COMMODORE SX-64 PORTABLE COLOR COMPUTER

Commodore's SX-64 computer is portable, powerful, and versatile. The SX-64 has all the capabilities of the best-selling Commodore 64, plus a built-in disk drive and a 5" color monitor. The SX-64 snaps together to form its own carrying case, and it's light enough to carry and use anywhere.

Like the Commodore 64, the SX-64 has 64K memory, 16 colors, a sound chip that lets you play music on your computer, and a video chip that lets you create complex graphics. In addition, all software designed on disk and cartridge for the Commodore 64 runs on the SX-64.

This easy-to-read user's guide contains all the information you need to set up your equipment, understand how to operate your new SX-64, and learn how to create simple BASIC programs. Technical information for beginners as well as experienced programmers is also included.

For additional programming information, consult the Commodore 64 Programmer's Reference Guide and the Commodore Peripherals Guide, available from your bookstore or Commodore dealer.

(C: commodore
COMPUTER

COMMODORE PORTABLE SX-64 COLOR COMPUTER

A Friendly Introduction to Your SX-64 Portable Color Computer

USER'S GUIDE

(C: commodore
COMPUTER

USER'S GUIDE STATEMENT

"This equipment generates and uses radio frequency energy. If it is not properly installed and used in strict accordance with the manufacturer's instructions, this equipment may interfere with radio and television reception. This machine has been tested and found to comply with the limits for a Class B computing device in accordance with the specifications in Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference in a residential installation. If you suspect interference, you can test this equipment by turning it off and on. If you determine that there is interference with radio or television reception, try one or more of the following measures to correct it:

- reorient the receiving antenna
- move the computer away from the receiver
- change the relative positions of the computer equipment and the receiver
- plug the computer into a different outlet so that the computer and the receiver are on different branch circuits.

If necessary, consult your Commodore dealer or an experienced radio/television technician for additional suggestions. You may also wish to consult the following booklet, which was prepared by the Federal Communications Commission:

"How to Identify and Resolve Radio-TV Interference Problems" This booklet is available from the U.S. Government Printing Office, Washington, D.C. 20402, Stock No. 004-000-00345-4."

COMMODORE PORTABLE SX-64 COLOR COMPUTER USER'S GUIDE

A Friendly Introduction to Your SX-64 Portable Color Computer

(: commodore
COMPUTER

TABLE OF CONTENTS

CHAPTER 1
SETTING UP

CHAPTER 2
GETTING STARTED

CHAPTER 3
BEGINNING BASIC

CHAPTER 4
WRITING SIMPLE PROGRAMS IN BASIC

INTRODUCTION

The Commodore Executive 64 is one of the best values in the home computer industry. You can use your 64K color computer for everything from business applications to household paperwork to exciting games. The 64 offers you lots of memory (64K), lots of color (16 different colors), lots of sound (music and sound effects), and lots of fun and practical uses. You can use prepackaged software, or you can write your own programs in easy-to-learn BASIC. All software designed for the Commodore 64 on disk or cartridge also runs on the Executive 64.

The Executive 64 models (the SX-64 and the DX-64) have all the capabilities of the best selling Commodore 64, plus one or two disk drives and a 5'' color monitor built-in. The Executive 64 snaps together to form its own carrying case, and it's light enough to carry and use anywhere.

This easy-to-read user's guide contains all the information you need to set up your equipment properly, understand how to operate your new SX-64 or DX-64, and learn how to create your own simple BASIC programs.

This user's guide is intended to introduce you to computers, but it is beyond the scope of this manual to tell you everything you need to know about computers or about BASIC. However, this guide does refer you to a variety of publications that explain the topics we present here in more detail. These publications include the Commodore 64 Programmer's Reference Guide and our Introduction to BASIC series.

For those of you who don't want to learn how to program, you won't have to search through the whole book to learn how to use Commodore prepackaged programs and games, or other prepackaged, third party software. We've put all the information you need to know right up front in Chapters 1 and 2.

SPRITE GRAPHICS

Many exciting features are waiting for you inside your Executive 64. Your new computer gives you the microcomputer industry's most advanced graphics, which we call SPRITE GRAPHICS. Sprite graphics let you:

- Design your own pictures in different colors, just like the ones you see on arcade-type video games.
- Animate as many as 8 different overlapping sprites at once.
- Double a sprite's size.
- Move your creations anywhere on the screen.
- Pass images in front or behind each other.
- Use automatic collision detection that tells the computer to do whatever you want when sprites touch each other.

You can use these features to write games and educational software. In addition, you'll see sprites used in the preprogrammed software you buy from Commodore.

MUSIC SYNTHESIS

The Executive 64 also has built-in music and sound effects that rival many well known music synthesizers. This part of your computer gives you:

- 3 independent voices, each with a full 9 octave piano-type range.
- 4 different waveforms (sawtooth, triangle, variable pulse, and noise).
- A programmable ADSR (attack, decay, sustain, and release) envelope generator.
- A programmable high, low, and bandpass filter that you can use for each voice.
- Variable resonance and volume controls.

If you want your music to play back with professional sound reproduction, the 64 computers let you connect your audio output to almost any high-quality amplification system.

LOW-PRICED PERIPHERALS

As your computer needs grow, so can your system. You can expand your system by connecting your Executive 64 to other pieces of equipment, known as peripherals, which include accessories like these:

- The VIC-1541 disk drive (as many as five at a time).
- The VIC 1525, 1526, and MPS-801 dot matrix printers, and the 1520 printer/plotter, for printed copies of your programs, letters, etc.
- The 1600 VICMODEM for access through your telephone to the massive databases of larger computers, and the services of hundreds of specialists and a variety of information networks.
- The Commodore 1701/1702 color monitor.
- The Z-80 microprocessor, for access to CP/M**, which offers a variety of applications software.

Commodore wants you to really enjoy your new Executive 64. As you learn, bear in mind that programming takes time to learn. Be patient with yourself as you go through the USER'S GUIDE. Before you start, please take a few minutes to fill out and mail in the owner/registration card that came with your computer. This will ensure that your new computer is properly registered with Commodore Headquarters and that you receive the most up-to-date information regarding future enhancements for your system.

NOTE: Many programs are under development while this manual is being produced. Please check with your local Commodore dealer and with Commodore User's Magazines and Clubs, which will keep you up-to-date on the many applications programs being written all over the world for the Commodore 64 and Executive 64.

*Commodore 64 and Executive 64 are trademarks of Commodore Electronics Ltd.

**CP/M is a registered trademark of Digital Research, Inc. Specifications are subject to change.

CHAPTER 1

SETTING UP

CHAPTER 1
SETTING UP

- Unpacking and Setting Up the Executive 64 Computer
- Transporting the Executive 64
- Troubleshooting Chart: Executive 64
- Color Adjustment
- Expanding Your Systems With Optional Peripherals

RELEASE
BUTTONS

SX 64

commodore

GAME
PORTS

CARTRIDGE
PORT

AUDIO/VIDEO
CONNECTOR

SERIAL
PORT

USER
PORT

POWER SOCKET

FUSE

POWER
SWITCH

STORAGE (SX-64 ONLY)

MONITOR
CONTROL PANEL

DISK DRIVE

DISK DRIVE
LIGHT

HANDLE
LOCK

HINGE

5" COLOR
MONITOR

KEYBOARD-TO-BASE
CONNECTOR

BASE-TO-
KEYBOARD
CONNECTOR
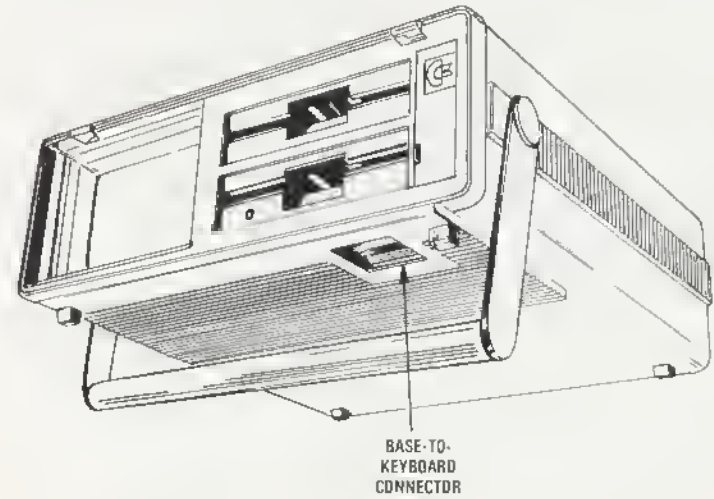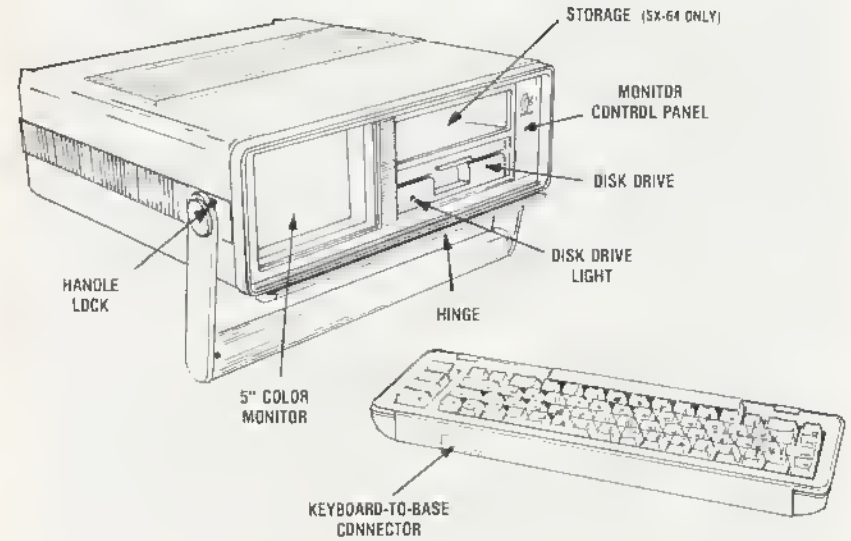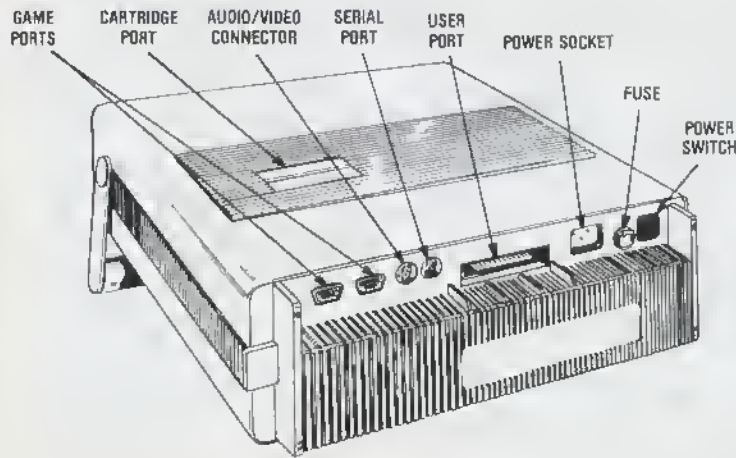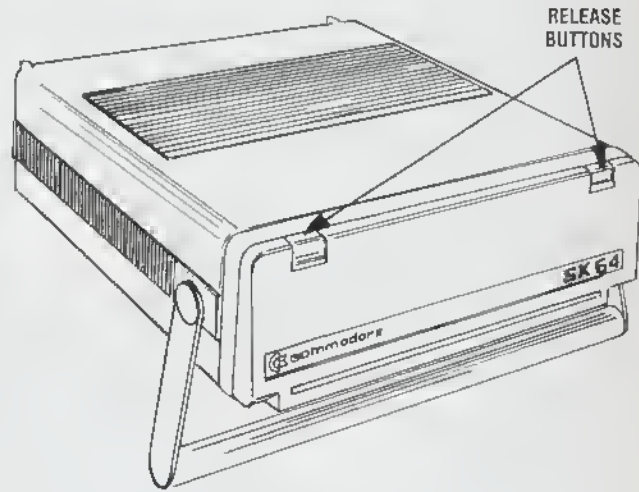
UNPACKING AND SETTING UP THE EXECUTIVE 64 COMPUTER

Remove your Executive 64 carefully from Its packaging. Inside you should find these items:

- The Executive 64 computer
- A black carrying case that contains two white boxes. One box contains a power cord; the other contains a smaller cable for connecting the keyboard.

If any of these items is missing when you first unpack your Executive 64, contact your dealer immediately for replacement.

Before you begin setting up your Executive 64 for the first time, take a few minutes to examine the diagrams in this section. Look over your computer, noting the locations of the outlet, called ports, where peripheral devices are plugged into the computer. Note the ON/OFF switch, the power cord outlet, and the handle locks. This examination will simplify the set up procedure.

Follow these Instructions each time you set up your Executive 64:

1. Stand the Executive 64 on its end so that the front of the computer and the handle are on top.
2. Unlock the handle. The locks are located at the base of each side of the handle, just under the circular blue hinges. Note that there are three dots on each hinge just above the lock tabs. Move both lock tabs to the center position.
3. Move the handle down towards the side wlth the four rubber feet. Set the handle so It is perpendicular to the sides of the computer. You may wish to readjust Ihe handle later.
4. Relock the handle lock tabs so the handle is stable. Make sure both lock tabs are locked and that the handle is secured.
5. Set the computer down carefully so that the handle serves as the front support.
6. Remove the front cover of the Executive 64 by pushing down the dark grey buttons at the top of the cover. Hold down the buttons while you ease the top of the cover towards you. Lift out the bottom of Ihe cover and place the keyboard on the table in front of the computer base.
7. Connect the keyboard to the base with the short, dark grey cable you found In the separate carrying case when you first checked the contents of the Executive 64. The larger, angled end of the cable goes into a slot on the bottom of the computer located just under the disk drive. You should lift up the computer base to see how this plug connects. Note the corresponding 2 rows of holes where the plug goes. Hold the plug with the rows of pins perpendicular to the computer base. In other words, the pins don't go in first; they should be on the side facing the back end of the computer. Push the plug up and in until it is securely installed.
8. Plug the other end of the cable into the port in the back of the keyboard. Note that the plug goes in only one way. Make sure the plug is secure.
9. Make sure your Executive 64 is turned OFF. The power switch is on the far left top of the back end as you face the monitor. Look at this end of the computer, and move the switch back and forth until you see the whlte circle on one side of the switch. When the computer is OFF, the circle is SHOWING.
10. Plug in the power cord. This cord goes in the back near Ihe power swltch. Note that the plug goes in only one way. Make sure Ihe cord is secure. Plug the other end Into a standard 120 volt outlet for three prong plugs.

11. Turn the computer on with the power switch discussed in step 9. When the computer is on, the red dlsk drive light (or lights, if you have two drives) come on and In a few seconds the screen displays the power-on message:

```
*****      SX-64 BASIC V2.0      *****
64K RAM SYSTEM  38911 BASIC BYTES FREE
READY.
```

12. Adjust the picture, if necessary. The background color should be white, the border should be cyan (light blue-green), and the characters should be blue. The picture controls are behind the small door with the Commodore logo. This door Is on the right front as you face the monitor. Use the button to open the door. The control knobs are identified inside the door. Use them as you would use the adjustments on a TV set. Note that the top knob controls the volume for your Executive 64.

Now you're ready Io begin using your Executive 64. If you have any problems, consult the trouble-shooting chart. Before you use the disk drive, be sure to consult Chapter 2.

TRANSPORTING THE EXECUTIVE 64

The Executive 64 is easy to pack and carry with you. Follow these steps to prepare your Executive 64 for carrying:

1. Remove any disks from the disk drives and turn the computer OFF. If you have an SX-64, do not carry disks in the computer's storage slot.
2. Remove any cartridges.
3. Turn off and disconnect any peripheral devices that are attached to your computer.
4. Remove the power cord from the back end of the computer.
5. Unplug the keyboard connection cable from the keyboard, and then remove the other end from the base of the computer.
6. Pack the cables in their boxes, and put the boxes in the carrying case. Note that the smaller cable goes in the smaller box.
7. Place the metal strip at the top of the keyboard into the long slot at the bottom of the computer's front, just under the monitor and disk drive.
8. Push the keyboard on the slot until the keyboard clicks into place in the buttons above the monitor and disk drive. Make sure the keyboard is securely attached.
9. Lift the computer up to stand on its back end.
10. Unlock the tabs on each side of the handle's base. The tabs are unlocked when they are in the center position. Make sure both tabs are unlocked.
11. Move the handle up to the top of the computer so the handle rests in the carrying position. The handle should be parallel to the sides of the computer's base.
12. Relock the tabs. The tabs are locked when they are in the side positions, not in the center position. Make sure both tabs are locked and that the handle is secure.
13. Attach the carrying case containing the cords to the Executive 64's handle.

The Executive 64 is now ready to be carried wherever you go. Be sure to handle the Executive 64 carefully when you transport it.

TROUBLESHOOTING CHART: Executive 64

Symptom	Cause	Remedy
Indicator Light not "On"	Computer not "On"	Make sure power switch is in "On" position
	Power cable not plugged in	Check power socket for loose or disconnected power cable.
	Power supply not plugged in	Check connection with wall outlet
	Bad fuse in computer	Take system to authorized dealer for replacement of fuse
No picture	Power not ON	Check previous remedies
	Malfunction	Take system to authorized Commodore dealer for repair
Random pattern on monitor when cartridge is in	Cartridge not properly installed	Turn off power and reinsert cartridge
Picture with poor or no color	Poorly adjusted picture	Adjust picture controls
Excess background noise	Volume too high	Adjust volume control
Picture OK, but no sound	Volume too low	Adjust volume control
Disk won't LOAD	Disk improperly inserted	Remove disk; insert according to directions (see Chapter 2)
	Drive door not closed	Make sure door is secure
	Bad disk	Insert another disk
	Malfunctioning drive	Take system to authorized dealer for repair

CURSOR

The flashing square under READY is called the cursor. It's a marker that shows where what you type on the keyboard will be displayed on the screen. As you type, the cursor moves ahead one space as the original cursor position is replaced with the character you typed. Try typing on the keyboard and watch the cursor move while characters you type are displayed on the screen.

COLOR ADJUSTMENT

There is a simple way to get a pattern of colors on the monitor so you can easily adjust the set. Even though you may not be familiar with the operation of the computer right now, just follow along, and you'll see how easy it is to use your computer.

First, look on the left side of the keyboard and locate the key marked `CTRL`. This stands for ConTROL and is used, in conjunction with other keys, to instruct the computer to do a specific task.

To use a control function, you hold down the `CTRL` key while pressing a second key.

Try this: hold the `CTRL` key while also pressing the `9` key. Then release both keys. Nothing obvious should have happened, but if you touch any key now, the screen will show the character displayed in reverse type, rather than normal type — like the opening message of anything you typed earlier.

Hold down the `SPACE BAR`. What happens? If you did the above procedure correctly, you should see a light blue bar move across the screen and then move down to the next line as long as the `SPACE BAR` is pressed.

```
READY

```
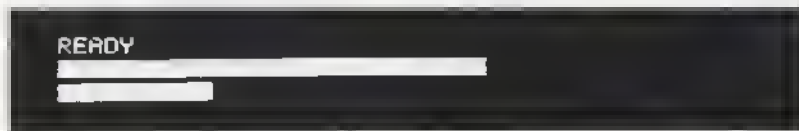
Now, hold `CTRL` while pressing any of the other number keys. Each of them has a color marked on the front. Anything displayed from this point will be in that color. For example, hold `CTRL` and the `8` key and release both. Now hold the `SPACE BAR`.

Watch the display. The bar is now in yellow! In a like manner you can change the bar to any of the other colors indicated on the number keys by holding `CTRL` and the appropriate key.

Change the bar to a few more different colors and then adjust the color and tint controls on your monitor so the display matches the color you selected.

The display should appear something like this:

At this point everything is properly adjusted and working correctly. The following chapters will introduce you to the BASIC language. However, you can immediately start using some of the many prewritten applications and games available without knowing anything about computer programming.

Each of these packages contains detailed information about how to use the program. It is suggested, though, that you read through the first few chapters of this manual to become more familiar with the operation of your new system.

EXPANDING YOUR SYSTEM WITH OPTIONAL PERIPHERALS

Commodore offers a variety of peripheral devices that expand the capabilities of your computer. These peripherals include:

- storage devices
- printers and plotters
- monitors
- modems for telecommunications
- game attachments
- speech and graphics modules
- desktop controllers

STORAGE DEVICES

Disk Drives

Commodore's disk drives let you store large amounts of information on 5¼" floppy diskettes. Diskettes offer fast storage and retrieval, and they automatically keep track of all your files in a directory, or table of contents, that you can display on your screen or print on a printer.

The SX-64 is equipped with one built-in disk drive; the DX-64 has two drives. In addition, you can add extra disk drives by daisy-chaining them to your computer. Daisy-chaining means connecting one drive to the computer, and then connecting additional drives to each other.

By acquiring the Commodore 64 IEEE Interface Expansion Card, you can also attach any IEEE disk drive, such as Commodore's CBM 8050 or 4040 Dual Floppy Disk Drives, to the Executive 64.

Chapter 2 contains detailed information on using disk drives.

PRINTING AND PLOTTING DEVICES

Printers

You can attach Commodore's 1525, 1526, or MPS-801 Printers to the Executive 64. These models are inexpensive dot matrix printers. By acquiring the Commodore 64 IEEE Interface Expansion Card, you can also attach any IEEE printer, such as Commodore's 6400 letter quality printer, or the high speed 8023 dot matrix printer, to the Executive 64.

Printer/Plotter

Commodore's 1520 Printer/Plotter prints and draws graphics in four colors (black, blue, red, and green). With the 1520, you can draw bar charts, pies, and a variety of complex graphics.

THE 1701/1702 MONITOR

Commodore's 14" color monitor offers a superior color picture with high resolution that enhances your computing experience. This monitor can be connected to the Executive 64. The monitor is connected to the computer by an 8-pin DIN cable. The 1701/1702 Color Monitor User's Guide that comes with the monitor clearly explains connections. You can also consult Appendix I for information about the pinouts in the 8-pin connector.

NOTE: You can also use a 5-pin cable to connect a monitor to the Executive 64. Just plug it into the 8-pin connector.

MODEMS FOR TELECOMMUNICATIONS

The 1600 VICMODEM

Commodore's inexpensive 1600 VICMODEM lets you use your telephone to connect your Executive 64 to other computers and computer services. These computer services include CompuServe, The Commodore Information Network, The Source, Dow Jones News/Retrieval, and others.

THE Z-80 MICROPROCESSOR AND CP/M® OPERATING SYSTEM

The Z-80 microprocessor turns your Executive 64 into a dual microprocessor home computer. The Z-80 gives you access to the popular CP/M Operating System, which offers a variety of software applications, including wordprocessing, widely-used business programs, high level computer languages (e.g., COBOL, FORTRAN), and other useful programs.

ATTACHMENTS FOR GAMES AND OTHER USES

Commodore offers joysticks, paddles, and trackballs that enhance game-playing on your computer. These attachments also have other applications. For example, the joystick works with Commodore's MAGIC DESK™ software series, which lets you perform many complex tasks without knowing any computer language. The joystick moves a hand that points to a picture representing the task you want to do.

THE COMMODORE SPEECH MODULE

Commodore's Speech Module makes your Executive 64 talk. The module comes with a built-in vocabulary of 235 utterances, and, with software support, it can speak in different voices. The Module is easily programmed in BASIC and also works with preprogrammed software on disk and cartridge. Some of Commodore's first talking cartridges include: the MAGIC DESK™ series, GORF, and WIZARD OF WOR.

COMMODORE GRAPHICS AIDS

Commodore provides a variety of graphics programming aids, including the SUPEREXPANDER 64 cartridge, which adds easy-to-learn graphic plotting and music programming commands to BASIC; SIMON'S BASIC, which adds 118 powerful new commands to BASIC, including programming help and graphics commands; and LOGO, an easy-to-learn programming language with TURTLE graphics.

SPECIAL DESKTOP CONTROLLER

Commodore will soon introduce a special device that controls the screen as you move the controller across a desktop. This new controller will be an optional enhancement for Commodore Software products, such as the MAGIC DESK Series.

MUSIC ATTACHMENTS

Commodore will also soon offer a Musical Keyboard and a 3-pad percussion attachment called the Digi-drum™. Both products will include special software packages. These attachments will increase the music making capabilities of the 64 computers.

CONNECTING TO A STEREO SYSTEM

The sound and music-making capabilities of the Executive 64 can be enhanced by connecting your computer to a high quality amplifier and stereo speakers. The 8-pin DIN cable discussed in the 1701/1702 Color Monitor section can also be used to connect your computer to an amplifier.

DESIGNING A COMPUTER SYSTEM FOR YOUR NEEDS

Commodore offers a variety of peripherals that let you create your own customized computer system. We offer different types of storage, printing, and telecommunications devices so you can choose what's best for you. For more information about Commodore peripherals, read The Commodore Peripherals Guide and the Commodore magazines discussed in Appendix R and consult your Commodore dealer.

CHAPTER 2

GETTING STARTED

CHAPTER 2
GETTING STARTED

- Communicating with your 64: The Keyboard
- Loading and Saving Programs
- How to Load Prepackaged Software
- How to Format a New Disk
- How to Save Programs
- Listing a Directory of programs on a Disk
- Files
- Sequential Files
- Relative Files
- Random Access Files
- Chart of BASIC Commands for Disks and Printers

COMMUNICATING WITH YOUR 64: THE KEYBOARD

The computer keyboard lets you communicate with your 64. You use the keys to tell the computer what you want it to do and to answer the questions the computer displays on the screen.

The keyboard looks like a regular typewriter, but the computer has special keys that let the 64 do more than a typewriter. While you read the next few pages, take a look at these special keys.

RETURN

The RETURN key tells the computer to look at what you typed and put this information in memory. The RETURN key also moves the cursor to the next line.

NOTE: Memory is all the information the computer currently knows without needing you to tell it where to look.

SHIFT

The SHIFT key works like the shift key on a regular typewriter: it lets you print capital letters or the top characters on double character keys.

When you are using the graphics on the front of the keys, the SHIFT key displays the graphic character on the RIGHT side of the key.
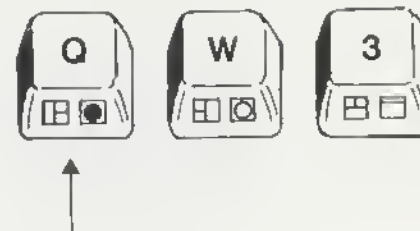
When you are using the four special function keys at the right side of the keyboard, the SHIFT key gives you the functions on the FRONT of the key (f2, f4, f6, and f8).

KEYS THAT LET YOU MAKE CHANGES

CRSR

The cursor is the little colored rectangle that marks your place on the screen. There are two CuRSoR keys:

↑CRSR↓   moves the cursor up and down
←CRSR→   moves the cursor left and right

You must use the SHIFT key with the ↑CRSR↓ key to move the cursor up, and with the ←CRSR→ key to move the cursor to the left.

You don't have to keep tapping a CRSR key to get it to move more than one space. Just hold it down until the cursor is where you want it.

INST/DEL

DEL stands for DELete. When you press the DEL key, the cursor moves back a space and erases the character that's there.

PRINT "ERROR"#■
PRINT "ERROR"■

When you DELete in the middle of a line, move the cursor just to the left of the character you want to DELete.

FIX IT AGAINS, SAM
FIX IT AGAINS■ SAM

Then press the DEL key. The characters to the right automatically move over to close up the space.

FIX IT AGAIN, SAM

INST stands for INSerT. You have to use the SHIFT key with the INST/DEL key when you want to insert characters in a line.

If you've left some characters out of a line, use the CRSR keys to move the cursor back to the error.

WHILE U WERE OUT
WHILE ■ WERE OUT

Then, while you hold down the SHIFT key, press the INST/DEL key until you have enough space to add the missing characters. INST doesn't move the cursor; it adds space between the cursor and the character to its right.

WHILE ■ U WERE OUT
WHILE YOU WERE OUT

Use the DEL and INST keys together to fix wrong characters.

WE'RE NUMBER TWO!
WE'RE NUMBER !
WE'RE NUMBER ■   !
WE'RE NUMBER ONE !

CLR/HOME

RESTORE

HOME moves the cursor back to the upper left corner of the screen. This is called the "HOME" position.

CLR stands for CLeaR. When you use the SHIFT key with the CLR/HOME key, the screen CLeaRs and the cursor returns to the home positon.

The RETORE key returns the computer to its normal state by RESTOREing the default conditions (e.g., the default screen color is blue, the default for I/O chips is OFF, etc.) RESTORE does such things as clear the screen, returning it to the original color, and turn off the picture· and sound-making chips.

NOTE: For RESTORE to work, you must hold down the STOP key while you press the RESTORE key.

For example, suppose you've just played a music program that also turned your screen red and yellow while it LISTed the program. When you press STOP and RESTORE at the end of the program, the last note from the program will cease, your screen will turn blue and the only thing displayed will be the READY prompt.

The keys on the right side of the keyboard, f1-f8, are function keys that you can program to perform a variety of tasks. The explanation of the GET statement in Chapter 5 tells you how to program function keys.

CTRL

The ConTRoL key lets you set colors and do other special tasks called control functions.

To set colors, hold down the CTRL key while you press the key with the color you want. You can get eight more colors with the C= key. Chapter 6 also has more about colors.

To get a control function, hold the CTRL key down while you press the other key. Control functions are commonly used in prepackaged software such as a word processing system.

RUN/STOP

You can halt a BASIC program while it is still RUNning by pressing the STOP key. You can also use the STOP key to halt a printout while it is still printing.

RUN lets you load a program automatically from diskette.

When you want to use the RUN key, you must also use the SHIFT key.

C= COMMODORE KEY

The Commodore key C= can do two things:

1. C= lets you switch back and forth between the upper and lower case display mode (the letters and characters on the tops of the keys) and the upper case/graphic display mode (capital letters and the graphics on the fronts of the keys).

   To switch modes, press the C= and SHIFT keys at the same time.

   When you first turn on your 64, it is in the upper case/graphic mode, which means that everything you type in is in capital letters. When you are in this mode, you can also print all the graphics on the fronts of the keys.

   • To print the graphic on the right side of a key, hold down the SHIFT key while you press the key with the graphic you want to print. You can only print the right side graphics when you are in the upper case/graphic mode.
   • To print the graphic on the left side of a key, hold down the C= key while you press the graphic key. You can print the left side graphic in either mode.
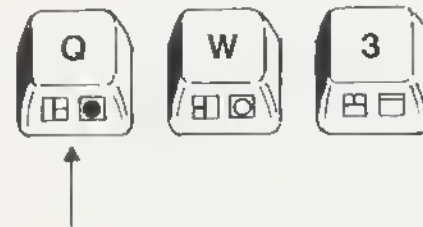
2. The C= key also lets you use the second set of eight colors shown on the color keys. To get these other colors, hold down the C= key while you press the number for the color you want.

LOADING AND SAVING PROGRAMS

The Executive 64 accepts programs from disk or cartridge. This means you can use prewritten software simply by loading it. But more important, the 64 lets you save your own programs for reuse. To reuse a program you wrote and saved on disk, all you do is load and run it.

Prepackaged software on disk or cartridge designed for the Commodore 64 also runs on the Executive 64.

HOW TO LOAD PREPACKAGED SOFTWARE

Loading Cartridges

You can use a special line of programs and games on cartridge with your 64. The programs include a wide variety of business and personal applications. The games are just like real arcade games, not imitations.

Follow these steps to load games and other cartridges:

1. Turn OFF your Executive 64.

   YOU MUST TURN OFF YOUR EXECUTIVE 64 BEFORE YOU INSERT OR REMOVE CARTRIDGES. IF YOU DON'T, YOU MAY DAMAGE THE CARTRIDGE AND THE COMPUTER.

2. Insert the cartridge in the slot on the top of your computer.
3. Turn on your 64.
4. Begin the game by typing the START key that's listed in the game's instruction sheet.

Loading Disks

Disks, which are often called "floppy disks", are really easy to use.

The steps are the same for loading preprogrammed disks and disks that you program yourself.

1. Insert a disk into your disk drive. Make sure the label on the disk is facing up. Put the disk in so that the labelled end goes in last. Look for a little notch on the disk (it might be covered with a little piece of tape). This notch must be on the left side as you put in the disk, assuming that you're facing your computer. Be sure the disk is all the way in.
2. Close the protective gate on the disk drive after you insert the disk. Just push down the lever.
3. Type LOAD "PROGRAM NAME", 8. The 8 is the code for disks. You need to type it here to let the computer know you're loading a disk.

   NOTE: You can LOAD the first program by using the * sign in place of the program name: LOAD "*", 8.

4. Press the RETURN key. The disk will spin and your screen will say:

   SEARCHING FOR PROGRAM NAME
   LOADING

   READY
   ■

5. Type RUN when the screen says READY and the cursor appears. Your software is ready to use.

HOW TO FORMAT A NEW DISK

When you're using a new, unprogrammed disk for the first time, you need to format it. Formatting, which is also called headering, prepares your disk by doing things like dividing the disk into blocks. Formatting also creates a directory that you use as a table of contents for the files you save on the disk. DO NOT header a preprogrammed disk.

You only have to format new disks, not disks that already have programs on them unless you want to erase the entire disk and reuse it.

To format a new disk, use this special version of the OPEN and NEW commands: ID = IDENTIFICATORE

OPEN 1,8,15,"NØ:<name>,<id>"

NØ tells the computer to header (NEW) the disk in drive 0. If you have a DX-64, header disks in drive Ø.

The name you use in this command goes in the directory as the name of the entire disk. Give the disk any name up to 16 characters.

The id is any two characters. Give the disk any id you want, but you should give every disk a different id code.

When the disk drive light goes off, type CLOSE 1 and press return.

BE CAREFUL! Headering a disk erases all information on the disk, if there is any. Header only a new disk or a disk you are willing to erase. Here are some examples of OPEN commands that header a disk:

OPEN 1,8,15,"NØ:MYFILE,A3"
OPEN 1,8,15,"NØ:$RECORDS,02"   ESEMPI

Now that you know how to header a disk, you are ready to use disks to write and save programs on your Executive 64. Appendix S contains more information on the OPEN command.

HOW TO SAVE PROGRAMS

When you want to reuse a program you've written, be sure to SAVE it before you LOAD another program. If you don't, you'll lose the program.

When you change a SAVEd program, you have to SAVE it again if you want to keep the new version.

When you reSAVE a program, you are replacing the old version with the new one. If you want to keep both the old and the changed versions, you have to give the new one a different name when you SAVE it.

