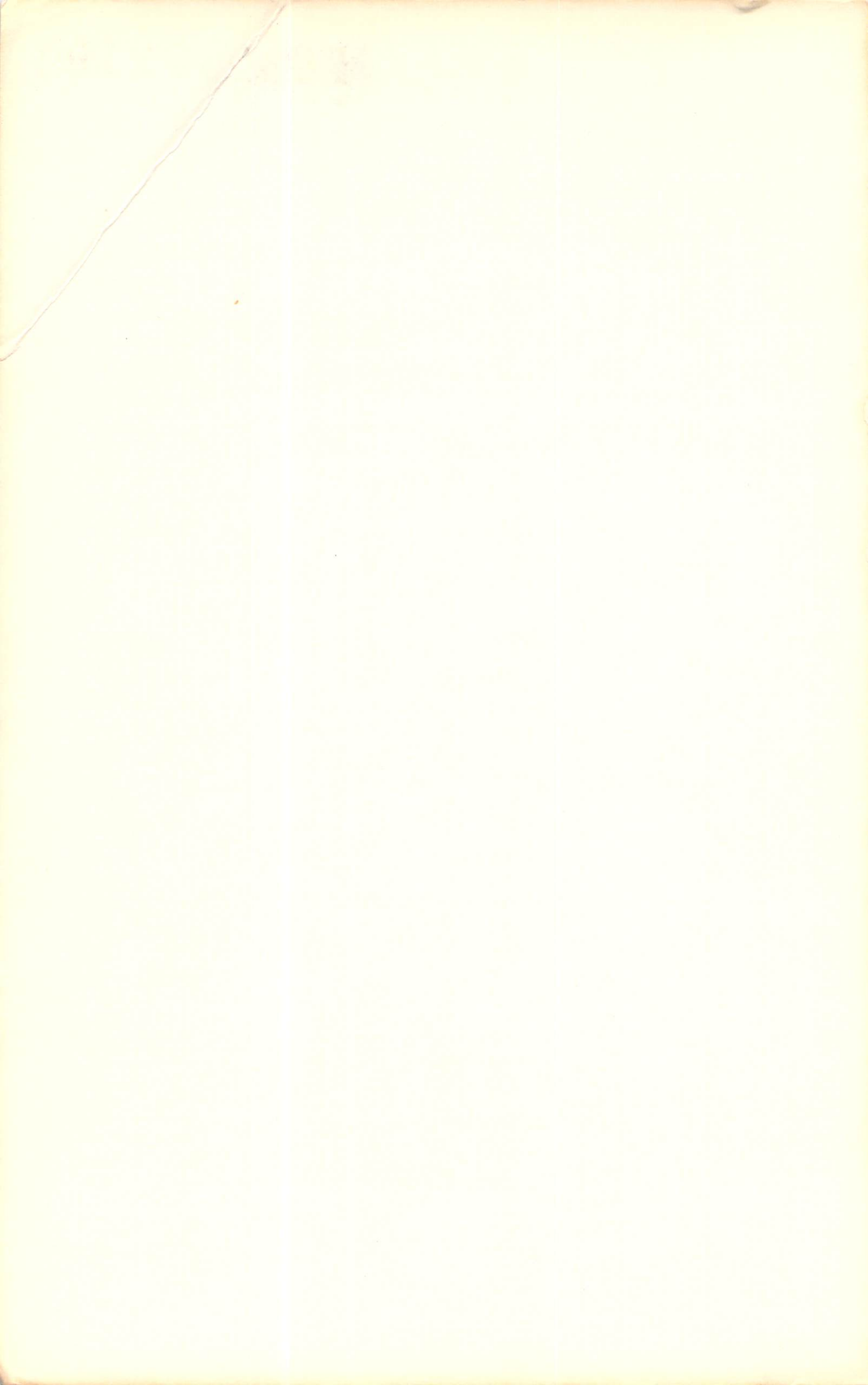
Ballantine/31596/$9.95 in USA • $12.95 in Canada

COMMODORE 64®
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



Commodore 64®
User's Handbook

COMMODORE 64®
USER'S HANDBOOK

Weber Systems Inc. Staff

Ballantine Books • New York

Commodore 64® 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. Originally published by Weber Systems, Inc.

Commodore 64 is a registered trademark of Commodore Business Machines, Inc. **Commodore 64 User's Handbook** is not sponsored, approved, or connected in any manner whatsoever with Commodore Business Machines, Inc. Commodore Business Machines, Inc. makes no warranty, either express or implied with regards to the information contained herein, its accuracy or its completeness.

C= is a registered trademark of Commodore Business Machines, Inc.

Commodore 1541 Disk Drive, Commodore 1540 Disk Drive, Commodore 1525 Printer, Commodore 1515 Printer, Datassette, VIC Modem, Commodore 64 BASIC, Commodore DOS, VIC-20, and PET/CBM are registered trademarks of Commodore Business Machines, Inc.

Library of Congress Card Number: 83-91224

ISBN: 0-345-31596-0

Manufactured in the United States of America

First Ballantine Books Edition: March 1984

10 9 8 7 6 5 4 3 2 1

CONTENTS

1. INTRODUCTION TO THE COMMODORE 64 and ITS PERIPHERALS 9

Overview 9. Specifications 11. Computer Memory 12. Peripherals & Accessories 14. Data Cassette Unit 14. Commodore 1541 Disk Drive 14. Commodore 1525 Printer 16. VIC Modem 16. Interface Cartridges 16. Software 17. Operating Systems 17. Languages 17. Applications Programs 18.

2. INSTALLATION & OPERATION OF THE COMMODORE 64 19

Installation 19. Connecting to a TV Set 19. Connecting to a Monitor and/or Audio System 21. Troubleshooting 23. Adjusting the Color on the Video Display 24. Keyboard 25. Restore Key 26. Return Key 26. Shift Keys 27. Commodore Key 27. Lower Case Characters 28. Cursor Up/Down Key 28. Cursor Right/Left Key 28. Clear/Home Key 29. Insert/Delete Key 29. Control Key 29. Color Keys 30. Reverse On/Reverse Off Keys 30. Run/Stop Key 30. Function Keys 30. Screen Display 31. Correcting Keyboard Entry Errors 31. Error Messages 31. Datassette Recorder 31. 1541 Disk Drive 33. Diskette Handling Procedures 33. Write Protecting a Diskette 35. Inserting a Diskette 35. DOS 35. DOS Commands 35. Formatting Blank Diskettes 36. 1515 and 1525 Printers 36. Loading and Executing Programs 36. Loading Package Programs 37.

3. BASIC PROGRAMMING FOR THE COMMODORE 64 39

Introduction 39. Compiled vs. Interpreted Languages 39. Immediate & Program Modes 40. Line Numbers 41. NEW Command 42. END Statement 43. Executing a Program 43. Program Lines & Display Lines 43. Multiple Statement Program Lines 43. Abbreviating Keywords 44. Listing a Program 45. Error Messages 46. BASIC Data Types 46. Strings 46. Numeric Data 47. Floating Decimal Point 47. Floating Point Numbers 48. Integers 48. Scientific Notation 49. Variables 50. BASIC Variables 50. BASIC Variable Names 51. Tables and Arrays 53. Expressions and Operators 55. Compound Expressions and Order of Evaluation 56. Arithmetic Operations 58. Relational Operators 59. Logical Operators 60. BASIC Statements 63. Remark Statements 63. Assignment Statements 64. DATA, READ Assignment Statements 64. Outputting Data 66. INPUT Statements 67. Loops 70. Nested Loops 71. Conditional Statements 72. Branching Statements 72. ON, GOTO Statement 74. Subroutines & GOSUB Statements 74. ON, GOSUB Statement 75. Commodore BASIC Functions 76. String Concatenation 77. ASCII 77. CHR$ and ASC Functions 78. PEEK & POKE 79. Advanced Input and Output Statements 80. Device Numbers 80. Input/Output Channels 81. OPEN 81. PRINT#, GET#, INPUT# 81. Bits and Bytes 82.

4. COMMODORE 64 BASIC REFERENCE GUIDE 85

ABS 86. AND 86. ASC 87. ATN 88. CHR$ 88. CLOSE 89. CLR 89. CMD 90. CONT 91. COS 91. DATA 92. DEF FN 93. DIM 94. END 95. EXP 96. FN 96. FOR 97. FRE 100. GET 100. GOSUB 102. GOTO 104. IF 106. INPUT 107. INT 109. LEFT$ 109. LEN 110. LET 111. LIST 111. LOAD 112. LOG 114. MID$ 115. NEW 116. NEXT 116. NOT 117. OPEN 119. OR 120. PEEK 122. POKE 122. POS 123. PRINT 124. READ 128. REM 129. RESTORE 129. RETURN 130. RIGHT$ 130. RND 131. RUN 132. SAVE 133. SGN 134. SIN 134. SPC 135. SQR 135. STOP 136. STR$ 136. SYS 137. TAB 137. USR 138. VAL 139. VERIFY 139. WAIT 141.

5. **COMMODORE 64 DATASSETTE CASSETTE RECORDER** 143

Introduction 143. Installing the Datassette 143. Data Files -- Files, Records, & Fields 145. Program Files 145. Prompts 145. Saving a Program File 146. Verifying a Program File 147. Loading a Program File 149. Reading and Writing Data to the Datassette 150. Opening Data Files 150. Closing Data Files 152. Writing to a Cassette Data File 152. Reading Data from a Cassette Data File 155.

6. **1541 DISK DRIVE** 157

Introduction 157. Types of Disks 158. Hard Disks 158. Winchester Disk Drives 158. Floppy Diskettes 158. Tracks and Sectors 160. Hard and Soft Sectors 162. Single and Double Sided Diskettes 164. Diskette Write Protection 164. Additional 1541 Features 165. Installing the 1541 166. Powering On 168. Inserting a Diskette into the 1541 169. Using the 1541 with the VIC-20 and Commodore 64 170. Disk Files 170. Filename Match Characters 171. Sequential vs. Random Access 172. Block Availability Map 172. Disk Directory 174. Loading & Saving Programs on Diskette 177. Loading Packaged Programs 177. Formatting a Diskette 177. Saving a Program File 178. Saving and Replacing a Program File 179. Loading a Program File 182. Verifying a Program File 183. DOS Commands 184. OPEN 184. PRINT# 185. NEW 185. COPY 186. RENAME 186. SCRATCH 187. INITIALIZE 188. VALIDATE 188. CLOSE 189. Error Channel 189. File Access 190. Sequential Access 190. Opening a Sequential File 191. PRINT# -- Sequential Files 192. INPUT# -- Sequential Files 193. GET# -- Sequential Files 194. Random Files 195. OPEN -- Random Files 196. BLOCK-WRITE 196. BLOCK-READ 198. BLOCK-ALLOCATE 199. BLOCK-FREE 200. BUFFER POINTER 201. USER1 202. USER2 203. Relative Files 205. Creating a Relative File 205. Opening a Relative File 205. Positioning the File Pointer 206. Programming the Disk Controller 209. BLOCK-EXECUTE 209. MEMORY-READ 209. MEMORY-WRITE 210. MEMORY-EXECUTE 210. USER Commands 211.

7. COMMODORE 64 PRINTER INSTALLATION & OPERATION 213.

Introduction 213. Installation 213. BASIC Statements 218. OPEN 218. CLOSE 219. CMD 220. PRINT# 221. Control Codes 223. Graphics -- CHR$(8) 225. Line Feed -- CHR$(10) 225. Carriage Return -- CHR$(13) 226. Wide -- CHR$(14) 226. Standard -- CHR$(15) 226. Tab -- CHR$(16) 227. Cursor Down -- CHR$(17) 228. Reverse -- CHR$(18) 229. Repeat -- CHR$(26) 229. Dot Address -- CHR$(27) & CHR$(16) 231. Cursor Up -- CHR$(145) 232. Reverse Off -- CHR$(146) 233.

8. COMMODORE 64 SOUND & GRAPHICS 235.

Display Colors 235. Text Colors 236. Graphics Characters 236. PRINT Statements 237. Screen Locations 238. POKE Statements 240. Moving Characters 240. The Character Set 241. Changing Characters 241. High Resolution Graphics 244. Bit Maps 244. Sprites 250. Defining a Sprite 250. Controlling a Sprite 251. Activating a Sprite 252. Locating a Sprite 252. Expanding Sprites 253. Color Sprites 253. DATA Assignments 253. Sound 256. Voices 256. Waveshapes 257. Frequency 256. Pulse Waves 259. Envelopes 260. Attack-Decay 261. Sustain-Release 262. Music 264.

Appendix A. Commodore 64 BASIC Error Messages 267

Appendix B. Commodore 64 Code Set 278

Appendix C. Screen Codes 282

Appendix D. Commodore BASIC Reserved Words 284

Appendix E. Musical Notes 286

Appendix F. Commodore 64 Memory Map 288

Appendix G. Useful Memory Locations 294

Appendix H. BASIC Keyword One Character Tokens 299

Index 301

CHAPTER 1.
INTRODUCTION TO THE COMMODORE 64 & ITS PERIPHERALS

INTRODUCTION

In this book, we will describe the Commodore 64 personal computer as well as many of the peripherals available for use with the 64, such as the Commodore 1541 disk drive, Datassette cassette recorder, and Commodore 1525 printer.

In chapter 1, we will discuss the features of the 64 and its peripherals. In Chapter 2, we will discuss the installation and operation of the 64 and its peripherals. A section on troubleshooting the 64 is included.

In Chapter 3, we will discuss programming the 64 in the BASIC language. Chapter 4 consists of a reference guide to the various BASIC commands and statements available for use with the 64.

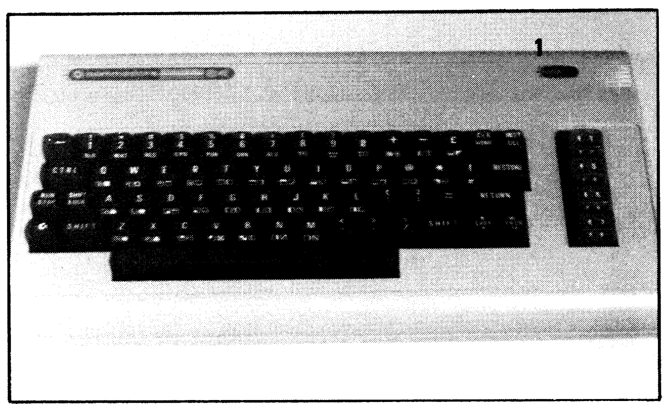
Chapters 5, 6, and 7 describe the installation, operation, and programming of the Datassette cassette recorder, the 1541 disk drive, and the 1525 printer, respectively.

Finally, Chapter 8 deals with the creation of graphics and sound on the Commodore 64.

COMMODORE 64 OVERVIEW

The Commodore 64 is pictured in Illustrations 1-1, 1-2, and 1-3. Notice the following features in these Illustrations.

Illustration 1-1. Commodore 64 (Top View)



1. Power Light

Keyboard--Used for inputting instructions and information into the 64.

Power Light--Indicates whether the 64 is powered on.

Power Cord Socket--Used to connect the power supply to the 64.

On/Off Switch--Used to power on/power off the 64.

Game Ports--Used to connect joysticks or other game controller devices.

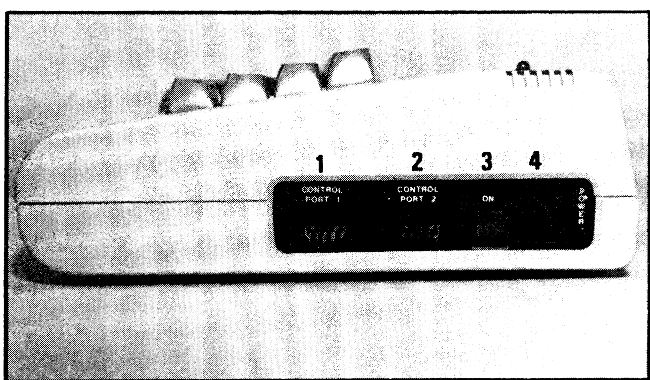
Cartridge Slot--Used to accept program or game cartridges.

Channel Selector--Used to select the TV channel on which the computer's video output will be displayed.

TV Connector--Used to supply both sound and video to a television set connected to the 64.

Audio & Video Output--Used to supply separate audio and video signals. The audio signal can be output through a high quality sound system. The video signal can be sent to a monitor.

Illustration 1-2. Commodore 64 (Side View)



1 & 2. Control Ports 3. On/Off Switch 4. Power Cord Socket

Serial Port--Used to connect either a single disk drive or a printer.

Cassette Interface--Used to connect the Datassette cassette recorder.

User Port--Used to attach various interface cartridges such as the RS232 communication cartridge or the VIC-MODEM.

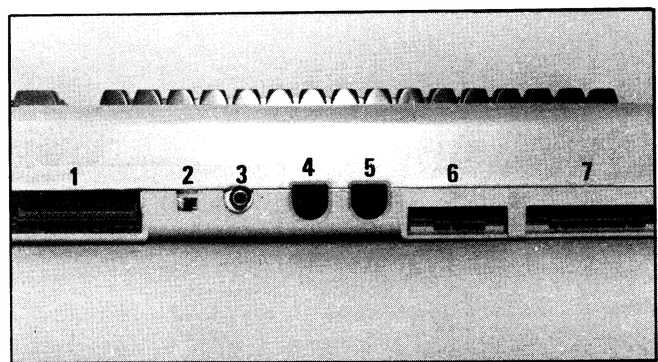
Commodore 64 Specifications

The 64 comes with the following components and accessories:

- Commodore 64 Computer Console
- AC Power Adapter (see Illustration 1-4)
- Video Cable (see Illustration 1-4)
- TV Switch Box (see Illustration 1-4)

The AC Power Adapter converts regular AC current to a low voltage that can be used by your 64. The AC Power Adapter can be plugged into any normal household outlet. The AC

Illustration 1-3. Commodore 64 (Rear View)



1. Game Port 2. Channel Selector 3. RF Modulator Output 4. Audio/Video Output 5. Serial Port 6. Cassette Interface 7. User Port

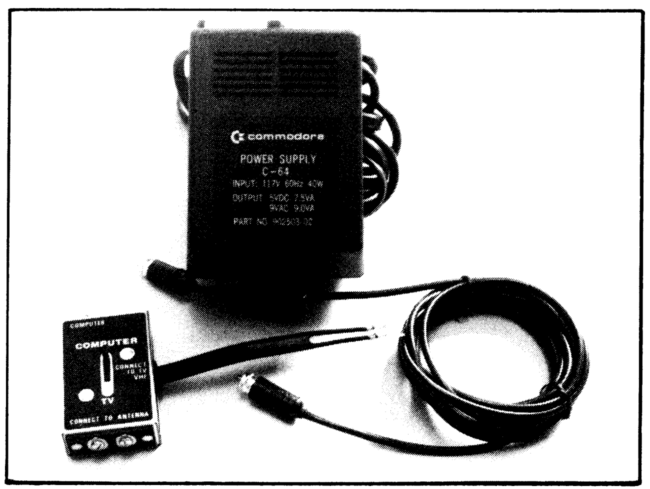
Power Adapter's small plug should be connected to the Power Cord Socket (see Illustration 1-2).

The video cable is used to connect the 64 to a TV set. When a TV set is used for video display, the video cable must be connected to a TV Switch Box, which is in turn connected to the television set. The 64 has a built-in RF modulator. An RF Modulator converts the 64's video signal so that it is compatible with a TV set.

Computer Memory

Computer memory is measured in units known as **bytes.** A byte is used to store a single character in the computer's memory. Bytes are represented in units of measurement known as **kilobytes** or **K.** 1K is the equivalent of 1024 bytes. Your 64 is supplied with 64K of RAM.

Illustration 1-4. AC Power Adapter, Video Cable, TV Switch Box



Computer memory can be one of two types; **ROM** or **RAM.** ROM stands for read-only memory. ROM will hold the data stored in it permanently. If the power to the 64 is shut off, the information stored in ROM will remain there. ROM contains the programs which are used to operate the 64 and allow it to interact with the user.

RAM stands for random-access memory.* The data stored in RAM can be changed. Applications programs are often transferred from diskettes or cassette to RAM. Any data stored in RAM is lost when the 64's power is turned off.

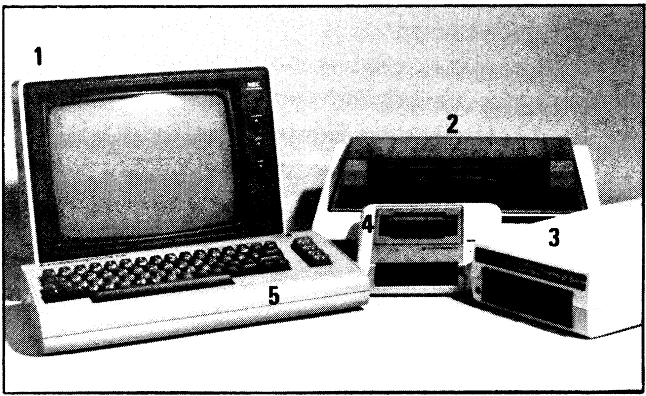
---

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

Commodore 64 Peripherals & Accessories

A complete Commodore 64 system is depicted in Illustration 1-5. Notice the available peripherals. These will be explained in the following sections.

Illustration 1-5. Commodore 64 System



1. Monitor 2. 1525 Printer 3. 1541 Disk Drive 4. Datassette Cassette Recorder 5. Console

Datassette Cassette Unit

Cassette tape can be used to store programs held in RAM. These programs can then later be transferred back into RAM from cassette tape. The Commodore Datassette, as pictured in Illustration 1-6, is designed for use with the 64. Several thousand characters can be stored on a standard cassette tape by using the Datassette.

Commodore 1541 Disk Drive

A disk drive is a much more efficient device for storing data than a cassette recorder. A disk drive allows greater storage capacity, quicker access to data, as well as fewer errors in data transfers.

The 1541 disk drive is designed for use with the Commodore 64. The 1541 disk drive is virtually identical to the 1540 disk drive. Both can be used with the Commodore VIC-20 computer. However, the 1540 disk drive must be modified by a Commodore dealer for use with the Commodore 64.

Illustration 1-6. Datassette Cassette Recorder

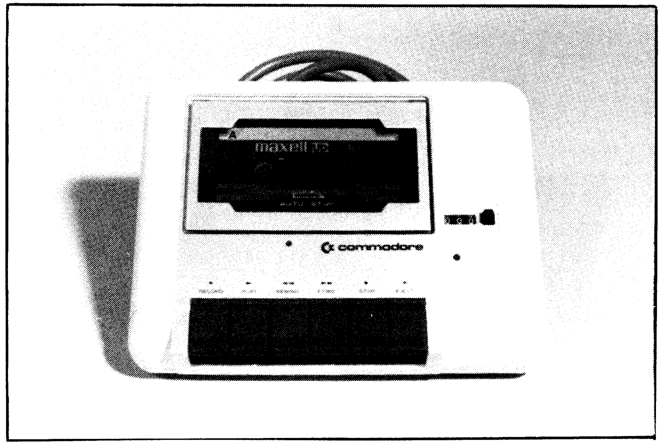
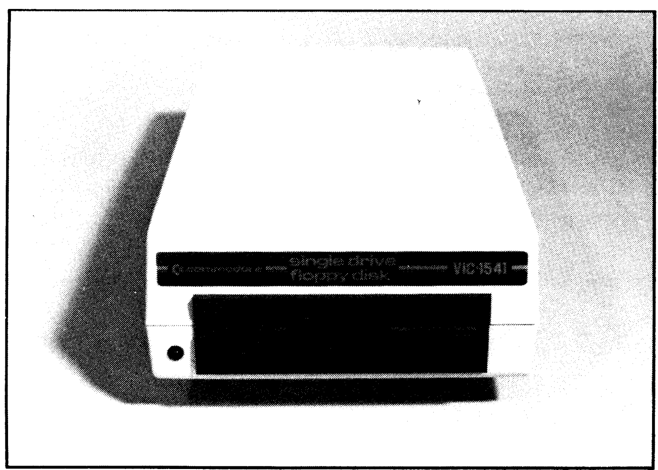


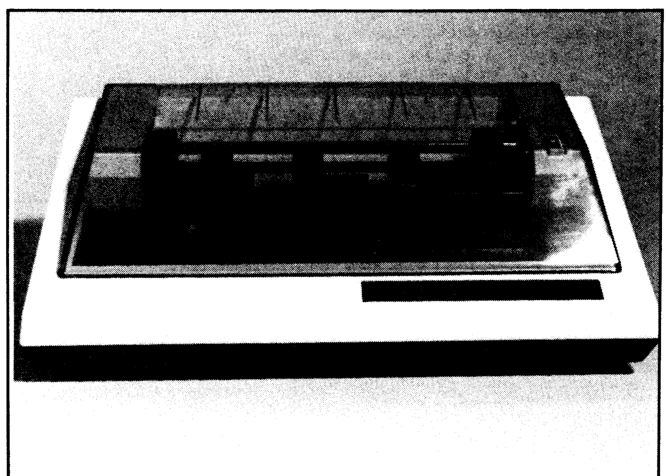
Illustration 1-7. Commodore 1541 Disk Drive



Commodore 1525 Printer

The Commodore 1525 printer can be used with the Commodore 64. The 1525 printer is virtually identical to the 1515, except for the fact that the 1515 can only be used with the VIC-20. The 1525 can be used with either the 64 or the VIC-20. The 1525 printer is pictured in Illustration 1-8.

Illustration 1-8. Commodore 1525 Printer



VIC Modem

The VIC Modem is a communications device which allows the Commodore 64 to communicate with other computers over telephone lines. The VIC Modem allows Commodore 64 owners to access large data bases such as the Source, Dow Jones News Retrieval Service, and Compuserve.

Interface Cartridges

Various interface cartridges are available for the Commodore 64 which allow a number of peripheral devices

such as printers, controllers, modems, and instruments to be connected to the 64.

An IEEE-488 Cartridge is available which allows the Commodore 64 to use a number of Commodore CBM peripherals such as CBM printers and disk drive units.

It is anticipated that Commodore will announce a Z80 cartridge shortly for use with the 64. This cartridge will allow the 64 to run CP/M. CP/M is perhaps the most widely used personal computer operating system. A great number of applications programs are available which run under CP/M.

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 64 operating system is stored permanently in ROM.

Languages

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

BASIC is the high-level language generally used with the 64. The 64's BASIC interpreter is contained in ROM.

Applications Programs

Applications programs are those written to accomplish a specific task. Examples of applications programs are games, word processing programs, financial forecasting programs, 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.

Applications programs for the 64 can also be stored in a permanent form on a ROM cartridge. This ROM cartridge can be plugged into the Cartridge Slot.

CHAPTER 2.
INSTALLATION and OPERATION OF THE COMMODORE 64

INSTALLATION

First of all, when you unpack your Commodore 64, save the carton and packing material. These should be used if the 64 is to be moved or stored.

The 64 is easy to install. First of all, position your 64, TV set or monitor, and any peripherals so that they are easily accessed. You will need two AC electrical outlets -- one for the 64, and the second for your TV set or video monitor.

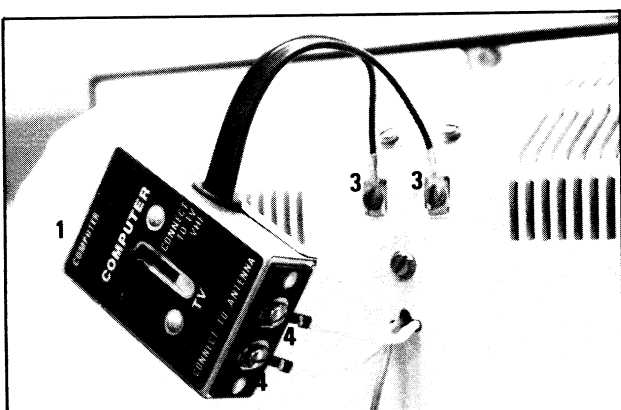
Locate the 64's On/Off switch on the right hand side of the console and be certain that it is positioned to Off. Next, plug the power supply unit's cord into an AC electrical outlet. Plug the other end into the 64's Power Cord Socket.

Connecting the Commodore 64 to a TV Set

Your next step is to connect the 64 to either a television set or a monitor. To connect the 64 to a TV set, first connect the video cable to the TV Connector on the rear of the 64. Since both ends of the video cable are identical, either can be connected to the TV Connector.

Your next step is to install the TV Switch Box to your TV set (see Illustration 2-1). The TV Switch Box has been designed so that it can be permanently installed on your TV set, as it allows regular TV reception as well as video output for the 64. The Switch Box has an adhesive backing that can be used to attach it to the back of your television.

Illustration 2-1. TV Set/Commodore 64 Installation



1. Switch Box 2. Computer/TV Switch 3. VHF Hookup
4. Antenna Hookup

The Switch Box contains a switch marked Computer/TV. When this switch is at the Computer position, the TV set receives its signals from the 64. When the switch is set to the TV position, the TV set receives its signals from your television antenna.

To install the TV Switch Box, first disconnect your television antenna from the VHF terminals at the back of your television. Then, connect the two wires leading from the Switch Box to the twin VHF terminals and tighten the screws.

Next, connect the video cable to the Switch Box. Be certain that the switch in the center of the Switch Box is set to "Computer" and turn on your TV set and 64.

Your TV set should be tuned into VHF channel 3 or 4--whichever corresponds to the setting of the 64's Channel Selector. If you wish channel 3, move the Channel Selector to the left. Channel 4 can be selected by moving the Channel Selector to the right. Select whichever channel has the weakest reception in your area.

When the 64 has been properly installed and powered on, the following screen display will appear.

```
****COMMODORE 64 BASIC V2****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY
■
```

If you wish to have the option of viewing either your television set or displaying the 64's output without having to disconnect the Switch Box, you can do so by connecting your VHF antenna leads to the terminals with the label "connect to antenna".

If your antenna is the round 75 ohm coax variety, you will need a 75 ohm to 300 ohm converter to connect your antenna cable to the RF modulator. See Illustration 2-2 for an example of a 75 ohm to 300 ohm converter.

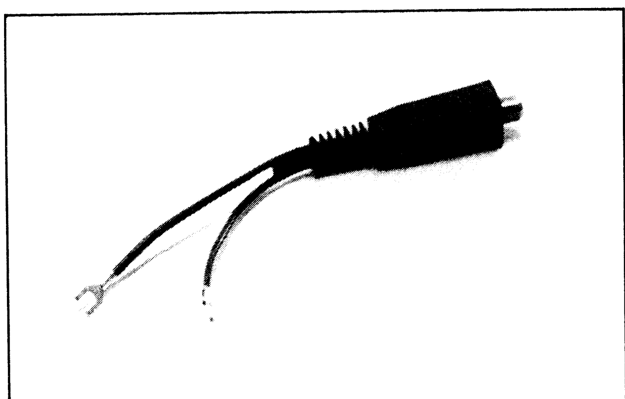
Once the VHF antenna leads have been connected to the RF modulator, your TV set can be viewed when the switch on the Switch Box is set to the TV setting.

Connecting the Commodore 64 to a Monitor and/or Audio System

The Commodore 64 can also be connected to a monitor for visual output and/or to a stereo system for audio output.

This connection is accomplished through the Audio and Video Output jack on the rear panel of the 64. A standard

Illustration 2-2. 75 Ohm to 300 Ohm Adapter



5-pin DIN audio cable can be used to make the physical connection. This cable is not supplied with the 64, but can be obtained from most electronics or audio stores.

The DIN cable connects directly to the Audio & Video Output jack. Two of the four pins on the opposite end of the cable carry the audio and video signals. The black connector carries the audio signal. This should be connected to either the Audio In jack of the monitor, the Audio In jack of a video cassette recorder, or the Auxiliary input of a stereo amplifier or receiver.

The white or red connector carries the video signal. This should be connected to the Video In jack of a monitor or a video cassette recorder.

Your 64 is now ready for operation. Should you have additional peripherals such as a Datassette recorder, 1541 Disk Drive, or 1525 printer, consult Chapters 5, 6, and 7 respectively for installation instructions.

TROUBLESHOOTING THE COMMODORE 64

If you do encounter a problem with your 64, chances are that you may be able to solve the problem yourself. Use Table 2-1 as an aid in finding the cause of the problem and its solution.

Table 2-1. Commodore 64 Troubleshooting Guide

Problem	Problem Origin	Problem Solution
No video display with power light off.	AC Power Adapter unplugged from AC outlet	Check connection between wall outlet and AC Power Adapter.
	Commodore 64 not connected to AC Power Adapter	Check connection between Power Socket and AC Power Adapter.
	Commodore 64 Power Switch is off.	Turn Power Switch to On position.
	Bad fuse	Take the 64 to an authorized Commodore Service Center for replacement of fuse.
No video display with power light on.	TV tuned to the wrong channel	Tune your TV to Channel 3 or 4--whichever corresponds to the setting on the Channel Selector.
	Video Cable not connected	Check connection between the 64 and the Video Cable. Also, check the connection between the Video Cable and the Switch Box.
	Switch Box not properly connected	Be certain that the Switch Box is properly connected. Check to be certain that the twin wires are connected to the VHF terminals--not the UHF.
	Switch Box Switch set incorrectly	Be certain that the switch in the center of the Switch Box is set to "Computer" when the 64 is in use.
Poor quality color or no color on TV set.	TV is improperly tuned or color is adjusted incorrectly.	Tune TV and/or adjust color.

Adjusting the Color on the Video Display

You may find it necessary to adjust your TV set so that it outputs the colors specified by the 64. By following a simple procedure, this adjustment can be made.

First of all, press the L key on your 64 several times. Then, locate the Control key on the upper left-hand side of the keyboard. Press the Control and 9 keys simultaneously, and then release them both.

After pressing Ctrl-9*, you might notice that no output appeared on the video screen. However, that does not mean that the video screen was not affected by Ctrl-9. Again, press the L key several times. Notice that L is displayed in reverse. Pressing Ctrl-9 causes subsequent screen characters to be displayed in reverse.

Next, press the space bar and hold it down. A light blue line will begin filling the current row on the screen, moving from left to right.

Release the space bar and press Ctrl-1. Again, press the space bar and hold it down. The line will again fill the screen row, however it will be black in color instead of light blue.

Try pressing Ctrl-2 and then hold down the space bar. The line will now be white.

Notice that the number keys (1-9) each have a color indicated on the front of the key. By pressing the Control key and the corresponding number key, the desired color will be output when the space bar is pressed.

---

*Ctrl-9 denotes pressing the 9 key while holding down the Control key.

If the colors displayed on your screen are incorrect, adjust the color and tint controls on your TV set so that they match the colors noted on the number keys.

To return to the normal screen display, hold down the Run/Stop key and press Restore. The screen will return to its normal display.

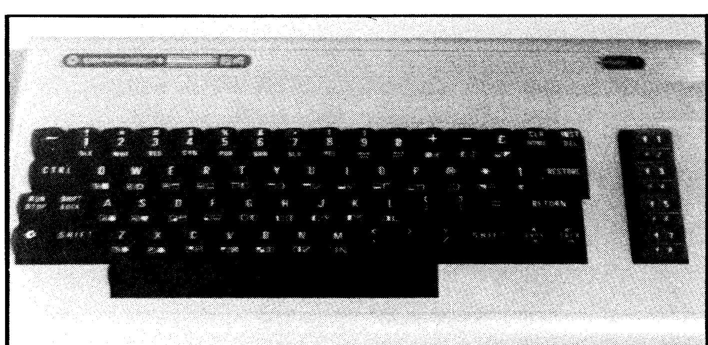
COMMODORE 64 KEYBOARD

The 64 keyboard (see Illustration 2-3) contains many of the same keys arranged in the same order as a regular typewriter. The 64 also contains several additional keys not found on a regular typewriter keyboard. These will be explained in detail later in this section.

Also, as you have probably already noted, many of the keys on the 64 keyboard contain one or two identification symbols on the front side of the key as well as on top of the key. We will explain the procedures for generating the characters displayed on the front side of these keys later in this section.

In the next sections, we will discuss the usage of all of the keys on the 64 keyboard. We recommend that you experiment with these keys as you read these sections. Do not worry about damaging the computer. Any error situation caused by keyboard entries can be corrected by merely turning the 64 off and then on again.

Illustration 2-3. Commodore 64 Keyboard



Restore Key

The Restore key function is used to reset the 64. When the 64 is reset, the computer reacts much as it does when it is first turned on. The screen is cleared when the 64 is reset. However, any programs stored in RAM are kept there, and can be listed or run.

To reset the 64, hold down the Run/Stop key and press the Restore key.

Return Key

As characters are entered via the keyboard, these characters are displayed on the video screen and also saved in memory. However, these characters are not actually interpreted by the computer until the Return key has been pressed. The Return key tells the 64 that the line into which characters are being typed has been finished.

When Return is pressed, the 64 will review the line just entered for errors. If any errors are found, an error message will be displayed.

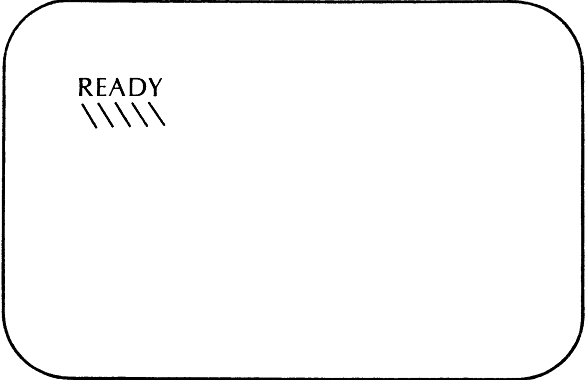
Shift Keys & Shift Lock Key

Like a typewriter, the 64 has two Shift keys as well as a Shift Lock key. However, unlike a typewriter, the keys for the letters (A-Z) generally produce upper-case letters on the 64 upon start-up when the Shift key is released.

