset to the beginning of the DATA list.

## Outputting Data

In some of our preceding examples, we touched upon the use of the PRINT statement to display data. The PRINT statement can be used to display both numeric and string data.

The following program statement,

```
100 PRINT "VENDOR LIST"
```

would display the following at the current cursor position:

```
VENDOR LIST
```

The first item in a PRINT statement is displayed at the cursor's current location.

When the Apple IIe is used on the 40 column mode, two or more data items could be output to the screen by separating these items with commas in the PRINT statement. Upon encountering a comma as a delimiter, PRINT will output the next item at the next tab stop. In Integer BASIC, tab stops are set at columns 1, 9, 12, 25, and 33. In Applesoft BASIC, tabs are set at columns 1, 16, and 33.

When the Apple IIe is used in the 80-column mode, only two tab stops are present. In Integer BASIC, the tab stops are located at columns 1 and 9. In Applesoft, they are located at columns 1 and 17.

The following examples illustrate the use of the comma with PRINT in the 40 and 80 column modes.

### Integer 40 Column

```
> PRINT 1,2,3,4,5,6
  1  2  3  4  5
  6
```

### Applesoft - 40 Column

```
] PRINT 1,2,3,4,5,6
  1      2      3
  4      5      6
```

**\*Integer 80 Column**

```
> PRINT 1,2,3,4,5,6
1          6
```

**\*Applesoft - 80 Column**

```
] PRINT 1,2,3,4,5,6
1          6
```

A semicolon can also be used to separate the items in a PRINT statement. A semicolon causes the next item in the PRINT statement to be displayed immediately after the preceding item. Unlike the use of the comma in a PRINT statement, when semicolons are used to separate items, no blank spaces appear between string or numeric values.

When a PRINT statement has finished execution, the cursor moves to the left margin of the following line. This is known as a **carriage return/line feed**.

If a comma or semicolon occurs at the end of a PRINT statement, the carriage return/line feed will be suppressed. If a comma is placed at the end of the PRINT statement, the next PRINT statement will begin output at the next print zone after the last item is displayed. If a semicolon is placed at the end of the PRINT statement, the next PRINT statement will begin output immediately following the last item displayed.

In this section, we have only discussed sending output to the video display. Output can also be sent to the printer. This is accomplished by executing the PR# statement prior to PRINT.

The usage of PR# to send data to the printer is discussed in Appendix H.

---

\*In the 80 column mode, each comma causes the subsequent value to be displayed in the second tab position. In our examples, 2,3,4,5, and finally 6 were successively displayed in the second tab position.

## INPUT Statements

Data can be input into the computer while a program is being executed. This is accomplished with the INPUT statement. For example, when the following statement is executed:

```
100 INPUT A
```

the computer will display a question mark and wait for the operator to enter a response. That entry will be assigned to the variable A. The entry must be ended by pressing the Enter key. Program execution will then resume.

The values of several numeric variables can be input with a single INPUT statement as shown in the example below.

```
200 INPUT X, Y, Z
```

When the preceding INPUT statement is executed, the INPUT prompt (?) will be displayed. The operator should then enter the data items for X, Y, and Z. Each input should be separated by a comma. The Return 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.

```
100, 200, 300 [Ret]*
```

Caution should be used when inputting string data. Be certain that your string entries do not contain a comma unless enclosed in quotation marks. A comma will be interpreted by Applesoft BASIC as a delimiter. Any data appearing after the comma will be treated as a separate INPUT statement data item.

---

\* [Ret] indicates pressing the Return Key.

For example, in the following program;

```

100 INPUT A$, B$
200 PRINT A$, B$
RUN
? SMITH, JOHN, JONES, TED
?EXTRA IGNORED
SMITH          JOHN

```

A\$ would be assigned "SMITH", and B\$ would be assigned "JOHN". "JONES" and "TED" would be ignored as the error message (?EXTRA IGNORED) illustrates.

Therefore, when inputting a string item, be certain a comma does not appear within that string.

The variable type used with INPUT should be of the same type as the data input. String data cannot be input into numeric variables. If this does occur, the error message ?REENTER will appear, and the operator will be prompted for a new entry. If a real number is input for an integer variable, the decimal portion of the real number will be truncated. If numeric data is input for a string variable, that data will be interpreted as a string and cannot be used in calculations.

It is good programming practice to include a prompt message in conjunction with the INPUT statement to remind the operator what data the computer is expecting. The prompt should be enclosed within quotation marks after INPUT. The prompt should be followed by a semicolon\* and the variable or variables into which data is to be input. The prompt message will be displayed on the screen followed by the ? prompt. An example of an INPUT statement with a prompt is given as follows.

```

100 INPUT "CUSTOMER NAME "; A$
200 PRINT A$

```

---

\* In Integer BASIC, the prompt is followed by a comma.

**GET**

Applesoft's GET statement allows a single character to be input via the keyboard. That character is not displayed on the screen. Also, the return key need not be pressed after the entry has been made. The following program illustrates the usage of GET.

```
70 PRINT "IF YOU WISH TO STOP—ENTER Y"
90 PRINT "OTHERWISE, PRESS ANY KEY"
100 GET Z$
130 IF Z$ = "Y" GOTO 950
140 PRINT Z$
150 GOTO 70
950 PRINT "PROGRAM ENDS"
999 END
```

If a string variable is used with GET, the character entered will be treated as a string value. If a numeric variable is used with GET, the character entered must be a number. If a non-numeric character is entered, the SYNTAX ERROR message will appear and the program will end.

**FOR, NEXT Loops**

Suppose that you needed to compute the squares 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
.
.
.
.
.
```

However, this method is very cumbersome. This 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 100 to 400 is known as a **loop**. When the computer encounters the FOR statement in line 100, the variable A is set to 1. X is then 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 is incremented by 1 (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 incremented 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 incremented. 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 incremented by 2 every time the NEXT statement was executed.

### Nested Loops

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

```

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

```

Our preceding example is used to read data into the numeric array R.

One error that you should take care to avoid when using nested loops is to end an outer loop before an inner loop is ended.

### Conditional Statements

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 or line number*

The IF statement sets up a question or a condition. If the answer to that question is true, the *statement or line number* following THEN is executed. If the answer is false, all instructions following THEN are ignored, and program execution will resume with the next line number in the program.

In the following example, if 1 is input for X, the Y is set equal to 1. Otherwise, Y's value remains 0.



```

20 INPUT X
50 Y = 0
100 IF X = 1 THEN Y = 1
200 PRINT X: PRINT Y

```

## Branching Statements

Branching statements change the execution pattern of programs from their usual line by line execution in ascending line number order. 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 at line 500 to line 999.

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

```

100 INPUT "ENTER THE AMOUNT ";A
200 IF A = 0 THEN GOTO 900
300 PRINT A
400 GOTO 100
900 INPUT "FINISHED ";B$
910 IF B$ = "N" THEN 100
999 END

```

In our preceding example, if the value input for A has a zero value, then the program will branch to line 900 where the operator will be prompted whether he has finished entering data. In line 910, the program will set up a condition where if the input was 'N', the program will branch to line 100. If the entry was not equal to 'N', the program will continue to line 999 where it will end.

Note in line 910 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 does not require the presence of GOTO, which is assumed.

### **ON, GOTO Statement**

The ON, GOTO statement is a combination of a conditional statement and a branching statement. The use of the ON, GOTO statement is illustrated in the following program.

```
10 INPUT A
20 ON A GOTO 40, 50
30 GOTO 99
40 PRINT "A = 1": GOTO 99
50 PRINT "A = 2"
99 END
```

If the variable or expression following ON evaluates to 1, program control branches to the first line number specified after GOTO; if 2, to the second; if 3, to the third, 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.

### **Subroutines & GOSUB Statements**

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 con-

suming. 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.

#### Illustration 4-5. BASIC Program With a Subroutine

```

10 INPUT "PAY TO THE ORDER OF ";A$
20 INPUT "CHECK AMOUNT ";X
30 IF X = 0 THEN 200
40 IF X < 0 THEN GOSUB 100
50 IF X > 1000 THEN GOSUB 100
60 IF (X > 0) AND (X < 1000) THEN PRINT A$,X
70 GOTO 10
Subroutine { 100 PRINT "NOT VALID AMOUNT"
            { 110 INPUT "TRY AGAIN ";X
            { 120 RETURN
            { 200 END

```

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.

## ON, GOSUB Statement

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 is executed. If X is 2, the subroutine at line 2000 is executed. If X is 3, the subroutine at line 3000 is executed. If X evaluates to 0 or to a number greater than 3, the statement immediately following the ON, GOSUB statement 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.

## Applesoft BASIC Functions

**Functions** are used in Applesoft BASIC to perform predefined calculations or operations on their arguments. All functions use the following format.

*function (argument)*

*function* is the keyword for the function. *argument* is a variable, constant, or expression which is to be stored in the operation defined by the function.

The following statement is an example of the use of the SQR function.

```
100 A = SQR(49)
```

In this example, A would evaluate at 7. SQR is the keyword which describes the square root function. The square root of 49 is, of course, equal to 7.

Functions can be used with arithmetic, relational, and Boolean expressions, as shown in the following statement.

$$100 X = 100 - 7 * \text{SQR}(49)$$

In an expression containing functions as well as arithmetic, relational, and/or Boolean operators, the function's value is calculated first. In our preceding example, the square root of 49 would be calculated, that value would be multiplied by 7, and the product subtracted from 100.

Applesoft BASIC also includes a number of functions for performing operations on strings. These include:

LEFT\$  
MID\$  
RIGHT\$

These functions can be used to extract one or more characters from a string.

The various Applesoft BASIC functions are described in Chapter 4.

### String Concatenation

The addition operator (+) can be used to join together or concatenate two strings. When concatenating strings, remember that the maximum length of a string in Applesoft BASIC is 255 characters.

The following program illustrates string concatenation.

```
100 A$ = "JOHN"
200 B$ = "BILL"
300 C$ = A$ + B$
400 PRINT C$
500 END
RUN
JOHNBILL
```

The subtraction operator (-) cannot be used to separate a portion of a string.

## ASCII

The IIe cannot store characters; it can only store numbers. Before characters can be stored, they must be converted to numbers. Computers use special numeric codes to store characters. Most microcomputers use a code known as ASCII (American Standard Code for Information Interchange).

The codes used by the IIe are listed in Appendix G. These codes can be activated by including them with the CHR\$ function with a PRINT statement. For example, the following:

```
PRINT CHR$(56)
```

would cause the number 8 to be output on the display.

## CHR\$ & ASC Functions

As mentioned earlier, characters are represented with ASCII codes. Applesoft BASIC's CHR\$ function can be used to translate an ASCII code to its equivalent character. The following short program illustrates the use of the CHR\$ function.

```
100 PRINT CHR$(54)
200 PRINT CHR$(55)
300 END
RUN
6
7
```

The CHR\$ function is often used to represent characters in a statement, when that character can not be represented in its text form. For example, in the following program,

```
100 PRINT CHR$(34); "JOHN JOHNSON"; CHR$(34)
200 END
RUN
"JOHN JOHNSON"
```

quotation marks are specified in the PRINT statement using their ASCII code and the CHR\$ function.

The ASC function returns the ASCII code equivalent for its string argument. If this string is longer than one character, the ASC function returns the ASCII code for just the first character in the string.

The following program illustrates the use of the ASC function:

```
100 A$ = "JOHN JOHNSON"
200 PRINT ASC(A$)
300 END
RUN
74
```

## PEEK AND POKE

The PEEK and POKE statements allow direct access to the IIe's memory. The argument of PEEK and POKE indicates the address in memory to be accessed. Every memory location can store a number in the range 0 through 255.

The PEEK function allows the user to examine the value stored in the memory location named as its argument. For example, in the following statement.

```
100 N = PEEK (1000)
```

the value stored at memory location 1000 will be assigned to the variable N.

The POKE statement is used to place a value in a specified memory location. POKE uses the following configuration,

```
POKE address, value
```

where the *value* specified is placed in the location given in *address*. *value* and *address* can either be constants or variables. For example, in the following statement,

```
100 POKE 2000, X
```

the value stored in variable X will be POKE'd into memory location 2000.

## STOPPING PROGRAM EXECUTION

A number of different methods are available for stopping program execution on the IIe. These will be discussed in the following sections.

### Control-C

A program can be stopped by pressing the Control and C keys simultaneously. If Control-C is pressed in response to an INPUT statement prompt, the Return key must be pressed after Control-C to stop execution.

When program execution is stopped by pressing Ctrl-C, the following message will be displayed (in Applesoft BASIC):

BREAK IN *line number*

The *line number* will be the line number where the program execution was stopped. Program execution can be resumed by entering CONT.

In Integer BASIC, the following message will be displayed when program execution is stopped using Ctrl-C.

STOPPED AT *line number*

Again, *line number* indicates the line where program execution was stopped. Program execution can be resumed by typing in CON.

### END

The END statement can be used to stop program execution. Program execution can be resumed once END has been executed by entering CONT. In Integer BASIC, program execution cannot be resumed once END has been executed.



## **STOP**

When a **STOP** statement is executed in Applesoft **BASIC**, program execution is halted, and the following message is displayed.

**BREAK IN** *line number*

Program execution can be resumed by entering **CONT**.

## **RESET**

Pressing **CONTROL-RESET** will also stop execution of a program.

# CHAPTER 4.

## APPLE BASIC REFERENCE GUIDE

---

### INTRODUCTION

In this chapter, we will provide descriptions of the various commands, statements, and functions used in Applesoft and Integer BASIC.

The following rules and abbreviations will be followed in this chapter in our configuration descriptions of the various BASIC commands, statements, and functions.

1. Any capitalized words are keywords.
2. Any words, phrases, or letters shown in lowercase italics identify an entry that must be made by the operator (unless enclosed within brackets).
3. Any items enclosed in brackets [ ] are optional.
4. An ellipsis (...) shows that an item may be repeated as often as desired.
5. Any punctuation marks, except the square brackets (ex. ; , =) must be included where they are shown.

---

\* Except [Ret].

**ABS**

- Applesoft
  - Integer
- 

The ABS function returns the absolute value of the *argument*. A number's absolute value is its value without regard to sign.

**Configuration**

ABS(*argument*)

The *argument* can be any numeric expression or numeric constant. In Integer BASIC, the numeric constant must be an integer.

**Example**

```
10 A = ABS(-1 * 7)
20 PRINT A, ABS(2.99)
]RUN [Ret]
7          2.99
```

In the preceding example, the absolute values of -7 and 2.99 are returned.

**AND**

- Applesoft
  - Integer
- 

AND is a logical math operator. This reserved word is generally used to compare two numeric expressions in the context of an IF, THEN statement.

**Configuration**

*expression1* AND *expression2*

*expression1* and *expression2* are Boolean expressions. If an *expression* was numeric (not zero), that *expression* would evaluate as true. For example, if an expression evaluated to 5,

AND would treat it as true. The following is the truth table for AND.

X	Y	X AND Y
true	true	true
true	false	false
false	true	false
false	false	false

In both Applesoft and Integer BASIC, a true is represented by a 1 and false by a 0.

### Example 1

```

10 A = 2
20 B = 3
30 IF (A = 2) AND (B = 3) THEN 60
40 PRINT "AND FAILED LOGICAL TEST"
50 GOTO 70
60 PRINT "AND PASSED LOGICAL TEST"
70 END
]RUN [Ret]
AND PASSED LOGICAL TEST

```

In the preceding example, line 30 first tested the value of A. Since A was set equal to 2 in line 10, the first *expression* was evaluated as true. The value of B was then tested. It too evaluated as true. Using the logical AND table, if *expression1* and *expression2* evaluated to true, then the whole AND *expression* evaluated as true. The program will then execute the THEN portion of the statement and will branch to line 60. At line 60, the message AND PASSED LOGICAL TEST was displayed.

### Example 2

```

PRINT (3 = 1 + 2) AND (-5)
1

```

In this example, 3 is set equal to 1 + 2, so the first *expression*

evaluates as true. The second *expression* (-5) is non-zero, so it is also evaluated as true. According to the AND truth table, if both *expressions* evaluate as true, then the whole *expression* is true. Applesoft and Integer BASIC represent true as 1, so a 1 is printed.

## ASC

- Applesoft
- Integer

The ASC function returns the ASCII code for the first character in its *argument*.

### Configuration

ASC(*argument*)

*argument* can be any string variable or constant.

### Example

```
]A$ = "A"
```

```
]PRINT ASC(A$), ASC("DEF")
65                68
```

In the preceding example, the character in the string A\$ was A. A's ASCII equivalent is 65. In the second string, the first character D will be used as the argument. A value of 68 is returned for the ASCII value of D.

## ATN

- Applesoft
- Integer

The ATN function is a trigonometric function that returns the arctangent of its *argument*.

### Configuration

ATN(*argument*)

The *argument* can be a numeric expression or numeric constant in radians. The value returned will be the primary angle

$$(-\frac{\pi}{2} < \text{angle} < \frac{\pi}{2})$$

### Example

```

10 PI = ATN(1) * 4
20 PRINT PI, ATN(TAN(.2))
]RUN [Ret]
3.14159266                               .2

```

In the preceding example, the arctangent of 1 returns the value  $\pi/4$ . Multiplying this value by 4 returns the value indicated.

In the second part of the PRINT statement, the argument .2 is returned. Since the ATN formula is the inverse of the TAN function, the value returned was the original argument.

## AUTO

- Applesoft
- Integer

The AUTO command generates a new line number every time the user presses Return.

### Configuration

AUTO *line number* [,*increment*]

Both *linenumber* and *increment* must be integers. The *linenumber* will be the first line number generated. The *increment* is the amount to be added to the current line number to generate the next line number. If the *increment* is not included, the *increment* will be set to 10 by default.

The AUTO command is generally used when entering programs. This saves the user the task of typing every line number.

The AUTO command is ended by pressing Ctrl-X and typing MAN.

### Example

```
AUTO 30  
AUTO 10, 5
```

The first command will generate the line numbers 30, 40, 50, and so on. The second command will generate the line numbers 10, 15, 20, 25, 30, ...

AUTO is not available in Applesoft BASIC.

## **CALL**

---

- Applesoft
- Integer

The CALL statement is used to execute a machine language subroutine.

### Configuration

CALL *expression*

*expression* evaluates to an integer between -65535 and +65535. In Integer BASIC, the value of the *expression* must be an integer between -32767 and +32767.

The *expression* is the location of the machine language subroutine.

In Applesoft, there are two values which will execute the same machine language subroutine. There is the positive address and the negative address. The conversion is as follows:

$$\text{positive address} - 65536 = \text{negative address}$$

This can be very useful in Integer BASIC. If there was a machine language subroutine located at location 64578, a CALL 64578 could not be used. This is due to the fact that the *expression* indicated is greater than the largest integer that can be used in Integer BASIC. Using the previous conversion equation, the CALL used would be CALL-958.

### Example

CALL-936

The preceding CALL executes a machine language subroutine at the given location. This CALL is identical to the HOME command in Applesoft.

## **CHR\$**

---

- Applesoft
- Integer

The CHR\$ function returns the ASCII character for the value given in the *argument*.

### Configuration

CHR\$(*argument*)

*argument* is a real number or an integer between 0 and 255. If the *argument* is a real number, its decimal portion will be truncated.

### Example

```
10 X$ = CHR$(80)
20 PRINT CHR$(65), X$
]RUN [Ret]
A                               P
```

The ASCII code for A is 65 and the code for P is 80.



## **CLEAR**

---

- Applesoft
- Integer

CLEAR initializes all variables, arrays, and strings to zero. CLEAR also initializes all DATA pointers, FOR...NEXT counters, subroutine pointers, etc.

### **Configuration**

CLEAR

CLEAR can be used anywhere in a program, but should not be used in a subroutine or FOR...NEXT loop.

### **Example**

```

10 A = 10
20 PRINT A
30 CLEAR
40 PRINT A
]RUN [Ret]
10
0

```

Line 30 sets variable A from 10 to 0.

## **CLR**

---

- Applesoft
- Integer

CLR sets all variables to 0, strings to null, and clears any dimensioned variables.

### **Configuration**

CLR

CLR can only be used in the immediate mode.

**Example**

```

A = 10
B$ = "P"
CLR
PRINT A, B$
0

```

The variable A is cleared to 0 and the string variable B\$ is set to null.

**COLOR**

- Applesoft
- Integer

The COLOR statement defines the next color to be displayed by the graphics statements PLOT, HLIN...AT, and VLIN...AT.

**Configuration**

COLOR = *expression*

The *expression* is an integer from 0 to 255. The computer can display a total of 16 different colors. The colors and their associated numbers are shown below.

0 Black	8 Brown
1 Magenta	9 Orange
2 Dark Blue	10 Grey
3 Purple	11 Pink
4 Dark Green	12 Green
5 Grey	13 Yellow
6 Medium Blue	14 Aqua
7 Light Blue	15 White

Beyond 15, the colors repeat (16 - Black, 17 - Magenta).

**Example**

```
10 GR
20 COLOR = 6
30 PLOT 0,0
40 END
```

The preceding program will place a blue square in the upper left hand corner of the screen.

**CON**

- Applesoft
- Integer

The CON statement resumes program execution at the next instruction.

**Configuration**

CON

This command is generally executed following a Ctrl-C.

**Example**

```
10 FOR X = 1 TO 10
20 PRINT X, X^2, X^3
30 NEXT X
40 END
> RUN [Ret]
1      1      1
2      4      8
4      16     64
5      25     125
Ctrl-C pressed → STOPPED AT 30
> CON [Ret]
6      36     216
7      49     343
8      64     512
9      81     729
10     100    1000
```

In the preceding example, a Ctrl-C was entered in as shown. The Ctrl-C stopped the execution of the program. Entering in CON continued program execution.

## CONT

- Applesoft
- Integer

CONT resumes program execution at the next instruction.

### Configuration

#### CONT

This command is generally executed following a STOP, END, or Ctrl-C.

#### Example

```
10 FOR I = 1 TO 5
20 PRINT I, I^2
30 IF I = 3 THEN STOP
40 NEXT I
]RUN [Ret]
```

```
1          1
2          4
3          9
```

```
BREAK IN 30
]CONT [Ret]
```

```
4          16
5          25
```

In the preceding example, program execution stopped in line 30 when I = 3. Typing in CONT continued program execution.

**COS**

- Applesoft
  - Integer
- 

The COS function is a trigonometric function that returns the cosine of its *argument*.

**Configuration**

COS(*argument*)

The *argument* is a numeric expression or numeric constant in radians.

**Example**

```
JPRINT COS(3.141592653)
-1
```

In the preceding example, the cosine of PI is returned.

**DATA**

- Applesoft
  - Integer
- 

The DATA statement contains a list of data items. These data items are read into the variables specified by the READ statement.

**Configuration**

DATA *item* [,*item*...]

*item* can either be a real number, integer, or string. The data items must be in the same order as they are read by the READ statement.

If a comma or colon is to be included in the string, the item should be enclosed in quotes. The following characters cannot

be placed in a DATA statement.

RETURN	Ctrl-H
ESC	Ctrl-M
”	Ctrl-U
(left arrow)	Ctrl-X
(right arrow)	

The preceding characters may be used in a program by executing the CHR\$ function.

The DATA statement can be located anywhere in a program. It does not have to precede the READ statement.

### Example

```

10 DATA "SMITH, JOE", JOHN BROWN
20 READ N1$, N2$
30 PRINT N1$, N2$
]RUN [Ret]
SMITH, JOE      JOHN BROWN

```

The READ interpreted the first string as SMITH, JOE because it was enclosed in quotes. The second string is read as JOHN BROWN.

## DEF FN

- Applesoft
- Integer

The DEF FN statement allows the user to define a function. This function can then be used in the same manner as any built-in function.

### Configuration

DEF FN *name (variable)* = *expression*

*name* is the name of the function. Like variable names, only the first two characters are significant. The *variable* can be any real numeric variable name. The *expression* can be a numeric constant or a numeric equation.

### Example

```

10 PI = ATN(1) *4:REM PI
20 DEF FNAR(X) = PI * X^2
30 FOR RAD = 1 TO 3
40 PRINT RAD, FNAR(RAD)
50 NEXT RAD
60 END
]RUN [Ret]
1          3.14159266
2          12.5663706
3          28.2743339

```

In line 10, PI is calculated so it can be used in the function definition. In line 20, the function for the area of a circle is defined. Line 40 then uses the function by passing the value of the radius to the function. The value of the area of the circle is then returned and printed by the PRINT statement.

## DEL

- Applesoft
- Integer

DEL deletes the lines given in the argument.

### Configuration

DEL a [,b]\*

*a* and *b* are integers greater than or equal to 0. In Applesoft, *b* must be greater than *a*. In Integer BASIC, if *b* is less than *a* only

---

\* The [,b] is only optional in Integer BASIC.

line *a* will be deleted.

If *a* is not an existing line number in the program, the next highest line number will be used. If *b* is not an existing line number in the program, the next lowest line number will be used.

The DEL can also be used as a program statement in Applesoft. If the DEL is used as a program statement, the specified lines will be deleted. However, program execution will halt after the statement has been executed. The CONT command will not resume program execution.

### Example

```

]LIST [Ret]
10 TEXT
20 CALL -936:REM HOME
30 VTAB 3
40 PRINT "HELLO"
50 END
]DEL 30,50 [Ret]
]LIST [Ret]
10 TEXT
20 CALL -936:REM HOME

```

## DIM

- Applesoft
- Integer

The DIM statement is used to allocate memory space for strings, arrays, or matrices.

### Configuration

Applesoft	$a (i[j]...) [,b (i[j]...]$
DIM	$a\% (i[j]...) [,b\% (i[j]...]$
	$a\$ (a\$ (i[j]...)) [,b\$ (i[j]...]$



Integer	$a(i[,b(i)...])$
DIM	$a$(i[,b$(i)...])$

$a$  and  $b$  are the variables to be dimensioned.  $i$  and  $j$  are integers. In Integer BASIC, the string variable must be dimensioned with 255 elements or less.

In Applesoft BASIC, all arrays, strings, and matrices are predefined with subscripts of 10. Above 10, the value in a DIM statement corresponds to the largest subscript that can be used in that variable. However, there is always a zero subscript. As a result, to save 100 values in a single dimension array, the correct DIM statement would be DIM A(99).

In Integer BASIC, the value in a DIM statement corresponds to the largest subscript that can be used with that variable. An Integer variable with a subscript of 0 is the same as a variable without the subscript. If TEST(0) = 27, then TEST is also equal to 27 and vice versa. With string variables, there is no 0 subscript.

The maximum size of strings and arrays depends on the amount of available memory at the time the DIM statement was executed.

If the DIM statement exceeds the amount of available memory, the following error will occur:

Applesoft BASIC     ?OUT OF MEMORY ERROR IN *line*

Integer BASIC     \*\*\*MEM FULL ERR  
STOPPED AT *line*

where *line* is the line of the DIM statement.

#### Example -- Applesoft

```
DIM A$(10,5), C%(2,20)
```

In the preceding example, 66 string spaces are allocated for A\$, and 63 integer variables are defined for C%.

**Example -- Integer BASIC**

```
DIM A(10), N$(5)
```

In the previous example, the DIM statement defines 10 spaces for the variable A, and N\$(5) defines one string of length 5.

**DRAW**

- Applesoft
- Integer

The DRAW statement plots a shape on the high-resolution graphics page.

**Configuration**

```
DRAW shapeno [AT X, Y]
```

*shapeno* is an integer between 0 and 255. X and Y are integers for the position of the shape. X must lie between 0 and 279. Y must lie between 0 and 191.

If AT X, Y is not given, the shape will be plotted at the last X, Y position designated.

The color, rotation, and size of the shape must have been previously defined.

**Example**

```
10 REM SET UP SCREEN
20 TEXT:HGR
30 FOR X = 7936 TO 7946
40 READ V:REM READ IN SHAPE
50 POKE X,V:REM POKE SHAPE INTO TABLE
60 NEXT X
70 REM TELL WHERE SHAPE IS AT
80 POKE 232,0:POKE 233,31
90 HCOLOR = 3
```

*program continued on the next page*

```

100 ROT = 0
110 SCALE = 10
120 DRAW 1 AT 50,30
130 H PLOT 20,70
140 SCALE = 6
150 ROT = 4
160 DRAW 1
170 DATA 1,0,4,0,39,36,45,53,54,63,0
180 END

```

In the preceding example, line 20 initialized the screen for high resolution graphics. The shape table is then read in the FOR NEXT loop in lines 30 to 60. These lines define the shape. Since Applesoft needs to know where the shape table was placed, the shape address is poked into memory in line 80. Lines 90, 100, and 110 then define the color, rotation, and size of the shape drawn at line 120. The shape drawn in line 160 will have the position defined at line 130, size in 140 and the rotation in 150.

## DSP

- Applesoft
- Integer

The DSP command is a debugging tool. It displays the variable's value each time its value is changed.

### Configuration

DSP *variable*

*variable* is a variable in the program to be traced.

The DSP command will display the *variable*, the variable's value, and line number each time its value is changed.

**Example**

```

10 DSP A
20 DSP B
30 A = 1
40 B = 7
50 C = A*B
60 B = C+A
70 END
]RUN [Ret]
#30 A = 1
#40 B = 7
#60 B = 8

```

In the preceding example, the line number, variable, and variable values are displayed when the indicated variable's values change. Notice that in line 50 the value of A and B is used, but not changed, so it is not displayed.

**END**

- Applesoft
- Integer

The END statement is used to stop program execution.

**Configuration**

END

The END statement is optional in Applesoft. If it is not used, the program will stop execution at the highest line number.

The END statement is optional in Integer BASIC, but if it is not included a NO END error will result. The program will essentially run the same with or without the END statement.

**Example**

```
999 END
```

**EXP**

- Applesoft
  - Integer
- 

The EXP function returns the value of e raised to the power of the *argument*.

**Configuration**

EXP (*argument*)

*argument* is a numeric constant or numeric expression. (e = 2.71828183).

**Example**

```
PRINT EXP(5)
148.413159
```

In the preceding example,  $e^5$  was returned.

**FLASH**

- Applesoft
  - Integer
- 

The FLASH statement turns on the FLASH video. Following the execution of the FLASH statement, any characters displayed by the computer will flash. The characters will alternate from black on white to white on black.

**Configuration**

FLASH

Any characters echoed by the computer (entered through the keyboard), will not flash.

The FLASH mode works by altering the standard ASCII code. So, any characters sent to the disk or printer while the FLASH mode

is on, may be sent with the incorrect codes.

The FLASH statement is equivalent to a POKE 50,127. The FLASH statement is turned off by the NORMAL statement.

### Example

```
] FLASH
] PRINT "***"
**
```

When the IIe's 80-column card is activated (either in the 40 or 80 column mode), the alternative character set will be active. The alternate does not include flashing characters. Therefore, FLASH does not function properly when the 80 column card is active.

## FOR..NEXT

- Applesoft
- Integer

The FOR..NEXT statements are used to execute a sequence of statements a set number of times.

### Configuration

```
FOR variable = a to b [STEP c]
.
.
.
NEXT [variable*] [,variable ...]
```

*variable* is a real variable in Applesoft. The *variable* is used as a counter. *a*, *b* and *c* are numeric expressions or constants. The numeric constant must be an integer in Integer BASIC. *a* is the initial value of the counter and *b* is the final value. The counter is incremented or decremented depending on the sign of *c*. If *c* is not given, it will be assumed as 1.

---

\* The variable in NEXT is only optional in Applesoft.

The program lines following the FOR statement will be executed until the NEXT statement is encountered. At this point, the counter is incremented (assuming positive STEP value) by the STEP value.

The value for the counter is then compared with its final value *b*. As long as the counter's value does not exceed the final value, the program will branch back to the statement following the FOR statement. This entire process will then be repeated.

When the counter's value exceeds the specified final value (*b*), the statement following the NEXT statement will be executed. This will exit the FOR..NEXT loop.

One FOR..NEXT loop may be placed within another FOR..NEXT loop. This is known as **nesting** or **nested loops**. When FOR..NEXT loops are nested, each FOR..NEXT loop must use a different variable name for the counter. Also, the NEXT statement for the inside loop must appear before the NEXT statement for the outside loop. However, if both loops end at the same point, a single NEXT statement may be used to end these. Be certain that the variable for the inside loop appears before the variable for the outside loop. A NEXT statement such as the following:

```
NEXT J,I
```

would be interpreted as follows:

```
NEXT J
NEXT I
```

### Example

```
10 FOR X = 1 TO -2 STEP -1
20 PRINT X
30 NEXT X
40 END
]RUN [Ret]
1
0
-1
-2
```

In the preceding example, the STEP value is -1 so the counter is decremented until its value is -2.

## **FRE**

---

- Applesoft
- Integer

The FRE function returns the number of free bytes in memory.

### **Configuration**

FRE (*argument*)

*argument* can be any legal expression. It makes no difference what the *argument* is.

If the amount of free bytes exceeds 32767, the FRE function will return a negative number. By adding 65536 to this number, you can compute the actual number of free bytes.

When FRE is used, a housekeeping will be performed before the function returns the number of free bytes. Housekeeping is a process where BASIC gathers all useful data by freeing any memory which was once used for strings, but which is currently unused. Memory for strings becomes unused when the string's length changes.

### **Example**

```

10 A = FRE (0)
20 IF A < 0 THEN A = A+65536
30 PRINT "NUMBER OF FREE BYTES IS";A
40 END
]RUN [Ret]
NUMBER OF FREE BYTES IS 36272

```



## GET

---

- Applesoft
- Integer

The GET statement inputs a single character from the keyboard. The character is not displayed on the screen.

### Configuration

GET *variable*

*variable* can be any legal Applesoft variable.

Although *variable* can be any variable, it is to the user's advantage to use a string variable and convert it to a numeric variable with the VAL function. If a numeric variable was used with GET, any non-numeric character entered will cause a syntax error and halt program execution.

### Example