When you want to SAVE a program you've written on disk, follow these simple steps:

1. Key in SAVE "PROGRAM NAME",8. The 8 is the code for disks. It tells the computer that you're using a disk.
2. Press RETURN. The disk makes a noise, and the computer displays this message when the program is saved:

   SAVING "PROGRAM NAME"
   OK
   READY

LISTING A DIRECTORY OF PROGRAMS ON A DISK

When you SAVE programs on a disk, the computer automatically makes a table of contents, or a DIRECTORY, of the names of the programs on the disk. You can display this directory to see what programs are on your disk. Follow these steps:

1. Key in: LOAD "$",8 and press RETURN. The computer displays this message:

   SEARCHING FOR $
   LOADING
   READY

2. Key in: LIST and press RETURN

Your programs names are displayed on your screen.

FILES

There are two types of files that you can store:

1. PROGRAM FILES, which store BASIC or machine language programs.
2. DATA FILES, which store numeric and string data that you can use in programs. A unit of a data file is called a RECORD; each record can contain one or more pieces of data.

There are three types of data files, which differ in the way data records are stored and accessed:

1. **SEQUENTIAL FILES**, which store data in records in the order you input the data. Data records are accessed sequentially, which means that you can't go directly to record 10; you must read the file from beginning to end.
2. **RELATIVE FILES**, which let you access data records directly by positioning a pointer to any record relative to the beginning of the file, which means that you can go directly to record 10.
3. **RANDOM ACCESS**, or user files, which store data records according to your design. Unlike the other file types, these files are not maintained by the disk operating system. You must design and maintain them yourself.

SEQUENTIAL FILES

FORMAT FOR OPENING A SEQUENTIAL FILE:

OPEN file#, device#, channel#, "0:name,type,direction"

The file number is the same as in all your other applications of the OPEN statement, and it is used throughout the program to refer to this particular file. The device# is usually 8. The channel# is a data channel, number 2 through 14. It is convenient to use the same number for both the channel# and file#, to keep them straight. The name is the file name (no wild cards or pattern matching if you're creating a write file). The type can be any of the ones from the chart below. You can abbreviate by using the first letter (P, S, U, R). The direction must be READ or WRITE. You can use just the first letter (R or W).

FILE TYPE	MEANING
PRG	Program
SEQ	Sequential
USR	User
REL	Relative

EXAMPLES OF OPENING SEQUENTIAL FILES:

OPEN 2, 8, 2, "0:DATA, S, W"

OPEN 8, 8, 8, "0:Program, P, R"

OPEN A, B, C, "0:" + A$ + "U, W"

If the file already exists, you can use the **replace** option in the OPEN statement. Simply add the @0: before the file's name in the OPEN statement.

EXAMPLE OF SEQUENTIAL FILE WITH REPLACE OPTION:

OPEN 2, 8, 2, "@0:DATA,S,W"

The 0: should always precede the name of the file or the drive will only allow you to use 2 of the available buffers.

WRITING TO AND READING FROM A SEQUENTIAL FILE

PRINT# and INPUT# and GET#

The PRINT# command works **exactly** like the PRINT statement, except that output is re-directed to the disk drive. The reason for the special emphasis on the word exactly is that all the formatting capabilities of the PRINT statement, as applies to punctuation and data types, applies here too. It just means that you have to be careful when putting data into your files.

FORMAT FOR WRITING TO FILE WITH PRINT#:

PRINT# file#, data list

The file# is the one from the OPEN statement when the file was created.

The data list is the same as the regular PRINT statement — a list of variables and/or text inside quotes. However, you must be especially careful when writing data so that it is as easy as possible to read back again later.

When using the PRINT# statement, if you use commas (,) to separate items in the list, the items will be separated by some blank spaces, as if it were being formatted for the screen. Semicolons (;) don't result in any extra spaces.

In order to more fully understand what's happening, here is a diagram of a sequential file created by the statement OPEN 5, 8, 5, "0:TEST,S,W":

	eof															
char	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...

The **eof** stands for the **end-of-file** marker. String data entering the file goes in byte by byte, including spaces.

For instance, let's set up some variables with the statement A$ = "HELLO"; B$ = "ALL": C$ = "BYE". Here is a picture of a file after the statement PRINT# 5, A$; B$; C$:

	H	E	L	L	O	A	L	L	B	Y	E	CR	eof
char	1	2	3	4	5	6	7	8	9	10	11	12	13

CR stands for the **CHR$ code 13**, the **carriage return**, which is automatically PRINTed at the end of every PRINT or PRINT# statement unless there is a comma or semicolon at the end of the line.

NOTE: Do not leave a space between PRINT and #, and do not try to abbreviate the command as ?#. See the appendixes in the user manual for the correct abbreviation.

FORMAT FOR INPUT# STATEMENT:

INPUT# file#, variable list

When using INPUT# to read data, there is no way to tell that it's not supposed to be one long string. You need something in the file to act as a separator. Characters to use as separators include the CR, a comma or a semicolon. The CR can be added easily by just using one variable per line in the PRINT# statement, and the system puts one there automatically. The statement PRINT# 5, A$: PRINT# 5, B$: PRINT# 5, C$ puts a CR after every variable being written, providing the proper separation for a statement like INPUT#5, A$, B$, C$. A line like Z$ = ",":PRINT# 5, A$ Z$ B$ Z$ C$ will do the job as well, and in less space. The file after that line looks like this:

	H	E	L	L	O	,	A	L	L	,	B	Y	E	CR	eof
char	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Putting commas between variables results in lots of extra space on the disk being used. A statement like PRINT# 5, A$, B$ makes a file that looks like:

	H	E	L	L	O						A	L	L		CR	eof
char	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	23	24

22

Numeric data written in the file takes the form of a string, as if the STR$ function had been performed on it before writing it out. The first character will be a blank space if the number is positive, and a minus sign (−) if the number is negative. Then comes the number, and the last character is the cursor right character. This format provides enough information for the INPUT# statement to read them in as separate numbers if several are written with no other special separators. It is somewhat wasteful of space, since there can be two unused characters if the numbers are positive.

Here is a picture of the file after the statement PRINT# 5, 1; 3; 5; 7 is performed:

		1	→		3	→		5	→		7	→	CR	eof
char	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

GET#

The GET# retrieves data from the disk, one character at a time.

FORMAT FOR THE GET# STATEMENT:

GET# file#, variable list

Data comes in byte by byte, including the CR, comma, and other separating characters. It is much safer to use string variables when using the GET# statement. You will get a BASIC error message if string data is received where a number was requested, but not vice-versa.

EXAMPLES OF GET# STATEMENT:

GET# 5, A$

GET# A, B$, C$, D$

GET 5, A

The GET# statement is extremely useful when examining files with unknown contents, like a file that may have been damaged by an experimental program. It is safer than INPUT# because there is a limit to the number of characters allowed between separators of INPUT variables. With GET#, you receive every character, and you can examine separators as well as other data.

23

RELATIVE FILES

FORMAT FOR THE OPEN STATEMENT TO CREATE RELATIVE FILE:

OPEN file#, device#, channel#, "name,L," + CHR$(record length)

EXAMPLES OF OPEN STATEMENT CREATING RELATIVE FILES:

OPEN 2, 8, 2, "FILE,L," + CHR$(100)

OPEN F, 8, F, A$ + ",L," + CHR$(Q)

OPEN A, B, C, "TEST,L," + CHR$(33)

RELATIVE FILE FORMAT

DATA BLOCK	
BYTE	DEFINITION
0,1	Track and sector of next data block.
2—256	254 bytes of data. Empty records contain FF (all binary ones) in the first byte followed by 00 (binary all zeros) to the end of the record. Partially filled records are padded with nulls (00).

SIDE SECTOR BLOCK	
BYTE	DEFINITION
0,1	Track and sector of next side sector block.
2	Side sector number. (0-5)
3	Record length.
4,5	Track and sector of first side sector (number 0)
6,7	Track and sector of second side sector (number 1)
8,9	Track and sector of third side sector (number 2)
10,11	Track and sector of fourth side sector (number 3)
12,13	Track and sector of fifth side sector (number 4)
14,15	Track and sector of sixth side sector (number 5)
16—256	Track and sector pointers to 120 data blocks.

Upon execution, the DOS first checks to see if the file exists. If it does, then nothing happens. The only way to erase an old relative file is by using the SCRATCH command, but **not** by using the replace option.

FORMAT FOR OPENING AN EXISTING RELATIVE FILE:

OPEN file#, device#, channel#, "name"

In this case, the DOS automatically knows that it is a relative file. This syntax, and the one shown in the above section, both allow either reading or writing to the file.

In order to read or write, you must, **before any operation**, position the file pointer to the correct record position.

The position command is sent to the command channel (15), so you must open an additional file for the command channel. In the following example, 15 is the file number of the command channel, and 8 is the file number of the relative file.

OPEN 15,8,15
OPEN 8,8,8

FORMAT FOR POSITION COMMAND:

PRINT#file#, "P" CHR$(channel# + 96) CHR$(rec#lo) CHR$(rec#hi) CHR$(position)

EXAMPLES OF POSITION COMMAND:

PRINT#15, "P" CHR$(CH + 96) CHR$(R1) CHR$(R2) CHR$(P)

PRINT#15, "P" CHR$(4 + 96) CHR$(R1) CHR$(R2) CHR$(P)

The 2-byte format for the record number is needed because one byte can only hold 256 different numbers, and we can have over 700 records in the file. The rec# lo contains the least significant part of the address, and the rec# hi is the most significant part. This could be translated to the actual record number by the formula $REC\# = REC\ HI * 256 + REC\ LO$.

If the record number is known, the high and low bytes can be determined as follows:

$REC\# \ HI = INT(REC\#/256)$
$REC\# \ LO = REC\# - REC\# \ HI * 256$

EXAMPLE:

PRINT#15, "P" CHR$(4 + 96) CHR$(R1) CHR$(R2) CHR$(0)

If $REC\# = 540$: $R2 = INT(540/256)$ . . . so $R2 = 2$
$R1 = 540 - R2*256$ . . . so $R1 = 28$

Let's assume we have a mailing list. The list consists of 8 pieces of data, according to this chart:

Field Name	Length		
		state	2
first name	12	zip code	9
last name	15	phone number	10
address line 1	20		
address line 2	20	— — — — — —	
city	12	TOTAL	100

This is how the record length is determined. We would probably want to allow an extra character in length for each field, to allow for separations; otherwise the INPUT# command would pick up a much longer piece of the file than needed, just like in sequential files. Therefore, we'll set up a file with a length of 108 characters per record.

When working with a new relative file that will soon be very large, it will save much time to create a record at the projected end of the file. In other words, if you expect the file to be 1000 records long, create a record# 1000 as soon as the file is created. This will force the DOS to create all intermediate records, making later use of those records much faster.

EXAMPLE OF CREATING LARGE FILE:

OPEN 1, 8, 15: OPEN 2, 8, 2, "0:REL,L," + CHR$(60)
PRINT#1, "P" CHR$(2 + 96) CHR$(0) CHR$(4) CHR$(1)
PRINT#2, "END"
CLOSE 2: CLOSE 1

RANDOM FILES, OR USER FILES

Random files let you access and update each of the individual 256-byte blocks of data stored on the disk. As was mentioned in the first chapter, there are a total of 683 blocks on the diskette, of which 664 are free on a blank diskette. Each block of data really means 1 Track and sector of the same name.

The diskette is divided into tracks, which are laid out as concentric circles on the surface of the diskette. There are 35 different tracks, starting with track 1 at the outside of the diskette to track 35 at the center. Track 18 is used for the directory, and the DOS fills up the diskette from the center outward.

Each track is subdivided into sectors. Because there is more room on the outer tracks, there are more sectors there. The outer tracks contain 21 sectors each, while the inner ones only have 17 blocks each. The table below shows the number of sectors per track.

Track and Block Format

TRACK NUMBER	SECTOR RANGE	TOTAL BLOCKS
1 to 17	0 to 20	21
18 to 24	0 to 18	19
25 to 30	0 to 17	18
31 to 35	0 to 16	17

The DOS contains commands for reading and writing directly to any track and sector on the diskette. There are also commands for checking to see which blocks (tracks & sectors) are available, and for marking off used blocks.

NOTE: You must be sure to maintain this information, especially when there are other types of files on the disk. Failure to do so could corrupt other files.

These commands are transmitted through the command channel (channel# 15), and tell the disk what to do with the data. The data must be read later through one of the open data channels.

OPENING A DATA CHANNEL FOR RANDOM ACCESS

When working with random access files, you need to have 2 channels open to the disk: one for the commands, and the other for the data. The command channel is OPENed to channel 15, just like other disk commands you've encountered so far. The data channel for random access files is OPENed by selecting the pound sign (#) as the file name. The # sign represents the buffer number (1-5) where "#" defaults to the next available buffer.

FORMAT FOR OPEN STATEMENT FOR RANDOM ACCESS DATA:

OPEN file#, device#, channel#, "#"

or

OPEN file#, device#, channel#, "#buffer#"

EXAMPLES OF OPENING RANDOM ACCESS DATA CHANNEL:

OPEN 5, 8, 5, "#"

OPEN A, B, C, "#2"

BLOCK-READ

FORMAT FOR BLOCK-READ COMMAND:

PRINT#file#, "BLOCK-READ:" channel; drive; track; sector

or abbreviated as

PRINT#file#, "B-R:" channel; drive; track; sector

This command will move one block of data from the diskette into the selected channel. Once this operation has been performed, the INPUT# and GET# statements can read the information.

BLOCK-WRITE

The BLOCK-WRITE command is the exact opposite of the BLOCK-READ command. First you must fill up a data buffer with your information, then you write that buffer to the correct location on the disk.

FORMAT FOR BLOCK-WRITE COMMAND:

PRINT#file#, "BLOCK-WRITE:" channel; drive; track; sector

or abbreviated as

PRINT#file#, "B-W:" channel; drive; track; sector

When the data is being put into the buffer, a pointer in the DOS keeps track of how many characters there are. When you perform the BLOCK-WRITE operation, that pointer is recorded on the disk.

BLOCK-ALLOCATE

In order to safely use random files along with regular files, your programs must check the BAM (Block Availability Map) to find available blocks, and change the BAM to reflect that you've used them. Once you update the BAM, your random files will be safe — unless you perform the VALIDATE command.

FORMAT FOR THE BLOCK-ALLOCATE COMMAND:

PRINT#file#, "BLOCK-ALLOCATE:" drive; track; sector

If you try a block that isn't available, the DOS will set the error message to number 65, NO BLOCK, and set the track and block numbers to the next available track and block number. Therefore, any time you attempt to write a block to the disk, you must first try to allocate that block. If that block isn't available, read the next block available from the error channel and then allocate that block.

BLOCK-FREE

The BLOCK-FREE command is the opposite of BLOCK-ALLOCATE, in that it frees a block that you don't want to use anymore for use by the system. It is vaguely similar to the SCRATCH command for files, since it doesn't really erase any data from the disk — just frees the entry, in this case just in the BAM.

FORMAT FOR BLOCK-FREE COMMAND:

PRINT#file#, "BLOCK-FREE:" drive; track; sector

or abbreviated as

PRINT#file#, "B-F:" drive; track; sector

USING RANDOM FILES

The most common method for keeping track of which blocks on the disk you used, is to build up a sequential file to go with each random file. Use this file to keep a list of record, track, and sector locations. This means that there are 3 channels open to the disk for each random file: one for the command channel, one for the random data, and the other for the sequential data. This also means that there are 2 buffers that you're filling up at the same time.

BUFFER-POINTER

The buffer pointer keeps track of where the last piece of data was written. It also is the pointer for where the next piece of data is to be read. By changing the buffer pointer's location within the buffer, you can get random access to the individual bytes within a block. This way, you can subdivide each block into records.

For example, let's take a hypothetical mailing list. The information such as name, address, etc., will take up a total of 64 characters maximum. We could divide each block of the random access file into 4 separate records, and by knowing the track, sector, and record numbers, we can access that individual record.

FORMAT FOR BUFFER-POINTER COMMAND:

PRINT#file#, "BUFFER-POINTER:" channel; location

or abbreviated as

PRINT#file#, "B-P:" channel; location

EXAMPLE OF SETTING POINTER TO 64TH CHARACTER OF BUFFER:

PRINT# 15, "B-P:" 5; 64

USER1 and USER2

The user commands are generally designed to work with machine language (see the next chapter for more on this). The USER1 and USER2 commands are special versions of the BLOCK-READ and BLOCK-WRITE commands, but . . . with an important difference: the way USER1 and USER2 work with the buffer-pointer.

The BLOCK-READ command reads up to 256 characters, but stops reading when the buffer-pointer stored with the block says that block is finished. The USER1 command performs the BLOCK-READ operation, but first forces the pointer to 255 in order to read the entire block of data from the disk.

FORMAT FOR USER1 COMMAND

PRINT#file#, "U1:" channel; drive; track; sector

or

PRINT#file#, "UA:" channel; drive; track; sector

There is no difference between the U1 and UA designations for this command.

The BLOCK-WRITE command writes the contents of the buffer to the block on the disk along with the value of the buffer-pointer. The USER2 command writes the buffer without disturbing the buffer-pointer value already stored on that block of the diskette. This is useful when a block is to be read in with BLOCK-READ, updated through the BUFFER-POINTER and PRINT# statements, and then written back to the diskette with USER2.

FORMAT FOR USER2 COMMAND:

PRINT#file#, "U2:" channel; drive; track; sector

or

PRINT#file#, "UB:" channel; drive; track; sector

CHART OF BASIC COMMANDS FOR DISKS AND PRINTERS

This chart lists the BASIC commands and statements that are used to perform operations with disk drives. Commands marked with a * sign can also be used to direct output to printers when you use the printer device number, which is 4. Disk Operating Systems (DOS) error messages are described in Appendix B.

Command	Function	Examples	
CLOSE *	Ends file or closes channel to a device	CLOSE 1	closes file 1, which must have been previously OPENed with the same number
CMD *	Redirects output from screen to device named	OPEN 1,8,"SCOTT"	file must be OPENed first
		CMD 1	sends file 1 output to disk
		OPEN 1,4:CMD 1	sends output to printer
GET#	Reads characters one by one from device named in OPEN statement	OPEN 1,8,"SCOTT" GET# 1,A$	OPENs file 1 on disk. reads from file 1; assigns data to A$
		GET# 1,A,B,C	reads from file 1; assigns data to A, then B, then C then back to A, etc.
LOAD	Brings programs into current memory from disk	LOAD "*",8	loads 1st program on disk
		LOAD "SCM",8	loads file SCM from disk
		LOAD "$",8	loads file directory (key in LIST to display)
INITIALIZE	Returns drive to the same state as when you turned it on. Use when error prevents disk operations.	OPEN 15,8,15,"I"	OPENs command channel (15). When command channel is OPEN, the INITIALIZE command is sent to the device named (the disk drive, device 8).
INPUT#	Retrieves data stored on disk	OPEN 1,8,"MEYER" INPUT# 1,A,B	OPENs file 1 on disk reads data, assigns to variables A and B
NEW	Headers diskette. Also erases diskette.	OPEN 15,8,15"N0: SCM,01"	OPENs file SCM. N0 means NEW disk in drive 0. SCM is the name of the disk. 01 is the disk id.

Command	Function	Examples	
OPEN *	Opens channel for input or output to a peripheral	OPEN 1,8,8,"D,W"	opens file D on disk. W means write to file. 8 is the secondary address, required for disk. See Appendix S.
		OPEN 1,4	opens channel to printer
		OPEN 1,8,0	opens file 1 to read from disk
PRINT# *	Writes data to a file	OPEN 1,8,8,"SCM"	opens disk file 1 to write
		PRINT# 1,"A";"B"	writes A and B to file 1
		CLOSE 1	ends writing to file 1
RENAME	Changes a file name.	OPEN 15,8,15,"R: newname = old name"	RENAMEs file 15.
		OPEN 15,8,15,"R: SCM = MEYER"	RENAMEs file MEYER, changing it to SCM
SAVE	Stores program	SAVE "SCM",8	stores file SCM on disk
SCRATCH	Erase files from disk	OPEN 15,8,15"S: GMS"	Erases file GMS
		OPEN 15,8,15"S: M*"	Erases all files that start with M.
VALIDATE	Reorganizes disk. Collects blocks taken up by improperly CLOSEd files. Do not use with Random Access (User) Files.	OPEN 15,8,15,"V" OPEN 15,8,15, "VALIDATE"	Validates file 15. Validates file 15.
VERIFY	Compares program in current memory to stored program	SAVE "SM",8	puts current program on disk
		VERIFY "SM",8	compares SAVEd version of SM with version still in current memory to verify correct program storage

Appendix S and the Commodore 64 Programmer's Reference Guide contain more information about these BASIC commands and statements.

The 64 Programmer's Reference Guide explains all BASIC version 2.0 commands and statements. The Commodore Peripheral Guide contains more detailed information and sample programs for using disk drives and printers. There books are available at bookstores and from your Commodore dealer.

CHAPTER 3

BEGINNING BASIC

CHAPTER 3
BEGINNING BASIC

- Printing and Calculating
- Mathematical Functions
- Multiple Calculations On One Line
- Execution Order in Calculations
- Combining PRINT's Capabilities

PRINTING AND CALCULATING

If you don't know BASIC, this section teaches you how to do some simple things like print words and calculate problems.

The PRINT statement tells the 64 computer to print something on the screen. PRINT is one of the most useful and powerful commands in the BASIC language. You can use it to display just about anything, including graphics and the results of computations. To use the PRINT command, follow these steps:

1. Key in the word PRINT. This tells the computer what kind of job you want it to do.
2. Key in a quotation mark. This tells the computer where the message you want to print begins.
3. Key in whatever you want to print on the screen.
4. Key in a closing quotation mark. This tells the computer where the message you want to print ends.
5. Press the RETURN key. This tells the computer to follow your instructions, which in this case is to print your message exactly as you typed it.

When you follow these steps, the computer prints your message and displays the READY prompt. It looks like this:

PRINT "I LOVE MY COMMODORE"   You key in this and press RETURN

I LOVE MY COMMODORE   The computer prints this
READY

The Executive 64 prints whatever you enclosed in quotes. Remember to key in both quotation marks.

If you make a mistake in your PRINT statement, use the INST/DEL key to correct your error. You can change as many characters as you like before you press the RETURN key.

If you made a mistake that you didn't catch before you pressed the RETURN key, the computer can't follow your instructions. Instead, it displays an error message to help you figure out what you did wrong. For example:

?SYNTAX ERROR

If you get this message, check over what you typed in to see where you made a mistake. The computer is very precise, and it can't follow instructions that contain spelling errors or other mistakes. To avoid mistakes, be sure you type things in the correct form.

Remember that the best way to get to know BASIC and your 64 is to try different things and see what happens.

USING PRINT TO CALCULATE

You can use PRINT to do more than just display what you put in quotation marks. You can also use it to perform calculations and automatically display the results. Follow these steps:

1. Key in PRINT
2. Key in the calculation you want to solve. DON'T enclose it in quotation marks.
3. Press the RETURN key. The computer displays the answer followed by the READY prompt.

Here's an example:

PRINT 12 + 12   Type this line and press RETURN
24

READY   The computer displays
■   the answer

Be sure you leave off the quotation marks when you want the computer to solve a problem. If you type the problem inside quotation marks, the computer assumes you just want to display the problem, not solve it. For example:

PRINT "12 + 12"   Key in this line
12 + 12   and press RETURN

READY   The computer displays
■   what's in quotes

So all you have to do to use PRINT as a calculator is omit the quotation marks. You can use PRINT to add, subtract, multiply and divide. You can also use exponents and perform advanced mathematical functions such as figuring square roots.

MATHEMATICAL FUNCTIONS

ADDITION

Use the plus sign (+) to tell the computer to add numbers. Remember to press RETURN after you type PRINT and the calculation. This tells the computer to follow your instructions.

SUBTRACTION

Use the minus sign (−) to subtract. Press the RETURN key at the end of the calculation. For example:

PRINT 12 − 9   Key in this and RETURN
3   The computer displays this

MULTIPLICATION

Use the asterisk (*) to multiply. You can't use the conventional x because the computer would think it's the letter x, not the multiplication sign. Press RETURN at the end of the calculation. For example:

PRINT 12 * 12   Key in this and RETURN
144   The computer displays this

DIVISION

Use the slash mark (/) for division. Press the RETURN key after you type the calculation. For example:

PRINT 144/12   Key in this and RETURN
12   The computer displays this

EXPONENTIATION

Use the up arrow (↑) to raise a number to a power. Press the RETURN key after you type the calculation. For example, to find 12 to the fifth power, type this:

PRINT 12 ↑ 5   Key in this and RETURN
248832   The computer displays this

This is the same as:

PRINT 12 * 12 * 12 * 12 * 12
248832

TIP:
BASIC has shortcuts that make programming even faster. One shortcut is abbreviating BASIC keywords. For example, you can use a ? In place of PRINT. Throughout this book, we'll show you other abbreviations for BASIC keywords. Appendix D lists these abbreviations and shows what Is displayed on the screen when you type the abbreviated form.

MULTIPLE CALCULATIONS ON ONE LINE

The last example shows that you can perform more than one calculation on a line. You can also perform different kinds of calculations on the same line. For example:

? 3 * 5 − 7 + 2   Key in this and RETURN
10   The computer displays this

So far our examples have used small numbers and simple problems. But the 64 can do much more complex calculations. The next example adds large numbers.

Notice that 78956.87 doesn't have a comma between the 8 and the 9. You can't use commas this way In BASIC. BASIC thinks commas indicate new numbers, so it would think 78,956.87 is two numbers: 78 and 956.87. Remember to press RETURN after you type the problem.

? 1234.5 + 3457.8 + 78956.87
83649.17

The next example uses a ten digit number. The 64 can work with numbers that have up to ten digits, but can only display nine digits in the answer. So the 64 rounds numbers that are more than nine digits. Numbers five and over are rounded up, and numbers four and under are rounded down. This means that 12123123.45 is rounded to 12123123.5. Because of rounding, the computer doesn't give the same answer you'd get if you added these numbers by hand. In this case, the answer is 12131364.817. You can see the difference rounding makes.

? 12123123.45 + 345.78 + 7895.687
12131364.9

The 64 prints numbers between 0.01 and 999,999,999 using standard notation, except for leaving out commas in large numbers. Numbers outside this range are printed using scientific notation. Scientific notation lets you express a very large or very small number as a power of 10. For example:

? 123000000000000000
1.23E + 17

Another way of expressing this number Is 1.23 * 10 ↑ 17. The 64 uses scientific notation for numbers with lots of digits to make them easier to read.

There is a limit to the numbers the computer can handle, even using scientific notation. These limits are:

Largest numbers: + / − 1.70141183E + 38
Smallest numbers: + / − 2.93873588E − 39

EXECUTION ORDER IN CALCULATIONS

If you tried to perform some mixed calculations of your own, you might not have gotten the results you expected. This is because the computer performs calculations in a certain order.

In this calculation:

20 + 8 / 2

the answer is 14 if you add 20 to 8 first, and then divide 28 by 4. But the answer is 24 if you first divide 8 by 2, and then add 20 and 4.

On the 64, you always get 24 because the computer always performs calculations in the same order. Problems are solved from left to right, but within that general movement, some types of calculations take precedence over others. Here is the order of precedence:

First: – minus sign for negative numbers, not for subtraction.
Second: ↑ exponentiation, left to right
Third: */ multiplication and division, left to right
Fourth: + – addition and subtraction, left to right

This means that the computer checks the whole calculation for negative numbers before doing anything else. Then it looks for exponents; then it performs all multiplication and division; then it adds and subtracts.

This explains why 20 + 8 / 2 is 24: 8 is divided by 2 before 20 is added because division has precedence over addition.

There is an easy way to override the order of precedence: enclose any calculation you want solved first in parentheses. If you add parentheses to the equation shown above, here's what happens:

? (20 + 8) / 2
14

You get 14 because the parentheses allow 20 and 8 to be added before the division occurs.

Here's another example that shows how you can change the order, and the answer, with parentheses:

? 30 + 15 * 2 – 3
57

? (30 + 15) * 2 – 3
87

? 30 + 15 * (2 – 3)
15

? (30 + 15) * (2 – 3)
– 45

The last example has two calculations in parentheses. As usual, they're evaluated from left to right, and then the rest of the problem is solved. When you have more than one calculation in parentheses, you can further control the order by using parentheses within parentheses. The problem in the innermost parentheses is solved first. For example:

? 30 + (15 * (2 – 3))
15

In this case, 3 is subtracted from 2, then 15 is multiplied by – 1, and – 15 is added to 30. As you experiment with solving calculations, you'll get familiar with the order in which mixed calculations are solved.

COMBINING PRINT'S CAPABILITIES

The 64 computers let you combine the two types of print statements that you've read about in this book. Remember that anything you enclose in quotation marks is displayed exactly as you type it.

The next example shows how you can combine the types of PRINT statements. The equation enclosed in quotes is displayed without being solved. The equation not in quotes is solved. The semicolon separates the two parts of the PRINT statement (semicolon means no space).

? "5 * 9 = "; 5 * 9   You key in this and RETURN
5 * 9 = 45   The computer displays this

Remember, only the second part of the statement actually solves the calculation. The two parts are separated by a semicolon. You always have to separate the parts of a mixed PRINT statement with some punctuation for it to work the way you want it to. If you use a comma instead of a semicolon, there is more space between the two parts when they're displayed. A semicolon leaves out space.

The Executive 64's screen is organized into 4 zones of 10 columns each. When you use a comma to separate parts of a PRINT statement, the comma works as a tab, sending each result into the next zone. For example:

?"total:";95,"shortage:";15
total:95   shortage:15

If you have more than four results, they are automatically displayed on the next line. For example:

? 2 * 3,4 – 6,2 ↑ 3,6 / 4,100 + (–48)
6   –2   8   [1.5
52

Here's the difference when you use semicolons:

? 2 * 3;4 – 6;2 ↑ 3;6 / 4;100 + (– 48)
6 – 2 8 1.5 52

You can use the difference between the comma and the semicolon in formatting PRINT statements to create complex displays.

CHAPTER 4

WRITING
SIMPLE PROGRAMS
IN BASIC

CHAPTER 4
WRITING SIMPLE PROGRAMS IN BASIC

- Line Numbers
- The GOTO Statement
- Using the LIST Command
- Editing Tips
- How to Use Variables
- Using FOR...NEXT Loops

So far this book has shown you how to do simple things with your 64. You've experimented with typing single lines of instructions into your computer and getting instant results by pressing the RETURN key. This easy way of doing things on your computer is called the IMMEDIATE or CALCULATOR mode.

But you'll probably want to use your computer to do more complex jobs that use more than one statement. When you combine a number of statements into a PROGRAM, you can use the full power of your 64.

To see how easy it is to write your first program on the 64, follow these steps:

1. Clear the screen by holding down the SHIFT key while you press the CLR/HOME key.
2. Key in NEW and press RETURN. This clears out information that might still be in the computer's memory after your experimenting.
3. Key in the following two lines exactly as they appear here:

    10 ? "EXECUTIVE 64"
    20 GOTO 10

4. Remember to press the RETURN key after each line. After you key in the first line and press RETURN, you'll notice that the computer doesn't respond to the PRINT command right away like it did before when you typed in the same kind of commands. This is because you are now beginning the command with a line number (10). When you use line numbers, the computer knows that you're writing a program, so it waits for you to finish keying in the whole program before following any of your instructions.
5. Key in RUN and press RETURN. The RUN command tells the computer that you've finished keying in program statements, and you're ready to have your instructions followed. Here's what happens when you RUN this program:

    EXECUTIVE 64
    EXECUTIVE 64
    EXECUTIVE 64
    EXECUTIVE 64
    EXECUTIVE 64
    EXECUTIVE 64

6. Stop the program's execution by pressing the RUN/STOP key. The computer continues to follow your orders by printing EXECUTIVE 64 over and over until you interrupt with the RUN/STOP key. Here's how your screen looks when you press STOP.

    EXECUTIVE 64
    EXECUTIVE 64
    EXECUTIVE 64
    EXECUTIVE 64
    EXECUTIVE 64
    EXECUTIVE 64
    BREAK IN 10
    READY

This simple program introduces several important concepts that are the basis for all programming.

LINE NUMBERS

We mentioned before in step 4 that line numbers tell the computer that you're writing a program. They also tell the computer in what order you want the statements in your program to executive. Without line numbers to tell the computer when to follow which instruction, the computer doesn't know what to do first.

The longer and more complex your program is, the more important it is to remember that the computer relies on you to tell it WHEN to do things, as well as WHAT to do. One good thing about this is that you can key in line 20 before line 10 because the computer just checks the line numbers to find out the order for executing the program. The computer doesn't check for the order your lines appear on the screen.

Another advantage of line numbers is that you can use the number to refer to the statement on the line. When you want to go back and repeat the execution of a statement, all you do is refer to it by line number in a GOTO statement, as you did in the example above.

THE GOTO STATEMENT

When you told the computer to RUN the sample program above, EXECUTIVE 64 was PRINTed repeatedly Instead of just once because of the GOTO statement in line 20.

The GOTO statement tells the computer to go directly to a specified line. Then the computer follows the instructions in the specified line and goes on to the next line.

You can use a GOTO statement to tell the computer to go back to a line that's already been executed. Or GOTO can tell the computer to skip forward, even if this means that some lines in the program don't get executed.

In our example, the program PRINTS the message in line 10 and moves to line 20. There, the GOTO statement tells the computer to go back to line 10 and do what line 10 says to do. So, the program prints the message in line 10 again, and then moves to line 20, which sends the computer back to line 10 and so on.

This repetition is called a LOOP. Because the example doesn't give the computer a way out of the loop, the circle repeats endlessly. You have to halt the cycle by Interrupting the program with the RUN/STOP key.

It's best to include a statement in your program that ends the loop so you don't have to use the RUN/STOP key. We'll explain more above ending loops later in this chapter.

USING THE LIST COMMAND

Now that you've interrupted execution of the sample program, key in LIST on your screen. Your program Is now displayed intact because it's still in the computer's memory, even though you interrupted the program's execution. The only difference is that the computer changed your ? Into the word PRINT. This doesn't affect your program, it's just the way the computer does things. When you use the LIST command, the computer also displays the lines of the program in correct numerical order, even If you entered the lines out of order.

One of the important differences between writing programs and entering single lines in the immediate/calculator mode is that you permanently lose an immediate statement once you execute it and clear the screen. But, until you start a new program, you can always get a program back just by keying in LIST.

From here, you can change the program, SAVE it, or RUN it again.

EDITING TIPS

When you make a mistake In a line you've keyed In, or when you just want to change a line, the 64 offers you a number of editing options.

1. You can retype a line any time, and the computer automatically substitutes the new line for the old one. All you have to do to replace a line is use the same line number. For example:

```
10 ? "My name is Sarah"
20 ? "I was born in California"
20 ? "I live in Pennsylvania"
RUN
My name is Sarah
I live In Pennsylvania
```

As you can see, the first line 20 never executes because It is replaced by the second line 20. If you now key in a LIST command, you'll see that only the second line 20 Is still part of the program.