If the Shift key is depressed with the key for a letter (A-Z) or one of the following symbols,

+ – £ @ * ↑

the graphics symbol on the right hand side of the front of the key will be generated. For example, if Shift-M was pressed five times, the following output would result.



When the Shift key is pressed with one of the function keys (f1 - f8), the function noted on the front of the key (f2, f4, f6, f8) will be generated.

Commodore Key

The Commodore key is located on the lower left hand side of the 64 keyboard and has the following symbol printed on its top.

As we mentioned in the last section, the Shift key is used to generate the graphics symbol on the right hand side of the front of the various graphics keys. The Commodore key is used to generate the graphics symbol on the left hand side of the front of these graphics keys.

Lower Case Characters

If both the Shift and Commodore keys are pressed simultaneously, the 64 is placed in the **text** mode. The text mode allows the user to generate lower case as well as upper case characters. However, only those graphics characters on the left hand side of the front of the graphics keys are available.

Lower case characters are generated by pressing keys in the Unshift mode. Upper case characters are generated by pressing keys in the Shift mode.

The graphics characters on the left hand side of the front of the graphics key are generated by pressing the Commodore key and the graphics key desired.

Cursor Up/Down Key

The Cursor Up/Down key is located to the immediate right of the shift key on the right hand side of the 64 keyboard. In the Unshift mode, the Cursor Up/Down key moves the cursor down one row every time it is pressed. In the Shift mode, the cursor is moved up one row every time Cursor Up/Down is pressed.

Notice that the Cursor Up/Down key has a **repeat** feature. That is, when the Cursor Up/Down key is pressed, the cursor movement function will be repeated until the key is released. This is also the case with the Cursor Right/Left key.

Cursor Right/Left Key

The Cursor Right/Left key moves the cursor to the left (in the

shift mode). Notice that both the Cursor Right/Left and Cursor Up/Down keys do not affect any characters over which they pass.

Clear/Home Key (CLR/HOME)

In the Unshift mode, the Clear/Home key moves the cursor to the screen's home position (the upper left hand corner). In the the Shift mode, pressing the Clear/Home key clears the screen as well as moves the cursor to the home position.

Insert/Delete Key (INST/DEL)

In the Unshift mode, pressing the Insert/Delete key causes the character to the immediate left of the cursor to be deleted. If the cursor is positioned in the middle of a line of characters when Insert/Delete is pressed, the character to the immediate left of the cursor will be deleted. All characters to the right of the cursor will move one position to the left to fill the space vacated by the character being deleted.

In the Shift mode, pressing the Insert/Delete key results in a space being inserted to the immediate right of the cursor. If the cursor is positioned within a line of characters, pressing Insert/Delete in the Shift mode will open up a space in the line into which a new character can be inserted.

The primary usage of the Insert/Delete key is in correcting keyboard entry errors.

Control Key (CTRL)

The Control Key can be used in conjunction with other keys to perform special functions. These functions can be defined within an applications program.

The Control key can also be used in combination with the Color keys to select Color keys to be output on the video display. The Control key must be depressed concurrently with the desired Color key.

Color Keys

The eight Color keys are located at the top of the keyboard. By pressing the Control or Commodore key simultaneously with a Color key, any subsequent character will be displayed in the color selected.

Reverse On/Reverse Off Keys (RVS ON, RVS OFF)

By pressing the Control and Reverse On keys simultaneously, all characters input via the keyboard will be displayed in reverse. For example, if the L key was depressed with Reverse On, the L would be displayed in dark blue against a light blue background. This is exactly the opposite of the normal display.

If the Control and Reverse Off keys are pressed simultaneously, normal output will resume.

Run/Stop Key

In the Unshift mode, Run/Stop instructs the 64 to stop whatever operation it is performing and to return control back to the operating system. Run/Stop in the Unshift mode generally is used to halt a program.

In the Shift mode, pressing Run/Stop instructs the 64 to begin loading a program from the Datassette tape cassette unit.

Function Keys (f)

The four function keys on the right hand side of the 64 keyboard can be programmed so that they execute a predefined operation or function.

These function keys can be programmed by the user to execute a great number of different tasks. By doing so, operations which are to be repeated can be undertaken merely by pressing the desired function key.

Commodore 64 Screen Display

The Commodore 64 screen display consists of 25 rows of 40 columns each. Therefore, a total of 1000 screen locations are available.

Correcting Keyboard Entry Errors

Keyboard entry errors can easily be corrected as long as Return has not been pressed to end the line. The following keys and key combinations can be used to position the cursor and make any necessary corrections.

Cursor Up/Down Key
Cursor Right/Left Key
Clear/Home Key
Insert/Delete Key
Return Key
Space Key

Error Messages

When an incorrect entry is made, the Commodore 64 will respond with an error message. The error message will provide some explanation of the error encountered. A list of the error messages and a description of each is provided in Appendix A.

Error messages inherent to an applications program being run on the computer may also be displayed. Consult the program's documentation for an explanation of these error messages.

DATASSETTE CASSETTE RECORDER

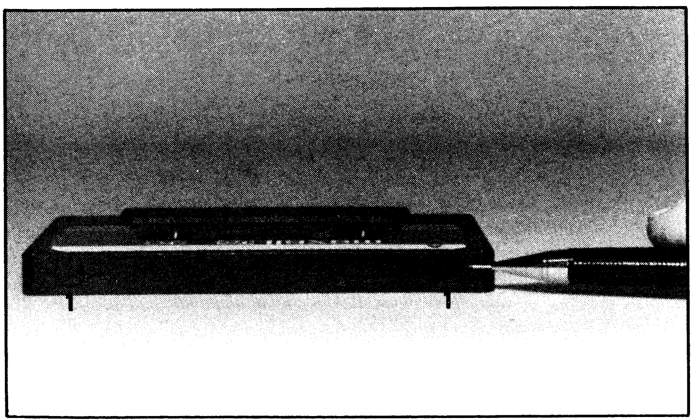
The Commodore Datassette Cassette Recorder allows the user to store programs on cassette tape for later use. The Datassette uses regular audio cassette tapes. As most programs use very little tape for storage, it is generally a good idea to use shorter rather than longer tapes.

Avoid using bargain brand cassettes, as these tend to jam in the machine. This, of course, could cause you to lose your programs.

When using cassette tapes to store programs, be sure to label the cassette with the names of the programs stored on it as well as any necessary data pertaining to the program.

Generally, cassettes have two write protect notches along their top edge, one for each side of the tape. These are shown in Illustration 2-4. Blank cassettes generally have tabs which cover the write protect notch. By removing these tabs and exposing the hole underneath, programs cannot be written onto the cassette tape. If you later wish to write to a cassette tape after the tabs have been removed, you can do so by covering the write protect notches with tape.

Illustration 2-4. Cassette Tape Write Protect Notches



1. Write Protect Notches

As mentioned in the preceding paragraph, one of the write protect notches is used to protect one side of the tape, while the other notch protects the second side. You can determine which write protect notch protects which side of the tape by holding the cassette so that the exposed tape is facing towards you, and the side that is to be protected is facing up. By removing the tab on the left side of the cassette, the side of the tape facing up will be protected. By removing the tab on the right side, the side of the cassette facing down will be protected.

The operation and usage of the Datassette cassette recorder will be explained in more detail in Chapter 5.

1541 DISK DRIVE

By including one or more 1541 Disk Drives in your computer system, you can save programs on diskette rather than on cassette. The 1541 uses standard 5¼ inch floppy diskettes. The storage capacity of these diskettes is 170K and they are compatible with other Commodore PET and CBM computer systems.

Diskette Handling Procedures

Rule 1. The first rule to remember when handling diskettes is to keep them away from any magnetic object -- or any metallic objects that could have become magnetized. Examples include magnets, telephones, and magnetized screwdrivers.

Rule 2. Always store diskettes in their dust covers.

Rule 3. Keep your diskette free from smoke or dust contamination. Dust accumulates a static charge which allows it to stick to the surface of the diskette. This is especially important to remember when storing diskettes in industrial areas, schools (chalk dust), warehouses, stock rooms, and dental offices (grinding of plaster). Keep your

computer work area relatively dust free. If possible, install an electrostatic air cleaner for the room.

Rule 4. Never touch, scratch, or attempt to clean a diskette's surface.

Rule 5. Always properly lable your diskettes. The label should include the data and contents at the least. A good procedure is to generate a directory for each diskette, then list that directly on the printer, cut out that listing, and paste or tape it to the diskette cover. This gives a complete guide to a diskette's contents.

Rule 6. Keep separate backup copies of your important diskettes in several different locations. In the event of a disaster (fire, theft, accidental erasure), you can copy your backup diskettes, and avoid losing data files.

Rule 7. Do not expose diskettes to excessive heat or direct sunlight.

Rule 8. Never fold, bend, or mutilate a diskette.

Rule 9. Never write on a diskette with a ball point pen. Always use a felt tip pen. A ball point pen can damage the diskette inside the cover.

Rule 10. Whenever the disk drive's power is turned on or off, be certain that the diskette is not completely inserted in the drive.

Rule 11. Try to keep the buildup of static electricity to a minimum in your computer room. Anti-static sprays are available for use on carpeting.

Write Protecting a Diskette

Most diskettes will have a small square notch cut out from the side of their protective envelope. If this notch is not covered, the computer will be able to write on the diskette. Covering the diskette write enable notch will prevent the computer from writing on the diskette.

Inserting a Diskette

Once the drive door has been opened, a diskette can be inserted into the drive. The diskette should be inserted with the label side facing up. The edge of the diskette opposite from the label should be placed into the diskette. This is the edge with the two small notches and the read/write opening.

Once the diskette has been inserted into the drive, close the drive door. Do not force the door shut. If the door does not close easily, remove the diskette and reinsert it. Never force the disk drive door shut as this may ruin the diskette.

DOS

In order for a disk drive to be used, a program known as the Disk Operating System or DOS must be present in memory. DOS controls all disk related activities. DOS will be discussed in more detail in Chapter 6.

DOS Commands

DOS includes a number of different commands which are useful in disk operations. These DOS commands will be discussed in detail in Chapter 6.

Formatting Blank Diskettes

Before a blank diskette can be used, it must first be **formatted.** Formatting can be defined as the preparation of a diskette so that it can accept data. Generally, this preparation involves initializing track, sector, and directory information for the disk controller.

The applications program that you are using may have a special routine for formatting blank diskettes. If this is the case, use that routine. If not, the DOS NEW command can be used to format blank diskettes. The usage of the DOS NEW command to format blank diskettes is covered in more detail in Chapter 6.

1515 & 1525 PRINTERS*

The Printer can be connected directly to the Commodore 64 console via the Serial Port. The Printer can print all of the characters and graphics symbols available on the computer keyboard.

The Printer is a dot matrix, tractor feed printer. The Printer uses sprocket feed paper up to 10 inches in width. The Printer outputs data in columns of 80 characters at 30 characters per second.

Operation, installation, and programming of the Printer will be discussed in detail in Chapter 7.

Loading and Executing Programs

The RUN command is used to execute a program stored in RAM. The use of RUN to execute programs will be discussed in Chapter 3.

---

* From here on, we will refer to the 1515 and 1525 collectively as the printer.

Programs can also be saved on diskette or cassette tape from RAM and later loaded from storage back into RAM for execution. The SAVE and LOAD commands are used to transfer BASIC programs between RAM and cassette or diskette. The usage of SAVE and LOAD is discussed in detail in Chapters 4, 5, and 6.

Loading Packaged Programs

Packaged programs may be available as plug-in cartridges, cassettes, or diskettes.

When loading a program contained on a plug-in cartridge, first of all turn off your computer and your TV set. Then, insert the cartridge. Turn your computer and TV set back on.

Be certain to turn off the Commodore 64 and TV set before inserting the cartridge. Otherwise, you might damage the computer.

The documentation provided with the program should name a key which when pressed will begin program execution. Press that key to start the program.

If your packaged program is on cassette, first of all, be certain that side 1 of the cassette has been completely rewound. Then, enter LOAD on the keyboard. The following message will be displayed:

PRESS PLAY ON TAPE

Press the play button the Datassette recorder. The screen will be blank as it searches the tape for the program. Once the program has been found, the following message will be displayed:

FOUND *program name*

Now, press the key specified in the documentation, and the

program will be run. By pressing the Run/Stop key, you can stop program execution.

If your packaged program is on diskette, insert the diskette containing that program so that its label is facing up and the side with the label is closest to you. When the disk is being properly inserted, the small notch on its edge will be on the left of the diskette.

When the diskette has been inserted into the drive, close the drive door by pressing down the lever on the front of the door.

Next, enter the following command,

LOAD *"program name"*,8

and press Return on the keyboard. The disk drive will begin to emit noises and the following message will be displayed on the screen.

```
SEARCHING FOR PROGRAM NAME
LOADING
READY
■
```

Once the Ready prompt appears, enter the RUN command on the keyboard, and the program will run.

A more extensive discussion of loading and saving programs on tape and diskette is included in Chapters 5 and 6.

CHAPTER 3.
BASIC PROGRAMMING FOR THE COMMODORE 64

INTRODUCTION

BASIC is probably the most widely used language with personal computers, with the Commodore 64 being no exception. The version of BASIC used with the Commodore 64 is the same version which is used with the VIC-20 and other PET/CBM models by Commodore.

Compiled vs. Interpreted Language

Computer languages are often distinguished as being either **compiled** or **interpreted** languages.

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 BASIC language which is standard on the Commodore 64 is an interpreted language.

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

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

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 and Execution

```
READY
NEW

READY
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

READY
■
```

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 only difference between an immediate mode statement and a program statement is the line number.

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 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 code.

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 computer'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 in 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 Commodore 64 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 the computer to return to the immediate mode. Obviously, the END statement should be the last line in your program.

Actually, Commodore 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.

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 is re-executed. As previously discussed, in the immediate mode each program line is 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 which 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 80 characters or 2 screen lines.

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 Commodore 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 Appendix D.

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.**

Commodore 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 A:PRINT B
```

Abbreviating Keywords

Many of the BASIC keywords can be abbreviated. For example, the keyword PRINT can be abbreviated with the symbol "?". Generally, however, keywords are abbreviated with a combination of a letter, the Shift key, and a second letter. For example, the keyword GOTO can be abbreviated as follows:

G Shift-O*

The various abbreviations for the keywords are contained in Appendix D.

---

* Press the G key, followed by pressing the O key while holding down one of the shift keys.

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 or if the computer is turned off, 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.

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 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.

Error Messages

When the Commodore 64 encounters a statement with an error, an **error message** will be displayed. The error message consists of the following:

? *error message* ERROR IN *line number*

The Commodore BASIC error messages are listed in Appendix A. The *line number* is only described when statements are entered in the program mode.

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 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 letters, numbers, symbols, and spaces. 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

Floating Decimal Point

Floating decimal point is the standard method of representing numeric data in the computer. 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 10 digits can be stored internally. However, only nine digits will actually be displayed. 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:

```
.566666667
```

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

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

-.0789
5
77.39
0
+.000001
67.98

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

Integers

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.

Commodore BASIC will convert integers into floating point notation before performing arithmetic operations. The most important difference between integers and floating point numbers is that an integer requires less memory storage area than a floating point value.

Scientific Notation

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

±*x* E ± yy

where;

± is an optional plus or minus sign.

x can either be an integer or floating point number. This position of the number 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 → 1 E6
 .000001 → 1 E-6
57500000 → 5.75 E+07
.00000479 → 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
1.21212121 E+11
```

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

Commodore BASIC cannot handle very large positive or negative numbers, or numbers very close to zero. Only numbers within the following limits are acceptable.

outer limits → ±1.70141183 E+38
inner limits → ±2.93873588 E–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.

Note the use of the LET statement in the preceding example. The LET statement is used to assign a value to a variable. Whenever a LET statement is used in a program, the value of the variable on the left side of the equation is to be replaced with the value appearing on the right.

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

```
100 LET A = 5
200 A = 5
```

BASIC Variable Names

Commodore BASIC allows a group of 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 Appendix D). Examples of reserved words are:

LET, GOTO, IF, READ, DATA

The following are examples of valid BASIC variable names:

A	JOHN
B23456	N4N
TOT	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 can 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$	MON$
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 A1% represents an integer.

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

Notice that only the integer value of A1% is output. The decimal portion is dropped or **truncated** as A1% can only accept integer values.

Keep in mind that although BASIC allows variable names with many characters, only the first 2 alphanumeric characters are recognized by the computer. In other words,

the variable names TENNIS and TENNESSEE cannot be distinguished by the Commodore 64.

However, the special symbols which identify the variable type (ie $,%) differentiate among variables with identical 2 character names. The following variable names would all be identified as unique by the computer:

TE%　　　TE　　　TE$

Tables and 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),..., 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 Commodore 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 DIM statement will dimension a numeric array of 16 elements.

```
100 DIM B(15)
```

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

```
100 DIM Z(5,2), B(100), C(2,3)
```

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.

---

* 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.

Illustration 3-2. Two-Dimensional Array

Columns

Rows	0	1	2
0			
1	(shaded)		
2			
3			

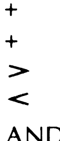
Expressions and Operators

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

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 examples were as follows:



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

Addition and subtraction have the same precedence level. Multiplication and division also have the same priority. Although relational expression are evaluated before Boolean expression, all of the relational operators have the same precedence level. The Boolean operators also share the same precedence level.

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

Table 3-1. Order of Evaluation

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

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 asterisk (*) 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 a minus sign is used to change a number's sign, that usage is known as a **unary** operation. Unary minus can be used to change the sign of a numeric constant or variable as shown below:

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

When unary minus is 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 Commodore BASIC, exponentiation is indicated with the up arrow symbol, ↑ .

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

8 * 3 + 7 ↑ 2

The preceding expression would evaluate to 73.

Relational Operators

The relational operators are defined as follows:

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

A relational operation evaluates to either true or false. For example, if the constant 1.0 was compared to the constant 2.0 to see whether they were equal, the expression would evaluate to false. In Commodore 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 expression.

5>7 → 0 (false)
3 = 3 → –1 (true)
2<>2 → 0 (false)
(2 = 2) * 4 → –4
(1>7) + 7 → 7

The first three examples are easy enough to understand. In the fourth example, the relational expression (2 = 2) is evaluated first as true of –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.

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

When comparing strings, 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 Commodore BASIC to combine 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 of 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 999
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 999. Otherwise, it will continue to the next statement.

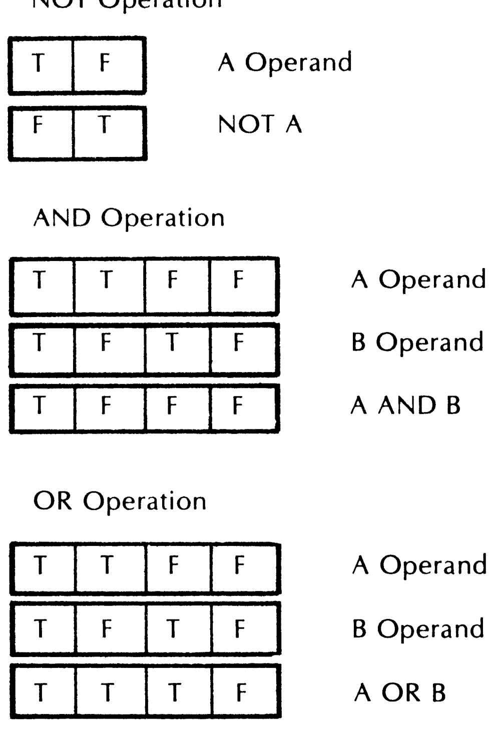
In the second example, the result of the logical operation

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 Operation

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

BASIC Statements

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

Remark Statements
Assignment Statements
Output Statements
Input Statements
Loops
Conditional Statements
Branching Statements
Subroutines
STOP, END Statements
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 MATRIX
```

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

In multiple line statement, the REM statement must be the final statement. The 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 NAME$ = "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$
```

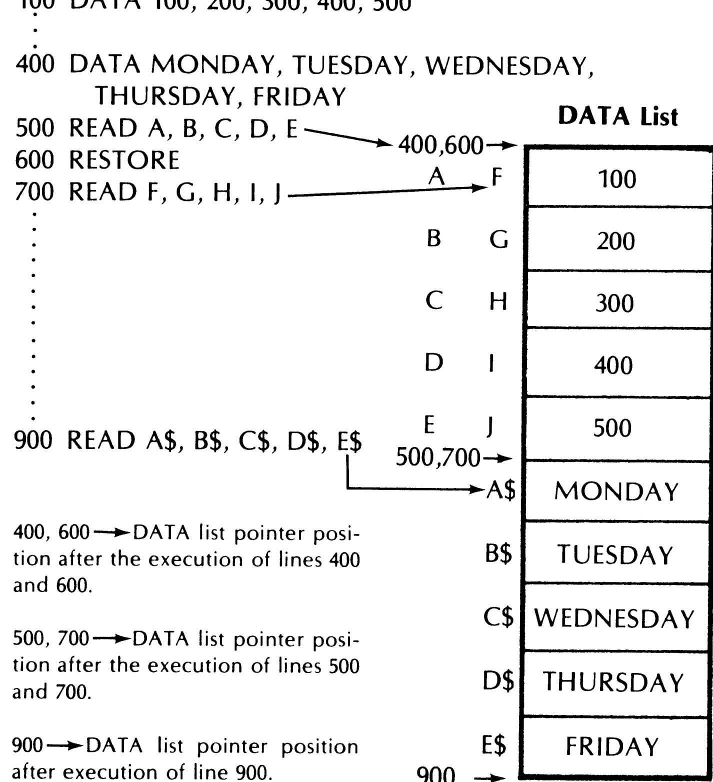
The DATA statement creates a list of constant values known as a DATA list. The items in the DATA list are assigned sequentially to the variables in the READ statement. A DATA list is depicted in Illustration 3-4.

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 at 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 variable, 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.

Two strings can be displayed on the same line with a single PRINT statement by separating the string constants or variables in the PRINT statement with commas. The following statements,

```
100 LET A$ = "JOHN"
200 PRINT A$, "BILL", "PETER"
```

would result in the display shown below:

```
JOHN      BILL      PETER
```

The display screen is divided into four separate zones when data is being output to the screen via the PRINT statement. The first zone begins at the far left side of the screen in the

first column (column 0). The zones begin in columns 0, 10, 20, and 30.

When a comma appears in a PRINT statement, the computer is instructed to begin printing the next parameter in the PRINT statement at the beginning of the next print zone. In our example above, "JOHN" is printed in print zone 1. The third parameter, "PETER" is displayed in the third zone of the same display line.

If a PRINT statement parameter consists of 10 or more characters, the next item will be printed in the next available zone. The following illustrates this principle.

```
100 LET A$ = "COMMODORE 64"
200 PRINT A$, "BILL", "PETER"
RUN
COMMODORE 64        BILL      PETER
```

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 values. Numeric values are separated from string values by one space. Numeric values are separated from each other by 2 spaces.

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 using the PRINT# statement in place of PRINT.

The usage of PRINT# to output data to the printer is discussed in Chapter 7.

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 Return 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
```

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 the computer as a delimiter. Any data appearing

after the comma will be treated as a separate INPUT statement data item.

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 that 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 ? REDO FROM START 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$
```

Loops

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

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

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 A (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 equal to 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 value 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.

```
050 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)
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. Also, be certain that every NEXT statement has a

matching FOR statment. If the BASIC interpreter cannot match every NEXT statement with a preceding FOR statement, an error will result.

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 X is equal to 0, then Y will be set equal to 1. If X is not equal to 0, Y will be set equal to 0.

```
050 Y = 0
100 IF X = 0 THEN Y = 1
```

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 a conditional statement. 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 the preceding example, if the value 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 consuming. By using **subroutines,** these additional entries will be unnecessary.

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

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

GOSUB *line number*

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

Illustration 3-5. 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 PRINT A$,X
               70 GOTO 200
             { 100 PRINT "NOT VALID AMOUNT"
Subroutine   { 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.

Commodore BASIC Functions

Functions are used 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 used 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 to 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 * 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.

Commodore 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 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 Commodore 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 computer 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 Commodore 64 uses codes somewhat different from the standard ASCII code set. The codes are listed in Appendix B.

Notice from Appendix B that not only do the keyboard characters and graphics symbols have corresponding ASCII codes, but that the various control and color keys have also been assigned ASCII codes.

These codes can be activated by including them with the CHR$ function with a PRINT statement. For example, the following,

```
PRINT CHR$(28)
```

would cause subsequent video characters to be output in red.

CHR$ and ASC Functions

As mentioned earlier, characters are represented on the computer as Commodore ASCII codes. The CHR$ function can be used to translate a character 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 cannot 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 ASCII 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 & POKE

The PEEK and POKE statements allow direct access to the computer'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 assigned to memory location 2000.

Advanced Input and Output Statements

To this point, we have discussed input and output only in the context of the INPUT and PRINT statements. Both of these automatically dictate the input and output device respectively. The INPUT statement expects an entry at the keyboard, and the PRINT statement outputs data to the screen.

Commodore BASIC allows input and output in forms other than that allowed for by the INPUT and PRINT statement. These include the following:

PRINT#
INPUT#
GET#

We will discuss these statements and how they relate to input and output in general in this chapter. In subsequent chapters, we will discuss how these statements can be used for specific input and/or output on the Datassette program recorder, VIC-1525 printer, and VIC-1541 disk drive.

Before we can completely understand these advanced input/output statements, we must first understand the concept of **device numbers** and **input/output channels.**

Device Numbers

Each input and output device has been assigned a number known as a device number. These device numbers are listed on page 119.

Input/Output Channels

Before communication can be undertaken between an input or output device, an input/output channel must first be associated with the device in question. The channel serves as a link between the BASIC program and the device in question. For example, when data is to be output by the program, it is output to the channel, which in turn was previously associated with a device.

OPEN

Before a channel can be used with a program, it must first be opened. When a channel is opened to an external device such as the Datassette recorder, disk drive, or printer, the computer reserves a memory area known as a buffer from which data will be sent to or received from the device specified.

OPEN is used with the following configuration:

OPEN *channel#, device#* [,*command*][,*string*]

The *channel#* specifies the channel number being opened. this *channel#* can range from 1 to 255, and will be used by subsequent INPUT# GET#, and PRINT# statements to access the device opened.

The *device#* specifies one of the devices listed on page 119. The *command#* indicates the operation for which the device is being opened. These are listed on page 120.

The *string* is generally a filename when the device specified is the cassette or diskette. The *string* can also contain information being sent to the printer or screen.

PRINT#, GET#, INPUT#

PRINT#, GET#, and INPUT# are all used to send data to or receive data from the device and channel specified in the OPEN statement.

Bits and Bytes

As mentioned earlier, the computer can only store numbers in its memory. An understanding of how these numbers are stored is helpful in using Commodore 64 sound and graphics.

The basic unit of information in the computer is called a **bit.** A bit can only have the value 0 or 1. This binary (2 state) system is particularly convenient for electronic equipment. Since it is easier to specify if a bit is "high" or "low" rather than its exact value, a binary system allows efficient transfer of data with virtually no errors. A bit can be thought of as a light switch that is either on or off.

Each memory location contains a set of 8 bits, or one **byte.** One byte can be represented by 8 switches. A set of 8 switches can have 256 different configurations, so one byte can be used to store 256 values.

Since each bit can have only 2 values, each bit represents a successive power of 2 (i.e. 1,2,4,8,16,32,64, and 128). Each switch is assigned one of these values. The value of the byte is the sum of the values of the switches that are on. The computer does not actually contain switches in the memory, but electronic components perform the same function.

If the value 1 represents a switch that is on, and 0 represents a switch which is off, the binary values of numbers are easily calculated, as demonstrated in Illustration 3-6.

Illustration 3-6. Binary Values

Bit Values								
128	64	32	16	8	4	2	1	
0	0	0	1	1	1	0	0	= 28
0	1	1	1	1	0	0	0	= 120
0	1	0	0	1	0	1	1	= 75

The value of one byte is the sum of the bit values which contain the value 1.

CHAPTER 4.
COMMODORE 64
BASIC REFERENCE GUIDE

This chapter provides descriptions and examples of the correct syntax for Commodore 64 BASIC. Each of the reserved words are listed in alphabetical order.

The following notation will be used to describe the configuration of each of the commands, statements, or functions.

1. Capitalized words are keywords.

2. Italicized items are parameters.

3. Items enclosed in brackets[ ]are optional.

4. Ellipsis (...) represents repetition.

5. Punctuation (except brackets) must be included as shown.

6. Braces { }are used to enclose a choice of items.

7. The following symbols will be used:

expression	Algebraic or logical expression (i.e. X > 5, 3 + X, X = 7)
X, Y, Z	Numeric variable name
X$, Y$, Z$	String variable name
a, b, c	Any number or numeric expression
string	String value (constant or variable)
data...	String or numeric value (constant or variable)

ABS

The ABS function returns the absolute value of its argument.

Configuration

X = ABS(*a*)

Example

```
PRINT  ABS(-81)
81
```

AND

AND is used between two expressions, and returns a non-zero value if they are both true, and 0 if one is false.

Configuration

expression AND *expression*

The conditions of true and false are represented in the computer by the logical values -1 and 0. As a result, the logical operators (AND, OR, and NOT) generally operate with the values -1 and 0. The AND operation can be explained by the following truth table.

EX1	EX2	RESULT
-1	-1	-1
-1	0	0
0	-1	0
0	0	0

AND is generally used in an IF/THEN statement with relational expressions. For example:

```
10 X = 10
20 Y = 30
30 IF X = 10 AND Y>100 THEN GOTO 999
40 PRINT "CONDITIONS WERE NOT MET"
999 END
RUN
CONDITIONS WERE NOT MET
```

In this example, AND is used in an IF/THEN statement which ends the program if both conditions are true. The first expression of the AND statement is X = 10. This is true because X is assigned the value 10 in line 10. The second expression, Y >100, is false because Y is assigned the value 30 in line 20. As a result, *EX1* is true and *EX2* is false. This corresponds to the truth table where *EX1=1* and *EX2=0*. The result from the table is 0 (false), so the condition of the IF/THEN statement is false, and the next line is executed.

When the AND operation is executed with numeric expressions, the arguments are truncated and converted to their binary equivalents. The value of the AND statement is the result of the AND operation on each bit of the values. For example:

```
     6    (binary 0110)
AND 10    (binary 1010)
     2    (binary 0010)
```

Since the binary equivalent of 6 is 110, and the binary equivalent of 10 is 1010, the AND operation for each bit has the result 0010. The numeric value of 0010 is 2. Any non-zero value is considered true.

ASC

The ASC function returns the ASCII code for the first character of a string. The argument of ASC can be a string variable or constant.

Configuration

X = ASC (*string*)

Example

```
100 B$ = "BANANA"
200 PRINT ASC(B$)
RUN
66
```

ATN

The ATN function returns the arctangent of the argument. The result will be in radians.

Configuration

X = ATN(*a*)

Example

```
PRINT ATN(.576)
 .522585482
```

CHR$

The CHR$ function returns the character with the ASCII code specified by the argument. The value of the argument must be between 0 and 255. If the argument is not an integer, it is truncated.

Configuration

X$ = CHR$(*a*)

Example

```
PRINT CHR$(65)
A
```

CLOSE

The CLOSE statement closes a channel that has been opened for input, output, or both.

The argument of a CLOSE statement must be the same as in the corresponding OPEN statement.

Configuration

CLOSE a

Example

CLOSE 3

CLR

The CLR command clears the values of the variables in the memory. The CLR statement also clears the arrays and matrices. User defined functions and file buffers are eliminated. The return locations for subroutines and FOR/NEXT loops are also cleared. However, the program in RAM is not affected.

Configuration

CLR

Example

```
100  A = 2:B = 3
200  PRINT A, B
300  CLR
400  PRINT A, B
RUN
2           3
0           0
```

CMD

The CMD statement sends output to an output device (usually the printer) instead of to the display.

Configuration

CMD *a* [, *string*]

The argument *a* of a CMD statement designates a file previously opened for an I/O device. When a CMD statement is executed, the output device that is specified becomes the general output device. Any PRINT or LIST statements that follow a CMD statement cause the output to be sent to the particular output device. The argument of a CMD statement must match the file number of a previous OPEN statement for an appropriate output device.

The optional string argument is generally used to print the title of a listing on the printer.

To return to the standard output mode, execute a PRINT # statement for the specified file number and then close that file. Once a PRINT# and CLOSE have been executed to the file specified by CMD, output will again be directed to the display rather than to the printer.

Example

```
10 OPEN 1, 4
20 CMD 1
30 LIST
40 PRINT#1
50 CLOSE 1
```

In the preceding example, the OPEN statement at line 10 opens an I/O channel for output to the printer. At line 20, the CMD statement designates file number 1 for general output. At line 30, the LIST statement causes the program in memory to be output.

Because of the CMD statement, the program is not displayed on the screen, but sent to the printer instead. The PRINT # statement at line 40 makes the display the general output device.

CONT

The CONT command allows a program which had been stopped to be restarted. The program could have been stopped either via the STOP key, the STOP statement, or the END statement. When CONT is executed, the program will resume execution at the exact point where it was stopped.

CONT cannot be used in instances where an error was encountered or where program lines were added or edited. If an attempt is made to use CONT in such a circumstance, the following error message will be displayed.

CAN'T CONTINUE ERROR

CONT can be a very useful tool in debugging programs. By inserting STOP statements at given locations within a program, execution will stop, and variable values can be checked for validity. CONT allows execution of a program which had been stopped to resume.

Configuration

CONT

COS

The COS function returns the cosine of its argument in radians.

Configuration

X = COS(*a*)

Example

```
20  X = COS(-6)
30  PRINT X
RUN
.960170287
```

DATA

The DATA statement supplies a list of information that is used in a program through READ statements. A DATA statement can include numeric values, string values, or both.

Data items are separated by commas. Therefore, string values that contain commas will be read as separate data items. For example, DATA DOE, JOHN is a DATA statement with two data items. However, DATA DOE.JOHN has only one item.

Commas, colons, graphics characters and cursor controls can be included in a data item if the entire string is enclosed in quotation marks.

Configuration

DATA *data* [, *data*]...

Data must be read into the correct type of variable. A string variable can accept data in any form.

Example

```
20 FOR I = 1 TO 5
30 READ A$:? A$
40 NEXT I
50 DATA TOM C., 25,,3 + 4 * %,247
RUN
TOM C.
25

3 + 4 * %
247
```

The preceding example shows correct data for a string variable. Notice the blank line in the output that corresponds to the two commas in a row. This is read as a string value with no characters and length equal to zero.

If only 4 data items had been supplied with this program, the OUT OF DATA error message would have been displayed to notify the user that not enough data was supplied. The DATA statement must supply at least as many DATA items as there are corresponding READ statement variables.

DEF FN

The DEF FN statement is used to define a function that is used repeatedly in a program.

Configuration

DEF FNY(*X*) = function definition

The name of the function (represented by Y) can be any variable name. The argument (represented by *X*) is a numeric variable name. The function definition is a set of operations that use the argument (X).

Example

```
10 M = 3.1415927
20 DEF FNS(X) = COS(X) + SIN(X)
30 PRINT FNS(M)
RUN
-1.00000003
```

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

DIM

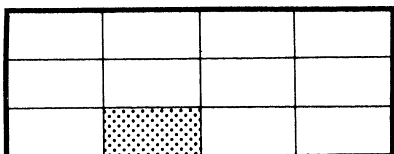
The DIM statement is used to set aside memory space for 1, 2, or 3 dimensional string or numeric variables. If an array variable is used in a program without having been dimensioned, that variable will automatically be dimensioned to 11 elements.

Configuration

$$\text{DIM} \left\{ \begin{matrix} \text{X}(a(,b)(,c) \\ \text{X\$}(a(,b)(,c) \end{matrix} \begin{bmatrix} ,\text{Y}(d(,e)(,f) \\ ,\text{Y\$}(d(,e)(,f) \end{bmatrix} \ldots \right\}$$

The lowest element number in an array is always 0. The highest element number is specified in the DIM statement. For example, DIM X(100) would dimension a numeric array of 101 elements. The first element would be identified as X(0) and the last as X(100).

Numeric variables can also be used with two or three subscripts. This results in a two or three dimensional array, or matrix. For example, if X is dimensioned in the statement DIM X(2,3) the following table would result.



The values in this table can be named by X with two subscripts (in parentheses). The first subscript is the row number, and the second is the column number. For example, the value of the shaded block would be X(2,1) because it is row 2, column 1.

Example

```
 20 FOR I = 0 TO 3
 30 FOR J = 0 TO 2
 40 READ X(I, J)
 50 PRINT X(I, J),
 60 NEXT J
 70 PRINT
 80 NEXT I
100 DATA 1, 2, 3
110 DATA 4, 5, 6
120 DATA 7, 8, 9
130 DATA 1, 2, 3
```

This example shows a technique for assigning a table of values, and printing the table. Executing this program would have the following result.