```
10 PRINT "PRESS A KEY";
20 GET A$
30 PRINT
40 PRINT "THE KEY PRESSED WAS";A$
50 PRINT
60 PRINT "IF THE KEY PRESSED WAS NOT A DISPLAYABLE
    CHARACTER IT WOULD NOT BE DISPLAYED"
70 END
```

In the preceding example, line 10 prompts the user to press a key. Line 20 waits for a key to be pressed. When a key is pressed, the character value of the key is assigned to the variable A\$. Line 40 then displays the character input. Line 60 is included because some keys generate characters that cannot be displayed (i.e. return key, ESC key and the space bar).

## **GOSUB, RETURN**

---

- Applesoft
- Integer

The GOSUB, RETURN statements are used to branch to a subroutine and then return from it.

### **Configuration**

GOSUB *line*

.

.

.

RETURN

*line* is the first line of a subroutine. In Applesoft, the GOSUB will branch to line 0 if *line* is omitted or an expression is used. In Integer BASIC, *line* can be an expression, where the expression evaluates to a line number of a subroutine.

A subroutine is called by the GOSUB statement. When the RETURN statement is encountered within that subroutine, program control will branch back to the statement following the GOSUB statement just executed.

Subroutines may appear at any point within the program. However, it is good programming practice to group all subroutines near the beginning of the program.

### **Example**

```

10 GOTO 60
20 PRINT X,
30 Y = X*X
40 PRINT Y
50 RETURN
60 X = 0
70 FOR I = 1 TO 3
80 X = X+1
90 GOSUB 20

```

*program continued on next page*

```

100 NEXT I
110 END
]RUN [Ret]
1      1
2      4
3      9

```

In the preceding example, line 10 jumps over the subroutine to the main program body. When the program reaches line 90, the GOSUB is executed. The program branches to line 20. When the RETURN in line 50 is reached, program execution jumps back to line 100. This process continues until the FOR counter reaches 3.

## **GOTO**

- Applesoft
- Integer

The GOTO statement branches program control to another program line.

### **Configuration**

GOTO *line*

*line* is the line number of the statement to be branched to.

### **Example**

```

10 PRINT "FIRST"
20 GOTO 40
30 PRINT "MIDDLE"
40 PRINT "LAST"
50 END

```

**GR**

- Applesoft
  - Integer
- 

The GR statement sets and clears the low resolution screen mode (40x40 with 4 lines of text at the bottom of the screen).

**Configuration****GR**

This statement should be executed before the graphics statements PLOT, HLIN.. AT, and VLIN..AT are used.

When the GR statement is executed, the color is automatically set to 0 (BLACK).

**Example**

```
10 GR
20 COLOR = 15
30 PLOT 19, 23
40 END
```

The preceding example should put a white square on the screen. If line 10 was omitted, a zero would be placed in the middle of the screen, because the low-resolution mode was not set.

If you wish to return to the normal mode, you can do so by executing the TEXT statement.

If full-screen graphics (40x48) is desired, this can be accomplished by executing POKE -16302,0 after GR has been executed. POKE -16301,0 can be used to restore the text window.

**HCOLOR**

- Applesoft
  - Integer
- 

The HCOLOR statement defines the next color to be displayed

by the graphics statements, H PLOT, DRAW, and XDRAW. HCOLOR is used in the high resolution graphics mode.

### Configuration

HCOLOR = *number*

*number* is a numeric expression or numeric constant that evaluates to a real number or integer between 0 and 7. Values outside this range will produce an error. The colors and their associated numbers are shown below.

0 - Black 1	4 - Black 2
1 - Green*	5 - Orange (Red)*
2 - Violet*	6 - Blue*
3 - White 1	7 - White 2

\*The actual color depends on the CRT.

H PLOT, DRAW, and XDRAW will all output lines in the color indicated by HCOLOR until a subsequent HCOLOR statement is executed.

## HGR

- Applesoft
- Integer

---

The HGR statement sets and clears the high-resolution graphics mode (280x160), with 4 lines of text at the bottom of the screen.

### Configuration

HGR

The HGR statement displays page one of the high-resolution screen, leaving 4 lines of text at the bottom. If full screen graphics (280x192) is preferred, the statement POKE -16302,0 will set the rest of the screen to graphics. A POKE -16301,0 restores the 4 lines of text.

**Example**

]HGR

When the preceding example is executed, the high-resolution graphics mode will be set and the screen will be cleared to black. There will also be four lines of text at the bottom. If the cursor is not visible, press the return key until it appears. The cursor may not be visible if it is located in the graphics area of the screen rather than the text area.

**HGR2**

- Applesoft
- Integer

HGR2 sets the screen to the high resolution graphics mode without the text lines at the bottom of the display (280x192). Page 2 of high-resolution screen memory is displayed by HGR2.

**Configuration**

HGR2

In the 80-column mode, HGR2 displays page 1 of screen memory rather than page 2. This can be illustrated by running the following example in both the 40 and 80 column modes. It appears that this is due to the fact that the softswitch that activates screen two does not work in the 80-column mode.

**Example**

```

10 HGR
20 HCOLOR = 2
30 H PLOT 100,100 TO 150,180
35 FOR I = 1 TO 1000: NEXT I
40 TEXT
50 INPUT "PRESS RETURN TO SEE PAGE 2";A$
60 HGR2
70 H PLOT 0,0 TO 100,0

```

## HIMEM

---

- Applesoft
- Integer

The HIMEM statement defines the address of the highest memory location available to a BASIC program.

### Configuration

HIMEM: *number*

*number* is a numeric constant or numeric expression. The value of *number* should indicate the highest available memory address. This value must lie between -65535 to 65535 (-32767 to 32767 in Integer BASIC).

The current value of HIMEM can be displayed by entering:

```
PRINT PEEK (116) *256 + PEEK (115) for Applesoft  
PRINT PEEK (77) *256 + PEEK (76) for Integer BASIC
```

If the HIMEM: is set lower than LOMEM or set so low that there is not enough room for the program to run, an out of memory error will occur.

HIMEM can only be used in the immediate mode in Integer BASIC.

The value of HIMEM is not changed by the commands NEW, CLEAR, RUN, and DEL.

The HIMEM statement is generally used to reserve memory for a machine language subroutine called by the BASIC program. The HIMEM statement keeps BASIC variable and array storage separate from the machine language subroutine.

**Example**

HIMEM: 33024

The preceding example sets high memory to memory address 33024. Variable and string storage will begin at this address and extend downward into memory.

**HLIN**

- Applesoft
- Integer

HLIN is used in the low resolution graphics mode to draw a horizontal line on the screen.

**Configuration**

HLIN *column 1, column 2 AT row*

*column 1, column 2, and row* can be either numeric constants or numeric expressions. *column 1* and *column 2* must lie in the range of 0 to 39. Also, the value of *column 1* must be less than or equal to *column 2*. *Row* must lie in the range of 0 to 47.

If an incorrect value is used for *column 1, column 2, or row*, the following error message will be displayed:

ILLEGAL QUANTITY ERROR

If HLIN is executed in the text mode, a line of characters rather than graphics points will be displayed. Also, if the low resolution graphics mode with the text mode is active, and HLIN plots to rows 42 to 47, a line of characters will be displayed.

**Example**

```
10 GR
20 COLOR = 3
30 HLIN 0, 39 AT 20
40 END
```



The preceding example will draw a purple line across the screen at row 20.

## **HOME**

---

- Applesoft
- Integer

The HOME statement clears the screen and places the cursor in the upper left hand corner of the screen.

### **Configuration**

HOME

The HOME command is not available in Integer BASIC. The cursor can be HOME'd in Integer BASIC by executing CALL-936.

### **Example**

HOME

## **HLOT**

---

- Applesoft
- Integer

The HLOT statement can be used to place a dot or draw a line on the high resolution graphics screen. The color of the dot must have been previously defined by the HCOLOR statement.

### **Configuration**

HLOT *column 1, row 1* [TO *column 2, row 2 ...*]  
HLOT TO *column, row*

*column, row, column 1, column 2, row 1, and row 2* are numeric constants or numeric expressions. *column, column 1, and column 2* must lie between 0 and 279. The *row, row 1, and row 2* must lie between 0 and 191.

If the HPLOT is used as shown in the first configuration without the optional (TO *column 2*, row 2), a dot will be plotted. The optional TO will connect the two dots. If the *column 1* and *row 1* preceding the TO are omitted, the line will be drawn from the previous point plotted to the point indicated by *column 2*, *row 2*.

### Example

```
10 HGR
20 HCOLOR = 3
30 HPLOT 0,0
40 HPLOT TO 0,50 TO 50,50
50 HPLOT TO 50,0 TO 0,0
```

The preceding example will draw a square in the upper left hand corner of the screen.

## HTAB

- Applesoft
- Integer

The HTAB statement positions the cursor at the location specified by its *argument*.

### Configuration

HTAB *argument*

*argument* is a numeric constant or numeric expression. The *argument* must be between 1 and 80.

The cursor will be moved to the position specified by the *argument*. HTAB moves the cursor without erasing any displayed characters.

**Example**

```

10 PRINT "1234567890"
20 HTAB 3 : PRINT 3;
30 HTAB 9 : PRINT 9;
40 HTAB 5 : PRINT 5
]RUN [Ret]
1234567890
  3 5  9

```

In the preceding example, line 20 places the cursor at position 3 and displays a 3. In line 30, the cursor is moved to position 9. The PRINT statement displays a 9. In line 40, HTAB moves the cursor back to position 5, and the PRINT statement displays a 5.

**IF..THEN**

- Applesoft
- Integer

The IF..THEN statement sets up a condition which will influence the program flow.

**Configuration**

IF *expression* THEN *statement* [:*statement...*]

*expression* is a conditional expression. *statement* can be any BASIC statement.

If the *expression* is evaluated as true, the THEN portion of the statement will be executed.

In Applesoft BASIC, if the *expression* evaluates as true, the *statement* following THEN will be executed. If the *expression* evaluates as false, the statement in the next program line will be executed.

In Integer BASIC, if the *expression* evaluates as true, the *statement(s)* following THEN will be executed. If the *expression*

evaluates as false, the *statement* immediately following THEN will not be executed. Program control will branch to the next *statement* even if that *statement* is on the same program line as the IF, THEN statement.

For example, if the following statement evaluated as true,

```
IF X=15 THEN PRINT "TRUE":PRINT X
```

the following would be displayed:

```
TRUE
15
```

If this statement evaluated as false, TRUE would not be displayed. However, the value for X would be displayed.

### Example

```
Applesoft  100 IF X > 8 THEN X = 0
           110 Y = Y+1
```

```
Integer    100 IF X > 8 THEN X = 0 : Y = Y+1
```

## IN#

- Applesoft
- Integer

---

IN# specifies the peripheral slot which will be providing subsequent input for the IIe.

### Configuration

IN# *argument*

*argument* is a numeric constant or numeric expression which specifies the peripheral slot. The numeric constant must be an integer for Integer BASIC. The value of the *argument* must be between 1 and 7.

If there is no peripheral in the specified slot, the system will hang. Press the Reset key to exit this situation.

### Example

IN#2

## INT

- Applesoft
  - Integer
- 

The INT function returns the integer value of the specified *argument*.

### Configuration

INT (*argument*)

*argument* is a numeric constant or numeric expression.

The value returned will always be less than or equal to the original value.

### Example

```
PRINT INT (1.7), INT (-1.7)
1          -2
```

In the above example, 1.7 is returned as a 1 and -1.7 is returned as -2.

## INVERSE

- Applesoft
  - Integer
- 

The INVERSE statement turns on the INVERSE (reverse) video. Following the execution of the INVERSE statement, any characters displayed by the computer will be in inverse (i.e. characters will be displayed as black characters on a white background.)

## Configuration

### INVERSE

The INVERSE mode works by altering the standard ASCII code. Therefore, any characters sent to the disk or printer while the INVERSE mode is on, may be sent with the incorrect codes.

The INVERSE statement is equivalent to POKE 50, 127. The INVERSE statement can be turned off by the NORMAL statement.

### Example

```
] INVERSE
] PRINT "++"
++
```

## INPUT

- Applesoft
- Integer

The INPUT statement accepts data entry from the keyboard or another input device while the program is being executed.

### Configuration

Applesoft	INPUT ["message";] <i>variable</i> [, <i>variable</i> ]
Integer	INPUT ["message",] <i>variable</i> [, <i>variable</i> ]

*message* is a string used as a prompt. *variable* can be any valid BASIC variable.

When an INPUT statement is executed, program execution will stop temporarily. If a prompt was included, the prompt will be displayed. In Applesoft, a question mark will be displayed if there is no prompt. In Integer BASIC, a question mark will follow the prompt if the *variable* is an integer variable.

After the INPUT statement has been executed, the user may enter the desired data at the keyboard. That data is assigned to the *variable(s)* listed in the INPUT statement. The number of data items entered must equal the number of *variables* listed. Also, the type of data entered must agree with the type specified in *variable*. The data items must be delimited by commas when input.

### Example

```
Applesoft  10 INPUT "ENTER A NUMBER"; A
           20 PRINT "THE NUMBER IS"; A
           30 END
           ]RUN [Ret]
           ENTER A NUMBER 4.5
           THE NUMBER is 4.5
```

```
Integer   10 INPUT "ENTER A NUMBER", A
           20 PRINT "THE NUMBER IS"; A
           30 END
           > RUN [Ret]
           ENTER A NUMBER? 4
           THE NUMBER IS 4
```

## LEFT\$

- Applesoft
- Integer

The LEFT\$ function returns the number of characters specified in the second expression of the argument to the leftmost of the string specified in the first part of the argument.

### Configuration

LEFT\$ (a\$,x)

a\$ is a string constant searched by the function. x is the number of characters to be returned.

Integer BASIC can duplicate this function by using string arrays.

### Example

```
10 A$ = "ABCDEFGG"
20 PRINT LEFT$(A$, 3)
30 END
]RUN [Ret]
ABC
```

The preceding LEFT\$ function returned the 3 leftmost characters in A\$.

In Integer BASIC, the statement PRINT A\$(1,3) would return the same characters. (Be sure that the variable A\$ has previously been dimensioned to 7 in Integer BASIC.)

## LEN

- Applesoft
- Integer

The LEN function returns the number of characters in a string.

### Configuration

LEN (a\$)

a\$ is a string constant.

### Example

```
PRINT LEN ("APPLE")
5
```



**LET**

- Applesoft
  - Integer
- 

The LET statement is an optional assignment statement. An assignment statement determines the value of an expression and then assigns that result to the variable named in the assignment statement.

**Configuration**

LET *variable* = *expression*

*variable* must be of the same data type as the expression. For example, if *variable* is a string, *expression* must also be a string. If *variable* is an integer or real number, then *expression* must also be numeric.

**Example**

```
10 LET C = 1+A
20 L = C*2
```

**LIST**

- Applesoft
  - Integer
- 

The LIST command is used to list the program stored in memory on the video display or other device.

**Configuration**

Applesoft	LIST a [{" <i>a</i> "}] <i>b</i> ]
	or
	LIST [{" <i>a</i> "}] <i>b</i> [{" <i>a</i> "}]
Integer	LIST a [, <i>b</i> ]

*a* and *b* are integers greater than or equal to 0. In Applesoft BASIC, if *a* is greater than *b*, no lines will be listed. In Integer BASIC, if *b* is less than *a* only line *a* will be listed.

If *a* is not a line number in the program, the next highest line number will be used. If *b* is not a line number in the program, the next lowest line number will be used.

In Applesoft, a LIST 0 statement will list all the lines in a program. For example, LIST 100,0 would list all the lines from line 100 to the end of the program.

In Applesoft, LIST can be frozen by Ctrl-S. Pressing any other key will resume LIST. The listing may be stopped by pressing Ctrl-C.

### Example

```
LIST 10 [Ret]
10 GR
LIST 10,50 [Ret]
10 GR
20 POKE -16302,0
30 COLOR = 3
40 GOSUB 5000
50 COLOR = 7
```

- Applesoft
- Integer

## LOAD

---

The LOAD command is used to load a program from a storage device to the computer.

### Configuration

```
Cassette LOAD
*Disk LOAD filename [,D drive][,V volume][,S slot]
```

*filename* is the name of the program. *drive* is the drive that the file is in and *volume* is the volume number of the diskette. *slot* is the slot the disk interface card is in.

---

\* LOAD is only interpreted as a BASIC reserved word when used with the cassette unit.

When using the cassette LOAD, first make sure that the current language is active (Applesoft or Integer BASIC). Position the tape to the beginning of the program, type LOAD, and press return. The cursor will disappear and after a few seconds the Apple will beep. The beep indicates that LOAD has started. When the second beep sounds, the LOAD will be finished. If an error occurred, turn off the computer, turn it on, and try again.

In the DOS LOAD, LOAD need only be entered with the program name, and the return key pressed. If the indicated file name is not present on the specified diskette, the FILE NOT FOUND error will occur.

### Example

```
LOAD
```

## LOG

---

- Applesoft
- Integer

The LOG function returns the natural log of the argument.

### Configuration

LOG (*argument*)

*argument* is a numeric constant or numeric expression greater than 0.

The natural log is undefined for negative numbers.

### Example

```
PRINT LOG (25)
3.21887583
```

**LOMEM:**

- Applesoft
  - Integer
- 

The LOMEM statement defines the address of the lowest memory location available for BASIC.

**Configuration**

LOMEM: *number*

*number* is a numeric constant or numeric expression. The value of *number* should be the lowest available memory address. This value must lie between -65535 to 65535 (-32767 to 32767 in Integer BASIC).

The current value of LOMEM can be displayed by entering PRINT PEEK (106) \* 256 + PEEK (105).

If LOMEM is set higher than the HIMEM an error will occur. LOMEM cannot be set lower than 2048.

LOMEM can only be used in the immediate mode in Integer BASIC. It cannot be used within a program.

LOMEM cannot be set lower than its current value. LOMEM can only be increased.

LOMEM will be reset by the NEW or DEL commands or by adding or changing a line.

**Example**

```
10 F = PEEK(106) * 256 + PEEK(105)
20 PRINT "LOMEM IS";F
30 LOMEM:3000
40 F = PEEK(106) * 256 + PEEK(105)
50 PRINT "LOMEM IS NOW";F
60 END
```

In the preceding example, LOMEM is calculated on line 10 and then displayed on line 20. LOMEM is then set to 3000 in line 30. Line 0 then recalculates the LOMEM value. The LOMEM setting is finally displayed by the PRINT statement in line 50.

## **MAN**

- Applesoft
  - Integer
- 

The MAN command is used to turn off the automatic generation of program lines.

### **Configuration**

#### **MAN**

When the computer is automatically generating program lines, Ctrl-X must be pressed before entering the MAN command.

### **Example**

```
> AUTO 10 [Ret]
> 10 REM TEST [Ret]
> 20 \           Ctrl-X entered by user
> MAN [Ret]
```

In the preceding example, the AUTO was turned off by the combination of Ctrl-X and MAN.

## **MID\$**

- Applesoft
  - Integer
- 

The MID\$ function returns the portion of a string specified by its argument.

**Configuration**

MID\$ (a\$, b[,c])

a\$ is a string constant. *b* and *c* are numeric constants or numeric expressions with a value between 0 and 255. *b* is the first character in a\$ being returned. *c* is the number of characters in a\$ being returned. If *c* is not included, all characters to the right of the position given in *b* will be returned.

The MID\$ function can be duplicated in Integer BASIC using string arrays.

**Example**

```
10 N$ = "COMPUTER"
20 PRINT MID$ (N$, 4, 3)
30 END
]RUN [Ret]
PUT
```

In the preceding example, the fourth position in the string N\$ is the starting position. The 3 indicates 3 characters. This could be duplicated in Integer BASIC by using N\$(4,6) in place of MID\$ (N\$,1,3). Integer BASIC also requires that the string variable had been dimensioned as DIM N\$(8).

**NEW**

- Applesoft
- Integer

---

The NEW command deletes the program in memory and clears all variables.

**Configuration**

NEW

The NEW command is generally used to free memory space before a new program is entered.

**Example**

```
> LIST [Ret]
  10 TEXT
  20 END
> NEW [Ret]
> LIST [Ret]
```

**NORMAL**

- Applesoft
  - Integer
- 

The NORMAL statement turns off the FLASH or INVERSE statements.

**Configuration**

NORMAL

The NORMAL statement sets the video output mode to white characters on a black background. The NORMAL statement is equivalent to the POKE 50,255 statement. Since NORMAL is not available in Integer BASIC, the POKE 50,255 can be used in its place.

**Example**

NORMAL

**NOT**

- Applesoft
  - Integer
- 

The NOT function logically compliments the value given in the *argument*.

**Configuration**NOT *argument*

*argument* is a numeric constant or numeric expression. In Integer BASIC the numeric constant must be an integer. If the *argument* evaluates to true (non-zero), false (zero) will be returned. If the argument evaluates to false (zero), true (one) will be returned.

NOT 1 = 0

NOT 0 = 1

**Example**

```

10 A = 2
20 IF NOT (A = 1)
   THEN PRINT "A DOES NOT EQUAL ONE"
30 END
]RUN [Ret]
A DOES NOT EQUAL ONE

```

**NO TRACE**

- Applesoft
  - Integer
- 

The NO TRACE command turns off the TRACE command.

**Configuration**

NO TRACE

The NO TRACE command may be used as a program statement.

**Example**

NO TRACE



**ON**

- Applesoft
  - Integer
- 

The ON statement is used in conjunction with GOTO and GOSUB. The statements are used to branch program control to one of several program lines depending on the value appearing after ON.

**Configuration**

```
ON exp GOTO line [,line ...]
ON exp GOSUB line [,line ...]
```

*exp* can be any numeric constant or numeric expression. *line* is the line number the program is to branch to.

The value of *exp* controls which *line* is to be branched to. For instance, if *exp* evaluates to 1, program control will branch to the line number given in the first *line*. If *exp* evaluates to 2, program control will branch to the second *line*, etc...

If the ON...GOSUB statement is being used, the line number specified in *line* must be that of a subroutine. In other words, a RETURN statement eventually will have to be executed to return program control.

If *exp* evaluates to zero or to a number greater than the number of *lines* specified after GOTO or GOSUB, the program will continue with the next executable statement.

**Example**

```
10 INPUT "ENTER A NUMBER BETWEEN 1 AND 4 ";I
20 ON I GOTO 60,80,100,120
30 PRINT
40 INPUT "PLEASE ENTER A NUMBER BETWEEN 1 AND 4 ";I
50 GOTO 20
60 PRINT "YOU ENTERED A ONE"
70 GOTO 130
```

*program continued on the next page*

```

80 PRINT "YOU ENTERED A TWO"
90 GOTO 130
100 PRINT "YOU ENTERED A THREE"
110 GOTO 130
120 PRINT "YOU ENTERED A FOUR"
130 END

```

In the preceding example, line 10 prompts the user to enter a number between one and four. In line 20, an ON...GOTO will branch control to a different line depending on the value of I. If I is one, program execution will branch to 60. If I is two, program execution will branch to 80, etc. If zero or a number greater than four was entered, program execution will continue to line 30.

## ONERR GOTO

- Applesoft
- Integer

The ONERR statement allows errors to be trapped. The statement then transfers program control to an error handling routine at the indicated line number.

### Configuration

ONERR GOTO *line*

*line* is the first line of the error handling routine. ONERR GOTO should be executed before the error has occurred.

When Applesoft executes a program, it executes the program line by line. If an error occurs during program execution, Applesoft will check to see if an ONERR GOTO statement has been executed. If no ONERR GOTO statement had been executed, Applesoft will halt program execution and display the error. Otherwise the program will branch to the *line* indicated in the ONERR GOTO statement.

POKE 216,0 turns off any previously executed ONERR GOTO statement.

To find out what error has occurred, execute PEEK (222). The value returned will be the error code. The following list indicates the various Applesoft error codes and their respective error messages.

Error Code	Error Message
0	NEXT without FOR
16	Syntax
22	RETURN without GOSUB
42	Out of DATA
53	Illegal Quantity
69	Overflow
77	Out of Memory
90	Undefined Statement
107	Bad Subscript
120	Redimensioned Array
133	Division by Zero
163	Type Mismatch
176	String Too Long
191	Formula Too Complex
224	Undefined Function
254	Bad Response to INPUT Statement
255	Ctrl C Interrupt Attempted

For a list of DOS errors and error codes, see Appendix F.

The RESUME command can be used to return the program to the beginning of the statement where the error occurred.

**Example**

```

10 ONERR GOTO 1000
20 INPUT "ENTER A NUMBER"; A
30 PRINT "A ="; A
40 END
1000 E = PEEK (222): REM ERROR CODE
1010 IF E = 255 THEN END: REM Ctrl-C ENTERED
1020 IF E = 254 THEN PRINT "INVALID ENTRY": RESUME
1030 IF E = 69 THEN PRINT "NUMBER TOO LARGE/
      TOO MANY DIGITS": RESUME
1040 PRINT "ERROR CODE"; E
1050 END

```

```

]RUN [Ret]
ENTER A NUMBER ONE [Ret]
INVALID ENTRY
ENTER A NUMBER 7E50 [Ret]
NUMBER TOO LARGE/TOO MANY DIGITS
ENTER A NUMBER 5 [Ret]
A = 5

```

In line 10, the ONERR routine is set. When an invalid number was entered in response to the INPUT statement in line 20, the program branched to line 1000 where E was assigned the proper error code. Lines 1010-1030 check the variable E for its value. If E = 255, then a Ctrl-C was entered. If E = 254, an invalid number was entered. If E = 69, the number entered was too large or had too many digits. Lines 1020 and 1030 incorporate the RESUME statement to branch back to line 20.

**OR**

- Applesoft
- Integer

---

OR is a logical math operator. This reserved word is generally used in conjunction with the IF...THEN statement.

### Configuration

*expression 1* OR *expression 2*

*expression 1* and *expression 2* are Boolean expressions. If the *expression* is numeric (non-zero), it will be evaluated to true. A zero is treated as false. A truth table for OR is illustrated below.

X	Y	X OR Y
true	true	true
true	false	true
false	true	true
false	false	false

In both Applesoft and Integer BASIC, a true is represented by a 1 and false by a 0.

### Example

```

10 A = 3
20 B = 5
30 IF (B < A) OR (B = 5) THEN 50
40 END
50 PRINT "EITHER B IS LESS THAN A"
60 PRINT "OR B IS EQUAL TO 5"
70 END

```

```

]RUN [Ret]
EITHER B IS LESS THAN A
OR B IS EQUAL TO 5

```

In the preceding example, B is not less than A, but B is equal to 5. Therefore, the whole OR expression is true, and the program branches to line 50.

**PDL**

- Applesoft
  - Integer
- 

The PDL function returns the value of one of the four different game controllers (paddles).

**Configuration**

PDL (*argument*)

*argument* is a numeric constant or numeric expression. The numeric constant must be an integer in Integer BASIC. The value of the *argument* must lie between 0 and 255. The value of the *argument* corresponds to the game controller. For example, X = PDL(0) returns the position of game controller number zero. The number returned will be between 0 and 255.

If the value of the *argument* is between 4 and 255, the PDL function will return an unpredictable number. Using an *argument* between 4 and 255 can also produce unwanted side effects which affect program execution.

**Example**

```
PRINT PDL(0)
120
```

**PEEK**

- Applesoft
  - Integer
- 

The PEEK function returns the contents of the memory address given in the *argument*.

**Configuration**

PEEK (*argument*)

*argument* is a numeric constant or numeric expression between -65535 and 65535 (-32767 to 32767 in Integer BASIC). The numeric constant must be an integer in Integer BASIC.

The decimal integer returned by the function will lie between 0 and 255.

### Example

```
PRINT PEEK (-857)
202
```

The preceding example returns the contents of location -857.

## **PLOT**

---

- Applesoft
- Integer

The PLOT statement plots a dot on the low resolution graphics screen. The color of the dot must have been previously defined by the COLOR statement.

### Configuration

PLOT *column, row*

*column* and *row* must be a numeric constant or a numeric expression. The numeric constant must be an integer for Integer BASIC. *column* must lie between 0 and 39 and *row* must lie between 0 and 47.

The PLOT occurs at the position specified. For example PLOT 3,5 would place a dot at row five and column three.

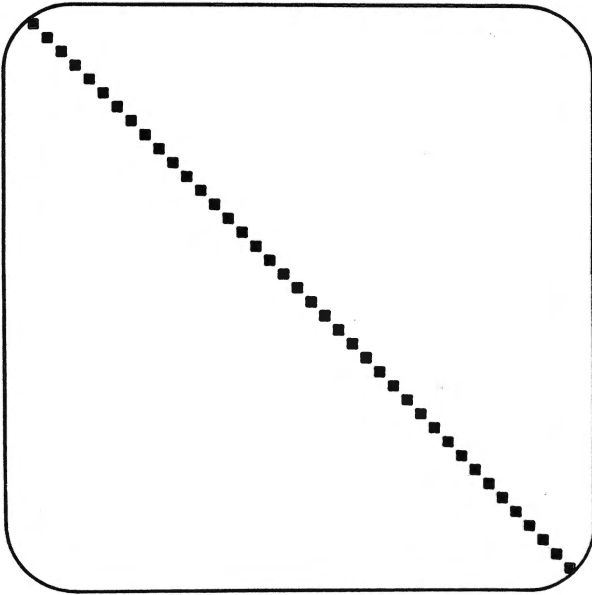
The origin (0,0) is located in the upper left hand corner of the screen.

If a PLOT statement is executed when the text mode is active a character will be placed where the dot should have appeared. The same will happen if a PLOT is made in the row range of 40 and 47 in the mixed graphics-text mode.

### Example

```
10 GR
20 COLOR = 3
30 FOR I = 0 TO 39
40 PLOT I,I
50 NEXT I
60 END
```

The preceding example will draw a diagonal line across the screen.



## **POKE**

- Applesoft
- Integer

The POKE statement stores one byte of information in the memory location specified.



## Configuration

POKE *address, value*

*address* and *value* are numeric constants or numeric expressions. *address* or *value* must evaluate to an integer in Integer BASIC. *address* lies between -65535 and 65535 (-32767 and 32767 in Integer BASIC.) *value* must lie between 0 and 255.

The POKE statement places the indicated *value* at the specified memory *address*. A POKE has no effect if the *address* is in ROM memory. If a POKE is not used carefully, it can disrupt the IIe's execution.

### Example

```
10 PRINT PEEK(7900)
20 POKE 7900,37
30 PRINT PEEK(7900)
40 POKE 7900,158
50 PRINT PEEK(7900)
60 END
```

In the preceding example, line 10 first displays what is currently in memory location 7900. The value 37 is then poked into memory on line 20. Line 30 displays the value at memory location 7900. The value 158 is then poked into memory and displayed in lines 40 and 50.

## POP

- Applesoft
- Integer

---

The POP statement causes a program to ignore the GOSUB or ON/GOSUB statement that was executed last.

## Configuration

POP

In effect, a GOSUB or ON/GOSUB statement is converted to a GOTO or ON/GOTO statement when POP is executed. The program "forgets" that it is in a subroutine. As a result, when a POP statement is executed, the next RETURN statement branches the program control to the line after the GOSUB statement before the previous GOSUB statement. In other words, the program "forgets" where the subroutine was called from, so it returns to a previous GOSUB statement.

A POP statement is used, in general, to exit a subroutine.

### Example

```

10 X = 5
20 Y = 10
30 GOSUB 100
40 END
100 PRINT X
110 IF X > 0 THEN POP:GOTO 130
120 RETURN
130 PRINT Y
140 END
]RUN [Ret]
5
10

```

The previous example contains a program that uses a POP statement to exit a subroutine. At line 10, X is assigned the value 5. At line 20, Y is assigned the value 10. At line 30, the subroutine at line 100 is called.

At line 100, the value of X is displayed. Line 110 is an IF/THEN statement that tests the condition  $X > 0$ . Since the value of X is greater than zero, the condition is true. As a result, the POP statement is executed, and the program control branches to line 130. At line 130, the value of Y is displayed.

Since the POP statement was executed, the program is no longer in the subroutine. If another RETURN statement is executed, the program will not return to line 30, where the subroutine was called. The program will return to the line of the previous GOSUB statement. Since there is no other GOSUB statement in this program, a RETURN statement would cause an error.

A POP statement can also be used to make the program ignore the previous FOR statement. When a POP statement is executed within a FOR/NEXT loop, the loop will not be repeated. However, an error occurs if a NEXT statement is executed for that loop.

## **POS**

- Applesoft
- Integer

The POS function returns the current horizontal position of the cursor.

### **Configuration**

POS (*argument*)

*argument* can be any legal Applesoft constant or expression.

The number returned will be an integer from 0 to 39. The leftmost position is 0.

### **Example**

```

]HTAB 7 : PRINT POS(0)
      6
]PRINT TAB(7); POS(0)
      6
]PRINT SPC(7); POS(0)
      7

```

In the previous example, the HTAB and TAB functions count the leftmost position as 1. The SPC and POS functions treat the leftmost position as 0.

## **PRINT**

- Applesoft
- Integer

PRINT is used to display information to the screen or to another output device.

### **Configuration**

PRINT [*expression*] [;...[*expression*]...]

*expression* can be any valid numeric or string constant or expression.

*expression* can include string and numeric variables, as well as string and numeric constants. Each variable name or constant must be separated by either a comma or a semicolon. When a comma separates the items in a PRINT statement, the display is divided into three display positions in Applesoft BASIC. These begin in columns 1, 17, and 33. In Integer BASIC, the display is divided into five fields whose display positions begin at columns 1, 9, 17, 25, and 33.

A PRINT statement can end with a comma, semicolon, or with no punctuation at all. A PRINT statement that ends with a semicolon causes any subsequent PRINT statement output to appear at the next position on the same row of output.

When a PRINT statement ends with a comma, the next PRINT statement output occurs at the next PRINT display position on the same row of output.

When a PRINT statement has no punctuation at the end, the next line of output automatically occurs on the next display line.

---

\* When the 80-column card is active the only two tab positions are set (columns 1,9 in Integer; 1,17 in Applesoft).

**PR#**

- Applesoft
  - Integer
- 

PR# specifies the peripheral slot which will be providing subsequent output for the IIe.

**Configuration**PR# *argument*

*argument* is a numeric constant or numeric expression which specifies the peripheral slot. The numeric constant must be an integer for Integer BASIC. The value of the *argument* must be between 1 and 7.

If there is not peripheral in the specified slot the system will hang. Press the reset key to exit this situation.

**READ**

- Applesoft
  - Integer
- 

A READ statement is used to assign values to variables. The values are taken individually from DATA statements in the order they appear in the program.

**Configuration**