2. You can easily erase a line you don't want just by keying in the line number and pressing the RETURN key. If you now key in LIST, you'll see that the line is gone, and so is the line number.

3. You can easily edit an existing line. Use the CuRSoR keys to move the cursor back to the line you want to change, and then just edit the line any way you want to. As soon as you press the RETURN key, the edited line will replace the old line. Remember to use the INST/DEL key to insert or delete.

When you finish editing, you can check your program again to verify changes by keying in the LIST command. Remember that LIST also puts lines In numerical order if you've keyed them In out of order.

Try editing our sample program by adding a semicolon to the end of the line, changing EXECUTIVE to COMMODORE, and omitting the 64. After you finish the changes, be sure to move the cursor past line 20 before you RUN the program. Here's how the program works now:

```
LIST
10 PRINT "COMMODORE";
20 GOTO 10
COMMODORE  COMMODORE COMMODORE COMMODORE
COMMODORE  COMMODORE COMMODORE COMMODORE
BREAK IN 10
READY
```

HOW TO USE VARIABLES

A variable is a symbol that stands for a value. Sometimes the value of a variable is unknown before you RUN a program. One of the purposes of a program may be to find one or more values for a variable. Look at this line from a program:

10 X = 28 + Y

In this equation, X and Y are variables. Suppose X stands for the number of days in a month. One of the best things about a variable is that you can reuse it in a program, so X can stand for the days in all the months, not just one month. This is where Y comes in. All months have 28 days, so Y stands for the days over 28. Later in this chapter there's a program that gives values to these two variables.

The most important thing now is understanding how variables work, because variables allow you to do complex tasks with you computer. Variables also let you write programs that are very reusable.

Imagine that your computer contains a bunch of little slots, like a bank of mail boxes. When you write a program, you can use some of these slots to hold values. All you do is give a name to the slots you need, and during the program you can put values into each slot by using the slot's name. For example, in the equation above, we used two slots by naming one X and one Y. At the beginning of a program, these slots have names, but they're empty. Here's what happens when you put a value in Y's slot:

X	Y					
	3					

Now the variable Y has the value 3. You can give Y this value just by writing this simple statement:

20 Y = 3

Since Y equals 28 plus Y, when you RUN the program X's slot gets a value, too.

X	Y					
31	3					

Here's how the program looks:

10 X = 28 + Y
20 Y = 3
30 ? "THE NUMBER OF DAYS IN MAY IS";X
RUN
THE NUMBER OF DAYS IN MAY IS 31

Here's another program that uses variables:

10 X% = 15
20 X = 23.5
30 X$ = "TOTAL:"
40 Y = X% + X
50 ? X$;Y

When you RUN the program, the imaginary slots look like this after line 30 is executed:

X%	X	X$	Y	
15	23.5	TOTAL:		

When you RUN the program, Y gets a value: 38.5

The above example uses the three types of variables:

TYPE	SYMBOL	DESCRIPTION	EXAMPLES	POSSIBLE VALUES
Integer	%	whole numbers	X%, A1%	15,102,3
Text string	$	characters in quotes	X$, AB$	"TOTAL:", "DAY 1"
Floating point		read (decimal) or whole numbers	X, AB	23.5, 12, 1.3E + 2

Be sure you use the right variable types in your programs. If you try to do something like assign a text string to an integer variable, your program won't work.

There are a few other things to keep in mind when you assign names to variables:

- A variable name can have one or two characters, not counting the special symbol used with integer and text string variables.
- You can use more than two alphabetic characters in a variable name, but the computer only recognizes the first two. So the computer would think PA, PARTNO and PAGENO are the same variable referring to the same "slot".
- A program is easier for people to read when you use longer variable names, but when you use more than two characters in a name, be sure the first two are unique.
- You can use X, X%, and X$ in one program because the special symbols % and $ make each variable name unique. The same is true of A2, A2%, and A2$.
- The first character must be alphabetic (A to Z). The second and any later characters can be either alphabetic or numeric (0 to 9). Remember that the computer ignores every character after the second unless it's a % or $ in the third position.
- Variable names can't contain BASIC keywords, which are also called reserved words. These are the words like PRINT and RUN that are part of the BASIC language. Appendix D lists all the BASIC reserved words.

Here's one more sample program that shows you how to use variables. This example also uses some of the other things you've learned so far.

```
NEW
10 X = 1.05
20 Y = 300
30 Z = X * Y
40 PRINT "SEATS AVAILABLE:";Y
50 PRINT "TICKETS AVAILABLE:";Z
60 Y = Y + 1
70 PRINT "OVERBOOKING POINT:";Y
RUN
SEATS AVAILABLE: 300
TICKETS AVAILABLE: 315
OVERBOOKING POINT: 301
```

Lines (10 - 30) assign variable names.

Lines 40 and 50 PRINT a message and the current value of variables Y and Z. Notice that at line 40, the value for Y is 300.

Line 60 gives Y a new value, and this new value is PRINTed in line 70. Line 60 shows that a variable can have more than one value in a program.

Line 60 also shows another of the powerful features of variables: you can make a variable equal to itself and another value. This isn't allowed in regular algebra, but this kind of statement is commonly used in programming. It means: take the current value of a variable, combine it with another value, and replace the first value of the variable with this new value. You can also use statements like these:

```
Y = Y - 1
Y = Y + X
Y = Y / 2
Y = Y * (X + 2)
```

USING FOR/NEXT LOOPS

We mentioned loops earlier in this chapter during the explanation of the GOTO statement. As you'll recall, loops are repeated executions of one or more lines in a program.

The FOR/NEXT statement lets you create very useful loops that control the number of times a segment of a program is executed. The FOR statement sets a limit on the number of times the loop will execute by assigning a range of values to a variable. For example:

```
FOR COUNT = 1 TO 4
```

The NEXT statement marks the end of a FOR/NEXT loop. When the program reaches a NEXT statement, the computer checks the FOR statement to see if the limit of the loop has been reached. If the limit hasn't been reached, the loop continues and the variable in the FOR statement is incremented by one. For example, if you add a FOR/NEXT loop to the program at the beginning of this chapter, here's what happens:

```
10 FOR CT = 1 TO 4
20 ? "EXECUTIVE 64    "; "COUNT=";CT
30 NEXT CT
RUN
EXECUTIVE 64    COUNT= 1
EXECUTIVE 64    COUNT= 2
EXECUTIVE 64    COUNT= 3
EXECUTIVE 64    COUNT= 4
```

Now that you've added the FOR/NEXT loop, you don't have to break in with the STOP key to halt the program's execution.

This FOR/NEXT loop works like this:

Line 10 gives the variable CT a range of values from 1 to 4, and tells the computer to execute the next lines until CT equals 4.

Line 20 tells the computer to print EXECUTIVE 64.

Line 30 tells the computer to add 1 to the current value of CT. As long as the value of CT remains within the range of 1 to 4, the program repeats, and EXECUTIVE 64 is PRINTed again. When CT equals 4, line 20 executes one more time. When line 30 again adds 1 to CT, the computer knows that CT is now out of range. So the computer stops executing the loop, and the program ends by itself.

CHAPTER 5

ADVANCED BASIC

CHAPTER 5
ADVANCED BASIC

INTRODUCTION

The next few chapters are for people who are familiar with BASIC programming language and the concepts necessary to write advanced programs.

Those of you who are just starting to learn how to program may find some of the information too technical to understand completely. But you'll find some simple examples that are written for new users in two chapters, SPRITE GRAPHICS and CREATING SOUND. These examples will give you a good idea of how to use the sophisticated graphics and sound capabilities available on your Executive 64.

If you want to learn more about writing programs in BASIC, check the bibliography in the back of this manual (Appendix N).

If you are already familiar with BASIC programming, the following chapters will help you get started with advanced BASIC programming techniques. You'll find extensive information about advanced programming in the COMMODORE 64 PROGRAMMER'S REFERENCE GUIDE, which is available through your local Commodore dealer.

SIMPLE ANIMATION

You can use some of the 64's graphic capabilities by putting together what you've learned so far in this manual, along with a few new concepts.

Try entering the following program to see what you can do with graphics. Notice that you can include cursor controls and screen commands WITHIN a PRINT statement. When you see something like < CRSR left > in a program listing, hold down the SHIFT key and press the < — CRSR — > key. The screen shows the graphic representation of a cursor left, which is two vertical reversed bars. The graphic representation of the SHIFTed CLR/HOME key is a reversed heart.

```
NEW
10   REM BOUNCING BALL
20   PRINT "(CLR/HOME)"
25   FOR X = 1 TO 10: PRINT "(CRSR/DOWN)":NEXT
30   FOR BL = 1 TO 40
40   PRINT"    (CRSR LEFT)";:REM      is a SHIFT – O
50   FOR TM = 1 TO 5
60   NEXT TM
70   NEXT BL
75   REM MOVE BALL RIGHT TO LEFT
80   FOR BL = 40 TO 1 STEP – 1
90   PRINT " (CRSR LEFT) (CRSR LEFT)   (CRSR LEFT)";
100  FOR TM = 1 TO 5
110  NEXT TM
120  NEXT BL
130  GOTO 20
```

(INDICATES NEW COMMAND)

(THESE SPACES ARE INTENTIONAL)

> **TIP:**
> All words in this text will be completed on one line. However, as long as you don't hit **RETURN** your 64 will automatically move to the next line even in the middle of a word.

When this program RUNs, it displays a bouncing ball moving across the screen from left to right and back again. Take a close look at the program to see how this is done.

Line 10 is a REMark that tells you what the program does. A REMark statement has no effect on the program itself.

Line 20 clears the screen.

Line 25 PRINTs ten cursor-down commands. This just positions the ball in the middle of the screen. Without this line, the ball would move across the top line of the screen.

Line 30 sets up a loop to move the ball 40 columns from left to right.

Line 40 does three things:

1. PRINTs a space to erase the previous ball positions.
2. PRINTs the ball.
3. Performs a cursor-left to get ready to erase the current ball position again.

Line 50 and 60 set up a loop that slows down the ball's movement. Without this loop, the ball would move too fast for you to see clearly.

Line 70 completes the loop set up in line 30 to PRINT balls on the screen. Each time the loop executes, the ball moves another space to the right. As you can see from the illustration, the program contains a loop within a loop. You can include up to ten loops within a loop. The only time you get in trouble is when the loops cross over each other. The loops have to be NESTED inside each other. In other words, if you start loop A and then start loop B inside loop A, you must finish loop B (the inside loop) first.

When you're writing a program with loops, it's a good idea to draw arrows from the beginning to the end of the loops. If your loops cross, the computer can't figure out what you want, so it can't execute your program.

Lines 80 through 120 just reverse the steps in the first part of the program, and move the ball from right to left. Line 90 is slightly different from line 40 because the ball is moving in the opposite direction, and you have to erase the ball to the right and move to the left.

Line 130 sends the program back to line 20 to start the whole process over again.

For a variation on the program, change line 40 to read:

40 PRINT "<SHIFT> < O >"

Run the program and see what happens now. Because you left out the cursor control, each ball remains on the screen until it is erased by the ball moving right to left in the second part of the program.

INPUT

Up to now, everything in a program has been set up before the program RUNs. Once you executed the program, you couldn't change or add anything. The INPUT statement lets you send information to a program WHILE it is RUNning. Not only does the program act on this information you supply, but the program won't continue until you supply it.

To get an idea of how INPUT works, type NEW, press RETURN, and enter this short program.

```
10 INPUT A$
20 PRINT "YOU TYPED ";A$
30 PRINT
40 IF A$ = "STOP" THEN END
50 GOTO 10
RUN
? GO
YOU TYPED GO

? CONTINUE
YOU TYPED CONTINUE

? STOP
YOU TYPED STOP
```

Here's what happens in this program:

Line 10 tells the computer to display a question mark to prompt you to INPUT a value for A$, and to wait until you supply the value before continuing the program execution.

Line 20 PRINTs a message and the INPUT value, and line 30 PRINTs a blank line.

Line 40 tells the computer to end the program immediately IF the value you INPUT for A$ is STOP.

Line 50 returns the program to line 10 so you can INPUT another value. IF line 40 is true because the last value you INPUT for A$ was STOP, then line 50 isn't executed.

You can INPUT numeric or string variables, and you can have the INPUT statement print a message along with a question mark to describe the kind of INPUT the computer is waiting for. For example, here's what happens when you add a prompt message to line 10 of the previous example:

```
10 INPUT "KEEP GOING";A$        Prompt message can't
RUN                             be more than 38
KEEP GOING? GO                  characters
YOU TYPED GO

KEEP GOING? STOP
YOU TYPED STOP
```

Here's a more complex example that demonstrates a lot ot what's been presented so tar, including the INPUT statement.

```
NEW
1    REM TEMPERATURE CONVERSION PROGRAM
5    PRINT "(CLR/HOME)"
10   PRINT "CONVERT FROM FAHRENHEIT OR CELSIUS (F/C)":INPUT A$
20   IF A$ = "" THEN 10
30   IF A$ = "F" THEN 100
40   IF A$ <> "C" THEN END
50   INPUT "ENTER DEGREES CELSUIS: ";C
60   F = (C*9)/5 + 32
70   PRINT C;" DEG. CELSIUS = "; F;" DEG. FAHRENHEIT"
80   PRINT
90   GOTO 10
100  INPUT "ENTER DEGREES FAHRENHEIT: ";F
110  C = (F - 32)*5/9
120  PRINT F;" DEG.FAHRENHEIT = ";C;" DEG. CELSIUS"
130  PRINT
140  GOTO 10
```

Line 10 uses the INPUT statement to print a prompt message and to wait tor you to type in a value for A$.

Lines 20, 30 and 40 check what you typed in and tell the computer where to go next. Line 20 tells the computer to go back to line 10 and ask for INPUT again IF nothing was typed In (IF just RETURN was pressed). Line 30 tells the computer to go straight to line 100 and perform the Fahrenheit-to-Celsius conversion IF the value you typed tor A$ is F.

Line 40 checks to be sure that you haven't typed in anything beside F or C. IF you have, line 40 ends the program. IF you typed In a C, the computer automatically moves to line 50 to perform the Celsius-to-Fahrenheit conversion.

It may seem like too much detail to Include all these IF statements to check what you INPUT. But this is a good programming practice that can spare you a lot of frustration. You should always try to be sure that your program takes care ot all possiblIities.

Back to the example: once the program knows what type ot conversion to make, the calculations are made. Then the program PRINTs the temperature you entered and the converted temperature.

The calculation this program performs is just straight math, usIng the standard formula tor temperature conversion. After the calculation tinishes and the answer is PRINTed, the program loops back and starts over.

Here's a sample execution of this program:

```
CONVERT FROM FAHRENHEIT OR CELSIUS (F/C): ?F
ENTER DEGREES FAHRENHEIT: 32
32 DEG. FAHRENHEIT = 0 DEG. CELSIUS

CONVERT FROM FAHRENHEIT OR CELSIUS (F/C): ?
```

After you RUN this program, you might want to save it on dIsk. This program, as well as others in this manual, can form part of your program library.

USING THE GET STATEMENT FOR DATA INPUT

GET lets you Input one character at a time from the keyboard without pressing the RETURN key. This really speeds up entering data in many cases.

When you RUN a program that has a GET statement, whatever key you press Is assIgned to the variable you include in the GET statement. Here's an example:

```
1    PRINT "< CLR/HOME >"
10   GET A$: IF A$ = "" THEN 10          No space between
20   PRINT A$;                           quotes
30   GOTO 10
```

Line 1 clears the screen.

Line 10 lets you type In any key on the keyboard. In ettect, the loop in line 10 tells the computer to wait until you type in a key before moving to line 20.

LIne 20 displays the keys you type on the screen.

Line 30 sends the program back to GET another character. It's Important to remember that the character you type in won't be displayed unless you PRINT it to the screen, as we've done in line 20.

The IF statement in line 10 is very important. GET continually works, even it you don't press a key (unlike INPUT, which waits tor your response), so the second part of line 10 continually checks the keyboard untIl you hit a key.

Try leaving out the second part of lIne 10 and see what happens.

To stop this program, press the RUN/STOP and RESTORE keys.

You can easily rewrite the beginning of the temperature conversion program to use GET Instead of INPUT. If you've SAVEd this program, LOAD it and change lines 10 and 20 like this:

```
10 PRINT "CONVERT FROM FAHRENHEIT OR CELSIUS (F/C)"
20 GET A$: IF A$ = "" THEN 20
```

USING GET TO PROGRAM FUNCTION KEYS

As you'll recall from an earlier chapter, we told you that the keys on the right side of the keyboard (f1 through f8) are function keys that you can program to perform a variety of tasks.

Here's how to program a function key:

1. Use a GET Statement to read the keyboard.
2. Use IF statements to compare the key you press to the CHR$ code for the function key you want to use. Every character on the keyboard has a unique CHR$ number. For example, the CHR$ code of f1 is 133. Appendix F lists the CHR$ code for all keys.
3. Use THEN statements to tell the computer what you want the function key to do.

When you RUN the program, all you so is press a function key you programmed, and the key will follow the instructions you gave it in the THEN statement. For example:

```
10 GET A$: IF A$ = "" THEN 10
20 IF A$ = CHR$(137) THEN A$ = CHR$(14)
30 IF A$ = CHR$(134) THEN A$ = "YOURS TRULY" + CHR$(13)
```

Line 10 tells the program to assign the key you press to the variable A$. As you'll recall from the previous example, the loop in line 10 continually checks the keyboard for input.

Line 20 programs function key 2, CHR$(137). Line 20 tells the computer to make A$ equal to CHR$(14) if you press function key 2. CHR$(14) is the switch from upper to lower case letters on the keyboard. When you RUN this program, you'll see that the characters on the screen immediately make this switch if you press f2.

Line 30 programs function key 3, CHR$(134). Line 30 tells the computer to make A$ equal to the character string YOURS TRULY and CHR$(13) if you press f3 during program execution. CHR$(13) is the code for the RETURN key.

THE CHR$ codes for the function keys are:

```
f1 = CHR$(133)      f2 = CHR$(137)
f3 = CHR$(134)      f4 = CHR$(138)
f5 = CHR$(135)      f6 = CHR$(139)
f7 = CHR$(136)      f8 = CHR$(140)
```

The COMMODORE 64 PROGRAMMER'S REFERENCE GUIDE has more information about programming tunction keys. You can purchase this extensive guide from your local Commodore dealer.

RANDOM NUMBERS AND OTHER FUNCTIONS

The 64 also has built-in functions that you can use to perform special tasks. Functions are like built-in programs included in BASIC. The great advantage of these built-in functions is that you don't have to type in a number of statements every time you want to perform a specialized calculation. Instead, all you do is type the command for the function you want and the computer does all the rest.

These built-in functions include figuring square roots (SQR), finding out the contents of a memory location (PEEK), generating random numbers (RND), etc. Appendix C lists all the functions available on your computer.

One function you can have a lot of fun with is the random number function, RND. If you want to design a game or an educational program, you'll often need to be able to program your computer to make up random numbers. For example, you'd need to do this to simulate the tossing of dice. Of course you could write a program that would generate these random numbers, but it's much easier to be able to do this just by calling upon the prewritten RaNDom number function.

To see how RND works, try this short program:

```
NEW
10 FOR X = 1 TO 10        IF YOU LEAVE OUT THE COMMA, YOUR
20 PRINT RND (1),         LIST OF NUMBERS APPEARS AS 1 COLUMN
30 NEXT
```

When you RUN this program, the screen displays:

```
.789280697          .664673958
.256373663          .0123442287
.682952381          3.90587279E − 04
.402342724          .879300926
.158209063          .245596701
```

Your numbers don't match? It would be incredible if they did because the program generates a completely random list of ten numbers.

If you RUN the program a few more time, you'll see that the results are always different. Though the numbers don't have a pattern, you'll notice a few consistencies about the list the program displays.

For one thing, the results are always between 1 and 0, but never equal to 1 or 0. For another, the numbers are real numbers (with decimal points).

Now, we started out to simulate dice tosses, and the results from this program aren't exactly what we're looking for. Now we'll add a few more features to this program to get what we want.

First, add this line to the program to replace line 20, and RUN the program again:

```
20 PRINT 6*RND(1),
RUN
```

```
3.60563664          4.52687513
5.48602315          1.09650123
3.10045018          4.39052168
3.91302584          5.06321506
2.32056144          4.10781302
```

Now we've got results larger than 1, but still have real numbers. To solve this, we'll use another function.

The INT function converts real numbers to integer (whole) numbers. So try replacing line 20 again:

```
20 PRINT INT(6*RND(1)),
RUN
```

```
2    3    1    0
2    4    5    5
0    1
```

Now we're even closer to our goal, but you'll notice that the numbers range from 1 to 5, not 1 to 6. So as a final step, we'll replace line 20 again:

```
20 PRINT INT(6*RND(1)) + 1
```

Now when you RUN the program, you'll get the results you want.

When you want to generate a range of real numbers instead of whole numbers, the formula is slightly different because you must subtract the lower limit of the range from the upper limit. For example, you can generate random numbers between 1 and 25 by typing:

```
20 PRINT RND(1)*(25 - 1) + 1
```

The general formula for generating random numbers in a certain range is:

```
NUMBER = RND(1) *(UPPER LIMIT - LOWER LIMIT) + LOWER LIMIT
```

GUESSING GAME

Here's a game that uses random numbers. This game not only uses the RND function, but it also introduces some additional programming theory.

When you RUN this program, the computer generates a random number, NM, whose value you'll try to guess in as few turns as possible.

```
NEW
1    REM NUMBER GUESSING GAME
2    PRINT "(CLR/HOME)"
5    INPUT "ENTER UPPER LIMIT FOR GUESS ";LI
10   NM = INT(LI*RND(1)) + 1
15   CN = 0
20   PRINT "I'VE GOT THE NUMBER.":PRINT
30   INPUT "WHAT'S YOUR GUESS"; GU
35   CN = CN + 1
40   IF GU>NM THEN PRINT "MY NUMBER IS LOWER": PRINT:GOTO 30
50   IF GU<NM THEN PRINT "MY NUMBER IS HIGHER": PRINT:GOTO 30
60   PRINT "GREAT! YOU GOT MY NUMBER"
65   PRINT "IN ONLY "; CN ;"GUESSES.":PRINT
70   PRINT "DO YOU WANT TO TRY ANOTHER (Y/N)";
80   GET AN$: IF AN$ = "" THEN 80
90   IF AN$ = "Y" THEN 2
100  IF AN$ < >"N" THEN 70
110  END
```

INDICATES NO SPACE AFTER QUOTATION MARK

You can specify how large the number will be at the start of the program. Then, it's up to you to guess what the number is.

A sample run follows along with an explanation.

```
ENTER UPPER LIMIT FOR GUESS? 25
I'VE GOT THE NUMBER.

WHAT'S YOUR GUESS ? 15
MY NUMBER IS HIGHER.

WHAT'S YOUR GUESS ? 20
MY NUMBER IS LOWER.

WHAT'S YOUR GUESS ? 19
GREAT! YOU GOT MY NUMBER
IN ONLY 3 GUESSES.

DO YOU WANT TO TRY ANOTHER (Y/N)?
```

The IF/THEN statement (lines 40-60) compare your guess to the random number (NM) generated by line 10. If your guess is wrong, the program tells you whether your guess is higher or lower than NM.

Each time you make a guess, line 35 adds 1 to CN. CN is a counter that keeps track of how many guesses you take to get the right number. The purpose of this game, of course, is to guess the number in as few tries as possible.

When you get the right answer, the program displays the message, GREAT! YOU GOT MY NUMBER, and tells you how many guesses you took.

Remember that the program creates a new random number each time you play the game.

You might want to add a few lines to the program that also specify the lower range of numbers generated by this game.

PROGRAMMING TIPS:

In lines 40 and 50, a colon separates multiple statements on a single line. This not only saves typing time, but it also conserves memory space.

Also notice that the IF/THEN statements in these two lines PRINT something before branching to another line.

YOUR ROLL

The following program simulates the throw of two dice. You can play this little game by itself, or use it as part of a larger game.

```
5    PRINT "Care to try your luck?"
10 PRINT "RED DICE = ";INT(RND(1)*6)+1
20 PRINT "WHITE DICE = ";INT(RND(1)*6)+1
30 "PRESS SPACE BAR FOR ANOTHER ROLL":PRINT
40 GET A$: IF A$ = "" THEN 40
50 IF A$ = CHR$(32) THEN 10
```

From what you've learned about BASIC and random numbers, see if you can follow what's going on in this program. As you may recall from the section on programming the function keys, CHR$(32) is the character string code for the space bar.

RANDOM GRAPHICS

As a final note on random numbers, and as an introduction to designing graphics, try entering and RUNning this program:

```
10 PRINT "<CLR/HOME>"
20 PRINT CHR$(205.5 + RND (1));
30 GOTO 20
```

The function CHR$ (CHaracter String) gives you a character, based on a standard code number from 0 to 255. Every character the 64 can print is encoded this way. Appendix F lists the CHR$ codes for all keys.

A quick way of finding out the code for any character is to use the function ASC (for the standard ASCII code). Type:

```
PRINT ASC("X")
```

X is the character you're checking. X can be any printable character, including graphics characters. You must enclose the character in quotation marks. Here's an example:

```
PRINT ASC("G")
71
```

The CHR$ function is the opposite of ASC.

```
PRINT CHR$(71)
G
```

If you type:

```
PRINT CHR$(205);CHR$(206)
```

the computer displays the two right side graphics on the M and N keys, which are the characters used in the little maze program you just tried.

The formula 205.5 + RND(1) tells the computer to pick a random number between 205.5 and 206.5 There is fifty-fifty chance that the random number will be above or below 206. CHR$ ignores fractional values, so half the time the character with code 205 is printed, and the rest of the time code 206 is displayed.

You can experiment with this program by adding or subtracting a couple of tenths from 205.5. This gives either character a greater chance of being displayed.

CHAPTER **6**

COLOR GRAPHICS

CHAPTER 6
COLOR AND GRAPHICS

HOW TO USE COLOR AND GRAPHICS ON YOUR COMPUTER

So far this book has presented some of the sophisticated computing capabilities of your 64. But one of the most exciting features of your new computer is its outstanding ability to produce 16 different colors and a lot of different graphics.

You've already seen a very simple demonstration of the graphics in the bouncing ball program and in the maze program at the end of the last chapter. This chapter introduces you to new concepts that explain graphic and color programming, and that suggest ideas for creating your own games and advanced animation.

PRINTING COLORS

When you tried the color alignment test in Chapter 1, you discovered that you can change text colors by simply holding down the CTRL key and pressing one of the color keys.

The 64 offers a full range of 16 colors. Though only eight colors are printed on the color keys, you can get eight more by holding down the C = key and pressing a color key. Here's a list of the colors:

KEYBOARD	COLOR	DISPLAY	KEYBOARD	COLOR	DISPLAY
CTRL 7	BLACK		C= 1	ORANGE	
CTRL 7	WHITE		C= 2	BROWN	
CTRL 3	RED		C= 3	LT. RED	
CTRL 4	CYAN		C= 4	GRAY 1	
CTRL 5	PURPLE		C= 5	GRAY 2	
CTRL 6	GREEN		C= 8	LT. GREEN	
CTRL 7	BLUE		C= 7	LT. BLUE	
CTRL 8	YELLOW		C= 8	GRAY 3	

When we showed you the boucing ball program in the last chapter, you saw that keyboard commands, such as cursor movement, can be written into PRINT statements. In the same way you can also add text color changes to your programs.

Type NEW and try experimenting with changing colors. Hold down the CTRL key and at the same time press the 1 key. Now release both keys and press the R key. Now hold down the CTRL key again and press the 2 key. Release the CTRL key and type the A key. Move through the numbers, alternating with the letters, and type out the word RAINBOW like this:

```
10 PRINT" R A I N B O W"
         ↑ ↑ ↑ ↑ ↑ ↑ ↑
   CTRL  1 2 3 4 5 6 7
```

You'll recall that cursor controls appear as graphic characters in the PRINT statement. Color controls are also represented as graphic characters. The color chart printed above shows the graphic characters that appear with each color. Because of the graphic characters that are displayed when you select color keys, your PRINT statement will look strange, but when you RUN the program, you'll see that only the text of the message is displayed. The letters in the message automatically change colors according to the color controls you placed in the PRINT statement.

Now try making up some examples of your own, mixing any number of colors within a single PRINT statement. Don't forget the second set of colors that you can get by holding down the C= key while you press a color key.

> **TIP:**
> After you RUN a program with color or mode (reverse) changes, you'll notice that the READY prompt and any additional text you key in is the same as the last color or mode change you made. To get back to the normal display, press these keys together:
>
> RUN/STOP and RESTORE

COLOR CHR$ CODES

Before you start reading this section, take a look at Appendix F, which lists the CHR$ codes for all keys on the keyboard.

As you looked over the list of CHR$ codes, you probably noticed that each color has a unique code, just like all the other keys and the keyboard controls. If you print the codes themselves by using the CHR$ function mentioned in the last chapter, you can get the same results you got by typing CTRL or C= and the color key in a PRINT statement.

For example, try this:

```
NEW
10  PRINT CHR$(147) : REM < CLR/HOME >
20  PRINT CHR$(28);"CHR$(28) CHANGES ME TO?"
RUN
```

CHR$(28) CHANGES ME TO?

When you RUN this program, the screen clears before the message in line 20 is PRINTed. The text should be red now.

In many cases, you'll find that it's much easier to use the CHR$ function to change colors, especially if you want to experiment. The next page shows another way to get a rainbow of colors. There are a number of similar lines in the program (40 through 110), so use the editing keys to spare yourself a lot of typing. See the notes at the end of the program listing to refresh your memory on editing procedures.

```
NEW
1     REM AUTOMATIC COLOR BARS
5     PRINT CHR$(147) : REM CHR$(t47) = CLR/HOME
10    PRINT CHR$(18); "            ";:REM REVERSE BAR
20    CL = INT(8*RND(1)) + 1
30    ON CL GOTO 40,50,60,70,80,90,100,t t0
40    PRINT CHR$(5);: GOTO 10
50    PRINT CHR$(28);: GOTO 10
60    PRINT CHR$(30);: GOTO 10
70    PRINT CHR$(31);: GOTO 10
80    PRINT CHR$(144);: GOTO 10
90    PRINT CHR$(156);: GOTO 10
100   PRINT CHR$(158);: GOTO 10
110   PRINT CHR$(159);: GOTO 10
```

Type lines 5 through 40 normally. Your display should look like this:

```
t   REM AUTOMATIC COLOR BARS
5   PRINT CHR$(147) : REM CHR$(147)= CLR/HOME
10 PRINT CHR$(18); "            ";:REM  REVERSE BARS
20 CL = INT(8*RND(1)) + 1
30 ON CL GOTO 40,50,60,70,80,90,100,110
40 PRINT CHR$(5);: GOTO 10
```
■

Use the < CRSR-up > key to position the cursor on line 40. Then type 5 over the 4 of 40. Now use the < CRSR-right > key to move over to the 5 in the CHR$ parentheses. Press SHIFT and INST/DEL to open up a space, and key in 28. Now just press RETURN with the cursor anywhere on the line.

The display should look like this now:

NEW

```
1   REM AUTOMATIC COLOR BARS
5   PRINT CHR$(147) : REM CHR$(147)= CLR/HOME
10  PRINT CHR$(18); "            " ;:REM REVERSE BAR
20  CL = INT(8*RND(1)) + 1
30  ON CL GOTO 40,50,60,70,80,90,100,110
50  PRINT CHR$(28);: GOTO 10
```

Don't worry about line 40; it's still there, as you can see by LISTing the program. Follow the same steps to modify line 40 with a new line number and CHR$ code until you've entered all the remaining lines. As a final check, LIST the entire program to make sure all the lines are right before you RUN it.

You probably understand the color bar program except for line 30. Here's a brief explanation of how this program works.

Line 5 prints the CHR$ code for CLR/HOME.

Line 10 turns on reverse type and prints 5 spaces, which turn out to be a bar since they're reversed. The first time through the program, the bar is light blue, the normal screen display color.

Line 20 uses the random function to select at random a color between 1 and 8.

Line 30 uses a variation of the IF/THEN statement, called ON/GOTO, which lets the program choose from as list of line numbers where the program will go next. If the ON variable (in this case CL) has a value of 1, the program goes to the first line number listed (here it's line 40). If the variable has a value of 2, the program goes to the second line listed, and so on.

Lines 40 through 110 just convert the random key colors to the appropriate CHR$ code for that color and return the program to line 10 to PRINT a section of the bar in that color. Then the whole process starts again.

See if you can figure out how to produce 16 random colors. Expand ON/GOTO to handle the additional colors and add the remaining CHR$ codes.

HOW TO USE PEEKS AND POKES

PEEKS and POKES let you search around inside your computer's memory and stick things in exactly where you want them.

You'll recall that in Chapter 4 we explained variables as being like little slots in the computer's memory, with the variable name as the slot's address. Well, imagine some more specially defined slots in the computer that stand for specific memory locations and that have numbers for addresses.

Your 64 looks at these memory locations to see what the screen's background and border colors should be, what characters to display on the screen and where to display them, etc.

You can change the screen colors, define and move objects, and even create music by POKEing a different value into the specific memory slots.

Imagine some memory slots looking something like this:

53280	53281	53282	53283
2	1		

| BORDER | BACKGROUND | | |
| COLOR | COLOR | | |

The first two slots are the memory locations for the border and background colors on your screen. We've put 2, the value for RED in the border color box, and 1, the value for WHITE in the background color box. Now try typing this:

POKE 53281,7   RETURN

The background color of your screen will change to yellow because we put the value 7, for yellow, in the location that controls backgound color.

Try POKEing different values into the background color location and see what result you get. Here's a list of the values to POKE for each color available on your 64:

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

Here's a little program that you can use to display various border and background color combinations:

```
NEW
10 FOR BA = 0 TO 15
20 FOR BO = 0 TO 15
30 POKE 53280,BA
40 POKE 53281,BO
50 FOR X = 1 TO 500: NEXT X
60 NEXT BO: NEXT BA
RUN
```

This program uses two simple loops to POKE various values to change the background and border colors. Line 50 contains a DELAY loop, which just slows the program down a little bit.

If you're curious about what value is currently in the memory location for background color, try this:

?PEEK (53280) AND 15

PEEK looks at a whole byte, but colors only use half a byte, called a nybble. To PEEK just this nybble, you have to add the AND 15 to your PEEK statement. If you used this PEEK after RUNning the previous program, you'd get 15 as the answer because the last border color POKEd was GRAY 3, which is 15.