```
RUN
1          2          3
4          5          6
7          8          9
1          2          3
```

END

An END statement ends the execution of the program. An END is not necessary at the end of a program because execution stops automatically after the last line of code. However, it is good programming technique to end BASIC programs with an END statement.

When a program has been halted by an END statement, a CONT statement will cause the program to resume at the statement following the END statement.

Configuration

```
END
```

Example

```
10 INPUT X
20 IF X<= 10 THEN END
30 PRINT "X IS LARGER THAN 10"
40 GOTO 10
```

The previous example will end only if a value of X is entered which is less than or equal to 10.

EXP

The EXP function returns the exponential of the argument. The exponential is the value of e (2.71828183...) raised to the power of the argument.

Configuration

X = EXP(*a*)

Example

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

FN

The FN statement is used to evaluate an argument according to the operation specified in a DEF FN statement.

Configuration

X = FNY(*a*)

The FN statement must include a function name (represented by Y), and an argument (represented by (a)). Any variable name can be used to name the function, but the name must correspond to a DEF FN statement.

The value of the argument is plugged into the equation of the DEF FN statement and evaluated. The function returns a numeric value.

Example

```
10 M = 3.1415927
20 DEF FNS(X) = COS(X) + SIN(X)
30 PRINT FNS(M)
RUN
-1.00000003
```

The previous example contains a program that has a FN statement at line 30. The function is named S and defined for X (COS(X) + SIN(X)) at line 20. At line 30, the FN statement evaluates the function S for the value of M. The function returns a numeric value that is displayed by the PRINT statement.

FOR

A FOR statement is used with a NEXT statement to form a repetitive loop within a program.

Configuration

FOR A = *a* TO *b* (STEP *x*)

Every FOR statement must have a corresponding NEXT statement.

Example

```
10 FOR J = 1 TO 5
20 PRINT J;
30 NEXT J
RUN
 1 2 3 4 5
```

In the previous example, the FOR/NEXT loop is repeated five times. Line 20 is the only statement inside the loop, however, any number of program lines can be placed within a loop.

In line 10, J is assigned the value 1. J is referred to as a counter. The value of J is printed. Then, the NEXT J statement is executed. Here, the program returns to the FOR statement, where J is incremented by one. This loop is repeated until J is set equal to 5. When the counter (J) has been set equal to the value (5), the loop has been executed, and the program will proceed with the statement following NEXT J.

A FOR/NEXT loop can use a STEP statement to increment the counter by a value other than 1.

Example

```
10 FOR J = 1 TO 2 STEP .5
20 PRINT J;
30 NEXT J
RUN
  1 1.5 2
```

The preceding example contains a FOR/NEXT loop which increments the value of J by .5 each time the loop is executed.

A FOR/NEXT loop can also be used to decrease the value of the counter. This can be accomplished by using the optional STEP statement within the FOR statement. If the STEP statement has a negative argument, the counter is decreased each time the loop is executed. The following example illustrates a FOR/NEXT loop where the counter is decremented rather than incremented.

Example

```
10 FOR K = 10 TO 5 STEP -2
20 PRINT K;
30 NEXT K
RUN
  10  8  6
```

This loop begins at line 10 by assigning the counter (K) the value 10. At line 20 the value of K is printed. When line 30 is encountered, execution continues at line 10, because the NEXT statement returns the program to the preceding FOR statement. The value of the counter is changed by the argument of STEP. Since the STEP value is -2, the counter is decreased by 2. The value of the counter is changed to 8. At line 20, the new value of K is printed. Line 30 is executed again, so the program returns to the FOR statement at line 10. The counter is again decremented by 2. The new value of K is 6. At line 20, this K value is printed.

When line 30 is executed again, the program does not return to line 10. The current value of the counter is 6, and if the counter was to be decremented again, the counter would be 4. However, 4 is less than the final value which is specified in the FOR statement (the argument of TO). As a result, the loop does not continue after K = 6 because another decrement would make the counter less than the final value (5).

If the counter of a loop is being incremented, the loop will be executed until the counter would exceed the final value if it were incremented again. For example: FOR J = 1 TO 4 STEP 2 would be executed with J equal to 1 and 3. The counter (J) would exceed the final value (4) if it were incremented again.

A FOR/NEXT loop should be executed as if it were a single statement. An attempt to branch into a FOR/NEXT loop will cause an error.

Example

```
10 GOTO 30
20 FOR I = 1 TO 10
30 PRINT I
40 NEXT I
RUN
0
? NEXT WITHOUT FOR ERROR IN 40
```

In general, branching out of a FOR/NEXT loop will not cause an error. However, exiting a loop before it has completed should be avoided.

FRE

The FRE function returns the number of bytes of memory available. The FRE function requires an argument, but that argument has no effect on the value returned.

Configuration

X = FRE (*a*)

If more than 32,000 bytes are available, the value returned will be negative. The following example contains an expression that always returns the correct amount of available memory.

Example

```
X = FRE(1) - (FRE(1)<0) * 65536
```

GET

A GET statement is used to assign a value to a variable. The input is accepted one character at a time from any input device.

Configuration

$$\text{GET [\# a,]}\begin{Bmatrix}X\\X\$\end{Bmatrix}$$

The first argument of a GET statement is the optional file number. The file number must be an integer between 1 and 255 inclusive, and corresponds to an appropriate OPEN statement.

If a file number is not specified, the input is taken from the keyboard.

The GET statement must include a variable name. The value of the input is assigned to the variable in the GET statement. A string variable name can be assigned any character, or none at all. A numeric variable, however, can accept only the characters 0-9 inclusive. If a GET statement with a numeric variable receives a non-numeric character, an error results.

If a GET statement is executed with a string variable, and no data is available, the variable is assigned the null string (a string with no character). If a GET statement is executed with a numeric variable, and no data is available, the variable is assigned the value 0. Unlike an INPUT statement, the program does not wait for a response to a GET statement.

When a GET statement includes the optional I/O channel number, the input is taken one character at a time from the device specified in the corresponding OPEN statement. The OPEN statement must include an input device as well as an input operation.

Example

```
10 GET Z : IF Z = 0 THEN 10
20 PRINT Z
30 END
```

The previous example contains a program that uses a GET statement to receive input from the keyboard. When the program is executed, the variable Z is set equal to 0 until a numeric character is entered at the keyboard. The IF/THEN statement at line 10 causes the GET statement to be repeated until a numeric value (other than zero) is entered.

When the GET statement receives acceptable input, the value of Z is displayed and the program ends.

GOSUB

The GOSUB branches program control to the subroutine beginning at the line number specified as its argument.

Configuration

GOSUB *line number*

Subroutines can be called from any part of a program. A RETURN statement, at the end of a subroutine, causes the program to resume execution with the statement directly after the GOSUB statement.

Subroutines are convenient to use when the same set of operations need to be repeated at different parts of a program.

Example

```
 10 FOR J = 0 TO 2
 20 GOSUB 100
 30 NEXT J
 40 J = 5
 50 GOSUB 100
 60 END
100 PRINT J;
110 RETURN
RUN
 0  1  2  5
```

The previous example illustrates a subroutine that is called 4 times, from 2 different parts of the program. In this example, only one statement is included in the subroutine. However, many statements can be included in a subroutine.

Line 10 begins a FOR/NEXT loop. The counter (J) is set equal to 0 the first time through the loop. Line 20 calls the subroutine at line 100. As a result, line 100 is executed next. The subroutine prints the value of J and proceeds to line 110. At line 110, the program is returned to the point where the subroutine was called (line 20).

The statement at line 30 is then executed. The NEXT statement causes the loop to be incremented and repeated. The counter (J) is set equal to 1, and the subroutine is called again from line 20. At line 100, the value of J is printed. Line 110 returns the program to line 20.

These steps are also repeated for J =2. When the loop has been executed 3 times, the program will proceed to the line 40. J is assigned the value 5, and the subroutine is called again at line 50. The subroutine prints the value of J. The program then returns to line 60 where it ends.

GOSUB can also be used with ON to branch a program to one of several subroutines.

Configuration

ON *expression* GOSUB *linenumber[,linenumber]...*

The expression after the ON statement indicates which line number the program proceeds to. This is called the control expression. The control is evaluated and truncated. If the value is negative or greater than 255, an error occurs. If the value of the control is 1, the program continues at the first line number after GOSUB. If the control is equal to 2, the program continues at the second line number after GOSUB, etc.

If the value of the control is 0 or greater than the number of line numbers in the statement, the line after the ON/GOSUB statement is executed.

Example

ON X GOSUB 100, 200, 300, 400

This statement executes the subroutine at line 100 if X = 1, If X = 2, the subroutine at line 200 is executed. If X = 3, the subroutine at line 300 is executed. If X = 4, the subroutine at line 400 is executed. If X = 0 or X is greater than 4, the next line is executed.

GOTO

The GOTO statement causes the program to proceed at the indicated line number.

Configuration

GOTO *linenumber*

Example

```
10 X = X + 1
20 IF X ↑ 2 > 50 THEN END
30 PRINT X;
40 GOTO 10
RUN
1 2 3 4 5 6 7
```

The previous example demonstrates the use of GOTO. Line 10 increases the value of X by 1. Line 20 ends the program when X squared is greater than 50. When line 40 is executed, the program returns to line 10. This program repeats lines 10 through 40 until the program is ended or branched out of the loop. The program ends when X = 8 because 8 squared is greater than 50.

GOTO is also used with an ON statement to branch a program to one of several lines.

Configuration

ON *expression* GOTO *linenumber* [,*linenumber*]...

The expression after the ON statement indicates which line number the program proceeds to. This is called the control expression. The control is evaluated and truncated. If the value is negative or greater than 255, an error occurs. If the value of the control is 1, the program continues at the first line number after GOTO. If the value is 2, the program continues at the second line number after GOTO, etc.

Example

```
10 FOR J = 1 TO 3
20 ON J GOTO 40, 50, 60
40 PRINT "J = 1":GOTO 70
50 PRINT "J = 2":GOTO 70
60 PRINT "J = 3"
70 NEXT J
```

The previous example is a program that uses an ON/GOTO branch. Line 10 begins a FOR/NEXT loop that is repeated 3 times. The first time through the loop, the counter (J) is set equal to 1. At line 20, the program is branched to the first line number after GOTO because the control (J) is equal to 1. At line 40, the message J = 1 is printed. The program is sent to line 70, where the NEXT J is chosen. Since the loop increments the counter by 1, the counter is set equal to 2 during the second execution of the loop. At line 20, the program is branched to line 50, the second line number after GOTO. At line 50, the message J = 2 is printed. The loop is repeated again with J set equal to 3. At line 20, the program branches to line 60, the third choice. At line 60, the message J = 3 is printed. The loop is complete so the program ends.

If the control (argument of ON) equals zero or a number greater than the number of choices, the statement that follows the ON/GOTO statement is executed.

IF

The IF statement is used with a THEN statement to branch a program if a particular condition is true.

Configuration

IF *expression* THEN { *statement [:statement]...* | *linenumber* }

The expression that follows IF can be logical or algebraic. Any algebraic expression that does not equal zero is considered true. The logical operators (AND, NOT and OR) can be used in the IF expression.

Example

```
10 X = 15
20 Y = 30
30 IF X>10 AND Y>20 THEN 50
40 PRINT "CONDITIONS NOT MET":END
50 PRINT "CONDITIONS HAVE BEEN MET"
RUN
CONDITIONS HAVE BEEN MET
```

The previous example shows two logical expressions and a logical operator in the IF/THEN statement (line 30). The AND will only be true when both conditions have been met. Since X = 15 (line 10) and Y = 30 (line 20), both of the conditions of line 30 are true. As a result, the program branches to line 50. At line 50, the message CONDITIONS HAVE BEEN MET is printed.

An END statement is used in line 40 to prevent both messages from being printed when the IF statement is false.

An IF/THEN statement can also be followed by statements instead of a line number.

Example

```
10 Y = 5
20 X = 10
30 IF X = 10 THEN PRINT X:PRINT Y
RUN
10
5
```

The previous example shows that statements can follow a THEN statement, separated by colons. If the condition is true, the statements are executed. If the condition is false, the program will continue at the next line, and the statements after the THEN statement are ignored. Since X = 10 (line 20), the conditions at line 30 (X=10) is true. As a result, the statements after THEN are executed and the values of X and Y are printed.

INPUT

An INPUT statement is used to assign values to variables.

Configuration

$$\text{INPUT} \begin{bmatrix} \#a, \\ \textit{“prompt”;} \end{bmatrix} \begin{Bmatrix} \text{X} \begin{bmatrix} ,\text{Y} \\ ,\text{Y\$} \end{bmatrix} \cdots \\ \text{X\$} \end{Bmatrix}$$

An INPUT statement can be used to accept values for the variables from any input device. If the optional file number (#*a*) is not specified, the data is taken from the keyboard.

When an INPUT statement is encountered (for the keyboard), a question mark appears on the display as a signal for the operator to enter data. A prompt message can also be included to print a message when an INPUT statement is executed.

An INPUT statement can include a combination of numeric and string variables. Values can be assigned to each variable with one response. The values of a multiple response must be separated by commas. If the response to an INPUT statement does not include enough values, two question marks appear on the display as a signal that more data is necessary. If too may values are included in a response, the message:

?EXTRA IGNORED

is displayed to notify the operator that at least one of the data items was not assigned to a variable.

If a string value is entered where a numeric value is expected, the message:

?REDO FROM START

is displayed to notify the operator that an incorrect type of value has been entered.

Numeric data can be entered in standard or scientific notation.

String values that include a comma, semicolon, colon or the EOL character (CHR$(13)) must be enclosed in quotation marks. String values that do not have any of these characters do not need quotation marks.

Example

```
10 INPUT "ITEM, QTY."; ITEM$, QTY
20 PRINT ITEM$,QTY
RUN
ITEM, QTY.? PENS, 1200 ←—— user's response
PENS        1200
```

The previous example contains a program that accepts a user's response through the keyboard. The INPUT statement at line 10 includes a prompt message that is displayed when the INPUT statement is executed. The INPUT statement requires a string value for the variable ITEM$, and a numeric value for the variable QTY. The example also includes a correct response as entered by the operator.

If the optional file number is used in an INPUT statement, the variables are assigned the data taken from the specified input device. An appropriate OPEN statement must precede the INPUT statement if a file number is used.

INT

The INT function returns the largest integer that is less than or equal to the argument.

Configuration

X = INT (*a*)

Examples

```
PRINT INT (13.9)
13
PRINT INT (-4.7)
-5
```

LEFT$

The LEFT$ statement is used to designate the leftmost characters of a string.

Configuration

Y$ = LEFT$ (*string*, a)

The LEFT$ 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 first characters in the string argument.

Example

```
10 A$ = "WILLIAM JONES"
20 PRINT A$
30 PRINT LEFT$(A$,7)
RUN
WILLIAM JONES
WILLIAM
```

The preceding example contains a program that uses a LEFT$ 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 first 7 characters 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 0, a string with no characters is returned.

LEN

The LEN function is used to return the number of characters and spaces in a string value.

Configuration

X = LEN (*string*)

Example

```
10  X$ = "WILLIAM JONES"
20  X = LEN (X$)
30  PRINT X
RUN
 13
```

LET

The LET statement is optional. It is used to assign a value to a variable.

Configuration

[LET] { *X* = *a* / *X$* = *string* }

Examples

LET X = 250
X = Y + 25

LIST

The LIST statement is used to display the program that is currently in the computer's memory.

Configuration

LIST [*linenumber*] — [*linenumber*]

The arguments of a LIST statement are optional. If the LIST statement is executed without any arguments, the entire program that is currently in memory is displayed on the screen. Holding down the Control key on the keyboard makes the listing proceed slowly.

The arguments of a LIST statement are used to specify a particular section of the program. If only one number appears in a LIST statement, the specified line of the program is displayed. If there is no corresponding line number in the program, the statement has no results.

If a LIST statement has one argument, followed by a dash, every line of the program with a line number greater than or equal to the argument is displayed.

If a LIST statement has one argument preceded by a dash, every line of the program with a line number less than or equal to the argument is displayed.

Examples

```
LIST 100
LIST 10-100
LIST 10-
LIST -100
```

The first statement in the preceding example displays line 100 if the program in memory contains a statement with line number 100. The second example displays all the lines in the program numbered from 10 to 100 inclusive. The third example displays line number 10 along with all the subsequent lines of the program. The fourth example displays all the lines of the program in memory up to and including line number 100.

LOAD

The LOAD statement is used to enter a program into the computer's memory.

Configuration

LOAD [*"program name"*][,*a*]

Programs are generally stored on a cassette tape or disk. The second argument of the LOAD statement specifies the device that is used to load the program. If the LOAD statement does not include a device number, the cassette recorder is assumed (device number 1). If a program is to be loaded from a disk, the second argument of the LOAD statement must specify device number 8.

The first argument of the LOAD statement is a program name. The program name is optional with the cassette recorder. If no program name is specified, the next program on the cassette tape is loaded into the computer's memory.

A program name is always necessary with the disk drive. However, if an asterisk (*) is used as a program name, the first program on the disk is loaded into the computer's memory.

The program name and device number can be represented by variables in a LOAD statement.

Examples

```
LOAD
LOAD "EXPENSES"
LOAD "*", 8
LOAD "EXPENSES",8
```

In the previous example, the first statement loads the next program that is recorded on the cassette tape. The second example statement also uses the cassette tape, but the program with the name "EXPENSES" is loaded into memory.

The third statement of the preceding example loads the first program that is stored on the disk. The fourth statement also uses the disk, but the program with the name "EXPENSES" is loaded into memory.

When a LOAD statement is executed for the cassette recorder, the prompt:

PRESS PLAY ON TAPE

is displayed. When the lever is pressed, the recorder begins to operate and the display is cleared. If the program name is not specified, the program name does not appear in the following messages.

When the program recorder finds a program on the tape, the message,

SEARCHING FOR *program name*

is displayed, followed by:

FOUND *program name.*

If the program names are the same in the two preceding messages, the computer loads the program when the Commodore key is pressed. If the program recorder did not find the program it was searching for, the Commodore key causes the recorder to proceed to the next program on the tape. Each program that is encountered causes an additional FOUND message to appear on the display.

When the program is being loaded into the computer's memory, the message,

LOADING

is displayed, followed by:

OK
READY

when the LOAD procedure is complete.

LOG

The LOG function returns the natural logarithm of the argument. The natural log function is undefined for zero and negative arguments. As a result, an illegal quantity error occurs when the argument is not positive.

Configuration

X = LOG (*a*)

Example

```
PRINT LOG (2.71828183)
1
```

MID$

The MID$ statement is used to designate characters in the middle of a string.

Configuration

Y$ = MID$ (*string*, a,b)

The MID$ function returns a string value. The first argument is a string constant or a string variable. The second and third arguments are numeric values. The first numeric argument determines the first character from the string argument that is returned. The second numeric argument determines the total number of characters that are returned.

Example

```
10 A$ = "JOHN PETER JONES"
20 PRINT MID$ (A$,6,5)
30 END
RUN
PETER
```

The previous example contains a program that uses the MID$ function. At line 10, the variable A$ is assigned a string value. At line 20, the PRINT statement includes a MID$ statement. The MID$ statement specifies the string value of A$. The second argument (6) specifies the sixth character in the string. The third argument (5) specifies the number of characters returned. As a result, the string value "PETER" is printed because "P" is the sixth character of the string (including spaces).

NEW

The NEW command eliminates the current program in the computer's memory.

Configuration

NEW

Example

NEW

NEXT

The NEXT statement is used with a FOR statement to form a repetitive section of a program.

Configuration

NEXT [*X*] [,*Y*]...

A FOR statement begins a loop, and a NEXT statement ends it. The FOR statement sets an initial value and a final value for the counter. The optional STEP statement specifies the amount that the counter is increased or decreased each time the loop is executed.

If a NEXT statement does not include a variable name, the most recent FOR/NEXT loop is incremented.

If a NEXT statement specifies a variable from a preceding FOR/NEXT loop, the loops that were skipped are automatically terminated.

If a NEXT statement contains more than one variable, the loop for the first variable must have been completed before the second loop is incremented.

Example

```
10 FOR J = 1 TO 10 STEP 2
20 PRINT J
30 NEXT
```

In the previous example, the variable J is the counter. The initial value of the counter is 1, and the final value is 10. The value of the counter is incremented by 2 each time the loop is executed.

The section of the program between the FOR and NEXT statements is repeated for each different value of the counter. Each time the NEXT statement is executed, the value of the counter is changed by the STEP argument value. The loop is repeated for each value of the counter. In the previous example, the loop is repeated 5 times, with the counter equal to 1, 3, 5, 7, and 9. The initial value of the counter (J) is 1, and it is increased by 2 each time the loop is executed because of the STEP 2 statement.

If no STEP statement is used, the counter value increases by 1 each time a NEXT statement is executed.

A FOR/NEXT loop can also have a decreasing counter. If the STEP argument is negative, the value of the counter decreases each time the loop is executed.

An increasing counter will repeat the loop until one more increase would make the counter greater than the final value. A decreasing counter will repeat the loop until one more decrease would make the counter less than the final value.

When a loop has been completed, the statement after the NEXT statement is executed.

NOT

The NOT statement is used as logical negation.

Configuration

NOT *expression*

The NOT operation is executed according to the following truth table.

A	NOT A
0	-1
-1	0

The conditions of true and false are represented by the values -1 and 0.

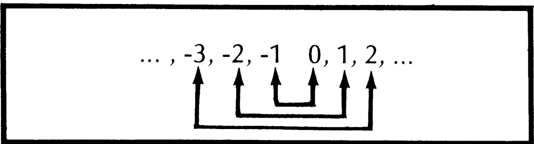
NOT statements are generally used in IF/THEN statements.

Example

```
10 Y = 5
20 IF NOT Y = 4 THEN 999
30 PRINT "Y IS EQUAL TO 4"
999 END
```

The program in the preceding example contains a NOT statement as part of an IF/THEN statement. At line 10, the variable Y is assigned the value 5. At line 20, the statement NOT Y = 4 is the condition of the IF/THEN statement. Since the value of Y is 5, the statement 5 = 4 is false. As a result, NOT Y = 4 is true, and program control branches to line 999. When Y has any value other than 4, the program has no output.

A NOT statement can also have a numeric argument. The NOT operation returns a numeric value according to the formula NOT A = (–A) – 1. A numeric argument for a NOT statement is always truncated, so a NOT statement is always evaluated to an integer. The NOT operation of integers can be represented by the following illustration.



OPEN

An OPEN statement is used to open an I/O channel for an input or output device.

Configuration

OPEN *a, b,* [*c*][,"*string*"]

The first argument of an OPEN statement specifies a file number. The file number can be any value from 1 to 255. The file number that is used in an OPEN statement is also used in any subsequent INPUT#, GET# or PRINT# statements for that device.

The second argument of an OPEN statement indicates the type of I/O device that is used. Each device has a device code, as listed in the following table.

device	**device code**
keyboard	0
cassette	1
RS-232	2
Display	3
Printer	4
Printer	5
Disk	8

The third argument of an OPEN statement indicates the I/O operation that is to be performed. The third argument is not necessary when the keyboard or display is used as an I/O device. The following table lists appropriate I/O operation codes for each device.

device	operation code	operation
Cassette	0	read
	1	write
	2	write*
Disk	1-14	data channel
	15	command channel
Printer	0	upper case/ graphics
	7	upper/lower case

*Put END OF TAPE marker at end.

The fourth argument has different functions for each device. With the cassette, the fourth argument designates the name of the file. In disk operation modes 1-14, the fourth argument designates the name of the file. In disk operation mode 15, the fourth argument designates the command. With the printer, the fourth argument is displayed as though it were in a PRINT statement.

Examples

```
OPEN 2, 0
OPEN 1, 1, 2, "RECORDS"
```

In the previous example, the first statement opens the keyboard for input. The second statement opens the cassette for storing the file "RECORDS". The End of Tape marker is recorded at the end of the file.

OR

The OR operator is generally used in IF/THEN statements to combine two or more conditions.

Configuration

expression OR *expression*

The OR operation is executed according to the following truth table.

A	B	A OR B
0	-1	-1
-1	-1	-1
-1	0	-1
0	0	0

The conditions of true and false are represented by the values -1 and 0.

Example

```
10 X = 5
20 Y = 10
30 IF Y > 5 OR X < 0 THEN 50
40 END
50 PRINT "Y IS GREATER THAN 5"
60 PRINT "OR X IS NEGATIVE"
70 END
RUN
Y IS GREATER THAN 5
OR X IS NEGATIVE
```

In the preceding example, the variables X and Y are assigned values at lines 10 and 20. At line 30, an IF/THEN statement includes an OR operator. The two expressions are Y > 5 and X < 0. Since Y >5 is true, and X < 0 is false, the OR operation is evaluated as true.

Since the condition of the IF/THEN statement is true, the program control branches to line 50. At lines 50 and 60 the output message is printed. When the condition of the IF/THEN statement is false, the program is ended at line 40.

When the OR operation is executed with numeric expressions, the arguments are truncated and converted to their

binary equivalents. The value of the OR statement is the result of the OR operation on each bit of the values. For example:

	5	(binary 0101)
OR	11	(binary 1011)
	15	(binary 1111)

Since the binary equivalent of 5 is 101, and the binary equivalent of 11 is 1011, the OR operation for each bit has the result 1111. The numeric value of 1111 is 15. Any non-zero value is considered true.

PEEK

The PEEK function is used to recover the value in a memory location.

Configuration

X = PEEK *(a)*

A memory location contains an integer value between 0 and 255. The argument of a PEEK statement refers to the memory location. An illegal quantity error occurs if the argument is negative or greater than 65535. If the argument *(a)* is not an integer, it is truncated.

Example

PRINT PEEK (83)

POKE

The POKE statement is used to store one byte of information in a particular memory location.

Configuration

POKE *a*, *b*

The first argument of a POKE statement is the memory location. If a POKE statement specifies a memory location that does not exist, the POKE statement has no effect. Also, if a POKE statement specifies a memory location that is part of the ROM, the POKE statement has no effect.

The second argument of a POKE statement is the value that is to be stored at the specified memory location. The value of the second argument represents one byte. As a result, the value must be an integer between 0 and 255.

If either of the arguments of a POKE statement is not an integer, it is truncated. An illegal quantity error occurs if the memory location specified is greater than 65535 or the value of the second argument exceeds 255. An illegal quantity error also results if either of these arguments are negative.

If the POKE statement is not used carefully, it can seriously disrupt the operation of the computer.

Appendix G contains information regarding commonly used memory locations.

POS

The POS function is used to return the position of the cursor.

Configuration

X = POS(*a*)

The value of the argument has no effect on the results. The value returned is the column number where the next output is to occur on the display. The value is always an integer from 0 through 39.

Example

```
10 PRINT TAB(10);
20 PRINT POS(0)
RUN
          10
```

PRINT

The PRINT statement is used to output characters to the display or an output device.

Configuration

PRINT [#a][*data* { , ; } *data*...]

The first argument of a PRINT statement is the optional file number. The file number must be used only when an output device other than the display is used.

In general, a PRINT statement includes string and numeric variables, as well as string and numeric constants. Each variable name or constant must be separated by either a comma or semicolon. When a comma separates the items in a PRINT statement, the display is divided into four columns. When a semicolon is used between items in a PRINT statement, the values are displayed adjacent to each other, with one space preceding and following numeric values.

A PRINT statement can end with a comma, semicolon, or no punctuation at all. A PRINT statement that ends with a semicolon allows the cursor to wait at the next position until another PRINT statement is executed. The cursor waits at the next available column when a PRINT statement ends with a comma. When a PRINT statement has no punctuation at the end, the next line of output automatically occurs on the next line.

Example

```
10 X = 5
20 Y = 25
30 PRINT "PRICE $";X,
40 PRINT "UNITS:";Y
RUN
PRICE $ 5          UNITS: 25
```

In the previous example, the program contains two PRINT statements for one line of output. At lines 10 and 20, the variables are assigned values. At line 30, the string constant "PRICE $" is printed, followed by the value of the variable X. Since a semicolon separates the values, they are printed adjacent to each other, separated by one space. The PRINT statement at line 30 ends with a comma, so the next output occurs on the same line, in the second column. The string constant "UNITS:" is followed by the value of the variable Y. These values are also separated by one space.

When a PRINT statement includes the optional file number, the output is sent to another device, instead of the display. The file number specified in a PRINT statement must correspond to a previously executed OPEN statement for an appropriate device and operation.

An INPUT statement is generally used to recover data that was stored on cassette or disk with a PRINT statement. In order for numeric values to be read with an INPUT statement, each numeric value must be followed by an EOL character. This can be accomplished by including a single numeric value with each PRINT statement.

Example

```
10 OPEN 1,1,1, "EXAMPLE"
20 FOR J = 1 TO 100
30 X = RND (9)
40 PRINT#1,X
50 NEXT J
60 CLOSE 1
```

The program in the previous example uses a PRINT statement to record 100 random numbers on cassette tape. At line 10, an I/O channel is opened for output to the cassette recorder. The file on the tape is called "EXAMPLE". Line 20 initiates a FOR/NEXT loop that is repeated 100 times. At line 30, a random value is assigned to the variable X. At line 40, the value at X is printed on the output device opened as file #1. When 100 values are recorded, this file is closed.

In addition to the general use of the PRINT statement, a PRINT statement can also be used for special output to the display.

In a PRINT statement, the movement of the cursor is represented by characters. When quotation marks are used in a PRINT statement, the cursor control keys do not move the cursor, but display a character instead. When the cursor control characters are encountered in an execution of a PRINT statement, the cursor moves appropriately. The cursor control characters have the following results.

```
CURSOR UP      "●"
CURSOR DOWN    "Q"
CURSOR LEFT    "]"
CURSOR RIGHT   "]"
```

Example

```
10 A$ = "VIC-20"
20 PRINT "●]]]]]] ";A$
30 END
RUN     VIC-20
READY
```

In the preceding example, the variable A$ is assigned the value "VIC-20" at line 10. At line 20, a PRINT statement includes the cursor control characters. The first character in quotation marks causes the cursor to move up one line. The next six characters each cause the cursor to move one space

to the right. After the cursor is moved, the value of A$ is displayed.

When the previous example program is executed, the output is displayed on the same line as the READY statement. This occurs because the cursor is moved up one line, and six spaces to the left before the value of A$ is displayed.

The CLR and HOME display command keys can also be represented with special characters in a PRINT statement. In a PRINT statement, in quotation marks, the character "S" is generated when the HOME key is pressed. Also, the character "♥" is displayed when the CLR key is pressed. These characters cause the screen to be cleared, or the cursor to be moved to the home position when they appear in a PRINT statement.

A PRINT statement can also determine the color of the output. Within a PRINT statement, in quotation marks, the numbered keys (1-8) control the color of the output. When each of these keys are pressed along with the Control key, or the Commodore key, they display special characters. When these characters are encountered in a PRINT statement, the color of the output changes.

The characters in a PRINT statement can also be displayed in a reverse mode. In a PRINT statement, the keystroke "Control-9" (when enclosed within quotation marks) causes the subsequent characters in that PRINT statement to be displayed in the color of the background (like a photographic negative).

When the "Control-9" combination of keys is entered, the character " R " is displayed. Any subsequent values are printed in the reverse mode until a " ▬ " character ("Control-0") is encountered, or the end of the line of output. The line of output is ended if the PRINT statement does not end with a comma or semicolon. Also, the line of output ends when a CHR$ (13) character is printed.

For example, the statement

PRINT " R VIC ▬ 20"

would cause the output VIC 20. The first character in quotation marks is the "Control-9" combination. The subsequent characters are printed in reverse until the "Control-0" character is encountered.

READ

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 { X / X$ [,Y / ,Y$] . . . }

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 a 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
```

(continued on next page)

```
50 DATA 12, JONES
RUN
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.

REM

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

A RESTORE statement is used to move the DATA statement pointer.

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

RESTORE

RETURN

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 the program control to return to the statement following the most recently executed GOSUB or ON/GOSUB statement.

Example

RETURN

RIGHT$

The RIGHT$ statement is used to return the rightmost characters of a string.

Configuration

Y$ = RIGHT$ (*string, a*)

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
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 0, a null string is returned.

RND

The RND function is used to generate "random" numbers.

Configuration

X = RND (*a*)

The computer uses a mathematical process to generate pseudo-random numbers greater than or equal to 0 but less than 1. The argument of the RND statement has an effect on the number that is returned. If the argument of a RND

statement is non-negative, the "random" number is derived from the previous "random" number. As a result, if a program has RND statements with constant arguments, the sequence of "random" numbers will be the same each time the computer is turned on.

An RND statement with a negative argument chooses a value independent of the last "random" number. However, for a fixed negative argument, the RND function returns the same value.

The most effective way to generate "random" numbers is to execute an RND (-TI) first, followed by RND statements with positive arguments. TI is the name of a variable that keeps changing while the computer is operating. As a result, the value of RND(-TI) is very rarely repeated.

Example

```
X = INT(RND(9)*100)
```

The previous example contains a statement that generates random integers from 0 to 99.

RUN

The RUN statement is used to execute the program that is currently in the computer's memory.

Configuration

RUN [*linenumber*]

The RUN statement can include a line number as an argument. If the specified line number appears in the program, execution begins with that line. If the specified line number does not appear in the program, the following message is displayed:

?UNDEF'D STATEMENT ERROR

If the RUN statement does not include an argument, execution begins at the first line of the program.

Example

RUN
RUN 100

SAVE

The SAVE statement is used to record a file on cassette or disk.

Configuration

SAVE [*"filename"*] [,*a*] [,*b*]

If a SAVE statement is executed with no parameters, the program that is currently in RAM is recorded on the cassette tape. If the optional filename is included in the SAVE statement (in quotation marks), the file is recorded under the specified filename. The filename must be included if the file is to be recorded on disk.

The second argument indicates the device. If the second argument is the number 8, the file will be recorded on disk. If the second argument is not used, or is the value 1, the file is recorded on the cassette.

If the second argument is 1 (cassette), the optional third argument can also be included. If the third argument is the value 1, an End of Tape marker is recorded after the file.

A SAVE statement records a file on tape at whatever position the tape is at. Be careful to position the tape at a place where no other files are present, or the previously recorded files will be erased.

A LOAD statement is used to recover files that were recorded with a SAVE statement.

Examples

```
SAVE "EXAMPLE"
SAVE "RESULTS",8
SAVE"RECORDS",1,1
```

The first example saves a file on cassette named EXAMPLE. The second example saves a file on disk named RESULTS. The third example saves a file named RECORDS on cassette, followed by an End of Tape marker.

SGN

The SGN function returns a +1 if its argument is positive, a -1 if negative, and a 0 if zero.

Configuration

```
X = SGN (a)
```

Example

```
100 A = 100
200 X = SGN (A)
300 PRINT X
RUN
1
```

SIN

The SIN function returns the sine of the angle specified as its argument. The argument will be assumed in radians.

Configuration

```
X = SIN (a)
```

Example

```
PRINT SIN (3.1415927/2)
.999999992
```

SPC

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
RUN
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.

SQR

SQR returns the square root of its argument.

Configuration

X = SQR (*a*)

Example

```
10 X = 49
20 PRINT SQR (X)
RUN
7
```

STOP

The STOP statement causes a halt in the execution of a 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.

STR$

STR$ returns the string representation of its argument.

Configuration

X$ = STR$(*a*)

Example

```
10 A$ = STR$(40)
20 PRINT A$
RUN
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".

SYS

The SYS statement is used to begin execution of a machine language program.

Configuration

SYS *location*

The argument of a SYS statement is the memory location where the machine language program begins. The argument must have a value between 1 and 65535 inclusive. When the machine language program is complete, the program control returns to the statement following the SYS statement.

Example

```
SYS 4450
```

TAB

The TAB statement is used to specify the column where the next item in a PRINT statement will be output.

Configuration

TAB (*a*)

The output that follows a TAB statement begins at the column specified by the argument.

Example

PRINT X$;TAB(8);Y$

In the previous example, the value of the variable X$ is displayed, starting in the first column. The value of the variable Y$ is displayed, starting in column number 8.

If the next available column is greater than the argument of a TAB statement, the TAB statement has no effect. The next output occurs at the next available column.

USR

The USR function is used to call an assembly language subroutine.

Configuration

X = USR(*a*)

When a USR statement is executed, the value of the argument is stored in the floating point accumulator. The value of the starting address of the machine language subroutine is determined by the values of memory locations 1 and 2.

The machine language subroutine operates on the value specified in the USR statement. When the subroutine is complete, the USR statement is assigned the value calculated in the subroutine (the final value of the floating point accumulator).

Example

```
X = USR(Y)
```

In the preceding example, the variable X is assigned the result of a machine language subroutine. The USR function will result in an error unless a machine language program actually exists.

VAL

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 a non-numeric string character is encountered. If the first character of a string is not a numeric value, the VAL function returns the value 0.

Configuration

X = VAL (*string*)

Example

```
100 A$ = "57A72B"
200 PRINT VAL(A$)
300 PRINT VAL(A$) + 2
RUN
57
59
```

VERIFY

The VERIFY statement is generally used to check if a program was recorded accurately.

Configuration

VERIFY [*program name, device*]

When a VERIFY statement is executed, the specified program (or the next program on cassette) is compared with the program in the computer's memory. If the two programs are exactly alike, the message,

```
OK
READY
```

is displayed on the screen. If the two programs are not exactly alike, the following message is displayed:

```
?VERIFY ERROR
READY
```

The VERIFY statement can be used with the cassette recorder to VERIFY a specific file, or the next program on the tape. The VERIFY procedure is very similar to the LOAD procedure. The display is cleared while the cassette recorder operates. Also, the recorder stops each time a program is encountered, and waits for the Commodore key to be pressed. In order to verify a program on the disk, the program name must be specified as well as the device code (8).