```
READ a [ ,b
        a$ [ ,b$ ... ]
```

Data items are assigned to variables in the order in which they appear in the program unless a RESTORE statement has been executed.

The type of variable in the READ statement must correspond to the type of data in the corresponding DATA statement. A numeric variable can only be assigned a numeric value. However, a string variable can accept any type of character or none at all.

A program must include at least as many data items as the number of variables in its READ statements unless a RESTORE statement is executed.

**Example**

```

20 READ X,X$
30 PRINT X$,X
40 END
50 DATA 12, JONES
]RUN [Ret]
JONES    12

```

The preceding example contains a program that has a READ statement. At line 20, the variables X and X\$ are assigned the values from the DATA statement at line 50. At line 30, the values of the two variables are displayed.

A READ statement can accept data from a DATA statement that appears anywhere in a program. A DATA statement does not have to precede the READ statement in order to be effective.

- Applesoft
- Integer

**RECALL**

RECALL is used to recover a numeric array from cassette tape which previously was saved on tape with the STORE statement.

**Configuration**

RECALL *array*

It is not necessary to use the same array variable name in the RECALL statement that was used in the STORE statement. However, that array should be dimensioned with the same number of elements in the same dimensions as the array that was stored. If the array being recalled was dimensioned with fewer elements than the stored array, the following error message will be displayed:

OUT OF MEMORY ERROR

If the array being recalled was dimensioned with more elements than the stored array, the values in the recalled array may be in the wrong order.

However, the recalled array may be dimensioned with more elements than the stored array if only one final dimension is larger. For example, if the stored array was dimensioned as follows,

$$\text{DIM A}(10,10)$$

and the recalled array was dimensioned as:

$$\text{DIM A}(12,10)$$

the values recalled would be out of order.

However, if the recalled array was dimensioned as,

$$\text{DIM A}(10,12)$$

the recalled values would be in the correct order with zeros stored in the extra array elements.

If the recalled array is dimensioned with the same number of elements as the stored array but with different dimensions, any of the following error messages or error conditions could result:

<b>ERR</b>	<b>OUT OF MEMORY ERROR</b>
Extra zero values in the recalled array	Data out of order in the array

The user is not prompted with cassette operating instructions during the execution of either STORE or RECALL. It is a good programming practice to precede STORE and RECALL statements with PRINT statement prompts instructing the operator to press the proper cassette recorder keys.

The cassette recorder must be in the record mode when STORE is executed. The recorder will beep once when recording begins, and will beep a second time when it ends.

The cassette recorder must be in the play mode when RECALL is executed. The recorder will beep once when the reading begins and a second time when it ends.

### Example\*

]LOAD STOR1 [Ret]

]LIST [Ret]

```

5 REM STOR1-- STORE EXAMPLE
10 DIM A(9)
15 DATA 1,2,3,4,5,6,7,8,9,10
20 FOR X = 0 TO 9
30 READ A(X)
40 NEXT X
50 INPUT "SET THE RECORDER ON RECORD
    AND PRESS RETURN"; A$
60 STORE A

```

]RUN [Ret]

SET THE RECORDER ON RECORD AND PRESS RETURN

]NEW [Ret]

! Press Play & Record on Cassette

]LOAD RECAL1 [Ret]

]LIST [Ret]

```

5 REM RECAL1-- RECALL EXAMPLE
10 DIM A(9)
20 INPUT "PRESS PLAY ON THE RECORDER
    AND PRESS RETURN"; A$
30 RECALL A
40 FOR X = 0 TO 9
50 PRINT A(X)
60 NEXT
]RUN [Ret]

```

*program continued on the next page*

---

\* This example assumes STOR1 and RECAL1 had previously been saved on diskette.



**PRESS PLAY ON THE RECORDER AND PRESS RETURN**

```
1                               † Rewind Recorder
2                               Press Play
3
4
5
6
7
8
9
10
]
```

**REM**

- Applesoft
  - Integer
- 

A REM statement is used to insert comments in a program. The REM statement is ignored by the BASIC interpreter.

**Configuration**

REM *remarks*

**Example**

REM INPUT ROUTINE

Any statements that follow a REM statement, on the same line, are also ignored by the computer. As a result, a REM statement is generally used on its own line or at the end of a multiple statement line.

**RESTORE**

- Applesoft
  - Integer
- 

A RESTORE statement is used to move the DATA statement pointer to the beginning of the DATA item list.

## Configuration

### RESTORE

The data in a program is read in order, starting with the first DATA statement item. In order to reread the data, a RESTORE statement is necessary.

When a RESTORE statement is executed, the next READ statement will assign to its first variable the first data value that appears in the program.

### Example

```

10 READ A1,B1,C1,X1$
20 PRINT A1,B1,C1
30 PRINT X1$
40 RESTORE
50 PRINT
60 READ A2,B2,C2,X2$
70 PRINT A2,B2,C2
80 PRINT X2$
90 READ X3$
100 PRINT X3$
110 DATA 32,-102,2.12,RECTOR,SPALL

```

In the preceding example, data is read into the variables indicated in line 10. The data is then displayed in lines 20 and 30. The RESTORE statement in line 40 allows the data items read in line 10 to be read again.

## RESUME

- Applesoft
- Integer

---

RESUME is used in Applesoft BASIC to resume program execution after an ONERR GOTO statement has branched program control to an error routine.

**Configuration**

## RESUME

If RESUME is executed without an error having previously occurred, the program will stop, the system will hang, or an error message may result.

**RETURN**

- Applesoft
  - Integer
- 

A RETURN statement is used to branch a program back to the line where the last subroutine was called.

**Configuration**

## RETURN

A subroutine is called with a GOSUB or ON/GOSUB statement. When the subroutine has been completed, a RETURN statement causes program control to return to the statement following the most recently executed GOSUB or ON/GOSUB statement.

**Example**

## RETURN

**RIGHT\$**

- Applesoft
  - Integer
- 

The RIGHT\$ statement is used to return the rightmost characters of a string.

**Configuration**

$b\$ = \text{RIGHT\$}(a\$,c)$

The **RIGHT\$** function returns a string value. The first argument is a string constant or a string variable. The second argument is a numeric value. The string returned consists of the number of characters specified by the numeric argument. These characters are the rightmost characters in the string argument.

### Example

```
10 A$ = "WILLIAM JONES"
20 PRINT A$
30 PRINT RIGHT$(A$,5)
]RUN [Ret]
WILLIAM JONES
JONES
```

The preceding example contains a program that uses a **RIGHT\$** statement. At line 10, the string variable **A\$** is assigned the value "WILLIAM JONES". At line 20, the value of **A\$** is printed. At line 30, the rightmost 5 characters of the value of **A\$** are displayed.

If the value of the numeric argument exceeds the length of the string argument, the entire string value is returned. If the value of the numeric argument is less than one or greater than 255, the following error message will be displayed:

ILLEGAL QUANTITY ERROR

## RND

- Applesoft
- Integer

The **RND** function is used to generate "random" numbers.

### Configuration

$X = \text{RND}(a)$

In Applesoft BASIC, **RND** will return a random number greater

than or equal to zero and less than one. If RND's argument (a) is positive, a new random number will be generated each time RND is executed.

RND can also be used with a negative argument. The same random number will be returned when RND is executed with the same negative value for a.

If the same negative argument is repeated followed by RND statements with positive arguments, the same series of random numbers will be generated. This is illustrated in the following example.

### Example

```
100 PRINT RND (-1)
200 PRINT RND (3)
300 PRINT RND (.22)
400 PRINT RND (.33)
500 PRINT RND (-1)
600 PRINT RND (.99)
700 PRINT RND (2.7)
800 PRINT RND (.77)
]RUN [Ret]
2.99196472E-08
.738207502
.272707136
.299733446
2.99196472E-08
.738207502
.272707136
.299733446
```

In Applesoft BASIC, if  $a = 0$ , RND will return the most recently generated random number.

In Integer BASIC, RND returns a random integer with a value greater than or equal to zero but less than a.

**ROT=**

- Applesoft
  - Integer
- 

ROT= sets the amount of rotation for a shape which is to be drawn with either DRAW or XDRAW.

**Configuration**

ROT=x

x can range from 0 through 255. The shape will be rotated 90 degrees clockwise for every increment of 16 in the value of x. For example, ROT=0 causes the shape to be drawn in the same position in which it was originally defined. ROT=16 causes the shape to be rotated 90° clockwise. ROT=32 causes the shape to be drawn upside down. ROT=64 causes the shape to be drawn in its original position.

The number of actual different rotations is limited by the SCALE= setting. Lower SCALE= settings will have fewer noticeable rotations. When SCALE= is set to one, there are only eight noticeable ROT= values. They are 0,8,16,24,32,40,48,56 and numbers greater than 63 which would use the MOD64 equivalent. If a number other than 0,8,16,24,32,40,48, or 56 is used, the shape will generally be drawn with the lower corresponding ROT= value.

ROT= is only regarded as a reserved word if the equal sign (=) is placed immediately following ROT without any intervening blanks, spaces or characters.

**Example**

ROT=32

The ROT= statement given in our example would cause a shape previously defined by DRAW or XDRAW to be rotated 180 degrees.

**RUN**

- Applesoft
  - Integer
- 

RUN is used to execute the BASIC program currently stored in memory. Prior to program execution, all variables, pointers, and stacks will be cleared.

**Configuration**

RUN [*linenumber*]

If *linenumber* is specified, execution will begin at the specified line. If no *linenumber* is specified, execution will begin with the lowest line number. If a non-existent *linenumber* is specified, one of the following error messages will result:

```
***BAD BRANCH ERR   Integer
?UNDEF'D STATEMENT ERROR   Applesoft
```

In Integer BASIC, RUN can only be executed in the immediate mode.

**SAVE**

- Applesoft
  - Integer
- 

The SAVE command is used to store a program on diskette or cassette.

**Configuration**

```
Cassette   SAVE
Diskette   SAVE filename [Dx] [,Vx] [,Sx]
```

When using the cassette SAVE, the cassette recorder's Play and Record keys must be depressed when SAVE is executed.

The IIe will beep when it begins to save the program and will

keep a second time when the recording has been completed. At this time, the operator should stop the cassette recorder.

SAVE is only interpreted as a BASIC reserved word when used with the cassette unit.

In the DOS SAVE, SAVE need only be entered with the program's *filename* and the return key pressed. If the indicated *filename* duplicates that of a file in the same language already on the diskette, the contents of the original program will be erased and replaced with the new program. If the indicated *filename* duplicates that of a file in a different language or with a different file type, the following message will be displayed:

#### FILE TYPE MISMATCH

#### Example

SAVE

#### **SCALE=**

- Applesoft
- Integer

---

SCALE= is used to set the size of shapes drawn by DRAW or XDRAW in high resolution graphics.

#### Configuration

SCALE=x

The integer value of *x* is multiplied by the size of the shape table. Therefore, when SCALE=1, a shape is drawn with the same scale as that with which it was defined. SCALE=2 causes the shape to be drawn at twice its defined size, etc. SCALE=0 causes the shape to be drawn at 256 times its defined size.



**Example**

```

10 REM SET UP SCREEN
20 TEXT:HGR
30 FOR X = 7936 TO 7946
40 READ V:REM READ IN SHAPE
50 POKE X,V:REM POKE SHAPE INTO MEMORY
60 NEXT X
70 REM TELL WHERE SHAPE IS AT
80 POKE 232,0: POKE 233,31
90 HCOLOR = 3
100 ROT = 0
110 FOR X = 10 TO 273 STEP 24
120 INPUT "ENTER SCALE ";S
130 IF S < 1 THEN X = 273: GOTO 160
140 IF S > 50 THEN X = 273: GOTO 160
150 SCALE = S: DRAW 1 AT X,100
160 NEXT X
170 DATA 1,0,4,0,39,36,45,53,54,63,0
180 END

```

In the preceding example, line 20 initialized the screen so that graphics could be drawn. The shape table was then read using the FOR NEXT loop in lines 30 to 60. Line 80 POKE'd the starting address of the shape. Lines 90 and 100 defined a color and rotation for the shape. Lines 110 to 160 then prompted the user to enter a scale size. The number entered will be the size of the next shape drawn on the screen. The program will then exit after the entry of eleven sizes or when the value entered is less than one or greater than fifty.

**SCRN**

- Applesoft
- Integer

---

SCRN is used in the low resolution graphics mode to return the color code of the point specified as its argument.

## Configuration

SCRN (x,y)

x and y can range from 0 to 47. If x is in the range from 0 to 39, SCR N will return the color code of the point whose column is indicated by x and whose row is indicated by y.

If x is in the range from 40-47 and y is in the range from 0-31, SCR N will return the color of the point whose column is equal to x-40 and whose row is equal to y+16.

If x is in the range from 40-47 and y+16 is in the range from 39-47, SCR N will return a number related to the text character in the text area before the graphics screen. If y+16 is in the range 48-63, SCR N will return a meaningless number.

When x is in the range 0-39 and y is in the range 0-23, the following expression will return the ASCII code of the character at that position.

$$\text{SCRN}(a, 2*6) + 16*\text{SCRN}(a, 2*b+1)$$

The character itself will be returned when the CHR\$ function is executed using the value returned by the above expression as its argument.

When SCR N is executed in the high resolution graphics mode, SCR N will continue evaluating the low resolution area of memory. The value returned by SCR N will have no relation to the high resolution screen.

SCR N is only regarded as a reserved word when the next non-space character is the left parenthesis.

**Example**

```

10 GR
20 COLOR = 2
30 PLOT 20,10
40 COLOR = 6
50 PLOT 30,35
60 PRINT SCRN(20,10), SCRN(30,35), SCRN(10,0)
70 END

```

In the preceding example, line 10 initializes the low resolution screen. Lines 20 to 50 plot two different colored dots on the screen. Line 60 displays the color value of the corresponding dots. Notice that the value of SCRN(10,0) is zero. There is a black dot at that position.

**SGN**

- Applesoft
- Integer

---

The SGN function returns a +1 if its argument is positive, a -1 if negative, and a 0 if zero.

**Configuration**

SGN (a)

**Example**

```

100 A = 100
200 X = SGN(A)
300 PRINT X
400 END
]RUN [Ret]
1

```

- Applesoft
- Integer

## **SHLOAD**

---

SHLOAD is used in Applesoft BASIC to load a high resolution shape table into memory from cassette tape.

### **Configuration**

#### **SHLOAD**

The shape table will be loaded in memory immediately below HIMEM. HIMEM will be set just underneath the shape table in order to protect it.

Be certain that HIMEM: has been set so that the execution of SHLOAD will not erase any programs or variables.

- Applesoft
- Integer

## **SIN**

---

The SIN function returns the sine of the angle specified as its argument. The argument will be assumed in radians.

### **Configuration**

$$X = \text{SIN}(a)$$

#### **Example**

```
PRINT SIN (3.1415927/2)
.999999992
```

**SPC**

- Applesoft
  - Integer
- 

The SPC statement is used to insert spaces in a PRINT statement.

**Configuration**

SPC (a)

The argument of the SPC statement specifies the number of blank spaces that will occur.

**Example**

```
10 X = 4
20 Y = 6
30 PRINT X;SPC(5);Y
40 END
]RUN [Ret]
4      6
```

In the previous example, the values of the variables X and Y are printed at line 30. The SPC statement within the PRINT statement causes the output to be separated by 5 extra spaces.

**SPEED**

- Applesoft
  - Integer
- 

The SPEED statement sets the speed at which characters are output.

**Configuration**

SPEED = x

x can range from 0 to 255, with 0 being the slowest speed and 255 the fastest.

**SQR**

- Applesoft
  - Integer
- 

SQR returns the square root of its argument.

**Configuration**

SQR (a)

**Example**

```
10 X = 49
20 PRINT SQR (X)
]RUN [Ret]
7
```

**STOP**

- Applesoft
  - Integer
- 

The STOP statement causes a halt in the execution of an Applesoft BASIC program.

**Configuration**

STOP

If STOP is executed in the program mode, the following screen message will be displayed,

BREAK IN XXX

where XXX is line number where STOP was executed.

CONT can be used to rescue program execution after it was halted by the execution of a STOP statement.

### Example

```
10 INPUT X
20 IF X = 10 THEN STOP
30 PRINT X
40 END
```

In the preceding example, if a value of 10 is input for X in line 10, the program execution will stop and the following message will be displayed.

BREAK IN 20

By entering CONT, program execution will resume with line 30.

## **STORE**

---

- Applesoft
- Integer

The STORE statement is used to save an Applesoft array on cassette tape. STORE is generally used in conjunction with the RECALL statement.

### Configuration

STORE *array*

For an explanation of STORE please refer to RECALL on page 141.

## **STR\$**

---

- Applesoft
- Integer

STR\$ returns the string representation of its argument.

### Configuration

X\$ = STR\$(a)

**Example**

```

10 A$ = STR$(40)
20 PRINT A$
30 END
]RUN [Ret]
40

```

In the preceding example, the string variable A\$ is assigned the string value "40". The STR\$ function converts the numeric value 40, to the string value "40".

**TAB**

- Applesoft
  - Integer
- 

In Applesoft BASIC, the TAB function moves the cursor to the right to the column specified as its argument. TAB must be used with a PRINT statement in Applesoft BASIC

**Configuration**

TAB (*column*)

TAB erases existing screen data as it moves to the right. If the specified *column* is not to the right of the current column position, the cursor will not move.

*column* can range from 1 to 255. If *column* is greater than that allowed for the output device (80 for video display), the cursor will move down to the next output line before tabbing will continue.



```

10 X = 1:Y = 2
20 PRINT
30 PRINT X;
40 PRINT TAB(120);
50 PRINT Y
60 END

```

In the preceding example, X is output at the leftmost column of the display line. TAB then moves the current print position to the middle of the next display line where Y is output. In Applesoft the TAB statement after produces erratic results if it is not the first item in the PRINT statement output list.

## TAB

- Applesoft
- Integer

In Integer BASIC, TAB positions the cursor at the specified column on the display line where the cursor is located.

### Configuration

TAB *column*

In Integer BASIC, the cursor will be moved to the left or right depending on the value of *column* and the current cursor position. *column* can range from 1 to 255. Existing screen data is not erased if TAB causes the cursor to move over that data.

### Example

```

10 PRINT
20 X = 1:Y = 2
30 PRINT X:TAB 20:PRINT Y:TAB 10:PRINT Y
40 END
> RUN [Ret]
1
2
2

```

**TAN**

- Applesoft
  - Integer
- 

TAN returns the tangent of its argument in radians.

**Configuration**

$$a = \text{TAN}(b)$$

**Example**

```
10 A = TAN(35*3.141593/180)
20 PRINT A
]RUN [Ret]
.700207639
```

**TEXT**

- Applesoft
  - Integer
- 

TEXT returns the screen to the text mode from any of the graphics modes.

**Configuration**

TEXT

TEXT does not necessarily clear the screen.

**TRACE**

- Applesoft
  - Integer
- 

TRACE displays the line number of each statement as it is executed. Generally, TRACE is used as a debugging tool.

## Configuration

### TRACE

TRACE can be turned off by executing NO TRACE.

## USR

- Applesoft
  - Integer
- 

USR passes program control to a machine language subroutine. USR's argument is evaluated and then placed in the floating point accumulator (9DH and A3H). A USR is then undertaken to 0AH, which is the subroutine's starting address.

Addresses 0AH to 0CH must contain a JMP to the starting address of the machine language subroutine.

Since USR is a function, it returns a numeric value. The value in the accumulator is returned when an RTS instruction is executed.

### Example

```
10 POKE 10,76 : POKE 11,0 : POKE 12,32
20 FOR X = 8192 TO 8198
30 READ V
40 POKE X,V
50 NEXT X
60 FOR X = 1 TO 4
70 PRINT X, USR(X)
80 NEXT X
90 DATA 165,157,105,3,133,157,96
100 END
```

In the preceding example, line 10 pokes the location of the machine language program to be executed. Lines 20 to 40 place the program in memory. Lines 60 to 80 then make use of the machine language program. The USR in this example multiplies the argument by 16.

**VAL**

- Applesoft
  - Integer
- 

The VAL function converts its string argument to a numeric value. The numeric characters in the string argument will be converted to their numeric equivalents until an unacceptable string character is encountered. The acceptable characters consist of the digits (0-9), the decimal point, a leading plus or minus sign, blank spaces, and in scientific notation an additional plus or minus sign, the letter E (for exponent), and an additional decimal point.

If the first character encountered by VAL is an unacceptable character, a value of zero is returned.

**Configuration**

VAL (a\$)

**Example**

```

10 A$ = "1.731E+02"
20 B$ = "+97.5"
30 C$ = "57CA"
35 D$ = "E59"
40 PRINT VAL(A$)
50 PRINT VAL(B$)
60 PRINT VAL(C$)
70 PRINT VAL(D$)
]RUN [Ret]
173.1
97.5
57
0

```

**VLIN**

- Applesoft
  - Integer
- 

VLIN is used to draw a vertical line in low-resolution graphics.

### Configuration

*VLIN row 1, row 2 AT column*

A vertical line will be drawn at the specified *column* from *row 1* to *row 2*. The color of the line will be determined by that specified in the last **COLOR** statement executed.

*row 1* and *row 2* must be in the range of 0 to 47. *column* must lie in the range 0 to 79. If a value outside of these ranges is used, the following error message will be displayed:

ILLEGAL QUANTITY ERROR

If **VLIN** is executed in the text mode, then a line of characters will be displayed rather than a line of graphics dots. This also occurs in the text window in the **GR** mode.

### Example

```
10 GR
12 COLOR = 3
15 FOR I = 1 TO 20
20 VLIN 10,30 AT I
25 NEXT I
30 END
```

## **VTAB**

- Applesoft
- Integer

---

**VTAB** moves the cursor to the indicated row on the screen. **VTAB** causes the cursor to move up and down but never sideways.

### Configuration

*VTAB row*

*row* can range from 1 to 24. A *row* value outside of that range

results in the following error message:

### ILLEGAL QUANTITY ERROR

#### Example

```

5 HOME : REM CALL -936
10 PRINT "ROW 1"
20 VTAB 2:PRINT "ROW 2"
30 VTAB 10:PRINT "ROW 10"
40 VTAB 20:PRINT "ROW 20"
50 END

```

## WAIT

- Applesoft
- Integer

WAIT can be used to suspend operation of an Applesoft BASIC program until a designated memory address is assigned a specified value.

### Configuration

WAIT *address*, *value 1*, [*value 2*]

*address* specifies the memory address whose value is to be decided. *value 1* and *value 2* are compared with *address* to determine whether the program proceeds or continues to wait.

*address* can range from -65535 to 65535. *value 1* and *value 2* can range from 0 to 255. When WAIT is executed, *value 1* and *value 2* are converted to their binary equivalent in the range from 0 through 11111111.

When *address* and *value 1* are the only arguments indicated with WAIT, each of the eight bits in *address* are AND'ed with each corresponding bit in the binary equivalent of *value 1*. If the result of the AND operation is non-zero for any one bit (the bit had a value of 1 in both *address* and *value 1*), the WAIT statement will

be completed and execution will be resumed. Otherwise, this test will continue until a non-zero value is returned.

When *value 2* is also specified, a two-part comparison process is used. First of all, each of the eight bits at *address* are XOR'ed with the corresponding bit in *value 2's* binary equivalent. Second, the result of the XOR comparison is AND'ed with the corresponding bit in *value 1's* binary equivalent. If a non-zero value is returned for any comparison, the program will proceed. Otherwise, the test will continue.

### Example

```

10 REM SET ADDRESS FOR GAME CONTROL 0 TO 0
20 POKE 49168,0
30 PRINT "THE PROGRAM HAS STARTED"
40 PRINT:PRINT
50 PRINT "EXECUTION WILL BE HALTED UNTIL
   THE BUTTON ON GAME CONTROL 0 IS PRESSED"
60 PRINT:PRINT
70 WAIT -16287,128
80 PRINT "THE BUTTON WAS PRESSED"
90 PRINT "THE PROGRAM WILL NOW END"

```

## **XDRAW**

- Applesoft
- Integer

XDRAW is used to draw a graphics shape in high resolution graphics.

### Configuration

XDRAW *shape* [AT *column*, *row*]

The color, scale, and rotation of the shape to be drawn must have been specified previously. *shape* denotes the number from the shape table of the shape to be drawn. *shape* must have a value

between 0 and the number of shapes in the table. The optional *column* and *row* denote the starting x and y coordinates for the shape.

XDRAW uses the color which is the compliment of the color which already exists at each point being plotted. The complementary colors are as follows:

0(Black)	→	3(White)
1(Green)	→	2(Violet)
4(Black)	→	7(White)
5(Orange)	→	6(Blue)

One other feature of XDRAW is that it allows a shape to be easily erased. If a shape is XDRAWn and then XDRAWn again with the same parameters, the shape will be erased without erasing the background.

### Example

```

10 REM SET UP SCREEN
20 TEXT:HGR
30 FOR X = 7936 TO 7946
40 READ V:REM READ IN SHAPE
50 POKE X,V:REM POKE SHAPE INTO MEMORY
60 NEXT X
70 REM TELL WHERE SHAPE IS AT
80 POKE 232,0:POKE 233,31
90 REM PUT UP A BACKGROUND
100 HCOLOR = 3
110 H PLOT 20,20 TO 60,20 TO 20,70
120 PRINT "BACKGROUND SET UP"
130 INPUT "PRESS RETURN TO DISPLAY SHAPES";A$
140 ROT = 0
150 SCALE = 10
160 XDRAW 1 AT 50,30
170 SCALE = 6
180 ROT = 4
190 XDRAW 1 AT 20,70

```

*program continued on next page*