In general, PEEK lets you see what value is currently in a specific memory slot. Try adding this line to your program to display the values of BORDER and BACKGROUND as the program RUNs.

```
25  PRINT CHR$(147); "BORDER = "; PEEK(53280) AND 15,
      "BACKGROUND = "; PEEK (53281) AND 15
```

SCREEN GRAPHICS

So far when you've PRINTed information, the computer has handled the information sequentially: one character PRINTed after the next, starting from the current cursor position, except when you asked for a new line, or used a comma in PRINT formatting.

You can PRINT data in a particular place by starting from a known place on the screen and PRINTing the correct number of cursor controls to format the display. But this takes time and program steps.

But just as there are certain locations in the 64's memory to control color, there are also memory locations that you can use to control screen locations.

SCREEN MEMORY MAP

The 64's screen can hold 1000 characters (40 columns by 25 lines), so there are 1000 memory locations set aside to represent what is on the screen. Imagine the screen as a grid, 40 by 25, with each square standing for one memory location.

Each memory location can contain one of the 256 different characters the 64 can display (see Appendix E). Each of these 256 characters is represented by a number from 0 to 255. If you POKE the value for a character into a specific screen memory location, that character will be displayed in that specific screen location.

Here's a grid that represents your screen, complete with the numbers of each screen memory location.
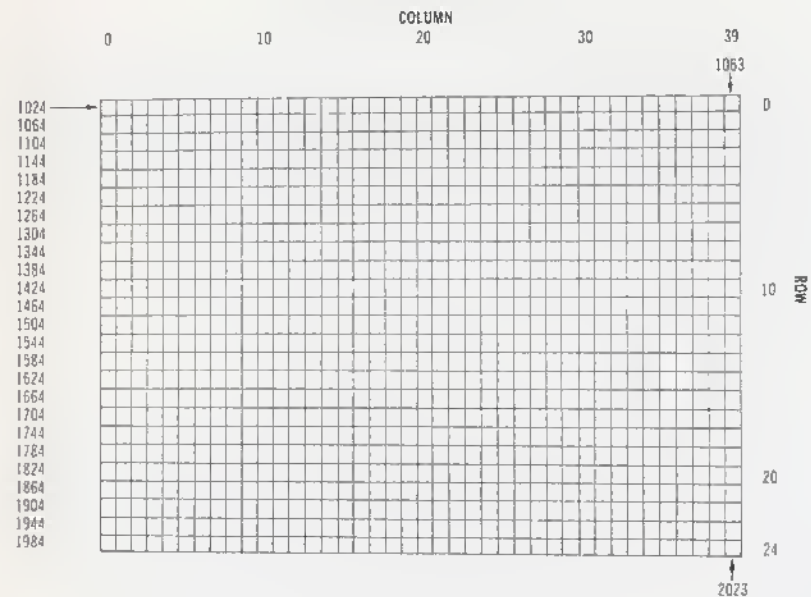
The 64's screen memory normally begins at memory location 1024 and ends at location 2023. Location 1024 is the upper left corner of the screen. Location 1025 is the position of the next character to the right, and so on. Location 1063 is the right-most position of the first row. Following the last character in a row, the next location is the left-most character on the next row down.

Suppose you want to control a ball bouncing on the screen. The ball is in the middle of the screen, column 29, row 12. The formula for calculating the memory location on the screen is:

$$POINT = 1024 + X + 40 * Y \underset{\text{column}}{\underline{\quad}} \text{row}$$

where X is the column and Y is the row.

Therefore, the memory location of the ball is:

$$POINT = 1024 + 20 + 480 \underset{\text{column}}{\underline{\quad}} \text{row (40x12)}$$
$$POINT = 1524$$

Clear the screen with SHIFT and CLR/HOME and type:

$$POKE \ 1524,81 \underset{\text{location}}{\underline{\quad}} \text{character code}$$

This POKE statement makes a ball appear in the middle of the screen. You have placed a character directly into screen memory without using the PRINT statement. However, you can't see the ball yet because it's the same color as the screen background.

COLOR MEMORY MAP

You can change the color of the ball that appeared by altering another range of memory. Type:

$$POKE \ 55796,2 \underset{\text{location}}{\underline{\quad}} \text{color}$$

This changes the ball's color to red.

Every spot on the 64's screen has TWO memory locations: one for the character code, and one for the color code. The color memory map begins at location 55296 (upper left corner), and continues on for 1000 locations. You use the same color codes, 0 through 15, that you used to change border and background colors, to directly change character color.

We can modify the formula for calculating screen memory locations to give us the locations to POKE colors. Here's the new formula:

$$COLOR \ PRINT = 55296 + X + 40 * Y \underset{\text{column}}{\underline{\quad}} \text{row}$$

MORE BOUNCING BALLS

Here's a revised bouncing ball program that directly prints on the screen using POKEs rather than cursor controls within PRINT statements. When you RUN this version, you'll see that it's much more flexible than the earlier program and it leads up to programming more sophisticated animation.

```
NEW
10   PRINT "  SHIFT CLR/HOME  "
20   POKE 53280,7 : POKE 53281,6
30   X = 1 : Y = 1
40   DX = 1 : DY = 1
50   POKE 1024 + X + 40*Y,81
60   FOR T = 1 TO 10 : NEXT
70   POKE 1024 + X + 40*Y,32
80   X = X + DX
90   IF X <= 0 OR X >= 39 THEN DX = -DX
100  Y = Y + DY
110  IF Y <= 0 OR Y >= 24 THEN DY = -DY
120  GOTO 50
```

Line 10 clears the screen.

Line 20 sets the background color to blue and the border color to yellow.

The X and Y variables in line 30 keep track of the ball's current row and column position. The DX and DY variables in line 40 are the horizontal and vertical direction of the ball's movement. When a +1 is added to the value of X, the ball moves to the right; when -1 is added, the ball moves to the left. A +1 added to Y moves the ball down a row, and a -1 added to Y moves the ball up a row.

Line 50 puts the ball on the screen at the current X,Y position. Line 60 is a delay loop, which is included to keep the ball on the screen long enough for you to be able to see it.

Line 70 erases the ball by putting a space (code 32) where the ball was on the screen.

Line 80 adds the direction factor to X.

Line 90 tests to see if the ball has reached one of the side walls, and reverses the ball's direction if there's a bounce. Lines 100 and 110 do the same thing for the top and bottom walls.

Line 120 sends the ball back to display and moves the ball again.

You can change the ball to any other character by changing the code in line 50 from 81 to another character code.

If you change DX or DY to 0 the ball bounces straight instead of diagonally.

We can also add a little intelligence to the bouncing ball program. So far the only thing you checked for is whether the ball is going out of bounds on the screen. Try adding the following lines to the program:

```
21   FOR L = 1 TO 10
25   POKE 1024 + INT(RND(1)*1000),166 ————CHR$CODE
27   NEXT L
115  IF PEEK(1024 + X + 40*Y) = 166 THEN DX = -DX : GOTO 80
```

Lines 21 to 27 put ten blocks on the screen in random positions. Line 115 PEEKs to see if the ball is about to bounce into a block, and, if so, it changes the ball's direction.

CHAPTER 7

INTRODUCTION
TO SPRITES

CHAPTER 7
INTRODUCTION TO SPRITES

- Bits, Bytes and Sprites
- Creating Sprites
- More on Sprites: Colors and Extra Movement

In previous chapters, we've shown you how to use graphic symbols in PRINT statements to create animation and other visual effects.

In chapter 6, we also showed you how to POKE character codes in specific screen memory locations, which put characters directly on the screen in the place you selected.

In both of these cases, you have to create objects from existing graphic symbols, so these methods take a lot of work. When you want to move the object, you must use a number of program statements to keep track of the object and move it to a new place. And sometimes the shape and resolution of the object isn't as good as you'd like it to be because of the limitations of using graphic symbols.

You can eliminate a lot of these problems by using sprites in animated sequences. A sprite is a high-resolution porgrammable object that you can make into just about any shape by using BASIC commands. All you have to do to move the object is simply tell the computer the position where you'd like the sprite to go. The computer takes care of the rest.

But this isn't all you can do with sprites. For example, you can change their color, you can tell if one object collides with another, you can make them go in front and behind each other, and you can easily expand their size.

You have to learn a few more details about your 64 and the way it handles numbers before you can use sprites. It's not difficult, though, so just follow the examples and you'll be making your own sprites do amazing things in no time.

BITS, BYTES, AND SPRITES

Before you can use sprites, it's important that you understand a few general things about how computers work, so here's a little about binary arithmetic.

Think of your computer's memory as a collection of BYTES, which break down into eight BITS per byte. A bit, which is the smallest amount of information a computer can store, can have only two values: ON or OFF. When a bit is ON, it has the value of 1; when a bit is OFF, it has the value of 0.

When you program your computer, what you're doing is turning bits ON or OFF. Since there are so many bits, and since each different configuration of ON and OFF bits sends a different message to the computer, you can do incredible things with your 64.

If you have two bits combined as a unit of meaning, you have four possible values. This illustration shows how the number of possible values increases with the number of bits:

NO. OF BITS	NO. OF VALUES	POSSIBLE COMBINATIONS	
1	2 ↑ 1 = 2	ON	1
		OFF	0
2	2 ↑ 2 = 4	ON and ON	1 1
		ON and OFF	1 0
		OFF and ON	0 1
		OFF and OFF	0 0
3	2 ↑ 3 = 8	ON and ON and ON	1 1 1
		ON and ON and OFF	1 1 0
		ON and OFF and ON	1 0 1
		ON and OFF and OFF	1 0 0
		OFF and ON and ON	0 1 1
		OFF and OFF and ON	0 0 1
		OFF and ON and OFF	0 1 0
		OFF and OFF and OFF	0 0 0

As you can see, the number of possible values equals 2 raised to the power of the number of bits. In other words, if you have 8 bits, you have 2↑8, or 256, different combinations of bits. Eight bits equal a BYTE, so a byte can store 256 different values.

When all eight bits are OFF, the byte equals zero; when all eight bits are ON, the byte equals 255 (since zero counts as the lowest of the 256 combinations, naturally the highest value is 255 rather than 256).

Any combination of a byte's ON's and OFF's, or 1's and 0's, converts to a decimal (i.e., base 10) value between 0 and 255. Here's a way to figure out the value of the byte when you have different combinations of ON and OFF bits:

Draw a table for the eight bits and fill in 1's for ON bits and 0's for OFF bits. Across the top of the table, write 128, 64, 32, 16, 8, 4, 2, and 1. These numbers are 2↑7, 2↑6, 2↑5, 2↑4, 2↑3, 2↑2, 2↑1, 2↑0, or 2 to the power of each of the eight bits, with the bits numbered 0 through 7. For example:

128	64	32	16	8	4	2	1
1	0	1	1	0	1	0	1

Now add up the values of the ON bits to find the value of the byte:

$128 + 32 + 16 + 4 + 1 = 181$

Here's a table showing BINARY to DECIMAL conversion. A zero indicates that a bit is OFF, and a 1 shows that a bit is ON. To find out the value of the entire byte, just add up the DECIMAL value of each ON bit, just as we did above.

BINARY TO DECIMAL CONVERSION

Decimal Value								
128	64	32	16	8	4	2	1	
0	0	0	0	0	0	0	1	2↑0
0	0	0	0	0	0	1	0	2↑1
0	0	0	0	0	1	0	0	2↑2
0	0	0	0	1	0	0	0	2↑3
0	0	0	1	0	0	0	0	2↑4
0	0	1	0	0	0	0	0	2↑5
0	1	0	0	0	0	0	0	2↑6
1	0	0	0	0	0	0	0	2↑7

TIP:
   Converting binary numbers to their decimal values is the basis for creating data to represent and manipulate sprites. Here's a program that does these conversions for you. Since you'll be using this program often, you should enter and save it.

```
5   REM BINARY TO DECIMAL CONVERTER
10 INPUT "ENTER 8-BIT BINARY NUMBER :";A$
12 IF LEN (A$) < > 8 THEN PRINT "8 BITS PLEASE...": GOTO 10
15 TL = 0 : C = 0
20 FOR X = 8 TO 1 STEP − 1 : C = C + 1
30 TL = TL + VAL(MID$(A$,C,1))*2 ↑ (X − 1)
40 NEXT X
50 PRINT A$;" BINARY ";" = ";TL;" DECIMAL"
60 GOTO 10
```

   At line 10 you enter a binary number as the string A$. Line 12 uses the LEN (length) function to check to be sure you entered 8 binary digits. If you didn't, the program asks for more and repeats line 10.
   In line 15, TL keeps track of the binary number's decimal value, and C indicates which bit is being worked on as the program goes through the loop.
   Line 30 updates the value of TL. Appendix C explains the VAL and MID$ functions.
   Line 50 PRINTs the binary and decimal values of the byte. Line 60 returns the program to the beginning.

A block of bytes strung together makes up a REGISTER. For sprite-making, each register is only one byte long. Each register controls a different function. For example, the ENABLE REGISTER controls whether the sprite is ON or OFF; the EXPAND X REGISTER can make the sprite longer, etc. When you work with sprites, keep in mind that a REGISTER is a BYTE that does a specific job.
   A series of registers makes up a chart called a REGISTER MAP. You can find the register map for the VIC chip in Appendix P.
   Now, another important thing about bytes is that each byte has a specific address in the computer's memory, so you can go to a specific byte if you know its address. You used this in the last chapter when we introduced PEEKing and POKEing.
   Sprites use only some of your computer's bytes. When you work with sprites, you're in closer contact with bytes than you are when you program in BASIC because you're working directly with byte addresses. When you work with sprites, you are directly controlling which bits are ON and which are OFF.
   To understand how to use sprites, you need to be able to figure exactly which bytes you want to use. In this chapter, we'll give you formulas for finding the bytes you want, and we'll give you sample programs for creating sprites. The Commodore 64 Programmer's Reference Guide goes into extensive detail on sprites.

CREATING SPRITES

Sprites are controlled by their own video display chip (VIC-II chip) in the 64. The chip does all the work of creating and keeping track of characters and graphics, creating colors, and moving the graphics around. All you have to do is tell the computer three things about the sprite:

- What it should look like
- What color it should be
- Where it should appear

The sprite's display chip has 46 different ON/OFF locations that act like internal memory locations. Each of these locations breaks down into a series of 8 bytes, and each byte can be either ON or OFF. So this means you have 368 ON/OFF locations for sprites. By POKEing the appropriate decimal value in the proper memory location, you can control the formation and movement of your sprites. We'll give you more detail about this later.

In addition to showing you sprite display chip locations, we'll show you how to use some of the 64's main memory to store data that defines the sprites. Finally, we'll use eight memory locations directly after the screen memory to tell the computer exactly which memory area each sprite will get its data from.

Sounds like a lot of details, but don't worry, you'll get the hang of it as we go through some examples.

Before you get started, here are some important things to know about sprites:

- Sprites are usually displayed in a special high resolution mode which turns the screen into a 320 dot wide by 200 dot high area.
- Each sprite object is 24 dots wide by 21 dots high.
- You can control up to eight sprites at a time. The sprites are numbered 0 to 7.
- Each sprite has individual color control.
- There's a sprite multi-color mode.
- You can use sprite-to-sprite and sprite-to-background collision detection.

But first you need to know what VIC-II chip locations control the functions you need. Remember that the sprites' chip has 46 different ON/OFF locations. Here's a partial map that shows some of these locations, which are called registers. Appendix P has a complete VIC chip register map, but the registers you'll need for sprites are listed here:

Registers	Description
0	X coordinate of sprite 0
1	Y coordinate of sprite 0
2-15	X and Y coordinates, paired like registers 0 and 1 for sprites 1-7
16	Most significant Bit — X coordinate
21	Sprite appear; A = appear, 0 = disappear
23	Expand sprite in Y direction
29	Expand sprite in X direction
39-46	Sprite 0-7 color

Now we'll show you how to write a program to create and move a sprite object. Here are some things you'll learn to do:

- Make the sprite(s) appear on the screen by POKEing into location 21, the register that turns ON sprites.
- Set the sprite pointer (memory locations 2040-2047) to show where sprite data should be read from.
- POKE actual data into memory.
- Move the sprite around by using a loop to update X and Y coordinates.
- Expand the sprite object, change colors, perform a variety of special tasks.

Suppose you want to create a balloon and have it float around. You can design the balloon in a 24 by 21 dot grid (see page 74).

The next step is to convert the graphic design into data the computer can use. Use a piece of lined paper or graphic paper to set up a sample grid that is 21 spaces down and 24 spaces across. Divide the 24 squares across the top into three sections, and number the squares in each section with this series of numbers: 128, 64, 32, 16, 8, 4, 2, 1 (see the illustration). These numbers are $2 \uparrow 7$, $2 \uparrow 6$, $2 \uparrow 5$, $2 \uparrow 4$, $2 \uparrow 3$, $2 \uparrow 2$, $2 \uparrow 1$, $2 \uparrow 0$. In other words, 2 to the power of the number of the sprite you're using. Recall that the eight sprites are numbers 0 through 7.

Number the squares down the left side 1-21 for each row. Write the word data at the end of each row. Now fill in the grid with any design, or use the balloon that we've drawn. It's easiest to outline the shape first and then go back and fill in the grid.

Think of the squares you filled in as ON, and substitute a 1 for each filled square. Think of the squares that aren't filled as OFF, and give each of them a zero.

Now you need to separate each row into thirds so you can convert the ONs and OFFs into pieces of data the computer can read. Dividing the rows is simple since you've already split each row into three sections.

Now each section of each row has eight squares, and each set of eight squares is equal to one piece of data called a BYTE. This means you have three bytes per row.

Starting on the first row of the balloon drawing and working from the left, the first eight squares are blank (0), so the value for that series (byte) is 0.

SERIES 1   SERIES 2   SERIES 3

128 32 8 2 | 128 32 8 2 | 128 32 8 2
64 16 4 1 | 64 16 4 1 | 64 16 4 1

ROW / COLUMN
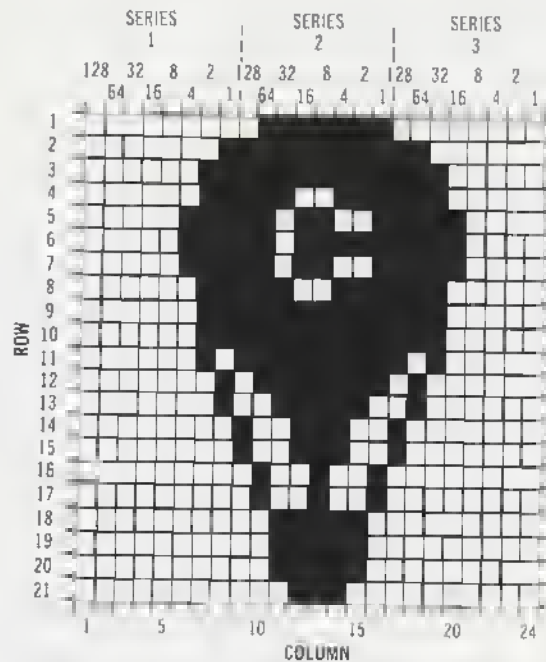
The middle series of squares looks like this (remember that a 1 is a filled dot and a 0 is a blank dot):

128	64	32	16	8	4	2	1
0	1	1	1	1	1	1	1

$$0 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 127$$
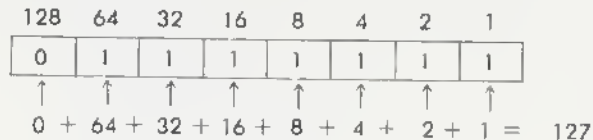
The third series of eight squares also contains only blanks, so it also equals zero. So, the data for the first line is:

DATA 0, 127, 0

The three series of dots that make up row two are calculated like this:

Series 1:

0	0	0	0	0	0	0	1

$$1 = 1$$

Series 2:

1	1	1	1	1	1	1	1

$$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

Series 3:

1	1	0	0	0	0	0	0

$$128 + 64 \qquad\qquad = 192$$
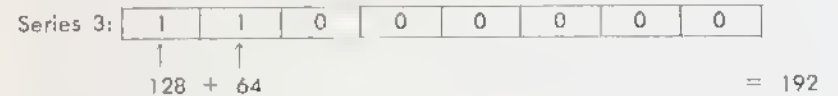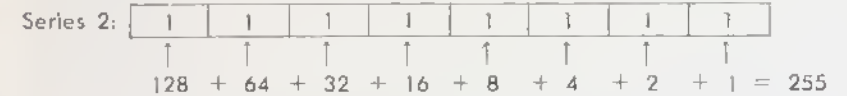
The data for the second row is:

DATA 1, 255, 192

Use this method to convert the three series of eight squares in each of the remaining rows.

Once you have done the rest of the conversions, you have the data you need to make the balloon.

Type in the following program:

```
1    REM UP, UP, AND AWAY
5    PRINT "(CLR/HOME)"
10   V = 53248 ; REM START OF DISPLAY CHIP
11   POKE V + 21,4 ; REM ENABLE SPRITE 2
12   POKE 2042,13 ; REM SPRITE 2 DATA FROM 13TH BLK
20   FOR N = 0 TO 62: READ O : POKE 832 + N,O: NEXT
30   FOR X = 0 TO 200
40   POKE V + 4,X: REM UPDATE X COORDINATES
50   POKE V + 5,X: REM UPDATE Y COORDINATES
60   NEXT X
70   GOTO 30
200  DATA 0,127,0,1,255,192,3,255,224,3,231,224
210  DATA 7,217,240,7,223,240,7,217,240,3,231,224
220  DATA 3,255,224,3,255,224,2,255,160,1,127,64
230  DATA 1,62,64,0,156,128,0,156,128,0,73,0,0,73,0
240  DATA 0,62,0,0,62,0,0,62,0,0,28,0
```

(GETS ITS INFO. FROM DATA)

(INFO. READ IN FROM O)

If you typed everything correctly, your balloon is sailing across the screen.

To understand what happened, you need to know where each sprite definition is located in memory.

Recall that each sprite is 24 by 21 dots, which equals 504 dots, which in turn equals 63 bytes (504/8 bits). So each sprite is 63 bytes plus 1 byte which is the sprite's place holder, or pointer. The pointers for each of the eight possible sprites are located together as the last eight bytes of the SCREEN MEMORY. Normally, the pointers begin at location 2040 for sprite 0, through 2047 for sprite 7:

2040	2041	2042	2043	2044	2045	2046	2047
sprite							sprite
0	1	2	3	4	5	6	7

Line 10 in the program sets V to the VIC chip's starting memory location. Then later in the program you can get the VIC chip memory location you want just by adding the appropriate register number to V.

Line 11 makes sprite 2 appear by POKEing a 4 into the SPRITE ENABLE REGISTER (21). Why POKE a 4? Because 4 is the decimal value for 2 to the second power. Recall that the decimal value for a sprite is 2 to the power of the number of the sprite. If you wanted to turn ON sprite 3, you'd use this statement: POKE V + 21,8. You turn a sprite ON by POKEing the decimal value for the sprite into the sprite enable register (21). Here's how register 21 looks after line 11 turns ON sprite 2:

	128	64	32	16	8	4	2	1	decimal values
REG. 21	0	0	0	0	0	1	0	0	
	7	6	5	4	3	2	1	0	sprite number

If you wanted to turn ON both sprites 2 and 3, you'd add 8 and 4, the decimal values for the sprites, and use this statement: POKE V + 21,12

Line 12 tells the computer where to find the data for sprite two. Recall that the pointer for sprite 2 is memory location 2042. This line POKEs a 13 into sprite 2's pointer, which means that sprite 2's data will be kept in the 13th series of sprite bytes.

Line 20 puts data into the 63 bytes that make up a sprite. READ O gets the data from the OATA statements in lines 200 through 240. The 832 in the POKE statement is the location of the 13th block of sprite bytes. As the loop in line 20 READs data, the values are POKEd into the 63 sprite bytes located in block 13.

Lines 30 through 50 control the sprite's movement across the screen. Line 40 POKEs the value of X into the VIC chip register location (4) that represents the sprite 2's X coordinate. The X coordinate represents horizontal movement. Line 50 POKEs the current value of X into the register location (5) that stands for sprite 2's Y coordinate, or vertical movement. This loop moves the sprite DOWN and TO THE RIGHT.

Line 70 starts the program over again.

Lines 200 through 240 contain the data for the sprite.

Try adding the following line and RUN the program again:

```
25 POKE V + 23,4: POKE V + 29,4: REM EXPAND SPRITE
```

Now the balloon has doubled in size because you POKEd the decimal value for sprite 2 into register 23, which controls a sprite's expansion in the Y, or vertical, direction, and into register 29, which controls a sprite's expansion in the X, or horizontal direction.

Now try adding sprite 3 to your screen by entering these lines:

```
11 POKE V + 21,12
12 POKE 2042,13: POKE 2043,13
30 FOR X = 1 TO 190
45 POKE V + 6,X
55 POKE V + 7,190 − X
```

Line 11 turns ON sprites 2 and 3 by POKEing their combined decimal values (4 and 8) into the sprite enable register (21).

Line 12 tells the computer to find the data for the sprites in the 13th block of VIC chip memory. Recall that 2042 is sprite 2's pointer, and 2043 is sprite 3's.

Lines 45 and 55 move sprite 3 around by POKEing values into sprite 3's X and Y coordinates (V + 6 and V + 7).

The following lines put sprite 4 on the screen, too:

```
11 POKE V + 21,28
12 POKE 2042,13: POKE 2043,13: POKE 2044,13
25 POKE V + 23,12: POKE V + 29,12
48 POKE V + 8,X
58 POKE V + 9,100
```

Line 11 turns ON sprites 2,3, and 4 by POKEing their combined decimal values (4, 8, and 16) into the sprite enable register (21).

Line 12 tells the computer to find the data for all three sprites in the 13th block of memory.

Line 25 doubles the size of sprites 2 and 3 by POKEing their combined decimal value into the registers that control sprite expansion (23 and 29).

Line 48 moves sprite 4 along the X axis (horizontally).

Line 58 positions sprite 4 halfway down the screen at location 100. Before, we've changed the Y coordinate throughout the program by using a loop (see line 50 in the original program). But now the value for the Y coordinate for sprite 4 (V + 9) stays the same during the program. This means that sprite 4 only moves horizontally.

MORE ON SPRITES: COLOR AND EXTRA MOVEMENT

Changing a sprite's color is even easier than moving it around the screen. You can use any of the 64's 16 colors with your sprites. The 16 colors are numbered 0 through 15. Chapter 6 and Appendix G list the colors and their codes.

Suppose you want to change sprite 1's color to light green, whose code is 13. Just enter this statement: POKE V + 45,13 (assuming you set V to 53248, the VIC chip starting memory location).

When you ran the sprite program in this chapter, you probably noticed that the sprites never moved all the way to the right edge of the screen. This happens because the X direction register can't hold a value over 255, but the screen is 320 dots wide.

There's a register on the memory map that lets you move an object across the entire screen. This register, location 16, is called the MOST SIGNIFICANT BIT (MSB) of the sprite's X direction location. In effect, this bit lets you move the sprite to a horizontal spot between 256 and 320.

Here's how the MSB works: after you've moved the sprite to X location 255, POKE the sprite's decimal value into register 16. For example, to move sprite 6 to horizontal locations 256 through 320, use this statement:

POKE V + 16,64

Then use a loop to move sprite 6 the 64 spaces from location 256 to 320:

FOR X = 0 TO 63: POKE V + 64,X: NEXT

The following program revises the original balloon program so that sprite 2 moves all the way across the screen:

```
10   V = 53248: POKE V + 21,4 : POKE 2042,13
20   FOR N = 0 TO 62 : READ O : POKE 832 + N,O : NEXT
25   POKE V + 5, 100
30   FOR X = 0 TO 255
40   POKE V + 4,X
50   NEXT
60   POKE V + 16,4
70   FOR X = 0 TO 63
80   POKE V + 4, X
90   NEXT
100  POKE V + 16,0
110  GOTO 30
```

Line 60 sets the most significant bit for sprite 2.

Lines 70 through 90 contain the loop that moves sprite 2 across screen locations 256 through 320.

Line 100 turns OFF the MSB so that sprite 2 can go back to the left edge of the screen. In other words, when the MSB is ON, the sprite can only move from locations 256 through 320. You have to turn the MSB back OFF before you can move the sprite from locations 0 through 255.

The best way to learn about sprites is to experiment with them. The Commodore 64 Programmer's Reference Guide has more information about sprites.

CHAPTER 8

MAKING MUSIC ON YOUR COMPUTER: FOR NON-PROGRAMMERS

CHAPTER 8

MAKING MUSIC ON YOUR COMPUTER: FOR NON-PROGRAMMERS

- Structure of a Sound Program
- Sample Sound Program
- Playing a Song on Your 64
- Creating Sound Effects

Your 64's sound-making features let you make music and sound effects. This chapter is an introduction to using your computer's sound chip, the SID chip. The Commodore 64 Programmer's Reference Guide has extensive information about music-makinng.

STRUCTURE OF A SOUND PROGRAM

Before we begin explaining how to program the sound chip, we want you to understand a little about the important SID chip memory locations associated with sound-making. These locations include:

- Volume
- Waveform
- High and low frequency
- The envelope generator:
  Attack
  Decay
  Sustain
  Release

VOLUME: There are 16 volume levels. To set the volume, POKE the level you want (0-15) at memory location 54296. Zero turns the volume OFF.

You only have to set the volume once at the beginning of your program. This volume setting controls all three of the 64's voices.

WAVEFORM: The waveform control STARTS and STOPS EACH NOTE. Each voice has its own waveform memory location that lets you define four different types of waveform by POKEing the start/stop numbers for the waveform you want. There are four waveform types:

Sawtooth:
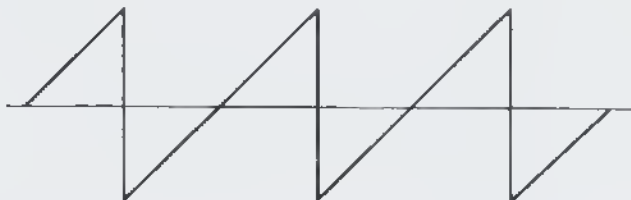
Triangular:
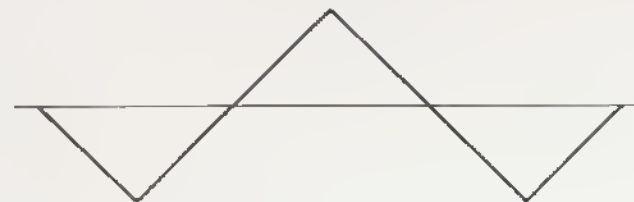
Pulse (variable rectangular waves):
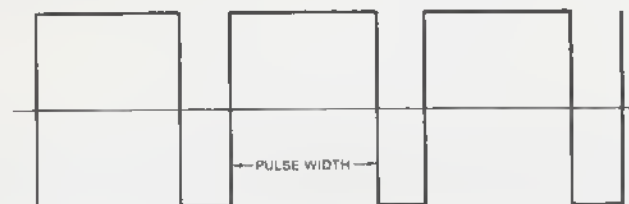
White noise (used mainly for sound effects):
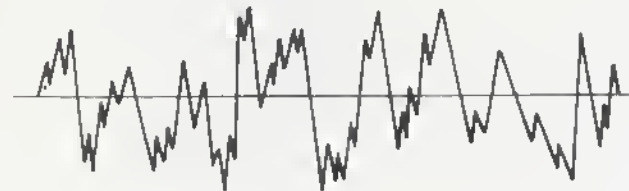
This chart shows the waveform memory locations for each voice:

WAVEFORM MEMORY LOCATIONS

VOICE 1	VOICE 2	VOICE 3
54276	54283	54290

This chart shows each waveform's start and stop numbers. These are the numbers you POKE into the memory location for the voice you're programming:

NOTE START/STOP NUMBERS

TRIANGLE ON/OFF	SAWTOOTH ON/OFF	PULSE ON/OFF	NOISE ON/OFF
17/16	33/32	65/64	129/128

For example, this statement STARTS a sawtooth waveform in voice 1, and STOPS a triangle waveform in voice 2:

POKE 54276,33: POKE 54283,16

FREQUENCY: A sound's pitch is determined by the FREQUENCY of the sound waves you produce when you make music. For each note in a music or sound program, you must include two separate POKEs: one for the high frequency byte, and one for the low frequency byte. The MUSICAL NOTE VALUE TABLE in Appendix M shows you the POKEs you need to play any note in the 64's eight octave range.

The high and low frequency memory locations are different for each voice you use, so you can program all three voices independently. This lets you create 3-voice music and exotic sound effects.

Here's a chart of the memory locations for each voice's high and low frequency bytes:

VOICE	FREQUENCY	POKE NUMBER
1	HIGH	54273
1	LOW	54272
2	HIGH	54280
2	LOW	54279
3	HIGH	54287
3	LOW	54286

Appendix M contains a list of all the high and low frequency POKE values for each note. Each note in each of the eight octaves has two different numbers.

To play a musical note, you must POKE the note's high frequency value into the high frequency location of the voice you want, and POKE the note's low frequency value into the voice's low frequency location. For example, here's how to POKE a C note from the fifth octave in voice 1:

```
10 POKE 54273,33.............................................. set high frequency
15 POKE 54272,135............................................ set low frequency
```

Use this line to POKE the same note in voice 2:

```
10 POKE 54280,33: POKE 54279,135
```

Here's a short program that POKEs C in voice 1:

```
5 FORL = 54272 TO 54296:POKEL,0:NEXT...... clear SID chip
10 V = 54296:W = 54276:A = 54277:.................. use variables to save
   S = 54278:H = 54273:L = 54272                typing time
20 POKEV,15:POKEA,190:POKES,89................ POKE vol, attack/decay
                                                 sustain/release
30 POKEH,33:POKEL,135................................. POKE hi/lo freq. notes
40 POKEW,33:FORT = 1TO200:NEXT............... start note, let it play
50 POKEW,32.................................................. stop note
```

The ENVELOPE GENERATOR: when a note is first struck, it rises from zero volume to its peak volume. The RATE at which this rise occurs is called the ATTACK. Then the note falls from the peak to some middle-range volume. The RATE of the note's fall is called the DECAY. The mid-range volume itself is called the SUSTAIN LEVEL. When the note stops playing, it falls from the SUSTAIN level to zero volume. The RATE of this fall is called the RELEASE. Here's a sketch of the four phases of a note:
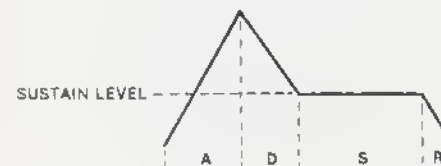
NOTE: ATTACK, DECAY, and RELEASE are RATES. SUSTAIN is a LEVEL.

In this chapter we'll refer to the ATTACK/DECAY memory locations, and the SUSTAIN/RELEASE memory locations. The reason we use these pairings is that each of the four settings needs only 1/2 byte, so attack and decay share a byte, as do sustain and release.