Generally, the VERIFY statement is executed after a program was saved. The results of the VERIFY procedure indicate if the program was saved without any errors.

The VERIFY procedure can also be used to locate free space on a cassette tape. If a VERIFY statement includes the name of the last file on the cassette tape, the computer will search for the specified file, and compare it with the current program in RAM. When the VERIFY procedure is complete, the tape will be located at the end of the last file on the tape.

Examples

```
VERIFY "PROGRAM3",8
VERIFY "BUDGET"
```

WAIT

In general, the WAIT statement is only used in unusual I/O operations. The WAIT statement halts the execution of the program until the value of a memory location satisfies the specified conditions.

Configuration

WAIT *X,Y[,Z]*

The first argument of the WAIT statement specifies a memory location. The second and third arguments are used to check the value of the memory location.

All of the operations of the WAIT statement are logical. Therefore, the numeric arguments are converted to their binary values. First, the binary value of the memory's value is compared to the second argument (bit by bit) with the AND operator. As a result, only those bits that equal 1 in both values remain. In effect, the AND operation is used to select only the relevant bits. If the third argument is not present, the program continues when the result of the AND operation is non-zero.

If the third argument is present, another operation is performed. The third argument is also converted to binary, and compared to the result of the AND operation. However, these values are compared using the exclusive-OR operation. The value of the result is zero unless the two values are not equal. The program will continue if the result is non-zero.

Example

```
10 POKE 4444,16
20 WAIT 4444,16,16
```

In the previous example, the value 16 is assigned to the memory location 4444. At line 20, the WAIT statement is

executed for the value of memory location 4444. Since the value of the specified location is 16, the operation 16 AND 16 is executed. Since 16=00010000 in binary, 16 AND 16 equals 16. The value of the AND operation is then compared to the third argument. Since 16=16, every bit of the two values are alike. As a result, the exclusive-OR operation returns all zeros. When this occurs, the program cannot continue.

CHAPTER 5.
COMMODORE 64 DATASSETTE CASSETTE RECORDER

INTRODUCTION

The Commodore 64 Datassette is used for storing BASIC programs or data on cassette tape. The process of transferring a program from RAM onto cassette tape (or any other storage device) is known as **saving** that program. Once a program has been saved, it can later be transferred back from the storage device into RAM. This process is known as **loading.**

Data can also be transferred back and forth between RAM and cassette tape. The process of sending data to cassette tape is known as **writing** the data. The retrieval of that data from cassette tape back into RAM is known as **reading** the data.

In this chapter, we will discuss the BASIC statements used to read and write data and to save and load programs. However, first we will discuss the concepts of data and program storage.

INSTALLING THE DATASSETTE

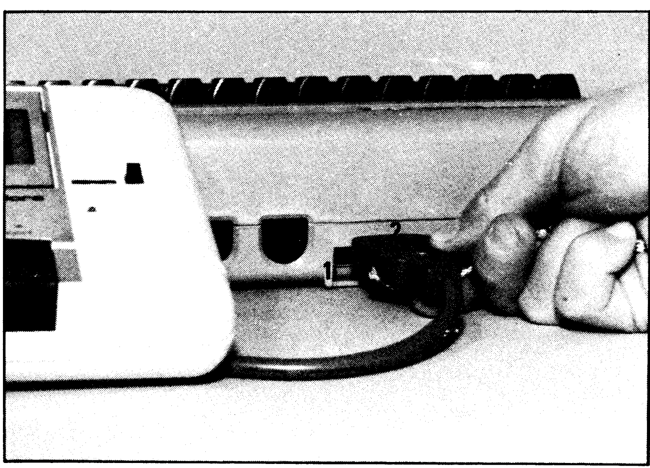
The Datassette has only one cord, which is connected to the Commodore 64 console. The computer console has six recepticles on the back for peripheral devices. Three of the recepticles are round, and the other three are rectangular. The smallest rectangular recepticle is used with the Datassette.

The plug of the Datassette has a long, narrow opening which accomodates the long, flat connector in the recepticle opening.

The connector on the computer console has a notch cut into it. The plug from the Datassette has a post which corresponds to this notch. When the plug is aligned correctly, it presses onto the connector easily, but firmly. When the plug is upside-down, the post and notch do not line up, and a good connection is impossible.

Refer to Illustration 5-1 as a guide to the installation of the Datassette to the Commodore 64.

Illustration 5-1. Installing the Datassette



1. Recepticle 2. Datassette Plug

DATA FILES -- FILES, RECORDS, & FIELDS

Data files can be visualized as being organized as **files, records,** or **fields.**

If we visualized the Datassette as a filing cabinet, a data file would be analogous to one file within that filing cabinet. For instance, if you kept a file filled with slips of paper containing the names and addresses of all of your cousins, that physical file would be analogous to a computer's data file.

Your data file could contain any number of slips of paper -- depending upon how many cousins you had. Each slip of paper containing the name and address of one of your cousins would be analogous to a **record** within a data file.

Each individual data item within a record is known as a **field.** In our example, the name of each cousin might be considered a field as well as the street address, city, state, zip code, and telephone number.

PROGRAM FILES

Programs are also stored as files. However, unlike data files, program files are not divided into records and fields.

We will discuss loading and saving program and data files in the following several sections.

Prompts

If none of the levers on the Datassette are engaged, the computer will display the correct instructions. However, if any of the levers on the Datassette are depressed, the prompts do not appear on the screen. If the wrong lever is pressed when the computer attempts to use the Datassette, the computer proceeds as if the correct levers were pressed.

If an incorrect lever is pressed, the programs and data on the tape may be destroyed. Always check the Datassette before executing an input or output command to be certain that the correct levers are depressed.

Saving a Program File

The SAVE command is used to send a program to the cassette unit. SAVE does not affect the program in RAM. When a program is saved on tape, it is automatically recorded twice, so as to allow the Commodore 64 to check for errors when the program is loaded back into RAM.

The following procedures should be followed to save a program on tape.

1. Enter SAVE followed by pressing Return. If you wish to include a name with the program on tape, you can do so by specifying the name after SAVE enclosed in quotes or by specifying the filename as a string variable. The program name can consist of up to 16 characters.

2. Once SAVE has been entered, the following message will appear on the screen:

   PRESS RECORD & PLAY ON TAPE

   Be certain that the recorder contains a blank cassette, then press the Play and Record buttons.

   The screen is cleared while the program is being saved. When the SAVE operation is complete, the cassette recorder stops, and the display resumes. The following messages appear on the screen after the program is recorded:

   SAVING *program name*
   READY.

Verifying a Program File

The VERIFY command is generally used after the SAVE command to check the program in the Commodore 64's RAM. This task is undertaken to be sure that the program was properly recorded.

The following procedures should be followed to verify a program stored on cassette tape.

1. Rewind the tape to an appropriate position.

2. Enter the VERIFY command. If a program name is included in the command, the computer will search for that program. If no program name is included, the next program on the tape will be verified.

3. The following message will then be displayed on the screen:

   PRESS PLAY ON TAPE

4. Once the Play button has been pressed, the recorder begins to operate, and the screen is cleared.

   The cassette recorder stops when the first file is encountered. The display resumes, and the following message appears:

   SEARCHING FOR *program name*
   FOUND *program name*

   The Commodore logo key, on the lower left side of the keyboard, is used to continue the VERIFY sequence.

   If the VERIFY statement did not include a filename, the file on the tape is checked against the program in memory when the Commodore key is pressed.

   If the VERIFY statement included a filename, the

computer will continue searching until the specified file is found. Each time a file is encountered, the recorder stops, and the message FOUND is displayed, followed by the filename (if the file has a name). Each time the recorder stops, the Commodore key can be used to continue.

5. When the appropriate file is located, the program stored on tape is checked against the program in memory. If the two copies match, the following message will appear:

```
VERIFYING
OK

READY.
```

If the two copies do not match, the following message will appear:

```
VERIFYING
?VERIFY ERROR
READY
```

If the copies do not match, try saving the program again.

The VERIFY command can also be used to position the tape past the end of the last program saved on it, so that any subsequent programs will not overwrite existing programs.

By executing VERIFY, the Commodore 64 will check the program on tape against the new program to be saved. Since these programs won't match, a Verify Error will appear. The new program in RAM will remain unchanged and the tape will be positioned past the end of the program just verified.

Be sure to change the Datassette from the playback to the record mode before attempting to save the new program onto tape.

Loading a Program File

The LOAD command is used to load a program from tape or disk in the Commodore 64's RAM. The following procedures should be followed to load a program from tape.

1. Enter the LOAD command and press the Return key. If the program name is included in the LOAD command, the computer will search for the specified program.

2. Once the LOAD command has been entered, the following message will be displayed:

   PRESS PLAY ON TAPE

   The display is cleared when the cassette recorder begins to operate.

   If the LOAD statement did not specify a filename, the next program on the tape is loaded into memory. When the beginning of the first file is encountered, the following messages are displayed:

   SEARCHING
   FOUND

   The computer waits for the Commodore key to be pressed before the program is loaded.

   When the Commodore key is pressed, the display is cleared and the cassette recorder is activated. When the program is loaded, the recorder stops, the display resumes, and the following messages are displayed:

   LOADING
   READY.

   If the LOAD statement specified a filename, the computer searches for the appropriate file. When the first file is encountered, the following messages are displayed:

SEARCHING FOR *program name*
FOUND *program name*

The computer waits for the Commodore key to be pressed before the process is continued. If the specified file is found, the program is loaded. If the specified file is not found, the recorder proceeds to the next file.

When the specified file is located, and the program has been located, the following messages are displayed:

LOADING
READY.

The RUN key on the keyboard can also be used to load and execute the next program on the tape. The RUN key is equivalent to the LOAD and RUN statements, but it is faster and easier.

READING AND WRITING DATA TO THE DATASSETTE

Commodore BASIC uses the PRINT# statement to write data to the Datassette. The INPUT# and GET# statements are used to recover data stored on cassette.

Opening Data Files

Before information can be read from or written to a data file, that file must first be opened. This is accomplished with the OPEN statement.

The data file can be read from or written to as long as it is open for the appropriate operation. To prevent access to the file, it must be closed. This is accomplished with the CLOSE statement.

The OPEN statement opens a channel for input or output to a device. That device can be an integral part of the Commodore 64 such as the screen or keyboard, or a completely external device such as the cassette unit, printer, or disk drive. When the OPEN statement is executed, a buffer is also set up for transmitting and/or receiving data.

The OPEN statement uses the following configuration:

OPEN *file#, device#, command#, string*

The *file#* can range from 1 to 255. This will be the same number specified with any GET#, PRINT#, and INPUT# statements associated with the same device.

device# specifies the device being opened. Device numbers are preassigned. These are listed on page 119. The device number for the cassette unit is 1.

command# refers to the operation for which the device is being opened. For the cassette unit, the *command#* is as follows:

0 -- read the tape file.
1 -- write to the tape file.
2 -- write to the tape file and place an End of Tape
marker at the end of the file.

The *string* is used with the cassette unit to specify a filename.

For example, the following OPEN statement,

OPEN 1,1,1, "TEXT.DAT"

would open channel 1 for writing to the file named TEXT.DAT.

Closing Data Files

Once an open file has been accessed, it is important to close that file so its channel can be assigned elsewhere. Also, if an open file is not closed, the FILE OPEN ERROR will occur if another OPEN statement is executed for that channel.

CLOSE is used with the following configuration:

CLOSE *file#*

For example, CLOSE 4 would close the channel opened under *file#* 4.

The channels are closed automatically when a RUN or NEW command is executed.

Writing to a Cassette Data File

As mentioned previously, the PRINT# statement can be used to send data to the cassette unit. PRINT# is used with the following configuration:

PRINT# *file#, data* [, or ; *data...*]

file# refers to the *file#* with which a channel was opened. *data* refers to the data to be output. This data can either be string or numeric constants or variables.

Numeric values which are output to the tape must be separated by the EOL character. Each numeric value must be either the only item in a PRINT# statement, the last item in a PRINT# statement, or must be followed by the EOL character (CHR$(13)).

The EOL character is automatically generated after execution of PRINT# unless a comma or semicolon appears at the end of the PRINT# statement. For this reason, a numeric variable cannot be followed by a comma or semicolon at the end of a PRINT# statement.

The following example illustrates three correct methods for using a PRINT# statement to output numeric values.

Example

```
100 OPEN 1, 1, 1
110 PRINT#1,A:PRINT#1,B:PRINT#1,C
120 PRINT#1,A;CHR$(13);B;CHR$(13);C
130 A$ = CHR$(13):PRINT#1,A;A$;B;A$;C
140 CMD 1:PRINT A:PRINT B:PRINT C:PRINT#1
150 CLOSE 1
```

The preceding example sends the values of the variables A, B, and C to the cassette recorder. At line 100, an I/O channel is opened for output to the Datassette.

At line 110, each PRINT# statement includes only one variable name. At line 120, only one PRINT# statement is used, but each variable is separated by the end-of-line character CHR$(13). At line 130, the string variable A$ is assigned the end-of-line character. In the PRINT# statement, each numeric variable is separated by A$. At line 140, the CMD command is used to make the device on I/O channel #1 the general output device. As a result, PRINT statements are used instead of PRINT# statements. The PRINT# statement at the end of line 140 is required by the CMD statement.

String values can be separated by semicolons, commas, or the EOL character. String output to the tape unit uses the same format as output to the screen. A string value that is followed by a comma in a PRINT# statement is recorded along with enough blank spaces to fill the current column on the screen. A string value that is followed by a semicolon in a PRINT# statement is recorded without any additional spaces.

String output to the Datassette is concatenated, and considered as one string value until the EOL character is encountered. As a result, several string values that are separated by semicolons (or commas) in a PRINT# statement will be regarded as one string value when they are read back into RAM with an INPUT# statement.

If you wish to send several string values to the Datassette via the PRINT# statement and read these back as individual strings via the INPUT# statement, each string must be separated with an EOL character. This is illustrated in the following example.

Example

```
10 A$ = "SEE"
20 B$ = "SPOT"
30 C$ = "RUN"
40 OPEN 1, 1, 1
50 PRINT#1,A$;B$;C$
60 PRINT#1,A$,B$,C$
70 PRINT#1,A$:PRINT#1,B$:PRINT#1,C$
80 CLOSE 1
```

In the preceding example, the string variables A$, B$, and C$ are assigned values at lines 10 through 30. At line 40, I/O channel #1 is opened for output to the cassette recorder.

At line 50, the three string values are output to the Datassette, separated by semicolons. The variables are recorded as one string value, SEESPOTRUN. When an INPUT# statement is used to recover the data, the three values can no longer be distinguished from one another.

At line 60, the three string values are output, separated by commas. Again, the values are recorded as one string value, but blank spaces are inserted between the values. The length of the string is 33 characters, including blank spaces. However, even though the data items are separated by blank spaces, the three data items are now considered one string value.

At line 70, each data item has a separate PRINT# statement. In this case, the data are recorded on the tape as individual string values. The values can be recovered individually with INPUT# statements. In effect, each data item is separated by the EOL character. As a result, separate string values are recorded in the same way as numeric values.

Reading Data From a Cassette Data File

INPUT# and GET# are used to read from a cassette data file. INPUT# reads one or more bytes or data into one or more variables, while GET# reads a single character of data into the variable specified.

INPUT# is used with the following configuration:

INPUT# *file#, variable,...*

The *file#* specified must currently be open for a read operation. Data will be read from the specified tape file into the *variable* or *variables* specified. This data will consist of ASCII characters. Data will be read into the variable until a CR (ASCII 13) is encountered.

INPUT# will interpret the data being read as either numeric or string, depending on the type of *variables* used as parameters. When a numeric variable is specified, the data being input will be interpreted as numeric data.

If no data is available to be read into the numeric variable, or if the data is invalid, an error will result.

When a string *variable* is specified, the data being input will be interpreted as string data. If no characters are read, the string variable will be assigned the null value.

The GET# statement uses the following configuration:

GET# *file#, variable*

The *file#* specified must currently be open for a read operation. A single byte can be read from the specified tape file into the *variable* specified. If no data is available, the null value will be supplied.

Unlike the INPUT# statement, the GET# statement can be used to receive special characters such as CR or quotation marks.

CHAPTER 6.
1541 DISK DRIVE

INTRODUCTION

The Commodore 1541 Disk Drive* is used for storing BASIC programs or data files on floppy diskettes.

A disk stores data in a magnetic form, much like data is stored on magnetic tape. The main difference between storage on a magnetic tape and storage on a disk is that the disk surface is round -- much like a record's surface.

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

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

---

* The VIC 1540 Disk Drive can be modified by your Commodore dealer to be used with the Commodore 64 computer. References to the 1541 Disk Drive in this chapter apply to the 1540 as well.

TYPES OF DISKS

There are three primary types of disks used by microcomputers; **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 6-1.

Floppy Diskettes

The most widely used type of disk storage with microcomputers is floppy disk storage. 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.

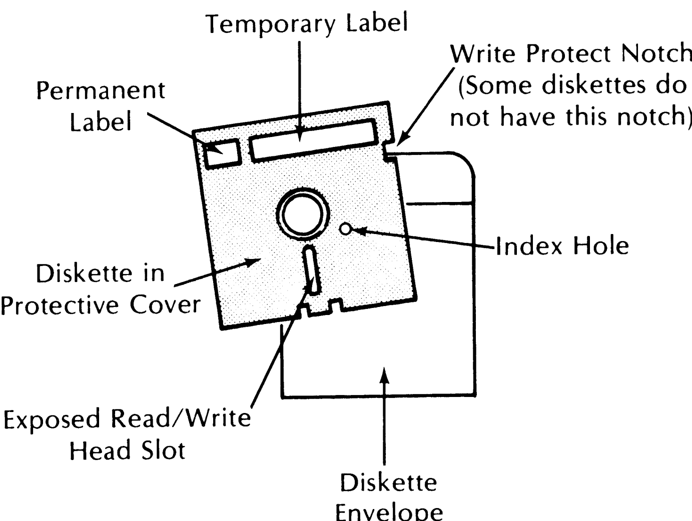
Illustration 6-1. Winchester Disk System

This cover protects the diskette from damage while it is being handled by the operator. A 5¼ inch diskette with its protective envelope is shown in Illustration 6-2.

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.

Floppy diskettes come in two sizes: 8 inch and 5¼ inch. The 5¼ inch diskettes are also known as mini-floppy diskettes. The 1541 Disk Drive uses mini-floppy diskettes.

Illustration 6-2. Mini-Floppy Diskette



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 6-3. Commodore's DOS divides a diskette into 35 tracks.

To further reduce the time necessary to search for a particular data item, DOS divides each track into 17 to 21 sectors, which are also shown in Illustration 6-3.

With 35 tracks available and 17 to 21 sectors per track, DOS divides each diskette into 683 sectors. However, 19 of these 683 sectors are used by DOS, and cannot be used to store programs or data.

The tracks on the diskettes used by the 1541 are numbered from track 1 at the outside of the diskette sequentially to track 35 at the inside of the diskette.

DOS fills the diskette with data from track 1 inward to track 35. Track 18 is reserved for the diskette directory.

As mentioned previously, each track is subdivided into sectors. The number of sectors on a track can vary depending upon the track number. Obviously, the outer tracks (those with lower track numbers) have more room and can store more data than the inner tracks (those with higher track numbers).

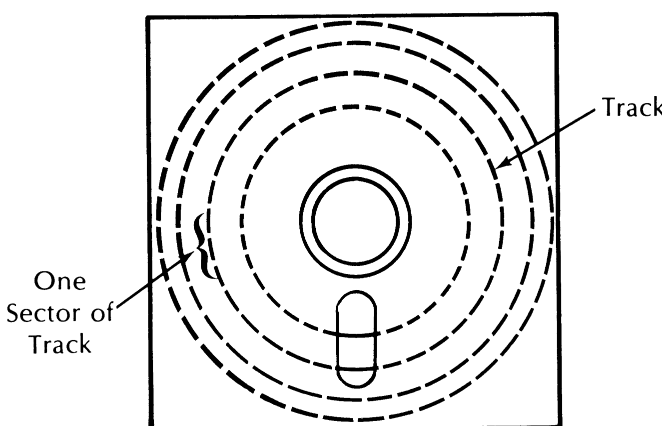
The number of sectors that the different tracks contain are listed in Table 6-1.

Table 6-1. Track and Sector Allotment

Track #	No. of Sectors	Sector No. Range
1-17	21	0-20
18-24	19	0-18
25-30	18	0-17
31-35	17	0-16

Each individual sector holds 256 bytes of data. When DOS has access to the track and sector where a particular data item is being stored, it will only have to search 256 bytes to find that item. The result of dividing the diskette surface into tracks and sectors is that access time is greatly decreased.

Illustration 6-3. Tracks and Sectors



Hard and Soft Sectors

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

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

Both the hard and soft sector methods involve the use of an index hole. The index hole is shown in Illustration 6-2. 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 6-2, 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 of 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 sectored diskettes. A hard sectored diskette contains a number of holes, each of which indicates the location of a sector. An extra hole is used to indicate the location of the first sector. The location of the various sectors is determined by counting the number of holes occuring after the first sector. A hard sectored diskette is depicted in Illustration 6-4.

Soft sectored diskettes have only one index hole as shown in Illustration 6-5. 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 1541 uses soft sectored diskettes.

Illustration 6-4. Hard Sectored Diskette

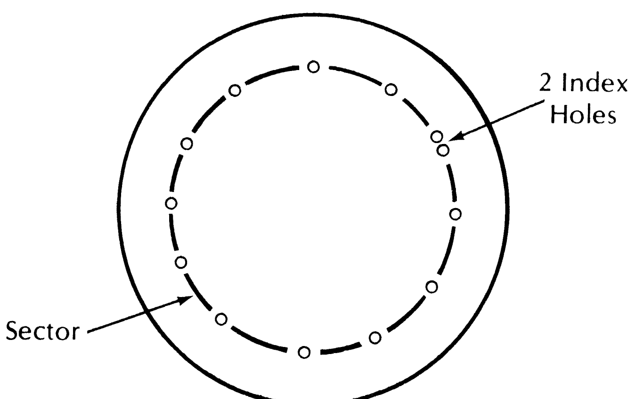
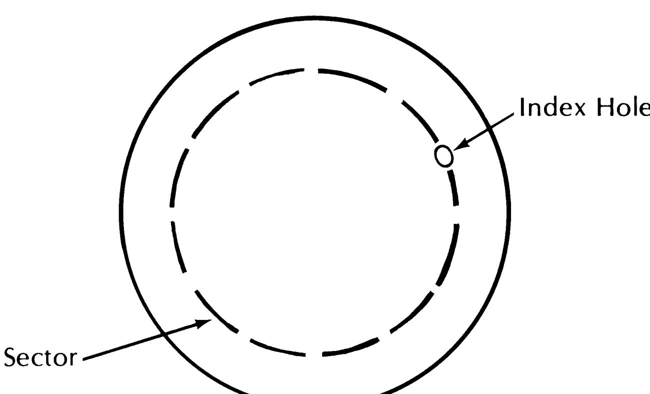


Illustration 6-5. Soft Sectored Diskette



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.

The 1541 uses single sided diskettes with a total storage capacity of 174,848 bytes. This figure is calculated by multiplying the 683 sectors by 256 bytes per sector.

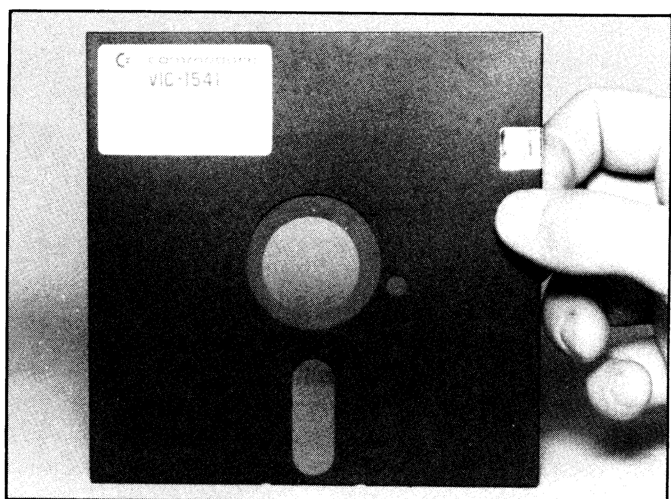
Diskette Write Protection

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 an 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 6-6.

Illustration 6-6. Write Protecting a 5¼ Inch Diskette



Additional 1541 Features

The 1541 contains 16K of ROM and 2K of RAM which is a sufficient amount of memory for both the disk controller and disk operating system (DOS). This memory means that the 1541 does not utilize any of the computer's memory during operation.

The 1541 uses a **pipeline** operating system. A pipeline operating system enables the disk unit to undertake DOS operations while the computer is performing some other operation. This greatly improves the efficiency of the computer system.

The 1541 disk drive is both read and write compatible with

the Commodore 2031 and 4040 drives. In other words, the same diskettes can be used on the 1541, Commodore 2031, and Commodore 4040 drives. The 1541 can also read information from diskettes created with the Commodore 2040 drive.

The 1541 uses a dual serial bus interface. The serial bus interface is similar to the IEEE-488 interface (used on Commodore's PET series of computers). However, the serial bus uses only one wire to transmit data, while the IEEE-488 uses eight wires.

Notice that there are two serial bus ports on the rear of the 1541 drive (see Illustration 6-7). This allows additional devices to be connected to the Commodore 64 by having these devices share the same serial bus.

These devices are installed by connecting them to another device's free serial bus port. In other words, each device is installed by connecting it to another device in a daisy chain manner. As many as five disk drives and one printer can be connected to the single serial bus in this fashion.

INSTALLING THE 1541

Upon unpacking the 1541 disk drive, you will find the disk drive unit, a grey power cable, a black serial bus cable, a manual, and a demonstration diskette.

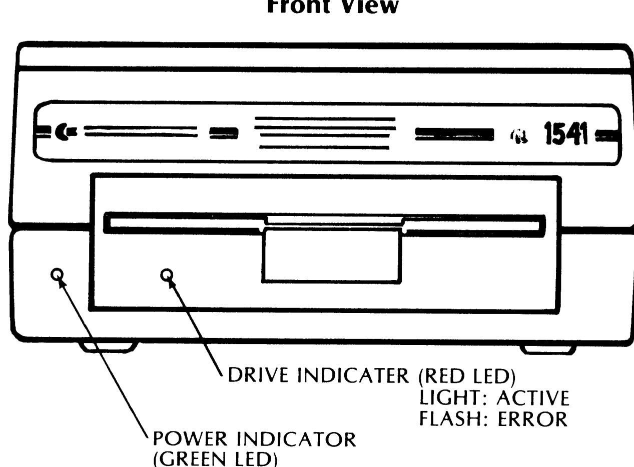
One end of the grey power cable should be connected to the AC input port on the rear of the 1541. The other end should be connected to a household AC outlet.

The serial bus cable is identical at both ends. Each end consists of a 6 pin DIN plug. Attach one end to the serial port on the disk drive and other end to one of the serial bus ports on the rear of the Commodore 64. However, be certain that the computer and the disk drive power switches are off before connecting the serial bus cable.

If you wish to install a second disk drive or a printer, you can do so by connecting a serial bus cable between that device and the second serial bus port on the rear of the 1541 connected to the Commodore 64. Refer to Illustration 6-8 for a typical installation.

Illustration 6-7. 1541 Disk Drive

Front View



Rear View

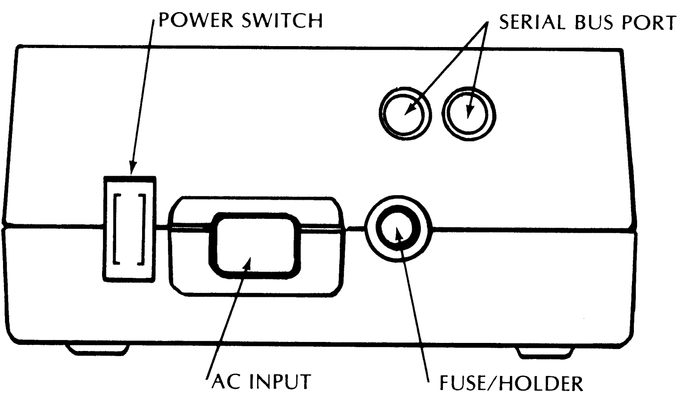
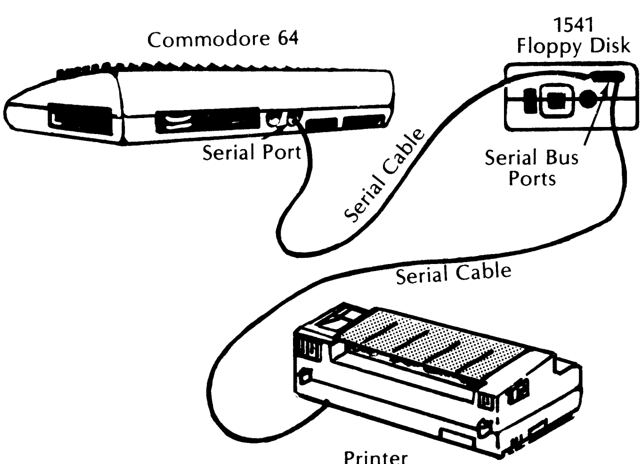


Illustration 6-8. Typical Commodore 64 Installation With More Than One Device



Powering On

Once you have connected the disk drives and printer to your Commodore 64, you can begin powering on these devices. Be certain to power on these devices in the correct order.

The computer console should be the last device to be powered on. It does not matter in what order the various disk drives and printer are powered on as long as the Commodore 64 is the last device powered on.

When the printer is powered on, the print head will move across the print line to its midpoint and then back again.

Before powering on the 1541 disk drive, be certain that any diskettes have been removed. When the 1541 is powered on, the red and green lights on the front of the unit will both light. The red light indicates the drive is active when it is

lighted. If the red light is flashing, an error is indicated. When the green LED is lighted, the power to the 1541 is on.

The drive will spin for several seconds with both the red and green lights on. The drive will then stop spinning, and the red light will be turned off.

When the Commodore 64 is powered on (after the 1541 has been powered on), both the red and green lights again will be lighted for several seconds as the drive spins. The red light will then be turned off, and the Ready prompt will appear on the video display.

Once the red light has been turned off on the 1541 drive, you may safely use the drive.

Inserting a Diskette into the 1541

First of all, never power the 1541 on or off without first removing any diskette in the drive. Doing so may destroy data on the diskette.

If the 1541's disk drive door is closed, it can be opened by gently pushing inward against the drive door latch. The drive door will then open. If a diskette is present in the drive when the drive door is opened, it will be ejected slightly.

Diskettes should be inserted into the 1541 with their label facing up. The write protect notch should be on the left side of the diskette as it is being inserted. The large oblong opening in the diskette cover should be on the side of the diskette facing the drive.

Gently slide the diskette into the drive. When the diskette is in place, you will feel a slight click, and the diskette will not be pushed back out of the drive. Once the diskette is in place, close the drive door by pulling the drive door latch down until it locks into place.

Using the 1541 with the VIC-20 & Commodore 64

The 1541 disk drive can be used with either the VIC-20 or the Commodore 64 computers. The 1540 disk drive can be modified by your Commodore dealer for use with the Commodore 64. The 1540 without modification will work only with the VIC-20.

The VIC-20 and Commodore 64 require data to be input at different speeds. The 1541 is set up to use a speed compatible with the Commodore 64. However, a software switch is available for selecting the speed to be used for data transfers.

To change to the speed used by the VIC-20, enter the following command once the drive has been powered on:

```
OPEN 15,8,15 "UI-":CLOSE 15
```

The following command will return the speed that is used by the Commodore 64:

```
OPEN 15,8,15,"UI+":CLOSE 15
```

The preceding commands incorporate the User command. These will be discussed in more detail later in this chapter.

DISK FILES

The Commodore 64 stores data in **files.** A disk file can contain either a BASIC program, a machine language program, or data. A 1541 diskette can contain as many as 144 files.

Files are assigned unique **filenames** of up to 16 characters. Filenames are optional for cassette and printer files, but are required for disk files.

Filenames must consist of a string of 1 to 16 characters. String variables can be used to represent filenames, as long as that

variable has been previously defined in the program.

Filenames often contain a suffix of three or fewer characters which designate the file type. The filename and filename extension generally are separated with a period. Together, the filename, filename extension, and the period separating the two must consist of 16 or fewer characters. Some of the more commonly used filename extensions and the type of file they represent are summarized in Table 6-2.

Table 6-2. Commonly Used Filename Extensions

Filename Extension	File Type
ASM	Assembly language source file.
BAK	Backup File.
BAS	File containing a BASIC program.
DAT	Data File.
OBJ	Assembly language program assembled into machine language. Also known as an object file.
TXT	Text File.

Filename Match Characters

Commodore DOS allows the use of the filename match characters, ? and *. These characters can be used to stand for any single character (?) or group of characters (*). for example, FILE?.DAT would match the following filenames:

FILE1.DAT
FILEZ.DAT

FILE?.DAT would not match the following filenames:

FILE.DAT
FILE1.BAS

FILE.* would match all of the following filenames:

FILE.BAS
FILE.TXT
FILE.DAT

Sequential vs. Random Access

The Datassette tape unit as discussed in Chapter 5, is a sequential device. In other words, it must read from the beginning of the tape directly to the end. The Datassette cannot jump around to different locations on the tape in order to read or write data.

The 1541 disk drive is a random access device. The 1541's read/write head can skip around the disk and access any one of the 683 blocks of data stored on the diskette.

Block Availability Map

Trying to keep track of which of the 683 available blocks on a diskette are free and which are in use would be a difficult task. Fortunately, Commodore's Disk Operating System (DOS) keeps track of the blocks with the Block Availability Map, or BAM.

The Block Availability Map is a list of the 683 blocks on the diskette. Each block is identified as either being in use or not being in use.

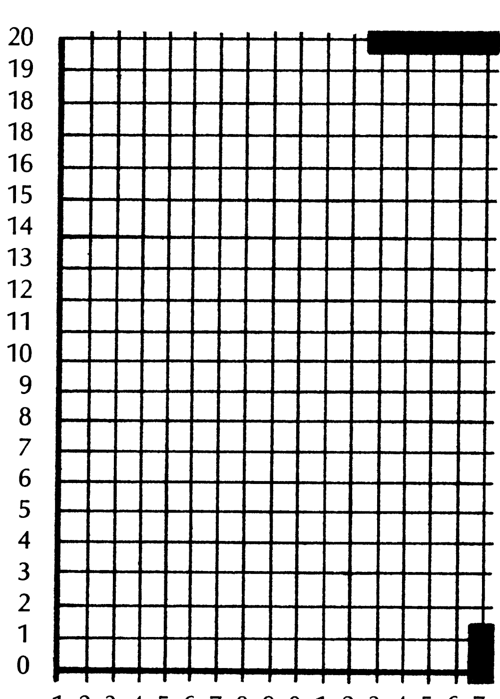
The BAM is stored in the middle of the diskette. The BAM is constantly updated as program files are saved and data files are closed. Note that the BAM is not updated until a file is closed. If a file is not closed, the data in that file may be lost.

The program VIEW BAM (found on the TEST/DEMO diskette supplied with your 1541) can be used to list a facsimile of the BAM on your screen. All 683 sectors will be depicted in a grid with the track number given at the bottom of the grid and the sector number along the left hand side.

A sample grid is pictured in Illustration 6-9. After the grid has been displayed, the number of Blocks Free will be displayed. The user will then be prompted by VIEW BAM to determine whether he wishes to display a grid for another diskette.

Refer to Table 6-3 for details on BAM format.

Illustration 6-9. VIEW BAM Grid



* NA in BAM signifies that the sector indicated is not available.

** Darkened areas indicate sectors with data written to it.

Table 6-3. BAM Format (Track 18, Sector 0)

Byte #	Byte Contents	Reference
0,1	18,01	Track and sector number of first directory block.
2	65	ASCII code for the character A. This denotes 4040 disk format.
3	0	Null flag. Reserved for subsequent DOS use.
4-143		Bit map of available blocks in tracks 1-35 (1 = block is available; 0 = not available). Each bit denotes 1 block

Disk Directory

The disk directory is a list of all program and data files stored on a diskette. The disk directory is also located in the middle of the diskette next to the BAM.

There is room for 144 file entries in the directory. Information stored in the disk directory includes filename, file type, a list of the blocks used, and the beginning block. The directory is updated whenever a program file is saved or a data file is opened for writing. Once the diskette has been inserted into the drive, the LOAD command can be used as follows to load a diskette's directory into RAM:

```
LOAD "$",8
```

The following messages will then appear on the display:

```
SEARCHING FOR  $
LOADING
READY
```

The directory is now present in the computer's memory. By entering the LIST command, the directory will be displayed on the screen. If the following command is entered,

OPEN 1,4:CMD 1:LIST:PRINT#1:CLOSE 1

the directory listing will be sent to the printer.

The directory header and directory format are detailed in Tables 6-4 and 6-5, respectively.

Table 6-4. Directory Header (Track 18, Sector 0)

Byte#	Byte Contents	Reference
144-161		Diskette Name.
162-163		Diskette ID.
164	160	Shifted Space.
165-166	50,56	ASCII code for 2A. This represents the DOS version and format type.
166-167	160	Shifted Spaces.
171-255	0	Unused.