```
200 INPUT "PRESS RETURN TO ERASE SHAPES";A$
210 XDRAW 1 AT 20,70
220 SCALE = 10
230 ROT = 0
240 XDRAW 1 AT 50,30
250 DATA 1,0,4,0,39,36,45,53,54,63,0
260 END
```

In the preceding example, line 20 initialized the screen so that graphics could be drawn. The shape is then read in the FOR NEXT loop in lines 30 to 60. Since Applesoft needs to know where the shape was placed, the shape address is POKE'd into memory in line 80.

A color is defined in line 100. The color definition is only necessary for the background, which consists of two lines. XDRAW does not need to define the color, however the size and the rotation should be defined. These are defined on lines 140, 150, 170, 180, 220 and 230.

The two shapes, with different sizes, rotations, and positions, are then XDRAW'n to the screen in lines 160 and 190. Notice that the shapes were drawn over the background (two lines). The two shapes were then drawn again over the previous shapes in lines 210 and 240. Because XDRAW plots the compliment of the color that is on the screen, it will erase the previous shape and keep the background intact.

# CHAPTER 5. CASSETTE AND DISK STORAGE WITH THE APPLE IIe

---

## Introduction

Generally, a disk drive is used for storage with the Apple IIe. However, a cassette recorder can also be used with the IIe for data storage. In this chapter, we will discuss cassette storage procedures followed by disk storage procedures.

If you are using the cassette recorder rather than the disk unit for data storage, keep the following points in mind:

- DOS is not available when using the cassette. Applesoft and the Monitor are the only available systems software.
- Since Integer BASIC is loaded from diskette, it is not available on cassette. Therefore, Integer BASIC programs cannot be loaded and run from cassette.

## CASSETTE INSTALLATION & OPERATION

Nearly any standard cassette recorder can be used with the IIe. The cassette unit is installed by attaching a double cable (or two individual cables) from the cassette in and out jacks at the rear of the IIe to the jacks on the cassette recorder. One cable should connect the cassette input jack on the IIe to the earphone or monitor jack on the cassette recorder. The other cable should connect the IIe's cassette output jack to the cassette recorder's microphone jack.

The cassette unit is operated in much the same manner with the IIe to record data as it is used to record sound. The play and record keys must be depressed to save data on the cassette. The play key must be depressed when data is to be read from the cassette into RAM. The rewind and fast forward keys are used to position the tape.

You may find it necessary to adjust the tone and volume controls in order for the cassette to properly record data. The easiest means of doing this is to try saving and reloading a program. If the program can be saved and then reloaded without any difficulties, the volume and tone controls are adjusted properly. If one of the following error messages appears:

ERR  
\*\*\*SYNTAX ERR

set the volume control higher and again attempt the saving and loading process.

### **SAVING & LOADING A PROGRAM ON CASSETTE**

The SAVE command is used to save a program from RAM to the cassette recorder. Prior to executing SAVE, be certain that a cassette has been placed in the cassette recorder and that the tape has been positioned as desired. Then, press the recorder's play and record keys and enter the following command at the keyboard:

SAVE [Ret]\*

The Apple IIe will beep once as it begins recording the program on tape, and will beep a second time when the recording process has been completed. After the second beep, press the recorder's stop key.

---

\*[Ret] indicates that the return key should be pressed.

If you wish to load a program from cassette tape into RAM, first of all enter NEW in order to erase any existing program lines from memory. Be certain that the tape has been rewound to the beginning point of the program to be loaded. Press the recorder's play key, and enter the following command:

LOAD

The Apple IIe will beep once as the loading process begins and a second time when it ends. After the second beep, press the recorder's stop key. You can verify the fact that the program has been loaded by executing LIST.

### **Storing & Loading Data on Cassette**

Numeric arrays can be stored and loaded on cassette using Applesoft's STORE and RECALL statements. These are discussed in detail in Chapter 4.

The Monitor's memory write and memory read commands can also be used to store and read data to and from tape. These will be discussed in detail in Chapter 7.

### **Apple IIe Disk Storage**

A disk drive can also be used with the Apple IIe to store data and/or programs. Disk storage is much more efficient than cassette tape storage. The majority of IIe owners are expected to use disk drives as their primary means of data storage.

### **TYPES OF DISKS**

There are three primary types of disks used by personal computers; **hard disks**, **Winchester disks**, and **floppy diskettes**. These will be described in the following sections.

#### **Hard Disks**

Microcomputer hard disk systems generally allow storage of 5 to 30 megabytes of data. One megabyte is the equivalent of one

million bytes. The hard disk itself is made of a rigid material with a magnetic coating. The disk drive and the hard disk are separate units. The operator can remove one hard disk and replace it with another.

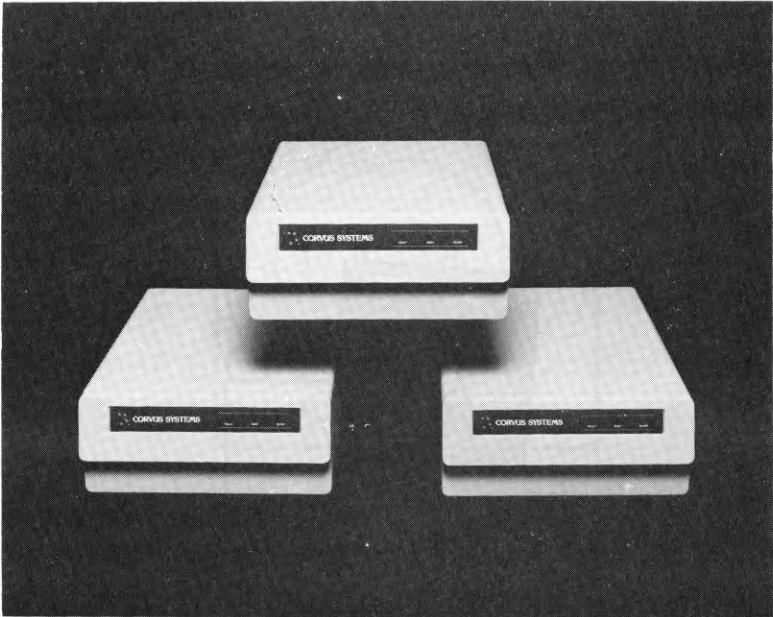
### **Winchester Disk Drives**

Winchester disk drives are designed so that from 6 to 10 times more data can be stored on their surface than on a standard floppy diskette. Winchester disks must be kept very clean as they are extremely vulnerable to dust, dirt and smoke.

Since they must be kept so clean, Winchester disks must be sealed inside of the disk drive. This means that Winchester disks cannot be changed.

Since Winchester disks cannot be removed, floppy disk systems often are used in conjunction with Winchester disks to allow for back-up storage. Winchester disk systems are generally used with microcomputers rather than hard disk systems. A Winchester drive is shown in Illustration 5-1.

### Illustration 5-1. Winchester Disk System



### Floppy Diskettes

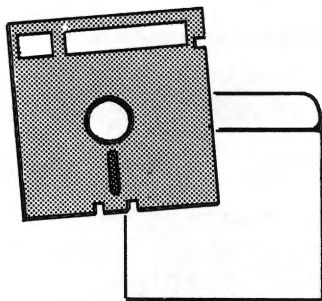
The most widely used type of disk storage with microcomputers is floppy disk storage. The Disk II Drive (used with the IIe) is a floppy disk drive. A floppy diskette consists of a round vinyl disk which is enclosed within a plastic cover. The diskette is generally stored in a diskette envelope.

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 Illustration 5-2.

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.

### **Illustration 5-2. Mini-Floppy Diskette**



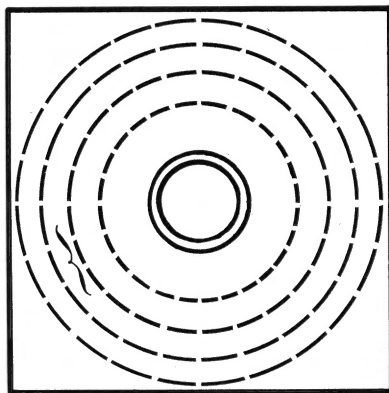
Floppy diskettes come in two sizes: 8 inch and 5 1/4 inch. The 5 1/4 inch diskettes are also known as mini-floppy diskettes. Apple's Disk II drives use mini-floppy diskettes.

### **Tracks and 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 Illustration 5-3.

### Illustration 5-3. Tracks and Sectors



In Apple's Disk II system, the diskette is divided into 35 different tracks.

To further reduce the time necessary to search for a particular data item, Apple's DOS\* divides each track into 16 sectors (see Illustration 5-3.)

Each sector can store up to 256 bytes of information. When the Disk II's read/write head\* is in place over a specific track, that track's 16 sectors will pass one by one underneath the read/write head as the diskette rotates.

Whenever DOS reads or writes information from the diskette, it does so in groups of 256 bytes. In other words, data is read from or written to the diskette one sector at a time.

---

\*DOS stands for disk operating system which will be explained in detail later.



## Hard and Soft Sectoring

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 Illustration 5-2. 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 Illustration 5-2 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 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 sector diskettes. A hard sector 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

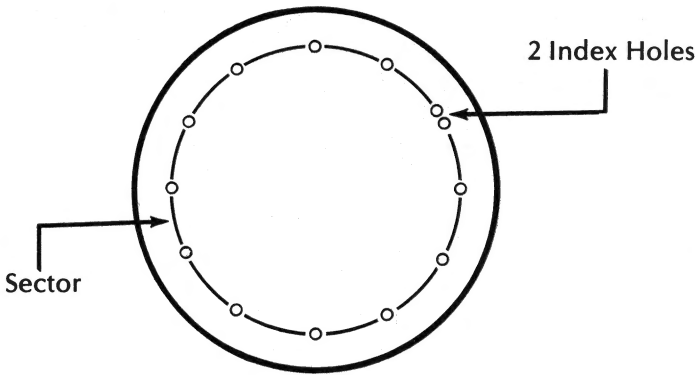
---

\*The 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.

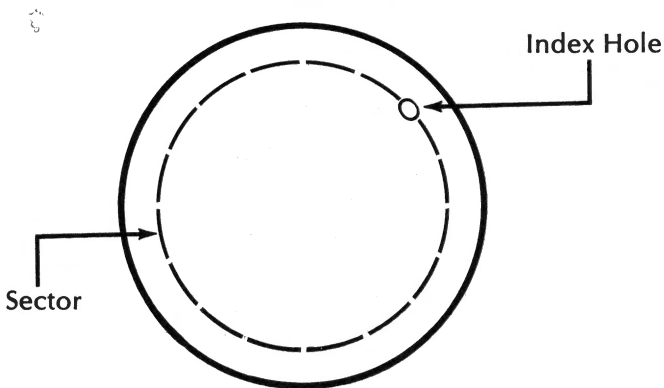
the various sectors is determined by counting the number of holes occurring after the first sector. A hard sectored diskette is depicted in Illustration 5-4.

Soft sectored diskettes have only one index hole as shown in Illustration 5-5. 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 Apple IIe Disk System uses soft-sectored diskettes.

**Illustration 5-4. Hard Sectored Diskette**



**Illustration 5-5. Soft Sectored Diskette**



### **Single and Double Sided Diskettes**

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.

### **Single, Double, and Quad Density Diskettes**

Density refers to a diskette's recording format, which in turn affects its capacity. Double density diskettes generally have a greater recording capacity than single density diskettes, while quad density diskettes generally have a greater capacity than either single or double.

The Apple IIe Disk Drive uses double-density diskettes with a capacity of 143,250 bytes.

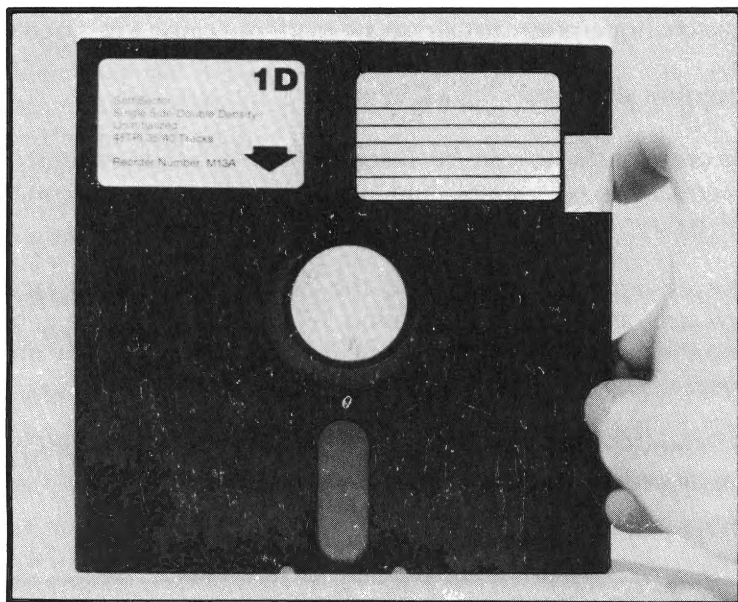
### **Diskette Write Protection.**

Diskettes have a notch on the side of their protective envelope which determines whether or not data can be written onto that diskette. On 8 inch diskettes, this notch is known as a write-protect notch. On 5¼ inch diskettes, it is known as a write-enable notch.

On a 8 inch diskette, information cannot be written onto the diskette unless this notch has been covered. On 5¼ inch diskettes, information cannot be written onto the diskette unless the notch is 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 Illustration 5-6.

**Illustration 5-6. Write Protecting a 5¼ inch Diskette**



### **Diskette Handling Rules**

Diskettes are easily damaged. Therefore, certain rules should be observed when handling or storing diskettes.

First of all, keep your diskettes clear of magnetic fields. Most devices with electric motors contain magnetic fields as do most telephones. Keep your diskettes clear of appliances with electric motors and telephones. However, it is safe to place diskettes on top of the IIe or the Disk II unit.

When writing on a diskette label, do so only with a felt tip pen. Never use a pencil or a ball point pen. Doing so can damage the diskette surface.

Exposure to the sun or to extreme heat can cause diskettes to warp. Be certain to keep your diskettes out of direct sunlight and away from heat sources.

Never touch the actual exposed surface (brown or grey) of the diskette. Handle only the diskette's black plastic cover. A scratch, fingerprint, dirt, or grease mark can cause a loss of data.

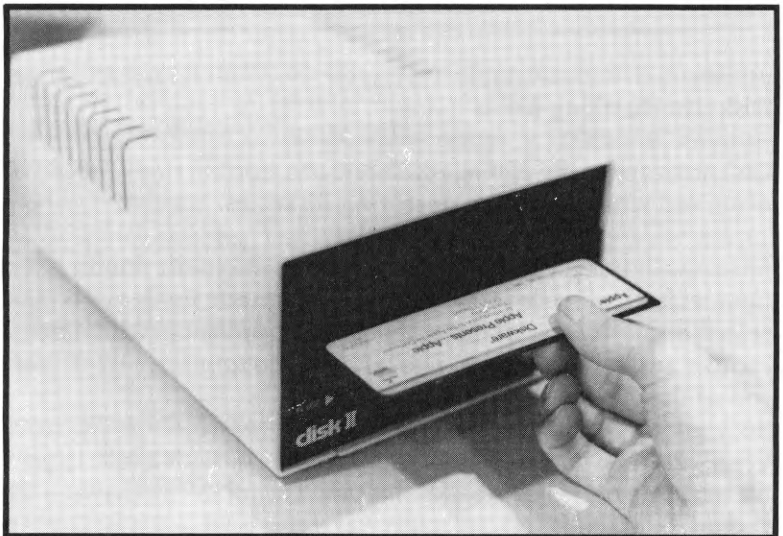
### **Inserting and Removing a Diskette**

Before a diskette can be inserted, the drive door must be opened. This is accomplished by gently pulling outward on the drive door lever.

The diskette is inserted into the drive with its label facing upwards, as shown in Illustration 5-7. Slide the diskette gently into the device until it clicks into place and then close the drive door.

To remove a diskette, merely open the drive door and gently pull the diskette out of the drive.

**Illustration 5-7. Inserting a Diskette**



## **DISK OPERATING SYSTEMS**

An operating system can be defined as a group of programs which manage the computer's operation. A disk operating system can be defined as a group of programs that manage the transfer of data to and from a storage device such as disk or magnetic tape.

The standard operating system provided with an Apple IIe with a Disk Drive II is Apple DOS 3.3 (referred to simply as DOS). Other operating systems such as PASCAL and CP/M are also available for the Apple IIe.

Apple's DOS will be discussed in detail in this chapter.

## **DISK II SYSTEM**

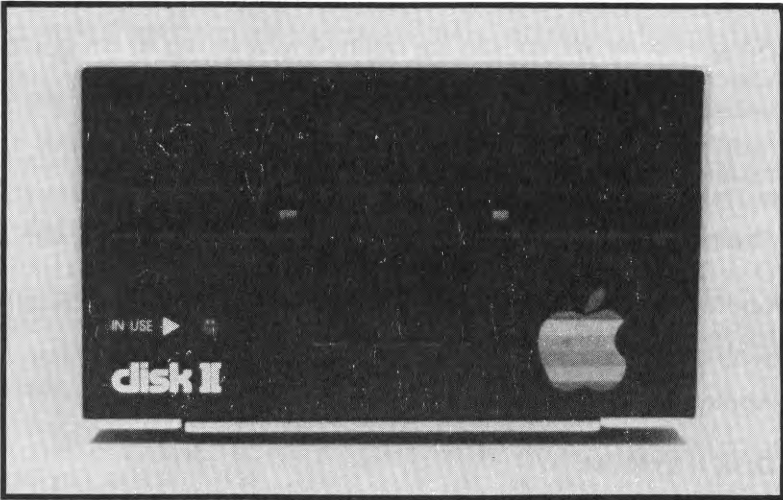
The name of the floppy disk drive used by the Apple IIe is the Disk II. The Disk II system includes the disk drive (see Illustration 5-8), disk controller card (see Illustration 5-9), a cable used to connect the disk controller card to the disk drive (see Illustration 5-9), a System Master diskette, and a BASICS diskette.

### **Installing The Disk II System**

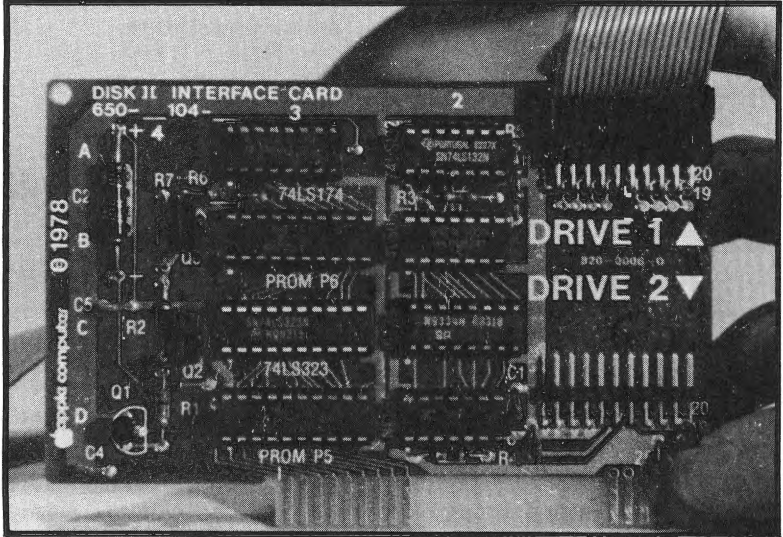
First of all, open the rectangular opening labeled 1 on the rear of the IIe. Next, insert the free end of the cable connected to the Disk II through this opening.

Locate one of the U-shaped clamps and two jack screws (see Illustration 5-10). Find the metal bar on the cable. Open the clamp slightly and slide it over the cable and the metal bar. The flat side of the clamp should be positioned against the cable. The raised side of the clamp should be positioned against the metal bar. Feed the cable through rectangular opening #1 until the clamp can be placed in the opening. Position the clamp in the opening, and install the jack screws into the nuts on the clamp. Tighten the screws with a small wrench. (See Illustration 5-11.)

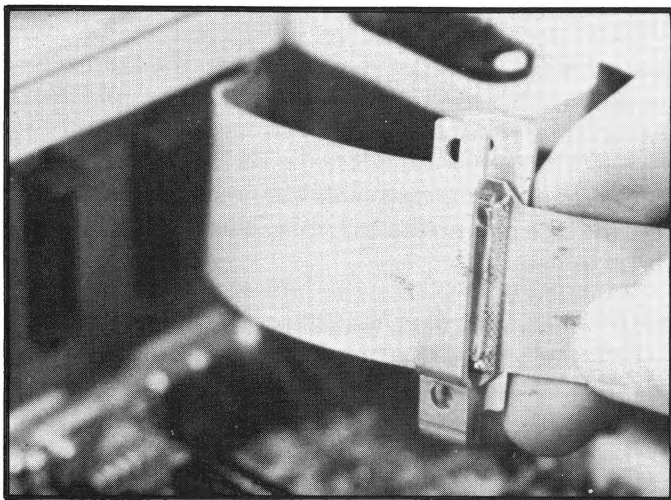
**Illustration 5-8. Disk II Drive**



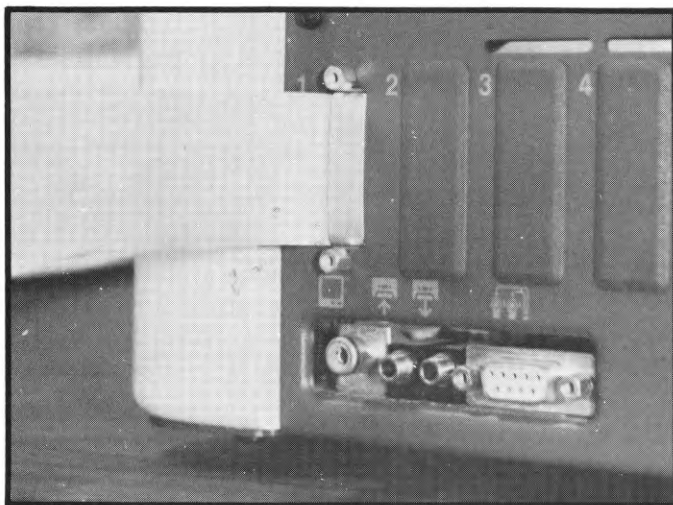
**Illustration 5-9. Disk Controller Card and Cable**



**Illustration 5-10. Inserting Clamp over Cable and Metal Flap**



**Illustration 5-11. Installing the Clamp on the Rear of the Apple IIe**



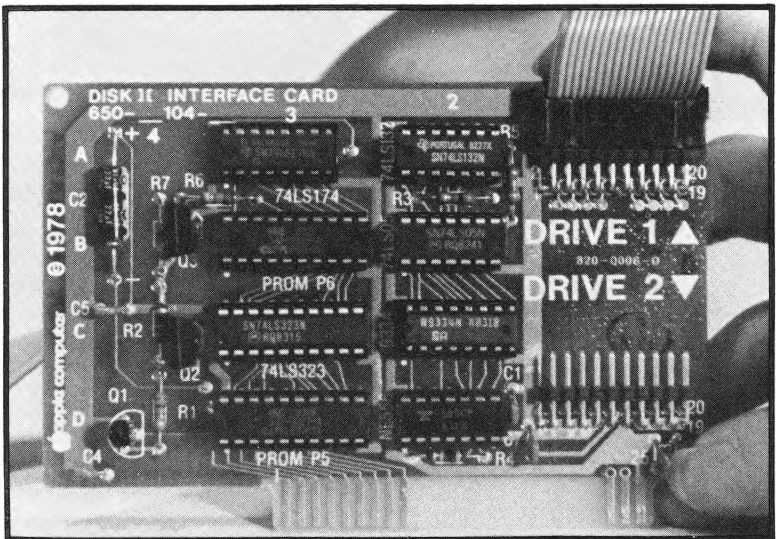


Next, install the cable from drive 1 to the pins labeled DRIVE 1 on the disk drive controller card. (See Illustration 5-12.)

Be sure to install the disk drive cable into the controller card before the card has itself been installed in the Apple IIe. Doing otherwise could result in damage.

Also, be certain that the disk cable connector has been properly inserted into the pins on the controller card. A faulty connection can result in damage.

**Illustration 5-12. Installing the Disk Drive Cable to the Disk Controller Card.**



Once the disk cable connectors have been properly connected to the disk controller card, the installation of the Disk II system can be completed by merely plugging the controller card into slot 6.

Before installing the disk controller card (or any other card), be certain that the power switch at the back of the IIe is off. If the power is on when a card is installed or removed, damage could result to the card, the IIe, or to both.

If you wish to install a second drive, follow the same procedure except insert the disk drive cable through opening #2 on the rear of the IIe, and attach the disk drive cable to the pins labeled DRIVE 2 on the disk controller card.

If you wish to install a third or fourth drive, a second disk controller card will have to be installed in slot 4. Insert the drive cables through openings 3 and 4 on the IIe's rear panel. The cable connector for drive 3 should be installed on the pins labeled DRIVE 1 on this disk controller card. The cable connector for drive 4 should be installed on the pins labeled DRIVE 2 on this card.

If you are installing more than one drive, it is a good idea to label your drives so as to prevent possible confusion. Remember that DOS must be booted from drive 1.

## **BOOTING DOS**

There are several different methods of booting DOS on the Apple IIe depending upon how your IIe system is configured as well as from which language DOS is being booted.

### **Autostart Boot**

The autostart boot is the easiest method of booting DOS 3.3. With the Apple IIe powered off, place the System Master Diskette in drive 1 and close the drive door. Then, turn the IIe's power on. The disk will spin for a few seconds and then stop. The screen display will resemble that depicted in Illustration 5-13.

### Booting from Integer or Applesoft BASIC

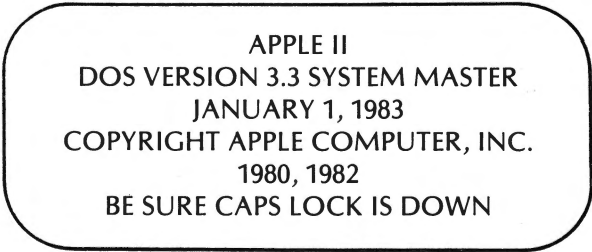
The prompt character for Integer BASIC is the greater than sign (>). The prompt for Applesoft BASIC is the left bracket ([). The same commands can be used to boot DOS from either Integer or Applesoft BASIC. This command consists of the letters IN or PR followed by the pound sign (#) and the slot number containing the controller card (generally (6)). The boot command should resemble one of the following:

PR#6

IN#6

Once the boot command has been entered, press the return key. Again, the drive will spin for several seconds, after which the screen display depicted in Illustration 5-13 will appear.

#### Illustration 5-13. DOS Start-up Screen Display



APPLE II  
DOS VERSION 3.3 SYSTEM MASTER  
JANUARY 1, 1983  
COPYRIGHT APPLE COMPUTER, INC.  
1980, 1982  
BE SURE CAPS LOCK IS DOWN

### Booting from the Monitor

The asterisk (\*) is the prompt for the assembly language monitor. One method of booting DOS from the monitor is to enter the following command and press return.

\* C600G

This causes DOS to be booted from drive 1 (assuming the disk controller card was installed in slot 6.)

Another method for booting DOS from the monitor is to enter the slot number of the drive to be booted from (generally 6) followed by Ctrl-P and Return.

### **Restarting DOS**

If the computer is already powered on, and you wish to restart the System using DOS, you can do so by as follows:

1. Insert the DOS diskette in drive 1.
2. Hold down the Open Apple key simultaneously with Ctrl-Reset. Release Ctrl-Reset and then release the Open Apple key.
3. The computer will beep. The drive's in-use light will come on and the drive will begin spinning.
4. When DOS has been loaded, the drive will stop and the DOS initial screen display will appear on the screen.

### **Booting DOS Conclusion**

Once you have booted DOS, you will still be able to execute most of your BASIC commands in the same fashion as if DOS had not been booted. A few BASIC commands will have additional capabilities under DOS. You will also have the option of executing DOS commands.

This is not the case in other operating systems such as CP/M where DOS commands cannot be entered when BASIC is active and vice versa.

### **Using 13-Sector Diskettes**

While DOS 3.3 initializes diskettes with 16 sectors per track, the earlier version of DOS (3.21 and 3.2) used diskettes with 13 sectors. The 13 sector diskettes can be run under DOS 3.3 by using the BASICS diskette rather than the System Master in the loading process. A prompt will appear instructing you to load your 13 sector diskette. Remove the BASICS diskette, insert the 13 sector diskette, and press return. You can now use the 13 sector diskette.

You can also convert 13 sector diskettes to 16 sector by using the CONVERT13 program located on the System Master. CONVERT13 does not actually change the diskette's format, but

instead copies files from the 13-sector diskette onto a 16-sector diskette (previously initialized for 16 sectors using INIT under DOS 3.3).

Upon running the program (by executing `RUN CONVERT13`), a menu will appear on the screen. Option 1 starts the conversion process. You will be prompted for the source and destination slot and drive numbers as well as the filename. You should then insert the diskette per your prompt response and press any key to begin. Continue this process until all desired files on the 13 sector diskette have been transferred. When the process has been completed, you can reinitialize the old 13 sector diskette under DOS 3.3 for use as a 16 sector diskette.

A copy-protected 13-sector disk can also be used under DOS 3.3 by performing a two-disk startup. First of all, place the DOS 3.3 System Master in drive 1, close the drive door, and restart the system. Then, enter the following command:

```
RUN START13
```

The following will appear:

```
Slot to boot from (DEFAULT = 6)
```

Place the 13-sector diskette in drive 1 and press return if your disk controller card was installed in slot 6. Otherwise, enter the number of the slot where the controller card was installed.

### Prompts

Once DOS 3.3 has been loaded, either DOS, Integer BASIC, or Applesoft BASIC commands can be entered. The prompt displayed on the screen indicates to the user which commands the IIe will accept. When the Applesoft prompt ( `]` ) is displayed, either Applesoft BASIC or DOS commands can be entered. When the Integer BASIC prompt is displayed ( `>` ), either Integer BASIC or DOS commands can be entered.

DOS does not have its own prompt. You can check whether or not DOS has been loaded by entering a DOS command. If the command works, DOS has been loaded. If the command does not work, DOS has not been loaded with the IIe, DOS is loaded during start-up.

**Error Message Format -- DOS, Applesoft, Integer**

If an incorrect command is entered, an error message will be generated. The format of the error message will indicate whether that error was generated by DOS, Integer BASIC, or Applesoft BASIC.

If a question mark precedes the error message, the error was generated by Applesoft BASIC as indicated in the following example:

? SYNTAX ERROR ← Applesoft BASIC

If three asterisks precede the error message, the error was generated by Integer BASIC as shown below:

\*\*\* SYNTAX ERROR ← Integer Basic

If no prefix precedes the error message, it was generated by DOS. The following error message would be displayed under DOS:

SYNTAX ERROR ← DOS

**DOS COMMANDS**

In the following sections we will discuss the usage of the various Apple DOS commands. These include:

APPEND	INIT	READ
BLOAD	LOAD	RENAME
BRUN	LOCK	RUN
BSAVE	MAXFILES	SAVE
CATALOG	MON	UNLOCK
CLOSE	NOMON	VERIFY
DELETE	OPEN	WRITE
EXEC	POSITION	

Before discussing the individual commands, we will discuss topics related to all of these commands. These topics include:

    Filename  
    Drive Number Specification  
    Slot Number Specification  
    Volume Specification

## **Filenames**

In Apple's DOS, a file must be referenced using its filename.

DOS filenames must adhere to certain rules. First of all, a filename must be from 1 to 30 characters in length. Any characters in excess of 30 will be dropped (or truncated). The first character of the filename must be a letter. Any keyboard character except the comma can be used in a filename.

Control characters can be included in filenames, although they will not be displayed in catalog listings as they are nonprinting characters. For example, the following filename:

TEXT↑                      ↑ indicates pressing the *Control* key

would be displayed as TEXT in the catalog listing. However, if an attempt was made to load the file using the filename TEXT, DOS would not be able to locate the file.

Using control characters within filenames offers an effective security measure, as others will not be able to identify the actual filename.

## **Drive Specification**

Generally, DOS commands allow the user to specify several optional parameters including the diskette's volume number, the slot containing the disk controller, and the disk drive being used.

Each disk controller can control two separate disk drives. If a disk

drive is not specified in a DOS command, that command will affect the diskette in drive 1. In other words, drive 1 is the default drive.

If you wish the command to affect drive 2, end the command with a comma followed by D2. The following command would result in the directory for the diskette in drive 2 being displayed:

CATALOG, D2

Once D2 has been specified, all subsequent DOS commands will affect the diskette in drive 2. In other words, drive 2 will now be the default drive.

If you wish to change the drive affected by DOS commands back to drive 1, merely end your DOS command with a comma followed by D1. The default drive will then be drive 1.

### **Slot Specification**

If you wish to use more than two drives with your IIe, an additional controller card must be installed in one of the IIe's expansion slots. Generally, the second controller card is installed in slot 4. Again two drives can be connected to this second controller card.

If you wish to reference drives 3 and 4, you must use a second parameter known as a **slot** parameter. You cannot reference these drives as D3 and D4.

The slot parameter follows the DOS command just like the drive parameter. The slot parameter must be separated from the DOS command or drive parameter with a comma. The slot parameter consists of the letter S followed by the slot number containing the disk controller to be accessed.

The DOS command and any filenames must be specified first in the command. However, the order of the parameters (drive slot) is not critical.



For example, the following DOS command:

CATALOG,S4,D2

would access the second drive installed on the controller card installed in slot 4.

The default slot number will initially be the slot from which DOS was booted. Regardless of the slot used to boot DOS, DOS must always be booted from drive 1 rather than drive 2. If a slot parameter is subsequently executed, the specified slot will be the new default.

When using the slot parameter with DOS commands, be certain that the specified slot contains a disk controller card. The following error will appear if the specified slot does not contain a disk controller card.

#### I/O ERROR

DOS will then suppose that the specified drive (which is in fact not connected to the system) is still running. DOS will wait for this non-existent controller and drive to signal that it has stopped running. In other words, DOS will be locked up even if the correct slot is specified.

If you do not care to save the program in memory, recovery from this error condition can be attained by simply pressing Reset. If you wish to save the program in RAM, you can do so by resetting the default slot parameter to the correct slot number. You can do so by entering:

CATALOG, Sx

where x is a valid slot number.

#### **Volume Specification**

A volume number can be assigned to a diskette when it is initialized. The volume number is specified during initialization

using the following configuration:

Vx

where x can be any integer from 1 to 254.

If we wished to initialize a diskette in drive 2 attached to the controller slot 4 using the name "BEGIN" for the greetings\* program, we could do so by issuing the following command:

```
** INIT BEGIN, S4, D2, V100
```

This command would assign the diskette a volume number of 100. The volume number cannot be changed unless the diskette is reinitialized.

Again, the order of the disk, slot, and volume specifications is not important. If a volume specification is not indicated, the diskette will be assigned a volume number of 254.

If the volume number is specified with a DOS command, DOS will check the diskette's volume number to be sure that it corresponds with the volume number indicated with the DOS command's V parameter. If you inadvertently specify the wrong volume number or insert the wrong diskette, the following error message will be displayed:

VOLUME MISMATCH

The use of the volume specification can prevent a diskette from accidentally being overwritten.

The DOS CATALOG command ignores the volume specification. The volume number is displayed in the catalog listing.

---

\* This program is run when the diskette is booted.

\*\* INIT will be discussed in detail later in this chapter.

## Syntax

We will use the following syntax rules in our discussion of DOS commands.

DOS commands	indicated in uppercase characters.
DOS User Specified Parameters	indicated in lower-case italics.
<i>filename</i>	indicates a filename.
D	indicates the drive specifier (1 or 2).
S	indicates the slot number (1-7).
V	indicates the volume number (1-254).

These syntax conventions will be followed in the configuration section for each DOS command.

## CATALOG - Examining the Diskette's Directory

A diskette can contain a number of different programs or data files. Each file generates a listing in the diskette's **directory**. A diskette directory can be defined as a file on the diskette containing information relating to all other files stored on that diskette. A directory is also often referred to as a **catalog**.

The directory in Apple DOS can hold up to 105 file entries. The directory is stored on track 17 of the diskette, with the first entry in sector 15 and the last in sector 1. Each directory entry includes the filename, file type, number of sectors (mod 256) used by the file, and the location of the file's track sector list.

The CATALOG command can be used to display a diskette's directory. Merely enter CATALOG and press return and a listing similar to that in illustration 5-14 will appear. If your screen fills with entries and the flashing cursor appears at the bottom of the screen, only a portion of the directory was displayed. Press any key to continue the listing. When prompt appears (]), the listing will have been completed.

**Illustration 5-14. CATALOG Listing (System Master Diskette)**

```

] CATALOG
DISK VOLUME 254

*A 006 HELLO
*I 018 ANIMALS
*T 003 APPLE PROMS
*I 006 APPLESOFT
*I 026 APPLEVSION
*I 017 BIO RHYTHM
*B 010 BOOT13
*A 006 BRIAN'S THEME
*B 003 CHAIR
*I 009 COLOR DEMO
*A 009 COLOR DEMOSOFT
*I 009 COPY
*B 003 COPY.OBJ 0
*A 009 COPYA
*A 010 EXEC DEMO
*B 020 FID
*B 050 FPBASIC
*B 050 INTBASIC

```

Notice the first entry in the catalog listing is the disk volume number.

Each file on the diskette contains a separate listing. This listing identifies the type of data contained in the file, whether or not the file is locked, the number of diskette sectors used by the file, and the \*filename.

---

\*Filenames are discussed in the next section.

The one-letter code to the left of the listing identifies the file type. An A indicates an Applesoft program file, while I identifies an Integer BASIC program file. A T indicates a text file, and a B identifies a binary image file.

If a file is locked, its type code will be preceded by an asterisk (\*). If no asterisk appears, the file is unlocked. The concept of locked and unlocked files will be discussed later in this chapter.

The number of sectors occupied by the file is displayed as a 3-digit number to the right of the file type. For example, from Illustration 5-14, HELLO occupies 6 sectors.

The minimum size of a diskette file is 1 sector. If a file exceeds 255 sectors, the sector catalogue listing number will be reset to zero at 255 sectors. For example, a file with 287 sectors would be displayed as 032 in the catalog listing.

### **Track/Sector List**

One item also contained in the directory entry for a file (but not displayed by a CATALOG listing) is the location of the sector that contains that file's **track/sector list**.

A track/sector list consists of a list of the diskette locations used by the file. The contents of a track/sector list are summarized in Table 5-1.

If the track/sector list extends beyond 122 file sector identifiers, the links will point to one or more subsequent sectors of the track/sector list. Otherwise, the links will be set to 0.

**Table 5-1. Track/Sector List**

<b>Byte No. (Hex)</b>	<b>Byte Contents</b>
0	Unused
1	Link: Contains track number where track/sector list is continued.
2	Link: Contains sector number where track/sector list is continued. If both link bytes = 0, the track/sector list is not continued.
3-4	Unused
5-6	Sector base number (identifies groups of 122 sectors).
7-B	Unused
C	Track No. of First Sector in File
D	Sector No. of First Sector in File
E	Track No. of Second Sector in File
F	Sector No. of Second Sector in File
10	Track No. of Third Sector in File
11	Sector No. of Third Sector in File
:	
:	(Continuation)
:	
FE	Track No. of 122nd Sector in File
FF	Sector No. of 122nd Sector in File

**INIT**

Before a diskette can be written to, it must first be **initialized**. When a diskette is initialized, any existing data on the diskette is erased, DOS is copied to the diskette, and any program in memory is copied to the diskette.

The program copied should be the **greetings** program. This is the program which is automatically run when the diskette is booted.

The greetings program generally is assigned the filename HELLO, however, any filename can be used.

### Configuration

INIT *filename*\* [Sx, Vx, Dx]

*filename* refers to the name assigned to the greetings program (generally HELLO). S, V, and D are the slot, volume, and drive identifiers respectively.

The first step in initializing a diskette is to boot DOS. Once DOS has been booted, remove the System Master Diskette from drive 1 and replace it with the diskette to be initialized. Then, type NEW to clear the memory, and key in your greetings program.

The following is a typical greetings program:

```
100  REM GREETINGS PROGRAM 1
200  PRINT "SLAVE DISKETTE-DOS 3.3 64K SYSTEM"
300  PRINT "JW--JULY, 28, 1983"
400  END
```

The following information should be included in the greetings program:

---

\* *filename* is generally HELLO.

- Whether the diskette is a slave\* or master\*
- Size of the System on which the diskette was created
- Date on which the diskette was initialized

Once the greetings program has been entered, enter the following command:

INIT HELLO

After the INIT command has been entered, press the return key. The disk will spin for about a minute as the diskette is being initialized. When the initialization process has been completed, the Applesoft prompt ( ] ) will be displayed.

Finally, try booting your newly initialized diskette. Upon booting, the messages specified in your greetings program should appear on the screen.

### Master and Slave Diskettes

INIT initializes a diskette as a **slave** diskette. A slave diskette's DOS is memory size dependent. i.e. the diskette commonly can be used on a system with the same amount of RAM as the system on which the diskette was created.

A slave diskette can be converted to a **master** diskette by using the MASTER CREATE program on your System Master diskette. A master diskette's DOS is self-relocating. This allows memory to be allocated more efficiently.

The following procedure should be followed to convert a slave diskette to a master.

Step 1: Insert the slave diskette into the drive and run the greetings program.

---

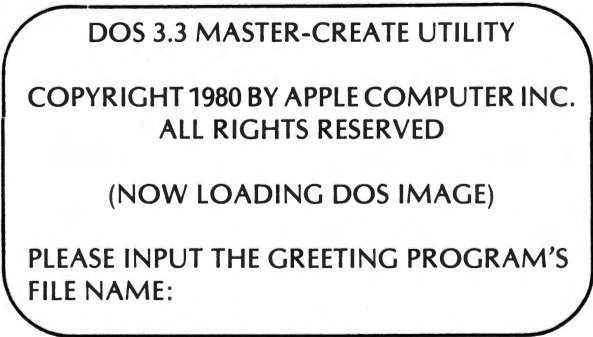
\*Discussed later in this section.



- Step 2: Edit the greetings program so that the initial message displayed by that program indicates that the diskette is a master. Then, save\* the edited greetings program on the slave diskette.
- Step 3: If you plan on changing the name of the greetings program (see the prompt on the display in Step 5), use\* **RENAME** at this point to change the program's name.
- Step 4: Remove the slave diskette and replace it with the System Master.
- Step 5: Boot DOS from the System Master and enter the following command from either Applesoft or Integer:

**BRUN MASTER CREATE**

The following screen display will then appear:



DOS 3.3 MASTER-CREATE UTILITY  
COPYRIGHT 1980 BY APPLE COMPUTER INC.  
ALL RIGHTS RESERVED  
  
(NOW LOADING DOS IMAGE)  
  
PLEASE INPUT THE GREETING PROGRAM'S  
FILE NAME:

- Step 6: Enter the program name used for the greetings program on the slave diskette (generally **HELLO**) and press the return key. The following screen will then appear:

---

\*The procedure for saving programs and renaming files will be discussed later in this chapter.

REMEMBER THAT MASTER DOES NOT CREATE THE GREETINGS PROGRAM, OR PLACE IT IN THE DISK DIRECTORY

THIS IS THE FILE NAME THAT WILL BE PLACED WITHIN THE IMAGE:

HELLO

PLACE THE DISKETTE TO BE MASTERED IN THE DISK DRIVE.

PRESS RETURN WHEN READY.

NOTE: IF YOU WANT A DIFFERENT FILE NAME, PRESS ESC .

- Step 7: Remove your System Master from the drive and replace it with the slave diskette being converted to a master. Press the return key to begin the conversion. The following message will be displayed when the conversion has been completed:

THE DISKETTE HAS BEEN UPDATED, YOU MAY REMOVE IT AT THIS TIME.

IF YOU WISH TO "MASTER" ANOTHER DISKETTE, PRESS [RETURN]

OTHERWISE PRESS ESC TO EXIT "MASTER"

- Step 8: After you have finished using MASTER CREATE, reboot DOS before entering any further commands.

The greetings program filename (specified in Step 5) is not actually placed in the diskette's catalog. This prompt merely identifies the program that DOS should run when the diskette is booted.

The newly created master diskette must contain a program with the filename specified in Step 5. If this is not the case, the following message will be displayed when you boot the diskette:

## FILE NOT FOUND

If you are changing the name of your greetings program, be sure to rename the program's filename as described in Step 3.

## LOAD

LOAD is used in DOS to load a program file from diskette into memory.

### Configuration

LOAD *filename* [Sx, Vx, Dx]

The following example would load the program file named PROGA from diskette into memory:

LOAD PROGA

If the specified *filename* cannot be located in the diskette directory, the following message will be displayed:

FILE NOT FOUND

If the file is located on the directory, DOS will check to be sure that the file is a program file. If the specified file is not a program file, the following message will be displayed:

FILE TYPE MISMATCH

If the specified file is located in the directory and it is a program file, the current program in RAM will be erased and the specified program file will be copied from the diskette file into RAM. That program can then be edited, renamed, listed, or run.

## SAVE

The SAVE command is used in DOS to save a program from RAM onto diskette.

### Configuration

SAVE *filename* [Sx, Vx, Dx]

When SAVE is executed, the specified drive will begin spinning and the in use lamp will light. This indicates that the program in RAM is being saved on diskette. When the drive stops spinning and the in-use light goes off, the prompt will reappear. The program will have been saved on diskette.

If a *filename* is specified with SAVE that is identical to a filename already on diskette, the file being saved will replace the file on diskette. The original file will be erased.

### DELETE

The DELETE command can be used to remove a file from a diskette.

### Configuration

DELETE *filename* [, Sx, Vx, Dx]

### RENAME

The RENAME command can be used to change the name of a diskette file.

### Configuration

RENAME *oldfilename*, *newfilename*

The file specified by *oldname* will be renamed as specified in *newname*. If *newfilename* specifies a filename already on diskette, DOS will still change the filename as indicated by RENAME. As a result, two files could be located on the same diskette with the same filename. To prevent confusion, avoid using the same filename for files located on the same diskette.

If a file is locked, it cannot be renamed. Exercise caution when

renaming the greetings program. If the greetings program is renamed, a new program should be saved on the diskette with the same filename as the original greetings program as DOS will continue to search for the greetings program under its original filename.

## **LOCK**

**Locking** is a feature which prevents a file from being inadvertently deleted or overwritten.

### **Configuration**

LOCK *filename* [, Sx, Vx, Dx]

Locked files are indicated on the catalog listing with an asterisk. Any attempt to DELETE or RENAME a file will result in the following error message:

FILE LOCKED

Also, if an attempt is made to SAVE a file with a filename identical to that of a locked file, the following message will appear (if both files are in the same language).

FILE LOCKED

If an attempt is made to SAVE a file using the name of a filename of a locked file in a different language, the following message will appear:

FILE TYPE MISMATCH

## **VERIFY**

Sometimes, information may not be recorded correctly on the diskette. The VERIFY command can be used to confirm that a file was correctly copied.

### Configuration

VERIFY *filename* [, Sx, Vx, Dx]

VERIFY uses the **checksum** value for each sector to determine whether or not that sector was copied correctly. The checksum value is calculated using the numeric value of each character in the sector, and is stored in the sector.

VERIFY recomputes the checksum for each sector in the file and then compares these computed checksums with those stored in the sector. If there are any discrepancies, the following error message will appear:

I/O ERROR

If the computed and stored check sums agree, the BASIC prompt will be displayed.

### MON & NOMON

MON is an abbreviation for MONITOR. The MON command is used to display information related to disk input and output. The NOMON command will turn the display off again.

MON is useful when debugging a program. Generally, data transferred between the screen and disk drives is not displayed. MON allows this data to be displayed, which can be helpful during the debugging process.

### Configuration

MON [ C ] [ , I ] [ , O ]

NOMON [ C ] [ , I ] [ , O ]

C indicates that commands to the disk are to be monitored. (ex. OPEN, READ, WRITE, etc).

I indicates that input from the disk is to be monitored (ex. when a file is being read).

O indicates that output to the disk is to be monitored (ex. when a file is to be written to).

C, I, and O can be specified in any order and must be delimited with commas. At least one of these three parameters must be included with either MON or NOMON for the command to have an effect. A MON remains in effect until either a NOMON, INT, FP, or DOS reboot are undertaken.

The default is NOMON C, I, O. This command indicates that no monitoring is to occur.

The different combinations of C, I, and O with MON are summarized in Table 5-2.

**Table 5-2. MON Command Summary**

<b>MON Command</b>	<b>Input and/or Output Monitored</b>
MON C, I, O	Disk input. Disk output. Commands to disk.
MON C, I	Disk input. Commands to disk.
MON C, O	Disk output. Commands to disk.
MON I, O	Disk input. Disk output.
MON O	Disk output.
MON I	Disk input.
MON C	Commands to disk.

**MAXFILES**

MAXFILES allows the users to specify the number of files which can be open at any one time. A maximum of 16 files can be open at any one time.

**Configuration**

MAXFILES x

x is an integer from 1 to 16. The default value for x is 3.

Each file specified by MAXFILES has 595 bytes of memory reserved as a buffer. One 256-byte section of the buffer is used for reading and a second 256-byte is used for writing. The remaining 83 bytes are used for housekeeping information.

One file buffer is required for the execution of each of the following DOS commands:

APPEND	EXEC	OPEN
BLOAD	FP	POSITION
BRUN	INIT	READ
BSAVE	INT	RENAME
CATALOG	LOAD	RUN
CHAIN	LOCK	SAVE
CLOSE	MON	VERIFY
DELETE	NOMON	WRITE

If the number of open disk files is at the limit set by MAXFILES and one of the aforementioned DOS commands is executed, the following error message will appear:

NO BUFFERS AVAILABLE

MAXFILES should be executed prior to loading a program as its execution in the immediate mode causes the erasure of Integer BASIC programs as well as difficulties with strings in Applesoft.



If MAXFILES is executed within a program, memory pointers will be altered. This can cause problems with GOSUB, GOTO, and other instructions. If you need to execute a MAXFILES command within an Integer BASIC program, do so using an EXEC file (discussed later).

If you need to execute a MAXFILES command in an Applesoft BASIC program, the command should be the first statement in the program and should be preceded with a Control-D character as illustrated below:

```
100 PRINT CHR$(4); "MAXFILES 6"
```

## EXEC

The EXEC command allows for the execution of an EXEC file.

### Configuration

```
EXEC filename, [ Rx, Sx, Vx, Dx ]
```

R indicates the relative field number within the EXEC file where execution is to begin. R0 is the default value which causes execution to commence at the beginning of the EXEC file. If R1 was specified, execution would begin with the file's second record.

When the EXEC command is executed, the indicated EXEC file is first opened. Then, the line specified by the R parameter is read and executed as if it had been entered via the keyboard in the immediate mode.

If the line is a valid program line, it will be stored in memory--just as if it had been entered in the immediate mode. If the line is a command such as LIST, CATALOG, SAVE, or RUN, it will be executed.

A sample EXEC file is listed below:

```

100 X = 0
200 FOR X = 1 TO 3
300 PRINT X
400 NEXT
500 END
RUN
NEW
CATALOG

```

Once this EXEC file has been created (as explained in the next section) with the filename SAMPLE, it can be executed by issuing the following:

```
EXEC SAMPLE
```

When this EXEC command is executed, the following will occur:

- A simple program will be entered into memory, and executed.
- The diskette's directory will be listed by CATALOG.

### Creating an EXEC File

First of all, a program must be written to create an EXEC file. An example of such a program is given below:

```

10 REM CREATE EXEC
20 D$ = CHR$(4):REM CTRL-D
30 PRINT D$; "OPEN SAMPLE"
40 PRINT D$; "WRITE SAMPLE"
45 PRINT "FP"
60 PRINT "100 X = 0"
70 PRINT "200 FOR X = 1 TO 3"
80 PRINT "300 PRINT X"
90 PRINT "400 NEXT"
100 PRINT "500 END"
110 PRINT "RUN"
120 PRINT "NEW"
130 PRINT "CATALOG"
140 PRINT D$; "CLOSE SAMPLE"

```

This program (named CREATE EXEC) will create an EXEC file named SAMPLE. Note on line 130 that the DOS command CATALOG need not be preceded by CTRL-D. This is the case with other DOS commands as well.

Once CREATE EXEC has been saved on diskette, the following command:

RUN CREATE EXEC

will result in the creation of an EXEC file named SAMPLE. This file can then be executed with the following command:

EXEC SAMPLE

## **BSAVE**

DOS allows the usage of binary files as well as BASIC program files. Binary files are denoted with the letter B in catalog listings. BSAVE is used to save a binary file on disk.

### **Configuration**

BSAVE *filename*, Axxx, Lxx, [ Sx, Vx, Dx ]

A indicates the beginning address in memory of the data to be stored on diskette. If A is specified in hexadecimal notation, it should be preceded with the dollar sign (\$). The address specified with A (xxx) should correspond to an actual memory address in the IIe.

L indicates the length in bytes of the portion to be saved. The maximum number of bytes that can be saved is 32767.

## **BLOAD**

BLOAD loads the contents of a binary file back into memory.

### Configuration

**BLOAD** *filename* [ , *Axxx* ] , *Lxxx* , [ *Sx* , *Vx* , *Dx* ]

The address parameter (A) is optional in BLOAD. If A is not specified, the file will be loaded at the address indicated when the file was saved.

### BRUN

BRUN performs a BLOAD, after which it performs a machine language JMP instruction to the address indicated by A. This will begin execution of the machine language program.

### Configuration

**BRUN** *filename* [ , *Axxx* ] , *Lxxx* , [ *Sx* , *Vx* , *Dx* ]

### Using DOS Commands in Programs

In certain situations, you may wish to use DOS commands within BASIC programs. A DOS command can be included within a BASIC program by placing the command within a PRINT statement and prefixing the command with the Control-D character (ASCII Code 4).

An example of the usage of a DOS command within an Applesoft program is given below:

```
10 D$ = CHR$(4)
20 PRINT D$; "CATALOG"
```

The preceding program will generate a catalog listing.

Note that we defined D\$ as Ctrl-D using the CHR\$ function. Since actually entering Ctrl-D via the keyboard does not generate a printing character, this is the preferred method for including Ctrl-D in a BASIC statement.

## SEQUENTIAL & RANDOM FILE ACCESS

Diskettes can be used to store information other than a program. For example, a diskette file can be used to store names and addresses on a mailing list, the result of a formula, a letter, or any other form of data. Such a file is known as a **text** file (or data file). Text files are denoted in the catalog listing with the letter T.

Text files are created, read from, or written to using the following DOS commands in an Applesoft or Integer BASIC program.

OPEN	APPEND
CLOSE	POSITION
READ	EXEC
WRITE	

OPEN, READ, WRITE, APPEND, and POSITION can only be used in the program mode. If an attempt is made to use these commands in the immediate mode, the following error message will appear:

NOT DIRECT COMMAND

CLOSE and EXEC may be used in the immediate mode.

Two types of text files are used, sequential text files and random text files.

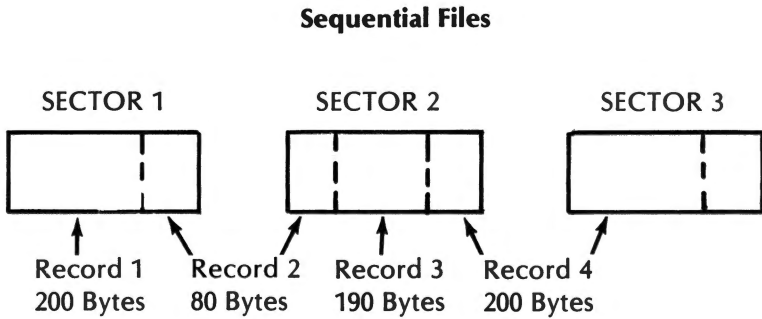
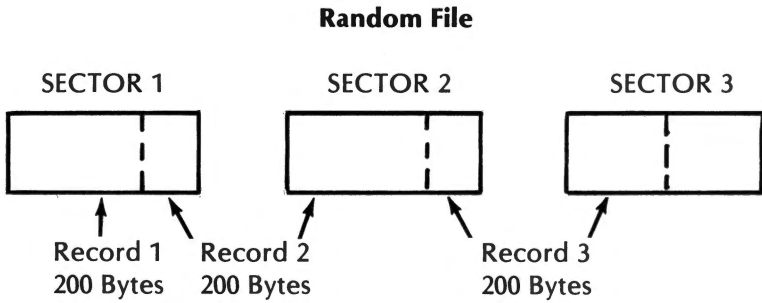
Each record of a sequential disk file is assigned exactly as much disk space as it requires. There are no blank spaces between records in a sequential file.

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 Illustration 5-15. Notice that the length of each record in the random file is constant at 100 bytes.

The record length of a sequential file is variable. The record length is the sum of the total space used by all the fields in each individual record.

### Illustration 5-15. Random and Sequential Data Files



**\*Example assumes 256 bytes per sector.**

Another difference between random and sequential files is the way in which each file is accessed. **Direct Access** of any record in a random file is possible regardless of that record's location within 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.

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.

### Opening Sequential Files

A diskette file must be opened before it can be accessed by a program. When a file is opened, DOS will check to see if the specified file is on disk, and if so, where it is located. OPEN also reserves a 595-byte buffer for input and output to the file.

### Configuration

OPEN *filename* [ , Sx, Vx, Dx ]

If the specified *filename* is not present on the diskette, DOS will create an entry for that *filename* in the diskette's directory.

OPEN must be specified within quotes in a PRINT statement. OPEN must be preceded with Ctrl-D. In the following statements, a filename TEXTA is created on the default drive:

```
100 D$ = CHR$(4)
200 PRINT D$; "OPEN TEXTA"
300 END
```

### Writing to Sequential Files

Information is sent to the diskette via the PRINT statement just as data is sent to the display or printer. However, before data can be

sent to a diskette file with PRINT, the DOS WRITE command must be executed. WRITE notifies DOS that the PRINT statement is being used to send data to a diskette file rather than to the display.

### Configuration

WRITE *filename*

After the WRITE command has been executed, subsequent PRINT statement output will be sent to the specified diskette file.

If an error is encountered, the error message will be sent to the diskette file. However, once an error message has been sent to the diskette file, the WRITE command will be cancelled.

As shown in the following example, WRITE must be issued with a PRINT statement and preceded with Ctrl-D.

```
100 D$ = CHR$(4)
200 PRINT D$; "WRITE TEXTA"
300 PRINT "THIS DATA IS BEING SENT TO THE DISKETTE"
```

When data is printed to a file, DOS will update an internal file pointer. This pointer denotes the position on the diskette surface where the next data items will be stored. The file pointer for a sequential data file can only be moved forward. The OPEN statement moves the file pointer to the beginning of the file.

When you are writing to a sequential data file with existing data, problems can occur unless the file is first erased. This is due to the fact that although PRINT statements will overwrite the existing file contents, the previous data may extend beyond the end of the new data.

This potential problem can be avoided by using the DELETE command prior to the OPEN command as shown in the following example:



```

100 D$ = CHR$(4)
110 PRINT D$; "OPEN TEXTA"
120 PRINT D$; "DELETE TEXTA"
130 PRINT D$; "OPEN TEXTA"
140 PRINT D$; "WRITE TEXTA"
150 PRINT "THIS IS THE NEW DATA"
160 PRINT D$; "CLOSE"
170 END

```

The DELETE command in line 120 will erase the previous contents of TEXTA. Notice the OPEN command in line 110. This command is issued to prevent a possible error in line 120. If the DELETE command was issued in line 120, and TEXTA was not present on the diskette, the following error message would be displayed:

FILE NOT FOUND

The OPEN command in line 110 will create TEXTA in case it did not already exist on the diskette.

The WRITE command must be cancelled should the user wish to resume sending output to the screen or printer rather than to the disk drive. Any DOS command will cancel WRITE. The safest method of cancelling WRITE is to issue the **null** command. The null command consists of the Ctrl-D character. The following program line would cancel WRITE:

165 PRINT D\$

assuming D\$ had been assigned the ASCII code for Ctrl-D.

### Reading Sequential Files

DOS allows the user to read data from the text file as well as to write data to it. The READ command specifies a disk file as the source for data input via the INPUT and GET (Applesoft only) statements.

#### Configuration

READ *filename*

The READ statement must be issued in a PRINT statement and must be preceded with the Ctrl-D character. Once a READ command has been issued, INPUT statements will accept data from the file specified in *filename* until a subsequent DOS command or an error cancels READ.

The following illustrates the use of READ:

```

100 D$ = CHR$(4)
110 PRINT D$; "OPEN TEXTA"
120 PRINT D$; "READ TEXTA"
130 INPUT X$
140 PRINT X$
150 PRINT D$; "CLOSE TEXTA"
160 END
] RUN
THIS IS THE NEW DATA
]
```

In Applesoft, the GET statement can also be used to read data from a disk file. Unlike INPUT, GET returns just one character. Therefore, if our text file included the following phrase:

THIS IS THE NEW DATA

after an OPEN and READ were executed, the first GET statement would return the letter T, the second GET statement would return the letter H, etc.

The following program illustrates the use of GET to read data from a text file:

```

100 D$ = CHR$(4)
110 PRINT D$; "OPEN TEXTA"
120 PRINT D$; "READ TEXTA"
130 X$ = " "
140 GET Y$
150 IF Y$ = CHR$(13) THEN 180
160 X$ = X$ + Y$
170 GOTO 140
```

```

180 REM RETURN CHARACTER ENCOUNTERED
190 PRINT CHR$(1); X$
200 END

```

Notice the inclusion of the CHR\$(1) prior to X\$ in line 190. This character is included in the PRINT statement because GET ignores the first character that is printed following its execution. If this character is Ctrl-D, the DOS command that follows will be printed rather than executed as a DOS command. The CHR\$(1) in the PRINT statement in line 190 can be ignored as it has no special meaning. This allows the program to function properly.

### **Closing a Sequential File**

Once a file has been opened, it should be closed. Failing to close a file can result in the loss of data in the file left open as well as possible loss of data on another diskette.

#### ***Configuration***

CLOSE [ *filename* ]

When CLOSE is used without a *filename* parameter, all open files on all diskettes will be closed. If *filename* is specified with CLOSE, the specified file will be closed.

### **APPEND**

The DOS APPEND command allows the user to place the file pointer for a sequential file at the end of the last data item. Since the file pointer is reset after a file has been closed and reopened, APPEND is a useful command--especially in situations where the user wishes to add text to the end of a sequential file.

#### ***Configuration***

APPEND *filename* [ , Sx, Vx, Dx ]

The APPEND command will place the file pointer to the first character position beyond the end of the file. If data is written to

the file after an APPEND has been executed, that data will be written immediately after existing file data. If an attempt is made to read data from the file after an APPEND has been executed, the following error message will be displayed:

END OF DATA

APPEND performs an OPEN on the specified file if that file already exists. If the specified file does not exist, the following error message will be displayed:

FILE NOT FOUND

In other words, unlike OPEN, APPEND will not create a file.

## POSITION

POSITION can be used to move the file pointer to any given field in the sequential file.

### Configuration

POSITION *filename* [ , Rxx]

*filename* specifies the file whose file pointer is to be moved. R indicates the relative field position, and xx indicates the number of fields to be moved forward. (POSITION can only be used to move the file pointer forward).

If  $xx = 0$ , then any READ or WRITE operation will be undertaken in the current field. If  $xx = 1$ , the current field will be skipped over, and any READ or WRITE operations will be undertaken in the next field. If  $xx = 2$ , the current field and the following field will be skipped, before any READ or WRITE operation will occur.

A file must have been OPEN'ed before POSITION can be executed. Since OPEN sets the file pointer to the beginning of the file, if POSITION is executed immediately after OPEN, the Rxx parameter will select the absolute field position within the file. In other words, the relative file position (selected by R) will

correspond to the absolute file position. This only occurs when the file pointer is positioned at the beginning of the file.

POSITION like all DOS commands will cancel a READ or WRITE command. For this reason, execute POSITION prior to the execution of READ or WRITE.

POSITION functions by examining the file byte by byte. When POSITION encounters the ASCII code for return, it assumes that the current field's ending point has been reached. If POSITION encounters an unused byte, it will assume that the field requested does not exist. The following error message will be displayed:

END OF DATA

The following program illustrates the use of POSITION:

```

100 D$ = CHR$ (4)
110 A$ = "JOHN": B$ = "BILL": C$ = "JACK"
120 PRINT D$; "OPEN TESTB"
130 PRINT D$; "WRITE TESTB"
140 PRINT A$:PRINT B$: PRINT C$
150 PRINT D$; "CLOSE TESTB"
160 PRINT D$; "OPEN TESTB"
170 PRINT D$; "POSITION TESTB, R2"
180 PRINT D$; "READ TESTB"
190 INPUT Z$
200 PRINT Z$
210 PRINT D$; "CLOSE TESTB"
220 END
]RUN [Ret]
JACK

```

### Storing Data in Disk Files

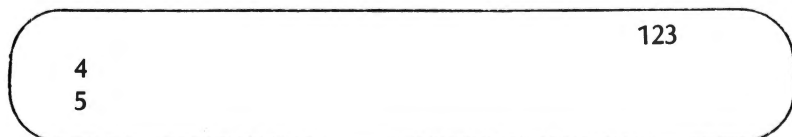
Caution must be exercised when storing numeric values in disk files. For example, if the following program was executed in Applesoft BASIC:

```

050 A = 0: B = 0
100 D$ = CHR$(4)
110 PRINT D$; "OPEN TEST C"
120 PRINT D$; "WRITE TEST C"
130 PRINT 1,2,3,CR,4,5
140 PRINT D$; "CLOSE TEST C"
150 PRINT D$; "OPEN TEST C"
160 PRINT D$; "READ TEST C"
170 INPUT A
180 PRINT A
190 INPUT B
200 PRINT B
202 INPUT C
204 PRINT C
210 PRINT D$; "CLOSE TEST C"
220 END
RUN

```

The following display would appear:



```

123
4
5

```

Notice from lines 170 and 180 that the first three of the PRINT statement parameters (in line 130) were concatenated to form a single number (123). The next output line consisted of the fourth PRINT statement (4). The final output line consisted of the fifth PRINT statement parameter.

This situation arises because of the format in which information is stored on disk files. When a PRINT statement is directed to the display, if commas are used to delimit parameters, each parameter is displayed at the next tab position on the screen.

However, when PRINT statement output is sent to a disk file, the commas are ignored. Therefore, all parameters are concatenated until the carriage return character (ASCII 13) is encountered. In DOS, the carriage return character is the only character which can be used to separate values.

In Integer BASIC, a carriage return character is output after the fifth tab stop. In Applesoft BASIC, a carriage return character is output after the third tab stop. In our Applesoft program example, a carriage return was inserted as indicated. This caused the PRINT statement output in line 130 to be concatenated as two separate numeric values.

Whenever you wish to output a number of items via the PRINT statement to different fields in the disk file, either separate PRINT statements should be used or each PRINT statement parameter should be delimited with the return code character. This is illustrated in the following program lines:

```
130 PRINT 1:PRINT 2:PRINT 3:PRINT 4:PRINT 5
```

or

```
100 D$ = CHR$(4):E$ = CHR$(13):REM CTRL-D  
    IS ASCII 4; RETURN IS ASCII 13
```

```
130 PRINT 1; E$; 2; E$; 3; E$; 4; E$; 5
```

If you substitute these lines in the example, the following output will appear:



```
1  
2  
3
```

## Opening and Closing A Random Access File

The following configuration is used to open a random access file:

### Configuration

OPEN *filename*, Lxx [ , Sx, Vx, Dx ]

L is known as the length parameter. L is a required parameter which denotes the length of each random access record. L can range from 1 to 32767.

The CLOSE statement is used to close a random access file just as it is used to close a sequential file.

## Reading and Writing to Random Files

The following configuration is used to read from or write to a random access file.

### Configuration

READ *filename*, Rxx [ , Sx, Vx, Dx ]

WRITE *filename*, Rxx [ , Sx, Vx, Dx ]

R is known as the **record** parameter. R moves the file pointer to the beginning of the specified record.

The following program illustrates reading from and writing to a random file.

```

10 D$ = CHR$ (4)
20 PRINT D$; "OPEN RANDOM, L20"
30 PRINT D$: "WRITE RANDOM,R2"
40 PRINT "RECORD NUMBER TWO"
50 PRINT D$; "WRITE RANDOM, R3"
60 PRINT "THIS IS RECORD 3"
70 PRINT D$; "WRITE RANDOM, R1"
80 PRINT "RECORD 1"
90 PRINT D$; "WRITE RANDOM, R4"
100 PRINT "RECORD 4"
110 PRINT D$; "CLOSE RANDOM"
120 GOTO 180
130 PRINT D$; "OPEN RANDOM, L20"
140 PRINT D$; "READ RANDOM, R"; R

```



```
150 INPUT X$
160 PRINT D$; "CLOSE RANDOM"
170 RETURN
180 INPUT "ENTER RECORD NUMBER "; R
190 GOSUB 130
200 PRINT X$
210 GOTO 180
```

### **Byte Parameter**

The B or byte parameter can be used with the READ, WRITE, and POSITION statements to move the file pointer to a specified byte within a sequential file or a specified byte within a specified field in a random file.

For example, the following command:

```
WRITE TEXTA, B20
```

would set the file pointer to the twenty-first byte in the sequential file named TEXTA. The first byte is byte 0. Any characters output to the disk file by subsequent PRINT statements will replace existing characters in TEXTA beginning with byte 21.

If R is specified, the B parameter sets the file pointer to the specified byte in the record indicated by R. In the following example:

```
READ TEXTA, R20, B4
```

the read operation will begin at the fifth byte in the twenty-first record.

When using the byte parameter, the user must know exactly what data has been stored in the file. Remember that spaces, commas, return codes, and all characters are stored in byte positions within a field.

## CHAPTER 6. APPLE IIe GRAPHICS

---

The Apple IIe has three different display modes — one text mode and two graphics modes. These formats may be combined into five different display formats. Both of the graphics modes are color capable. These two modes will be discussed in this chapter.

The Apple IIe can generate some of the finest color graphics available on a medium priced computer system (\$1000-\$2000). This is quite amazing, in light of the fact that the Apple was one of the first color capable home computers.

### LOW RESOLUTION GRAPHICS

The low resolution graphics mode is used in two of the previously mentioned display formats. A graphics-only display is available that has a resolution of 40 x 48 pixels.\* Also a graphics plus text display is available that has a resolution of 40 x 40 pixels. In this mode, four lines of text are located beneath the graphics display.

### Commands

The GR command is used to call the low resolution graphics mode. This command configures the computer hardware to display the graphics plus text format. This command also clears the display memory so that the screen will initially be black. The GR command can be used in a program or executed directly from the keyboard, as can all graphics commands.

---

\*A pixel can be defined as a single screen coordinate.

GR                    set low resolution  
                         graphics & text

Besides clearing the screen, the GR command also sets the low resolution "next color" register to 0. This register stores the numeric value of the next color to be displayed.

The four lines of text that appear beneath the graphics mode can be replaced with 8 more rows of the graphics mode. This extends the display resolution to 48 x 40 pixels. This switch can be accomplished by reading or writing to memory location 49234. The easiest means of accomplishing this is by poking the location.

POKE 49234,0      full screen graphics

To replace the four lines of text without erasing any graphics information on the screen, read or write to memory location 49235. Again, this can be accomplished by poking the location.

POKE 49235,0      graphics & text  
                         (no erase)

Before any graphics information can be plotted on the screen, a color must be selected. This is accomplished through use of the COLOR command. The correct syntax of this command is as follows:

COLOR = x

x represents a number between 0 and 255. The color corresponding to x modulo 16 is selected. For example, if x = 2 or 18 then color number two is selected. The colors and their associated numbers are listed in Table 6-1.

**Table 6-1 Color Numbers.**

0- black	8- brown
1- magenta	9- orange
2- dark blue	10- grey
3- purple	11- pink
4- dark green	12- green
5- grey	13- yellow
6- medium blue	14- aqua
7- light blue	15- white

After a COLOR has been selected, information can be plotted to the screen. This is accomplished by using the PLOT command. The correct syntax of this command is as follows:

PLOT x,y

x is the column number. y is the row number. The column numbers extend from 0 (left) to 39 (right). The row numbers extend from 0 (top) to 47 (bottom). For example the following program plots two orange squares in the upper righthand corner of the screen.

10	GR	→	set low resolution mode
20	COLOR = 9	→	select ORANGE = 9
30	PLOT 39,1	→	plot first square
40	PLOT 38,0	→	plot second square

If a graphics & text format is being displayed, plotting to a row number between 40 and 47 will cause a character to be displayed at that location.

VLIN and HLIN can be used to plot consecutive pixels. VLIN and HLIN are abbreviations for Vertical LINE and Horizontal LINE respectively. The correct syntax for the VLIN command is as follows:

VLIN y<sub>1</sub>,y<sub>2</sub> AT x

$y_1$  and  $y_2$  represent the range of row numbers.  $x$  represents the column number. The following VLIN command will plot every pixel in column 30 from row 4 to row 24.

VLIN 4, 24 AT 30

The correct syntax for an HLIN command is as follows:

HLIN  $x_1, x_2$  AT  $y$

$x_1$  and  $x_2$  represent the range of column numbers.  $y$  represents the row number. The following HLIN command will plot every pixel in row 14 from column 2 to column 37.

HLIN 2, 37 AT 14

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 SCRN takes as its arguments the row and column numbers, and returns the color number. The correct syntax of the SCRN command is as follows:

$X = \text{SCRN}(x, y)$

$x$  represents the column number.  $y$  represents the row number. Upon execution of the preceding command,  $X$  will be assigned the value of the color at screen location column  $=x$ , row  $=y$ .

## Uses of Low Resolution Graphics

The use of low resolution graphics is usually limited to drawings and simple charts. This is because low resolution graphics is not well-supported in BASIC.

### Charts

Simple bar charts are easily implemented by using low resolution graphics. The text window may be used for documentation or labels. For example, the following program displays a monthly sales chart.

```

10 DIM A(12)
20 FOR I = 1 TO 12
30 A(I) = 30 * RND(1)
40 NEXT I
50 GR:HOME
60 PRINT "MONTH: J F M A M J J A S O N D"
70 FOR I = 1 TO 12
80 COLOR = I
90 VLIN A(I),39 AT 5 + 2 * I
100 NEXT I

```

Lines 10-40 initialize the array A with random values. Line 50 clears the entire screen. Line 60 prints the headings on the column. Lines 70-100 plot the bar charts.

### Writing a Game Program

In this section, the game, "BARACADE" will be designed. The object of the game is to avoid the baracades 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 BARACADE, he may page through the chapter. All program lines may easily be distinguished from the rest of the text. To play BARACADE, merely enter every line belonging to the program.

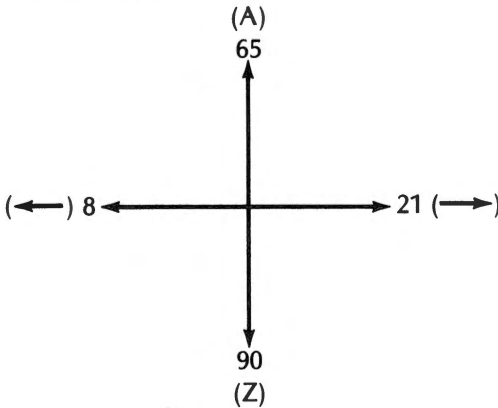
The first step in designing "BARACADE" is to program the computer to draw a trail. The following statements accomplish this.