This chart shows you the memory locations for the ATTACK/DECAY byte and the SUSTAIN/RELEASE byte for each of the three voices:

VOICE	RATE/LEVEL	POKE NUMBER
1	attack/decay	54277
1	sustain/release	54278
2	attack/decay	54284
2	sustain/release	54285
3	attack/decay	54291
3	sustain/release	54292

We mentioned before the ATTACK and DECAY rates share the same byte. So you must COMBINE ATTACK and DECAY settings by adding them. Then you enter the total in a POKE statement.

The ATTACK rates occupy the 2 ↑ 7, 2 ↑ 6, 2 ↑ 5, and 2 ↑ 4 bits, so the values are 128, 64, 32, and 16. DECAY rates are 2 ↑ 3, 2 ↑ 2, 2 ↑ 1, and 2 ↑ 0, or 8, 4, 2, and 1. Suppose you want to set a HIGH ATTACK rate and a LOW DECAY rate: add the high attack value (128) to the low decay value (2), and POKE 130 to the appropriate memory location.

This chart shows you what numbers to POKE for ATTACK and DECAY rates:

HIGH ATTACK	MEDIUM ATTACK	LOW ATTACK	LOWEST ATTACK	HIGH DECAY	MED. DECAY	LOW DECAY	LOWEST DECAY
128	64	32	16	8	4	2	1

NOTE: You can increase the ATTACK rate by adding together all the ATTACK values: 128 + 64 + 32 + 16 = 240, which is the MAXIMUM ATTACK RATE.

You can increase the DECAY rate by adding together all the DECAY values: 8 + 4 + 2 + 1 = 15, which is the MAXIMUM DECAY RATE.

If you set an attack rate with no decay rate, the decay rate is automatically zero, and vice versa. For example, it you POKE 54277,64, you're setting a medium attack rate with zero decay tor voice 1.

Here are some sample ATTACK/DECAY POKEs:

	VOICE	ATTACK	DECAY
POKE 54277,66	1	MED (64)	LOW (2)
POKE 54284,100	2	MED (64) + LOW (32)	MED (4)
POKE 54291,15	3	ZERO	MAX (8 + 4 + 2 + 1)
POKE 54284,255	2	MAX (128 + 64 + 32 + 16	MAX + 8 + 4 + 2 + 1)

Here's a sample program that illustrates what you can do with attack/decay settings:

```
10 FORL = 54272TO54296:POKEL,0:NEXT... Clears the SID chip
20 POKE54296,15 .......................................... Set maximum volume
30 POKE54277,64 .......................................... Set attack/decay
40 POKE54273,162:POKE54272,37 .............. POKE one note in voice 1
50 PRINT"PRESS ANY KEY" ....................... Screen message
60 GETK$:IFK$ = ""THEN60 ........................ Check the keyboard
70 POKE54276,17:FORT = 1TO200:NEXT ..... Start triangle waveform
80 POKE54276,16:FORT = 1TO50:NEXT ....... Stop note
90 GOTO50 .................................................. Repeat execution
```

After you RUN the program a few times, try changing the ATTACK/DECAY setting by changing line 30:

30 POKE 54277,190

Now RUN the program again and notice the difference in the note. Try other combinations of attack and decay settings to get an idea of how you can use different attack/decay rates to create a variety of sound effects.

SUSTAIN/RELEASE SETTING. Like ATTACK/DECAY, SUSTAIN/RELEASE share a byte. But remember that this sharing doesn't mean that SUSTAIN and RELEASE are alike. SUSTAIN is a LEVEL, while release, attack and decay are RATES.

SUSTAIN is a proportion of maximum volume. You can sustain, or hold, notes and sounds at any of 16 volume levels.

This table shows you what numbers to POKE for sustain/release values:

HIGH SUSTAIN	MEDIUM SUSTAIN	LOW SUSTAIN	LOWEST SUSTAIN	HIGH RELEASE	MED. RELEASE	LOW RELEASE	LOWEST RELEASE
128	64	32	16	8	4	2	1

NOTE: You can increase the SUSTAIN level by adding together all the SUSTAIN values: 128 + 64 + 32 + 16 = 240, which is the MAXIMUM SUSTAIN LEVEL. A SUSTAIN level of 128 is approximately 50% of volume. You can increase the RELEASE rate by adding together all the RELEASE values: 8 + 4 + 2 + 1 = 15, which is the MAXIMUM RELEASE RATE.

Combine the sustain level and release rate the same way you combine the attack and decay rates: add the two values and POKE the total to the memory location of the voice you want.

To see the effects of the sustain level setting add this line to the last sample program:

35 POKE 54278,128

Now RUN the program again and note the change. With line 35, we tell the computer to sustain the note at a HIGH SUSTAIN LEVEL (128). You can vary the duration of a note by changing the count in line 70. Remember that the sustain level maintains a note at a proportion of the volume as the note falls from its peak volume; this isn't the same thing as the note's duration.

To see the effect of the release rate, try changing line 35 to POKE 54278,89 (sustain = 80, release = 9).

SAMPLE SOUND PROGRAM

This brief sound program summarizes what you've learned so far about making music on your 64:

1. Choose the voice(s) you want to use. Recall that each voice uses different memory locations into which you'll POKE values for waveform, attack rate, etc. You can play 1, 2, or 3 voice together, but this program only uses voice 1.
2. Clear the SID chip: 5 FORL = 54272 TO 54296: POKEL,0:NEXT
3. Set VOLUME: 10 POKE54296,15
4. Set ATTACK/DECAY rates:
   to define how fast a note rises
   to and falls from its peak
   volume level (0-255):   20 POKE54277,190
5. Set SUSTAIN/RELEASE to
   define level to hold note and
   rate to release it: 30 POKE54278,248
6. Find note you want to play In
   the TABLE OF MUSICAL
   NOTES in App. M and enter
   the HIGH-FREO. and LOW-
   FREO. values for that note
   (each note requires 2 POKEs):   40 POKE54273,16:POKE54272,195
7. Start WAVEFORM (here,
   TRIANGLE):   50 POKE54276,17
8. Enter a timing loop to time be-
   tween notes (we use 250 for a
   quarter note):   60 FORT = 1TO250:NEXT
9. STOP note by turning off
   chosen waveform: 70 POKE54276,16

Here's a longer program that further demonstrates your 64's music-making abillties:

```
NEW
5 REM MUSICAL SCALE ...........................
7 FORL = 54272TO54296:POKEL,0:NEXT clears SID chip
10 POKE 54296,15 ...................................... sets volume
20 POKE54277,7:POKE54278,133 ............ sets a/d/s/r
50 READ A ...................................................... READs 1st number from line 110
55 IF A = − 1 THENEND ............................ ENDs loop
60 READ B ...................................................... READs 2nd number
80 POKE54273,A:POKE54272,B ............... POKEs 1st number from line 110 as
                                              HI-FREO and 2nd number as LOW-
                                              FREO.
85 POKE54276,17 ...................................... starts note
90 FORT = 1TO250:NEXT:POKE54276,16 lets note play, then stops It
95 FORT = 1TO50:NEXT ........................... sets time for RELEASE, time be-
                                              tween notes
100 GOTO20 ................................................. restarts program
110 DATA 16,195,18,209,21,31,22,96 ........ lists note value
120 DATA 25,30,28,49,31,165,33,135 ........ from chart in App. M. Each part of
                                              numbers = one note (16 and 19 =
                                              4th octave C)
999 DATA-1 ................................................... ENDs program (see line 55)
```

You can change to a sawtooth wave by changing line 85 to read POKE54276,33 and line 90 to read FORT = 1TO250:NEXT:POKE54276,32. Changing the waveform can dramatically change the sound your computer produces.

You can also change the sound in other ways. For example, you can change the harpsichord-like sound in the previous program to a banjo-like sound by changing the ATTACK/DECAY rate of each note. Do this by changing line 20 to read:

20 POKE54277,3:POKE54278,0 .... creates banjo effect by setting zero
                                                 SUSTAIN

As this program demonstrates, your 64 can sound like a variety of musical instruments.

PLAYING A SONG ON YOUR 64

The next program lets you play a line from a song, "Michael Row Your Boat Ashore". The program uses the PULSE waveform, which is a variable width rectangular wave. The third and fourth POKEs in line 10 define the pulse width for this song.

In this song, we use a duration count of 125 for an eighth note, 250 for a quarter note, 375 for a dotted quarter note, 500 for a half note, and 1000 for a whole note. When you program your own songs, you can increase or decrease these numbers to match a particular tempo or your own musical taste.

```
2 FORL=54272TO54296: POKEL,0: NEXT
5 S=54272
10 POKES+24,15: POKES+5,88: POKES+3,15: POKES+2,15: POKES+6,89
20 READH: IFH=-1THENEND
30 READL
40 READD
60 POKES+1,H: POKES,L: POKES+4,65
70 FORT=1TOD: NEXT: POKES+4,64
80 FORT=1TO50: NEXT
90 GOTO 20
100 DATA33,135,250,42,62,250,50,60,250,42,62,125,50,60,250
105 DATA56,99,250
110 DATA50,60,500,0,0,125,42,62,250,50,60,250,56,99
115 DATA1000,50,60,500
120 DATA-1
```

Line 2 clears the SID chip.

Line 5 assigns the lowest SID chip memory location to the variable S. Throughout the rest of the program, we just add the number of the SID register to this variable. For example, POKES+24,15 POKEs 15 to the volume register, which is 54296, or 54272 + 24.

Line 10 POKEs values into:
1. The volume register: POKES+24,15
2. Voice 1, ATTACK/DELAY rates: POKES+5,88
3. Pulse width: POKES+3,15 and POKES+2,15
4. Voice 1, SUSTAIN level/RELEASE rate: POKES+6,89

Line 20 READs the first number from the DATA statement. If that number is -1, the program ENDs automatically. This occurs when the final DATA statement (line 120) is read.

Line 30 READs the second number from the DATA list.

Line 40 READs the third number from the DATA list.

Line 60 POKEs:
1. The value for H that was assigned in the READH statement in line 20. Until -1 is read, this value is assigned to the HIGH FREQUENCY register.
2. The value for L that was assigned in the READL statement in line 30. This value is assigned to the LOW FREQUENCY register. Together these two POKEs determine the pitch for one note.
3. The value that turns ON the variable pulse waveform for voice 1.

Line 70 uses a loop to set the duration for the note being played. The value for D is assigned in the READ statement in line 40. As you can see, the numbers in the DATA lists are clustered into threes: the first number (e.g., 35) is the high frequency value for a note, the second number (e.g., 135) is the low frequency value for the same note, and the third number (e.g., 250) sets the duration for that note (e.g., a quarter note C).

Line 80 is a timing loop that determines release time between notes.

Line 90 sends the program back to READ the number set for another note.

Lines 100 through 120 contain all the DATA for the line from this song.

CREATING SOUND EFFECTS

Your 64's SID chip lets you create a wide variety of sound effects, such as an explosion during a game, or a buzzer that warns you when you've made a mistake.

Here are just a few suggestions for creating sound effects:

1. Vary rapidly between two notes to create a tremor sound.
2. Use the multivoice effects to play more than one voice at a time, with each voice independently controlled, so you have different noises at once. Or use one voice as an echo or response to another voice.
3. Use the different pulse widths to create different sounds.
4. Use the NOISE WAVEFORM to make white noise to accent tonal sound effects, create explosion noises, gunshots, footsteps, or alarms. When you use the noise waveform with the same musical notes that create music, you can create different types of white noise.
5. Combine several HIGH/LOW FREQUENCIES in rapid succession across different octaves.
6. Try POKEing the extra note settings in Appendix M.

Here are some sample sound effects programs. The Commodore 64 Programmer's Reference Guide contains more examples as well as more information on creating sound effects.

DOLL CRYING

```
NEW
5 FORL = 54272TO54296:POKEL,0:NEXT..... Clears SID chip
10 S = 54272
20 POKES + 24,15 ........................................ Sets volume
30 POKES + 4,65 ......................................... Turns ON pulse waveform in
                                                          voice 1
40 POKES + 5,15 ........................................ Sets attack/decay rate
50 FORX = 200TO5STEP − 2 ....................... Sets timing loop for RELEASE or
                                                          time between notes
60 POKES + 1,40:POKES,X:NEXT ............... Sets hi/lo frequencies
70 FORX = 150TO5STEP − 2 ....................... Sets faster timing loop
80 POKES + 1,40:POKES,X:NEXT ............... Sets hi/lo frequencies
90 POKES + 4,64 ......................................... Turns OFF pulse waveform
```

SHOOTING

```
NEW
5 FORL = 54272TO54296:POKEL,0:NEXT..... Clears SID chip
10 S = 54272
20 FORX = 15TO0STEP − 1 .......................... Sets up volume loop
30 POKES + 24,X ......................................... POKEs X to vol. register.
40 POKES + 4,129 ....................................... Starts NOISE waveform
50 POKES + 5,15 ......................................... Sets ATTACK/DECAY rate
60 POKES + 1,40 ......................................... Sets high frequency
70 POKES,200:NEXT ................................... Sets low frequency
80 POKES + 4,128 ....................................... Stops NOISE waveform
90 POKES + 5,0 ........................................... POKEs 0 to attack/decay
100 GOTO20 ................................................ Repeats program
```

The loop that begins in line 20 sets up fading volume so that the sound of the gunshot starts at high volume (15) and fades to 0 as the loops executes.

Press the RUN/STOP key to end this program.

As we've said before, the best way to learn a new area of programming is to experiment.

CHAPTER **9**

ADVANCED
DATA HANDLING

CHAPTER 9

ADVANCED DATA HANDLING

- READ and DATA Statements
- Calculating Averages
- Subscripted Variables
- Dimensioning Arrays
- Simulated Dice Roll with Arrays
- Two-dimensional Arrays

READ AND DATA STATEMENTS

So far we've shown you how to assign values to variables directly (A = 2), and how to assign values while the program is RUNning (INPUT and GET).

But often you'll find that neither way suits your needs for variable assignment in a program, especially when you have large amounts of data.

In the chapter 7 when we introduced sprites, we used READ and DATA statements to assign values for sprites. Here's a short program that shows you how these two statements work together:

```
10 READ X
20 PRINT "X IS  NOW :";X
30 GOTO 10
40 DATA 1, 34, 10.5, 16, 234.56

RUN

X IS NOW : 1
X IS NOW : 34
X IS NOW : 10.5
X IS NOW : 16
X IS NOW : 234.56

?OUT OF DATA ERROR IN 10
READY
```
■

Line 10 READs a value from the DATA statement at line 40 and assigns the value to X.

Line 30 tells the computer to return to line 10, where the READ assigns the next value in the DATA statement to X. The loop continues until all the DATA values are read.

There are a few important rules you must remember when you use DATA statements:

- Follow the DATA statement format precisely:

  40 DATA 1, 34, 10.5, 16, 234.65

  Comma separates each item

- Use:
    - integer numbers (e.g., 34),
    - real numbers (e.g., 234.65),
    - numbers expressed in scientific notation (e.g., 2.4E + 04),
    - words (as long as you use a string variable in the READ statement),
  but DON'T use:
    - variables or
    - arithmetic operations
  in DATA statements. The items listed below are treated as strings if you try to READ them, and you can only READ them as strings with string variables in the READ statement.

$$DATA\ A,\ 23/56,\ 2*5,\ B\ +\ 2$$

When you use a READ statement, you can only get values from a DATA statement because the two statements work as partners. Each time you READ a value, the computer knows to move to the next value in the DATA statement. In effect, there's a pointer in the computer that keeps track of your place in the DATA statement. After READing the first value, the DATA statement looks like this:

$$40\ DATA\ 1,\ 34,\ 10.5,\ 16,\ 34.56$$
pointer

When the last DATA value has been assigned to the variable in the READ statement and the computer tries to execute the loop again, the OUT OF DATA ERROR is displayed.

Here's an example that shows one way to avoid the OUT OF DATA ERROR.

```
NEW

10 FOR X = 1 to 3
15 READ A$
20 PRINT "A$ IS NOW : "; A$
30 NEXT
40 DATA THIS, IS, FUN

RUN

A$ IS NOW : THIS
A$ IS NOW : IS
A$ IS NOW : FUN
READY
```

This time we put the READ statement inside a FOR/NEXT loop that limited the number of READings to equal the numbers of items in the DATA statement.

As long as you know how many items will be in your DATA statements, this method is fine. But often either you won't know or you won't want to bother to count.

Sometimes the best way to avoid an OUT OF DATA ERROR is to end your DATA statement with a FLAG. A flag is some value that would not ordinarily appear in your DATA list, such as a negative number, a very large number, a very small number, or a special word, such as END or STOP. When you use a flag, add an IF/THEN statement to tell the computer to branch to another part of the program when the flag is read. For example:

```
10 READ A
15 IF A < 0 THEN END
20 DATA 13, 35, 29, - 999
25 PRINT "TOTAL = "; A
30 GOTO 10
```

This program READs and PRINTs a value for A until it reaches − 999. Line 15 tells the computer to END the program immediately when a negative value is read.

There is also a way to reuse the items in a DATA statement by RESTOREing the data pointer to the beginning of the DATA list. Try adding this line:

```
45 RESTORE
```

to the second program in this chapter and RUN it again. You'll see that the data pointer has been RESTOREd to the first item in the DATA list, and that you can reREAD all the items.

CALCULATING AVERAGES

Here's a program that READs a set of numbers from a DATA list and calculates their average. This program also uses a flag to tell the computer when to stop READing DATA.

```
NEW

5  T = 0 ; CT = 0
10 READ X
20 IF X = - 1 THEN 50: REM CHECK FOR FLAG
25 CT = CT + 1
30 T = T + X : REM UPDATE TOTAL
40 GOTO 10
50 PRINT "THERE WERE "; CT;"VALUES READ"
60 PRINT "TOTAL = ";T
70 PRINT "AVERAGE = "; T/CT
80 DATA 75, 80, 62, 91, 87, 93, 78, - 1


RUN
THERE WERE 7 VALUES READ
TOTAL = 566
AVERAGE = 80.8571429
```

Line 5 sets CT, the CounTer, and T, the Total, to zero.

Line 10 READs a value from the DATA list and assigns it to X.

Line 20 checks to see if the value read to X is our flag (− 1). If it is, then the program skips lines 25-40 and goes straight to line 50.

Line 25 adds one to CT, the counter, if the value of X is not the flag.

Line 30 adds X to T, the running total.

Line 40 sends the program back to repeat line 10.

Line 50, which isn't executed until line 10 READs the flag, PRINTs the number of values read (CT).

Line 60 PRINTs the total of the numbers read (T).

Line 70 PRINTs the average.

You can also use more than one variable in the READ statement. You can mix the types of DATA in a DATA list when you also mix the types of variables in the READ statement. Here's a program that does just that. It READs a name and some scores and then calculates the average of the scores.

```
NEW

10 READ N$,A,B,C
20 PRINT N$;"'S SCORES WERE: ";A;" ";B;" ";C
30 PRINT "AND THE AVERAGE IS: ";(A + B + C)/3
40 PRINT: GOTO 10
50 DATA MIKE, 190, 185, 165, DICK, 225, 245, 190
60 DATA JOHN, 155, 185, 205, PAUL, 160, 179, 187

RUN

MIKE'S SCORES WERE: 190  185  165
AND THE AVERAGE IS : 180

DICK'S SCORES WERE: 225  245  190
AND THE AVERAGE IS : 220
```

Line 10 READs a value for each of the variables. The DATA statement lists its items in the same order that the READ statement expects to find them. In other words, there's a name to go with the string variable, and numbers to go with the integer variables.

SUBSCRIPTED VARIABLES

So far we've only used simple BASIC variables such as X and X$. It's doubtful that you'll write a program that requires more variable names than all the combinations of letters and numbers available in BASIC, but you might want to be able to group variable names together when you're using groups of data.

Subscripted variables let you use variable names so that they are obviously grouped together. For example:

A (0), A (1), A (2), A (3)

The numbers in parentheses are the SUBSCRIPTS of variable A. Be aware that the variable A1 does NOT equal the subscripted variable A (1).

You can use variables and arithmetic operation as subscripts. For example:

A (X) A (X + 1) A (4 − 1) A (2 ↑ X)

The expressions within the parentheses are evaluated according to the same rules for arithmetic operations outlined in Chapter 3.

Subscripted variables, like simple variables, name a memory location within the computer. But only subscripted variables name values that are organized into an ARRAY.

An ARRAY is understood by the computer to be a unit, such as a list or a table, of related values.

The following example uses subscripted variables to calculate an average:

```
5   PRINT CHR$(147)
10 INPUT "HOW MANY NUMBERS :";X
20 FOR A = 1 TO X
30 PRINT "ENTER VALUE # ";A;:INPUT B(A)
40 NEXT
50 SU = 0
60 FOR A = 1 TO X
70 SU = SU + B(A)
80 NEXT
90 PRINT : PRINT "AVERAGE = "; SU/X
```

RUN

```
HOW MANY NUMBERS :? 5
ENTER VALUE # 1  ? 125
ENTER VALUE # 2  ? 167
ENTER VALUE # 3  ? 189
ENTER VALUE # 4  ? 167
ENTER VALUE # 5  ? 158
```

AVERAGE = 161.2

Line 5 clears the screen.
Line 10 asks you to enter the total number of items you'll INPUT at line 30.
Line 20 sets up a loop that makes A the subscript for the array B. The loop adds 1 to A for every execution. This updates array B.
Line 30 prompts you to INPUT a value for the subscripted variable B (A).
Lines 50 through 80 keep a running total (SU) of the numbers INPUT.
Line 90 PRINTs the average.

Each time the INPUT loop executes, A is increased by 1, so the next value entered is assigned to the next element in array B. At the end of the program, array B looks like this:

B (1)	125
B (2)	167
B (3)	189
B (4)	167
B (5)	158

After you INPUT all the values, they are stored in array B. You can now access these values just by using the subscripted variables. For example, see what happens when you add these lines:

```
100 PRINT B (X − 1)
120 PRINT B (3)
130 PRINT B (X − 3)
```

DIMENSIONING ARRAYS

If you try to enter more than ten numbers in an array, you'll get a DIMENSION ERROR. Arrays of more then ten elements need to be predefined in a DIMENSION statement. For example, if you want an array to hold 25 values, you'd write this statement in your program:

DIM B (25)

You can also use a variable in a DIMension statement. For example, in the last program you could have used this statement since X equaled the total number of values in array B:

15 DIM B (X)

But be careful when you use variables to define arrays: once an array is DIMensioned, it can't be reDIMensioned in another part of the program. So don't use a variable whose value will change in the program.

You can use more than one array in a program, and you can DIMension them all on the same line:

10 DIM A (12), B (35), C (3,5)

Arrays A and B are one-dimensional arrays, but C is a two-dimensional array. One-dimensional arrays just have ROWS of data, but two-dimensional arrays have both rows and columns of data, just like a chart. Array C has 3 rows and 5 columns. Rows are always listed first in a DIMension statement.

SIMULATED DICE ROLL WITH ARRAYS

As you begin writing more complex programs, you'll find that subscripted variables cut down on the number of statements and make programs simpler to write.

For example, a single subscripted variable can keep track of the number of times each face on a die turns up in a dice roll:

```
1  REM DICE SIMULATION : PRINT CHR$(147)
10 INPUT "HOW MANY ROLLS:";X
20 FOR L = 1 TO X
30 R = INT(6*RND(1))+1
40 F(R) = F(R) + t
50 NEXT L
60 PRINT "FACE", "NUMBER OF TIMES"
70 FOR C = 1 TO 6 : PRINT C, F(C): NEXT
```

Line 10 asks you how many times you'll throw the dice iin the simulated roll.

Line 20 sets up a loop to count the number of dice rolls so that the program ends on the Xth roll.

Line 30 makes R equal to the random number rolled.

Line 40 sets up the array F, for FACE, which keeps track of how many times each face turns up. Whatever value R acquires in the dice roll becomes the subscript for the array, and line 40 adds one to the appropriate array variable. For example, every time a 2 is thrown, F (2) is increased by one.

Line 70 PRINTs the number of times each face shows up. Here's a sample RUN:

```
HOW MANY ROLLS: ? 1000
FACE                    NUMBER OF TIMES
1                       148
2                       176
3                       178
4                       166
5                       163
6                       169
```

Now we'll show you how much longer your program would be if you didn't use an array:

```
10 INPUT "HOW MANY ROLLS:";X
20 FOR L = 1 TO X
30 R = INT(6*RND(1))+1
40 IF R = 1 THEN F1 = F1 + 1 : NEXT
41 IF R = 2 THEN F2 = F2 + 1 : NEXT
42 IF R = 3 THEN F3 = F3 + 1 : NEXT
43 IF R = 4 THEN F4 = F4 + 1 : NEXT
44 IF R = 5 THEN F5 = F5 + 1 : NEXT
45 IF R = 6 THEN F6 = F6 + 1 : NEXT
60 PRINT "FACE", "NUMBER OF TIMES"
70 PRINT 1, F1
71 PRINT 2, F2
72 PRINT 3, F3
73 PRINT 4, F4
74 PRINT 5, F5
75 PRINT 6, F6
```

As you can see, the program has twice as many lines. The longer the program, the more space and time you can save when you use arrays.

TWO-DIMENSIONAL ARRAYS

As we mentioned before, two-dimensional arrays have both rows and columns, like a chart or a table. Two-dimensional arrays have two subscripts: the first one is for the ROW number; the second is for the COLUMN number. For example:

> A (4,6) has 4 ROWS
> and 6 COLUMNS

Here's what array A would look like as a two-dimensional grid in memory:

	0	1	2	3	4	5	6
0							
1							
2							
3							
4							

You'll notice that there's a zeroth row and column, so when you DIMension A (4,6), you're creating an array with 5 rows and 7 columns, or 35 elements.

You can access any element of a two-dimensional array by using its row and column subscripts. For example, suppose you want to assign 255 to A (3,4):

> 10 LET A(3,4) = 255

Here's what the grid looks like now:

	0	1	2	3	4	5	6
0							
1							
2							
3					255		
4							

Two-dimensional arrays follow the same rules as one-dimensional arrays:

DIMensioning:	DIM A (20,20)
Assigning data values:	A(1,1) = 255
Assigning values to other values:	AB = A(1,1)
PRINTing values:	PRINT A(1,1)

Here's an example of how two-dimensional arrays can be used. This example keeps track of responses to a club questionnaire like this:

CLUB QUESTIONNAIRE

Q1: ARE YOU IN FAVOR OF RESOLUTION #1?

1 — YES   2 — NO   3 — UNDECIDED

Let's suppose there are four questions, so the array, which we'll call A, will be DIMensioned A(4,3). Here's how the array table looks:

	YES	NO	UNDECIDED
QUESTION 1			
QUESTION 2			
QUESTION 3			
QUESTION 4			

The program that keeps track of the responses is on the next page. This program uses many of the programming techniques that have been presented so far.

Lines 30-65 PRINT the questions in numerical order and ask you to INPUT the response.

Line 70 adds one to the appropriate array element. Remember that R is the question number, and the questions are in rows. C is the response number, and the responses are in columns.

Line 90 asks if you have another set of responses to INPUT.

Lines 110 and 120 tell the program where to go, depending on your response to line 90.

Lines 130-170 PRINT the total number of each response.

Each time you INPUT a response at line 61, line 70 updates the right element of the array. Recall the R is the question number and C is the response number, so if your response to question 2 is 3 (undecided), line 70 adds one to array element A(2,3).

You'll notice that we didn't use the zeroth row and column in this example. You don't have to use this row and column, but remember that they are always present in every array you use.

```
20 PRINT " 〈CLR/HOME〉 "
30 FOR R = 1 TO 4
40 PRINT "QUESTION # : "; R
50 PRINT " 1-YES 2-NO 3-UNDECIDED"
60 PRINT "WHAT WAS THE RESPNSE : ";
61 GET C : IF C 〈1 or C〉 3 THEN 61
65 PRINT C: PRINT
70 A(R,C) = A(R,C) + : REM UPDATE ELEMENT
80 NEXT R
85 PRINT
90 PRINT "DO YOU WANT TO ENTER ANOTHER": PRINT
   "RESPONSE (Y/N)";
100 GET A$ : IF A$ = " " THEN 100
110 IF A$ = "Y" THEN 20
120 IF A$ 〈〉  "N" THEN 100
130 PRINT " 〈CLR/HOME 〉 ";"THE TOTAL RESPONSES
    WERE:":PRINT
140 PRINT SPC(18);"RESPONSE"
141 PRINT "QUESTION","YES","NO","UNDECIDED"
142 PRINT "-------- --------------------------"
150 FOR R = 1 TO 4
160 PRINT R, A(R,1), A(R,2), A(R,3)
170 NEXT R
RUN
```

```
QUESTION # : 1
1-YES 2-NO 3-UNDECIDED
WHAT WAS THE RESPONSE : 1

QUESTION # : 2
1-YES 2-NO 3-UNDECIDED
WHAT WAS THE RESPONSE : 1
```

And so on . . .

THE TOTAL RESPONSES WERE:

		RESPONSE	
QUESTION	YES	NO	UNDECIDED
1	6	1	0
2	5	2	0
3	7	0	0
4	2	4	1

APPENDICES

APPENDICES

INTRODUCTION

Now that you've gotten to know your 64, we want you to know that our customer support does not stop here. You may not know it, but Commodore has been in business for over 23 years. In the 1970's we introduced the first self-contained personal computer (the PET). We have since become the leading computer company in many countries of the world. Our ability to design and manufacture our own computer chips lets us bring you new and better personal computers at prices way below what you'd expect for this level of technical excellence.

Commodore is committed to supporting you, the end user. We also assist you by supporting the dealer who sold you your computer, magazines that publish how-to articles showing you new applications or techniques, and software developers who produce programs on cartridge and disk for use with your computer. We encourage you to establish or join a Commodore "user club" where you can learn new techniques, exchange ideas and share discoveries. We publish two separate magazines which contain programming tips, information on new products and ideas for computer applications. (See Appendix N).

In North America, Commodore provides a "Commodore Information Network" on the CompuServe Information Service. To use this network, all you need is your 64 computer and our low cost VICMODEM telephone interface cartridge (or other compatible modem).

The following APPENDICES contain charts, tables, and other information which help you program your Executive 64 faster and more efficiently. They also include important information on the wide variety of Commodore products you may be interested in, and a bibliography listing of over 20 books and magazines which can help you develop your programming skills and keep you current on the latest information concerning your computer and peripherals.

APPENDIX A
AVAILABLE SOFTWARE
PROGRAMMING AIDS & COMPUTER LANGUAGES

C64 101 — ASSEMBLER 64
Designed for experienced Assembly language programmers. Package includes everything needed to create, assemble, load and execute 6500 series Assembly language code. Macro assembler • Two machine language monitors • Editor and loaders • Support routines • User manual.

C64 103 — DISK BONUS PACK
The Commodore 64 DISK BONUS PACK Introduces the experienced and inexperienced user to our exciting personal computer. You will find: entertainment games, utilities to assist in your programming, music and video programs to demonstrate the abilities of the 64, and several educational and personal programs.

C64 104 — SUPER EXPANDER (VSP)
The SUPER EXPANDER 64 is a powerful extension of the BASIC language. Previously, you had to Peek or Poke specific memory registers in order to access many of the computer's features. This package provides the commands you need to use Commodore's graphics, music, and sound capabilities.

C64 105 — LOGO
LOGO is an educational language for students. Because LOGO is so versatile, it is simple enough for pre-schoolers and sophisticated enough to challenge college graduates. A small triangular cursor, called a Turtle, is used to draw the graphics.

LOGO includes the following features: four different types of screens; cataloging commands; a TRACE command for programming and debugging; LIST processing; and the ability to read music, graphics, and text files previously created with LOGO.

C64 106 — PILOT
PILOT for the 64 is the most powerful version of PILOT available. With this program, you can define your own characters, create colorful, movable objects called sprites, and create music and a variety of sound effects. This is a superior instructional tool . . . educators can write their own software which combines fun, entertainment, and learning. Programs written on COMMON PILOT can also be run without any conversions. Also, a RUN-ONLY VERSION of the program is included.

C64 107 — PET EMULATOR
Converts most software designed for Commodore PET computers to software for your 64. A valuable teaching tool utilizing your 64 and PET educational software.

C64 109 — SCREEN EDITOR

Design your own screens! The Screen Editor helps you design software on your 64 by letting you create and edit your own screens. This programming tool is for users with some computer experience. The Screen Editor helps you design computer applications with functions such as these: Screen field editor; Data field definition and editing; Cursor movement from field to field, both backwards and forwards; Quick screen design capabilities.

C64 110 — CP/M® 2.2 OPERATING SYSTEM

The CP/M® 2.2 Operating System turns your 64 into a dual processor home computer. This system lets you expand the software applications you can use with your 64. When you add this easy-to-install system, you can begin using some of the many available CP/M® programs, including widely used business applications, word processing, and high level computer languages.

'CP/M is a registered trademark of Digital Research, Inc.

BUSINESS SOFTWARE — FOR HOME AND OFFICE

C64 200 — EASY CALC 64

EasyCalc 64 is an easy-to-use electronic spread sheet. EasyCalc has editing functions and help screens, and it lets you print bar charts and individually formatted tables. EasyCalc also lets you view up to four pages at once on the screen.

C64 202 — EASY FINANCE I

Shows you how to make the most out of your hard-earned money by calculating 12 loan concepts for you. Principal, regular payment, last payment, and remaining balance are just some of the functions EASY FINANCE can determine. You need absolutely no programming knowledge to use this product.

C64 204 — EASY MAIL 64

Now your address files can be simple to manage! Keep track of names and addresses, simplify label printing. EasyMail has all the features you need to prepare special mailings by searching your address files for specific categories such as zip code. Especially useful for clubs and small businesses. Here are some of EasyMail's features: Entry, change, or deletion of name and address by name or number; One or two abreast address labels; A complete printout of all the data in your records; A HELP function that you can request at any time.

C64 207 — EASY SCRIPT

Powerful word processor with table producing capabilities; comprehensive printer controls; easy up-date facilities; easy document handling and much, much more.

C64 208 — EASY SPELL 64

Now you can produce flawless writing! EasySpell 64 offers all these features: Automatically corrects spelling errors; Counts the words in your manuscript; Has a built-in 20,000 word lexicon that lets you add words not already stored there; Is designed to be used with EasyScript 64, an easy-to-use word-processor. Because you can add to the lexicon, EasySpell can easily be adapted for writing reports in specialized fields such as medicine, law, and science. All you do is add the words you use to the lexicon!

C64 210 — WORD/NAME MACHINE

Commodore's most easy-to-learn and easy-to-use wordprocessing package. Designed for beginners and perfect for letters, address lists, and notes. Some of the features available in these companion programs, Word Machine and Name Machine: Easy-to-understand menus that let you choose what you want to do; Overtyping, inserting, and deleting of text; Three print formats: Draft, Informal, Formal; Personalized form letters; Easy-to-write name and address file that lets you select by category; Prompts for data you input into the name and address file; Prints an easy-to-use telephone and name and address book; Prints name and address labels.