Table 6-5. Directory Format (Track 18, Sector 1)

Byte#	Byte Contents	Reference
0,1		Track & sector number of subsequent directory block.
2-31		File entry 1
34-63		File entry 2
66-95		File entry 3
98-127		File entry 4
130-159		File entry 5
162-191		File entry 6
194-223		File entry 7
226-255		File entry 8

Format for Directory File Entries

Byte#	Byte Contents	Reference
0	128 + File Type	The file type (see below) is OR'ed with $80 to denote correctly closed files (0 = DELeted;1 = SEQuential; 2 = PROGram;3 = USER;4 = RELative).
1,2		Track & sector # of first data block.
3-18		Filename.
19,20		Used for relative files only. Track and sector number of first side sector block.
21		Used for relative file only. Record size.
26,27		Track and sector number of replacement file when OPEN@ is used.
28,29		Number of bytes in file: low-byte, high-byte format.

LOADING & SAVING PROGRAMS ON DISKETTE

Loading Packaged Programs

Many readers may only be interested in loading packaged programs from diskette. The procedure for doing so is simple. First of all, insert the diskette containing the program to be run in the drive. Next, enter the following command:

LOAD "PROGRAM NAME",8 ➚ *

The following message will be displayed on the screen:

```
SEARCHING FOR PROGRAM NAME
LOADING
READY
■
```

The Ready prompt will appear when the desired program has been loaded into RAM. Once the program has been loaded, it can be run by merely entering the RUN command.

Formatting a Diskette

We recommend that you actually practice using the commands to save and load programs as you read the following sections. To do so, you will need to prepare a blank floppy diskette for use in your 1541 drive.

* ➚ signifies pressing the Return key.

All blank diskettes must be **formatted** before they can be used by DOS. Formatting is a process where existing data is erased from the diskette surface and a pattern is then recorded on that surface which allows data to be written to or read from the diskette's surface.

The OPEN, PRINT, and NEW commands are used in DOS to format a diskette. OPEN and PRINT are used to set up a command channel between the computer and the disk drive. The NEW command actually formats the diskette.

The following commands could be used to format a blank diskette:

```
OPEN 15,8,15
PRINT#15,"NEW0:VIC,01"
```

In the preceding example, the disk being formatted was assigned the name VIC and assigned an ID code of 01. The ID code can be any 2 characters.

The disk name is placed in the directory and the ID code is placed both in the directory as well as on every block of the diskette. The disk name and ID code are checked by DOS. If an incorrect ID is encountered, an error situation will result. This helps prevent the user from inadvertently writing data to the wrong diskette.

Saving a Program File

The SAVE command is used to save a program onto diskette from RAM. SAVE is used with the following configuration with the 1541 disk drive:

SAVE *"filename"*, *device* [,*command*]

When SAVE is used with a disk file, the *filename* must be specified as well as the *device* number (8).

The *command* is optional. If a *command* of 1 is specified for the program when it is saved on diskette and a *command* of 1 is again specified when that program is loaded, the loading will occur at the same address from which the program was saved. In general, the command is used when saving a machine language program.

Suppose that the 1541 contained a blank formatted diskette and that the following program had been entered into RAM via the keyboard:

```
100 PRINT "1"
200 PRINT "2"
300 PRINT "3"
400 END
```

By issuing the following SAVE command,

```
SAVE "PROGRAMA",8
```

the preceding program would be saved on the diskette with the filename PROGRAMA.

Saving and Replacing a Program File

Suppose that you wished to make a change in a program file stored on diskette. One way of doing so would be to load the program in RAM, make the necessary changes, erase the old unchanged file, and save the new changed file on diskette.

These last two steps can be combined so that the old version of the file is erased and is replaced by the new version. This can be accomplished by prefixing the filename with the characters @0: in the SAVE command. These characters instruct DOS to replace any program file on the diskette that has the same filename as the new program file in RAM with that new program file.

For example, the following command,

```
SAVE "@0:PROGRAMA",8
```

would cause any existing program file on diskette with the filename PROGRAMA to be erased, after which the program file in RAM will be saved on diskette with the filename PROGRAMA.

The following series of keyboard entries will serve as an illustration of the use of SAVE with @ to both save and replace a file on diskette.

```
READY
100 PRINT "1"
200 PRINT "2"
300 PRINT "3"
SAVE "PROGRAMA",8

SAVING PROGRAMA
```

In the preceding entries, PROGRAMA was entered via the keyboard and then saved on diskette. The following will appear when PROGRAMA is loaded and listed to the screen.

```
READY
LOAD "PROGRAMA",8

SEARCHING FOR PROGRAMA
LOADING
READY
LIST

100 PRINT "1"
200 PRINT "2"
300 PRINT "3"
READY
```

Additional program lines can then be added to PROGRAMA. The new versions of PROGRAMA can be saved on disk and will replace the earlier version of PROGRAMA by executing the SAVE@ command as shown in the following example.

```
READY
400 PRINT "4"
500 PRINT "5"
SAVE "@0:PROGRAMA",8

SAVING @0:PROGRAMA
READY
```

We will then test our entry by loading and listing the new version of PROGRAMA as shown in the following.

```
READY
LOAD "PROGRAMA",8

SEARCHING FOR PROGRAMA
LOADING
READY
LIST

100 PRINT "1"
200 PRINT "2"
300 PRINT "3"
400 PRINT "4"
500 PRINT "5"
READY
```

Loading a Program File

You may have noticed our use of LOAD in the preceding section. The LOAD command is used to load a program file from diskette into RAM. LOAD is used with the following configuration with the 1541 disk drive:

LOAD *"filename", device* [*,command*]

LOAD is used in a different fashion with diskette files than with cassette files. First of all, the program's filename must be included when using LOAD with diskette files. With cassette files, the inclusion of the filename is optional.

Also, the disk unit's device number (8) must be included in the LOAD command. Since the cassette unit's device number (1) is the default value for the device number, it need not be included in the LOAD command. If a device

number is omitted in a LOAD command, the Commodore 64 will search for the program file on the cassette unit.

The final LOAD statement parameter, *command,* is optional. *command* determines the address in RAM where the program will be loaded. If *command* is omitted or a value of zero is specified, the program will be loaded at the beginning of the BASIC program memory area.

In certain situations, the user may not wish to load a program at the beginning address for BASIC program storage. By specifying the value 1 for *command,* the specified program will be loaded at the same address from which it was saved. A *command* value of 1 is generally used for saving machine language programs.

Verifying a Program File

The VERIFY command allows the user to compare the program currently stored in memory with the program on disk. VERIFY is generally used immediately after the SAVE command to be certain that the program was stored without any errors. VERIFY is used with the following configuration with disk files:

VERIFY *"filename", device*

When VERIFY is used with disk files, the device number for the disk drive (8) must be specified.

If the program in RAM does not match the specified diskette program file, the following message will be displayed:

?VERIFY ERROR

If the VERIFY ERROR is encountered, the user should try again to save the program on disk.

DOS COMMANDS

OPEN

OPEN is used in BASIC to create a data file. OPEN can also be used to set up a command channel between the computer and the disk drive. When used in this fashion, OPEN is used with the following configuration:

Configuration -- OPEN (for command channel)

OPEN *filenumber, devicenumber, channel, string*

The *filenumber* can be any number from 1 to 255. However, in practice, only numbers between 1 and 127 should be used. The *filenumber* will be used in the program to identify which file is to be accessed.

The default *devicenumber* for the disk is 8.

The *channel* refers to the channel which is used to communicate with the disk. Channels 0 through 15 are available. Channels 0 and 1 are reserved for the operating system for use with the LOAD and SAVE commands. Channel 15 is reserved for sending DOS commands, and is known as the **command channel.** Channels 2 through 14 can be used for sending or retrieving data from disk files.

If a *string* is specified, that data is sent to the *filenumber* indicated. The *string* parameter is often used to send a DOS command to the indicated channel.

Example -- OPEN (for command channel)

OPEN 15,8,15,"NEW0:ONE,01"

In the preceding example, OPEN is used to send the NEW command to the disk drive (8) via the command channel (15). The NEW command causes the diskette in drive 1 to be formatted and assigned the name, ONE, and the ID code, 01.

PRINT#

The PRINT# statement works with disk files in a manner very similar to the usage of PRINT to output data to the screen. The difference is that PRINT# outputs data to a filenumber previously opened with the OPEN statement.

When PRINT# is used with a data channel, the specified data is sent into a buffer in the disk drive after which it is transferred to diskette. When PRINT# is used with the command channel (15), it will transmit DOS commands to the disk drive.

Configuration

PRINT# *filenumber, string*

Example

```
OPEN 15,8,15
PRINT#15,"N0:TWO,02"
```

In the preceding example, PRINT# is used as the command channel to send the NEW command to the disk drive. The diskette will be formatted, assigned the name TWO, and the ID code, 02.

Be certain that the filenumber specified with PRINT# has previously been opened. If the filenumber is not open, an attempt to execute a PRINT# to that filenumber will result in the following error message:

?FILE NOT OPEN ERROR

NEW (DOS Command)

As discussed previously, the NEW command is used to format a diskette. NEW also assigns the diskette a name and ID code.

Configuration

PRINT#15,"NEW0:*name, identification*"

COPY (DOS Command)

COPY can be used to make a separate copy of a program or data file on the diskette in the same drive with a different filename. COPY can be used on dual drives (such as the 4040) to copy programs or data files to diskettes on different drives.

Configuration

PRINT#15,"COPY0:*newname* = 0:*oldname*"

This configuration illustrates the use of COPY to make a second copy of a program or data file on the diskette in the same drive. *oldname* signifies the name of the file being copies, while *newname* is the name of the copy. 0 stands for the number of the disk drive -- drive 0.

Example

PRINT#15,"COPY0:PROGRAMB = 0:PROGRAMA"

In the preceding example, a copy of PROGRAMA is created on the diskette in drive 0. That copy is named PROGRAMB.

RENAME (DOS Command)

The RENAME command allows the user to change the name of a disk file after it has been added to the disk directory.

Configuration*

PRINT#15,"RENAME0:*newname* = *oldname*"

---

* RENAME can be abbreviated as R.

Example

PRINT#15,"R0:ONE = NINE"

In the preceding example, the file named NINE will be renamed as ONE. A file must be opened before RENAME can be executed.

SCRATCH (DOS Command)

The SCRATCH command allows the user to erase files no longer needed from the diskette. A single file can be erased via SCRATCH, or several files can be erased at once by using filename match characters.

Configuration*

PRINT#15,"SCRATCH0:*filename*"

filename indicates the file or files to be erased.

Example

```
                       READY
                       LOAD "$",8      } load & list diskette directory
                       LIST
                       FIVE  \
                       TWO    } current diskette files
                       NINE   |
                       FOUR  /
                       OPEN 15,8,15      } Scratch all files with
                       PRINT#15,"S0:F*"  } names beginning with "F"
load & list directory { LOAD "$",8
                      { LIST
                       TWO  } new directory after execution
                       NINE } of SCRATCH command
                       READY
```

* SCRATCH can be abbreviated as S.

INITIALIZE (DOS Command)

The INITIALIZE command allows the user to recover from a DOS error condition. When INITIALIZE is executed, the disk drive will be returned to the same condition as when it was initially powered on.

Configuration*

PRINT#15,"INITIALIZE"

VALIDATE (DOS Command)

The VALIDATE command is used to reorganize the space on the diskette. After a number of SAVE and SCRATCH commands have been executed, small scattered blocks of unused disk space tend to accumulate between files. These blocks are generally too small to be used. VALIDATE reorganizes the diskette so that its space is fully utilized.

Configuration*

PRINT#15,"VALIDATE"

The VALIDATE command also collects all blocks used by data files that were opened but never closed, and makes this disk space available for storage. Once VALIDATE has been executed, these files can no longer be accessed.

One caution when using VALIDATE is never to use this command on a diskette where random files are being stored. When VALIDATE is executed, any blocks allocated to random files will be disallocated.

---

* INITIALIZE can be abbreviated as I.
  VALIDATE can be abbreviated as V.

CLOSE

Always be certain that you close files once access to them has been completed. When a file is closed, DOS will finalize that file's directory entry and will also allocate that file's diskette blocks in the BAM. If a file is not properly closed, its data may be erased.

Configuration

CLOSE *filenumber*

filenumber should be the same as that specified in the corresponding OPEN statement for that file.

One point of caution when using the disk drive is that if a BASIC program generates an error condition, all files will be closed in BASIC, but will not be closed in DOS on the disk drive.

If this occurs, be certain to reinitialize the disk drive so that all disk files will be closed. This can be accomplished with the following entry:

```
OPEN 15,8,15,"I"
```

Unless open disk files are closed, data can be lost.

Finally, be certain that the command (or error) channel (#15) is the final data channel to be closed. This should also be the first channel opened. All data files should be closed prior to the closing of the command (or error) channel.

If the command (or error) channel is closed while other files are open, these files will closed by DOS on the disk drive. However, BASIC will still regard these files as being open. This can cause programming problems.

Error Channel

When a DOS error occurs, the red light on the 1541 disk

drive blinks on and off. When this occurs, the error channel (#15) contains a description of the error condition.

In order to determine the cause of a DOS error, the four values can be read from the error channel with an INPUT# statement. The error channel must be opened for input before the INPUT# statement can be used.

The following BASIC program can be used to display the error messages.

```
100 OPEN 15,8,15
200 INPUT#15,A,B$,C,D
300 PRINT A,B$,C,D
```

When a DOS error occurs, RUN the preceding program to display the description of the error.

The first value that is read from the error channel is a number that corresponds to the type of error. The second value is a description of the error situation. The third and fourth values correspond to the track and sector where the error occured.

When no error has occured on the disk drive, the message, "NO ERROR" is returned in the error channel.

FILE ACCESS

File Access refers to the means by which information can be placed in or retrieved from a data storage medium such as a floppy diskette. The 1541 supports three types of access: sequential, random, and relative. Each of these will be discussed in detail in the following sections.

Sequential Access

In cases where data must be read or written in a consecutive order, the means of access is known as **sequential.** A cassette

recorder can only use sequential access. Sequential files can be used on the 1541, although other types of access are also allowed on this device.

When data is written to a sequential file on the diskette, it is transferred one byte at a time from RAM, through a buffer, onto the diskette. When data is read from a sequential file, the data is transferred one byte at a time from the diskette, through a buffer, into RAM.

Opening a Sequential File

The following format is used for opening a sequential file:

Configuration

OPEN *fileno., deviceno., channel,* "0:*name, type, direction*"

fileno is the number by which the file will later be referenced. Generally, the *deviceno.* (device number) for the 1541 will be 8. Any *channel* from 2 to 14 inclusive can be used. Many programmers find it convenient to specify identical file and channel numbers in order to prevent confusion.

The *name* is the filename assigned to the file. Filename match characters may not be used when a write file is being created.

The *type* may be any one of the following:

PRG	Program
SEQ	Sequential
USR	User
REL	Relative

The *type* may be specified as the first letter of any one of the preceding rather than all three letters.

The *direction* can be either READ or WRITE. Again *direction* may be abbreviated with just the first letter (R or W).

Examples -- OPEN With Sequential files

```
OPEN 5,8,5,"0:ADDR,S,W"
OPEN 6,8,6,"0:MAILIT,P,R"
OPEN 3,8,3,"@0:PARTS,S,W"
```

Notice the use of @ in the final example. Including an @ before the filename in an OPEN statement has the same effect as it did in a SAVE statement. In other words, the file already open will be replaced by the file with the same name specified after @.

PRINT# -- Sequential Files

The PRINT# statement functions exactly as does the PRINT statement, except that data is sent to the disk drive rather than to the screen. The same rules of formatting apply to PRINT# that apply to PRINT.

Configuration

PRINT#*filename, data* { , ; } ...

filename is the name under which the file was opened. *data* consists of variables and/or text data enclosed within quotes.

Individual data items in a PRINT# statement must be delimited with a comma, semicolon, or carriage return. Generally, the preferred delimiter is the carriage return or semicolon. The comma, when used as a delimiter, results in blank spaces (corresponding to the PRINT statement output positions) being output to the disk file. When the semicolon is used as a delimiter, extra blank spaces are not output.

Example

```
140 OPEN 5,8,5,"0:STUDENT,S,W"
150 A$ = "JOHN":B$ = "BILL"
160 PRINT#5,A$
170 PRINT#5,B$
180 CLOSE 5
190 OPEN 5,8,5,"0:STUDENT,S,R"
200 INPUT#5,Y$,Z$
210 PRINT Y$,Z$
220 CLOSE 5
RUN
JOHN       BILL
```

INPUT# -- Sequential Files

The INPUT# statement is used to read data in from a sequential file.

Configuration

INPUT# *file, variable, ...*

The data items in the file being read by INPUT# must be properly delimited. Otherwise, these data items will be read as one long string by INPUT#.

The preferred method of inserting a delimiter between data items is the use of the carriage return character (or CR). A CR is automatically output at the end of a PRINT# statement -- as long as it is not ended with a comma or semicolon. The inclusion of a comma or semicolon at the end of a PRINT# statement suppresses the generation of the CR code.

Therefore, one method of inserting the CR code as a delimiter between data items is to include only one value with each separate PRINT# statement. This concept is illustrated in the previous example.

A string variable can also be used to insert a CR between data items. The variable is assigned the value "CHR$(13)"*, and is inserted between the data items in a PRINT# statement.

An example of this concept is given below. Notice in line 160 that X$ is defined as CHR$(13), or the string representation of the ASCII code for the carriage return.

Example

```
140 OPEN 5,8,5,"STUDENT,S,W"
150 A$ = "JOHN":B$ = "BILL"
160 X$ = CHR$(13)
165 PRINT#5,A$,X$,B$
170 CLOSE 5
180 OPEN 5,8,5,"0:STUDENT,S,R"
190 INPUT#5,Y$,Z$
200 PRINT Y$,Z$
210 CLOSE 5
RUN
JOHN          BILL
```

The result would be the same for this program if line 160 was deleted and line 165 was edited as follows:

```
165 PRINT#5,A$ CHR$(13) B$
```

GET# -- Sequential Files

The GET# statement is used to retrieve one character of data from a disk file.

Configuration

GET# *filenumber, variable*

---

* 13 is the ASCII code for the CR.

Example*

```
500 OPEN 5,8,5,"STUDENT,S,R"
600 GET#5,A$,B$,C$,D$,E$,F$,G$,H$,I$
700 PRINT A$,B$,C$,D$,E$,F$,G$,H$,I$
800 CLOSE 5
RUN
J         O         H         N

          B         I         L
L
```

Notice from the example that each individual byte in the file was returned by GET# including the carriage return delimiter.

Random Files

Unlike sequential files, **random access files** allow any record to be accessed within that file without first searching all preceding records.

Commodore DOS allows for two different types of random access files; random files and relative files. We will discuss random files in this section and relative files in the next section.

DOS contains commands which can be used to read from or write to any track and sector on a diskette. Commands are also available for checking which sectors are available as well as to denote any sectors being used.

These commands are transmitted via the command channel (#15). They instruct the disk what to do with the data being transmitted. The data is transmitted via one of the open data channels.

* This example assumes that STUDENT still exists on your diskette (see previous example).

OPEN -- Random Files

When using random files, two channels must be open to the disk. One channel must have been opened for DOS commands, and the second for transmitting data.

The command channel is opened via the same manner as discussed earlier for sending DOS commands. The following OPEN statement could be used to open a command channel in conjunction with a random file.

OPEN 15,8,15

The data channel for random files is opened by specifying the # sign for the filename, or by specifying the # sign followed by a buffer number.

Configuration

OPEN *filenumber, deviceno., channel,"#"*
OPEN *filenumber, deviceno., channel, "#bufferno."*

Examples

OPEN 5,8,5,"#" ⟶ no buffer specified
OPEN 4,8,4,"#2" ⟶ buffer #2

BLOCK-WRITE

The BLOCK-WRITE command is used to send data from a buffer to a specified track and sector on the disk.

Configuration*

PRINT#*filenumber*,"BLOCK-WRITE:" *channel;driveno.; track;sector*

---

* BLOCK-WRITE can be abbreviated as B-W.

Example

```
 10 INPUT "TRACK";TRACK
 20 INPUT "SECTOR";SECTR
 30 OPEN 15,8,15
100 OPEN 1,8,4,"#"
110 INPUT X$
120 IF X$ = "-99.99" THEN 140
130 PRINT#1,X$:GOTO 110
140 PRINT#15,"B-W:"4;0;TRACK;SECTR
150 CLOSE 1:CLOSE 15
```

The preceding example contains a program that uses the BLOCK-WRITE DOS command. The INPUT statements at lines 10 and 20 are used to specify the track and sector on the disk that the data is written to.

The OPEN statement at line 30 opens the command channel to the disk drive. The OPEN statement at line 100 opens the buffer for the data.

Line 110 is an INPUT statement that accepts data from the keyboard. The data "-99.99" is a flag value that indicates the end of the data. When this value is entered, the IF, THEN statement at line 120 causes the program control to branch to line 140. If a valid data item is entered (except for -99.99), the PRINT# statement at line 130 outputs the data to the buffer. Program control then returns to line 110 where more data is accepted.

When the flag value is encountered, the data in the buffer is recorded on the disk. Line 140 contains the BLOCK-WRITE command. When the BLOCK-WRITE is complete, the buffer and the command channel are closed at line 150.

The following two sections include examples that can be used with this program to select an unused sector on the disk and recover the data from the file.

BLOCK-READ

The BLOCK-READ command can be used to move one block of data from the diskette to the channel specified. Once the BLOCK-READ operation has been completed, either INPUT# or GET# can be executed to read the data.

Configuration *

PRINT#*filenumber*, "BLOCK-READ:"*channel;drive; track;block*

Example

```
 10 INPUT "TRACK";TRACK
 20 INPUT "SECTOR";SECTR
200 OPEN 15,8,15
210 OPEN 1,8,4,"#"
220 PRINT#15,"B-R:"4;0;TRACK;SECTR
230 GET#1,X$
240 IF ST = 0 THEN PRINT X$;:GOTO 230
250 CLOSE 1:CLOSE 15
```

The preceding example contains a program that uses the BLOCK-READ DOS command. The INPUT statements at lines 10 and 20 are used to specify the track and sector on the disk that the data is read from.

The OPEN statement at line 200 opens the command channel to the disk drive. The OPEN statement at line 210 opens the buffer for the data.

Line 220 contains the BLOCK-READ command that causes the data in the specified sector on the disk to be read into the buffer. The GET# statement at line 230 takes one character from the buffer and assigns that character to the variable X$.

ST is a variable that is maintained by the computer. The value of ST depends on the last I/O operation that was performed.

---

* BLOCK-READ can be abbreviated as B-R.

Concerning the disk drive buffer, ST is set equal to zero when data is being read. The value of ST is changed to 64 if there is no more data available.

As a result, the IF, THEN statement at line 240 causes the data to be read until there is no more data available. Line 250 closes the buffer as well as the disk drive command channel.

BLOCK-ALLOCATE

When using random files, you must know which blocks are available for use and which are not. If an attempt is made to use a block that is not available, when the error channel is read, DOS will return error number 65 (NO BLOCK), and will set the track and sector numbers to the next track and sector which are available for use.

Before an attempt is made to write to a block with BLOCK-WRITE, the user should first try to allocate that block with the BLOCK-ALLOCATE command. If the block in question is unavailable, the user can determine the next available block by reading the error channel.

Configuration*

PRINT#*filenumber,* "BLOCK-ALLOCATE:"*drive;track; block*

Example

```
10 INPUT "TRACK";TRACK
20 INPUT "SECTOR";SECTR
30 OPEN 15,8,15
40 PRINT#15,"B-A:"0;TRACK;SECTR
50 INPUT#15,E,E$,T,S
60 IF E$ = "OK" THEN 90
70 PRINT "OCCUPIED"
80 TRACK = T:SECTR = S
90 PRINT "TRACK";TRACK,"SECTOR";SECTR
```

* BLOCK-ALLOCATE can be abbreviated as B-A

In the preceding example, lines 10 and 20 are used to input the values of the desired track and sector. The OPEN statement at line 30 is used to open the command channel for the disk drive.

Line 40 contains the BLOCK-ALLOCATE command. If the specified block is available, the error channel returns the message "OK". If the specified block is not available, the error channel returns the message "NO BLOCK" as well as the track and sector of the next available block.

As a result, the INPUT# statement at line 50 is used to read the error channel. Line 60 contains an IF, THEN statement that tests the value of the error message. If the error message is "OK", then the program control branches to line 90, where the values of TRACK and SECTR are printed.

If the specified sector is occupied, the error message is "NO BLOCK". Also, the variables T and S are assigned the number of the next available track and sector. When a specified track and sector are occupied, the conditional expression at line 60 is false, so the statement at line 70 is executed.

The message "OCCUPIED" is displayed to notify the operator that the selected position on the disk is not available. The assignment statements at line 80 are used to change the values of the variables TRACK and SECTR to the next available track and sector numbers. The PRINT statement at line 90 displays the new values of TRACK and SECTR.

BLOCK-FREE

The BLOCK-FREE command frees a block that is no longer needed by a random file. BLOCK-FREE acts in an exact opposite fashion to BLOCK-ALLOCATE. BLOCK-FREE does not in itself erase data. It merely frees that block's entry in the BAM.

Configuration*

PRINT#*filenumber,* "BLOCK-FREE:"*drive;track;sector*

Example

100 PRINT#15,"B-F:"0;T;S

The track and sector number whose entries are to be freed in the BAM are specified by the variables T and S in the preceding example.

BUFFER-POINTER

The BUFFER POINTER command can be used to move the **buffer pointer** within the disk drive's buffer.

The buffer pointer indicates the position within the buffer where the last data item was written, as well as the location where the next data item will be read. By altering the position of the buffer pointer within the buffer, random access can be gained to the individual bytes within a block of data. This effectively allows the user to segregate data blocks into individual records.

Configuration*

PRINT#*filenumber,*"B-P:"*channel;location*

channel refers to the channel number. *location* refers to the position of the byte within the buffer to which the buffer pointer is to be moved.

---

* BLOCK-FREE can be abbreviated as B-F.
BUFFER POINTER can be abbreviated as B-P.

Example

```
10 INPUT "TRACK";TRACK
20 INPUT "SECTOR";SECTR
30 OPEN 15,8,15
40 OPEN 1,8,4,"#"
50 PRINT#15,"B-R:"4;0;TRACK;SECTR
60 INPUT "BYTE NUMBER";N
70 PRINT#15,"B-P:"4;N
80 GET#1,X$
90 PRINT X$
100 CLOSE 1:CLOSE 15
```

The preceding example contains an example of the BUFFER POINTER command. Lines 10 and 20 are used to input the track and sector number of the desired file. The OPEN statement at line 30 opens the error channel to the disk. At line 40, the buffer channel is opened.

The BLOCK-READ command is used at line 50 to write the contents of the sector in the buffer file. The INPUT statement at line 60 is used to assign the desired byte number to the variable N. At line 70, the pointer is moved to the byte specified by the variable N. For example, if N is set equal to 18, the pointer will be moved to the eighteenth byte in that file.

The GET# statement at line 80 assigns the character in the specified buffer location to the variable X$. At line 90, the value of X$ is displayed on the screen. At line 100, the buffer channel and the command channel are closed.

USER1

USER1 is a special version of the BLOCK-READ command. USER1 is generally used with machine language programs.

BLOCK-READ reads characters from the diskette into the specified channel until the buffer pointer stored with that

block indicates that the block has been completed. USER1 alters the buffer pointer to 255 prior to the read operation so that the entire block of data is read from the diskette.

Configuration*

PRINT#*filenumber,*"U1:"*channel,drive,track,block*

USER2

USER2 is a special version of the BLOCK-WRITE command. USER2 is generally used with machine language programs.

BLOCK-WRITE generally writes the buffer's contents to the specified block on the diskette along with the value of the buffer pointer. USER2 writes the contents of the buffer without altering the buffer pointer value already stored on the specified block.

Configuration*

PRINT#*filenumber,*"U2:"*channel,drive,track,block*

Relative Files

Relative files are much more convenient for handling data then random files, as files may be subdivided into records as well as fields. Relative files can be randomly accessed.

With relative files, the DOS inventories the tracks and sectors in use. Also, a single record can run from one block into the next. This is possible because DOS uses **side sectors** to point to the beginning of each record.

---

* UA can be substituted for U1
  UB can be substituted for U2.

Each side sector can point to as many as 120 records. A file can include as many as six side sectors. Therefore, a relative file can contain as many as 720 records. Since each record can consist of a block of data, a relative file can fill an entire diskette.

The format for a relative file is outlined in Table 6-6.

Table 6-6. Relative File Format

Data Block Format

Byte#	Reference
0,1	Next Data Block's Track and Sector Number.
2-256	254 available data bytes. Records partly filled with data are padded with nulls (00). Empty records contain FF in the first byte followed by 00 to the end of the record.

Side Sector Block

Byte#	Reference
0,1	Next data block's track and sector number.
2	Side sector number (0-5).
3	Record length.
4,5	Track and sector of first side sector (0).
6,7	Track and sector of second side sector (1).
8,9	Track and sector of third side sector (2).
10,11	Track and sector of fourth side sector (3).
12,13	Track and sector of fifth side sector (4).
14,15	Track and sector of sixth side sector (5).
16-256	Track and sector pointers to 120 data blocks.

Creating a Relative File

The OPEN statement is used as shown below to create a relative file.

Configuration

OPEN *filenumber, deviceno., channel, "name,* L," + CHR$(*record length*)

Example -- Relative File Creation

OPEN 3,8,3,"TEXTA,L," + CHR$(254)

When an OPEN statement is executed to create a relative file, DOS will initially check to determine whether or not that file exists. If it does, the OPEN statement will have no effect.

Unlike sequential files, the replace option (@) cannot be used to erase and replace an existing relative file. That file must be erased with the SCRATCH command.

Opening an Existing Relative File

Once a relative file has been created, the following format can be used to open that file.

Configuration -- Opening Existing Relative File

OPEN *filenumber, deviceno., channel, "filename"*

Example -- Opening Existing Relative File

OPEN 3,8,3,"TEXTA"

When the preceding configuration is used for opening relative files, DOS will open the relative file in question.

Positioning the File Pointer in a Relative File

Before reading from or writing to a relative file, the relative file's **file pointer** must be positioned correctly. The file pointer indicates the next record to be accessed in the relative file.

Configurations

PRINT#*filenumber*, "P" CHR$(*channel*) CHR$(*lowrec*) CHR$(*highrec*)

PRINT#*filenumber*, "P" CHR$(*channel*) CHR$(*lowrec*) CHR$(*highrec*) CHR$(*position*)

In the preceding configurations, 2 bytes are necessary to store the file pointer record number. Since one byte can only store a maximum of 256 numbers, and since as many as 720 records can be stored in a relative file, two bytes are needed to store the file pointer record pointer.

lowrec contains the least significant portion of the record number. *highrec* contains the most significant portion of the record number. The following formula can be used to translate *lowrec* and *highrec* to the file pointer record number.

Record Number = *highrec* * 256 + *lowrec*

position is an optional parameter which indicates the position within the record. Generally, a value of 1 is specified.

The *filenumber* specified must be that opened for the command channel.

Example -- Writing to a Relative File

```
10 OPEN 15,8,15
20 OPEN 2,8,2,"TEXTA,L," + CHR$(100)
30 PRINT#15,"P" CHR$(2)CHR$(1)CHR$(0)CHR$(1)
40 INPUT "NAME";A$
50 IF A$ = "Q" THEN 80
60 INPUT "ADDRESS";B$
70 PRINT#2,A$,B$:GOTO 40
80 CLOSE 2:CLOSE 15
```

The preceding example contains a program that writes to a relative file. The OPEN statement at line 10 opens the command channel for the disk drive. The OPEN statement at line 20 creates a relative file with 100 characters per record.

Line 30 specifies the position of the pointer. In this statement, CHR$(2) specifies the relative file created at line 20. The next two characters, CHR$(1)CHR$(0) specify the record number within the file. These values correspond to record number 1. The last character, CHR$(1), specifies the position within the record.

At lines 40 and 60, data is accepted for the variables A$ and B$. The variable A$ is set equal to "Q" to indicate the end of the data.

Line 70 outputs the values of A$ and B$ to the relative file. Also, program control returns to line 40 where more input is accepted. The record number is automatically incremented.

When the flag value (Q) is input for A$, the program control is branched to line 80 where the I/O channel and the command channel are closed.

Example -- Reading From a Relative File

```
10 OPEN 15,8,15
20 OPEN 1,8,3,"0:TEXTA,L," + CHR$(100)
30 INPUT "RECORD NUMBER";X
40 PRINT#15,"P" CHR$(3)CHR$(X)CHR$(0)CHR$(1)
50 INPUT#1,A$
60 PRINT A$
70 IF ST = 0 THEN 50
80 CLOSE 1:CLOSE 15
```

The preceding example contains a program that reads data from a relative file. This example is designed to recover the data from the relative file TEXTA. Data can be written to this file with the last example program (Writing to a Relative File).

The OPEN statement at line 10 opens the command channel for the disk drive. The OPEN statement at line 20 initiates an I/O channel for the relative file.

The INPUT statement at line 30 is used to assign the desired record number to the variable X. Line 40 moves the pointer to the specified record number. The first character, CHR$(3), specifies the relative file. The second two characters, CHR$(X)CHR$(0), move the pointer to the record number specified by the value of the variable X. The last character, CHR$(1) specifies the first position in the record.

The INPUT# statement at line 50 assigns a value from the record to the variable A$. At line 60, the value of A$ is displayed on the screen.

ST is a variable that is automatically updated by the computer. The value of ST is determined by the last I/O operation performed. When the last item of the record is read, the value of ST changes from 0 to 64. The variable ST is convenient to use when an unknown number of data items are included in a record. If the number of data items is known, it is not necessary to use the ST variable.

The data is read and displayed until the value of ST does not equal 0. When all the data is read, the I/O channel and the command channel are closed.

PROGRAMMING THE DISK CONTROLLER

The following commands allow the programmer to design routines that operate directly on the disk controller.

BLOCK-EXECUTE

This command is used to load a block containing a machine language subroutine from the diskette. Execution will commence at address 0 in the buffer and will continue until a RTS instruction has been executed.

Configuration*

PRINT#*filenumber,*"BLOCK-EXECUTE:"*channel;drive; track;block*

MEMORY-READ

The disk drive contains 16K of ROM and 2K of RAM (used for the disk drive buffers). MEMORY-READ (as well as the other MEMORY disk controller commands) allows the reader to access the disk drive ROM and RAM.

Configuration

PRINT#*filenumber,*"M-R:"CHR$(*low*)CHR$(*high*)

low and *high* are the low and high bytes of the memory address whose contents are to be read. The next byte read using GET# through channel 15 will be from the address specified by MEMORY-READ in the disk controller's mem-

---

* BLOCK-EXECUTE can be abbreviated as B-E.

ory. Any successive GET# statements specifying channel 15 will read the next successive bytes in the disk controller's memory. GET# should be used rather than INPUT# to read these memory addresses.

Example

```
100 OPEN 15,8,15
200 INPUT "ADDRESS";X
300 X1 = INT(X/256):X2 = X - X1 * 256
400 PRINT#15,"M-R:" CHR$(X2)CHR$(X1)
500 GET#15,Z$
600 PRINT ASC(Z$ + CHR$(0))
700 INPUT "ENTER N TO END";A$
800 IF A$<>"N" THEN 200
900 END
```

MEMORY-WRITE

The MEMORY-WRITE command allows up to 34 bytes to be written to the disk controller's memory.

Configuration

PRINT#*filenumber,*"M-W:" CHR$(*low*)CHR$(*high*);
#*characters, byte data*

filenumber will be the command or error channel (15). *low* gives the low byte of the address. *high* gives the high byte of the address. #*characters* specifies the number of characters to be written. *byte data* specifies the data to be written.

Example

```
PRINT#15 "M-W:"CHR$(4)CHR$(4);1;CHR$(99)
```

MEMORY-EXECUTE

Any routine stored in the disk drive memory, ROM or RAM, can be executed via the MEMORY-EXECUTE command.

Configuration

PRINT#*filenumber,*"M-E:"CHR$(*low*)CHR$(*high*)

filenumber should be the filenumber specified for the command channel. *low* gives the low byte of the memory address where execution is to begin. *high* gives the high byte of that address.

Example

100 PRINT#15,"M-E:"CHR$(0)CHR$(20)

USER Commands

In addition to the USER1 and USER2 commands discussed previously, the USER commands can be used as jumps to a table of locations in the disk drive's RAM memory. These locations are summarized in Table 6-7.

Configuration

PRINT#*filenumber, "user command"*

Example

PRINT#15,"U3"

Table 6-7. USER Command Jump Table

USER Command	Reference
U1 or UA	BLOCK-READ without changing buffer pointer.
U2 or UB	BLOCK-WRITE without changing buffer pointer.
U3 or UC	Jump to 0500H
U4 or UD	Jump to 0503H
U5 or UE	Jump to 0506H
U6 or UF	Jump to 0509H
U7 or UG	Jump to 050CH
U8 or UH	Jump to 050FH
U9 or UI	Jump to FFAH
U; or UJ	Power-up vector
UI+	Set Commodore 64 speed
UI–	Set VIC-20 speed

CHAPTER 7.
COMMODORE 64 PRINTER INSTALLATION & OPERATION

INTRODUCTION

This chapter provides installation and operating instructions for the 1525 printer. The 1525 is virtually identical to the 1515. The major difference lies in the fact that the 1515 can only be used with the VIC-20, while the 1525 can also be used with the Commodore 64.

The 1525 is capable of many output functions including upper and lower case letters, numbers, and graphics characters. The 1525 printer can also output double-width and reverse characters, as well as custom design graphics characters.