```

10 GR: HOME
140 Y = 20 : X = 20 : A = 65
150 COLOR = 4
155 GET A$
160 IF PEEK(-16384) > 127 THEN GET A$ : A = ASC(A$)
170 IF A = 8 THEN X = X-1
180 IF A = 21 THEN X = X+1
190 IF A = 65 THEN Y = Y-1
200 IF A = 90 THEN Y = Y+1
220 PLOT X,Y
250 GOTO 160

```

Line 10 clears the screen and enables low resolution graphics. Line 140 sets the initial position at screen location (20,20). This is the center of the screen. The A = 65 statement that also appears in line 140 selects "up" as the initial direction of movement.



These values represent the ASCII coded values for the controller keys: A, Z, → and ←.

Lines 160-250 set up a loop that monitors the keyboard and acts accordingly. Line 160 checks for a keypress and sets the direction variable, A. The expression PEEK (-16384) > 127 is true whenever a key has been pressed. Lines 170-200 recalculate the position variables. Executing the program is the best way to understand how it operates. By the way, the program will not operate correctly if an 80-column card is presently active. See Chapter 8 for instructions about how to deactivate the card.

Recall from our description of BARACADE, that one of the rules was that the player was not allowed to collide with his trail. The SCRN function will be used to check for collision. If the following line is added to the program, collisions will be detected.

```
210 IF SCRN (X,Y) > 0 THEN 260
```

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.

```

260 PRINT "COLLISION"
270 FOR J = 1 TO 12
280 PRINT CHR$(7);:NEXT J
290 INPUT A$
300 GOTO 10

```

The PRINT statement in line 280 activates the consol speaker. Line 290 delay's a new game from starting until return has been pressed.

A BARACADE's score can be easily kept track of by merely incrementing a variable each time a move has been completed.

```

130 I = 0
230 I = I + 1
240 HOME: PRINT I

```

A final technicality remains — creation of a playing field. A square field with scattered barriers was chosen.

```

20 COLOR = 9
30 VLIN 0,39 AT 0
40 VLIN 0,39 AT 39
50 HLIN 0,39 AT 0
60 HLIN 0,39 AT 39

```

Line 20 sets the playfield color to orange (9). Lines 30-60 draw a square border around the field.

```

70 FOR K = 1 TO 6
80 GOSUB 310
90 HLIN L,L + 5 AT M
100 GOSUB 310
110 VLIN L,L + 5 AT M
120 NEXT K

```

The previous addition to the program draws randomly located barriers around the playfield. The subroutine at 310 must pro-



vide random values for the variables L and M, which have a range from 0 to 34 inclusive.

```
310 L = 34*RND(1)
320 M = 34*RND(1)
330 RETURN
```

The ideas in this section by no means exhaust the possibilities that could be added to "BARACADE." 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.

## HIGH RESOLUTION GRAPHICS

The Apple can also be configured to display one of two high resolution graphics formats. A graphics-only display is available that has a resolution of 280 x 192 pixels. Also, a graphics plus text display is available that has a resolution of 280 x 160 pixels. In this mode, four lines of text are located beneath the graphics display.

### Commands

All high resolution graphics commands directly parallel their low resolution counterparts. Therefore, familiarity with the low resolution commands will be assumed throughout the remainder of this chapter.

The HGR command can be used to configure the computer to display the graphics plus text format. The command clears the display memory (2000-4000 Hex), so that the screen will initially be black.

```
HGR      set high-resolution
          graphics + text
```

The HGR command does not clear the high resolution "next color" register.

The four lines of text that appear beneath the graphics mode can

be replaced with 32 more rows of the graphics mode to increase the display resolution to 280 x 192 pixels. This is accomplished in the same manner as was a similar switch using low resolution graphics.

```
POKE 49234,0    full screen graphics
POKE 49235,0    graphics + text
                  (no erase)
```

A second high resolution command is:

```
HGR2    set high resolution graphics only.
```

This command clears then displays high resolution page 2. Page 2 is located at memory addresses 4000-6000 Hex. Use of this command has one advantage over HGR — more memory space is available for program use. HGR2 has the disadvantage that it does not easily support a graphics plus text display. As a side note — if the command GR is issued while HGR2 is in effect, low resolution page 2 will be displayed. BASIC does not support this display mode. Enter TEXT to recover from this error.

An HCOLOR command must be used before any graphics may be output to the high resolution display. This command selects the color that will be displayed next. The correct syntax of this command is as follows:

```
HCOLOR = x
```

x represents a number between 0 and 7. Each number corresponds to a specific color. This information is contained in Table 6-2.

**Table 6-2.**

0- black	4- black
1- green	5- orange
2- purple	6- blue
3- white	7- white

High resolution graphics has a single command that can be used to output data to the screen. HPLOT is more flexible than PLOT, HLIN, and VLIN combined. In HPLOT's simplest form, it functions as low resolution's PLOT command.

HPLOT  $x,y$

$x$  is the column number.  $y$  is the row number. The column numbers extended from 0 to 279. The row numbers extend from 0 to 191. The computer stores the column number and row number of the last plotted coordinate.

The next form of the command is as follows:

HPLOT TO  $x,y$

The computer executes this command by drawing a straight line from the last plotted point to the point with coordinates  $(x,y)$ . The line color will be the last color selected by the HCOLOR command. These two versions of the HPLOT command may be combined as follows:

HPLOT  $x_1,y_1$  TO  $x_2,y_2$

This command will cause a line to be drawn from screen coordinate  $(x_1,y_1)$  to screen coordinate  $(x_2,y_2)$ .  $x_2$  and  $y_2$  will be stored as the last plotted coordinate.

The reserved word TO may appear more than once in a single HPLOT command.

HPLOT 1,1 TO 20,100 TO 200,30

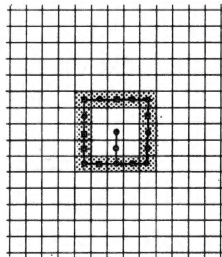
The previous command will initially cause a line to be drawn from (1,1) to (20,100). A second line will then be drawn from (20,100) to (200,30). Finally, the value 200 will be stored as the column number, and the value 30 will be stored as the row number.

### Shape Table

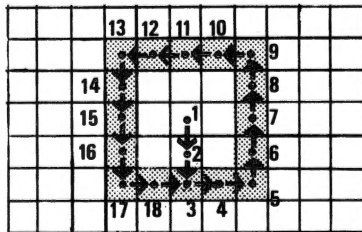
BASIC has five commands that allow the manipulation of shapes. These commands are:

- SCALE
- ROT
- DRAW
- XDRAW
- SHLOAD.

Before these commands may be used, a shape table must be defined. The first step in the definition of the table is to draw the shape on paper. Graph paper works best.



Suppose a square is to be defined. Draw the square on graph paper. Now, starting at the center of the figure, connect all points in the shape with a continuous line. Use only 90° angles on the turns. Next, add arrows on the lines to indicate the direction that was used to connect the points. Numbering is a good practice here.



The arrows are called plotting vectors. These vectors must be translated into their binary codes. The following table supplies these binary codes.

Draw vectors	100	↑	up
	101	→	right
	110	↓	down
	111	←	left
Move vectors	000	↑	up
	001	→	right
	010	↓	down
	011	←	left

An example of a move vector could be vector #2 from the square. A move vector plots nothing onto the screen. It merely moves the starting position of the next vector. An example of a draw vector could be vector #11. A draw vector draws a line.

The easiest way to accumulate the necessary binary information is by using a table as shown on page 237.

**Table 6-3. Vectors**

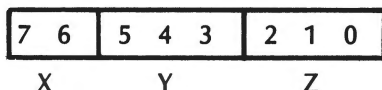
vector #	direction	binary code
1	↓ move	010
2	↓ move	010
3	→ plot	101
4	→ plot	101
5	↑ plot	100
6	↑ plot	100
7	↑ plot	100
8	↑ plot	100
9	← plot	111
10	← plot	111
11	← plot	111
12	← plot	111
13	↓ plot	110
14	↓ plot	110
15	↓ plot	110
16	↓ plot	110
17	→ plot	101
18	→ plot	101

The Apple IIe stores information in words of eight bit binary numbers. Because the binary direction codes are only three bits long, it would be wasteful to put only one plotting vector in a memory location. Usually two and sometimes three plotting vectors can be stored in a specific memory address.

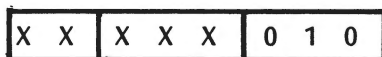
A few rules must be followed while packing the binary codes into memory:

- 1— all bytes are read from right to left.
- 2— if all remaining sections of a byte contain all zeros the rest of the byte will be ignored.
- 3— only a move instruction may be placed in section X of a byte.

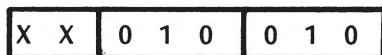
A memory byte may be divided as follows.



To pack the binary codes in Table 6-3 into memory locations, perform the following steps. Place the binary code for vector #1 into section Z of a byte.



Place the binary code for vector #2 into position Y.

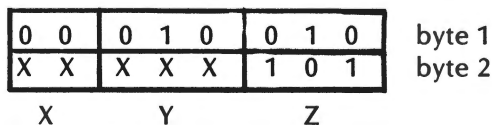


If the next vector is a move vector, place it in section X. In our example, the square, it is not a move vector (vector #3 = plot  $\rightarrow$ ). If the next vector is not a move vector, place two zeros in section X.



Notice that a "move  $\uparrow$ " vector may not be placed in section X. In fact, in any byte, a "move  $\uparrow$ " vector must always have another command to the left of it. This is due to rule 2, as stated previously.

After a byte has been filled, start filling the next byte in a similar manner (vector #3 into section Z of byte 2).



This process should be continued until the list of vectors has been exhausted.

0 0	0 1 0	0 1 0	byte 1
0 0	1 0 1	1 0 1	2
0 0	1 0 0	1 0 0	3
0 0	1 0 0	1 0 0	4
0 0	1 1 1	1 1 1	5
0 0	1 1 1	1 1 1	6
0 0	1 1 0	1 1 0	7
0 0	1 1 0	1 1 0	8
0 0	1 0 1	1 0 1	9

The last byte should be set to zero.

```

00010010
00101101
00100100
00100100
00111111
00111111
00110110
00110110
00101101
00000000 = 0

```

These byte values should now be converted to hexadecimal. This is done by repartitioning the bytes into sections of 4 bits each. Use Table 6-4 to convert a four digit binary number into a single hexadecimal digit.

0001	0010	12
0010	1101	2D
0010	0100	24
0010	0100	24
0011	1111	3F
0011	1111	3F
0011	0110	36
0011	0110	36
0010	1101	2D
0000	0000	00



**Table 6-4. Binary-Hex Conversion**

0 — 0000	8 — 1000
1 — 0001	9 — 1001
2 — 0010	A — 1010
3 — 0011	B — 1011
4 — 0100	C — 1100
5 — 0101	D — 1101
6 — 0110	E — 1110
7 — 0111	F — 1111

The series of bytes previously calculated is called a shape definition.

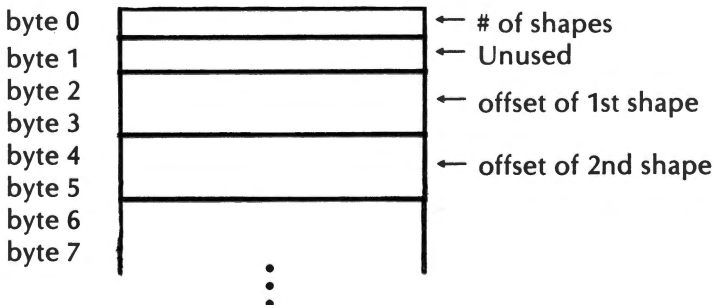
12 2D 24 24 3F 3F 36 36 2D 00

A little more information will be needed to complete the shape table.

### Shape Table Directory

The shape table directory contains the number of shape definitions in the table. It also points to the starting location of each shape definition. A maximum of 255 shapes may be defined in a shape table.

The first byte of the shape table contains the number of shape definitions in the table. The second byte is not used. Starting with the third byte, a table of indices is stored, which references the starting addresses of the individual shape definitions.



The offset is the value that must be added to the starting address of the table to obtain the starting address of a specific shape definition.

An example will be developed using the previously calculated shape definition of a square. The shape table will be stored at memory locations 1F00—1F0D. Therefore, memory location 1F00 must contain the number of shape definitions in the table. In our case, this is one.

1F00 01

The value stored in the next location is insignificant.

1F01 00

The memory location 1F02 and 1F03 must contain the value of the offset of shape definition number one. The starting address of the shape definition has been chosen at 1F04. The offset from 1F00 is, therefore, 4 bytes.

$$\begin{array}{r} 1F04 \\ -1F00 \\ \hline 0004 \end{array}$$

high byte = 00    low byte = 04

This value must be stored in memory locations 1F02-1F03. Values must be stored low byte first. This means that the 04 is stored in location 1F02.

1F02 04 low byte  
1F03 00 high byte

Our completed shape table appears as follows:

1F00	01	} directory
1F01	00	
1F02	04	
1F03	00	
1F04	12	} shape definition
1F05	2D	
1F06	24	
1F07	24	
1F08	3F	
1F09	3F	
1F0A	36	
1F0B	36	
1F0C	2D	
1F0D	00	

This shape table maybe entered into the computer by using the monitor.

\*1F00: 01 00 04 00 12 2D 24 24 3F 3F 36 36 2D 00

The starting address of the shape table must be stored at memory location E8. The monitor command:

\* E8 : 00 1F

accomplishes this, as would two BASIC POKE commands.

```
] POKE 232,0
] POKE 233,31
```

### Saving a Shape Table

To save a shape table on cassette tape, three data values are required. The starting address and ending address of the shape table must be known. Also the difference between the two must be known. In our example,

STARTING ADDRESS	1F00
ENDING ADDRESS	1F0D
DIFFERENCE	0D

The difference must be stored in the memory location 0, low byte first. From the monitor, enter the following:

```
* 0: 0D 00
```

Now write all necessary information to cassette tape by entering the following:

```
* 0.1W 1F00.1F0DW (no return)
```

Do not press the RETURN key until the RECORD button has been pressed on your recorder.

To use the tape, rewind it to the beginning of the stored shape table. Next, enter the following BASIC command.

```
SHLOAD
```

The console speaker will be activated twice upon completion of the command.

A shape table can also be stored on a floppy disk. The BASIC BSAVE command could be used to store the table.

### **BSAVE TABLE,A\$1F00,L\$0E**

The shape table may be loaded back into the computer by executing the following.

### **BLOAD TABLE**

After Binary LOADING the table into the computer from disk, memory location E8 must be loaded with the starting location of the shape table. This is automatically accomplished when SHLOAD is executed with a cassette tape.

### **Using the Shape Table**

BASIC has four commands which can be used to display and manipulate previously defined shapes.

DRAW  
XDRAW  
ROT  
SCALE

### **SCALE**

The SCALE command gives the size at which a shape is to be displayed. The SCALE command takes the following form,

**SCALE = x**

x is a numeric argument with a range of 0-255. This command should be executed before DRAWing any shape to the screen.

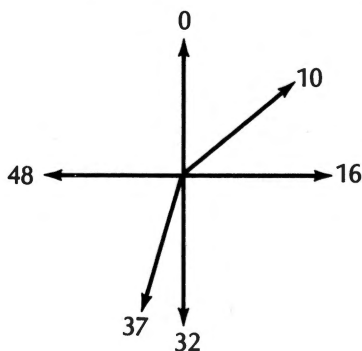
If x = 10 then the Apple IIe will draw each vector in the shape definition to a length of 10 pixels.

### **ROT**

The ROT command rotates the shape around its center. The syntax of this command is as follows:

**ROT = x**

$x$  is a numeric argument with a range of 0-63.



When  $x = 16$ , any shape that is drawn will be displayed with a rotation of  $90^\circ$ . The variable  $x$  may assume a value between 0-255. The number used to calculate the rotation is  $x$  modulo 64.

## DRAW

The syntax of the DRAW command is as follows:

DRAW  $n$  AT  $x,y$

$n$  is the number of the shape to be drawn.  $x,y$  is the coordinate on which the shape is to be centered. DRAW will plot the shape using the previously chosen rotation, scaling, and color (ROT, SCALE, and HCOLOR respectively). If DRAW is used without specifying a coordinate, the shape will be drawn at the last location that was plotted to.

## XDRAW

The XDRAW command allows the erasing of a previously displayed shape. XDRAW will not change any of the background graphics. The syntax of this command is very similar to that of the DRAW command.

XDRAW  $n$  AT  $x,y$

$n$  is the index of the shape in the shape table.  $x,y$  is the position from which to erase the shape.

In order to erase a previously displayed shape, the ROTation, SCALE, and HCOLOR variables must be the same as when the shape was drawn.

## PROGRAMMING WITH SHAPE TABLES

Shape tables can be used in a variety of implementations.

They are especially useful for movement and pattern replication. The following programs serve as examples of the effects of executing the four shape table commands.

### PROG I

```
10 HGR2
20 FOR I = 1 TO 40
30 SCALE = I
40 ROT = I
50 HCOLOR = I/6
60 DRAW 1 AT 140,96
70 NEXT I
80 GOTO 20
```

### PROG II

```
10 HGR2
20 SCALE = 20
30 HCOLOR = 3
40 FOR I = 0 TO 63
50 ROT = I
60 DRAW 1 AT 140,96
70 XDRAW 1 AT 140,96
80 NEXT I
90 GOTO 40
```

**PROG III**

```
10 HGR2
20 ROT = 1
30 FOR J = 0 TO 7
40 HCOLOR = J
50 FOR I = 1 TO 40
60 SCALE = I
70 DRAW 1 AT 140,96
80 NEXT I
90 NEXT J
100 GOTO 30
```





# CHAPTER 7.

## THE SYSTEM MONITOR

---

### Introduction

The **System Monitor** is a set of machine language programs built into the Apple IIe. The System Monitor, or operating system, was written to control the system functions of the Apple. The system functions include:

- monitoring the keyboard and peripherals for inputs
- displaying output on the screen
- saving and retrieving programs from cassette
- controlling the speaker
- allowing the user access to machine language

The operating system's programs are used by high level languages such as Applesoft and Integer BASIC. The Apple Disk Operating System also uses these machine language programs.

Appendix I lists many of the more useful machine code programs as well as their starting locations. These programs have been written as subroutines, so that other programs may access them.

The remainder of this chapter will focus on the subset of programs that allows the user direct access to machine language. The balance of the chapter assumes at least a rudimentary knowledge of 6502 machine language.

### Activating and De-activating the Monitor

When the computer is powered-up, a disk boot is attempted, after which control is given to Applesoft BASIC. In order to activate the Monitor, type:

## CALL -151

This immediate mode statement tells the Apple to jump to memory location FF69<sub>16</sub> which is the starting address of the monitor. The monitor should now have displayed its asterisk prompt.

In a disk based system, there are two versions of the monitor. The newer version is stored in ROM. During the power-up sequence, when the computer boots DOS, the old version is loaded into RAM.

The newer version has the same standard input and output subroutines, but a few features are different. For example, because the older version was written for the Apple II, it doesn't support the arrow keys for cursor movement. When INT is typed in Applesoft to activate Integer BASIC, the old monitor is also activated. It remains active until either FP is typed to return to Applesoft or PR#3 is typed to activate the 80-column firmware. *Part of the 80-column initialization sequence loads the newer monitor from ROM to RAM.* After this has occurred, the new monitor will remain active until the computer has been shut off.

The old monitor has one advantage over the newer version. The old monitor has a mini-assembler built in. The new version does not. This means that Integer BASIC must have control, when the monitor is activated to use the mini-assembler.

To return to BASIC, type CTRL-B, the return-to-BASIC command. Control is given to whichever BASIC had control when you activated the monitor. The CTRL-B command erases all variables and any BASIC program stored in RAM. It is analogous to typing NEW in BASIC.

Since erasing all variables and BASIC program storage could prove quite unproductive, the monitor offers the option of returning without erasing these. The CTRL-C, command accomplishes the return to BASIC without destroying the program or variables. The user can generate the same effect as CTRL-C by pressing CTRL-RESET or by entering the command line 3D0G.

## Commanding the Monitor

The important thing to remember when using the monitor is that it is very picky. In other words, monitor commands must be entered with the correct syntax. In this section, the monitor commands will first be presented in detail. There will also be a short lookup section at the end of the chapter.

The monitor accepts command lines up to 255 characters in length. All command lines must end with a [return]. Commands contain three types of information: **command characters**, **address values** and **data values**. Command characters are single alphabetic characters, control characters and punctuation marks, usually preceded by an address value.

Addresses and data values are expressed in hexadecimal notation. Addresses can consist of as many as four digits; data can consist of up to two. If an address or data value is entered with fewer than the maximum number of digits respectively, the monitor automatically adds leading zeros. If a value greater than the maximum is typed in, the monitor only recognizes the rightmost field of digits. For example, if 78FE4 is entered as an address only 8FE4 will be recognized. Likewise if 788FE is entered as a data value, then only FE will be recognized.

Quite a few examples are contained in this chapter. To make for easier reading, the computer's responses will be displayed in bold. Because of the random nature of the computer's memory after power-up, some of the data values which the computer displays may differ from these values in our examples.

### Memory Examine

To examine the contents of any memory location, just type the hexadecimal address of the location, followed by return.

```
*E300  
E300-88  
  
*FF  
00FF-ED
```

In the second example, the monitor supplied the leading zeros.

After the monitor received the address, it printed that address followed by its contents.

It also did one more thing which was not quite so apparent. It stored the memory location that was examined as the last opened address. Although this seems rather unnecessary now, this extra step will prove very convenient when other monitor commands are used.

### Memory Dump

Examining one location at a time is fine, but this could be quite tedious should, say, 100 consecutive locations need to be displayed. The monitor's answer to this dilemma is referred to as a **memory dump**.

A memory dump is accomplished by typing a period (.), followed by an address. The monitor will display the contents of each of the bytes which follow the opened address up to the final address, as specified in the command line. The amount of data displayed depends upon the difference between these two addresses.

```
*E000
E000-20
```

Hex E000 is now the last opened address.

```
*.E00F
E001 -00      F0      4C      B3      E2      85      33
E008 -4C      ED      FD      60      8A      29      20      F0
```

Data is always displayed in groups of eight or less. This simplifies the reading of the data. For example, the contents of memory location E009 is easily read as ED. The monitor again stores the location of the last displayed location, E00F.

Sometimes it is difficult to know how many locations one wishes

to display. The user can quickly step through memory eight bytes at a time by merely pressing the [return] key. The monitor will display eight bytes in a row, but it is particular which eight it displays. The monitor will display an address ending with a zero or an eight followed by the contents of that address as well as the data of the next eight bytes. Therefore, the first time [return] is pressed, the balance of whichever line the monitor would have displayed will be printed. Each successive [return] produces the next display line.

Monitor commands can be combined into one command line. For example, to display the contents of locations D300 to D31C Hex, type:

\*D300.D31C

<b>D300-AF</b>	<b>D3</b>	<b>48</b>	<b>20</b>	<b>9A</b>	<b>D3</b>	<b>68</b>	<b>20</b>
<b>D308-2E</b>	<b>D0</b>	<b>AE</b>	<b>23</b>	<b>03</b>	<b>60</b>	<b>20</b>	<b>F9</b>
<b>D310-D2</b>	<b>4C</b>	<b>7D</b>	<b>D0</b>	<b>AD</b>	<b>25</b>	<b>03</b>	<b>4A</b>
<b>D318-20</b>	<b>90</b>	<b>D3</b>	<b>20</b>	<b>75</b>			

### Register Examine

When the monitor is activated, the contents of the registers in the 6502 are saved in page-zero. The only register not saved is the program counter. The accumulator, X-register, Y-register, process status register and stack pointer are saved in memory locations 45 through 49, respectively. Instead of typing 45.49 to display the contents of these registers, the monitor provides the register examine, CTRL-E, command. After displaying the contents of these registers, the monitor stores 45, the address of the accumulator, as the opened address.

\*CTRL-E

**A=18 X=FE Y=FF P=B0 S=F8**

### Changing Memory

We have now seen how any location in memory can be displayed. If Picasso could only have looked at his canvas without changing it, how much of an artist would he have been? The same concept applies to the programmer and his canvas, the computer.

The computer cannot be made to do useful work unless it can be given instructions. This is done by changing the contents of memory.

To extend the metaphor a bit further, (since) the Apple IIe is a half-finished canvas, it can understand BASIC commands, store copies of programs on disk and perform numerous other input and output functions. However, if one just randomly dabbed his paint brush around the computer's memory, a nicely started painting could be ruined.

There are a few places not to play around. The most important of these is page-zero. Page-zero is where BASIC and DOS store important variables about themselves. For example, the location of the BASIC program is stored in page-zero. If page-zero must be accessed (usually because of the 6502's indirect addressing techniques), locations F9 to FD can always be used safely.

Keeping this cautionary note in mind, here is how to change the contents of memory. First, open the location to be changed.

```
*357
0357-FF
```

Now, type a colon followed by the new value you wish to store.

```
*:47
*357
0357-47 ← new value stored
```

These commands can be combined.

```
*357:86
*357
0357-86 ← just to check
```

Just as there was a shortcut for examining a large block of memory, there is also a shortcut for changing large portions of memory. To place the values 04 53 F8 8B FF in successive locations starting at 3703, enter the following:

```
*3703: 4 53 F8 8B FF
*3703
3703-04 ← to check
*(ret)
53 F8 8B FF
```

To store data in consecutive memory locations, type the address followed by a colon and the data. Each data item should be separated with a blank space. The monitor can accept as many data values as it can fit on a command line (255 characters).

The monitor stores the address after the last changed address as the opened address. Therefore, to continue entering consecutive data values, just type a colon followed by the remainder of the data.

```
*300:1 2 3 4 5
```

```
*:6 7 8 9 A B C D E F
```

```
*300.30E
```

```
0300-01      02      03      04      05      06      07      08
0308-09      0A      0B      0C      0D      0E      0F
```

### Changing Registers

When the monitor executes any program using the GO command (described later), it first loads the 6502's registers with the values in locations 45 through 49. To alter these values, first execute a register examine.

```
*CTRL-E
```

```
A=0F X=FC Y=07 P=B0 S=11
```

Then, type a colon followed by the new values.

```
*: 1 2 3 4 5
```

```
*CTRL-E
```

```
A=01 X=02 Y=03 P=04 S=05
```

This command sequence works because the examine register command stored 45 as the opened address.

### Move Data

Suppose that the data which is to be stored in one memory block already exists in another memory block. The act of reading that data followed by re-entering it would involve wasted effort. The



monitor supplies the move, M, command to accomplish this task more easily.

To move a block of data, the monitor must know the data's source as well as its destination. It seems that this would require four addresses: the start and end of both the source and destination. One of these addresses supplies redundant information, the ending address of the destination. The format of the instruction, therefore, is as follows:

XXXX<YYYY.ZZZZ M

XXXX stands for the Hex destination starting address. YYYY refers to the source starting address. ZZZZ refers to the source ending address. The less than symbol can be thought of as a funnel which channels data from the source to the destination.

```

*300: 0 0 0 0 0 0
*: 0 0 0 0 0 0
*: 0 0 0 0
*300.30F
0300- 00 00 00 00 00 00 00 00
0308- 00 00 00 00 00 00 00 00

380: 10 20 30 40 50 60 70 80
*:90 A0 B0 C0 D0 E0 F0 FF
*380.38F
0380- 10 20 30 40 50 60 70 80
0388- 90 A0 B0 C0 D0 E0 F0 FF

*300 < 380.38FM
*300.30F
0300- 10 20 30 40 50 60 70 80
0308- 90 A0 B0 C0 D0 E0 F0 FF

```

### Comparing Blocks of Memory

The monitor's verify, V, command compares two blocks of memory. If these two blocks are identical, the monitor will return the asterick prompt. If these blocks are not identical, the

monitor will display the source address of the discrepancy, its contents, and what data should have been there.

The format of the verify command is identical to that of the move command, except for the trailing command letter.

XXXX<YYYY.ZZZZ V

XXXX is the starting address of the destination (what the source is compared to). YYYY is the starting address of the source, and ZZZZ is the ending address of the source.

\*300<E000.E00FM

\*300<E000.E00FV

\*300.30F

<b>0300-</b>	<b>20</b>	<b>00</b>	<b>F0</b>	<b>4C</b>	<b>B3</b>	<b>E2</b>	<b>85</b>	<b>33</b>
<b>0308-</b>	<b>4C</b>	<b>E0</b>	<b>FD</b>	<b>60</b>	<b>8A</b>	<b>29</b>	<b>20</b>	<b>F0</b>

E000-E00F contains the same data as 300-30F.

\*308:AA BB

\*300<E000.E00FV

**E008- 4C (AA)**

**E009- ED (BB)**

Because of the manner of which the move and verify commands are executed, certain programming tricks are possible. To store a pattern of data in memory procede as follows:

Store the pattern to be replicated in the first position of the range.

\*1000: AA BB CC DD

If N is the number of data values in the pattern, and if SSSS is the start address and EEEE is the ending address; enter the following:

SSSS+N<SSSS.EEEE-N M

To replicate AA BB CC DD from 1000 to 103F, enter the following:

```
*1004<1000.103BM
*1000.103F
```

<b>1000- AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>	<b>AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>
<b>1008- AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>	<b>AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>
<b>1010- AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>	<b>AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>
<b>1018- AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>	<b>AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>
<b>1020- AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>	<b>AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>
<b>1028- AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>	<b>AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>
<b>1030- AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>	<b>AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>
<b>1038- AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>	<b>AA</b>	<b>BB</b>	<b>CC</b>	<b>DD</b>

Verify can be used in a similar manner to determine whether a pattern repeats itself in memory. For example, to determine if the pattern in the preceding example does indeed exist in memory, type the following.

```
*1004<1000.103BV
```

This command is useful in evaluating whether an area of memory has been set to a certain value. First clear page-20.

```
*2000:0
*2001<2000.20FEM
```

Next, change the following two locations.

```
*2040: 47
*2080: FF
```

Finally, check page-20.

```
*2001 < 2000.20FEV

203F- 00 (47)
2040- 47 (00)
207F- 00 (FF)
2080- FF (00)
```

The monitor indicates the discrepancy at location 2040 with the first two lines of output. It treats the error at 2080 in the same manner.

### **Saving and Retrieving Data with the Cassette**

The monitor's write, W, command writes the contents of a range of memory locations onto cassette tape. The command syntax is straightforward. The command is entered by typing the starting address; a period; the ending address; and a W. The following example will store the data in locations 1000 to 1040 on cassette tape.

```
*1000.1040W ← Don't press the return here
```

Note that we did not immediately end the command by pressing return. The reason for this was to first allow for the recorder's play/record button to be pressed. Press record, followed by return. After return has been pressed, the monitor will write a 10 second steady tone leader, and will then record the data and the checksum. When the recording process has been completed, the monitor will cause the console speakers to beep.

The monitor's read, R, command is similar in operation to the write command. Enter the address of the range where the data is to be stored, then type R (again, no return).

```
*2000.2040 R
```

To ready the cassette for input to the computer, rewind the tape to the beginning of the leader tone. A steady tone will be emitted. Press the recorder's play key, wait a few seconds, and then press return.

The monitor will read the data and the checksum, and will then verify that the data is correct. If no errors are discovered the monitor will respond with a beep. However, if the checksum does not match the stored data, the message ERR will be displayed.

The checksum is a number which the Apple calculates from the data when it is saved. This number is unique for any specific data. Therefore, if the checksum does not verify when it is read, an error most likely occurred when the data was written to the cassette.

### Saving and Retrieving Data from Disk

Unfortunately, the cassette is slow and very prone to errors. Therefore most users prefer a disk drive for data storage. However, the monitor does not include commands for reading or writing to diskette. In order to save assembly code on disk, the user must first activate BASIC. Then, the binary save (BSAVE) or binary load (BLOAD) commands must be executed in order to save or read the data.

To save the same memory addresses as were saved in the cassette example (1000-1040), the following BASIC command would be required:

```
*CTRL-C ← into BASIC
] or > BSAVE TXTA,A$1000,L$41
```

TXTA is the name of the disk file. A stands for **at** memory location. L stands for **length** of memory block. In the preceding example, the block of data beginning at address \$1000 and extending to \$1040 would be saved on disk with the filename, TXTA.

To recall this block of data from disk, type:

```
] or > BLOAD TXTA,A$2000
```

This command will load the file at locations 2000 to 2040. If "A\$2000" had not been entered on this line, the binary file would have been loaded at the same locations from which it had been saved (1000-1040).

It is good practice to verify that data written to the cassette unit has been saved accurately. Although errors are encountered

with much less frequency on disk writes, it is also good practice to verify those as well.

To verify that the machine code has been faithfully reproduced, first save the code, using the procedures outlined previously. Then, load the information back into the computer, but load it at a different address. The monitor's verify command can now be used to compare the two sections of code. If the data had been saved correctly, the two sections will be identical.

```
*1000.10FF W
rewind tape
*2000.20FF R
*1000<2000.20FF V
*
```

If the verify command causes any addresses and data values to be displayed, the save was not successful. If only the asterisk prompt was displayed, the save worked perfectly.

### Other Input/Output Commands

The format in which information is displayed on the screen is controlled by the monitor's Inverse and Normal commands. If "I" is included in a command line, all monitor inputs and outputs displayed thereafter will be displayed in inverse video. This process can be reversed by including "N" as part of a subsequent command line.