C64 212 — EASY FINANCE II

Helps you make the right financial decisions by showing you how to make the most out of sixteen investment concepts. The program calculates such functions as future investment value, initial investment, and internal rate of return. EASY FINANCE II is very clear and easy to use; you need absolutely no programming knowledge to use this product.

C64 213 — EASY FINANCE III

Is an advanced version of EASY FINANCE II; it shows you how to make the most out of sixteen more investment concepts. Financial terms and practices are clarified and functions such as discount commercial paper, financial management rate of return, and financial leverage and earnings per share are calculated for you.

C64 214 — EASY FINANCE IV

Is a business management package that shows managers how to make the right decisions about production, inventory, control, compensation, and much more. Lease purchase analysis, depreciation switch, and optimal order quantity are some of the 21 functions this program will calculate for you.

C64 215 — EASY FINANCE V

Shows you how to make the most out of statistics such as payoff matrix analysis, regression analysis forecasting, and apportionment by ratios. It is completely pre-programmed and very easy to use. With just a few simple keystrokes, you can improve your finances!

C64 216 — THE MANAGER

General data base to handle user created files. Interfaces with Wordprocessing, accumulates totals on screen and creates subfiles. Sorts from any field. Powerful report printing capabilities.

C64 220 — GENERAL LEDGER

Customized Chart-of-Accounts. Convenient entry of cash receipts; disbursements; general journal transactions. Interfaces with other accounting modules for automated posting of G/L transactions. Maintains account balances monthly, quarterly, yearly, previous quarters and years. Departmental reporting and many other features.

C64 221 — RECEIVABLE/BILLING

Maintains customer master tile. Automatic billing with credit checking, item descriptions, unit pricing, and extensions when interfaced with Order Entry/Point-of-Sale and Inventory Management Modules. Flexible billing of any item and unit price when used as stand-alone system. Automatically posts billings. Interfaces with the general ledger for automated postings. Automatically ages open receivables by 30, 60, 90 day categories.

C64 222 — ACCOUNTS PAYABLE/CHECKWRITING

Maintains vendor master tile. Automatic aging of open invoices. Automatically combines invoices by vendor and prints checks with full remittance detail. Provides open item aging report by 30, 60, 90 day categories. Interfaces with the general ledger for automated postings.

C64 223 — PAYROLL

Master record of each employee pay activity. Automatically calculates period pay amount including all tax and miscellaneous deductions. Prints payroll checks with full pay deduction detail. Comprehensive management reporting including W2's and 941's. General ledger interface for automated postings.

C64 224 — INVENTORY MANAGEMENT

Perpetual inventory records for all stock items. Processes stock receipts, stock issues, stock orders, and stock adjustments with full audit trail. Optional interface to Order Entry/Point-of-Sale for automated billing of inventory items and automated stock-on-hand reduction. Capability to selectively calculate reorder level and economic order quantity by inventory category. Assists management in parts ordering by generating shortage and re-order reports.

GAMES & RECREATION

C64 601 — JUPITER LANDER   *Joystick or Keyboard*

You must carefully land your spacecraft on the only solid landing site on Jupiter. As long as your fuel holds out, you can make exploratory landings to try to find a solid place. But you must land below the yellow zone or your ship will crash.

C64 602 — KICKMAN   *Joystick or Keyboard*

In KICKMAN, you maneuver a clever unicyclist who scores points by catching balloons, ghosts, and PAC-MAN. KICKMAN uses his head and his feet to catch the falling objects. During some phases of the game, KICKMAN bursts the balloons to score points.

C64 603 — SEAWOLF   *Paddles*

As the commander of a torpedo boat, you must sink all enemy ships. Your targets are fast-moving P.T. boats, cruisers, and freighters. You can play SEAWOLF with one or two players.

C64 604 — SPEED/BINGO MATH   *Joystick or Keyboard*

These two math games help you build math skills and have fun at the same time. SPEED MATH gives you a time limit to solve a variety of math problems. BINGO MATH asks you to solve math problems and use the answer to play BINGO. You can play against the clock or against your friends.

C64 605 — RADAR RAT RACE   *Joystick or Keyboard*

In RADAR RAT RACE, you are a mouse trying to score points by eating ten pieces of cheese randomly placed in a maze. While you search for the cheese, you're pursued by rats, and you must avoid deadly black cats. You have three lives and limited time to score points.

C64 606 — CLOWNS   *Paddles*

Score points in CLOWNS by popping a skyful of colored balloons. You have two CLOWNS, one on a seesaw, and one sailing through the air bursting balloons. When the clown comes down, you must catch him with the seesaw and send the other clown up to pop balloons.

C64 609 — VISIBLE SOLAR SYSTEM   *Keyboard*

You are the commander of a spaceship on a journey through our Solar System. The ship has a cruising range of over 1 billion miles and is packed with computerized equipment to help you discover more about the planets that you visit.

C64 610 — TOOTH INVADERS   *Joystick*

In this game for children, the evil D.K. spreads plaque on your teeth. You must floss and clean the teeth before they fall out. As each tooth is cleaned, it glows and changes color. You can kill D.K. by running over him, but he always comes back! When you succeed in cleaning all the teeth, a shower of fluoride rains down from a cloud.

C64 613 — LAZARIAN   *Joystick*

You are the pilot of a space fighter stationed in a remote sector of the galaxy. Your mission is to rescue stranded starships and to defend your sector against a variety of threats. There are three phases in the game, and you get three ships to complete your mission.

C64 614 — OMEGA RACE   *Joystick, Paddles, Keyboard*

You are an Omegan ship competing against the Droids, the most powerful force in the galaxy. You must destroy the enemy ships and the photon and vapor mines the Droids have planted in space.

C64 616 — LEMANS   *Paddle*

Your car is at the pole position in the LeMans Grand Prix. You must pass as many cars as possible to score enough points to stay in the race. The race takes you over hazardous terrain, including icy roads, divided highways, dark roads, and dangerous curves. You must be careful not to wreck your car as you race to the finish line.

C64 617 — PINBALL SPECTACULAR   *Paddle*

PINBALL SPECTACULAR is like playing pinball on your 64. This game offers fast action and great sound effects.

C64 625 — ZORK I   *Keyboard (Disk)*

ZORK I involves the Great Underground Empire; it confronts you with perils and predicaments ranging from the mystical to the macabre. You strive to discover the Twenty Treasures of ZORK and escape with them and your life!

C64 626 — ZORK II
Keyboard

ZORK II gives you a helpful robot, an elusive unicorn, a princess who needs rescuing, and a demon who demands all of your treasure. Most of the creatures in ZORK II cannot be harmed by your sword, bombs, or poison, as they could in ZORK I.

C64 627 — ZORK III
Keyboard

ZORK III involves a delightful and curious place with ancient high-arched aqueduct systems, swirling mists of the Land of Shadow, and a brilliant pit of fire. You should use logical problem-solving in this game; ZORK III gives you a theme, morals, and meaning in your adventures.

C64 628 — DEADLINE
Keyboard

You are the chief of detectives trying to solve the murder of a millionaire philanthropist. You have twelve simulated hours to uncover the truth before the wrong will is read to the eager heirs. The mystery is confusing, and it's easy to get lost as you try to piece together a convincing case before it's too late.

C64 629 — STARCROSS
Keyboard

STARCROSS is another mind-bending science fiction adventure game. You are a space prospector in a one-person ship searching for black holes. You couldn't afford the best equipment, so you must do your best with a smart-aleck computer and a mass detector.

C64 630 — SUSPENDED
Keyboard

SUSPENDED takes place on a computer-controlled planet that is facing a catastrophe. The main computer has just malfunctioned; you take control and guide six robots to help you manually maintain the planet while they try to discover and repair the cause of the computer's breakdown.

ART & MUSIC

C64 402 — MUSIC MACHINE

The MUSIC MACHINE cartridge turns your 64 keyboard into a music synthesizer. Now you can fully use the music-making capabilities of your computer even if you can't program it or don't know how to play music. The MUSIC MACHINE is both fun and educational.

C64 403 — MUSIC COMPOSER

You don't have to be a musician to make music on your 64. This package teaches you everything you need to know to play songs and create sound effects on your computer. The manual also supplies sample songs for you to play.

EDUCATION & LEARNING AIDS

C64 310 — EASYLESSON 64 and EASYQUIZ 64

EasyLesson and EasyQuiz let you use your 64 as a teaching tool. With these two programs, you can easily create lessons and tests on topics that you choose. Here are some of the things you can do with these education programs: Create a pool of questions to use as a lesson; Classify questions into as many as seven categories; Print out each lesson or quiz you create; Shuffle test questions before each quiz; Give the test in either flash card or multiple choice format.

PUBLIC DOMAIN SERIES

C64 700 — BUSINESS I

This package contains fourteen programs that cover a variety of business applications. The programs include calculating interest, simulating business operations, and simulating stock transactions.

C64 701 — GEOGRAPHY I

This educational package contains thirteen quizzes about countries, states, and their capitals. There is also a program that lets you print out a map of England.

C64 702 — ENGLISH I

14 programs that teach you more about English. Word games, poetry, word definitions, and parts of speech.

C64 703 — ENGLISH II

16 programs help sharpen your language skills. Spelling and vocabulary drills, lessons about homonyms, verb forms, and punctuation.

C64 704 — ENGLISH III

Collection of 15 programs from Commodore's English language series concentrating on teaching you more about the parts of speech.

C64 705 — ENGLISH IV

15 more English language programs — drills on rhyming, unscrambling words, and spelling.

C64 706 — ENGLISH V

13 programs — mostly spelling drills and games that ask you to unscramble words. Also a quiz about some of Shakespeare's plays.

C64 707 — ENGLISH VI

This part of the English language series is made up mostly of tests on word meanings, spelling, and vocabulary. 12 programs in all.

C64 708 — ENGLISH VII

The seventh collection of English language programs offers 3 challenging word games.

C64 709 — MATHEMATICS I

Educational package drills you in arithmetic, gives you algebra lessons, and challenges you with math games.

C64 710 — MATHEMATICS II

Math educational package includes 15 programs.

C64 711 — MATHEMATICS III

Collection of math education programs offers lessons and drills on metric conversion, factoring, probability, math, and algebra. Total of 15 programs.

C64 712 — MATHEMATICS IV
Fourteen more math education programs. Lessons and drills cover factoring and algebra.

C64 713 — MATHEMATICS V
Fifteen more math education programs. Learn while you have fun.

C64 714 — MATHEMATICS VI
Collection of math education programs including more drills, quizzes, and math games. 15 programs.

C64 715 — MATHEMATICS VII
Sixteen advanced math programs — prime numbers, and roots.

C64 716 — MATHEMATICS VIII
Final package in the math education series contains eleven programs. Includes drills, games, and lessons on algebra, trigonometry, slopes and factoring.

C64 717 — COMPUTER SCIENCE I
These thirteen programs show you more about using your computer. The programs include sorting lists, drawing graphics, and creating animation.

C64 718 — SCIENCE I
Collection of science education programs, mostly about chemistry, but other sciences included.

C64 719 — SCIENCE II
13 science programs. Lessons and quizzes cover several topics in physics and chemistry and other sciences.

C64 720 — SCIENCE III
12 more science education programs, including a variety of scientific areas.

C64 721 — SCIENCE IV
This education software package contains an assortment of lessons and quizzes on various sciences, particularly biology.

C64 722 — TECHNOLOGY I
This package of ten programs covers a variety of topics related to technology and how recent advances affect our lives. You'll learn more about computers as you have fun learning.

C64 723 — HISTORY I
This package offers four challenging quizzes on history and famous people. There is also a fascinating simulation of nineteenth century American elections.

C64 724 — GAMES I
Here is an opportunity to learn while you have fun on your computer! 14 fun and educational games.

C64 725 — GAMES II
A collection of 13 educational games. Word guessing, logic games, math games and a simulation of Hi-Q.

C64 726 — GAMES III
More fun and educational games that let you play and learn on your 64. Seven games in this collection.

APPENDIX B
DESCRIPTION OF DOS ERROR MESSAGES

NOTE: Error message numbers less than 20 should be ignored with the exception of 01 which gives information about the number of files scratched with the SCRATCH command.

20: READ ERROR (block header not found)
The disk controller is unable to locate the header of the requested data block. Caused by an illegal sector number, or the header has been destroyed.

21: READ ERROR (no sync character)
The disk controller is unable to detect a sync mark on the desired track. Caused by misalignment of the read/writer head, no diskette is present, or unformatted or improperly seated diskette. Can also indicate a hardware failure.

22: READ ERROR (data block not present)
The disk controller has been requested to read or verify a data block that was not properly written. This error message occurs in conjunction with the BLOCK commands and indicates an illegal track and/or sector request.

23: READ ERROR (checksum error in data block)
This error message indicates that there is an error in one or more of the data bytes. The data has been read into the DOS memory, but the checksum over the data is in error. This message may also indicate grounding problems.

24: READ ERROR (byte decoding error)
The data or header as been read into the DOS memory, but a hardware error has been created due to an invalid bit pattern in the data byte. This message may also indicate grounding problems.

25: WRITE ERROR (write-verify error)
This message is generated if the controller detects a mismatch between the written data and the data in the DOS memory.

26: WRITE PROTECT ON
This message is generated when the controller has been requested to write a data block while the write protect switch is depressed. Typically, this is caused by using a diskette with a write a protect tab over the notch.

27: READ ERROR (checksum error in header)
The controller has detected an error in the header of the requested data block. The block has not been read into the DOS memory. This message may also indicate grounding problems.

28: WRITE ERROR (long data block)
The controller attempts to detect the sync mark of the next header after writing a data block. If the sync mark does not appear within a predetermined time, the error message is generated. The error is caused by a bad diskette format (the data extends into the next block), or by hardware failure.

29: DISK ID MISMATCH
This message is generated when the controller has been requested to access a diskette which has not been initialized. The message can also occur if a diskette has a bad header.

30: SYNTAX ERROR (general syntax)
The DOS cannot interpret the command sent to the command channel. Typically, this is caused by an illegal number of file names, or patterns are illegally used. For example, two file names may appear on the left side of the COPY command.

31: SYNTAX ERROR (invalid command)
The DOS does not recognize the command. The command must start in the first position.

32: SYNTAX ERROR (invalid command)
The command sent is longer than 58 characters.

33: SYNTAX ERROR (invalid file name)
Pattern matching is invalidly used in the OPEN or SAVE command.

34: SYNTAX ERROR (no file given)
The file name was left out of a command or the DOS does not recognize it as such. Typically, a colon (:) has been left out of the command.

39 SYNTAX ERROR (invalid command)
This error may result if the command sent to command channel (secondary address 15) is unrecognized by the DOS.

50: RECORD NOT PRESENT
Result of disk reading past the last record through INPUT#, or GET# commands. This message will also occur after positioning to a record beyond end of file in a relative file. If the intent is to expand the file by adding the new record (with a PRINT# command), the error message may be ignored. INPUT or GET should not be attempted after this error is detected without first repositioning.

51: OVERFLOW IN RECORD
PRINT# statement exceeds record boundary. Information is truncated. Since the carriage return which is sent as a record terminator is counted in the record size, this message will occur if the total characters in the record (including the final carriage return) exceeds the defined size.

52: FILE TOO LARGE
Record position within a relative file indicates that disk overflow will result.

60: WRITE FILE OPEN
This message is generated when a write file that has not been closed is being opened for reading.

61: FILE NOT OPEN
This message is generated when a file is being accessed that has not been opened in the DOS. Sometimes, in this case, a message is not generated; the request is simply ignored.

62: FILE NOT FOUND
The requested file does not exist on the indicated drive.

63: FILE EXISTS
The file name of the file being created already exists on the diskette.

64: FILE TYPE MISMATCH
The file type does not match the file type in the directory entry for the requested file.

65: NO BLOCK
This message occurs in conjunction with the B-A command. It indicates that the block to be allocated has been previously allocated. The parameters indicate the track and sector available with the next highest number. If the parameters are zero (0), then all blocks higher in number are in use.

66: ILLEGAL TRACK AND SECTOR
The DOS has attempted to access a track or block which does not exist in the format being used. This may indicate a problem reading the pointer to the next block.

67: ILLEGAL SYSTEM T OR S
This special error message indicates an illegal system track or sector.

70: NO CHANNEL (available)
The requested channel is not available, or all channels are in use. A maximum of five sequential files may be opened at one time to the DOS. Direct access channels may have six opened files.

71: DIRECTORY ERROR
The BAM does not match the internal count. There is a problem in the BAM allocation or the BAM has been overwritten in DOS memory. To correct this problem, reinitialize the diskette to restore the BAM in memory. Some active files may be terminated by the corrective action. NOTE: BAM = Block Availability Map

72: DISK FULL
Either the blocks on the diskette are used or the directory is at its entry limit. DISK FULL is sent when two blocks are available on the 1541 to allow the current file to be closed.

73: DOS MISMATCH (73, CBM DOS V2.6 1541)
DOS 1 and 2 are read compatible but not write compatible. Disks may be interchangeably read with either DOS, but a disk formatted on one version cannot be written upon with the other version because the format is different. This error is displayed whenever an attempt is made to write upon a disk which has been formatted in a non-compatible format. (A utility routine is available to assist in converting from one format to another.) This message may also appear after power up.

74: DRIVE NOT READY
An attempt has been made to access the 1541 single Drive Floppy Disk without any diskettes present in either drive.

APPENDIX C
EXECUTIVE 64 BASIC

This manual has given you an introduction to the BASIC languge — enough tor you to get a feel for computer programming and some ot the vocabulary involved. This appendix gives a complete list ot the rules (SYNTAX) ot 64 BASIC, along with concise descriptions. Please experiment with these commands. Remember, you can't damage the computer by just typing in programs, and the best way to learn computing is by experimenting.

This appendix is divided into sections according to the different types ot operations in BASIC. These include:

1. **Variables and Operators:** describes the different type ot variables, legal variable names, and arithmetic and logical operators.
2. **Commands:** describes the commands used to work with programs, such as editing, storing, and erasing.
3. **Statements:** describes the BASIC program statements used in numbered lines ot programs.
4. **Functions:** describes the string, numeric, and print tunctions.

VARIABLES

The 64 uses three types ot variables in BASIC. These are real numeric, integer numeric, and string (alphanumeric) variables.

Variable names may consisl of a single letter, a letter tollowed by a number, or two letters.

An integer variable is specitied by using the percent (%) sign after the variable name. String variables have the dollar sign ($) after their name.

EXAMPLES

Real Variable Names: A, A5, BZ
Integer Variable Names: A%, A5%, BZ%
String Variable Names: A$, A5$, BZ$

ARRAYS are lists ot variables with the same name, using numbers called subscripts lo specify the element ot the array. Arrays are detined using the DIM statement, and may contain tloating point, integer, or string variables. The array variable name is followed by a set of parentheses ( ) enclosing the number ot variables in the list.

A(7), BZ%(11), A$(50), PT(20,20)

NOTE: There are three variable names which are reserved tor use by the 64, and may not be defined by you. These variables are: ST, TI, and TI$. ST is a status variable which relates to input/output operations. The value ot ST will change it there is a problem loading a program trom disk or tape.

TI and TI$ are variables which relate to the real-time clock built into the 64. The variable TI is updaled every 1/60th ot a second. It starts at 0 when the computer is turned on, and is reset only by changing the value ot TI$.

TI$ is a string which is constantly updated by the system. The tirst two characters contain the number ot hours, the 3rd and 4th characters the number ot minutes, and the 5th and 6th characters are the number ot seconds. This variable can be given any numeric value, and will be updated from lhat point.

TI$ = "101530" sets the clock to 10:15 and 30 seconds AM.

This clock is erased when the computer is lurned ott, and starts at zero when the system is turned back on.

OPERATORS

The arithmetic operators include the following signs:

+   Addition
−   Subtraction
*   Multiplication
/   Division
↑   Raising to a power (exponentiation)

On a line containing more than one operator, there is a set order in which operations always occur. It several operations are used together on the same line, the computer assigns priorities as follows: First, exponentiation. Next, multiplication and division, and last, addition and subtraction.

You can change the order ot operations by enclosing within parentheses the calculation to be performed tirst. Operations enclosed in parentheses will take place betore other operations.

There are also operations tor equalities and inequalities:

=   Equal To
<   Less Than
>   Greater Than
<= Less Than or Equal To
>= Greater Than or Equal To
<> Not Equal To

Finally, there are lhree logical operators:

AND
OR
NOT

These are used most often to join multiple formulas in IF . . . THEN statements. For example:

IF A = B AND C = D THEN 100 (Requires both parts to be true)

IF A = B OR C = D THEN 100 (Allows either part to be true)

COMMANDS

CONT (Continue)

This command is used to restart the execution of a program which has been stopped by either using the STOP key, a STOP statement, or an END statement within the program. The program will restart at the exact place from where it lett oft.

CONT will not work if you have changed or added lines to the program (or even just moved the cursor), or if the program halted due to an error, or if you caused an error before trying to restart the program. In these cases you will get a CAN'T CONTINUE ERROR.

LIST

The LIST command allows you to look at lines of a BASIC program in memory. You can ask for the entire program to be displayed, or only certain line numbers.

LIST	Shows entire program
LIST 10 -	Shows only from line 10 until end
LIST 10	Shows only line 10
LIST – 10	Shows lines from beginning until 10
LIST 10-20	Shows line from 10 to 20, inclusive

LOAD

This command is used to transfer a program from disk into memory so the program can be used. The command is followed by a program name enclosed within quotes. The name is followed by a comma and a number or numeric variable, which acts as a device number to indicate where the program is coming from.

The disk drive is device #8.

LOAD A$,8	Looks for program whose name is in the variable A$
LOAD "HELLO",8	Looks for program called HELLO on the disk drive
LOAD "*",8	Looks for first program on disk

NEW

This command erases the entire program in memory, and also clears out any variables that may have been used. Unless the program was SAVEd, it is lost. **BE CAREFUL WHEN YOU USE THIS COMMAND.**

The NEW command can also be used as a BASIC program statement. When the program reaches this line, the program is erased. This is useful if you want to leave everything neat when the program is done.

RUN

This command causes execution of a program, once the program is loaded into memory. If there is no line number following RUN, the computer will start with the lowest line number. If a line number is designated, the program will start executing from the specified line.

RUN	Starts program at lowest line number
RUN 100	Starts execution at line 100
RUN X	UNDEFINED STATEMENT ERROR. You must always specify an actual line number, not a variable representation

SAVE

This command will store the program currently in memory on disk.

Type SAVE followed by a name in quotes or a string variable; the computer will give the program that name. The name is followed by a device number.

| SAVE A$,8 | Stores on disk with name in A$ |
| SAVE "HELLO",8 | Stores on disk with name HELLO |

VERIFY

This command causes the computer to check the program on disk against the one in memory. This is proof that the program is actually SAVEd, in case the disk is bad, or something went wrong during the SAVE.

VERIFY followed by a program name, or a string variable, will search for that program and then check. Device number is included with the verify command. command.

| VERIFY "HELLO",8 | Searches for HELLO on disk, then checks |

STATEMENTS

CLOSE

This command completes and closes any files used by OPEN statements. The number following CLOSE is the file number to be closed.

| CLOSE 2 | Only file #2 is closed |

CLR

This command will erase any variables in memory, but leaves the program itself intact. This command is automatically executed when a RUN command is given.

CMD

CMD sends the output which normally would go to the screen (i.e., PRINT statements, LISTs, but not POKEs onto the screen) to another device instead. This could be a printer, or a data file on disk. This device or file must be OPENed first. The CMD command must be followed by a number or numeric variable referring to the file.

OPEN 1,4	OPENs device #4, which is the printer
CMD 1	All normal output now goes to printer
LIST	The program listing now goes to the printer, not the screen

To send output back to the screen, CLOSE the file with CLOSE 1.

DATA

This statement is followed by a list of items to be used by READ statements. Items may be numeric values or test strings, and items are separated by commas. String items need not be inside quote marks unless they contain space, colon, or comma. If two commas have nothing between them, the value will be READ as a zero for a number, or an empty string.

DATA 12, 14.5, "HELLO, MOM", 3.14, PART1

DEF FN

This command allows you to define a complex calculation as a function with a short name. In the case of a long formula that is used many times within the program, this can save time and space.

The function name will be FN and any legal variable name (1 or 2 characters long). First you must define the function using the statement DEF followed by the function name. Following the name is a set of parentheses enclosing a numeric variable. Then follows the actual formula that you want to define, with the variable in the proper spot. You can then "call" the formula, substituting any number for the variable.

```
10 DEF FNA(X) = 12*(34.75 − X/.3)
20 PRINT FNA(7)
```
7 is inserted where
X is in the formula

For this example, the result woud be 137.

DIM

When you use more than 11 elements of an array, you must execute a DIM statement for the array. Keep in mind that the whole array takes up room in memory, so don't create an array much larger than you'll need. To figure the number of variables created with DIM, multiply the total number of elements plus one in each dimension of the array.

```
10 DIM A$(40), B7(15), CC%(4,4,4)
```

41 ELEMENTS   16 ELEMENTS   125 ELEMENTS

You can dimension more than one array in a DIM statement. However, be careful not to dimension an array more than once.

END

When a program encounters an END statement, the program halts, as if it ran out of lines. You may use CONT to restart the program.

FOR ... TO ... STEP

This statement works with the NEXT statement to repeat a section of the program a set number of times. The format is:

FOR (Var. Name) = (Start of Count) TO (End of Count) STEP (Count By)

The loop variable will be added to or subtracted from during the program. Without any STEP specified, STEP is assumed to be 1. The start count and end count are the limits to the value of the loop variable.

```
10 FOR L = 1 TO 10 STEP .1
20 PRINT L
30 NEXT L
```

The end of the loop value may be followed by the word STEP and another number or variable. In this case, the value following STEP is added each time instead of 1. This allows you to count backwards, or by fractions.

GET

The GET statement allows you to get data from the keyboard, one character at a time. When GET is executed, the character that is typed is assigned to the variable. If no character is typed, then a null (empty) character is assigned.

GET is followed by a variable name, usually a string variable. If a numeric variable was used and a nonnumeric key depressed, the program would halt with an error message. The GET statement may be placed into a loop, checking tor any empty result. This loop will continue until a key is hit.

```
10 GET A$: IF A$ = ""THEN 10
```

GET#

The GET# statement is used with a previously OPENed device or file, to input one character at a time from that device or file.

```
GET #1,A$
```

This would input one character from a data file.

GOSUB

This statement is similar to GOTO, except the computer remembers which program line it last executed before the GOSUB. When a line with a RETURN statement is encountered, the program jumps back to the statement immediately following the GOSUB. This is useful if there is a routine in your program that occurs in several parts of the program. Instead of typing the routine over and over, execute GOSUBs each time the routine is needed.

```
20 GOSUB 800
```

GOTO OR GO TO

When a statement with the GOTO command is reached, the next line to be executed will be the one with the line number following the word GOTO.

IF ... THEN

IF ... THEN lets the computer analyze a situation and take two possible courses of action, depending on the outcome. If the expression is true, the statement following THEN is executed. This may be any BASIC statement.

If the expression is false, the program goes directly to the next line.

The expression being evaluated may be a variable or formula, in which case it is considered true if nonzero, and false if zero. In most cases, there is an expression involving relational operators (=, <, >, <=, >=, <>, AND, OR NOT).

```
10 IF X > 10 THEN END
```

INPUT

The INPUT statement allows the program to get data from the user, assigning that data to a variable. The program will stop, print a question mark (?) on the screen, and wait for the user to type in the answer and hit RETURN.

INPUT is followed by a variable name, or a list of variable names, separated by commas. A message may be placed within quote marks, before the list of variable names to be INPUT. If more than one variable is to be INPUT, they must be separated by commas when typed.

```
10 INPUT "PLEASE ENTER YOUR FIRST NAME";A$
20 PRINT "ENTER YOUR CODE NUMBER";:INPUT B
```

INPUT#

INPUT# is similar to INPUT, but takes data from a previously OPENed file.

10 INPUT#1, A

LET

LET is hardly ever used in programs, since it is optional, but the statement is the heart of all BASIC programs. The variable name which is to be assigned the result of a calculation is on the left side of the equal sign, and the formula on the right.

10 LET A = 5
20 LET D$ = "HELLO"

NEXT

NEXT is always used in conjunction with the FOR statement. When the program reaches a NEXT statement, it checks the FOR statement to see if the limit of the loop has been reached. If the loop is not finished, the loop variable is increased by the specified STEP value. If the loop is finished, execution proceeds with the statement following NEXT.

NEXT may be followed by a variable name, or list of variable names, separated by commas. If there are no names listed, the loop started is the one being completed. If variables are given, they are completed in order from left to right.

10 FOR X = 1 TO 100: NEXT

ON

This command turns the GOTO and GOSUB commands into special versions of the IF statement. ON is followed by a formula, which is evaluated. If the result of the calculation is one, the first line on the list is executed; if the result is 2, the second line is executed, and so on. If the result is 0, negative, or larger than the list of numbers, the next line executed will be the statement following the ON statement.

10 INPUT X
20 ON X GOTO 10,20,30,40,50

OPEN

The OPEN statement allows the 64 to access devices such as the disk for data, a printer, or even the screen. OPEN is followed by a number (0-255), to which all following statements will refer. There is usually a second number after the first, which is the device number.

The device numbers are:

0 Screen
4 Printer
8 Disk

Following the device number may be a third number, separated again by a comma, which is the secondary address.

In the case of the disk, the number refers to the buffer, or channel, number. In the printer, the secondary address controls features like expanded printing. See the Commodore 64 Programmer's Reference Manual for more details.

10 OPEN 1,0	OPENs the SCREEN as a device
20 OPEN 2,8,8,"D"	OPENs the disk for reading, file to be searched for is D
30 OPEN 3,4	OPENs the printer
40 OPEN 4,8,15	OPENs the data channel on the disk

Also see: CLOSE, CMD, GET#, INPUT#, and PRINT#, system variable ST, and Appendix B.

POKE

POKE is always followed by two numbers, or formulas. The first location is a memory location; the second number is a decimal value from 0 to 255, which will be placed in the memory location, replacing any previously stored value.

10 POKE 53281,0
20 S = 4096*13
30 POKE S + 29,8

PRINT

The PRINT statement is the first one most people learn to use, but there are a number of variations to be aware of. PRINT can be followed by:

Text String with quotes
Variable names
Functions
Punctuation marks

Punctuation marks are used to help format the data on the screen. The comma divides the screen into four columns, while the semicolon suppresses all spacing. Either mark can be the last symbol on a line. This results in the next thing PRINTed acting as if it were a continuation of the same PRINT statement.

10 PRINT "HELLO"
20 PRINT "HELLO",A$
30 PRINT A + B
40 PRINT J;
60 PRINT A,B,C,D

Also see: POS, SPC and TAB functions

PRINT#

There are a few differences between this statement and PRINT. PRINT# is followed by a number, which refers to the device or data file previously OPENed. This number is followed by a comma and a list to be printed. The comma and semicolon have the same effect as they do in PRINT. Please note that some devices may not work with TAB and SPC.

100 PRINT#1,"DATA VALUES"; A%, B1, C$

READ

READ is used to assign information from DATA statements to variables, so the information may be put to use. Care must be taken to avoid READing strings where READ is expecting a number, which will give a TYPE MISMATCH ERROR.

REM (Remark)

REMark is a note to whomever is reading a LIST of the program. It may explain a section of the program, or give additional instructions. REM statements in no way affect the operation of the program, except to add to its length. REM may be followed by any text.

RESTORE

When executed in a program, the pointer to which an item in a DATA statement will be READ next is reset to the first item in the list. This gives you the ability to re-READ the information. RESTORE stands by itself on a line.

RETURN

This statement is always used in conjunction with GOSUB. When the program encounters a RETURN, it will go to the statement immediately following the GOSUB command. If no GOSUB was previously issued, a RETURN WITHOUT GOSUB ERROR will occur.

STOP

This statement will halt program execution. The message, BREAK IN xxx will be displayed, where xxx is the line number containing STOP. The program may be restarted by using the CONT command. STOP is normally used in debugging a program.

SYS

SYS is followed by a decimal number or numeric value in the range 0-65535. The program will then begin executing the machine language program starting at that memory location. This is similar to the USR function, but does not allow parameter passing.

WAIT

WAIT is used to halt the program until the contents of a memory location changes in a specific way. WAIT is followed by a memory location (X) and up to two variables. The format is:

WAIT X,Y,Z

The contents of the memory location are first exclusive-ORed with the third number, if present, and then logically ANDed with the second number. If the result is zero, the program goes back to that memory location and checks again. When the result is nonzero, the program continues with the next statement.

NUMERIC FUNCTIONS

ABS(X) (absolute value)

ABS returns the absolute value of the number, without its sign (+ or −). The answer is always positive.

ATN(X) (arctangent)

Returns the angle, measured in radians, whose tangent is X.

COS(X) (cosine)

Returns the value of the cosine of X, where X is an angle measured in radians.

EXP(X)

Returns the value of the mathematical constant e(2.71827183) raised to the power of X.

FNxx(X)

Returns the value of the user-defined function xx created in a DEF FNxx(X) statement.

INT(X)

Returns the truncated value of X, that is, with all the decimal places to the right of the decimal point removed. The result will always be less than, or equal to, X. Thus, any negative numbers with decimal places will become the integer less than their current value.

LOG(X) (logarithm)

Will return the natural log of X. The natural log to the base e (see EXP(X)). To convert to log base 10, simpy divide by LOG(10).

PEEK(X)

Used to find out contents of memory location X, in the range 0-65535, giving a result from 0-255. PEEK is often used in conjunction with the POKE statement.

RND(X) (random number)

RND(X) returns a random number in the range 0-1. The first random number should be generated by the formula RND(− TI) to start things off differently every time. After this, X should be a 1 or any positive number. If X is zero, the result will be the same random number as the last one.

A negative value for X will reseed the generator. The use of the same negative number for X will result in the same sequence of "random" numbers.

The formula for generating a number between X and Y is:

$$N = RND(1)*(Y - X) + X$$

where,
Y is the upper limit
X is the lower range of numbers desired.

SGN(X) (sign)

This function returns the sign (positive, negative, or zero) of X. The result will be + 1 if positive, 0 if zero, and − 1 if negative.

SIN(X) (sine)

SIN(X) is the trigonometric sine function. The result will be the sine of X, where X is an angle in radians.

SQR(X) (square root)
This function will return the square root of X, where X is a positive number or 0. If X is negative, an ILLEGAL QUANTITY ERROR results.

TAN(X) (tangent)
The result will be the tangent of X, where X is an angle in radians.