Installation

The printer should be placed on a level surface where it will not be exposed to direct sunlight. The printer should also be protected from excessive dust or humidity. Extreme hot or cold temperatures should be avoided, as well as quick changes in temperature.

Step 1.

Be sure that the printer and the computer are turned off, and insert the printer's plug into a standard wall outlet. Do not turn on the printer or the computer until installation is complete.

Step 2.

Lift the clear plastic dust cover and note the black tube that protects the carriage in shipment. Remove the tube by pulling the tag that is attached at the end.

Step 3.

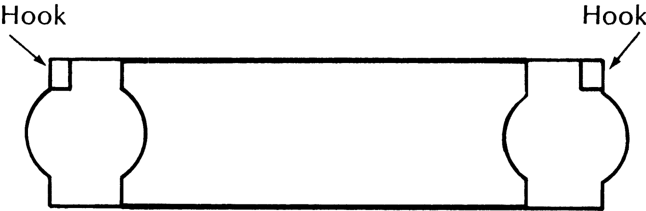
Connect the printer to the computer with the cable that is provided with the printer. The cable has a 6-pin connector at each end. One end attaches to the 6-pin, round recepticle at the back of the printer. The other end attaches to a 6-pin recepticle on the computer console (or disk drive). Be sure to line up the plug with the recepticle before inserting. The plug and recepticle can both be damaged if the plug is forced in. When the plug and recepticle are aligned correctly, the plug can be inserted smoothly and snugly.

Step 4.

The ribbon for the printer is also provided separately. Remove the ribbon cartridges from the package, and remove the clear plastic cover of the printer. The two cartridges for the ribbon are attached to the two metal brackets at both sides of the carriage at the front of the printer (See Illustration 7-2).

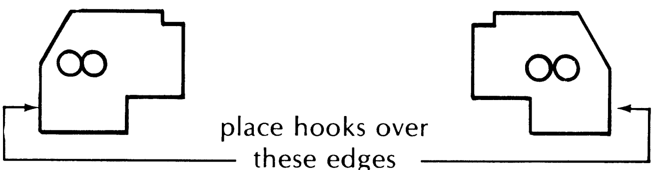
Make note of the hooks on the ribbon cartridges (Illustration 7-1).

Illustration 7-1. Rectangular Hooks on Ribbon Cartridges



The rectangular hooks should be on the front, underside of the cartridges when they are installed. The metal brackets for the ribbon cartridges have a straight edge that accomodates the rectangular hooks on the ribbon cartridges (Illustration 7-2).

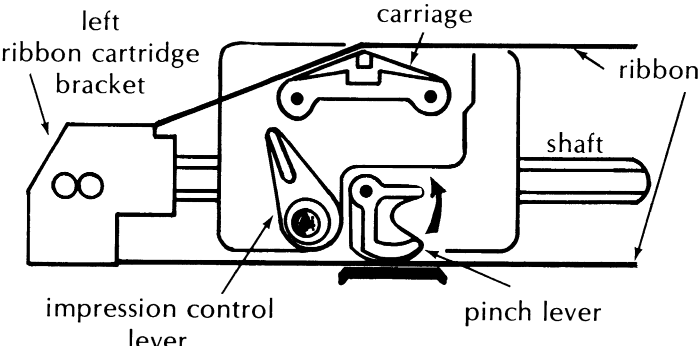
Illustration 7-2. Brackets for Ribbon Cartridges



Place the right ribbon cartridge on the ribbon cartridge bracket by placing the hook on the underside of the cartridge over the straight, outside edge of the bracket. The ribbon cartridge does not snap into place, but it should be secure in its place.

The ribbon must also be properly positioned on the carriage. The carriage is located directly in front of the roller on the left side. Do not attempt to move the carriage manually.

Illustration 7-3. Ribbon Installation on Carriage



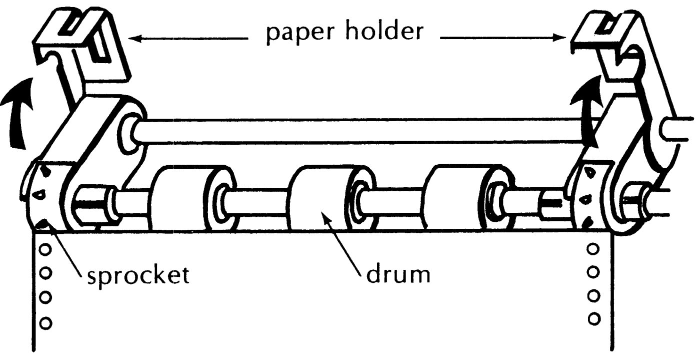
Place the ribbon between the carriage and the roller. Be careful not to twist the ribbon. Place the ribbon behind the pinch lever by moving the pinch lever foreward as indicated in Illustration 7-3.

When the ribbon is properly installed on the carriage, place the left ribbon cartridge on the ribbon cartridge bracket. The left cartridge is installed in the same manner as the right cartridge.

Step 5.

Lift the paper holders from the sprocket units (Illustration 7-4).

Illustration 7-4. Paper Holders on the Sprocket Units



The sprocket unit on the left is fixed, but the right sprocket can be moved to accomodate paper from 4½ inches to 10 inches wide. The drums (Illustration 7-4) should be evenly spaced between the sprocket units.

Insert the paper from the upper rear side of the printer, under the clear plastic guard. When the paper emerges from between the roller and carriage, pull 4 or 5 inches of the paper through the printer. Position the sprocket units so the paper holes fit on the sprockets. The paper should be as tight

as possible without ripping the holes. When both sides of the paper have been fitted over the sprockets, place the paper holders over the sprockets until they snap into place. The paper advance wheel on the right side of the printer can be used as an aid in inserting the paper, but it can only move the paper forward.

Step 6.

The mode selection switch is located near the cable recepticle on the back of the computer. The switch should be in the position labeled 4 if only one printer is being used. If two printers are being used, the switch should be in the position labeled 5. The printer is now ready to operate. Turn on the power switch on the left side of the back of the printer.

When the printer is turned on, the power indicator on the front of the printer should glow. Also, the carriage should move to the center, then return to the left margin. If a problem develops, repeat the installation steps carefully.

The printer has a test procedure that can be selected with the mode switch, near the cable recepticle. When the switch is set to the T position, the printer begins printing the entire character set. The test procedure ends when the switch is set to position 4 or 5. The test ends also if the printer is turned off.

The results of the test should be carefully examined. If the print is too light or too dark, the print head pressure can be adjusted with the impression control lever on the carriage (Illustration 7-3). If the print is too dark, move the lever counterclockwise. If the print is too light, move the lever clockwise. The lever should always be positioned in one of the holes.

When the print head pressure is adjusted, the printer is ready to be used with the computer.

When the computer console is turned on, the carriage on the printer should move to the center, then return to the left margin.

BASIC STATEMENTS

The printer can be manipulated with several BASIC commands. The PRINT# and CMD statements are used along with OPEN and CLOSE.

OPEN

An OPEN statement must be executed before any output can be sent to the printer. The configuration of an OPEN statement is as follows:

OPEN *file number, device number, mode, string*

The *filenumber* can be any number from 1 to 255. The value of the *filenumber* is truncated, and used to designate the connection between the printer and the computer. Any subsequent PRINT#, CMD, or CLOSE statements must have a corresponding *filenumber* to specify the printer.

The *device number* corresponds to the position of the test mode switch on the back of the printer. Generally, the *device number* is 4, but the switch may also be set at position number 5. In this case, the *device number* should be 5.

The *mode* argument of an OPEN statement can be either 0 or 7. Mode 0 corresponds to the upper case letters, and graphics characters. Mode 7 corresponds to the lower case and upper case letters. The *mode* argument is optional. If the *mode* is not specified, mode 0 will be assumed.

The *string* argument is also optional. A string value that is included in an OPEN statement is printed when the first PRINT# statement is executed.

Example

```
10 OPEN 1,4
20 PRINT#1,"ABC"
30 PRINT#1,"♠|—"
40 CLOSE 1
50 OPEN 1,4,7
60 PRINT#1,"ABC"
70 PRINT#1,"♠|—"
80 CLOSE 1
```

When the preceding example is executed, the following output appears on the printer:

```
ABC
♠|—
abc
ABC
```

In the preceding example, the OPEN statement at line 10 designates file number 1 for the printer. The PRINT# statements at lines 20 and 30 output the characters ABC and ♠|— to the printer. Notice that the graphics characters correspond to shift-A, shift-B, and shift-C. Since the OPEN statement does not include a mode number, the printer is in mode 0. As a result, the upper case letters and graphics characters are printed.

At line 40, the file is closed. At line 50, the file is opened again, in print mode 7. As a result, the lower and upper case letters are printed instead of the upper case and graphics characters.

CLOSE

The CLOSE statement is used to eliminate the connection between the computer and printer. Since the computer can only accomodate 10 open files at any time, the printer file may need to be closed while the pinter is not in use. It is generally good practice to close a file when it is not in use. If the program ends when a file is open, some of the data may

be lost. Be sure to close all the open files before the program ends.

The file number in a CLOSE statement must correspond to the file number in a previously executed OPEN statement. The correct use of the CLOSE statement is illustrated in the preceding example.

CMD

The CMD statement is used to designate the printer as the general output device. After a CMD statement has been executed, any subsequent LIST or PRINT statements will cause the output to be sent to the printer instead of to the display.

The argument of a CMD statement must correspond to the file number of an open file.

To return to normal output operation, a PRINT# statement must be executed just prior to the CLOSE statement. The following example demonstrates the use of a CMD statement.

Example

```
10 OPEN 1,4
20 CMD 1
30 FOR T = 1 TO 10
40 PRINT T;
50 NEXT
60 PRINT#1:CLOSE 1
```

In the preceding example, file number 1 is opened for the printer at line 10. At line 20, a CMD statement is executed for file number 1. Lines 30 through 50 consist of a FOR, NEXT loop that includes a PRINT statement. Since a CMD statement was executed, the PRINT statement at line 40 causes the output to be sent to the printer. As a result, the following output is sent to the printer.

1 2 3 4 5 6 7 8 9 10

At line 60, the PRINT# statement is necessary to close the file to the printer, in addition to the standard CLOSE statement.

When a CMD statement is executed, the output occurs on the printer in the same format as the output would occur on the screen.

PRINT#

The PRINT# statement is used to send output to the printer in the same manner that a PRINT statement is used to send output to the screen. The first argument of a PRINT# statement must correspond to the file number that is open for the printer. The data to be output must be separated from the file number by a comma.

Numeric variables, string variables, numeric constants, or string constants can be included in PRINT# statements.

Example

```
110 OPEN 1,4
120 PRINT#1,27,X,"JONES",A$
130 CLOSE 1
```

The previous example contains a section of a program that includes a PRINT# statement. The OPEN, PRINT#, and CLOSE statements all have the same file number. The PRINT# statement includes a numeric constant (27), a numeric variable (X), a string constant (JONES), and a string variable (A$). When the PRINT# statement is executed, the constants are printed, as well as the values of the specified variables.

In a PRINT# statement, each data item must be separated by a comma or a semicolon. When the data is separated by a commas, ten blank spaces are inserted between the data items. However, numeric values are always printed with a

blank space immediately before and after the value. As a result, numeric values which are separated with a comma in a PRINT# statement, will have twelve blank spaces between them. Similarly, a string value and a numeric value are separated by eleven spaces.

When a semicolon is used to separate data items in a PRINT# statement, no additional spaces are inserted in the output. Since numeric values include a preceding and following space, numeric values, separated by a semicolon, are printed with two blank spaces between them. String values are separated from numeric values by only one space.

Example

```
10 OPEN 1,4
20 PI = 3.1415927
30 E = 2.7182818
40 PRINT#1,"PI EQUALS:";PI,
50 PRINT#1,"E EQUALS:";E
60 CLOSE 1
```

The preceding example contains a program that uses PRINT# statements. At line 10, a file is opened for output to the printer. At lines 20 and 30, the numeric variables E and PI are assigned values.

Line 40 contains a PRINT# statement that outputs the string value PI EQUALS:, as well as the value of the numeric variable PI. Since the data items are separated by a semicolon, no additional spaces are inserted in the output. However, one blank space separated the output because numeric values are always preceded by one space. Since line 40 ends with a comma, the next output occurs on the same line of output, separated by 10 additional spaces.

At line 50, the first data item that is output is the string value E EQUALS:. Since the previous output was a numeric value, eleven spaces separate the output. One space follows the numeric value, and ten spaces are inserted because of the comma. The last two items of output are separated by only one space.

The output of this sample program is as follows:

PI EQUALS: 3.1415927 E EQUALS: 2.7182818

A comma or semicolon that is used at the end of a PRINT# statement prevents the printer from proceeding to the next line.

CONTROL CODES

The 1525 printer can perform many special functions as specified by the **control codes.** The control codes are special characters that are generated with the CHR$ function. These characters are not visible, but they represent the printer functions.

The control codes are summarized in Table 7-1.

Table 7-1. Control Codes

Function	Character Number
GRAPHICS	8
LINE FEED	10
RETURN	13
WIDE	14
STANDARD	15
TAB	16
CURSOR DOWN	17
REVERSE	18
REPEAT	26
DOT ADDRESS	27
CURSOR UP	145
REVERSE OFF	146

Control characters are sent to the printer with a PRINT# statement. Generally, the control characters are specified with the CHR$ function. The characters can appear in a PRINT# statement, or can be assigned to a string variable.

If several control characters are assigned to a string variable, they must be "added" together. As a result, characters in an

assignment statement must have a plus sign (+) between each character.

Several control characters can appear in a PRINT# statement without any punctuation between them.

Examples

```
A$ = CHR$(17)
A$ = CHR$(27) + CHR$(16) + CHR$(1) + CHR$(2)
PRINT#1,CHR$(17);"OUTPUT"
PRINT#1,CHR$(27)CHR$(16)CHR$(1)CHR$(2)
```

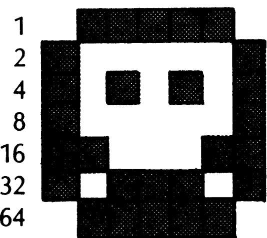
Control characters can only be sent to the printer with a PRINT# statement. As a result, when control characters are assigned to a string variable, the string variable must appear in a subsequent PRINT# statement.

Graphics -- CHR$(8)

The graphics mode allows the user to design a special character for the printer. The character can consist of any combination of the 49 dots on a 7 by 7 matrix. The dot pattern can be established by the following procedure.

Draw an array of 7 rows and 7 columns. Fill in the desired dot pattern, and label the rows as in Illustration 7-5.

Illustration 7-5. Dot Pattern for Graphics Mode



When the desired dot pattern has been determined, find the sum of the row numbers that are filled in for each column. For example, in Illustration 7-5, the first column of the pattern has a dot in row numbers 2, 4, 8, 16, and 32. Put the sums of each column in a DATA statement. For Illustration 7-5, the DATA statement includes the values 62, 81, 101, 97, 101, 81, and 62.

Each of these values must be added to 128, and converted to a character with the CHR$ function. Each of these characters must then be assigned to a string variable. This can be done with the following example program.

Example

```
10 OPEN 1,4
20 FOR T = 1 TO 7:READ X
30 X$ = X$ + CHR$(X + 128)
40 NEXT
50 PRINT#1,CHR$(8)X$
60 CLOSE 1
70 END
80 DATA 62,81,101,97,101,81,62
```

In the preceding example, file number 1 is opened for the printer at line 10. Line 20 originates a loop that reads each data value, adds it to 128, converts it to a character, and assigns it to a string variable. At line 50, the PRINT# statement indicates the graphics mode, CHR$(8), as well as the string variable that contains the data.

In the graphics mode, the printer outputs 9 lines per inch instead of 6 lines per inch. As a result, the lines of output are closer together in the graphics mode.

Line Feed -- CHR$(10)

The line feed control character causes the printer to move down one line on the page.

Carriage Return -- CHR$(13)

The carriage return control character causes the printer to move to the beginning of the next line.

Wide -- CHR$(14)

The wide control character causes subsequent output to be printed with twice its normal width. To return to normal width characters, print the standard control code CHR$(15) character.

Standard -- CHR$(15)

The standard control character is used to return to the standard mode of output. Standard output is the upper case and graphics character mode. Characters are output at normal width.

Tab -- CHR$(16)

The tab control character is used to select the position of output on the printer. When the CHR$(16) statement is followed by a number (in quotation marks), the next output to the printer begins in the specified column.

The tab control code is used to select the position of output on the printer. When the CHR$(16) statement is followed by a number (in quotation marks), the next output to the printer begins in the specified column.

Example

```
10 OPEN 1,4
20 PRINT#1,CHR$(16)"03OUTPUT"
30 CLOSE 1
40 END
```

In the preceding example, file number 1 is used for output to the printer. At line 20, the program includes a PRINT# statement with the tab control code. The two characters that

follow CHR$(16) determine the starting column of the output. Since the first two characters of the string are 03, the output begins in column number 3. The characters "03" do not appear in the output. Only the word OUTPUT is printed.

The two digit number does not necessarily need to be part of an output string.

Example

```
10 OPEN 1,4
20 A$ = "OUTPUT"
30 PRINT#1,CHR$(16)"03";A$
40 CLOSE 1
50 END
```

In the preceding example, the value of the variable A$ is printed three columns from the left margin. The results of the two examples of the tab control code are identical.

The control character CHR$(16) can be assigned to a string variable if it is used often in a program. The next example program uses this technique to obtain the same results as the previous two examples.

Example

```
10 OPEN 1,4
20 TAB$ = CHR$(16)
30 PRINT#1,TAB$"03";"OUTPUT"
40 CLOSE 1
50 END
```

In the above example, the string variable TAB$ is used to replace the CHR$(16) statement.

The tab control code can be used several times in the same PRINT# statement. However, the print head will only move forward. As a result, the column numbers in multiple tab control code statements must be in increasing order. For example, if the printer has moved past a particular column, it

will not move backward if a tab control code is executed for that column.

Example

```
10 OPEN 1,4
20 TAB$ = CHR$(16)
30 PRINT#1,TAB$"03";"A";TAB$"20";"B"
40 CLOSE 1
50 END
```

In this example, the letter A is printed in column number 3. The letter B is printed in column 20. Notice that the leftmost tab position is specified first.

If the printer cannot comply with a tab control code, the output occurs at the next available column.

Cursor Down -- CHR$(17)

The cursor down control code causes the printer to display the upper and lower case characters instead of the upper case and graphics characters. The cursor down mode is equivalent to an OPEN statement with the mode value of 7.

Example

```
10 OPEN 1,4
20 PRINT#1,"ABCDE"
30 PRINT#1,CHR$(17);"ABCDE"
40 CLOSE 1
50 END
```

In the preceding example, line 20 includes a PRINT# statement with a string argument. Since the printer defaults to the cursor up mode, the upper case letters are printed. At line 30, the cursor down function is specified in the PRINT# statement. As a result, the output is printed as lower case letters. The following output results from the previous program:

ABCDE
abcde

The CHR$(15) function is used to return to the standard mode of output. The CHR$(15) function can also be used to return to the cursor up mode.

Reverse -- CHR$(18)

In the reverse mode, the characters are displayed against a dark colored background. After a CHR$(18) is printed, any subsequent ouput appears in the reverse mode until a carriage return is executed. The reverse mode can be used in several PRINT# statements, without repeating the control character, as long as each PRINT# statement ends with a comma or semicolon, and does not include a carriage return.

Example

```
10 OPEN 1,4
20 PRINT#1,CHR$(18);"OUTPUT1";
30 PRINT#1,"OUTPUT2"
40 PRINT#1,"OUTPUT3"
50 CLOSE 1
60 END
```

In the preceding example, the PRINT# statement at line 20 invokes the reverse mode. As a result, the message OUTPUT1 appears in reverse mode. Since the first PRINT# statement ends with a semicolon, the message OUTPUT2 is printed adjacent to OUTPUT1, in reverse mode also. The PRINT# statement in line 30 has no punctuation at its end, so a carriage return is implied. As a result, the message OUTPUT3 is printed on the next line, in the standard mode.

Repeat -- CHR$(26)

The repeat control code is used to print a specified number of repetitions of a dot pattern. The repeat mode can only be used in the graphics mode. As a result, the CHR$(8) and

CHR$(26) control characters must always be executed together.

The dot pattern is determined in the same manner as the graphics mode, but only one column can be specified. For example, to repeat a three stripe pattern (Illustration 7-6), add the sum of the desired dot positions to 128.

Illustration 7-6. Three Stripe Pattern for Repeat Mode

1	●
2	○
4	○
8	●
16	○
32	○
64	●

The sum of the desired pattern elements is 1 + 8 + 65 = 73. Therefore, the data for the repeated character is 73 + 128 = 201.

In general, a PRINT# statement for a repeated dot pattern must have four characters. The first character specifies the graphics mode. The second character specifies the repeat mode. The third character represents the number of repetitions, and the fourth specifies the dot pattern.

Example

```
 5 A$ = CHR$(8) + CHR$(26) + CHR$(9) + CHR$(201)
10 OPEN 1,4
15 PRINT#1,A$
20 CLOSE 1
25 END
```

In this example, the string variable A$ is assigned 4 characters in line 5. These are the four characters that will be used in the repetitive PRINT# statement.

The first character of A$ specifies the graphics mode. The second character specifies the repeat mode. The third character indicates that the dot pattern is repeated 9 times. The fourth character specifies the data for the dot pattern (Illustration 7-6).

When the previous example program is executed, the three stripe pattern is printed in the first 9 columns.

Dot Address -- CHR$(27) + CHR$(16)

The dot address control code is used to position the print head at a particular location. The dot address control code is similar to the tab control code except that the dot address code divides one line of output into 480 positions instead of 80. The dot address mode is more difficult to use, but allows more flexibility.

The positioning of the print head requires 4 characters in a PRINT# statement. The first two characters specify the dot address mode. The second two characters specify the column number of the desired position.

Since there are 480 columns on a printed line, and only 256 values for the CHR$ function, two characters are needed. The second position character can be assigned any value from 0 to 255. The first position character can only be assigned the value 0 or 1. The value of the first character is multiplied by 256 and added to the second. For example, column number 300 would be represented by CHR$(1) + CHR$(44). For line numbers less than or equal to 255, the first position character value is 0, and the position is indicated in the second character.

Example

```
10 OPEN 1,4
20 A$ = CHR$(27) + CHR$(16) + CHR$(1) + CHR$(2)
30 PRINT#1,A$"OUTPUT"
40 CLOSE 1
50 END
```

In the preceding example, the four characters that are necessary for the dot address are assigned to the string variable A$. The first two characters specify the dot address mode. The second 2 characters specify the column number of the print head. Since the first value is 1 and second is 2, the column number is 256 + 2 = 258. Therefore, when the PRINT# statement at line 30 is executed, the print head will move to column number 258 and print the message OUTPUT.

Cursor Up -- CHR$(145)

The cursor up control code is used to return to the upper case letter and graphics character mode. The printer generally operates in the cursor up mode. As a result, the CHR$(145) function is only necessary when the cursor down mode has been specified earlier.

Example

```
10 OPEN 1,4
20 PRINT#1,"ABCDE"
30 PRINT#1,CHR$(17);"ABCDE"
40 PRINT#1,CHR$(145);"ABCDE"
50 CLOSE 1
60 END
```

In this example, the output at line 20 is printed in the cursor up mode. At line 30, the cursor down mode is specified, so the lower case characters will be printed. At line 40, the cursor up mode is specified, so the output appears in upper case letters again.

The printed results of the example are as follows:

```
ABCDE
abcde
ABCDE
```

Reverse Off -- CHR$(146)

The reverse off code is used to change the output mode from reverse to standard. The reverse mode is automatically turned off when a carriage return character is encountered. However, in order to turn off the reverse mode without proceeding to the next line, the CHR$(146) control character must be sent to the printer.

Example

```
10 OPEN 1,4
20 PRINT#1,CHR$(18);"OUTPUT1";
30 PRINT#1,CHR$(146);"OUTPUT2"
40 CLOSE 1
50 END
```

The preceding example contains a program which permits output on the same line in both the reverse and standard modes. At line 20, the reverse mode is used to display the message OUTPUT1. The semicolon at the end of line 20 suppresses the carriage return character. At line 30, the CHR$(146) character is used to turn off the reverse mode. As a result, the message OUTPUT2 is displayed in the standard mode.

CHAPTER 8.
COMMODORE 64 SOUND & GRAPHICS

The Commodore 64 computer is capable of producing many types of graphics displays with sixteen different colors. This chapter provides an explanation of the techniques used to generate graphics displays.

Display Colors

The computer's display can include any one of 16 colors for the background, as well as the border of the display. As a result, there are 256 possible combinations of background and border colors.

The computer has two memory locations that are reserved for the display's background color and border color. The memory location for the border color is 53280. The location for the background color is 53281. By using POKE statements, the values in these memory locations can be changed.

Each color is designated by a value from 0 to 15. These values, as listed in Table 8-1, are assigned to memory locations 53280 and 53281 to control the colors of the display.

For example, execute the following statements:

```
POKE 53280,0
POKE 53281,0
```

When the two POKE statements are executed, the border and background of the display both become black (color 0).

Table 8-1. Color Values

0	BLACK	8	ORANGE
1	WHITE	9	BROWN
2	RED	10	LIGHT RED
3	CYAN	11	GRAY #1
4	PURPLE	12	GRAY #2
5	GREEN	13	LIGHT GREEN
6	BLUE	14	LIGHT BLUE
7	YELLOW	15	GRAY #3

Text Colors

The color of the characters on the display can also be changed. The control key and the Commodore key (lower left) are used with the keys numbered 1 through 8 to choose one of 16 colors for the text. When the Control key is held down while a numbered key is pressed, the color of the cursor (and subsequent characters) changes. Eight more text colors can be chosen by holding down the Commodore key and pressing a numbered key. The text colors are listed in Table 8-2.

Graphics Characters

Most of the keys on the Commodore 64 keyboard have graphics symbols associated with them. When one of these keys is pressed in the shift mode, the graphics symbol on the right front side of the key will be generated. When the Commodore key is pressed with one of the graphics keys, the graphics symbol on the front left hand side of the key will be generated.

Table 8-2. Color Selection Keystrokes

Keystroke	Color	Keystroke	Color
control-1	Black	C=-1	Orange
control-2	White	C=-2	Brown
control-3	Red	C=-3	Light Red
control-4	Cyan	C=-4	Gray #1
control-5	Purple	C=-5	Gray #2
control-6	Green	C=-6	Light Green
control-7	Blue	C=-7	Light Blue
control-8	Yellow	C=-8	Gray #3

PRINT Statements

Graphics can be produced in the program mode by using the PRINT statement. The graphics characters to be output must be enclosed in quotation marks following PRINT.

The following PRINT statement,

PRINT "●○—_1"

would result in the following output:

●○—_1

The color characters can also be included within a PRINT statement. This is accomplished by pressing the Control or Commodore key and the desired color key simultaneously within the PRINT statement. When the color key combination is pressed, a special character appears in the PRINT statement. These characters (Table 8-3) do not appear in the output, but are used to change the color of the output. For example, the following PRINT statement would output the characters displayed in green:

PRINT "[↑]●o—_1"
●o—_1
[↑]* Press Control-6

Table 8-3. Color Selection Characters

Black	■	Orange	[♣ reversed]
White	[E] *	Brown	[◺] *
Red	[£] *	Light Red	[☒] *
Cyan	◥	Gray #1	[○] *
Purple	▐	Gray #2	[♣]
Green	[↑] *	Light Green	[▯] *
Blue	[←] *	Light Blue	[♦]
Yellow	[π] *	Gray #3	[⊞] *

Screen Locations

The display of the Commodore 64 is represented by an array of 40 columns and 25 rows. The computer reserves 2 locations in RAM for each display position. One location is used to designate the character and the other location is used to designate the color. The display array can be represented by a grid as shown in Illustration 8-1.

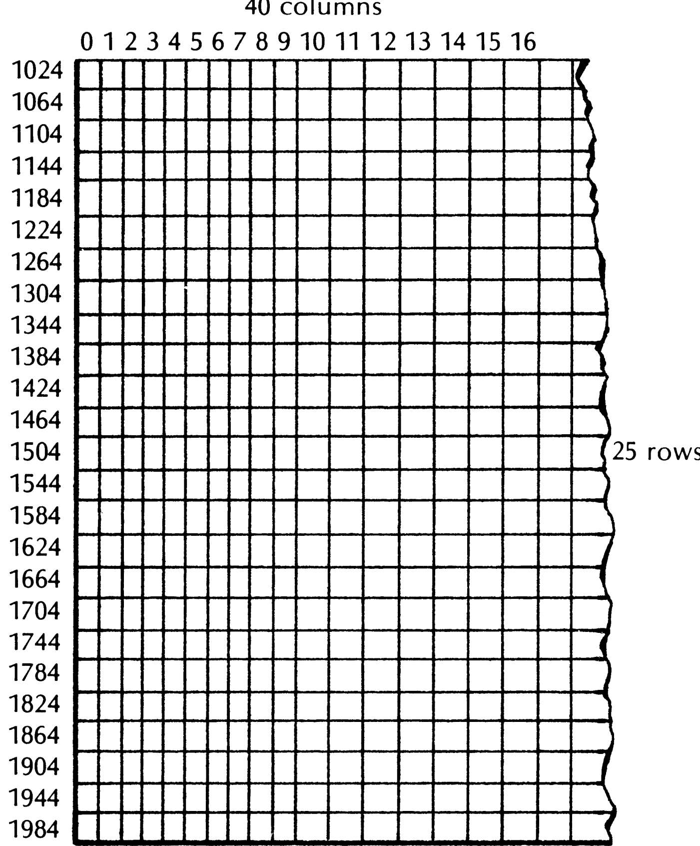
The character memory for the display begins at location 1024 and ends at location 2023. The color memory occupies locations 55296 through 56295.

Since there are 256 characters in each Commodore 64 character set, the values 0 through 255 are used in the character memory location. The values 0 through 15 are used in the color memory locations to correspond to the 16 colors listed in Table 8-1.

---

* These characters are displayed in reverse.

Illustration 8-1. Display Memory Grid



The character memory location for any position on the display can be calculated by the following formula:

$$location = 1024 + column + 40 * row$$

Since the color memory grid is identical to the location memory grid, a color location can be calculated from the following formula,

$$location = 55296 + column + 40 * row$$

or simply:

color location = *character location* + 54272

POKE Statements

The contents of a memory location are altered with a POKE statement. The first argument of a POKE statement is the memory location. The second argument is the value assigned to the specified location. For example, the following program,

```
10 N = 1024 + 20 + 40 * 12
20 POKE N,36
30 POKE N + 54272, 7
40 END
```

displays a yellow dollar sign in row number 12, column number 20. The character value for a dollar sign is 36. The color value for yellow is 7.

A POKE statement can be used to change the value of any memory location in RAM. If a POKE statement specifies a memory location that is part of ROM, the POKE statement has no effect.

Moving Characters

A character can be moved on the display merely by changing the memory location specified in the POKE statement. However, when a new screen location is specified, the character remains in the preceding location also. As a result, when moving characters, it is necessary to replace the previous position of the character with a blank space. Also, the previous character can be deleted by changing its color to the background color, as illustrated in the following example.

Example

```
10 POKE 53280,0:POKE 53281,0
20 FOR N = 1024 TO 1054
30 POKE N,ASC(">")
40 POKE N + 54272,7
50 POKE N + 54271,0
60 FOR T = 1 TO 50:NEXT T
70 NEXT N
```

In the preceding example, the POKE statements in line 10 set the background and border color to black. At line 20, a FOR, NEXT loop is established that sets N equal to the first 31 character positions on the screen. The POKE statement in line 30 places the "greater than" character on the screen, at location N.

The POKE statement in line 40 makes the character appear as yellow on the display. At line 50, the POKE statement changes the color of the preceding character to black. As a result, when the symbol moves forward, the symbol in the preceding space disappears. The FOR, NEXT loop in line 60 causes a delay, so that the results of the program can be observed at each step.

The Character Set

The information that the Commodore 64 uses to generate characters is stored in ROM, and therefore cannot be altered. However, the value that is used to specify the location of the character data is stored in RAM. As a result, an alternate set of characters can be used if the computer is instructed to take the character data from a section of RAM.

Since each character is represented by 8 bytes, it takes a great deal of effort to invent a whole new set of 255 characters. A more practical approach is to copy the whole character set from ROM, change only the necessary characters and leave the rest of the data intact.

Changing Characters

The first step in selecting programmable characters is to specify the memory location that the new character set will occupy. A location beginning at 12288 for the new data would be convenient. In order to choose this location as the new character set, execute the following POKE statement:

```
10 POKE 53272,(PEEK(53272) AND 240) + 12
```

In order to prevent the data from being erased by the operation of the computer, the next statement reserves a section of the memory for the character set:

```
20 POKE 52,48:POKE 56,48:CLR
```

The following POKE statements are required to establish the proper conditions for the transfer of data.

```
30 POKE 56334,PEEK(56334) AND 254
40 POKE 1,PEEK(1) AND 251
```

A FOR, NEXT loop is used to repeat the statement in line 60. The loop must repeat 8 times (once for each byte) for each character being transferred. This program copies the character data for CHR$(10) through CHR$(127).

```
50 FOR J = 0 TO 1016
60 POKE J + 12288, PEEK(J + 53248)
70 NEXT J
```

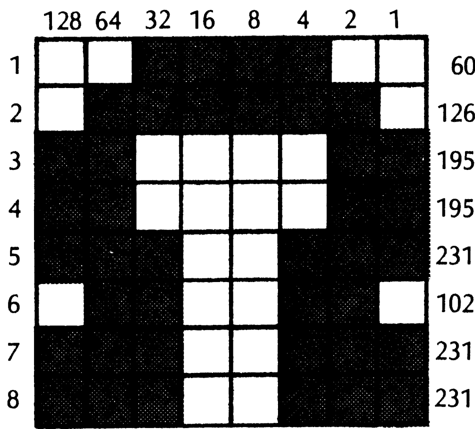
Two additional POKE statements are needed to return to the normal operation of the computer.

```
80 POKE 1, PEEK(1) OR 4
90 POKE 56334, PEEK(56334) OR 1
```

Now that the character set data is in RAM, it can be easily changed. Any of the characters can be changed, but the @ sign is a convenient example because its starting address is

already known (this was the very first character put in RAM). Since each character is a 8 × 8 array of bits (8 bytes), a character can be designed as in Illustration 8-2.

Illustration 8-2. Character Design Procedure



Determine which elements of the array are desired in the new character. Fill in the blocks and add the values of each block for every row. For example, the first row of the diagram has an element in column numbers 32, 16, 8, and 4. The sum of these values is 60. Repeat this procedure for each row of the diagram.

The eight values that are calculated correspond to the eight bytes of data for the character. If these eight values are put in the memory addresses where the code for the @ symbol resides (12288-12296), the new character will be displayed whenever the @ key is pressed. This is achieved in the following statements:

```
100 FOR J = 1 TO 8
110 READ X
120 POKE 12287 + J,X
130 NEXT J
140 DATA 60, 126, 195, 195
150 DATA 231, 102, 231, 231
160 END
```

Due to the large amount of data being exchanged, the program presented in this section requires more than 15 seconds to complete execution. Be sure to include all the preceding statements (10-160) in the program.

High Resolution Graphics

High resolution graphics on the Commodore 64 are achieved by dividing the display into 64,000 picture elements. The screen has 200 rows and 320 columns. Each picture element can be controlled individually with one bit of memory. As a result, high resolution graphics require 64,000 bits (8000 bytes) of memory.

Bit Maps

A **bit map** is a section of the computer's memory that is used to control a high resolution graphics display. The bit map is similar to the display memory grid mentioned earlier in this chapter. However, the locations on the bit map are not as easily determined.

A convenient memory location to begin the bit map is 8192. The first 8 bytes are used to control the first character position on the screen. The 8 bits of the first byte control the 8 picture elements at the top of the first character position. The second byte controls the second row of the first character position, etc. As a result, each character is represented by a 64 element array similar to the programmable character array of Illustration 8-2.

The second group of 8 bytes in the bit map are used to control the second character location on the screen. The configuration of the bit map is displayed in Illustration 8-3.

Illustration 8-3. Bit Map Configuration

	First Column	Second Column	
First Row	byte 0 byte 1 byte 2 byte 3 byte 4 byte 5 byte 6 byte 7	byte 8 byte 9 byte 10 byte 11 byte 12 byte 13 byte 14 byte 15	etc.
Second Row	byte 320 byte 321 byte 322 byte 323 byte 324 byte 325 byte 326 byte 327	byte 328 byte 329 byte 330 byte 331 byte 332 byte 333 byte 334 byte 335	etc.
	etc.	etc.	

At first it seems nearly impossible to control this bit map in an orderly, efficient manner. However, if the picture elements on the screen are numbered according to their position on the display, the operation is quite simple.

If the picture elements are designated by their column and row on the display, with the following notation:

(column, row)

the picture element (0,0) occurs in the upper left corner of the display. Similarly, the element (319,199) is located in the lower right corner of the display.

In order to convert the coordinates (*column, row*) to a bit map location, there are four values that must be calculated for each bit. First of all, the character location of the bit needs to be determined. Next, the row of the bit within that character is calculated. Finally, the bit number within that row is calculated. Illustration 8-4 provides an example of this concept.