```
*E000
E000- 4C
*I E000 N E000
E000- 4C ← inverse video
E000- 4C ← normal
*
```

The monitor's input and output can also be redirected to peripheral devices. The printer command directs the monitor's output. The keyboard command determines the device from which the monitor will accept input. The printer and keyboard com-

mands perform the same function as the BASIC PR# and IN# commands, respectively.

The syntax for the printer command is as follows:

**\*# CTRL-P**

# represents the slot number of the peripheral device to which data is to be sent.

For example, the following commands would dump the contents of addresses 300 to 3FF to the peripheral device with its card in slot #1.

**\*1 Ctrl-P  
\*300.3FF**

The preceding command causes subsequent monitor output to be directed to the device whose card is in slot #1. If you wish to redirect output to the screen, you can, do so by executing the following command:

**\*0 Ctrl-P**

The keyboard command functions in a manner quite similar to the printer command, except that data will be accepted from the specified device rather than output to it. The following command causes subsequent data to be accepted from the device whose controller card is in slot #1.

**\*1 Ctrl-K**

Data can again be accepted from the keyboard by executing the following command.

**\*0 Ctrl-K**

The monitor can input and output through a single peripheral. For example, suppose a modem is connected to a serial port with

its card in peripheral slot #5. The computer could accept information from the modem as well as information to it. By entering the following commands, the computer would be instructed to use the modem.

\*5 CNTL-P

\*5 CNTL-K

## **MACHINE LANGUAGE PROGRAMMING**

### **Introduction**

The process of writing a machine language program is a long and tedious one compared to the process of writing a program in BASIC. Why then would a programmer wish to write a program in machine language? The answer is speed. Machine language programs execute at a rate anywhere from 10 to 1000 times faster than a BASIC program.

### **Mini-Assembler**

An assembler greatly facilitates the process of writing a machine language program. An assembler translates assembly language mnemonics to operation codes which can be executed by the microprocessor. Without an assembler, the programmer would have to look up each operation code, one-by-one.

A mnemonic is a short abbreviation for an assembly language instructions. For example, JSR is the mnemonic which stands for the jump to subroutine operation. The microprocessor does not understand mnemonics. It does however understand hexadecimal codes. The purpose of the assembler is to translate a mnemonic (ex. JSR) to its equivalent hex value (#20).

The mini-assembler, built into Integer BASIC, is not full-fledged assembler. The mini-assembler does not support labels, nor does it remember the source once return has been pressed. The mini-assembler merely converts the mnemonic entered in the command line into its equivalent hex value(s).



The assembler then stores these hex values in memory. The original mnemonic entry will be lost. Since the mnemonic is not saved in a source file, modification of the machine language program would be difficult.

### Activating the Mini-Assembler

Integer BASIC's assembler, can be executed by entering the following command:

```
>CALL -2458
```

The mini-assembler could also be called directly from the monitor as long as Integer BASIC had previously been active. The mini-assembler can be loaded from the monitor by executing the following command:

```
*F666G
```

The G denotes the monitor's G command. This command will be explained in detail later in this chapter.

These two commands are known as the assembler enter commands. These commands cause a JMP (Jump) to be executed to the starting address of the mini-assembler. Once this instruction has been executed, the mini-assembler will function and its prompt (!) will be displayed.

### Entering the First Program Line

Once the mini-assembler's prompt (!) has been displayed, the programmer must input a line consisting of the assembly language program's starting address followed by the first mnemonic instruction. These must be separated by a colon. This is shown in the following example.

```
!300:JSR FBDD
0300- 20 DD FB JSR $FBDD
```

### Entering Subsequent Program Lines

The assembler is now ready to accept another program line. To enter the next instruction in the next consecutive memory location, enter a blank space followed by the next instruction. Be sure to include the space as it is important. The following is an example of the correct syntax for the entry of a subsequent program lines.

```
! JMP 300
0303- 4C 00 03   JMP   $0300
```

If any errors are encountered in an input line, the assembler will cause the speaker to beep and will display an arrow ( ^ ) beneath the offending character.

A second error can occur if a branch instruction is attempted in a program of more than  $FF_{16}$  bytes.

### Returning to the Monitor

Suppose the assembler was in use and the programmer wished to return to the monitor. This can be accomplished by prefixing the monitor command (FF69G) with the \$ character as shown in the following example.

```
!$FF69G
```

When \$ prefixes a monitor command while the assembler is active, this monitor, is executed. The FF69G monitor command executes a jump to the monitor's starting location. The \$ is not limited to usage with only FF69G. It can be used with any monitor command.

### Converting Assembly Language Hex Codes Back into Mneumonics

The monitor includes a list command which allows the programmer to convert hex data in memory back into mneumonics. When the following command is entered:

\*300L

The following data will be displayed:

0300- 20	DD	FB	JSR	\$FBDD
0303- 4C	00	03	JMP	\$0300
0306- 00			BRK	
0307- 00			BRK	
0308- 00			BRK	
0309- 00			BRK	
030A-00			BRK	
030B- 00			BRK	
030C-00			BRK	
030D-00			BRK	
030E- 00			BRK	
030F- 00			BRK	
0310- 00			BRK	
0311- 00			BRK	
0312- 00			BRK	
0313- 00			BRK	
0314- 00			BRK	
0315- 00			BRK	
0316- 00			BRK	

The 300 indicates the starting address of the first assembly language instruction (in hex) which is to be converted back into a mnemonic. The L indicates the list command.

In our example, all data displayed after 306 will be random as assembly language instructions were only entered at addresses 300-305 inclusive.

The L command automatically fills the screen (20 lines) with the contents of memory beginning with the address specified as its argument.

When a listing is complete, the monitor stores the last line that was listed as the program counter. If the L command is used without an argument, the monitor uses the program counter as the first line to be listed. Using a series of single L's, the pro-

grammer can conveniently list a program which would fill more than one screen.

### Executing a Machine Language Program

The monitor's GO command is used to execute a machine language program.

When GO is executed, the microprocessor's registers will be loaded with the values stored in addresses \$45 to \$49 as follows:

- The microprocessor's accumulator will be loaded with the value stored in address \$45.
- The X register will be loaded with the value stored in \$46.
- The Y register will be loaded with the value stored in \$47.
- The processor status register will be loaded with the value stored in \$48.
- The stack pointer will be loaded with the value stored in \$49.

Once the registers have been loaded, a JSR (Jump to Subroutine) command will be executed to the last opened memory location.

Since the monitor treats all programs as subroutines, a RTS (Return from Subroutine) instruction should be the last instruction in the program.

We will illustrate the usage of GO using the following example (which you probably have already entered).

```
!300:JSR FBDD  
! JMP 300
```

This program can be executed by entering the following:

```
*300 G
```

Once you have started the program, you can stop it by pressing Ctrl-Reset.

This program causes the speaker to buzz. This is accomplished using the monitor bell subroutine. This is located at memory address FBDD. The second instruction (JMP 300) causes the program to repeatedly jump to the bell subroutine. This results in a continuous buzzing being emitted from the speaker.

The various I/O subroutines are listed in Appendix I. These can be executed via a machine language program.

### Creating a Custom Monitor Command

The monitor's CTRL-Y command causes program control to be transferred to the instruction at memory location 3F8. Addresses 3F8-3FF are available for usage by the programmer. Generally, because only 8 bytes are available, a JMP instruction will be stored at these addresses. However, a machine language program could also be stored there.

One useful application of the CTRL-Y command is automatic execution of the mini-assembler. By storing the following data values at locations 3F8-3FA:



CTRL-Y will result in automatic execution of the mini-assembler.

The programmer could avoid having to enter the preceding command by adding the following program line to the DOS "Hello" file. This can be accomplished by entering the following:

```

]UNLOCK HELLO
]LOAD HELLO
]5 POKE 1016, 76: POKE 1017,102:POKE 1018,246
]SAVE HELLO
]LOCK HELLO
  
```

After this entry has been made, upon power-up, the command to start the assembler will be resident in location 3F8-3FA. Ctrl-Y could then be used to automatically activate the monitor.

## LOOK UP SECTION

### Into/Out of Monitor

CALL-151	Activates Monitor.
CNTL-RESET	Returns to BASIC.
CNTL-C	— program and
3D0G	variables intact
CNTL-B	Returns to BASIC.
	— program and
	variables erased

### Memory Examine

nnnn	Displays contents of address nnnn
mmmm.nnnn	Displays contents of addresses mmmm through nnnn
[return]	Displays contents of as many as eight addresses following the last opened address

### Memory Change

nnnn: dd <sub>1</sub> dd <sub>2</sub> ...	Store data values in consecu- tive locations starting with nnnn
---	---

### Register Examine

CTRL-E	Display contents of 6502's reg- isters which are loaded prior to program execution
--------	--

CTRL-E: dd<sub>A</sub> dd<sub>X</sub> dd<sub>Y</sub> dd<sub>P</sub> dd<sub>S</sub> Stores values for registers

### Move and Verify

xxxx<yyyy.zzzz M Copy contents of range of addresses yyyy through zzzz to range starting at xxxx

xxxx<yyyy.zzzz V Compares the contents of addresses yyyy through zzzz with the range beginning with xxxx.

### Saving and Loading Program

ssss.eeee W Saves contents of addresses ssss to eeee onto tape.

ssss.eeee R Loads from tape to addresses ssss to eeee.

BSAVE NAME, A\$xxxx,L\$yyyy Saves in disk file "NAME" contents of the range of addresses, starting at xxxx with length yyyy.

BLOAD NAME, A\$xxxx Loads memory addresses starting at xxxx with disk file "NAME". If A\$xxxx is not specified, loads at address at which data was saved.

### Input/Output

N Sets normal display mode.

I Sets inverse display mode.

# CNTL-P Directs output to peripheral with card in slot #.

# CNTL-K Accepts input from peripheral with card in slot #.

**Mini-assembler**

CALL-2458 Activates assembler  
F666G

\$FF69G Activates monitor.

CNTL-RESET Activates INT BASIC

\$ Executes monitor command while using assembler.

**Running/Listing**

nnnn G Executes machine language program at location nnnn

nnnn L Disassembles 20 machine language instructions starting at location nnnn

**Your Command**

CNTL-Y Jumps to machine language subroutine at location 3F8.





## CHAPTER 8. THE 80-COLUMN BOARD

---

Unlike its predecessors, the Apple IIe has an additional slot built into its motherboard. The slot is called the **auxiliary expansion slot**. This slot accepts one of two peripheral expansion cards. One of the cards provides the Apple IIe with extended display capability as well as an additional 64K of memory. The other card provides only for the extended display. Since the extended display capability of both cards is identical, all references to the cards in this chapter shall be "The Apple 80-column board".

The Apple 80-column board allows the computer to display a full 80 columns on the screen. It also allows for additional editing features, including screen editing.

The 80-column board can be used with Pascal, CP/M\*, or BASIC. The 80-column board is automatically switched on whenever Pascal or CP/M is used. However, this is not the case with BASIC.

When using CP/M, all features of the 80-column board are automatically activated. When using Pascal, all features except the cursor up and cursor down keys are functional. This can be remedied through execution of the SET UP program provided on the Pascal Disk, Apple 3.

As mentioned earlier, set up with BASIC is more complex. This is due to the fact that the Apple IIe was designed to be software compatible with the Apple II and Apple II+ computers.

---

\* CP/M is a trade mark of Digital Research.

### **Activating the 80-Column Board in BASIC**

To enable the 80-column board, first power-up the computer as usual. Then, after Applesoft has control of the computer, enter the following,

PR#3

to activate the board. It is usually convenient to depress the CAPS LOCK key since Applesoft does not understand lowercase letters.

The statement PR#3 may seem a bit obscure. The PR#3 command is executed because the auxiliary expansion slot is hard-wired to peripheral slot number three. Recall that PR# is the BASIC statement used to activate a peripheral for output.

Since non-Apple manufactured 80-column boards are traditionally used in peripheral slot #3, the auxiliary expansion slot has been hard-wired to this slot.

This wiring arrangement has one drawback. If any peripheral card is inserted in slot #3, it will be rendered inactive by the 80-column board.

The effect of activating the 80-column board is that the screen is cleared. Next, the 80-column cursor will be displayed. The 80-column cursor is only half as wide as the standard BASIC cursor. It is solid instead of checkered, and does not blink.

Everytime the computer is powered-up, the 80-column card must be activated before it is used. This may prove to be quite tedious. This tedium is inevitable in a cassette based system. However, this situation can be remedied in a disk based system.

The 80-column board can be activated each time a specific disk is used through modification of the HELLO program on that disk. The DOS HELLO program is a BASIC program that is executed during the power-up procedure. The computer runs this program automatically. The HELLO program contains information

used during the power-up procedure.

Certain parts of this program should not be changed. These parts begin with line number 10. This allows only line numbers 0 through 9 to be used safely for the user's modification of the program.

The BASIC statement,

```
1 PRINT CHR$(4);"PR#3"
```

when added to the HELLO program, will cause the 80-column board to be activated at power-up. This statement accomplishes its task by printing the DOS command PR#3.

In order to change the HELLO program, first unlock it, then load it into the computer.

```
] UNLOCK HELLO
] LOAD HELLO
```

Add the following BASIC statement into the program by entering it in response to the Applesoft prompt (]).

```
] 1 PRINT CHR$(4);"PR#3"
```

List the program to verify that line 1 has indeed been entered correctly. Finally, save and lock the new HELLO program.

```
] SAVE HELLO
] LOCK HELLO
```

### **Deactivating the 80-Column Board**

When the 80-column board is activated, output to another device is difficult. The disk drive presents no problems, but attempting printer output may adversely effect the display.

In order to output to a printer, the 80-column board must be disabled. This is the case even if the 80-column board is in its

40-column mode (described later in this chapter). Remember, if the cursor is solid and does not blink, then the 80-column board is active.

To deactivate the board, type the [escape] key, followed by CONTROL-Q.



The screen will revert to a 40 column display, and the blinking checkerboard cursor will return at the bottom of the screen.

CONTROL-RESET will also deactivate the 80-column board. However, the user is advised against this technique. The use of CONTROL-RESET while the 80-column board is active will disrupt display, and may cause the erasure of any RAM-resident programs.

### **Selecting 40 or 80 Columns While Board is Active**

Occasionally, a 40-column display may be desirable. However, it may not be desirable to forfeit the extra editing capability of the Apple 80-column board. This dilemma may be resolved via the use of the 40-column mode of the Apple 80-column text board. In the 40-column mode, the text board is active, although only 40 columns are displayed.

A switch to the 40-column display can be accomplished by either of two commands. Typing CONTROL-Q, while the board is active will enable the 40-column mode. Notice that an escape does not precede the CONTROL-Q. Upon receipt of this command, the computer will display any text which had previously been displayed in the leftmost 40-columns of the 80-column display. The width of each character will be doubled. Pressing the escape key, followed by pressing the 4 key, causes identical results.

Either CONTROL-R or escape,8 will return the display to the 80-column mode. Any text that had been displayed in the 40-

column mode will be redisplayed in the leftmost 40-columns of the 80-column display. The right hand side will be cleared.

The following four commands may also be used in programs to change the display mode during program execution.

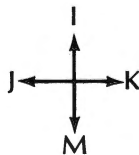
or	ESC 4	Select
	CONTROL-Q	40-column mode
or	ESC 8	Select
	CONTROL-R	80-column mode

### Moving the Cursor

The 80-column board comes equipped with numerous editing features. These features can be implemented through the use of the ESC and CONTROL keys. When using a control command, the CONTROL key must be held down before another key is pressed. When using an escape command, the ESC key must be pressed and released before another key can be pressed.

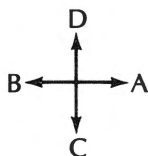
The easiest editing keys to understand are undoubtedly the arrow keys. When in the escape mode, these keys allow the cursor to be moved across the display field, without destroying any of the text already displayed. This allows the user to edit text anywhere on the screen. The escape mode is entered by pressing the ESC key. It is exited by pressing the space bar. While in the escape mode, the cursor becomes an inverse video plus sign.

While in the escape mode, the I, J, K, and M keys perform the same function as the up-arrow, left-arrow, right-arrow, and down-arrow keys, respectively.



The four keys, A, B, C, and D may also be used to accomplish

cursor movement. However, after a single keystroke, the computer automatically exits the escape mode.



Moving the cursor up one row would require either of the following key sequences:

or    ESC D  
      ESC ↑ SPACE

To move the cursor up four rows would require either of the two following key sequences.

ESC D ESC D ESC D ESC D  
ESC ↑ ↑ ↑ ↑ SPACE

Use of ESC,D is cumbersome for cursor moves of more than a few rows. The same idea applies to the use of ESC,A, ESC,B, and ESC,C.

### Editing Functions that Clear Parts of the Display

It is often necessary to clear all or parts of a display, so that new information can be displayed neatly. The simplest of the commands in this category is the Clear Screen command. When a CONTROL-L is issued, either through the keyboard or in a program, the display is cleared and the cursor is positioned in the upper lefthand corner of the screen.

CONTROL-L        clear screen

A similar command is the "clear to end of screen" command. When the CONTROL-K command is used, the screen is cleared from the current cursor position to the end of the screen. The

position of the cursor remains unchanged.

CONTROL-K      clear to end of screen

In order to clear only a specific row, the CONTROL-Z command is used. When this command is issued, the row in which the cursor resides is cleared. The cursor position remains unchanged.

CONTROL-Z      clear line

The CONTROL-] command will clear the line in which the cursor resides from the current cursor position to the end of that line.

CONTROL-]      clear rest of line

### Scrolling the Display

The entire screen display can be scrolled up or down, without moving the cursor position. If any information is scrolled off either end of the screen, that test will be lost.

CONTROL-V      scroll down  
CONTROL-W      scroll up

### Use of CONTROL Codes from BASIC

All of the screen editing features of the 80-column board may be implemented through BASIC. These features are used by printing the escape or control character. Since the characters cannot be entered directly into PRINT statements from the keyboard, the CHR\$ function must be used to generate the control characters.

The ESCAPE codes are generally not used in programs because of their complexity.

The following example uses the scroll up and scroll down commands.



```

10 HOME:HTAB 15
20 PRINT "BOUNCING"
30 FOR I = 1 TO 20
40 PRINT CHR$(22); ← scroll down
50 NEXT I
60 FOR I = 1 TO 20
70 PRINT CHR$(23); ← scroll up
80 NEXT I
90 GOTO 30
    
```

**Table 8-1. CONTROL Codes**

CONTROL Code	Equiv. ESC Code	ASCII	Function
G		7	ring bell
H	B, ←, J	8	backspace
J	C, ↓, M	10	line feed
K	F	11	clear to end of screen
L	@	12	clear screen
M		13	return
N		14	set normal display (only in program)
O		15	set inverse display (only in program)
Q	4	17	set 40-column mode
R	8	18	set 80-column mode
U	CONTROL-Q	21	deactivate 80-column board (in program)
V		22	scroll up
W		23	scroll down
Y		25	home (no clear)
Z		26	clear line
\	A, →, K	28	forward space
]	E	29	clear to end of line

## BASIC Support of the 80-Column Board

The 80-column board does not effect the functioning of the majority of BASIC's majority commands. Only four of the commands function differently when used with the 80-column board. These are as follows:

```
HOME
FLASH
HTAB
INVERSE
```

### Tabbing

Horizontal tabbing, while using the 80-column board, does not operate in exactly the same manner as it does without the board. If the numeric argument of the HTAB function is greater than 40, an automatic wrap-around occurs. If an HTAB is attempted to any of the columns greater than 40, an automatic wrap-around occurs even though the display is capable of 80 columns. Therefore, nothing is displayed in the last 40 columns.

This is remedied by the substitute HTAB command. This command is as follows:

```
POKE 1403,XX
```

XX represents the column number to which the next screen output will be directed. HTAB XX may still be used for the first 40 columns if the programmer so desires. The substitute command will only operate correctly when the 80-column display is being used.

```
100 VTAB 5
110 POKE 1403,65
120 PRINT "OUTPUT"
```

Execution of the previous example will output the text, OUTPUT, at screen location 65,6 (65th column, 5th row).

## Use of INVERSE, FLASH, and HOME

The FLASH command is not supported by the 80-column board. However, FLASH can be used while the 80-column board is inactive. If FLASH is active during the activation of the 80-column board, the screen will turn white, and LIST'ing a program may return an unintelligible result. To recover, enter the NORMAL command.

The 80-column board extends the INVERSE capability of the Apple IIe to include lowercase letters. Without an active 80-column board, only capital letters can be displayed in INVERSE video.

The HOME command works as usual, with one exception. This exception occurs if the computer is in the inverse mode when the HOME command is issued. While the 80-column board is inactive, the HOME command causes the screen to be blackened, but all subsequent output will appear in INVERSE video. However, while the board is active, the HOME command causes the entire screen to be colored white, with all subsequent output in INVERSE video.

## Uppercase-Restrict Mode

A very convenient feature of the 80-column board is the uppercase-restrict mode. Upon entering this mode, all lowercase entries will be interpreted as capital letters, unless they are entered between quotation marks. This feature relieves the programmer of the tedium of repeatedly pressing the CAPS LOCK key.

ESC,R	activates uppercase-restrict
ESC,T	deactivates uppercase-restrict

When programming, the uppercase-restrict mode is useful for the entry of PRINT statements. BASIC requires that all commands be in capital letters. It is often desirable to have a program output in both lowercase and uppercase letters. Without an

uppercase-restrict mode, a program line such as,

```
10 PRINT "Tucson, Arizona"
```

would have been very difficult to enter. The uppercase-restrict mode allows the line to be entered using the SHIFT key only twice.



**APPENDIX A. APPLESOFT BASIC RESERVED WORDS & TOKENS**

ABS	(212)	HTAB	(150)	REM	(178)
AND	(205)	IF	(173)	RESTORE	(174)
ASC	(230)	IN#	(139)	RESUME	(166)
AT	(197)	INPUT	(132)	RETURN	(177)
ATN	(225)	INT	(211)	RIGHT\$	(233)
CALL	(140)	INVERSE	(158)	RND	(219)
CHR\$	(231)	LEFT\$	(232)	ROT=	(152)
CLEAR	(189)	LEN	(227)	RUN	(172)
COLOR=	(160)	LET	(170)	SAVE	(183)
CONT	(187)	LIST	(188)	SCALE=	(153)
COS	(222)	LOAD	(182)	SCRN(	(215)
DATA	(131)	LOG	(220)	SGN	(210)
DEF	(184)	LOMEM:	(164)	SHLOAD	(154)
DEL	(133)	MID\$	(234)	SIN	(223)
DIM	(134)	NEW	(191)	SPC(	(195)
END	(128)	NEXT	(130)	SPEED=	(169)
EXP	(221)	NORMAL	(157)	SQR	(218)
FLASH	(159)	NOT	(198)	STEP	(199)
FN	(194)	NOTRACE	(156)	STOP	(179)
FOR	(129)	ON	(180)	STORE	(168)
FRE	(214)	ONERR	(165)	STR\$	(228)
GET	(190)	OR	(206)	TAB(	(192)
GOSUB	(176)	PDL	(216)	TAN	(224)
GOTO	(171)	PEEK	(226)	TEXT	(137)
GR	(136)	PLOT	(141)	THEN	(196)
HCOLOR=	(146)	POKE	(185)	TO	(193)
HGR	(145)	POP	(161)	TRACE	(155)
HGR2	(144)	POS	(217)	USR	(213)
HIMEM:	(163)	PRINT	(186)	VAL	(229)
HLIN	(142)	PR#	(138)	VLIN	(143)
HOME	(151)	READ	(135)	VTAB	(162)
HPlot	(147)	RECALL	(167)	WAIT	(181)
				XDRAW	(149)

**APPENDIX B. INTEGER BASIC RESERVED WORDS**

ABS	END	LET	PDL	SAVE
AND	FOR	LIST	PEEK	SCRN
ASC	GOSUB	LOAD	PLOT	SGN
AT	GOTO	LOMEM:	POKE	STEP
AUTO	GR	MAN	POP	TAB
CALL	HIMEM:	MOD	PRINT	TEXT
COLOR=	HLIN	NEW	PR#	THEN
CON	IF	NEXT	REM	TO
DEL	IN#	NOT	RETURN	TRACE
DIM	INPUT	NOTRACE	RND	VLIN
DSP	LEN	OR	RUN	VTAB

**APPENDIX C. DOS RESERVED WORDS**

APPEND	CHAIN	INIT	POSITION	SAVE
BLOAD	CLOSE	LOAD	READ	UNLOCK
BRUN	DELETE	LOCK	RENAME	VERIFY
BSAVE	EXEC	OPEN	RUN	WRITE

## Appendix D. APPLESOFT BASIC ERROR MESSAGES

When an error occurs in Applesoft BASIC, an error message will appear, and the program will return to the command level (i.e. the ] prompt will appear). The error message will be displayed using the following configuration:

? name ERROR IN line

where *name* indicates the error name and *line* denotes the line number where the error occurred. If the error occurred in the immediate mode, *line* will be omitted.

The program listing in memory is not affected by an error nor are the stored variable values. However, all GOSUB and FOR loop counters are reset to 0.

When an error does occur, the error's code will be stored in memory address 222. By PEEKing this address, the error code can be determined.

The various Applesoft errors are discussed in the following table.

Error Message	Error Code	Error Description
?BAD SECTOR ERROR	107	The program referenced an array element with a subscript not defined in a DIM statement.
?CAN'T CONTINUE ERROR	None	This error occurs when the program attempted one of the following: <ul style="list-style-type: none"> <li>• Execution of CONT when a program was not present in RAM.</li> <li>• Execution of CONT after the occurrence of a program error.</li> <li>• Execution of CONT after the program had been edited.</li> </ul>
?DIVISION BY ZERO ERROR	133	Division by zero is not allowed.



Error Message	Error Code	Error Description
?FORMULA TOO COMPLEX ERROR	191	This error indicated that a string expression was used that was too complex for Applesoft BASIC to comprehend. The expression should be broken into 2 or more parts.
?ILLEGAL DIRECT ERROR	None	A statement was used in the immediate mode which is only allowed in the program mode (ex. INPUT, GET, DEF FN).
?ILLEGAL QUANTITY ERROR	53	A value was used with a math or string function that was out of range.
?NEXT WITHOUT FOR ERROR	0	This error will be generated when a NEXT statement is encountered without a corresponding FOR statement. This error is often the result of different variables being used with FOR and NEXT.
?OUT OF DATA ERROR	42	A READ statement was executed which contained variables for which DATA statement values were not available.
?OUT OF MEMORY ERROR	77	<p>This error can be the result of any of the following:</p> <ul style="list-style-type: none"> <li>● LOMEM: set too high.</li> <li>● HIMEM: set too low.</li> <li>● Use of a program that was too large for available memory.</li> <li>● Use of excessive nesting of FOR, NEXT loops, GOSUBS, or parentheses.</li> <li>● Use of an overly complicated expression.</li> <li>● Use of too many variables.</li> </ul>

Error Message	Error Code	Error Description
?REDIM'D ARRAY ERROR	120	A second DIM statement for an array was encountered after that array had already been dimensioned. This error usually occurs when an array had been previously dimensioned by default and a DIM statement was subsequently executed for the same array variable.
?RETURN WITHOUT GOSUB ERROR	22	A RETURN statement was executed which did not have a corresponding GOSUB statement.
?STRING TOO LONG ERROR	176	A statement attempted to concatenate two or more strings with a resultant string in excess of 255 characters.
?SYNTAX ERROR	16	The statement used an incorrect spelling, incorrect punctuation, illegal character, etc.
?TYPE MISMATCH ERROR	163	A numeric value was indicated where a string value should have been used or vice versa.
?UNDEF'D FUNCTION ERROR	224	A user-defined function was referenced which has not been defined using DEF FN.
?UNDEF'D STATEMENT ERROR	90	An attempt was made to branch to a line number that did not exist.

## APPENDIX E. INTEGER BASIC ERROR MESSAGES

Error Message	Error Description
***BAD BRANCH ERR	The statement attempted to branch to a line number which does not exist.
***BAD NEXT ERR	A NEXT statement was executed without a corresponding FOR.
***BAD RETURN ERR	A RETURN statement was executed without a corresponding GOSUB statement.
***DIM ERR	An attempt was made to dimension an array that had previously been dimensioned.
***MEM FULL ERR	An insufficient amount of memory is available.
***NO END ERR	An END statement should be included as the last program line in Integer BASIC programs.
***RANGE ERR	This error is generated when an array variable is used with an illegal subscript (i.e. a subscript larger than that for which the array was DIM'ed). This error is also generated when an illegal argument is used in HLIN, VLIN, PLOT, TAB, or VTAB.
RETYPE LINE	This message is displayed when an illegal entry was made in response to an INPUT statement.
***STRING ERR	This message can be generated by almost any illegal string operation.

<b>***STR OVFL ERROR</b>	An attempt was made to assign a string a greater number of characters than it had been dimensioned for.
<b>***SYNTAX ERROR</b>	The statement used an incorrect spelling, incorrect punctuation, illegal character etc.
<b>***TOO LONG ERROR</b>	This error occurs when over 128 characters are included on a single statement line or when over 12 parentheses have been nested.

**APPENDIX F. DOS ERROR MESSAGES**

When a DOS error is encountered in the context of an Applesoft program, the corresponding DOS error code will be placed in memory address 222. The error code can be determined by executing PRINT PEEK(222).

<b>Error Message</b>	<b>Error Code</b>	<b>Error Description</b>
DISK FULL	9	This error occurs when a new file or new data is to be written to a disk on which insufficient space is available to store the new information.
END OF DATA	5	This error is generated when a READ statement tries to read beyond the end of a text file.
FILE LOCKED	10	The program attempted to execute a WRITE, DELETE, RENAME, SAVE, or BSAVE statement with a locked file.
FILE NOT FOUND	6	A file is referenced which does not exist on the diskette. This error condition usually occurs as the result of using an incorrect filename.
FILE TYPE MISMATCH	13	<p>An incorrect file type was used with a DOS command.</p> <ul style="list-style-type: none"> <li>• BLOAD, BSAVE, and BRUN can only be used with binary files.</li> <li>• CHAIN can only be used with Integer BASIC program files.</li> <li>• LOAD, RUN, and SAVE can only be used with program files.</li> <li>• PEN, READ, WRITE, APPEND, EXEC, and POSITION can only be used with text files.</li> </ul>

Error Message	Error Code	Error Description
I/O ERROR	8	<p>A disk access operation was unsuccessful. This error is generally caused by one of the following:</p> <ul style="list-style-type: none"> <li>• Drive door open.</li> <li>• Diskette not inserted in drive.</li> <li>• Diskette not initialized.</li> <li>• Defective diskette.</li> </ul>
LANGUAGE NOT AVAILABLE	1	<p>The operator attempted to access Integer or Applesoft BASIC when that language was not available. This error is also generated by attempting to load or run a program when the program's language was not available.</p>
NO BUFFERS AVAILABLE	12	<p>All available file buffers are in use.</p>
NOT DIRECT COMMAND	15	<p>The following DOS commands cannot be executed in the immediate mode, and can only be executed within a program:</p> <ul style="list-style-type: none"> <li>• APPEND</li> <li>• POSITION</li> <li>• OPEN</li> <li>• READ</li> <li>• WRITE</li> </ul>
PROGRAM TOO LARGE	14	<p>A DOS command tried to load a program file from the diskette into RAM when insufficient memory was available for the program file.</p>
RANGE ERROR	2 or 3	<p>An illegal parameter was specified with a DOS parameter. This error is generally due to an incorrect drive, slot, or volume specification.</p>

<b>Error Message</b>	<b>Error Code</b>	<b>Error Description</b>
SYNTAX ERROR	11	A filename or parameter was misspelled or incorrect punctuation was used.
VOLUME MISMATCH	7	The volume parameter specified differs from the volume number of the diskette being accessed.
WRITE PROTECTED	4	DOS attempted to access a write protected diskette via one of the following commands: <ul style="list-style-type: none"><li>● SAVE</li><li>● BSAVE</li><li>● WRITE</li></ul>

**APPENDIX G. ASCII CHARACTER SET**

<b>Character</b>	<b>Dec. Value</b>	<b>Hex Value</b>	<b>Keystroke</b>
NUL	0	00	Ctrl-@
SOH	1	01	Ctrl-A
STX	2	02	Ctrl-B
ETX	3	03	Ctrl-C
EOT	4	04	Ctrl-D
ENQ	5	05	Ctrl-E
ACK	6	06	Ctrl-F
BEL	7	07	Ctrl-G
BS	8	08	Ctrl-H
HT	9	09	Ctrl-I
LF	10	0A	Ctrl-J
VT	11	0B	Ctrl-K
FF	12	0C	Ctrl-L
R	13	0D	Ctrl-M
SO	14	0E	Ctrl-N
SI	15	0F	Ctrl-O
DLE	16	10	Ctrl-P
DC1	17	11	Ctrl-Q
DC2	18	12	Ctrl-R
DC3	19	13	Ctrl-S
DC4	20	14	Ctrl-T
NAK	21	15	Ctrl-U
SYN	22	16	Ctrl-V
ETB	23	17	Ctrl-W
CAN	24	18	Ctrl-X
EM	25	19	Ctrl-Y
SUB	26	1A	Ctrl-Z
ESC	27	1B	ESC
FS	28	1C	Ctrl-\
GS	29	1D	Ctrl-]
RS	30	1E	
US	31	1F	Ctrl-Shift--

This appendix lists ASCII codes 0 through 127. ASCII codes 128 through 255 repeat codes 0 through 127.



Character	Dec. Value	Hex Value	Keystroke
SP	32	20	Space Bar
!	33	21	Shift-1
"	34	22	Shift-'
#	35	23	Shift-3
\$	36	24	Shift-4
%	37	25	Shift-5
&	38	26	Shift-7
'	39	27	
(	40	28	Shift-9
)	41	29	Shift-0
*	42	2A	Shift-8
+	43	2B	Shift-=
,	44	2C	,
-	45	2D	-
.	46	2E	.
/	47	2F	/
0	48	30	0
1	49	31	1
2	50	32	2
3	51	33	3
4	52	34	4
5	53	35	5
6	54	36	6
7	55	37	7
8	56	38	8
9	57	39	9
:	58	3A	Shift-;
;	59	3B	;
<	60	3C	Shift-,
=	61	3D	=
>	62	3E	Shift-.
?	63	3F	Shift-/

Character	Dec. Value	Hex Value	Keystroke
@	64	40	Shift-2
A	65	41	Shift-A
B	66	42	Shift-B
C	67	43	Shift-C
D	68	44	Shift-D
E	69	45	Shift-E
F	70	46	Shift-F
G	71	47	Shift-G
H	72	48	Shift-H
I	73	49	Shift-I
J	74	4A	Shift-J
K	75	4B	Shift-K
L	76	4C	Shift-L
M	77	4D	Shift-M
N	78	4E	Shift-N
O	79	4F	Shift-O
P	80	50	Shift-P
Q	81	51	Shift-Q
R	82	52	Shift-R
S	83	53	Shift-S
T	84	54	Shift-T
U	85	55	Shift-U
V	86	56	Shift-V
W	87	57	Shift-W
X	88	58	Shift-X
Y	89	59	Shift-Y
Z	90	5A	Shift-Z
[	91	5B	[
\	92	5C	\
]	93	5D	]
^	94	5E	Shift-6
_	95	5F	Shift- -

Character	Dec. Value	Hex Value	Keystroke
'	96	60	'
a	97	61	A
b	98	62	B
c	99	63	C
d	100	64	D
e	101	65	E
f	102	66	F
g	103	67	G
h	104	68	H
i	105	69	I
j	106	6A	J
k	107	6B	K
l	108	6C	L
m	109	6D	M
n	110	6E	N
o	111	6F	O
p	112	70	P
q	113	71	Q
r	114	72	R
s	115	73	S
t	116	74	T
u	117	75	U
v	118	76	V
w	119	77	W
x	120	78	X
y	121	79	Y
z	122	7A	Z
{	123	7B	Shift-[
	124	7C	Shift-\
}	125	7D	Shift-]
~	126	7E	Shift-
⌘	127	7F	Delete

## APPENDIX H. APPLE IIe PRINTER USAGE

### INTRODUCTION

The Apple IIe outputs data to the printer just as it does to the screen. To send data to the printer, a PR# statement specifying the printer card's slot number must be executed to cause data to be output to the printer rather than to the screen. A second PR# statement must be executed if data is to subsequently be sent to the screen rather than the printer.

Either a parallel or serial printer can be used with the IIe using a parallel or serial interface card respectively. Generally, the printer card is placed in either slot 1 or 2.

### Printer Control Codes

Generally, printer output can be modified through the use of control codes. These codes are generally output to the printer via the PRINT statement. Printer control codes can be used to change the page length, line length, character size, character set, as well as a number of other features.

In the example at the end of this section, the form feed character is sent to the printer in line 170 and the character indicating condensed characters is sent in line 190.

Notice that the CHR\$ function is used to send the printer control characters. Since, CHR\$ is not available in Integer BASIC, the printer control codes must be output in some other manner. This is generally accomplished by keying in the control code character within a pair of quotation marks. Generally, the control code is non-printing, so the keystrokes will not be echoed on the screen.

If our example program was written in Integer BASIC, lines 170, 190 and 195 would be modified as follows to send the form feed character, output condensed characters, and turn off the condensed character mode.