USR(X)
When this function is used, the program jumps to a machine language program whose starting point is contained in memory locations. The parameter X is passed to the machine language program, which will return another value back to the BASIC program. Refer to the Commodore 64 Programmer's Reference Manual for more details on this function and machine language programming.

STRING FUNCTIONS

ASC(X$)
This function will return the ASCII code of the first character of X$.

CHR$(X)
This is the opposite of ASC, and returns a string character whose ASCII code is X.

LEFT$(X$,X)
Returns a string containing the leftmost X characters of $X.

LEN(X$)
Returned will be the number of characters (including spaces and other symbols) in the string X$.

MID$(X$,S,X)
This will return a string containing X characters starting from the Sth character in X$.

RIGHT$(X$,X)
Returns the rightmost X characters in X$.

STR$(X)
This will return a string which is identical to the PRINTed version of X.

VAL(X$)
This function converts X$ into a number, and is essentially the inverse operation from STR$. The string is examined from the leftmost character to the right, for as many characters as are in recognizable number format.

```
10 X = VAL("123.456")          X = 123.456
10 X = VAL("12A13B")           X = 12
10 X = VAL("RIU017")           X = 0
10 X = VAL("−1.23.45.67")      X = −1.23
```

OTHER FUNCTIONS

FRE(X)
This function returns the number of unused bytes available in memory, regardless of the value of X. Note that FRE(X) will read out in negative numbers if the number of unused bytes is over 32K.

POS(X)
This function returns the number of the column (0-39) at which the next PRINT statement will begin on the screen. X may have any value and is not used.

SPC(X)
This is used in a PRINT statement to skip X spaces forward.

TAB(X)
TAB is also used in a PRINT statement; the next item to be PRINTed will be in column X.

APPENDIX D
ABBREVIATIONS FOR BASIC KEYWORDS

As a time-saver when typing in programs and commands, Commodore's BASIC for the 64 lets the user abbreviate most keywords. The abbreviation for PRINT is a question mark. The abbreviations for other words are made by typing the first one or two letters of the word, followed by the SHIFTed next letter of the word. If the abbreviations are used in a program line, the keyword will LIST in the full form.

Command	Abbreviation	Looks like this on screen	Command	Abbreviation	Looks like this on screen
ABS	A SHIFT B	A ▯	END	E SHIFT N	E ▱
AND	A SHIFT N	A ▱	EXP	E SHIFT X	E ♣
ASC	A SHIFT S	A ♥	EN	NONE	FN
ATN	A SHIFT T	A ▯	FOR	F SHIFT O	F ▯
CHR$	C SHIFT H	C ▯	FRE	F SHIFT R	E ▱
CLOSE	CL SHIFT O	CL ▯	GET	G SHIFT E	G ▱
CLR	C SHIFT L	C ▯	GET#	NONE	GET#
CMD	C SHIFT M	C ◣	GOSUB	GO SHIFT S	GO ♥
CONT	C SHIFT O	C ▯	GOTO	G SHIFT O	G ▯
COS	NONE	COS	IF	NONE	IE
DATA	D SHIFT A	D ▲	INPUT	NONE	INPUT
DEE	D SHIFT E	D ▱	INPUT#	I SHIFT N	I ▱
DIM	D SHIFT I	D ◥	INT	NONE	INT

Command	Abbreviation	Looks like this on screen	Command	Abbreviation	Looks like this on screen
LEFT$	LE SHIFT F	LE ▭	RIGHT$	R SHIFT I	R ◥
LEN	NONE	LEN	RND	R SHIFT N	R ▱
LET	L SHIFT E	L ▯	RUN	R SHIFT U	R ◪
LIST	L SHIFT I	L ◥	SAVE	S SHIFT A	S ▲
LOAD	L SHIFT O	L ▯	SGN	S SHIFT G	S ▯
LOG	NONE	LOG	SIN	S SHIFT I	S ◥
MID$	M SHIFT I	M ◥	SPC(	S SHIFT P	S ▯
NEW	NONE	NEW	SQR	S SHIFT Q	S ●
NEXT	N SHIFT E	N ▭	STATUS	ST	ST
NOT	N SHIFT O	N ▯	STEP	ST SHIFT E	ST ▭
ON	NONE	ON	STOP	S SHIFT T	S ▯
OPEN	O SHIFT P	O ▯	STR$	ST SHIFT R	ST ▭
OR	NONE	OR	SYS	S SHIFT Y	S ▯
PEEK	P SHIFT E	P ▭	TAB(	T SHIFT A	T ▲
POKE	P SHIFT O	P ▯	TAN	NONE	TAN
POS	NONE	POS	THEN	T SHIFT H	T ▯
PRINT	?	?	TIME	TI	TI
PRINT#	P SHIFT R	P ▭	TIME$	TI$	TI$
READ	R SHIFT E	R ▭	USR	U SHIFT S	U ▼
REM	NONE	REM	VAL	V SHIFT A	V ▲
RESTORE	RE SHIFT S	RE ▼	VERIFY	V SHIFT E	V ▭
RETURN	RE SHIFT T	RE ▯	WAIT	W SHIFT A	W ▲

APPENDIX E
SCREEN DISPLAY CODES

The following chart lists all of the characters built into the 64's character sets. It shows which numbers should be POKEd into screen memory (locations 1024-2023) to get a desired character. Also shown is which character corresponds to a number PEEKed from the screen.

Two character sets are available, but only one set at a time. This means that you cannot have characters from one set on the screen at the same time you have characters from the other set displayed. The sets are switched by holding down the **SHIFT** and **C=** keys simultaneously.

From BASIC, POKE 53272,21 will switch to upper case mode and POKE 53272,23 switches to lower case.

Any number on the chart may also be displayed in REVERSE. The reverse character code may be obtained by adding 128 to the values shown.

If you want to display a solid circle at location 1504, POKE the code for the circle (81) into location 1504: POKE 1504,81.

There is a corresponding memory location to control the color of each character displayed on the screen (locations 55296-56295). To change the color of the circle to yellow (color code 7) you would POKE the corresponding memory location (55776) with the character color: POKE 55776,7.

Refer to Appendix G for the complete screen and color memory maps, along with color codes.

NOTE: The following POKEs display the same symbol in set 1 and 2: 1, 27-64, 91-93, 96-104, 106-121, 123-127.

SCREEN CODES

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	C	c	3	F	f	6
A	a	1	D	d	4	G	g	7
B	b	2	E	e	5	H	h	8

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
I	I	9	%		37	♠	A	65
J	j	10	&		38		B	66
K	k	11	'		39		C	67
L	l	12	(		40		D	68
M	m	13	)		41		E	69
N	n	14	*		42		F	70
O	o	15	+		43		G	71
P	p	16	,		44		H	72
O	q	17	−		45		I	73
R	r	18	.		46		J	74
S	s	19	/		47		K	75
T	t	20	0		48		L	76
U	u	21	1		49		M	77
V	v	22	2		50		N	78
W	w	23	3		51		O	79
X	x	24	4		52		P	80
Y	y	25	5		53	●	Q	81
Z	z	26	6		54		R	82
[		27	7		55	♥	S	83
£		28	8		56		T	84
]		29	9		57		U	85
↑		30	:		58	⊠	V	86
←		31	;		59	○	W	87
SPACE		32	<		60	♣	X	88
!		33	=		61		Y	89
"		34	>		62	♦	Z	90
#		35	?		63			91
$		36			64			92

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
[graphic]		93	[graphic]	[graphic]	105	[graphic]		117
[graphic]	[graphic]	94	[graphic]		106	[graphic]		118
[graphic]	[graphic]	95	[graphic]		107	[graphic]		119
SPACE		96	[graphic]		108	[graphic]		120
[graphic]		97	[graphic]		109	[graphic]		121
[graphic]		98	[graphic]		110	[graphic]	✓	122
[graphic]		99	[graphic]		111	[graphic]		123
[graphic]		100	[graphic]		112	[graphic]		124
[graphic]		101	[graphic]		113	[graphic]		125
[graphic]		102	[graphic]		114	[graphic]		126
[graphic]		103	[graphic]		115	[graphic]		127
[graphic]		104	[graphic]		116			

Codes from 128-255 are reversed images of codes 0-127.

APPENDIX F
ASCII AND CHR$ CODES

This appendix shows you what characters will appear if you PRINT CHR$(X), for all possible values of X. It will also show the values obtained by typing PRINT ASC("x"), where x is any character you can type. This is useful in evaluating the character received in a GET statement, converting upper/lower case, and printing character based commands (like switch to upper/lower case) that could not be enclosed in quotes.

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	0	CRSR ↓	17	"	34	3	51
	1	RVS ON	18	#	35	4	52
	2	CLR HOME	19	$	36	5	53
	3	INST DEL	20	%	37	6	54
	4		21	&	38	7	55
WHT	5		22	'	39	8	56
	6		23	(	40	9	57
	7		24	)	41	:	58
DISABLES SHIFT C=	8		25	*	42	;	59
ENABLES SHIFT C=	9		26	+	43	<	60
	10		27	,	44	=	61
	11	RED	28	−	45	>	62
	12	CRSR	29	.	46	?	63
RETURN	13	GRN	30	/	47	@	64
SWITCH TO LOWER CASE	14	BLU	31	0	48	A	65
	15	SPACE	32	1	49	B	66
	16	!	33	2	50	C	67

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
D	68	♠	97	⊓	126	Grey 3	155
E	69	[graphic]	98	◣	127	PUR	156
F	70	[graphic]	99		128	CRSR	157
G	71	[graphic]	100	Orange	129	YEL	158
H	72	[graphic]	101		130	CYN	159
I	73	[graphic]	102		131	SPACE	160
J	74	[graphic]	103		132	[graphic]	161
K	75	[graphic]	104	f1	133	[graphic]	162
L	76	[graphic]	105	f3	134	[graphic]	163
M	77	[graphic]	106	f5	135	[graphic]	164
N	78	[graphic]	107	f7	136	[graphic]	165
O	79	[graphic]	108	f2	137	[graphic]	166
P	80	[graphic]	109	f4	138	[graphic]	167
Q	81	[graphic]	110	f6	139	[graphic]	168
R	82	[graphic]	111	f8	140	[graphic]	169
S	83	[graphic]	112	SHIFT RETURN	141	[graphic]	170
T	84	●	113	SWITCH TO LOWER CASE	142	[graphic]	171
U	85	[graphic]	114		143	[graphic]	172
V	86	♥	115	BLK	144	[graphic]	173
W	87	[graphic]	116	CRSR	145	[graphic]	174
X	88	[graphic]	117	RVS OFF	146	[graphic]	175
Y	89	⊠	118	CLR HOME	147	[graphic]	176
Z	90	○	119	INST DEL	148	[graphic]	177
'[	91	♣	120	Brown	149	[graphic]	178
£	92	[graphic]	121	Lt. Red	150	[graphic]	179
]	93	♦	122	Grey 1	151	[graphic]	180
↑	94	[graphic]	123	Grey 2	152	[graphic]	181
←	95	[graphic]	124	Lt. Green	153	[graphic]	182
[graphic]	96	[graphic]	125	Lt. Blue	154	[graphic]	183

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
[graphic]	184	[graphic]	186	[graphic]	188	[graphic]	190
[graphic]	185	[graphic]	187	[graphic]	189	[graphic]	191

CODES	192-223	SAME AS	96-127
CODES	224-254	SAME AS	160-190
CODE	255	SAME AS	126

APPENDIX G
SCREEN AND COLOR MEMORY MAPS

The following charts list which memory locations control placing characters on the screen, which locations are used to change individual character colors, and they show character color codes.

SCREEN MEMORY MAP

COLUMN

The actual values to POKE into a color memory location to change a character's color are:

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

For example, to change the color of a character located at the upper left-hand corner of the screen to red, type: POKE 55296,2.

COLOR MEMORY MAP

COLUMN
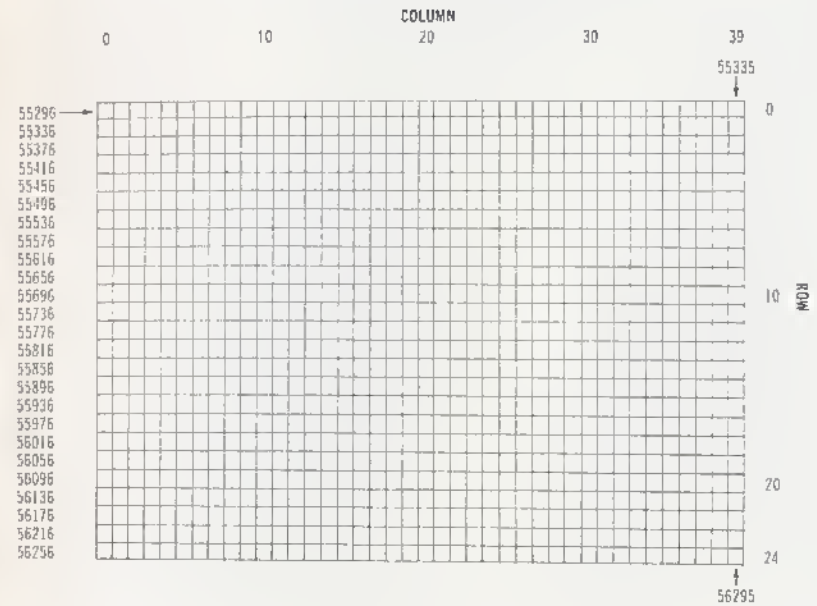
APPENDIX H
DERIVING MATHEMATICAL FUNCTIONS

Functions that are intrinsic to 64 BASIC may be calculated as follows:

FUNCTION	BASIC EQUIVALENT
SECANT	SEC(X) = 1/COS(X)
COSECANT	CSC(X) = 1/SIN(X)
COTANGENT	COT(X) = 1/TAN(X)
INVERSE SINE	ARCSIN(X) = ATN(X/SQR(– X*X + 1))
INVERSE COSINE	ARCCOS(X) = – ATN(X/SQR(– X*X + 1)) + π/2
INVERSE SECANT	ARCSEC(X) = ATN(X/SQR(X*X – 1))
INVERSE COSECANT	ARCCSC(X) = ATN(X/SQR(X*X – 1)) + (SGN(X) – 1* π/2)
INVERSE COTANGENT	ARCOT(X) = ATN(X) + π/2
HYPERBOLIC SINE	SINH(X) = (EXP(X) – EXP(– X))/2
HYPERBOLIC COSINE	COSH(X) = (EXP(X) + EXP(– X))/2
HYPERBOLIC TANGENT	TANH(X) = EXP(– X)/(EXP(x) + EXP(– x))*2 + 1
HYPERBOLIC SECANT	SECH(X) = 2/(EXP(X) + EXP(– X))
HYPERBOLIC COSECANT	CSCH(X) = 2/(EXP(X) – EXP(– X))
HYPERBOLIC COTANGENT	COTH(X) = EXP(– X)/(EXP(X) – EXP(– X))*2 + 1
INVERSE HYPERBOLIC SINE	ARCSINH(X) = LOG(X + SQR(X*X + 1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X) = LOG(X + SQR(X*X – 1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X) = LOG((1 + X)/(1 – X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X) = LOG((SQR(– X*X + 1) + 1/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X) = LOG((SGN(X)*SQR(X*X + 1/x)
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X) = LOG((X + 1)/(X – 1))/2

APPENDIX I
PINOUTS FOR INPUT/OUTPUT DEVICES

This appendix is designed to show you what connections may be made to the Executive 64.

1) Game I/O
2) Cartridge Slot
3) Audio/Video
4) Serial I/O (Disk/Printer)
5) Modulator Output
6) User Port

Control Port 1

Pin	Type	Note
1	JOYAO	
2	JOYA1	
3	JOYA2	
4	JOYA3	
5	POT AY	
6	BUTTON A/LP	
7	+ 5V	MAX. 50mA
8	GND	
9	POT AX	
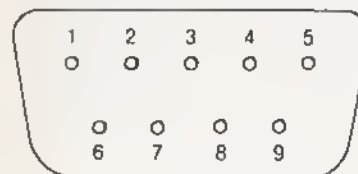
Control Port 2

Pin	Type	Note
1	JOYB0	
2	JOYB1	
3	JOYB2	
4	JOYB3	
5	POT BY	
6	BUTTON B	
7	+ 5V	MAX. 50mA
8	GND	
9	POT BX	

Cartridge Expansion Slot

Pin	Type
1	GND
2	+ 5V
3	+ 5V
4	IRQ
5	R/W
6	Dot Clock
7	I/O 1
8	GAME
9	EXROM
10	I/O 2
11	ROML

Pin	Type
A	GND
B	ROMH
C	RESET
D	NMI
E	S 02
F	A15
H	A14
J	A13
K	A12
L	A11
M	A10

Cartridge Expansion Slot

Pin	Type
12	BA
13	DMA
14	D7
15	D6
16	D5
17	D4
18	D3
19	D2
20	D1
21	D0
22	GND

Pin	Type
N	A9
P	A8
R	A7
S	A6
T	A5
U	A4
V	A3
W	A2
X	A1
Y	A0
Z	GND

```
22 21 20 19 18 17 16 15 14 13 12 11 10 9  8  7  6  5  4  3  2  1
■■■■■■■■■■■■■■■■■■■■■■
 Z  Y  X  W  V  U  T  S  R  P  N  M  L  K  J  H  F  E  D  C  B  A
```

Audio/Video: 8 Pin

Pin	Type	Note
1	LUM/SYNC	Luminance/SYNC (monoclo) output
2	GND	
3	AUDIO OUT	
4	VIDEO OUT	Composit signal output
5	AUDIO IN	
6	COLOR OUT	Chroma signal output
7	NC	No connection
8	NC	No connection
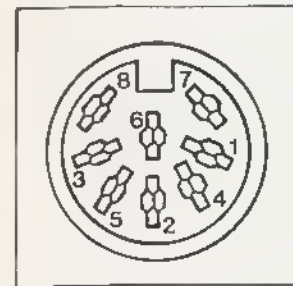
Serial I/O

Pin	Type
1	SERIAL SROIN
2	GND
3	SERIAL ATN IN/OUT
4	SERIAL CLK IN/OUT
5	SERIAL DATA IN/OUT
6	RESET

User I/O
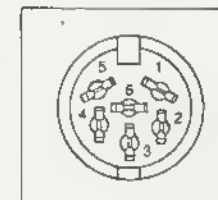
Pin	Type	Note
1	GND	
2	+ 5V	MAX. 100mA
3	RESET	
4	CNTI	
5	SP1	
6	CNT2	
7	SP2	
8	PC2	
9	SER. ATN IN	
10	9 VAC	MAX. 100mA
11	9WAC 18V cc	MAX. 100mA
12	GND	

Pin	Type	Note
A	GND	
B	FLAG2	
C	PB0	
D	PB1	
E	PB2	
F	PB3	
H	PB4	
J	PB5	
K	PB6	
L	PB7	
M	PA2	
N	GND	

```
1  2  3  4  5  6  7  8  9  10 11 12
```
```
A  B  C  D  E  F  H  J  K  L  M  N
```

142

APPENDIX J
PROGRAMS TO TRY

We've included a number of useful programs for you to try with your 64. These programs will prove both entertaining and useful.

```
100 print"█jotto      jim butterfield"
120 input"█want instructions";z$:ifasc(z$)=78goto250
130 print"█try to guess the mystery 5-letter word"
140 print"█you must guess only legal 5-letter"
150 print"words, too..."
160 print"you will be told the number of matches"
170 print"(or 'jots') of your guess."
180 print"█hint: the trick is to vary slightly"
190 print"  from one guess to the next: so that"
200 print"  if you guess 'batch' and get 2 jots"
210 print"  you might try 'botch' or 'chart'"
220 print"  for the next guess..."
250 data bxbsf,lpccz,dbdif,esfbe,pggbm
260 data hpshf,lbudi,djwjm,kpmmz,lbzbl
270 data sbkbi,mfwfm,njnjd,boofy,qjqfs
280 data rvftu,sjwfs,qsftt,puufs,fwfou
290 data xfbwf,fyupm,nvtiz,afcsb,gjaaz
300 data uljdl,esvol,gmppe,ujhfs,gblfs
310 data cppui,mzjoh,trvbu,hbvaf,pxjoh
320 data uisff,tjhlu,bymft,hsvnq,bsfob
330 data rvbsu,dsffq,cfmdi,qsftt,tqbsl
340 data sbebs,svsbm,tnfum,gspxo,esjgu
400 n=50
410 dim n$(n),z(5),y(5)
420 for j=1ton:readn$(j):nextj
430 t=ti
440 t=t/1000:ift>=1thengoto440
450 z=rnd(-t)
500 g=0:n$=n$(rnd(1)*n+1)
510 print "█i have a five letter word:":ifr>0goto560
520 print "guess (with legal words)"
530 print "and i'll tell you how many"
540 print "'jots', or matching letters,"
550 print "you have...."
560 g=g+1:input "your word";z$
570 if len(z$)<>5thenprint"you must guess a
    5-letter word!":goto560
580 v=0:h=0:m=0
590 forj=1to5
600 z=asc(mid$(z$,j,1)):y=asc(mid$(n$,j,1))-1:ify=64theny=90
610 ifz<65orz>90thenprint"that's not a word!":goto560
620 ifz=65orz=69orz=73orz=79orz=85orz=89thenv=v+1
630 ifz=ythenm=m+1
640 z(j)=z:y(j)=y:nextj
650 ifm=5goto800
660 ifv=0orv=5thenprint"come on..what kind of
    a word is that?":goto560
670 for j=1to5:y=y(j)
680 for k=1to5:ify=z(k)thenh=h+1:z(k)=0:goto700
690 next k
700 next j
710 print"█████████████████████";H;"JOTS"
720 ifg<30goto560
730 print"i'd.better tell you.. word was '";
```

143

Left column (page 144):

```
740 forj=1to5:printchr$(y(j));:nextj
750 print"'":goto810
800 print"you got it in only";g;"guesses."
810 input"@another word";z$
820 r=1:ifasc(z$)<>78goto500


1 rem *** sequence
2 rem
3 rem *** from pet user group
4 rem *** software exchange
5 rem *** po box 371
6 rem *** montgomeryville, pa 18936
7 rem
50 dim a$(26)
100 z$="abcdefghijklmnopqrstuvwxyz"
110 z1$="12345678901234567890123456"
200 print"@@@enter length of string to be sequenced@"
220 input "maximum length is 26 ";s%
230 if s%<1 or s%>26 then 200
240 s=s%
300 for i=1 to s
310 a$(i)=mid$(z$,i,1)
320 next i
400 rem randomize string
420 for i=1 to s
430 k=int(rnd(1)*s+1)
440 t$=a$(i)
450 a$(i)=a$(k)
460 a$(k)=t$
470 next i
480 gosub 950
595 t=0
600 rem reverse substring
605 t=t+1
610 input "how many to reverse ";r%
620 if r%=0 goto 900
630 if r%>0 and r%<=s goto 650
640 print "must be between 1 and ";s: goto 610
650 r=int(r%/2)
660 for i=1 to r
670 t$=a$(i)
680 a$(i)=a$(r%-i+1)
690 a$(r%-i+1)=t$
700 next i
750 gosub 950
800 c=1: for i=2 to s
810 if a$(i)>a$(i-1) goto 830
820 c=0
830 next i
840 if c=0 goto 600
850 print "@you did it in ";t;" tries"
900 rem check for another game
910 input "@want to play again ";y$
920 if left$(y$,1)="y" or y$="ok" or y$="1" goto 200
930 end
950 print
960 print left$(z1$,s)
970 for i=1 to s: print a$(i);:next i
980 print "@"
990 return
```

This program courtesy of Gene Deals

144

Right column (page 145):

```
90 REM PIANO KEYBOARD
100 PRINT"□ ▨ ▨ ▨ | ▨ ▨ ▨ | ▨ ▨ | ▨ ▨ "
110 PRINT" ▨ ▨ ▨ | ▨ ▨ ▨ | ▨ ▨ | ▨ ▨ "
120 PRINT" ▨ ▨ ▨ | ▨ ▨ ▨ | ▨ ▨ | ▨ ▨ "
130 PRINT" ▨ | | | | | | | | | | | 1 "
140 PRINT" ▨Q W E R T Y U I O P @ * ↑ "
150 PRINT"@'SPACE' FOR SOLO OR POLYPHONIC"
160 PRINT"@'F1,F3,F5,F7' OCTAVE SELECTION"
170 PRINT"@'F2,F4,F6,F8' WAVEFORM@"
180 PRINT"HANG ON, SETTING UP FREQUENCY TABLE..."
190 S=13*4096+1024:DIMF(26):DIMN(293)
200 FORI=0TO28:POKES+I,0:NEXT
210 F1=7040:FORI=1TO26:F(27-I)=F1,8+.30:F1=F1/2^(1/12):NEXT
220 K$="Q2WDEP5T6YU19OOP@-+↑N"
230 FORI=1TOLEN(K$):N(ASC(MID$(K$,I))=I:NEXT
240 PRINT"□                              "
250 AT=0:DE=0:SU=15:RE=9:SV=SU+16*RE:AV=AT*16+DE:
    WV=16:N=0:M=1:OC=4:HB=256:Z=T
260 FORI=0TO2:POKES+5+I*7,AT*16+DE:POKES+6+I*7,SU*16+RE
270 POKES+2+I*7,4000AND255:POKES+3+I*7,4000/256:NEXT
280 POKES+24,15:REM+16+64:POKES+23,7
300 GETA$:IFA$=""THEN300
310 FP=F(K)ASC(A$))*V/M:T=V+T:CP=S+T*4:IFFP=0THEN500
320 POKES+6+T,Z:REM FINISH DEC/SUS
325 POKES+5+T,Z:REM FINISH ATT/REL
330 POKECP,8:POKECP,0:REM FIX OFF
340 POKES+T,FP-HB*INT(FP/HB):REM SET LO
350 POKES+1+T,FP/HB:REM SET HI
360 POKES+6+T,SV:REM SET DEC/SUS
365 POKES+5+T,AV:REM SET ATT/REL
370 POKECP,WV+1:FORI=1TO50+AT:NEXT
375 POKECP,WV:REM PULSE
380 IFP=1THENV=V+1:IFV=3THENV=0
400 GOTO300
500 IFA$="▨"THENM=1:OC=4:GOTO300
510 IFA$="▨"THENM=2:OC=3:GOTO300
520 IFA$="▨"THENM=4:OC=2:GOTO300
530 IFA$="▨"THENM=8:OC=1:GOTO300
540 IFA$="▨"THENM=0:WV=16:GOTO300
550 IFA$="▨"THENN=1:WV=32:GOTO300
560 IFA$="▨"THENN=2:WV=64:GOTO300
570 IFA$="▨"THENN=3:WV=128:GOTO300
580 IFA$=" "THENP=1-P:GOTO300
590 IFA$="□"THEN200
590 GOTO300
590 PRINT"HIT A KEY"
810 GETA$:IFA$=""THEN810:WAIT FOR A KEY
820 PRINTA$:RETURN
```

NOTES:

Line 100 uses (SHIFT CLR/HOME),	Line 530 uses (I7)
(CTRL 9),(CTRL 1),(SHIFT 8).	Line 540 uses (I2)
Line 150 uses (CRSR DOWN)	Line 550 uses (I4)
Line 240 uses (CRSR UP)	Line 560 uses (I6)
Line 500 uses (I1)	Line 570 uses (I8)
Line 510 uses (I3)	Line 590 uses (SHIFT CLR/HOME)
Line 520 uses (I5)	

145

APPENDIX K
CONVERTING STANDARD BASIC PROGRAMS TO EXECUTIVE 64 BASIC

If you have programs written in a BASIC other than Commodore BASIC, some minor adjustments may be necessary before running them on the 64. We've included some hints to make the conversion easier.

String Dimensions

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the Commodore BASIC statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the Commodore BASIC operator for string concatenation.

In Commodore 64 BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to J, must be changed as follows:

Other BASIC	Commodore BASIC
A$(I) = X$	A$ = LEFT$(A$,I − 1) + X$ + MID$(A$,I + 1)
A$(I,J) = X$	A$ = LEFT$(A$,I − 1) + X$ + MID$(A$,J + 1)

Multiple Assignments

To set B and C equal to zero, some BASICs allow statements of the form:

10 LET B = C = 0

Commodore BASIC on the 64 would interpret the second equal sign as a logical operator and set B = −1 if C = 0. Instead, convert this statement to:

10 C = 0 : B = 0

Multiple Statements

Some BASICs use a backslash (/) to separate multiple statements on a line. With Commodore BASIC, separate all statements by a colon (:).

MAT Functions

Programs using the MAT functions available on some BASICs must be rewritten using FOR . . . NEXT loops to execute properly.

APPENDIX L
ERROR MESSAGES

This appendix contains a complete list of the error messages generated by the 64, with a description of causes.

BAD DATA String data was received from an open file, but the program was expecting numeric data.

BAD SUBSCRIPT The program was trying to reference an element of an array whose number is outside of the range specified in the DIM statement.

BREAK Program execution was stopped because you hit the STOP key.

CAN'T CONTINUE The CONT command will not work, either because the program was never RUN, there has been an error, or a line has been edited.

DEVICE NOT PRESENT The required I/O device was not available for an OPEN, CLOSE, CMD, PRINT#, INPUT#, or GET#.

DIVISION BY ZERO Division by zero is a mathematical oddity and not allowed.

EXTRA IGNORED Too many items of data were typed in response to an INPUT statement. Only the first few items were accepted.

FILE NOT FOUND No file with that name exists.

FILE NOT OPEN The file specified in a CLOSE, CMD, PRINT#, INPUT#, or GET#, must first be OPENed.

FILE OPEN An attempt was made to open a file using the number of an already open file.

FORMULA TOO COMPLEX The string expression being evaluated should be split into at least two parts for the system to work with, or a formula has too many parentheses.

ILLEGAL DEVICE NUMBER Occurs when you try to access a device illegally (e.g., LOADing from keyboard, screen, or RS-232C).

ILLEGAL DIRECT The INPUT statement can only be used within a program, and not in direct mode.

ILLEGAL QUANTITY A number used as the argument of a function or statement is out of the allowable range.

LOAD There is a problem with the program on disk.

MISSING FILE NAME LOADs and SAVEs from the serial port (e.g., the disk) require a file name to be supplied. Key in the file name.

NEXT WITHOUT FOR This is caused by either incorrectly nesting loops or having a variable name in a NEXT statement that doesn't correspond with one in a FOR statement.

NOT INPUT FILE An attempt was made to INPUT or GET data from a file which was specified to be for output only.

NOT OUTPUT FILE An attempt was made to PRINT data to a file which was specified as input only.

OUT OF DATA A READ statement was executed but there is no data left unREAD in a DATA statement.

OUT OF MEMORY There is no more RAM available for program or variables. This may also occur when too many FOR loops have been nested, or when there are too many GOSUBs in effect.

OVERFLOW The result of a computation is larger than the largest number allowed, which is 1.70141884E + 38.

REDIM'D ARRAY An array may only be DIMensioned once. If an array variable is used before that array is DIM'd, an automatic DIM operation is performed on that array setting the number of elements to ten, and any subsequent DIMs will cause this error.

REDO FROM START Character data was typed in during an INPUT statement when numeric data was expected. Just re-type the entry so that it is correct, and the program will continue by itself.

RETURN WITHOUT GOSUB A RETURN statement was encountered, and no GOSUB command has been issued.

STRING TOO LONG A string can contain up to 255 characters.

?SYNTAX ERROR A statement is unrecognizable by the 64. A missing or extra parentheses, misspelled keywords, etc.

TOO MANY FILES You tried to OPEN more than 10 files at one time.

TYPE MISMATCH This error occurs when a number is used in place of a string, or vice-versa.

UNDEF'D FUNCTION A user defined function was referenced, but it has never been defined using the DEF FN statement.

UNDEF'D STATEMENT An attempt was made to GOTO or GOSUB or RUN a line number that doesn't exist.

VERIFY The program on disk does not match the program currently in memory.

APPENDIX M
MUSIC NOTE VALUES

This appendix contains a complete list of Note#, actual note, and the values to be POKEd into the HI FREQ and LOW FREQ registers of the sound chip to produce the indicated note.