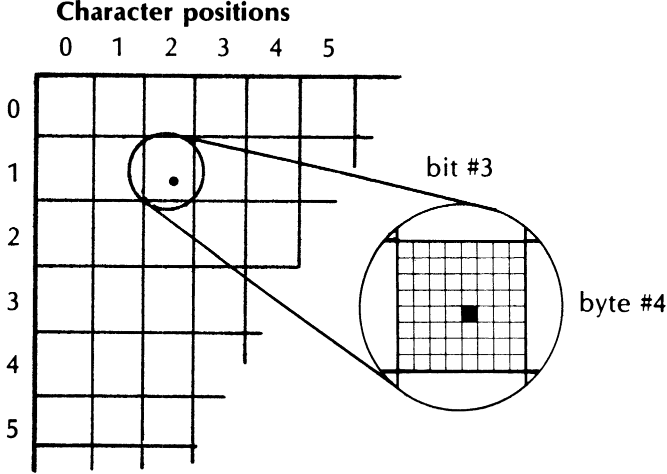
Consider the picture element (20,12). The character position can be calculated by the following formula:

X = INT(COL/8)
Y = INT(ROW/8)

X represents the column number of the character. Y represents the row number. It can be shown that the point (20,12) lies within the character in column 2, row 1.

Illustration 8-4. Calculating a Bit Memory Location

Point (20,12)



The AND operator can be used to determine the byte number within the character. Recall that the AND operator compares its arguments bit-by-bit, and returns the number that has a bit equal to 1 only where both arguments have a bit equal to 1. In the example, the byte number is calculated by the formula:

BYTE = ROW AND 7

The binary value of 12 is 00001100, and the binary value of 7 is 00000111. As a result, the value of 12 and 7 is:

```
      00001100
AND   00000111
      --------
      00000100
```

which equals 4.

The calculation for the bit number within the byte is similar to the previous formula. However, the bits are numbered 0 through 7 from right to left. As a result, the formula,

BIT = 7 - (COL AND 7)

is used. For the element (20,12), COL equals 20. The expression (COL AND 7) is evaluated as follows:

```
      00010100
AND   00000111
      --------
      00000100
```

Therefore, the bit value for the point (20,12) is 7 - 4 = 3.

The last formula combines the results of the preceding formulae to compute the desired memory location.

LOC = 8192 + Y * 320 + X * 8 + BYTE

A POKE statement is used to change the value of the memory location that controls the specified picture element. Since one byte controls 8 picture elements, only one bit of the memory should be changed for each element. Therefore, a PEEK statement is used to observe the initial value of the memory location. An OR is used to change only one bit of the memory value. The following statement actually causes the picture element to be illuminated.

```
POKE LOC, PEEK(LOC) AND (255 - 2 ↑ BIT)
```

The expression 2 ↑ BIT is used to convert the bit value to a decimal number.

Before this method can be used to generate graphics displays, several other statements are necessary. The following statements are used to specify location 8192 for the bit map, and to enter the high resolution mode.

```
POKE 53272, PEEK(53272) OR 8
POKE 53265, PEEK(53265) OR 32
```

The memory locations 1024 to 2023 must be assigned the color values of the character locations on the screen. Also, the bit map area of the memory must be initialized with a FOR, NEXT loop before the program begins. The following FOR, NEXT loops initialize the bit map and set the picture element color to blue (color number 6).

```
FOR J = 8192 TO 16191:POKE J, 255:NEXT J
FOR J = 1024 TO 2023:POKE J, 6:NEXT J
```

The following example demonstrates the use of the high resolution graphics mode. The statements for calculating the locations are placed in a subroutine to simplify the program.

Lines 10 through 60 clear the display and set the initial conditions outlined above. The POKE statement at line 20 sets the border color to black. Since the bit map contains 8000 bytes, it takes about 30 seconds to execute the loop at line 50.

Example

```
  10 PRINT CHR$(147)
  20 POKE 53280,0
  30 POKE 53272, PEEK(53272) OR 8
  40 POKE 53265, PEEK(53265) OR 32
  50 FOR J = 8192 TO 16191:POKE J, 255:NEXT J
  60 FOR J = 1024 TO 2023: POKE J, 6:NEXT J
 100 FOR ROW = 10 TO 50 STEP 10
 110 FOR COL = 0 TO 319
 120 GOSUB 1000
 130 NEXT COL
 140 NEXT ROW
 999 GOTO 999
1000 X = INT(COL/8)
1010 Y = INT(ROW/8)
1020 BYTE = ROW AND 7
1030 BIT = 7 - (COL AND 7)
1040 LOC = 8192 + Y * 320 + X * 8 + BYTE
1050 POKE LOC, PEEK(LOC) AND (255 - 2 ↑ BIT)
1060 RETURN
```

Lines 100 through 140 contain two FOR, NEXT loops which determine the picture elements that are illuminated. The variable ROW is assigned the values 10, 20, 30, 40, and 50. For each of these values, the variable COL is assigned the values 0 through 319. As a result, 5 horizontal lines are displayed on the screen. The GOSUB statement at line 120 branches the program control to line 1000 for each new value of ROW and COL.

The subroutine from line 1000 through 1060 performs the necessary calculations to plot the data on the display.

The GOTO statement at line 999 is used to maintain the display. To return to standard mode of output, press the Run/Stop and Restore keys. Due to the large number of

calculations in this program, it requires about 2½ minutes to execute. The final display is five blue horizontal lines on a dark background.

SPRITES

Sprites are graphics characters that can be easily moved around the display. Once a sprite is defined, the movement of the image is simple. Up to 8 sprites can be controlled at any time.

Defining a Sprite

A sprite is defined on a 24 x 21 array of picture elements. As a result, it takes 63 bytes of data to define a sprite. The method of determines the data for a sprite is similar to the method used in Illustration 8-2. However, since a sprite is 24 characters wide, 3 bytes are required for each row of the image. An example of defining a sprite is shown in Illustration 8-5.

Each byte of the diagram in Illustration 8-5 should be numbered as follows:

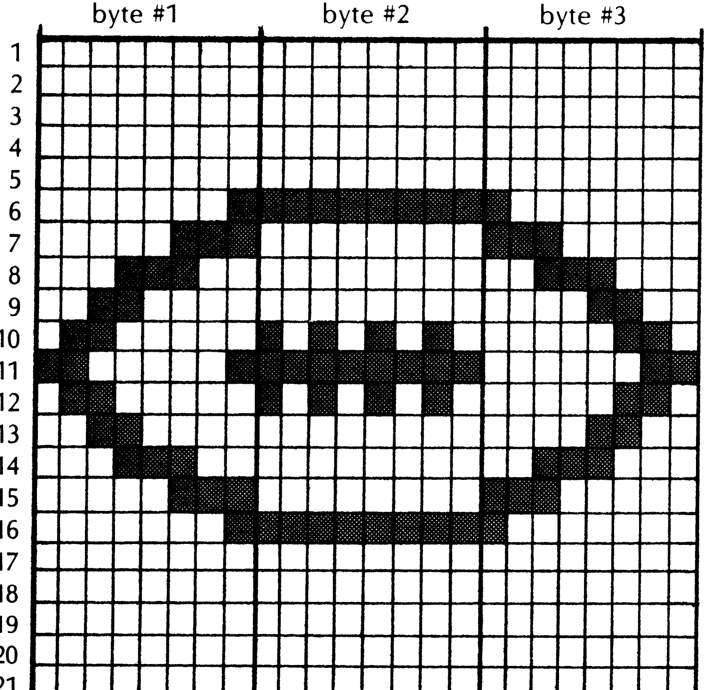
128	64	32	16	8	4	2	1

For each desired block in the diagram, the value of that block should be added to the other values in that byte to form one data item. For example, the first five lines of the sprite illustration are blank, so the first 15 data items are zero. However, the middle of the tenth line appears as follows:

Since the blocks numbered 128, 32, 8, and 2 are all filled in, byte number 29 is equal to 128 + 32 + 8 + 2 = 170.

When the values of the 64 bytes are calculated, the values are

Illustration 8-5. Determining Sprite Data



stored in memory with POKE statements. These memory values define the shape of the sprite.

Controlling a Sprite

There are several memory locations that are used to control sprites. The memory location that designates the sprite (display chip) section of memory is 53248. Most of the memory locations are in the vicinity of this location, so 53248 is usually assigned to a variable.

In this section, a register refers to the memory location in the display chip. As a result, the value of a register must be added

to 53248 to designate a memory location. A summary of the register functions is listed in Table 8-4.

Activating a Sprite

Register 21 designates the sprites that appear on the display. Each bit of the value in register 21 controls one of the sprites. The sprites are numbered 0 through 7, so the value of register 21 should be 2 ↑ N, where N is the number of the desired sprite. If more than one sprite is in use, the value of register 21 should be:

$$2^N + 2^N + \ldots$$

For example, to use sprites 2 and 4, the value of register 21 should be:

$$2^2 + 2^4 = 20$$

Locating a Sprite

Register 0 controls the X coordinates of sprite 0. Register 1 controls the Y coordinate of sprite 0. Registers 2 through 15 control the coordinates for sprites 1 through 7 as listed in Table 8-4. These registers can have any value from 0 through 255. However, a Y coordinate of 255 causes the sprite to move about two thirds of the way across the screen.

In order to move the sprite further to the right, the value of register 16 must be changed. Since register 16 maintains the most significant bit for all 7 sprites, the values must be entered in the same manner as values are entered into register 20. For example, if sprites 1 and 3 have X coordinates greater than 255, the value of register 16 must be:

$$2^1 + 2^3 = 10$$

Expanding Sprites

A sprite can be expanded to twice its normal width, or twice its normal height. Register 23 is used to double the height of a sprite. Since one register controls all 8 sprites, add 2 ↑ N to the value of register 23, where N is the number of the sprite to be widened.

Register 29 is used in the same way to double the width of a sprite.

Color Sprites

Registers 39 through 46 are used to determine the color of the sprites 0 through 7, respectively. Any color value can be assigned to these memory locations to change the color of a sprite.

DATA Assignments

Memory locations 2040 through 2047 instruct the computer where to find the data for sprites 0 through 7. Data is arranged in blocks of 64 bytes for each sprite. The initial memory location must be an integer multiple of 64. Memory locations 2040 through 2047 must contain the block number for the sprite data. For example, if sprite number 1 is being used, and the value of location 2041 is 12, the sprite data must begin at location 12 * 64 (which equals 768).

Table 8-4. Sprite Registers

Register#	Operation	Sprite#
0	X coordinate	Sprite 0
1	Y coordinate	Sprite 0
2	X coordinate	Sprite 1
3	Y coordinate	Sprite 1
4	X coordinate	Sprite 2
5	Y coordinate	Sprite 2
6	X coordinate	Sprite 3
7	Y coordinate	Sprite 3
8	X coordinate	Sprite 4
9	Y coordinate	Sprite 4
10	X coordinate	Sprite 5
11	Y coordinate	Sprite 5
12	X coordinate	Sprite 6
13	Y coordinate	Sprite 6
14	X coordinate	Sprite 7
15	Y coordinate	Sprite 7
16	X coordinate, highest bit	All Sprites
21	Activate Sprite	All Sprites
23	Expand Length	All Sprites
29	Expand Height	All Sprites
39	Color	Sprite 0
40	Color	Sprite 1
41	Color	Sprite 2
42	Color	Sprite 3
43	Color	Sprite 4
44	Color	Sprite 5
45	Color	Sprite 6
46	Color	Sprite 7

Example

```
 10 PRINT CHR$(147)
 20 FOR A = 0 TO 62:READ X:POKE 14 * 64 + A,X:NEXT A
 30 POKE 2043, 14
 40 Q = 53248
 50 POKE Q + 21, 8
 60 POKE Q + 23, 8
 70 POKE Q + 29, 8
 80 POKE Q + 42, 9
 90 FOR X = 0 TO 350 STEP 2
100 IF X>255 THEN X2 = 1:X1 = X - 255:GOTO 120
110 X1 = X:X2 = 0
120 POKE Q + 6,X1
130 POKE Q + 16,8 * X2
140 POKE Q + 7, 100 * (1 - SIN(X/100)) + 100
150 NEXT X
160 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
170 DATA 1,255,128,7,0,224,28,0,56,48,0
180 DATA 12,96,170,6,193,255,3,96,170,6
190 DATA 48,0,12,28,0,56,7,0,224,1,255
200 DATA 128,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

The preceding example contains a program that demonstrates the use of sprite graphics. The data for this program was calculated from Illustration 8-5.

Line 20 contains a FOR, NEXT loop that reads the 63 data items and assigns them to memory locations 896 through 958. This is the fourteenth block of memory.

The POKE statement at line 30 indicates that the data for sprite number 3 is located in the fourteenth block of memory.

Line 50 activates sprite number 3. The POKE statements at lines 60 and 70 expand the figure in both directions.

The POKE statement at line 80 makes the figure appear brown on the display.

Line 90 initiates a FOR, NEXT loop that generates X values from 0 to 350. Since the X coordinate of the display can exceed 255, register 16 must be used to store the most significant bit for the X coordinate. The IF, THEN statement at line 100 takes care of this problem. If X exceeds 255, the variable X2 is assigned the value 1 and the variable X1 is assigned the difference between X and 255.

At line 120, the value of X1 is assigned to the X coordinate register for sprite number 3.

At line 130, register 16 is assigned the value 0 if X does not exceed 255. Otherwise, register 16 is assigned the value 8.

The POKE statement at line 140 assigns a value to the Y coordinate register for sprite number 3. The arched path of the football results from the 1 - SIN(X/100) term of this calculation.

SOUND

The Commodore 64 computer has many sound and music features. Three voices are available for creating music or sound effects. Sound is generated in three waveshapes as well as noise, with a wide range of frequencies. Also, the attack, decay, sustain, and release of the tone are adjustable.

The use of these features does not require any previous knowledge of music. All of these functions are controlled with simple BASIC statements.

Voices

The three voices that are used for music are associated with separate memory locations. Each voice is operated independently, except for the volume control. A voice is activated when the appropriate POKE statements have been executed. The POKE statements are used to select the

waveshape, frequency, attact, decay, sustain, and release of the voices. Register 54296 is used to control the volume of all the voices collectively.

Waveshapes

There are three waveshapes that are used by the Commodore 64. Each waveshape has a distinct sound associated with it. The three waveshapes, triangle, sawtooth, and pulse, are depicted graphically in Illustration 8-6.

Each waveshape is assigned a value that is used to activate a voice. Also, the waveshape is assigned a value that is used to turn a voice off. Table 8-6 lists the values used to turn a voice on and off.

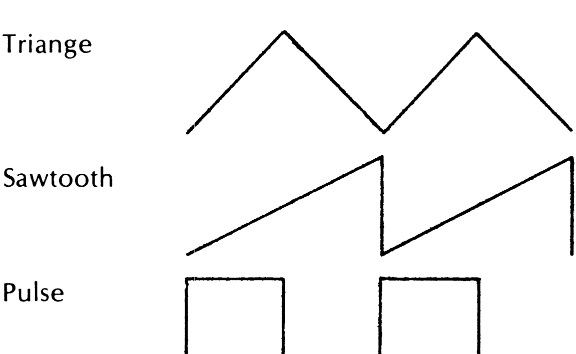
Table 8-5. Voice Waveform Registers

Voice	Register
1	54276
2	54283
3	54290

Table 8-6. Waveshape Data

Triangle		Sawtooth		Pulse		Noise	
ON	OFF	ON	OFF	ON	OFF	ON	OFF
17	16	33	32	65	64	129	128

Illustration 8-6. Waveshapes



Frequency

The pitch of the voice depends on the frequency selected. Each voice has two registers that are used to control the frequency. These registers are listed in Table 8-7.

Table 8-7. Frequency Registers

Voice	High Register	Low Register
1	54273	54272
2	54280	54279
3	54287	54286

Appendix E contains a table of values that correspond to 8 octaves of musical notes. Each note has two corresponding data items. Each voice requires both values to produce a selected note.

Example*

```
100 FOR J = 54272 TO 54296
110 POKE J,0
120 NEXT J
160 POKE 54296,15
170 POKE 54273,47
180 POKE 54272,107
200 POKE 54276,33
210 FOR J = 1 TO 1000:NEXT J
220 POKE 54276,0
```

The preceding example contains a program that generates a F# note in the fifth octave. The FOR, NEXT loop in lines 100 through 120 is used to clear the memory locations that are used for sound. The POKE statement in line 160 sets the volume to the maximum value (15). The minimum value for the volume is 0.

Lines 170 and 180 are used to set the high and low frequency registers for voice number 1. The values for these registers are taken from the musical note table in Appendix E.

Line 200 contains a POKE statement that sets the waveshape register for voice number 1 equal to 33. This statement indicates that the output is a sawtooth wave.

The FOR, NEXT loop in line 210 causes the computer to pause while the tone is produced. The program ends at line 220 when the tone is shut off.

Pulse Wave

The sound of a pulse wave can be changed by altering the duty cycle. The duty cycle is defined as the amount of time the pulse is "on" compared to the time of one complete cycle. The concept of duty cycle is displayed in Illustration 8-7.

---

* This example is modified to obtain the example on page 263.

Illustration 8-7. Duty Cycle of Pulse Waves

duty cyle = 1/3 duty cycle = 1/2 duty cycle = 4/5

The duty cycle of the pulse wave can be adjusted to generate a wide range of sounds. The duty cycle is selected by assigning values to 2 registers, HI and LO. The formula used to calculate the duty cycle is as follows:

duty cycle = (HI * 256 + LO)/4095

The registers used to set the duty cycle for each voice are listed in Table 8-8.

Table 8-8. Duty Cycle Registers

Voice	HI	LO
1	54275	54274
2	54282	54279
3	54289	54288

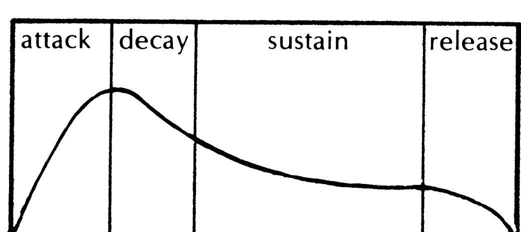
Envelope

An envelope of a tone is a way of describing how the volume of the tone changes. For example, when a door slams, the noise dies away quickly. However, when a bell is struck, the tone dies away slowly. Obviously, the envelopes of these sounds are very different.

There are four terms used to describe an envelope. The **attack** is the time it takes the sound to reach its initial, maximum volume. The **decay** is the time it takes the sound to die off to an intermediate, stable volume. The **sustain** is the amount of time the tone remains at the intermediate

volume. The **release** is the amount of time it takes to die off to zero. A typical envelope is portrayed in Illustration 8-8.

Illustration 8-8. A Typical Envelope



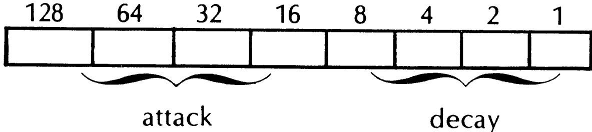
The Commodore 64 uses two registers to control the envelope of each voice. The registers for attack-decay and sustain-release are listed in Table 8-9.

Table 8-9. Envelope Control Registers

Voice	Attack-Decay	Sustain-Release
1	54277	54278
2	54284	54285
3	54291	54292

Attack-Decay

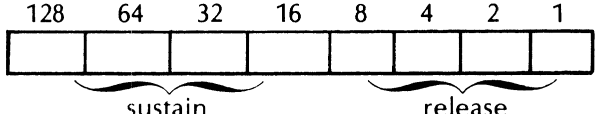
The attack-decay registers are divided into two parts. The highest 4 bits are used to control the attack. The lowest 4 bits are used to control the decay.



The length of the attack depends only on the 128, 64, 32, and 16 bits. As a result, if this register is assigned the value 128 + 64 + 32 + 16 = 248, the voice would have the maximum possible attack, but no decay. On the other hand, if this register is assigned the value 8 + 4 + 2 + 1 = 15, the voice will have no attack, but the maximum amount of decay.

Sustain-Release

The sustain-release registers are also divided into two parts. The operation of these registers is analogous to the attack decay registers.



The length of the sustain depends only upon the values 128, 64, 32, and 16. Combinations of sustain and release values must be added together in the same manner as attack and decay values.

An intermediate value of sustain combined with a small amount of release correspond to the value 64 + 2 = 66.

The effects of attack, decay, sustain, and release are demonstrated in the following example.

Example*

```
 10 INPUT "ATTACK";A
 20 INPUT "DECAY";D
 30 INPUT "SUSTAIN";S
 40 INPUT "RELEASE";R
100 FOR J = 54272 TO 54296
110 POKE J,0
120 NEXT J
140 POKE 54277, 16 * 2↑(A - 1) + 2↑(D - 1)
150 POKE 54278, 16 * 2↑(S - 1) + 2↑(R - 1)
160 POKE 54296,15
170 POKE 54273,47
180 POKE 54272,107
190 FOR T = 1 TO 100
200 POKE 54276,33
210 FOR J = 1 TO 10:NEXT J
220 POKE 54276,32
230 NEXT T
240 GOTO 10
```

This example program is a modification of the example program on page 259. Be sure to alter lines 210 and 220. The remainder of the previous example is unchanged.

Lines 10 through 40 have been included to allow different values of attack, decay, sustain, and release. The input for these parameters should be one of the values 1, 2, 3, or 4.

The larger values for input correspond to greater attack, decay, etc. For example, the values 4, 1, 1, 1 would cause a long attack, but a short decay, sustain, and release.

The POKE statements at lines 140 and 150 convert the input values to the correct format, and assign the values to the attack-decay and sustain-release registers. The calculations in these statements are valid only if the variables have the values 1, 2, 3, or 4.

---

*This example is modified again on page 265.

Line 160 determines the volume. Lines 170 and 180 determine the note (F#). The FOR, NEXT loop that begins at line 190 repeats the sound 100 times. The POKE statement at line 200 specifies a sawtooth wave. The waveshape data must be specified in the program after the attack-decay and sustain-release data.

The FOR, NEXT loop at line 210 causes a small delay while the sound is produced. The POKE statement at line 220 turns off the sound, but allows it to be repeated. When a 0 is used in this statement, the voice must be redefined before it can be activated again.

At line 240, the program control returns to line 10 for more data. To end this program press Run/Stop and Restore.

MUSIC

Music can be programmed on the Commodore 64 by using READ and DATA statements to input the data. Since each note requires two data items for the frequency, each note of music requires at least two values in a DATA statement. Generally, a data item is also used for the duration of the tone. Variables can also be used for the volume, waveshape, and envelope controls. The following example is a modification of the example program on page 263.

Example