```
170 PRINT " "
190 PRINT " "; "THIS IS AN EXAMPLE OF
    CONDENSED CHARACTER PRINTING"
195 PRINT " "
```

Press Ctrl-L  
Press Ctrl-O  
Press Ctrl-R

### **Sending Program Listing to the Printer**

The LIST statement causes program listings to be sent to the screen. However, these can alternatively be sent to the printer by preceding LIST with a PR# command.

#### **Example Program**

```
50 REM THIS EXAMPLE UTILIZED AN EPSON PRINTER
100 REM THIS EXAMPLE MAY NOT WORK IF YOUR ARE
    USING A DIFFERENT PRINTER
110 REM DISPLAY DATA ON THE SCREEN
120 PRINT "THIS IS AN EXAMPLE OF SCREEN OUTPUT"
130 REM OUTPUT DATA VIA PRINTER AS
    NORMAL CHARACTERS
140 PR# 1
150 PRINT "THIS IS AN EXAMPLE OF PRINTER OUTPUT"
160 REM SEND FORM FEED CHARACTER (FF) TO PRINTER
170 PRINT CHR$ (12)
180 REM SEND CONDENSED CHARACTER CODE (SI) TO PRI
190 PRINT CHR$ (15); "THIS IS AN EXAMPLE OF CONDENSED
    CHARACTER PRINTING"
195 PRINT CHR$ (18)
200 REM SEND OUTPUT TO SCREEN
210 PR# 0
220 PRINT "DATA IS AGAIN OUTPUT TO THE SCREEN"
230 END
```

## Appendix I. Machine Language Subroutines

The Apple IIe's operating system has been written as a series of subroutines. These subroutines can easily be used in the user's own programs. In order to use these subroutines, load the 6502's registers with any necessary data. Then execute a jump to subroutine (JSR) to the subroutine's starting address.

Although the starting addresses and functions of the monitor subroutines are identical in the Apple II+ and Apple IIe, the routines themselves are not. For this reason, if a program is to be compatible with both machines, it must only call monitor subroutines at their starting address.

The descriptions of the subroutines will be as follows.

NAME (address)	A = 00	X = 00	Y = 00
description			
⋮			
⋮			

The values of the 6502's accumulator (A), X-register (X), and Y-register (Y) after the execution of the routine are given on the same line as the name and starting address. A "??" indicates that the register's contents have been scrambled. A "--" indicates that the subroutine has not changed the contents of that register. Hex values shall be indicated with a prefix of "\$".

**BELL** (\$FF3A/65338) A = \$87 X = -- Y = --

BELL writes the CONTROL-G (bell) character to the current output device.

**BELL1** (\$FBDD/64477) A = ?? X = ?? Y = --

BELL1 generates a 1000 Hz tone with a duration of 0.1 seconds on the console speaker.

**CLREOL** (\$FC9C/64668) A = ?? X = -- Y = ??

CLREOL clears the text line in which the cursor resides, from the cursor position to the end of line.

**CLREOP** (\$FC42/64578) A = ?? X = -- Y = ??

CLREOP clears the text window from the current cursor position to the bottom of the screen.

**COUT** (\$FDED/65005) A = ?? X = ?? Y = ??

COUT calls the character output subroutine. The character output subroutine (usually COUT1) must have its starting address in locations (\$36-\$37). The character to be output should be in the accumulator.

**COUT1** (\$FDF0/65008) A = ?? X = ?? Y = ??

COUT1 displays the character to be output (accumulator) on the screen. The character will be displayed at the current cursor position. Afterwards, the cursor position will be advanced. COUT1 takes care of control characters, return, linefeed, and bell.

**CROUT** (\$FD8E/64910) A = -- X = -- Y = --

CROUT outputs a carriage return (\$0D) to the current output device.

**GETLN** (\$FD6A/64874) A = -- X = length of line Y = --

GETLN accepts an entire string of characters, storing them in the input buffer (\$200). The prompting character should be stored in location \$33. The X-register will contain the length of the string upon return from the subroutine.

**GETLNZ** (\$FD67/64871) A = -- X = length of input Y = --

GETLNZ sends a carriage return to the output device, and then calls GETLN.

**GETLN1** (\$FD6F/64879) A = -- X = length of input Y = --

GETLN1 is the same as GETLN, except that no prompt is displayed. If the input line is cancelled, either because of too many backspaces or a CONTROL-X, then GETLN will be executed using the prompt in location \$33.

**HOME** (\$FC58/64600) A = -- X = -- Y = --

HOME clears the screen and positions the cursor in the upper left corner of the screen.

**IOREST** (\$FF3F/65343) A = ?? X = ?? Y = ??

IOREST restores the 6502's internal registers to the values stored in locations \$45 to \$49.

**IOSAVE** (FF4A/64354) A = ?? X = ?? Y = --

IOSAVE stores the contents of the 6502's registers into locations \$45 through \$49.

**KEYIN** (\$FD1B/64795) A = inputted key X = -- Y = --

KEYIN pauses for a keypress and then stores the key in the accumulator. KEYIN also randomizes the random number seed at locations \$4E and \$4F.

**MOVE** (\$FE2C/65048) A = ?? X = -- Y = --

MOVE is identical to the monitor's move subroutine. MOVE gets its argument from:

- \$42-\$43 destination address
- \$3C-\$3D starting source address
- \$3E-\$3F ending source address

**PRBL2** (\$F94A/63818) A = -- X = ?? Y = --

PRBL2 outputs from 1 to 256 blanks to the current output device. The X-register should be loaded with the number of blanks to



output before executing this subroutine. X = 00 corresponds to 256 blanks.

**PRBYTE** (\$FD00/64986) A = ?? X = -- Y = --

PRBYTE outputs the hexadecimal value stored in the accumulator to the current output device.

**PREAD** (FB1E/64286) A = ?? X = -- Y = value of controller

PREAD reads the hand control specified in the X-register (0-3). It then returns that value in the Y-register.

**PRHEX** (\$FDE3/64995) A = ?? X = -- Y = --

PRHEX outputs the lower nibble of the accumulator, one hex digit, to the current output device.

**PRNTAX** (\$F941/63809) A = ?? X = -- Y = --

PRNTAX outputs a four digit hex number. The upper two digits must be stored in the accumulator. The lower two digits must be stored in the X-register.

**RDCHAR** (\$FD35/64821) A = ?? X = ?? Y = ??

RDCHAR is another input subroutine that retrieves characters from the standard input subroutine. RDCHAR also interprets ESCAPE codes.

**RDKEY** (\$FD0C/64780) A = character X = -- Y = --

RDKEY calls the input subroutine. The input subroutine's starting address must be stored in locations \$38-\$39. This routine is usually KEYIN.

**READ** (\$FEFD/65277) A = ?? X = ?? Y = ??

READ converts a series of tones into digital data and then stores this data. The following locations must contain pointers before

calling the subroutine.

- \$3C-\$3D first byte to store data in.
- \$3E-\$3F last byte to store data in.

READ also computes and verifies the checksum. The checksum is calculated with a running exclusive-OR.

**SETINV** (\$FE80/65152) A = -- X = -- Y = \$3F

SETINV sets the display format of all successive characters to inverse video.

**SETNORM** (\$FE84/65156) A = -- X = -- Y = \$FF

SETNORM sets the display format of all successive characters to normal.

**VERIFY** (\$FE36/65078) A = ?? X = ?? Y = ??

VERIFY is identical in operation to the monitor's verify command. Before execution of this command, certain pointers must be set up.

- \$42-\$43 destination address
- \$3C-\$3D starting address
- \$3E-\$3F ending source address

**WAIT** (\$FCA8/64680) A = 00 X = -- Y = --

WAIT causes a delay of a specific amount of time. The length of this delay must be loaded into the accumulator. The delay time is calculated using the following formula.

$$\text{Delay} = \frac{1}{2}(26 + 27A + 5A^2) \text{ microseconds}$$

**WRITE** (\$FECD/65229) A = ?? X = ?? Y = ??

WRITE converts digital data into a series of tones for the cassette. Two pointers must be set up before calling WRITE.

- \$3C-\$3D address of first data byte
- \$3E-\$3F address of last data byte

This subroutine also writes a ten second leader and computes a checksum.

## Monitor Subroutines -- Graphics

The monitor contains subroutines that can be used to manipulate the graphics display.

**CLRSCR** ( $\$F832/63538$ ) A = ?? X = -- Y = ??

CLRSCR clears all 48 lines of the low resolution graphics screen to black.

**CLRTOP** ( $\$F836/63542$ ) A = ?? X = -- Y = ??

CLRTOP clears the low resolution graphics display. However, CLRTOP only clears the top 40 lines.

**HLINE** ( $\$F819/63513$ ) A = ?? X = -- Y = ??

HLINE plots a horizontal string of pixels of the color set by SETCOL or NEXTCOL. HLINE is used only with a low resolution display. Prior to calling HLINE, the accumulator must be loaded with the vertical coordinate; the Y-register must be loaded with the leftmost horizontal coordinate; and memory location  $\$2C$  must be loaded with the rightmost horizontal coordinate.

**NEXTCOL** ( $\$F85F/63583$ ) A = -- X = -- Y = --

NEXTCOL adds 3 to the current low resolution color.

**PLOT** ( $\$F800/63488$ ) A = ?? X = -- Y = --

PLOT sets a certain pixel of the low resolution screen to the color last selected by SETCOL or NEXTCOL. The vertical position is loaded into the accumulator. The horizontal position is loaded in the Y-register.

**SCRN** (\$F871/63601) A = color of block X = -- Y = --

SCRN reads the color value at a specific location on the low resolution graphics screen. SCRN places this value into the accumulator. The accumulator should be loaded with the horizontal position. The Y-register should be loaded with the vertical position.

**SETCOL** (\$F864/63588) A = -- X = -- Y = --

SETCOL sets the current low resolution color to the value in the accumulator. The colors and their corresponding values are listed in Table 6-1.

**VLINE** (\$F828/63528) A = ?? X = -- Y = --

VLINE plots a vertical string of pixels of the color set by SETCOL or NEXTCOL. VLINE is used with the low resolution display. Prior to calling VLINE, the accumulator must be loaded with the top vertical coordinate; the memory location \$2D must be loaded with the bottom vertical coordinate; and the Y-register must be loaded with the horizontal coordinate.

**WSI** (\$F3F2/62450) A = ?? X = ?? Y = ??

WSI clears the currently used high resolution screen. WSI clears all 192 lines whether or not a text window is being used.

**WSI1** (\$F3FA/62454) A = ?? X = ?? Y = ??

WSI1 sets the currently used high resolution screen to the most recently plotted color. The subroutine will not work from BASIC unless preceded by a plot.

**APPENDIX J. PROGRAMS ON THE SYSTEM MASTER DISKETTE**

<b>Program Name</b>	<b>Purpose</b>
HELLO	An Applesoft greetings program which is automatically run by DOS.
APPLESOFT	An Integer BASIC greetings program which is automatically run by DOS if Applesoft is unavailable.
BOOT13	A binary program which allows the usage of 13 sector diskettes.
CHAIN	A binary program that allows one Applesoft BASIC program to be loaded and run from within another Applesoft BASIC program file.
CONVERT13	An Applesoft BASIC program that runs MUFFIN, which in turn converts 13-sector diskettes to 16-sector diskettes.
COPY	Used in Integer BASIC to copy diskettes.
COPY-OBJ0	A machine language subroutine called by COPY and COPYA.
COPYA	Used in Applesoft BASIC to copy diskettes.
FID	Used by FILEM.
FILEM	Applesoft program which allows the user to perform a number of DOS functions via a main menu.
FPBASIC	Binary disk file of Applesoft BASIC.

INTBASIC	Binary disk file of Integer BASIC.
LOADER.OBJ	Loads Integer BASIC into RAM.
MASTER	Applesoft program which runs the binary program MASTER.CREATE.
MASTER.CREATE	Used to create system independent diskettes.
MUFFIN	Binary program used to convert diskettes from 13-sector to 16-sector format.
RENUMBER	Applesoft program used to merge two program files or renumber a program's lines.
SLOT#	Applesoft program which identifies the defaults for slot and drive number.
START13	Applesoft program which runs BOOT13



# INDEX

- ABS 82
- Absolute Value 82
- Accumulator 267
- Alternative Character Set 20
- AND 59-61, 82-84
- APPEND 218-219
- Apple Computer Inc 11
- Apple 80-Column Board 273-283
- Apple II 11
- Apple IIe 11, 13
- Apple IIe, cassette input jack 22
- Apple IIe, cassette output jack 22
- Apple IIe, cassette recorder 169-170
- Apple IIe, controller cards 24-25
- Apple IIe, disk drive 169-224
- Apple IIe, power supply 17
- Apple IIe, rear panel 15
- Apple IIe, speaker 17-18
- Apple IIe, system 12
- Apple IIe, video display 18-22
- Apple Writer 28
- Applesoft BASIC 11, 14, 27, 39, 169
- Applesoft BASIC, error messages 287-289
- Applesoft BASIC, prompt 32, 39, 188
- Applesoft BASIC, switch to Integer 188
- Applesoft BASIC, tokens 285
- Applications Programs 27-28
- Arithmetic Expressions 54, 57
- Arithmetic Operators 56
- Arrays 52, 95-96, 171
- ASC 72, 78, 84
- ASCII Code 84, 87, 295-298
- Assembler 26-27
- Assembly Language 27
- Assignment Statements 62, 120
- ATN 84-85
- AUTO 72-73, 85-86, 124
- Autostart Boot 185
- Auxiliary Expansion Slot 273
- Auxiliary Slot 14
- B Parameter 224
- BASIC 273
- BASIC Diskette 181
- BASIC, Applesoft 11, 14, 27, 39, 169
- BASIC, functions 75
- BASIC, Integer 11, 27
- BASICS Diskette 181
- Bits 16
- BLOAD 210-21, 260, 270
- Boolean Expression 52, 82
- Boolean Operators 56, 59-61
- Branching Statement 72
- BRUN 211
- BSAVE 210, 243-244, 266, 270
- Byte Parameter 224
- Bytes 16
- CALL 86-87
- Caps Lock Key 34, 36
- Carriage Return Character 222
- Carriage Return/Line Feed 66
- Cassette Recorder 22-23, 169-171
- Cassette Recorder, input jack 22
- Cassette Recorder, interface 22
- Cassette Recorder, output jack 22
- Cassette, LOAD 171
- Cassette, installation 169
- Cassette, operation 169-170
- Cassette, SAVE 170
- CATALOG 194-196
- CBASIC 40
- Character Set 20
- Character Set, alternative 20
- Character set, primary 20
- Charts 228
- Checksum 205
- CHR\$ 77, 87, 93, 211, 299
- CLEAR 88
- Clearing the Display 278-279
- CLOSE 218, 223
- CLR 88-89
- Coefficient 49



- COLOR 89-90, 134-135, 226-227
- Colors 89, 108, 227
- Colors, graphics 21
- Compiled Code 26
- Compiled Languages 40
- Compiler 26
- Compound Expression 55
- Compuserve 25
- CON 79, 90-91
- Concatenation 76
- Conditional Statement 71-72
- Constants 50
- CONT 79, 91, 95, 157-158
- Control Codes 279-280
- Control-C 79
- Controller Cards 24
- COS 92
- Counter 101-103
- CP/M 14, 25-26, 181, 273
- CPU 11
- Ctrl-D 211
- Ctrl-K 279
- Ctrl-L 278
- Ctrl-P, monitor 261-263, 270
- Ctrl-Q 276-277
- Ctrl-R 276-277
- Ctrl-V 279
- Ctrl-W 279
- Ctrl-Y, monitor 268-269, 271
- Ctrl-Z 279
- Ctrl-] 279
- Cursor Control 277-278
- Cursor Control Keys 35
  
- DATA 63-64, 92-93, 140-141, 144-145
- Data Types 46
- Debugging 98-99
- DEF FN 93-94
- DEL 94-95
- DELETE 203
- DELETE key 34
- Delimiter 63
- Density 178
- DIM 53-54, 95-98, 142
- Dimensions 53-54
- Direct Access 214
- Directory 194-195
- Disk Controller Card 181
- Disk Drive 169-224
- Disk II 11, 23
- Disk II System 181-182
- Disk II System, installation 181, 183-185
- Disk Operating Systems 181
- Diskette 171, 173-174, 199-202
- Diskette, Catalog 194-195
- Diskette, Directory 194-195
- Diskette, double-density 178
- Diskette, double-sided 178
- Diskette, handling rules 179-180
- Diskette, index hole 177
- Diskette, initialization 197-198
- Diskette, inserting 180
- Diskette, master 199-202
- Diskette, quad density 178
- Diskette, removing 180
- Diskette, single-density 178
- Diskette, single-sided 178
- Diskette, slave 199-202
- Diskette, slot number 191-192
- Diskette, volume number 192-194
- Diskette, write protection 178
- Diskette, 13 sectors 187-188
- Display 11
- Display Line 43
- Display Screen, formatting 65
- DOS 26, 151, 169, 175
- DOS 3.3 11, 181
- DOS Commands 189-220
- DOS, autostart boot 185
- DOS, booting 185-187
- DOS, DELETE 203
- DOS, drive specification 190-191
- DOS, Error Messages 292-294
- DOS, filenames 190
- DOS, LOAD 202
- DOS, LOCK 204
- DOS, monitor boot 186-187
- DOS, RENAME 203
- DOS, reserved words 286
- DOS, restoring 187
- DOS, SAVE 202
- DOS, slot specification 191
- DOS, VERIFY 204-205
- Double-Density Diskettes 178
- Dow Jones News & Quotes 25

- Down-Arrow 36
- DRAW 97-98, 108, 235, 244-245
- Drive Specification 190
- DSP 98
- Dynamic RAM 17
  
- Editing 45, 46
- END 42, 79 99
- Error Codes 130
- Error Message 45, 130, 189
- Error Messages, Applesoft BASIC 287-289
- Error Messages, DOS 292-294
- Error Messages, Integer BASIC 290-291
- ESC 4 277
- ESC 8 277
- ESC Key 36
- EXEC 208-210
- EXEC File 208-120
- Executing a Program 43
- EXP 100
- Expansion Slots 13-14
- Exponent 49
- Exponentiation 57
- Expressions 54-56
- Expressions, compound 55
- Expressions, simple 55
  
- File Pointer 219
- File Type 196
- File, locked 196
- File, size 196
- File, unlocked 196
- Filenames 190
- Files, data storage 221-222
- FLASH 100-101, 126, 281-282
- Flashing Format 20
- Floating Decimal Point 47
- Floating Point 27
- Floating Point Language 39
- Floppy Diskettes 171, 173
- FOR 69-71, 101-102, 138
- FORTRAN 27
- FP 188
- FRE 103
- Functions 75-76
  
- Game Controllers 133
- Game I/O Connector 24
- Game Paddles 24
- GET 69, 104
- GO 255
- GO, monitor 267
- GOSUB 72-75, 105-106, 128, 136-138, 146
- GOTO 106, 128-129, 136-137
- GR 107, 225-226
- Graphics 225
- Graphics Mode, high resolution 20-21
- Graphics Mode, low resolution 20-21
- Graphics Modes 225
- Graphics, colors 21, 89, 108
- Graphics, full screen 226
- Graphics, high resolution 97-98, 108-110, 113, 166-168, 232-234
- Graphics, low resolution 107, 111, 152-154, 225-228
- Graphics, shape 97-98, 149, 151-152, 235-244
- Graphics, with text 226-227
- Greeting Program 197-198
  
- Hard Disks 171-172
- Hard Sectoring 176-177
- HCOLOR 107-108, 112-113, 233
- HGR 108-109, 232-233
- HGR2 109-110
- High Resolution Graphics 20-21, 97-98, 108-110, 113, 166-168, 232-234
- Housekeeping 103
- HIMEM 110-111, 123
- HLIN 89, 107, 111, 227-228
- HOME 112, 281-282
- HPLOT 108, 112-113, 234
- HTAB 113-114, 139, 281
  
- IF 71-72, 82, 114-115
- Immediate Mode 40, 41
- IN# 115-116, 186
- Index Hole 176
- Index Variables 70
- INIT 197-199

- Initializing, diskettes 197-198
- INPUT 67-68, 117-118
- INPUT prompt 67-68
- Installation 29
- INT 116, 188
- Integer 48
- Integer BASIC 11, 27, 39, 169
- Integer BASIC, error messages 290-291
- Integer BASIC, prompt 39, 118
- Integer BASIC, reserved words 286
- Integer BASIC, switch to Applesoft 188
- Interpreters 26-27
- Interpreted Languages 40
- INVERSE 116-117, 126, 281-282
- Inverse Command, monitor 261-263, 270
- Inverse Format 20
- Inverse Video 116-117
- IOU Circuit 14
  
- JMP 264-268
- Jobs, Stephen 11
- Joystick 24
- JSR 263, 267
  
- Keyboard 11
- Keyword 44
  
- Language Translators 26
- LEFT\$ 76, 118-119
- LEN 119
- LET 62, 120
- Line Numbers 41, 85-86
- Links 196-197
- LIST 44-45, 120-121
- List Command, monitor 265-266
- Listing a Program 44
- LOAD 121-122, 202
- LOAD, cassette 171
- LOCK 204
- Locked File 195
- LOG 122
- Logical Operators 59-61
- Logo 27
- LOMEM 110, 123-124
- Loop 70-71, 101-106
  
- Low Resolution Graphics 20-21, 107, 111, 152-154, 225-228
  
- Machine Language 249
- Machine Language Programs 263
- Machine Language Subroutine 86-87 110
- Main Board 13
- MAN 86, 124
- Mantissa 49
- MAXFILES 207-208
- Master Diskette 199-202
- Memory Change, monitor 269
- Memory Dump, monitor 252-253
- Memory Examine, monitor 251-252
- Memory Examine, monitor 269
- MID\$ 76, 123-125
- Mini-Assembler 263-264
- Mini-Assembler, activating 264
- Mini-Assembler, program entry 264-265
- Mini-Assembler, return to monitor 265
- Mini-floppy Diskettes 174
- Mixed Modes 22
- MMU Circuit 14
- Mnemonic 263-264
- Modem 25
- MON 205-206
- Monitor 18-19, 169, 205-206, 249-271
- Monitor DOS boot 186-187
- Monitor, Ctrl-P 261-263, 270
- Monitor, Ctrl-Y 268-269, 271
- Monitor, GO 255, 267
- Monitor, activating 249-250
- Monitor, address values 251
- Monitor, changing memory 253-255
- Monitor, command characters 251
- Monitor, commands 251
- Monitor, custom command 268-269
- Monitor, data values 251
- Monitor, deactivating 249-250
- Monitor, inverse command 261, 270
- Monitor, list command 265-266
- Monitor, memory change 269
- Monitor, memory dump 252-253
- Monitor, memory examine 251-252, 269

- Monitor, normal command 261, 270
- Monitor, read command 259
- Monitor, register examine 253, 269
- Monitor, subroutines 301-307
- Monitor, verify 256-259, 261
- Monitor, write command 259
- Move Command, monitor 255-256, 270
- Muffin 187-188
- Multiple Statement Program lines 43, 44
  
- Nested Loop 71, 102
- Nesting 102
- NEW 42, 125-126
- NEXT 69-71, 101-102, 138
- NO END Error 99
- NO TRACE 127
- NOMON 205-206
- NORMAL 126
- Normal Command, monitor 261-263, 270
- Normal Format 20
- NOT 59-61, 126-127
- Null Command 216
- Numeric Data 46-47
  
- ON 128-129, 136-138, 146
- ON, GOSUB 75, 128-129
- ON, GOTO 73, 128-129
- ONERR 129, 131, 145
- OPEN 214, 219-220, 223
- Open Apple Function Key 34, 36
- Operands 55
- Operating System 26
- Operator 55-56
- OR 59-61, 131-132
  
- Paddles 133
- Page-Zero 254
- Parentheses 55-56
- Parallel Communications 24
- Parallel Interface Card 23-24
- PASCAL 27, 181, 273
- PDL 133
- PEEK 78, 133-134
- PFS 28
- PILOT 27
  
- Pixel 225
- Plan 80 28
- PLOT 89, 107, 134, 227
- POKE 78, 135-136
- POP 136
- POS 138-139
- POSITION 219-220
- Power Supply 17
- PR# 66, 140, 186, 299-301
- Primary Character Set 20
- PRINT 65, 77, 139-140, 156
- PRINT, commas 65
- PRINT, semicolon 66
- Printer 23-24
- Printer 298-299
- Program Debugging 98-99
- Program Execution 43
- Program Lines 43
- Program Mode 40, 41
- Prompt 67-68, 188
  
- Quad-Density Diskettes 178
- Quick File II 2
  
- RAM 11, 13-14
- RAM 16
- RAM, dynamic 17
- RAM, static 17
- Random File Access 212-214
- Random File, CLOSE 223
- Random File, OPEN 223
- Random File, READ 223
- Random File, WRITE 223
- Random Numbers 147-148
- READ 63-64, 92-93, 140-141, 145, 216-218, 219-220, 223, 224
- Read Command, monitor 259
- Read/Write Head 176
- RECALL 141-143, 158, 171
- Record Length 212
- Register Examine, monitor 253, 269
- Relational Expression 54
- Relational Operators 56, 58
- REM 144
- Remark Statements 62
- RENAME 203
- Reserved Word 44, 285-286
- Reset Key 37, 80

- RESTORE 64, 140, 144
- RESUME 130-131, 145-146
- RETURN 34, 36, 74, 128, 105-106, 146
- Reverse Video 116-117
- RF Modulator 18-19
- RIGHT\$ 141-147
- RIGHT\$ 76
- RND 147-148
- ROM 11, 13 16
- ROT 149, 235, 244
- RTS 267
- RUN 150
- Run-Time Monitor 26
  
- S 192
- SAVE 150-151, 202-203
- SAVE, cassette 170
- SCALE= 149, 151-152, 235, 244
- Scientific Notation 49-50
- SCRN 152-154, 228
- Scrolling 279
- Sectors 174-177
- Self-Test 31
- Sequential File 212-222
- Sequential 212-213
- Sequential File, CLOSE 218
- Sequential File, OPEN 214
- Sequential File, READ 216-218
- Sequential File, WRITE 215-216
- Serial Communications 24
- Serial Interface Card 23-24
- Shape 149, 151-152
- Shape Table 155, 235-247
- Shape Table, directory 240
- Shape Table, programming 246-247
- Shape Table, saving 243
- Shift 35
- SHLOAD 155, 235, 243-244
- Simple Expressions 55
- Single-Density Diskette 178
- SIN 155
- Slave Diskette 199-202
- Slot 191-192
- Slot Specification 191
- Soft Sectoring 175-177
- Soft Switch 17-18
- Software 25-27
- Solid Apple Function Key 34, 36
- Source Code 26
- Space Bar 35
- SPC 156
- Speaker 17-18
- Special Character Keys 34
- SPEED 156
- SQR 75-76, 157
- Statement 43
- Static RAM 17
- STEP 70, 101-102
- STOP 80, 157-158
- STORE 141-143, 158, 171
- STR\$ 158-159
- String Variables 51-52
- Strings 46, 95-96
- Subroutines 73-75
- Subscript 53
- Subscripted Variables 53
- Super Serial Card 25
- Switch Box 18
- Syntax, DOS 194
- System Master Diskette 11, 181
- System Monitor 249-271
- TAB 139, 159-160
- Tab Key 34, 36
- Tab Stops 65
- Tabbing 281
- Tables 52
- TAN 85, 161
- TEXT 107, 161
- Text Data 46
- Text File 212
- Text Mode 225
- Text Mode, 40 column 20
- Text Mode, 80 column 20
- The Source 25
- THEN 71-72, 82, 114-115
- Tokens 285
- TRACE 127
- TRACE 161-162
- Track/Sector List 196-197
- Tracks 174-175
- Troubleshooting 29
- Truncated 52
- TV Set 18-19
  
- Unary Operators 57
- Unlocked File 195

Up-Arrow 36  
 Uppercase-Restrict Mode 282  
 USR 162  
  
 V 193  
 VAL 104, 163  
 Variable Name 51-52  
 Variables 50-52  
 Variables, subscripted 53  
 Vectors 236-239  
 VERIFY 204  
 Verify Command, monitor 256-259,  
     261  
 Video Display 18  
 Video Display, Inverse 116-117  
 Video Display, reverse 116-117  
 Visicalc 28  
 VLIN 89, 107, **163-164**, 227-228  
 Volume Specification 192-193  
 Volume Specification 192-193, 195  
 VTAB 164-165  
  
 WAIT 165-166  
 Winchester Disks 171-173  
 Wordstar 28  
 Wozniak, Stephen 11  
 WRITE 219-220, **223-224**  
 Write Command, monitor 259  
 Write-Enable Notch 178  
 Write-Protect Notch 178-179  
  
 X-Register 267  
 XDRAW 108, 149, 151, **166-168**, 235,  
     244-246  
 Y-Register 267  
  
 Z80 26

### Special Characters

40-Column Mode 65-66  
 6502 Microprocessor 14  
 6502B Microprocessor 13, 14  
 80-Column Mode 65-66  
 80-Column Board 273-283  
 80-Column Board, activating 274-275  
 80-Column Board, deactivating  
     275-276  
 80-Column Board, selecting 40  
     columns 276-277  
 80-Column Board, selecting 80  
     columns 276-277  
 80-Column Text Card 14

## **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 PC® & XT® USER'S HANDBOOK**  
**IBM BASIC® USER'S HANDBOOK**  
**VIC-20® USER'S HANDBOOK**  
**KAYPRO® USER'S HANDBOOK**  
**COMMODORE 64® USER'S HANDBOOK**





## APPLE IIe® USER'S HANDBOOK

The Apple IIe personal computer has impressive computing capabilities. The APPLE IIe USER'S HANDBOOK provides clear, concise, and complete instructions which allow the user to master these capabilities.

The APPLE IIe USER'S HANDBOOK is written in a simple, concise manner so that even a first-time user can understand the Apple IIe, yet it still contains a wealth of advanced information for the experienced user. A complete guide to the operation and programming of the Apple IIe and its related peripheral equipment is included.

The following topics are covered in detail:

- Installation and Operation
- BASIC Programming
- DOS and Disk II Usage
- Printer Installation and Operation
- Graphics and Sound
- 80 Column Card
- Monitor
- Reference Guide to Integer and Applesoft BASIC Commands
- Reference Guide to DOS Commands

The APPLE IIe USER'S HANDBOOK is a must for any user or potential user of the Apple IIe computer.