```
 10 INPUT "ATTACK";A
 20 INPUT "DECAY";D
 30 INPUT "SUSTAIN";S
 40 INPUT "RELEASE";R
100 FOR J = 54272 TO 54296
110 POKE J,0
120 NEXT J
140 POKE 54277, 16 * 2 ↑ (A - 1) + 2 ↑ (D - 1)
150 POKE 54278, 16 * 2 ↑ (S - 1) + 2 ↑ (R - 1)
160 POKE 54296,15
165 READ N1, N2, Q:IF Q = 99.99 THEN END
170 POKE 54273,N1
180 POKE 54272,N2
190 FOR T = 1 TO Q
200 POKE 54276,33
210 FOR J = 1 TO 10:NEXT J
220 POKE 54276,32
230 NEXT T
240 GOTO 165
300 DATA 4,48,10
310 DATA 8,97,10
320 DATA 16,195,30
330 DATA 33,135,20
340 DATA 0,0,99.99
```

Be sure to add line 165 and the five DATA statements to the previous example. Also be sure to modify lines 170, 180, 190, and 240.

This program uses the variables N1, N2, and Q to represent the two note values, and the duration of the tone. These values are read at line 165.

Any number of DATA statements can be used in this program, as long as three data items in the correct order are used for each note.

This program generates a C note in octaves 2 through 5. The table of values for musical notes is found in Appendix E.

The value 99.99 is used as a flag in this program to indicate the last data item. The values of N1 and N2 in the last DATA statement have no effect, since the program ends when the flag value is read.

APPENDICES

Appendix A. Commodore 64 BASIC Error Messages

When a statement cannot be successfully executed, an error message is displayed. An error can occur in direct mode, or within a program. Generally, an error stops the execution of a program or an I/O operation. When the problem is corrected, the program can be executed again, but the CONT command cannot be used to continue a program after an error.

The error messages are always preceded by a question mark and followed by the word ERROR. If an error occurs in a program, the line number of the problem is also displayed. The following error message is an example of the format of error messages in the program mode:

?DIVISION BY ZERO ERROR IN 40

In the direct mode, the same error would have the following format:

?DIVISION BY ZERO ERROR

The following tables include a brief explanation of each error message.

Table A-1. BASIC Error Messages

Error Message	Error Description
BAD DATA	String data was transmitted from an open file when the program expected numeric data.
BAD SUBSCRIPT	An attempt was made to use an array element whose subscript had not been within the range defined by the DIM statement.
CAN'T CONTINUE	The CONT command will not function properly. This is generally caused by a program error.
DEVICE NOT PRESENT	An attempt was made to execute an OPEN, CLOSE, CMD, PRINT#, INPUT#, or GET# with the necessary input or output device being unavailable.
DIVISION BY ZERO	Division by zero is not allowed.
EXTRA IGNORED	More items were entered in response to an INPUT statement than allowed. Any excess items will be disregarded.
FILE NOT FOUND	A specified file in a LOAD statement cannot be located on the designated cassette or disk.

Error Message	Error Description
FILE NOT OPEN	Before a file can be accessed, that file must first have been opened.
FILE OPEN	An OPEN statement used the same file number as that of a file already open.
FORMULA TOO COMPLEX	The expression should be separated into two or more parts.
ILLEGAL DIRECT	An INPUT statement cannot be used in the immediate mode.
ILLEGAL QUANTITY	The argument of a function is outside of the range allowed.
LOAD	A problem exists with a program stored on cassette tape.
NEXT WITHOUT FOR	A NEXT statement does not have a corresponding FOR statement.
NOT INPUT FILE	The program tried to input data from a file opened for output.
NOT OUTPUT FILE	The program tried to output data to a file opened for input.
OUT OF DATA	Not enough data items available for READ statement.

Error Message	Error Description
OUT OF MEMORY	All available RAM memory is in use and no more is available for program or variable storage.
OVERFLOW	A number was encountered which is greater than 1.70148-83E+38 or less than -1.7014188-3E+38.
REDIM'D ARRAY	An array can only be dimensioned once in a program. Remember, if an array element is referenced without being dimensioned, that array is automatically dimensioned for 10 elements.
REDO FROM START	Characters were input in response to an INPUT statement when numeric data was expected. The program will continue when the correct data type is entered.
RETURN WITHOUT GOSUB	The program encountered a RETURN statement without a corresponding GOSUB statement.
STRING TOO LONG	A string can contain a maximum of 255 characters.

Error Message	Error Description
SYNTAX	A statement cannot be comprehended by the BASIC interpreter. This is generally caused by incorrect spelling of reserved words or incorrect punctuation.
TYPE MISMATCH	This error occurs when a number is assigned for a string variable or a string assigned to a numeric variable.
UNDEF'D FUNCTION	The program encountered a user defined function which was not previously defined with the DEF FN statement.
UNDEF'N STATEMENT	A GOTO or GOSUB statement referenced a line number which does not exist.
VERIFY	The program on tape or diskette does not match the program in RAM.

Table A-2. DOS Error Messages

Error Number	Error Message	Error Description
0		No error exists.
1		No error exists. Returns data regarding the number of files scratched with the scratch command.
2-19		Unused.
20	READ ERROR (block header not found)	The disk drive controller could not find the header of the specified data block. This error could have been caused by referencing an illegal sector number, or by the header having been erased.
21	READ ERROR (no sync character)	This error is generated when the disk controller cannot locate the sync mark on the chosen track. This error can be caused by not having a diskette inserted in the drive, using an unformatted diskette, a misaligned read/write head or some other hardware problem.
22	READ ERROR (data block not present)	The disk controller attempted to read or verify a data block that had not been written correctly to the diskette. This error message generally appears when the BLOCK commands are used. In these instances, an illegal track and/or sector was requested.

Error Number	Error Message	Error Description
23	READ ERROR (checksum error in data block)	This error message results when data has been read into memory, but the checksum of that data is in error. This generally indicates an error in one or more of the data items. This error can be caused by faulty grounding.
24	READ ERROR (byte encoding error)	Data or a header has been read into memory with an incorrect bit pattern in a byte of data. This error may also be caused by faulty grounding.
25	WRITE ERROR (write-verify error)	This error results if the disk controller recognizes a difference between the data written to the diskette and that in memory.
26	WRITE PROTECTION	This message is generally the result of an attempt to write to a diskette which has been write-protected.
27	READ ERROR (checksum error in header)	The disk controller discovered an error in the header of the data block to be read into memory. As a result, it was not read. This error may also be the result of faulty grounding.
28	WRITE ERROR (long data block)	After writing a data block, the disk controller will attempt to find the sync mark of the next header. This error is generated when the disk controller cannot locate this sync mark within a specified period of time. This error can be caused by

Error Number	Error Message	Error Description
		malfunctioning hardware or by a faulty diskette format (when data extends into the next block).
29	DISK ID MISMATCH	This error message is usually generated when the controller attempts to access a diskette which had not previously been initialized. However, this error can also result when a diskette has an incorrect header.
30	SYNTAX ERROR (general syntax)	This error is generally caused by illegal syntax in the use of a DOS command.
31	SYNTAX ERROR (invalid command)	DOS is unable to interpret the command. Be certain that the command begins in the first position.
32	SYNTAX ERROR (long line)	The command issued was over the maximum allowed (55 characters).
33	SYNTAX ERROR (invalid file name)	An invalid filename was used in an OPEN or SAVE statement.
34	SYNTAX ERROR (no file given)	This error is often the result of a filename being omitted from a DOS command or the use of a filename that was not allowed in DOS. This error is often the result of the colon (:) being omitted from the command.
39	SYNTAX ERROR (invalid command)	This error occurs when the command sent to the command channel (secondary address = 15) cannot be interpreted by DOS.

Error Number	Error Message	Error Description
50	RECORD NOT PRESENT	This error is often caused when an INPUT# or GET# statement attempts to read beyond the last record in a file. The record pointer should be repositioned before attempting to re-execute INPUT# or GET#. This error can also result in a relative file from positioning to a record beyond the end of file. If this was done in order to add a new record with PRINT#, this error message can be ignored.
51	OVERFLOW IN RECORD	This error results from the PRINT# statement attempting to write beyond a record's limit. The excess data is truncated. The total number of characters in a record must not exceed its defined size. Remember, the carriage return character is sent at the end of the record and is included in the record size.
52	FILE TOO LARGE	The record position within a relative file signifies that a disk overflow will result from the intended disk operation.
60	WRITE FILE OPEN	This error results from an attempt to open a file for a read operation that had previously been opened for a write operation and had not been closed.
61	FILE NOT OPEN	This error is generated when an attempt is made to access a file which had not previously been opened.

Error Number	Error Message	Error Description
62	FILE NOT FOUND	The requested file does not exist on the specified drive.
63	FILE EXISTS	An attempt was made to create a file already in existence on the specified diskette.
64	FILE TYPE MISMATCH	The file type entry does not match the file type given in the directory for that file.
65	NO BLOCK	This error message is generated as a result of an attempt to use the Block-Allocate command. This error occurs when the block specified to be allocated had already been allocated. If the NO BLOCK error is generated, the track and sector number of the next available block will be indicated. If a zero is indicated, then all blocks higher in number are currently in use.
67	ILLEGAL SYSTEM T OR S	An illegal system track or sector was indicated.
70	NO CHANNELS (available)	This error results when the channel specified is unavailable. This error also occurs when all channels are in use. A maximum of six files may be open under direct access channels. A maximum of five sequential files may be open.
71	DIRECTORY ERROR	This error is the result of a problem in the Block Availability Map. This can be caused either by a problem in the block allocation or by the Block Availability Map having

Error Number	Error Message	Error Description
		been overwritten in memory. This error condition may be corrected by reinitializing the diskette as this will rewrite the BAM in memory. However, open files may be erased by this action.
72	DISK FULL	This error condition is the result of the directory having been filled (144 files) or all blocks on the diskette being in use.
73	DOS MISMATCH	Versions 1 and 2 of DOS are read-compatible but not write-compatible. In other words, a DOS 1 diskette can be read with DOS 2 and vice versa. However, a diskette written under DOS 1 cannot be written upon by DOS 2 and vice versa. This is due to the fact that DOS 1 and DOS 2 format diskettes in a different manner.
74	DRIVE NOT READY	The drive was accessed without any diskettes present inside.

Appendix B. Commodore 64 Code Set

This appendix outlines the characters associated with the Commodore 64 Code Set. By entering PRINT CHR$(X), where X is one of the following codes, the associated character will be displayed. Also, by entering PRINT ASC("Y") where Y is the character, the code associated with that character will be displayed.

Code	Character	Code	Character
0		22	
1		23	
2		24	
3		25	
4		26	
5	WHT	27	
6		28	RED
7		29	CRSR →
8	DISABLES SHIFT C=	30	GRN
9	ENABLES SHIFT C=	31	BLU
10		32	SPACE
11		33	!
12		34	"
13	RETURN	35	#
14	SWITCH TO LOWER CASE	36	$
15		37	%
16		38	&
17	CRSR ↓	39	'
18	RVS ON	40	(
19	CLR HOME	41	)
20	INST DEL	42	*
21		43	+

Code	Character	Code	Character
44	,	74	J
45	-	75	K
46	.	76	L
47	/	77	M
48	0	78	N
49	1	79	O
50	2	80	P
51	3	81	Q
52	4	82	R
53	5	83	S
54	6	84	T
55	7	85	U
56	8	86	V
57	9	87	W
58	:	88	X
59	;	89	Y
60	<	90	Z
61	=	91	[
62	>	92	£
63	?	93	]
64	@	94	↑
65	A	95	←
66	B	96	
67	C	97	
68	D	98	
69	E	99	
70	F	100	
71	G	101	
72	H	102	
73	I	103	

Code	Character	Code	Character
104		134	f3
105		135	f5
106		136	f7
107		137	f2
108		138	f4
109		139	f6
110		140	f8
111		141	SHIFT RETURN
112		142	SWITCH TO UPPER CASE
113	●	143	
114		144	BLK
115	♥	145	CRSR↑
116		146	RVS OFF
117		147	CLR HOME
118	╳	148	INST DEL
119	○	149	
120	♣	150	
121		151	
122	♦	152	
123	┼	153	
124		154	
125		155	
126	π	156	PUR
127		157	CRSR←
128		158	YEL
129		159	CYN
130		160	SPACE
131		161	
132		162	
133	f1	163	

Code	Character	Code	Character
164		178	
165		179	
166		180	
167		181	
168		182	
169		183	
170		184	
171		185	
172		186	
173		187	
174		188	
175		189	
176		190	
177		191	

Codes 192-223 are the same as 96-127.
Codes 224-254 are the same as 160-190.
Code 255 is the same as 126.

Appendix C. Screen Codes

The following tables contain the various characters and their related codes in the two character sets. When these codes are POKE'd to a screen memory address (1024 to 2023) the corresponding character will be output. Conversely, if a screen memory address is PEEK'd, the code for the corresponding character will be returned.

As discussed in Chapter 2, the character sets can be changed by simultaneously pressing the Shift and the Commodore keys. Also, the POKE statement can be used as follows to change character sets:

POKE 53272,21 → upper case character set

POKE 53272,23 → lower case character set

Code	Set 1	Set 2	Code	Set 1	Set 2	Code	Set 1	Set 2
0	@		16	P	p	32	Space	
1	A	a	17	Q	q	33	!	
2	B	b	18	R	r	34	"	
3	C	c	19	S	s	35	#	
4	D	d	20	T	t	36	$	
5	E	e	21	U	u	37	%	
6	F	f	22	V	v	38	&	
7	G	g	23	W	w	39	'	
8	H	h	24	X	x	40	(	
9	I	i	25	Y	y	41	)	
10	J	j	26	Z	z	42	*	
11	K	k	27	[		43	+	
12	L	l	28	£		44	,	
13	M	m	29	]		45	-	
14	N	n	30	↑		46	.	
15	O	o	31	←		47	/	

Code	Set 1	Set 2	Code	Set 1	Set 2	Code	Set 1	Set 2
48	0		80		P	112		
49	1		81		Q	113		
50	2		82		R	114		
51	3		83	♥	S	115		
52	4		84		T	116		
53	5		85		U	117		
54	6		86		V	118		
55	7		87		W	119		
56	8		88	♣	X	120		
57	9		89		Y	121		
58	:		90	♦	Z	122		☑
59	;		91			123		
60	<		92			124		
61	=		93			125		
62	>		94	π		126		
63	?		95			127		
64			96	Space				
65	♠	A	97					
66		B	98					
67		C	99					
68		D	100					
69		E	101					
70		F	102					
71		G	103					
72		H	104					
73		I	105					
74		J	106					
75		K	107					
76		L	108					
77		M	109					
78		N	110					
79		O	111					

128-255 correspond to the reverse of characters 0-127.

Appendix D. Commodore BASIC Reserved Words

This appendix contains the Commodore BASIC reserved words along with their keyboard abbreviations, the screen display for these abbreviations.

Keyword	Keyboard Abbrev.	Screen Display	Keyword	Keyboard Abbrev.	Screen Display
ABS	A↑B	A □	INT	*	*
AND	A↑N	A □	LEFT$	LE↑F	LE □
ASC	A↑S	A ♥	LEN	*	*
ATN	A↑T	A □	LET	L↑E	L □
CHR$	C↑H	C □	LIST	L↑I	L □
CLOSE	CL↑O	CL □	LOAD	L↑O	L □
CLR	C↑L	C □	LOG	*	*
CMD	C↑M	C □	MID$	M↑I	M □
CONT	C↑O	C □	NEW	*	*
COS	*	*	NEXT	N↑E	N □
DATA	D↑A	D ♠	NOT	N↑O	N □
DEF	D↑E	D □	ON	*	*
DIM	D↑I	D □	OPEN	O↑P	O □
END	E↑N	E □	OR	*	*
EXP	E↑X	E ♣	PEEK	P↑E	P □
FN	*	*	POKE	P O	P □
FOR	F↑O	F □	POS	*	*
FRE	F↑R	F □	PRINT	?	?
GET	G↑E	G □	PRINT#	P↑R	P □
GET#	*	*	READ	R↑E	R □
GOSUB	GO↑S	GO ♥	REM	*	*
GOTO	G↑O	G □	RESTORE	RE↑S	RE ♥
IF	*	*	RETURN	RE↑T	RE □
INPUT	*	*	RIGHT$	R↑I	R □
INPUT#	I↑N	I □	RND	R↑N	R □

Keyword	Keyboard Abbrev.	Screen Display	Keyword	Keyboard Abbrev.	Screen Display
RUN	R↑U	R	SYS	S↑Y	S
SAVE	S↑A	S ♠	TAB	T↑A	T ♠
SGN	S↑G	S	TAN	*	*
SIN	S↑I	S	THEN	T↑H	T
SPC	S↑P	S	TO	*	*
SQR	S↑Q	S ●	USR	U↑S	U ♥
STEP	ST↑E	ST	VAL	V↑A	V ♠
STOP	S↑T	S	VERIFY	V↑E	V
STR$	ST↑R	ST	WAIT	W↑A	W ♠

↑ means press a Shift key

* means no abbreviation exists

Appendix E. Musical Notes

Each musical note generated by the Commodore 64 computer requires two bytes of data. This appendix includes the data necessary for the high register and low register for 8 octaves of notes.

Octave: 0

Note	High Byte	Low Byte
C	1	12
C#	1	28
D	1	45
D#	1	62
E	1	81
F	1	102
F#	1	123
G	1	145
G#	1	169
A	1	195
A#	1	221
B	1	250

Octave: 1

Note	High Byte	Low Byte
C	2	24
C#	2	56
D	2	90
D#	2	125
E	2	163
F	2	204
F#	2	246
G	3	35
G#	3	83
A	3	134
A#	3	187
B	3	224

Octave: 2

Note	High Byte	Low Byte
C	4	48
C#	4	112
D	4	180
D#	4	251
E	5	71
F	5	152
F#	5	237
G	6	71
G#	6	167
A	7	12
A#	7	119
B	7	223

Octave: 3

Note	High Byte	Low Byte
C	8	97
C#	8	225
D	9	104
D#	9	247
E	10	143
F	11	48
F#	11	218
G	12	143
G#	13	78
A	14	24
A#	14	239
B	15	210

Octave: 4

Note	High Byte	Low Byte
C	16	195
C#	17	195
D	18	209
D#	19	239
E	21	31
F	22	96
F#	23	181
G	25	30
G#	26	49
A	28	49
A#	29	223
B	31	165

Octave: 5

Note	High Byte	Low Byte
C	33	135
C#	35	134
D	37	162
D#	39	223
E	42	62
F	44	193
F#	47	107
G	50	60
G#	53	57
A	56	99
A#	59	190
B	63	75

Octave: 6

Note	High Byte	Low Byte
C	67	15
C#	71	12
D	75	69
D#	79	191
E	84	125
F	89	131
F#	94	214
G	100	121
G#	106	115
A	112	199
A#	119	124
B	126	151

Octave: 7

Note	High Byte	Low Byte
C	134	30
C#	142	24
D	150	139
D#	159	126
E	168	250
F	179	6
F#	189	172
G	200	243
G#	212	230
a	225	143
A#	238	248
B	253	46

Appendix F. Commodore 64 Memory Map

The Commodore 64 has three types of memory: RAM, ROM, and Input/Output (I/O). Illustration F-1 depicts a typical Commodore 64 memory configuration, and Table F-1 lists the reference for each memory location.

Illustration F-1. Typical Commodore 64 Memory Configuration

Memory	Address
8K KERNAL* ROM	E000-FFFF
4K I/O or CHARACTER ROM	D000-DFFF
4K RAM	C000-CFFF
8K BASIC ROM	A000-BFFF
8K ROM PLUG IN	8000-9FFF
16K RAM	4000-7FFF
16K RAM	0000-3FFF

* KERNAL is the part of the operating system that controls input, output, and memory management in the Commodore 64.

Table F-1. Commodore 64 Memory Map

Memory Address		Reference
Hexadecimal	Decimal	
0000	0	Data Direction Register
0001-0002	1-2	I/O Register
0003-0004	3-4	Float Fixed Vector
0005-0006	5-6	Fixed Float Vector
0007	7	Search Character
0008	8	Scan - quotes Flag
0009	9	TAB Column Save
000A	10	0 = LOAD; 1 = VERIFY
000B	11	Input buffer pointer # subscript
000C	12	Default DIM Flag
000D	13	Type: 00 = Numeric; FF = String
000E	14	Type: 00 = Floating Point; 80 = Integer
000F	15	DATA Scan; LIST Quote; Memory Flag
0010	16	Subscript/FN x Flag
0011	17	$00 = INPUT;$40 = GET;$98 = READ
0012	18	TAN Sign; Comparison Flag
0013	19	Present I/O Prompt Flag
*0014-0015	20-21	Integer Value
0016	22	Pointer--Temporary Stack String
0017-0018	23-24	Most Recent Temp String Vector
0019-0021	25-33	Temporary String Stack
0022-0025	34-37	Utility Pointer Area
0026-002A	38-42	Multiplication Product Area
*002B-002C	43-44	Pointer: Start of BASIC
*002D-002E	45-46	Pointer: Start of Variables
*002F-0030	47-48	Pointer: Start of Arrays
*0031-0032	49-50	Pointer: End of Arrays
*0033-0034	51-52	Pointer: String Storage
0035-0036	53-54	Utility String Pointer
*0037-0038	55-56	Pointer: Memory Limit
0039-003A	57-58	Current BASIC Line Number
003B-003C	59-60	Last BASIC Line Number
003D-003E	61-62	Pointer: BASIC Statement for CONT
003F-0040	63-64	Current DATA Line Number
0041-0042	65-66	Current DATA Address
*0043-0044	67-68	Input Vector
0045-0046	69-70	Current Variable Name
0047-0048	71-72	Current Variable Address
0049-004A	73-74	FOR/NEXT Variable Pointer
004B-004C	75-76	Y = Save; op-save; BASIC Pointer save

* Useful Memory Locations

Memory Address		Reference
Hexadecimal	Decimal	
004D	77	Accumulator--Comparison Accumulators
004E-0053	78-83	Misc. Work Area
0054-0056	84-86	Jump Vector For Functions
0057-0060	87-96	Misc. Numeric Work Area
*0061	97	Accumulator #1: Exponent
*0062-0065	98-101	Accumulator #1: Mantissa
*0066	102	Accumulator #1: Sign
0067	103	Series Evaluation Constant Pointer
0068	104	Accumulator #1: High Order
*0069-006E	105-110	Accumulator #2: Exponent, etc.
006F	111	Sign Comparison--Accum. #1 vs. #2
0070	112	Accumulator #1: Low Order
0071-0072	113-114	Cassette Buffer Length--Series Pointer
*0073-008A	115-138	Subroutine To Get Next Character
007A-007B	122-123	BASIC Pointer
008B-008F	139-143	RND Seed
*0090	144	ST - Status Word
0091	145	Keyswitch PIA: STOP & RVS Flag
0092	146	Timing Constant For Tape Unit
0093	147	Load = 0; Verify = 1
0094	148	Serial Output; Deferred Character Flag
0095	149	Serial Deferred Character
0096	150	Cassette Sync Number
0097	151	Register Save
*0098	152	Number of Open Files
*0099	153	Input Device
*009A	154	Output Device (CMD)
009B	155	Tape Character Parity
009C	156	Flag: Byte Received
009D	157	Direct=$80/RUN=$00 Output Control
009E	158	Tape Pass 1 Error Log/Char Buffer
009F	159	Tape Pass 2 Error Log Corrected
*00A0-00A2	160-162	Jiffy Clock
00A3	163	Serial Bit Count/EOI Flag
00A4	164	Cycle Count
00A5	165	Countdown; Tape Write Bit Cnt.
00A6	166	Pointer: Tape Buffer
00A7	167	RS-232 Input Bits
00A8	168	RS-232 Input Bit Counter
00A9	169	RS-232 Start Bit Flag
00AA	170	RS-232 Input Buffer

* Useful Memory Locations

Memory Address		Reference
Hexadecimal	Decimal	
00AB	171	RS-232 Input Parity
00AC-00AD	172-173	Pointer: Tape Buffer, Scrolling
00AE-00AF	174-175	Tape End Addresses/End of Program
00B0-00B1	176-177	Tape Timing Constants
*00B2-00B3	178-179	Pointer: Start of Tape Buffer
00B4	180	Tape Time (1 = Enable)/RS-232 Output Bit Counter.
00B5	181	Tape EOT/RS-232 Bit to Send
00B6	182	Read Character Error/RS-232 Output Byte Buffer
*00B7	183	No. of Characters in Filename
*00B8	184	Current Logical File
*00B9	185	Current Secondary Address
*00BA	186	Current Device
*00BB-00BC	187-188	Filename Pointer
00BD	189	RS-232 Output Parity
00BE	190	#Blocks to Write/Read
00BF	191	Serial Word Buffer
00C0	192	Tape Motor Interlock
00C1-00C2	193-194	I/O Start Address
00C3-00C4	195-196	Tape Load Register
*00C5	197	Current Key Depressed
*00C6	198	No. of Characters in Keyboard Buffer
*00C7	199	Screen Reverse Flag
00C8	200	Pointer: End-of-Line For Input
00C9-00CA	201-202	Input Cursor Log (Row,Column)
*00CB	203	Print Shifted Characters
00CC	204	Cursor Enable (0 = Flashing)
00CD	205	Cursor Timing Countdown
00CE	206	Character Beneath Cursor
00CF	207	Cursor in Blink Phase
00D0	208	Input From Screen/From Keyboard
*00D1-00D2	209-210	Pointer to Screen Line
*00D3	211	Position of Cursor on Screen Line
00D4	212	Quote Mode Flag for Editor
*00D5	213	Current Screen Line Length
*00D6	214	Cursor Row
00D7	215	Last Inkey/Checksum/Buffer
*00D8	216	# of INSERTS outstanding
*00D9-00F0	217-240	Screen Line Link Table
*00F3-00F4	243-244	Screen Color Pointer
00F5-00F6	245-246	Keyboard Decoder Pointer
00F7-00F8	247-248	RS-232 Rcv. Pointer
00F9-00FA	249-250	RS-232 TX Pointer

* Useful Memory Locations

Memory Address		Reference
Hexadecimal	Decimal	
*00FB-00FE	251-254	Operating System Free Zero Page Space
00FF	255	BASIC Storage
0100-010A	256-266	Work Area Floating to ASCII
0100-013E	256-318	Tape Error Log
0100-01FF	256-511	Processor Stack Area
*0200-0258	512-600	BASIC Input Buffer
*0259-0262	601-610	Logical File Table
*0263-026C	611-620	Device Number Table
*026D-0276	621-630	Secondary Address Table
*0277-0280	631-640	Keyboard Buffer
*0281-0282	641-642	Start of Memory for Oper. System
*0283-0284	643-644	Top of Memory for Oper. System
0285	645	Serial Bus Timeout Flag
*0286	646	Current Color Code
0287	647	Color Under Cursor
*0288	648	Screen Memory Page
*0289	649	Keyboard Buffer Maximum Size
*028A	650	Key Repeat (128 = Repeat All Keys)
*028B	651	Repeat Speed Counter
028C	652	Repeat Delay Counter
*028D	653	Keyboard Shift/Control Flag
028E	654	Last Keyboard Shift Pattern
028F-0290	655-656	Keyboard Table Pointer
0291	657	Shift Mode Switch (0 = Enabled; 128 = Locked)
0292	658	Auto Scroll Down Flag (0 = On; Non-Zero = Off)
0293	659	RS-232 Control Register
0294	660	RS-232 Command Register
0295-0296	661-662	Non-Standard (Bit Time/2-100)
0297	663	RS-232 Status Register
0298	664	No. of Bits to Send
0299-029A	665-666	Baud Rate Bit Time
029B	667	RS-232 Receive Pointer
029C	668	RS-232 Input Pointer
029D	669	RS-232 Transmit Pointer
029E	670	RS-232 Output Pointer
029F-20A0	671-672	Store IRQ During Tape Operations
02A1-02FF	673-767	Program Indirects
0300-0301	768-769	Error Message Link
0302-0303	770-771	BASIC Warm Start Link
0304-0305	772-773	Crunch BASIC Tokens Link

* Useful Memory Locations

Memory Address		Reference
Hexadecimal	Decimal	
0306-0307	774-775	Print Tokens Link
0308-0309	776-777	Start New BASIC Code Link
030A-030B	778-779	Token Evaluation
030C	780	Storage for 6502 .A Register
030D	781	Storage for 6502 .X Register
030E	782	Storage for 6502 .Y Register
030F	783	Storage for 6502 .SP Register
0310-0313	784-787	USR Function Address
0314-0315	788-789	Hardware (IRQ) Interrupt Vector
0316-0317	790-791	Break Interrupt Vector
0318-0319	792-793	NMI Interrupt Vector
031A-031B	794-795	OPEN Vector
031C-031D	796-797	CLOSE Vector
031E-031F	798-799	Set-Input Vector
0320-0321	800-801	Set-Output Vector
0322-0323	802-803	Restore I/O Vector
0324-0325	804-805	INPUT Vector
0326-0327	806-807	Output Vector
0328-0329	808-809	Test-STOP Vector
032A-032B	810-811	GET Vector
032C-032D	812-813	Abort I/O Vector
032E-032F	814-815	User Vector
0330-0331	816-817	Link to Load RAM
0332-0333	818-819	Link to Save RAM
0334-033B	820-827	Unused
*033C-03FB	828-1019	Cassette Buffer
03FC-03FF	1020-1023	Unused
0400-07E7	1024-2023	Screen Memory
07E8-07FF	2024-2047	Sprite Pointers
0800-7FFF	2048-32767	BASIC Program Area
8000-9FFF	32678-40959	Cartridge ROM
A000-BFFF	40960-49151	BASIC ROM
C000-CFFF	49152-53247	RAM
D000-D02E	53248-54271	Video Interface Controller
D400-D7FF	54272-55295	Sound Interface Device
D500-D7FF	54528-55295	SID Images
D800-DBFF	55296-56319	Color RAM
DC00-DCFF	56320-56575	Complex Interface Adapter #1
DD00-DDFF	56576-56831	Complex Interface Adapter #2
DE00-DFFF	56832-57342	Reserved for future use
E000-FFFF	57343-65535	KERNAL ROM

* Useful Memory Locations

Appendix G. Useful Commodore 64 Memory Locations

Table G-1. Useful Memory Locations

Memory Address		Reference
Hexadecimal	Decimal	
0014-0015	20-21	BASIC stores integer variables used in calculations.
002B-002C	43-44	Beginning of BASIC program in memory. Address 43 contains the low byte; 44 contains the high byte. Use PEEK(43) + PEEK(44) * 256 to compute the start of BASIC in decimal.
002D-002E	45-46	Numeric variables begin at this address.
002F-0030	47-48	Arrays begin at this address.
0031-0032	49-50	Ending address of arrays.
0033-0034	51-52	Bottom of string storage. Storage progresses from top of available memory down to the top of arrays.
0037-0038	55-56	Top of free RAM. If this address is lowered, some RAM can be preserved by keeping BASIC data from being stored in this area.
0043-0044	67-68	INPUT statement Jump Vector.
0061-0066	97-102	Floating Point Accumulator #1.
0069-006E	105-110	Floating Point Accumulator #2.
0073-008A	115-138	CHRGET subroutine stored at these addresses. This subroutine retrieves the next BASIC character from machine language.

Memory Address		Reference
Hexadecimal	Decimal	
0090	144	Status word ST.
0098	152	Number of open files.
0099	153	Input device number. Generally 0 (keyboard).
009A	154	Output (CMD) device. Generally 3 (screen).
00A0-00A2	160-162	Jiffy clock. TI and TI$ return values stored at these addresses.
00B2-00B3	178-179	Pointer to beginning of the tape buffer.
00B7	183	Number of characters in the file-name.
00B9	185	Secondary address currently in use.
00BA	186	Device number currently being accessed.
00BB-00BC	187-188	Pointer to location of filename in memory.
00C5	197	Number corresponding to the key currently depressed. A value of 64 indicates no key is depressed. The values for the various keys are given in Table G-2. If more than one key is depressed, the highest key value will appear in this address.
00C6	198	Number of characters in keyboard buffer.
00C7	199	Reverse on/off flag (1 = on; 0 = off).
00D1-00D2	209-210	Pointer to screen line.

Memory Address		Reference
Hexadecimal	Decimal	
00D5	213	Current screen line length (can be 21, 43, 65, or 87).
00D6	214	Current screen row where cursor is located.
00D8	216	Remaining number of spaces in the INSERT mode. If a 0 is POKE'd to this address, the INSERT mode will be turned off.
00D9-00F0	217-240	Screen line link table. A value of 158 indicates the line finishes at its end. A value of 30 indicates that the line continues to the next line.
00F3-00F4	243-244	Pointer to the current space in color memory.
00FB-00FE	251-254	Page zero available memory addresses.
0200-0258	512-600	INPUT statement buffer.
0259-0262	601-610	Logic 1 file table for files currently open.
0263-026C	611-620	Device number table for open files.
026D-0276	621-630	Secondary address table.
0277-0280	631-640	Keyboard buffer.
0283-0284	643-644	Pointer to the top of the memory for the operating system.
0286	646	Color code in effect. This will be the color number placed in color memory during PRINT operations.
0288	648	Screen memory page.

Memory Address		Reference
Hexadecimal	Decimal	
0289	649	Sets maximum keyboard buffer size. If this is set greater than 10, pointers may be erased.
028A	650	Flag for keyboard repeat (0 = only cursor repeats; 128 = all keys repeat)
028B	651	Length of delays before repeating a key.
028D	653	Keyboard shift, Commodore and Control key flags. Pressing shift sets the 1 bit; Commodore sets the 2 bit; and Control sets the 4 bit.
033C-03FB	828-1019	Cassette buffer. When this area is not used for inputting, it can be used for POKE'ing data or for machine language program storage.

Table G-2. Location 197 Depressed Key Values

Key	Value	Key	Value	Key	Value
0	1	25	none	50	T
1	3	26	X	51	U
2	5	27	V	52	O
3	7	28	N	53	@
4	9	29	,	54	↑
5	+	30	/	55	f5
6	£	31	CRSR↑↓	56	2
7	DEL	32	space	57	4
8	←	33	Z	58	6
9	W	34	C	58	8
10	R	35	B	60	0
11	Y	36	M	61	—
12	I	37	.	62	HOME
13	P	38	none	63	f7
14	*	39	f1		
15	RETURN	40	none		
16	none	41	S		
17	A	42	F		
18	D	43	H		
19	G	44	K		
20	J	45	:		
21	L	46	=		
22	;	47	f3		
23	CRSR ⇄	48	Q		
24	STOP	49	E		

Appendix H. BASIC Keyword One Character Tokens

In order to save memory space, BASIC keywords are stored as one character tokens. These tokens are listed in this appendix.

<table>
<tr><th>Keyword</th><th>Code</th><th>Keyword</th><th>Code</th><th>Keyword</th><th>Code</th></tr>
<tr><td>END</td><td>128</td><td>PRINT</td><td>153</td><td><</td><td>179</td></tr>
<tr><td>FOR</td><td>129</td><td>CONT</td><td>154</td><td>SGN</td><td>180</td></tr>
<tr><td>NEXT</td><td>130</td><td>LIST</td><td>155</td><td>INT</td><td>181</td></tr>
<tr><td>DATA</td><td>131</td><td>CLR</td><td>156</td><td>ABS</td><td>182</td></tr>
<tr><td>INPUT#</td><td>132</td><td>CMD</td><td>157</td><td>USR</td><td>183</td></tr>
<tr><td>INPUT</td><td>133</td><td>SYS</td><td>158</td><td>FRE</td><td>184</td></tr>
<tr><td>DIM</td><td>134</td><td>OPEN</td><td>159</td><td>POS</td><td>185</td></tr>
<tr><td>READ</td><td>135</td><td>CLOSE</td><td>160</td><td>SQR</td><td>186</td></tr>
<tr><td>LET</td><td>136</td><td>GET</td><td>161</td><td>RND</td><td>187</td></tr>
<tr><td>GOTO</td><td>137</td><td>NEW</td><td>162</td><td>LOG</td><td>188</td></tr>
<tr><td>RUN</td><td>138</td><td>TAB(</td><td>163</td><td>EXP</td><td>189</td></tr>
<tr><td>IF</td><td>139</td><td>TO</td><td>164</td><td>COS</td><td>190</td></tr>
<tr><td>RESTORE</td><td>140</td><td>FN</td><td>165</td><td>SIN</td><td>191</td></tr>
<tr><td>GOSUB</td><td>141</td><td>SPC(</td><td>166</td><td>TAN</td><td>192</td></tr>
<tr><td>RETURN</td><td>142</td><td>THEN</td><td>167</td><td>ATN</td><td>193</td></tr>
<tr><td>REM</td><td>143</td><td>NOT</td><td>168</td><td>PEEK</td><td>194</td></tr>
<tr><td>STOP</td><td>144</td><td>STEP</td><td>169</td><td>LEN</td><td>195</td></tr>
<tr><td>ON</td><td>145</td><td>+</td><td>170</td><td>STR$</td><td>196</td></tr>
<tr><td>WAIT</td><td>146</td><td>-</td><td>171</td><td>VAL</td><td>197</td></tr>
<tr><td>LOAD</td><td>147</td><td>.</td><td>172</td><td>ASC</td><td>198</td></tr>
<tr><td>SAVE</td><td>148</td><td>/</td><td>173</td><td>CHR$</td><td>199</td></tr>
<tr><td>VERIFY</td><td>149</td><td>|</td><td>174</td><td>LEFT$</td><td>200</td></tr>
<tr><td>DEF</td><td>150</td><td>AND</td><td>175</td><td>RIGHT$</td><td>201</td></tr>
<tr><td>POKE</td><td>151</td><td>OR</td><td>176</td><td>MID$</td><td>202</td></tr>
<tr><td>PRINT#</td><td>152</td><td>></td><td>177</td><td>Unused</td><td>203-254</td></tr>
<tr><td></td><td></td><td>=</td><td>178</td><td>π</td><td>255</td></tr>
</table>

INDEX

Abbreviation of Keywords 44, **284**
ABS Function 87
AC Power Adapter **11**, 13
Addition 58
AND 86
Applications Program 18
Arctangent 88
Arrays 53, 94
Arrays, DIM Statement 54, **94**
Arrays, Sze 54
ASC Function 87
ASCII 60, 77, 87
Assignment Statement 64
ATN Function 88
Attack 260, **261**
Audio, Video Output Jack 21
Audio Output 10, **21**

Background Color 23
BAM 172
Binary Values 83
Bit Binary Values 83
Bit Maps 244
Bit Memory Location 246
Bits 82
Block Availability Map - See BAM
Boolean Operators 60
Border Color 235
Branching Sttements **72**, 105, 106
Byte 12, **82**

Calculator Mode - See Immediate Mode
Carriage, Printer 215
Carriage Return 226
Carriage Return/Line Feed 67
Cartridge Slot 10
Cassette, Write Protection 32
Cassette Interface 11
Cassette Recorder - See Datassette
Cassette Tape 14
Cassette, Packaged Programs - See Datassette
Channel Selector 10, **20**
Character Memory Location 238
Character Set 241
Character Width 226
Characters, Changing 241
Characters, Designing 241
Characters, Moving 241
Characters, Special 224
CHR$ Function **78**, 88
CHR$(8) 230
CHR$(13) 226

Note to Reader: Please note that for selected topics with several page references, one reference may be in bold type. The page noted in bold denotes the primary reference or definition of that topic.

CHR$(17) 228
CHR$(145) 232
CLOSE Statement 89
CLR Command 89
CMD Statement 90
Color 127, 235
Color, Background 235
Color, Border 235
Color, Text 236
Color Adjustment 24
Color Memory Location 236
Color Values 236
Commodore Key 27
Compiled Code 39
Compiled Language 39
Compiler 39
Constants 50
CONT Command 91
COS Function 91
Cosine 91
CP/M 17
CR as a Delimiter 193
Cursor Control Characters 126
CURSOR DOWN 228
CURSOR UP 232

Data Files 145
Data Files, Closing 152
Data Files, Opening 150
Data Output 66
DATA, READ Statement 64
DATA Statement 92
Data Types, BASIC 46
Datassette 14, 15, 31, **143**
Datassette, Closing Data Files 152
Datassette, Installation 143
Datassette, Loading a Program File 155
Datassette, Verifying a Program File 147
Decay 260, **261**
Decimal Point 47
Decimal Point, Floating 47
Decimal Point, Functions & Positions 48
DEF FN 93
Device Code 119
Device Numbers 80
DIM Statement 54, 55, **94**
Dimension Statement - See DIM Statement
DIN Cable 22
Direct Mode - See Immediate Mode
Directory Format 176
Directory Header 175
Disk 157
Disk, Hard 158
Disk, Winchester 158
Disk Controller BLOCK-EXECUTE 209
Disk Controller MEMORY-EXECUTE 210
Disk Controller MEMORY-WRITE 210
Disk Controller Programming 209
Disk Controller USER Commands 211
Disk Directory 174
Disk Drive 14, 15, 33, **157**
Disk Drive, Installation 166
Disk Drive, Powering On 168
Disk Files 170
Disk Operating System - See DOS
Diskette, Closing Files 189
Diskette, DOS Commands - See DOS Commands
Diskette, Erasing Files 187
Diskette, Errors 188
Diskette, Floppy 158
Diskette, Formatting 36, **177-178**
Diskette, Handling 33
Diskette, Insertion 35, 38, **169**
Diskette, Loading a Program File 182
Diskette, Packaged Programs 37, 177
Diskette, Replacing a Program File 179
Diskette, Single and Double Sided 164
Diskette, Utilizing Space 188
Diskette, VERIFY 183
Diskette, Write Protection 35, **164**
Display Format 124-127
Display Format, Columns 138

Display Format, Spaces 135
Display Line 43
Display Memory Grid 239
Display Screen **21**, 31
Display Zones 66
Division 58
DOS 35
DOS Commands 35, **184**
DOS Commands, CLOSE 189
DOS Commands, COPY 186
DOS Commands, INITIALIZE 188
DOS Commands, NEW 185
DOS Commands, OPEN 184
DOS Commands, PRINT# 164
DOS Commands, SCRATCH# 188
DOS Commands, VALIDATE 188
Dot Address 231
Duty Cycle 259-260

END Statement 43, **95**
Envelope 260
Error Channel 189
Errors, Correction of Entry 31
Errors, Messages 31, 46, **267**
Execution Sequence 41
EXP Function 96
Exponentiation 58
Expressions 55
Expressions, Arithmetic 58
Expressions, Compound 56
Expressions, Order of Evaluation 56, 57
Expressions, Parentheses within 56
Expressions, Simple 56
Expressions, Types 55

Fields 145
File Access 190
File Type 171
Filename Match Characters 171
Filenames 170
Files **145**, 170
Files, Data 145
Files, Program 145
Floating Decimal Point 47
Floppy Diskette 158
FN 96
FOR, NEXT 70, **97-100**, 103, 105, 116
FRE Function 100
Frequency 258
Function Definition 93
Functions 76

GET# 81, 155
GOSUB 72, 75, **102**, 130
GOTO 72, **104**
Graphics 235-256
Graphics, Characters 236
Graphics, High Resolution 244
Graphics, POKE Statements 240
Graphics, PRINT Statements 237
Graphics, Screen Locations 238

Home 127

I/O Device 119
IF, THEN 72, **106**, 118, 120
Immediate Mode 40
Index Variable 71
Indirect Mode - See Program Mode
INPUT Statement 67, **107**
INPUT# 81, 155
Input/Output Channel **81**, 119
Installation 19
INT 109
Integers 48
Interface Cartridge **16**, 17
Interpreted Language 39
Interpreter **17**

Keyboard 10, **25**
Keyboard, CLR HOME 29
Keyboard, Color Keys 30
Keyboard, Commodore Key 27
Keyboard, CTRL 29
Keyboard, Cursor Right/Left 28
Keyboard, Cursor Up/Down 28
Keyboard, Functions Keys 30
Keyboard, INST DEL 29
Keyboard, RESTORE 26
Keyboard, RETURN 26
Keyboard, RUN STOP 30
Keyboard, RVS OFF 30
Keyboard, SHIFT 27
Keyword 44

Keyword, Abbreviations 44, **284**
Kilobyte 12

Language, Compiled 39
Language, Interpreted 39
Languages 17
LEFT$ Statement 109
LEN 110
LET Statement 51, **111**
Line Numbers 41
LIST Command 45, **111**
LOAD 37, **112**, 149
Loading 143, **149**
LOG 114
Logarithm 114
Logical Negation 117
Logical Operations - See Boolean Operations
Loop **70**, 97-100, 103, 105, 116
Loops, Nested 71
Lower Case Characters 28

Memory 12, 122
MEMORY-EXECUTE 210
Memory Location 137, **294**
MEMORY-READ 209
MEMORY-WRITE 210
MID$ Statement 115
Mode Selection Switch 217
Modem 16
Multiple Statement Program Lines 43
Multiplication 58
Music 264
Music, Program Examples 265

Nested Loops 71
NEW Command 42, **116**
NEXT Statement 116
NOT Statement 117
Numbers, Floating Point 48
Numbers, Negative 48
Numbers, Outer/Inner Limits 50
Numeric Data 46, **47**

ON, GOSUB 75, 103, 130
ON, GOTO 74
On/Off Switch 10, 19
OPEN Statement 81, **119**, 125, 150, 184
Operands 56
Operation, Boolean 60
Operation, Unary 58
Operation Code 120
Operators 55
Operators, Arithmetic 58
Operators, Logical **60**, 106
Operators, Relational 59
OR 120
Output **66**, 124, 126
Output Format **124**, 126, 127
Output Format Column 138
Output Format Spaces 135

Packaged Programs, Cassette - See Datassette
Packaged Programs, Diskette - See Diskette
Paper **216**, 217
PEEK 79, **122**
Pipeline Operating System 165
POKE 79, **122**
POKE Statement, Graphics 240
POS Function 123
Power Cord Socket 10, 19
Power Indicator 10
Power Supply Unit 19
Power Switch, Disk Drive 168
Power Switch, Commodore 64 - See On/Off Switch
Power Switch, Printer 217
Print Format, Column 138
Print Format, Spaces 135
Print Head Pressure 217
Print Parameters 67
PRINT Statement 66, **124**
PRINT# Statement 81, 152, **185**, 221
Printer 16, 336
Printer, BASIC Statements 218
Printer, Carriage 215
Printer, CLOSE 219
Printer, CMD 220
Printer, Control Codes 223
Printer, Graphics Mode **224**, 232
Printer, Impression Control Lever 217

Printer, Line Feed CHR$(10) 225
Printer, Mode Argument 218
Printer, OPEN 218
Printer, Power Switch 217
Printer, Print Format 221-227
Printer, PRINT# 221
Printer, Ribbon Cartridges 214-216
Printer, Test Procedure 217
Printer, Upper & Lower Case Letters 219, **228**, 232
Printer, 1525 Installation 213
Program Files **145**
Program Lines 41, 43
Program Lines, Addition of 42
Program Lines, Changing 42
Program Lines, Deletion of **42**, 116
Program Lines, Multiple Statement 43
Program Mode 40
Program Name 113
Programs, Erasing 42
Programs, Executing 36, **43**
Programs, Listing 45
Programs, Loading 36
Programs, Packaged 37
Programs, Recording on Cassette or Diskette 133
Programs, Recovering from Cassette or Diskette 134
Programs, Saving 143, **146**
Prompt Message 69
Prompts, Datassette 145
Pulsewave 259

RAM 13
Random Access **157**, 172
Random Access Files 195
Random Access Memory - See RAM
Random Files 195
Random Files, BLOCK-ALLOCATE 199
Random Files, BLOCK-FREE 200
Random Files, BLOCK-READ 198
Random Files, BLOCK-WRITE 196
Random Files, BUFFER-POINTER 201
Random Files, OPEN 196
Random Files, USER1 202
Random Files, USER2 203
Random Numbers 131
Read Only Memory - See ROM
READ Statement 128
Read/Write Head 157
Reading Data 143, **155**
Records 145
Relative Files 203
Relative Files, Filepointer 206
Relative Files, FORMAT 204
Relative Files, OPEN 205
Relative Files, Opening 205
Relative Files, Reading from 208
Relative Files, Writing to 207
Release 261, **262**
REM 64, **129**
Remark Statements 63, **129**
Repeat CHR$(26) 229-230
Repetitions of Patterns 229-230
Reserved Word **44**, 51, 284
RESTORE 129
Retrieval - See Reading Data
RETURN Statement 130
Reverse CHR$(18) 229
Reverse Mode 229
Reverse Off CHR$(146) 233
RF Modulator 12, 21
Ribbon Cartridges 214-216
RIGHT$ Statement 130
RND Function 131
ROM 13, 18
RUN Statement 132
Runtime Monitor 39

SAVE Statement **133**, 146
Saving 143
Scientific Notation 49
Screen Display 21, 31
Sectors 160-161
Sectors, Hard & Soft 162
Sequential Access **157**, 172, 190
Sequential Files 190
Sequential Files, GET# 194
Sequential Files, Opening 190
Sequential Files, PRINT# 192
Sequential Files, Types 191
Serial Bus 166

Serial Port 11
SGN Function 134
Side Sectors 203
SIN Function 134
Sine 134
Software 17-18
Sound 82, **256**
Sound Program Examples **259**, 263, 265
Source Code 39
SPC 135
Special Characters 224
Specifications 11
Sprites 250
Sprites, Activating 252
Sprites, Color 253
Sprites, Controlling 251
Sprites, Data Assignments 253
Sprites, Definition 250
Sprites, Example Program 255
Sprites, Expanding 253
Sprites, Locating 252
Sprites, Registers
SQR 135
Square Root 135
ST 208
Standard CHR$(15) 226
Standard Output 226
Statement, Assignment 64
Statement, Branching 72
Statement, CLOSE 89
Statement, CMD 90
Statement, Conditional 72
Statement, Contents of 43
Statement, DATA 92
Statement, Definition of 43
Statement, DIM 94
Statement, INPUT **68**, 80
Statement, Multiple 44
Statement, ON, GOSUB 75, **102**, 130
Statement, ON, GOTO 74
Statement, Subroutines & GOSUB 74
Statement Parameters 44
Statements, Advanced Input and Output 80
STEP 98, **117**
STOP Statement 136
Storing Programs 143, **146**
STR$ 136
String Concatenation 77
String Variables 52
Strings **46**, 60, 115
Subroutine Machine Language 138
Subroutines **74**, 102
Subtraction 58
Sustain 260, **262**
SYS 137

TAB 137
Tab CHR$(16) 226-228
Tab Control 226
Tables 53
Text Colors 236
Text Data 46
Text Mode 28
Tracks 160, 161
Troubleshooting 23
TV Connection 10, 11, 13, **19**

Unary Operation 58
User Port 11
USR Function 138

VAL Function 139
Variable Index 71
Variable Names **51**, 53
Variable Names, Floating Point 52
Variable Names, Integer 52
Variable Names, Invalid 52
Variable Names, Numeric 52
Variable Names, String 52
Variable Names, Valid 52
Variable Type 53
Variables **50**, 53
Variables, Subscripted 53
VERIFY **139**, 147, 182
VIC Modem - See Modem
VIC 1540 Disk Drive - See Disk Drive
VIC 1541 Disk Drive - See Disk Drive
Video Cable 11, 12, 13, **19**, 20
Video Display, Color Adjustments 29

Video Output 10
Voices 256

WAIT Statement 140
Waveshapes 257-258
Wide CHR$(14) 226
Write Protection 32, 35, 164
Write Protection, Diskette - See
Diskette
Writing Data 143, 152

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

The Commodore 64 computer has impressive computing, graphics, and sound-generating capabilities. The COMMODORE 64 USER'S HANDBOOK provides clear, concise, and complete instructions which allow the user to master these capabilities.

The COMMODORE 64 USER'S HANDBOOK is written in simple, concise language that even a first-time user can understand, yet it still contains enough information to be a useful guide for an experienced user. A complete guide to the set-up, operation, and programming of the Commodore 64 is included.

The following topics are covered in detail:

- Installation of the Commodore 64
- BASIC Programming Techniques
- Reference Guide to BASIC Commands & Functions
- Using the Datassette Cassette Tape Recorder
- Using the 1540 and 1541 Disk Drives
- Using the 1515 and 1525 Printers
- Programming Sound and Graphics

The COMMODORE 64 USER'S HANDBOOK is a must for any Commodore 64 user or potential user.

ISBN 0-345-31596-0

Cover printed in USA