MUSICAL NOTE		OSCILLATOR FREQ		
0	C-0	268	1	12
1	C#-0	284	1	28
2	D-0	301	1	45
3	D#-0	318	1	62
4	E-0	337	1	81
5	F-0	358	1	102
6	F#-0	379	1	123
7	G-0	401	1	145
8	G#-0	425	1	169
9	A-0	451	1	195
10	A#-0	477	1	221
11	B-0	506	1	250
16	C-1	536	2	24
17	C#-1	568	2	56
18	D-1	602	2	90
19	D#-1	637	2	125
20	E-1	675	2	163
21	F-1	716	2	204
22	F#-1	758	2	246
23	G-1	803	3	35
24	G#-1	851	3	83
25	A-1	902	3	134
26	A#-1	955	3	187
27	B-1	1012	3	244
32	C-2	1072	4	48
33	C#-2	1136	4	112
34	D-2	1204	4	180
35	D#-2	1275	4	251
36	E-2	1351	5	71
37	F-2	1432	5	152
38	F#-2	1517	5	237
39	G-2	1607	6	71
40	G#-2	1703	6	167
41	A-2	1804	7	12
42	A#-2	1911	7	119
43	B-2	2025	7	233
48	C-3	2145	8	97

MUSICAL NOTE		OSCILLATOR FREQ		
49	C#-3	2273	8	225
50	D-3	2408	9	104
51	D#-3	2551	9	247
52	E-3	2703	10	143
53	F-3	2864	11	48
54	F#-3	3034	11	218
55	G-3	3215	12	143
56	G#-3	3406	13	78
57	A-3	3608	14	24
58	A#-3	3823	14	239
59	B-3	4050	15	210
64	C-4	4291	16	195
65	C#-4	4547	17	195
66	D-4	4817	18	209
67	D#-4	5103	19	239
68	E-4	5407	21	31
69	F-4	5728	22	96
70	F#-4	6069	23	181
71	G-4	6430	25	30
72	G#-4	6812	26	156
73	A-4	7217	28	49
74	A#-4	7647	29	223
75	B-4	8101	31	165
80	C-5	8583	33	135
81	C#-5	9094	35	134
82	D-5	9634	37	162
83	D#-5	10207	39	223
84	E-5	10814	42	62
85	F-5	11457	44	193
86	F#-5	12139	47	107
87	G-5	12860	50	60
88	G#-5	13625	53	57
89	A-5	14435	56	99
90	A#-5	15294	59	190
91	B-5	16203	63	75
96	C-6	17167	67	15
97	C#-6	18188	71	12
98	D-6	19269	75	69
99	D#-6	20415	79	191
100	E-6	21629	84	125
101	F-6	22915	89	131
102	F#-6	24278	94	214
103	G-6	25721	100	121
104	G#-6	27251	106	115
105	A-6	28871	112	199
106	A#-6	30588	119	124

MUSICAL NOTE		OSCILLATOR FREQ		
107	B-6	32407	126	151
112	C-7	34334	134	30
113	C#-7	36376	142	24
114	D-7	38539	150	139
115	D#-7	40830	159	126
116	E-7	43258	168	250
117	F-7	45830	179	6
118	F#-7	48556	189	172
119	G-7	51443	200	243
120	G#-7	54502	212	230
121	A-7	57743	225	143
122	A#-7	61176	238	248
123	B-7	64814	253	46

FILTER SETTINGS

Location	Contents
54293	Low cutoff frequency (0-7)
54294	High cutoff frequency (0-255)
54295	Resonance (bits 4-7) Filter voice 3 (bit 2) Filter voice 2 (bit 1) Filter voice 1 (bit 0)
54296	High pass (bit 6) Bandpass (bit 5) Low pass (bit 4) Volume (bits 0-3)

APPENDIX N
BIBLIOGRAPHY

Addison-Wesley	"BASIC and the Personal Computer", Dwyer and Critchfield
Compute	"Compute's First Book of PET/CBM"
Cowbay Computing	"Feed Me, I'm Your PET Computer", Carol Alexander
	"Looking Good with Your PET", Carol Alexander
	"Teacher's PET — Plans, Quizzes, and Answers"
Creative Computing	"Getting Acquainted With Your VIC 20", T. Hartnell
Dilithium Press	"BASIC Basic-English Dictionary for the PET", Larry Noonan
	"PET BASIC", Tom Rugg and Phil Feldman
Faulk Baker Associates	"MOS Programming Manual", MOS Technology
Hayden Book Co.	"BASIC From the Ground Up", David E. Simon
	"I Speak BASIC to My PET", Aubrey Jones, Jr.
	"Library of PET Subroutines", Nick Hampshire
	"PET Graphics", Nick Hampshire
	"BASIC Conversions Handbook, Apple, TRS-80, and PET", David A. Brain, Phillip R. Oviatt, Paul J. Paquin, and Chandler P. Stone
Howard W. Sams	"The Howard W. Sams Crash Course in Microcomputers", Louis E. Frenzel, Jr.
	"Mostly BASIC: Applications for Your PET", Howard Berenbon
	"PET Interfacing", James M. Downey and Steven M. Rogers
	"VIC 20 Programmer's Reference Guide", A. Finkel, P. Higginbottom, N. Harris, and M. Tomczyk
Little, Brown & Co.	"Computer Games for Businesses, Schools, and Homes", J. Victor Nagigian, and William S. Hodges
	"The Computer Tutor: Learning Activities for Homes and Schools", Gary W. Orwig, University of Central Florida, and William S. Hodges
McGraw-Hill	"Hands-On BASIC With a PET", Herbert D. Peckman
	"Home and Office Use of VisiCalc", D. Castlewitz, and L. Chisauki
Osborne/McGraw-Hill	"PET/CBM Personal Computer Guide", Carroll S. Donahue
	"PET Fun and Games", R. Jeffries and G. Fisher
	"PET and the IEEE", A. Osborne and C. Donahue
	"Some Common BASIC Programs for the PET", L. Poole, M. Borchers, and C. Donahue
	"Osborne CP/M User Guide", Thom Hogan
	"CBM Professional Computer Guide"
	"The PET Personal Computer Guide"
	"The 8086 Book", Russell Rector and George Alexy
P. C. Publications	"Beginning Self-Teaching Computer Lessons"
Prentice-Hall	"The PET Personal Computer for Beginners", S. Dunn and V. Morgan
Reston Publishing Co.	"PET and the IEEE 488 Bus (GPIB)", Eugene Fisher and C. W. Jensen
	"PET BASIC — Training Your PET Computer", Ramon Zamora, Wm. F. Carrie, and B. Albrecht
	"PET Games and Recreation", M. Ogelsby, L. Lindsey, and D. Kunkin
	"PET BASIC", Richard Huskell
	"VIC Games and Recreation"
Telmas Courseware Ratings	"BASIC and the Personal Computer", T. A. Dwyer, and M. Critchfield
Total Information Services	"Understanding Your PET/CBM, Vol. 1, BASIC Programming"
	"Understanding Your VIC", David Schultz

Commodore Magazines provide you with the most up-to-date information for your Executive 64. Two of the most popular publications that you should seriously consider subscribing to are:

COMMODORE — *The Microcomputer Magazine* is published bi-monthly and is available by subscription ($15.00 per year, U.S., and $25.00 per year, worldwide).

POWER/PLAY — *The Home Computer Magazine* is published quarterly and is available by subscription ($10.00 per year, U.S., and $15.00 per year worldwide).

SPRITE REGISTER MAP

Register Dec	# Hex	D87	D86	D85	D84	D83	D82	D81	D80	
0	0	SOX7							S0XO	SPRITE O X Component
1	1	SOY7							S0YO	SPRITE O Y Component
2	2	S1X7							S1XO	SPRITE 1 X
3	3	S1Y7							S1YO	SPRITE 1 Y
4	4	S2X7							S2XO	SPRITE 2 X
5	5	S2Y7							S2YO	SPRITE 2 Y
6	6	S3X7							S3XO	SPRITE 3 X
7	7	S3Y7							S3YO	SPRITE 3 Y
8	8	S4X7							S4XO	SPRITE 4 X
9	9	S4Y7							S4YO	SPRITE 4 Y
10	A	S5X7							S5XO	SPRITE 5 X
11	B	S5Y7							S5YO	SPRITE 5 Y
12	C	S6X7							S6XO	SPRITE 6 X
13	D	S6Y7							S6YO	SPRITE 6 Y
14	E	S7X7							S7XO	SPRITE 7 X Component
15	F	S7Y7							S7YO	SPRITE 7 Y Component
16	10	S7X8	S6X8	S5X8	S4X8	S3X8	S2X8	S1X8	S0X8	MSB of X COORD.
17	11	RCB	ECM	BMM	BLNK	RSEL	YSCL2	YSCL1	YSCL0	Y SCROLL MODE
18	12	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0	RASTER
19	13	LPX7							LPX0	LIGHT PEN X
20	14	LPY7							LPY0	LIGHT PEN Y

Register Dec	# Hex	D87	D86	D85	D84	D83	D82	D81	D80	
21	15	SE7							SE0	SPRITE ENABLE (ON/OFF)
22	16	N.C.	N.C.	RST	MCM	CSEL	XSCL2	XSCL1	XSCL0	X SCROLL MODE
23	17	SEXY7							SEXY0	SPRITE EXPAND Y
24	18	VS13	VS12	VS11	VS10	CB13	CB12	CB11	N.C.	SCREEN Character Memory
25	19	IRQ	N.C.	N.C.	N.C.	LPIRO	ISSC	ISBC	RIRQ	Interrupt Request's
26	1A	N.C.	N.C	N.C.	N.C.	MLPI	MISSC	MISBC	MRIRO	Interrupt Request MASKS
27	1B	BSP7							BSP0	Background-Sprite PRIORITY
28	1C	SCM7							SCMO	Multicolor SPRITE SELECT
29	1D	SEXX7							SEXXO	SPRITE EXPAND X
30	1E	SSC7							SSCO	Sprite-Sprite COLLISION
31	1F	SBC7							SBCO	Sprite-Background COLLISION

Register # Dec	Hex	Color
32	20	BORDER COLOR
33	21	BACKGROUND COLOR 0
34	22	BACKGROUND COLOR 1
35	23	BACKGROUND COLOR 2
36	24	BACKGROUND COLOR 3
37	25	SPRITE MULTICOLOR 0
38	26	SPRITE MULTICOLOR 1

Register # Dec	Hex	Color
39	27	SPRITE 0 COLOR
40	28	SPRITE 1 COLOR
41	29	SPRITE 2 COLOR
42	2A	SPRITE 3 COLOR
43	2B	SPRITE 4 COLOR
44	2C	SPRITE 5 COLOR
45	2D	SPRITE 6 COLOR
46	2E	SPRITE 7 COLOR

COLOR CODES

Dec	Hex	Color
0	0	BLACK
1	1	WHITE
2	2	REO
3	3	CYAN
4	4	PURPLE
5	5	GREEN
6	6	BLUE
7	7	YELLOW

Dec	Hex	Color
8	8	ORANGE
9	9	BROWN
10	A	LT. REO
11	B	GRAY 1
12	C	GRAY 2
13	D	LT. GREEN
14	E	LT, BLUE
15	F	GRAY 3

LEGENO
ONLY COLORS 0-7 MAY BE USED IN MULTICOLOR CHARACTER MOOE

APPENDIX P
6566/6567 (VIC-II) CHIP REGISTER MAP

The 6566/6567 are multi-purpose color video controller devices for use In both computer video terminals and video game applicatlons. Both devices contaln 47 control registers which are accessed via a standard 8-bit microprocessor bus (65XX) and will access up to 16K of memory for display information. The various operating modes and options within each mode are described.

ADDRESS	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	DESCRIPTION
00 ($00)	M0X7	M0X6	M0X5	M0X4	M0X3	M0X2	M0X1	M0X0	MOB 0 X-position
01 ($01)	M0Y7	M0Y6	M0Y5	M0Y4	M0Y3	M0Y2	M0Y1	M0Y0	MOB 0 Y-position
02 ($02)	M1X7	M1X6	M1X5	M1X4	M1X3	M1X2	M1X1	M1X0	MOB I X-position
03 ($03)	M1Y7	M1Y6	M1Y5	M1Y4	M1Y3	M1Y2	M1Y1	M1Y0	MOB 1 Y-position
04 ($04)	M2X7	M2X6	M2X5	M2X4	M2X3	M2X2	M2X1	M2X0	MOB 2 X-position
05 ($05)	M2Y7	M2Y6	M2Y5	M2Y4	M2Y3	M2Y2	M2Y1	M2Y0	MOB 2 Y-position
06 ($06)	M3X7	M3X6	M3X5	M3X4	M3X3	M3X2	M3X1	M3X0	MOB 3 X-position
07 ($07)	M3Y7	M3Y6	M3Y5	M3Y4	M3Y3	M3Y2	M3Y1	M3Y0	MOB 3 Y-position
08 ($08)	M4X7	M4X6	M4X5	M4X4	M4X3	M4X2	M4X1	M4X0	MOB 4 X-position
09 ($09)	M4Y7	M4Y6	M4Y5	M4Y4	M4Y3	M4Y2	M4Y1	M4Y0	MOB 4 Y-position
10 ($0A)	M5X7	M5X6	M5X5	M5X4	M5X3	M5X2	M5X1	M5XD	MOB 5 X-position
11 ($0B)	M5Y7	M5Y6	M5Y5	M5Y4	M5Y3	M5Y2	M5Y1	M5Y0	MOB 5 Y-position
12 ($0C)	M6X7	M6X6	M6X5	M6X4	M6X3	M6X2	M6X1	M6X0	MOB 6 X-position
13 ($0D)	M6Y7	M6Y6	M6Y5	M6Y4	M6Y3	M6Y2	M6Y1	M6Y0	MOB 6 Y-position
14 ($0E)	M7X7	M7X6	M7X5	M7X4	M7X3	M7X2	M7X1	M7X0	MOB 7 X-position
15 ($0F)	M7Y7	M7Y6	M7Y5	M7Y4	M7Y3	M7Y2	M7Y1	M6Y0	MOB 7 Y-position
16 ($10)	M7XB	M6XB	M5XB	M4XB	M3X8	M2X8	M1X8	M0XB	MSB of X-position
17 ($11)	RC8	ECM	BMM	DEN	RSEL	Y2	Y1	Y0	, See text
18 ($12)	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0	Raster register
19 ($13)	LPX8	LPX7	LPX6	LPX5	LPX4	LPX3	LPX2	LPX1	Light Pen X
20 ($14)	LPY7	LPY6	LPY5	LPY4	LPY3	LPY2	LPY1	LPY0	Light Pen Y
21 ($15)	M7E	M6E	M5E	M4E	M3E	M2E	M1E	M0E	MOB Enable
22 ($16)	—	—	RES	MCM	CSEL	X2	X1	X0	See text
23 ($17)	M7YE	M6YE	M5YE	M4YE	M3YE	M2YE	M1YE	M0YE	MOB Y-expand
24 ($1B)	VM13	VM12	VM11	VM10	CB13	CB12	CB11	—	Memory Pointers
25 ($19)	IRQ	—	—	—	ILP	IMMC	IMBC	IRST	Interrupt Register
26 ($1A)	—	—	—	—	ELP	EMMC	EMBC	ERST	Enable Interrupt
27 ($1B)	M7DP	M6DP	M5DP	M4DP	M3DP	M2DP	M1DP	M0DP	MOB-DATA Priority
28 ($1C)	M7MC	M6MC	M5MC	M4MC	M3MC	M2MC	M1MC	M0MC	MOB Multicolor Sel
29 ($1D)	M7XE	M6XE	M5XE	M4XE	M3XE	M2XE	M1XE	M0XE	MOB X-expand
30 ($1E)	M7M	M6M	M5M	M4M	M3M	M2M	M1M	M0M	MOB-MOB Collision
31 ($1F)	M7D	M6D	M5D	M4D	M3D	M2D	M1D	M0D	MOB-DATA Collision
32 ($20)	—	—	—	—	EC3	EC2	EC1	EC0	Exterior Color
33 ($21)	—	—	—	—	B0C3	B0C2	B0C1	B0C0	Bkgd #0 Color
34 ($22)	—	—	—	—	B1C3	B1C2	B1C1	B1C0	Bkgd #1 Color
35 ($23)	—	—	—	—	B2C3	B2C2	B2C1	B2C0	Bkgd #2 Color
36 ($24)	—	—	—	—	B3C3	B3C2	B3C1	B3C0	Bkgd #3 Color
37 ($25)	—	—	—	—	MM03	MM02	MM01	MM00	MOB Multicolor #0
38 ($26)	—	—	—	—	MM13	MM12	MM11	MM10	MOB Multicolor #1
39 ($27)	—	—	—	—	M0C3	M0C2	M0C1	M0C0	MOB 0 Color
40 ($28)	—	—	—	—	M1C3	M1C2	M1C1	M1C0	MOB 1 Color
41 ($29)	—	—	—	—	M2C3	M2C2	M2C1	M2C0	MOB 2 Color
42 ($2A)	—	—	—	—	M3C3	M3C2	M3C1	M3C0	MOB 3 Color
43 ($2B)	—	—	—	—	M4C3	M4C2	M4C1	M4C0	MOB 4 Color
44 ($2C)	—	—	—	—	M5C3	M5C2	M5C1	M5C0	MOB 5 Color
45 ($2D)	—	—	—	—	M6C3	M6C2	M6C1	M6C0	MOB 6 Color
46 ($2E)	—	—	—	—	M7C3	M7C2	M7C1	M7C0	MOB 7 Color

NOTE A dash indicates a no connect. All no connects are read as a "1".

APPENDIX Q
EXECUTIVE 64
SOUND CONTROL SETTINGS

This handy table gives you the key numbers you need to use in your sound programs, according to which of the 64's 3 voices you want to use. To set or adjust a sound control in your BASIC program, just POKE the number from the second column, followed by a comma (,) and a number from the chart . . . like this: POKE 54276,17 (Selects a Triangle Waveform for VOICE 1).

Remember that you must set the VOLUME before you can generate sound. POKE54296 followed by a number up to 15 to set the volume for all 3 voices.

It takes 2 separate POKEs to generate each musical note. For example POKE54273,33:POKE54272,135 designates low C in the sample scale below.

Also, you aren't limited to the numbers shown in the tables. If 33 doesn't sound "right" for a low C, try 34. To provide a higher SUSTAIN or ATTACK rate than those shown, add two or more SUSTAIN numbers together. (Examples: POKE54277,96 combines two attack rates (32 and 64) for a combined higher attack rate, POKE54277,20 provides a low attack rate (16) and a medium decay rate (4).

SETTING VOLUME — SAME FOR ALL 3 VOICES													
VOLUME CONTROL	POKE54296	Settings range from 0 (off) to 15 (loudest)											

TO CONTROL THIS SETTING	POKE THIS NUMBER:	FOLLOWED BY ONE OF THESE NUMBERS (0 to 15 or 0 to 255 depending on range)											
TO PLAY A NOTE		C	C#	D	D#	E	F	F#	G	G#	A	A#	B
HIGH FREQUENCY	54273	33	35	37	39	42	44	44	50	53	56	59	63
LOW FREQUENCY	54272	135	134	162	223	62	193	107	60	57	99	190	75

WAVEFORM	POKE	TRIANGLE		SAWTOOTH		PULSE		NOISE	
	54276	17		33		65		129	

PULSE RATE (Pulse Waveform)									
HI PULSE	54275	A value of 0 to 15 (for Pulse waveform only)							
LO PULSE	54274	A value of 0 to 255 (for Pulse waveform only)							

ATTACK/DECAY	POKE	ATK4	ATK3	ATK2	ATK1	DEC4	DEC3	DEC2	DEC1
	54277	128	64	32	16	8	4	2	1

SUSTAIN/RELEASE	POKE	SUS4	SUS3	SUS2	SUS1	REL4	REL3	REL2	REL1
	54278	128	64	32	16	8	4	2	1

TO PLAY A NOTE		C	C#	D	D#	E	F	F#	G	G#	A	A#	B
HIGH FREQUENCY	54280	33	35	37	39	42	44	47	50	53	56	59	63
LOW FREQUENCY	54279	135	134	162	223	62	193	107	60	57	99	190	75

WAVEFORM	POKE	TRIANGLE		SAWTOOTH		PULSE		NOISE	
	54283	17		33		65		129	

PULSE RATE (Pulse Waveform)									
HI PULSE	54282	A value of 0 to 15 (for Pulse waveform only)							
LO PULSE	54281	A value of 0 to 255 (for Pulse waveform only)							

ATTACK/DECAY	POKE	ATK4	ATK3	ATK2	ATK1	DEC4	DEC3	DEC2	DEC1
	54284	128	64	32	16	8	4	2	1

SUSTAIN/RELEASE	POKE	SUS4	SUS3	SUS2	SUS1	REL4	REL3	REL2	REL1
	54285	128	64	32	16	8	4	2	1

TO PLAY A NOTE		C	C#	D	D#	E	F	F#	G	G#	A	A#	B
HIGH FREQUENCY	54287	33	35	37	39	42	44	44	50	53	56	59	63
LOW FREQUENCY	54286	135	134	162	223	62	193	107	60	57	99	190	75

WAVEFORM	POKE	TRIANGLE		SAWTOOTH		PULSE		NOISE	
	54290	17		33		65		129	

PULSE RATE (Pulse Waveform)									
HI PULSE	54289	A value of 0 to 15 (for Pulse waveform only)							
LO PULSE	54288	A value of 0 to 255 (for Pulse waveform only)							

ATTACK/DECAY	POKE	ATK4	ATK3	ATK2	ATK1	DEC4	DEC3	DEC2	DEC1
	54291	128	64	32	16	8	4	2	1

SUSTAIN/RELEASE	POKE	SUS4	SUS3	SUS2	SUS1	REL4	REL3	REL2	REL1
	54292	128	64	32	16	8	4	2	1

TRY THESE SETTINGS TO SIMULATE DIFFERENT INSTRUMENTS

Instrument	Waveform	Attack/Decay	Sustain/Release	Pulse Rate
Piano	Pulse	9	0	HI-0, Lo-255
Flute	Triangle	96	0	Not applicable
Harpsichord	Sawtooth	9	0	Not applicable
Xylophone	Triangle	9	0	Not applicable
Organ	Triangle	0	240	Not applicable
Collape	Triangle	0	240	Not applicable
Accordian	Triangle	102	0	Not applicable
Trumpet	Sawtooth	96	0	Not applicable

MEANINGS OF SOUND TERMS

ADSR — Attack/Decay/Sustain/Release

Attack — rate sound rises to peek volume

Decay — rate sound falls from peek volume to sustain level

Sustain — prolong rate at certain volume level

Release — rate at which volume falls from sustain level

Waveform — "shape" of sound wave

Pulse — tone quality of Pulse Waveform

NOTE: Attack/Decay and Sustain/Release settings should always be POKEd in your program BEFORE the Waveform is POKEd.

APPENDIX R
6581 SOUND INTERFACE DEVICE (SID) CHIP SPECIFICATIONS

CONCEPT

The 6581 Sound Interface Device (SID) is a single-chip, 3-voice electronic music synthesizer/sound effects generator compatible with the 65XX and similar microprocessor families. SID provides wide-range high-resolution control of pitch (frequency), tone color (harmonic content), and dynamics (volume). Specialized control circuitry minimizes software overhead, facilitating use in arcade/home video games and low-cost musical instruments.

FEATURES

- 3 TONE OSCILLATORS
    Range: 0-4 kHz
- 4 WAVEFORMS PER OSCILLATOR
    Triangle, Sawtooth,
    Variable Pulse, Noise
- 3 AMPLITUDE MODULATORS
    Range: 48 dB
- 3 ENVELOPE GENERATORS
    Exponential response
    Attack Rate: 2 ms-8 s
    Decay Rate: 6 ms-24 s
    Sustain Level: 0-peak volume
    Release Rate: 6 ms-24 s
- OSCILLATOR SYNCHRONIZATION
- RING MODULATION

DESCRIPTION

The 6581 consists of three synthesizer "voices" which can be used independently or in conjunction with each other (or external audio sources) to create complex sounds. Each voice consists of a Tone Oscillator/Waveform Generator, an Envelope Generator and an Amplitude Modulator. The Tone Oscillator controls the pitch of the voice over a wide range. The Oscillator produces four waveforms at the selected frequency, with the unique harmonic content of each waveform providing simple control of tone color. The volume dynamics of the oscillator are controlled by the Amplitude Modulator under the direction of the Envelope Generator. When triggered, the Envelope Generator creates an amplitude envelope with programmable rates of increasing and decreasing volume. In addition to the three voices, a programmable Filter is provided for generating complex, dynamic tone colors via subtractive syntheses.

SID allows the microprocessor to read the changing output of the third Oscillator and third Envelope Generator. These outputs can be used as a source of modulation information for creating vibrato, frequency/filter sweeps and similar effects. The third oscillator can also act as a random number generator for games. Two A/D converters are provided for interfacing SID with potentiometers. These can be used for "paddles" in a game environment or as front panel controls in a music synthesizer. SID can process external audio signals, allowing multiple SID chips to be daisy-chained or mixed in complex polyphonic systems.

SID CONTROL REGISTERS

There are 29 eight-bit registers in SID which control the generation of sound. These registers are either WRITE-only or READ-only and are listed below in Table 1.

Table 1. SID Register Map

162

The following BASIC commands and statements let you perform a variety of operations on disk drives and any compatible Commodore printer.

CLOSE

TYPE: I/O Statement
FORMAT: CLOSE < file number >

Action: This statement shuts off any data file or channel to a device. The file number is the same as when the file or device was OPENed (see OPEN statement and the section on INPUT/OUTPUT programming).

When working with storage devices like disks, the CLOSE operation stores any incomplete buffers to the device. When this is not performed, the file will be unreadable on the disk. The CLOSE operation isn't as necessary with other devices, but it does free up memory for other files. See your external device manual for more details.

EXAMPLES of CLOSE Statement:

```
10 CLOSE 1
20 CLOSE X
30 CLOSE 9 * (1 + J)
```

CMD

TYPE: I/O Statement
FORMAT: CMD < file number > [,string]

Action: This statement switches the primary output device from the TV screen to the file specified. This file could be on disk, printer, or an I/O device like the modem. The file number must be specified in a prior OPEN statement. The string, when specified, is sent to the file. This is handy for titling printouts, etc.

When this command is in effect, any PRINT statements and LIST commands will not display on the screen, but will send the text in the same format to the file.

To re-direct the output back to the screen, the PRINT# command should send a blank line to the CMD device before CLOSEing, so it will stop expecting data (called "un-listening" the device).

Any system error (like **?SYNTAX ERROR**) will cause output to return to the screen. Devices aren't un-listened by this, so you should send a blank like after an error condition. (See your printer or disk manual for more details.)

163

EXAMPLES of CMD Statement:

OPEN 4, 4: CMD 4, "TITLE" : LIST: REM LISTS PROGRAM ON
   PRINTER
PRINT# 4: CLOSE 4: REM UN-LISTENS AND CLOSES PRINTER

10 OPEN 1, 8, 4, "TEST": REM CREATE SEO FILE
20 CMD 8: REM OUTPUT TO TAPE FILE, NOT SCREEN
30 FOR L = 1 TO 100
40 PRINT L: REM PUTS NUMBER IN DISK BUFFER
50 NEXT
60 PRINT# 1: REM UNLISTEN
70 CLOSE 1: REM WRITE UNFINISHED BUFFER, PROPERLY
   FINISH

GET#

TYPE: I/O Statement
FORMAT: GET# ⟨ file number⟩ , ⟨variable list⟩

Action: This statement reads characters one-at-a-time from the device or file specified. It works the same as the GET statement, except that the data comes from a different place than the keyboard. If no character is received, the variable is set to an empty string (equal to "") or to 0 for numeric variables. Characters used to separate data in files, like the comma (,) or ▓▓▓▓▓ key code (ASC code of 13), are received like any other character.

When used with device #3 (TV screen), this statement will read characters one by one from the screen. Each use of GET# moves the cursor 1 position to the right. The character at the end of the logical line is changed to a CHR$ (13), the ▓▓▓▓▓ key code.

EXAMPLES of GET# Statement

5   GET# 1, A$
10 OPEN 1, 3: GET# 1, Z7$
20 GET# 1, A, B, C$, D$

INPUT#

TYPE: I/O Statement
FORMAT INPUT# ⟨ file number⟩ , ⟨variable list⟩

Action: This is usually the fastest and easiest way to retrieve data stored in a file on disk. The data is in the form of whole variables of up to 80 characters in length, as opposed to the one-at-a-time method of GET#. First, the file must have been OPENed, then INPUT# can fill the variables.

The INPUT# command assumes a variable is finished when it reads a RETURN code (CHR$(13)), a comma (,), semicolon (;), or colon(:). Quote marks can be used to enclose these characters when writing if they are needed (See PRINT# statement).

If the variable type used is numeric, and non-numeric characters are received, a **BAD DATA** error results. INPUT# can read strings up to 80 characters long, beyond which a **STRING TOO LONG** error results.

When used with device #3 (the screen), this statement will read an entire logical line and move the cursor down to the next line.

EXAMPLES of INPUT# Statement:

10 INPUT# 1, A
20 INPUT# 2, A$, B$

LOAD

TYPE: Command
FORMAT LOAD " ⟨ file-name⟩ " , ⟨ device ⟩ [, ⟨ address ⟩]

Action: The LOAD statement reads the contents of a program file from disk into memory. That way you can use the information LOADed or change the information in some way. The disk unit is normally device number 8. The LOAD closes all open files and, if it is used in direct mode, it performs a CLR (clear) before reading the program. If LOAD is executed from within a program, the program is RUN. This means that you can use LOAD to "chain" several programs together. None of the variables are cleared during a chain operation.

If you are using file-name pattern matching, the first file which matches the pattern is loaded. The asterisk in quotes by itself ("*") causes the first file-name in the disk directory to be loaded. If the file-name used does not exist or if it is not a program file, the BASIC error message ?**FILE NOT FOUND** occurs.

If you use the secondary address of 1 this will cause the program to LOAD to the memory location from which it was saved.

EXAMPLES of LOAD Command:

LOAD A$,8	(Uses the name in A$ to search)
LOAD "*",8	(LOADs first program from disk)
LOAD "$",8:LIST	(LOADs and LISTs disk directory)
LOAD "FUN",8 SEARCHING FOR FUN LOADING READY.	(LOAD a file from disk)
LOAD "GAME ONE",8,1 SEARCHING FOR GAME ONE LOADING READY.	(LOAD a file to the specific memory location from which the program was saved on the disk)

OPEN

TYPE: I/O Statement
FORMAT: OPEN ⟨file-num⟩ , ⟨device⟩ [,⟨ address⟩] [," ⟨file-name⟩ [, ⟨type⟩] [, ⟨mode⟩]"]

Action: This statement OPENs a channel for input and/or output to a peripheral device. However, you may NOT need all those parts for every OPEN statement. Some OPEN statements require only 2 codes:

1) LOGICAL FILE NUMBER
2) DEVICE NUMBER

The ⟨ file-num⟩ is the logical file number, which relates the OPEN, CLOSE, CMD, GET#, INPUT#, and PRINT# statements to each other and associates them with the file-name and the piece of equipment being used. The logical file number can range from 1 to 255 and you can assign it any number you want in that range.

NOTE: File numbers over 128 were really designed for other uses so it's good practice to use only numbers below 127 for file numbers.

Each peripheral device (printer, disk drive) in the system has its own number which it answers to. The ⟨device⟩ number is used with OPEN to specify on which device the data file exists. Peripherals like disk drives or printers also answer to several secondary addresses. Think of these as codes which tell each device what operation to perform. The device logical file number is used with every GET#, INPUT#, and PRINT#.

The file-name can also be left out, but later on in your program you can NOT call the file by name if you have not already given it one.

For disk files, the secondary addresses 2 thru 14 are available for data-files, but other numbers have special meanings in DOS commands. You must use a secondary address when using your disk drive(s). (See your disk drive manual for DOS command details.)

The ⟨ file-name ⟩ is a string of 1-16 characters and is optional for printer files. If the file ⟨ type ⟩ is left out the type of file will automatically default to the *Program* file unless the mode is given. *Sequential* files are OPENed for reading ⟨ mode ⟩ = R unless you specify that files should be OPENed for writing ⟨mode⟩ = W is specified. A file ⟨ type ⟩ can be used to OPEN an *existing* Relative file. Use REL for ⟨ type ⟩ with Relative files. Relative and Sequential files are for disk only.

If you try to access a file before it is OPENed the BASIC error message ?FILE **NOT OPEN** will occur. If you try to OPEN a file for reading which does not exist the BASIC error message ?FILE **NOT FOUND** will occur. If a file is OPENed to disk for writing and the file-name already exists, the DOS error message **FILE EXISTS** occurs. If a file is OPENed that is already OPEN, the BASIC error message **FILE OPEN** occurs. (See Printer Manual for further details.)

EXAMPLES of OPEN Statements:

10 OPEN 2, 8, 4 "DISK-OUTPUT, SEQ,W"	(Opens sequential file on disk) — For Write
10 OPEN 50, 0	(Keyboard input)
10 OPEN 12, 3	(Screen output)
10 OPEN 130, 4	(Printer output)
10 OPEN 1,2,0, CHR$(10)	(Open channel to RS-232 device)
10 OPEN 1,4,0, "STRING"	(Send upper case/graphics to the printer)
10 OPEN 1,4,7, "STRING"	(Send upper/lower case to printer)
10 OPEN 1,5,7, "STRING"	(Send upper/lower case to printer with device #5)
10 OPEN 1,8,15, "COMMAND"	(Send a command to disk)
10 OPEN 1,8,1, "NAME,L" + CHR$(X)	(Relative file OPEN (1st time) where X is the length of the relative record)
10 OPEN, 1,8,1,"NAME"	(Relative or sequential read)

PRINT#

TYPE: I/O Statement
FORMAT: PRINT# ⟨ file-number ⟩ [⟨ variable ⟩] [⟨,/;⟩ ⟨variable⟩]...

Actions: The PRINT# statement is used to write data items to a logical file. It must use the same number used to OPEN the file. Output goes to the device-number used in the OPEN statement. The ⟨ variable ⟩ expressions in the output-list can be of any type. The punctuation characters between items are the same as with the PRINT statement and they can be used in the same way. The effects of punctuation are different in two significant respects.

If no punctuation finishes the list, a carriage-return and a line-feed are written at the end of the data. If a comma or semicolon terminates the output-list, the carriage-return and line-feed are suppressed. Regardless of the punctuation, the next PRINT# statement begins output in the next available character position. The line-feed will act as a stop when using the INPUT# statement, leaving an empty variable when the next INPUT# is executed. The line-feed can be suppressed of compensated for as shown in the examples below.

The easiest way to write more than one variable to a file on disk is to set a string variable to CHR$(13), and use that string in between all the other variables when writing the file.

EXAMPLES of PRINT# Statement

1)
```
10 DPEN 1,8,4, "MY FILE"
20 R$ = CHR$(13)                    (By Changing the CHR$(13) to
30 PRINT# 1,1;R$;2;R$;3;R$;4;R$;5   CHR$(44) you put a "," between each
40 PRINT# 1,6                       varible. CHR$(59) would put a ";"
50 PRINT# 1,7                       between each variable.)
```

2)
```
10 CD$ = CHR$(44): CR$ = CHR$(13)
20 PRINT# 1, "AAA"CO$"BBB",     AAA,BBB   CCCDDDEEE
   "CCC";"DDD";"EEE"CR$         (carriage return)
   "FFF"CR$;                    FFF(carriage return)
30 INPUT#1, A$,BCDE$,F$
```

3)
```
5   CR$ = CHR$(13)
10 PRINT#2,"AAA";CR$;"BBB"      (10 blanks)AAA
20 PRINT#2, "CCC";             BBB
                               (10 blanks) CCC
30 INPUT#2, A$,B$,DUMMY$,C$
```

SAVE

TYPE: Command
FDRMAT: SAVE " ⟨file-name⟩ " , ⟨device-number⟩ [, ⟨address⟩]

Action: The SAVE command is used to store the program that is currently in memory onto a diskette file. The program being SAVEd is only affected by the command while the SAVE is happening. The program remains in the current computer memory even after the SAVE operation is completed until you put something else there by using another command. The file type will be "prg" (program). The SAVE statement can be used in your programs and execution will continue with the next statement after the SAVE is completed.

When saving programs onto a disk, the ⟨file-name⟩ must be present.

EXAMPLES of SAVE Command:

```
SAVE "FUN DISK",8       (SAVES on disk (device 8 is the disk))

SAVE A$,8               (Store on disk with the name A$)
```

VERIFY

TYPE: Command
FDRMAT: VERIFY " ⟨file-name⟩ " , ⟨device⟩

Action: The VERIFY command is used, in direct or program mode, to compare the contents of a BASIC program file on disk with the program currently in memory. VERIFY is normally used right after a SAVE, to make sure that the program was stored correctly on tape or disk.

For disk files (device number 8), the file-name must be present. If any difference in program text are found, the BASIC error message ?VERIFY ERRDR is displayed.

A program name can be given either in quotes ("") or as a string variable.

EXAMPLE ot VERIFY Command:

```
9000 SAVE "ME",8
9010 VERIFY "ME",8        (Looks at device 8 for the program)
```

INDEX

MEMO

MEMO

SIMPLE VARIABLES

Type	Name	Range
Real	XY	±1.70141183E+38
		±2.938735881E-39
Integer	XY%	±32767
String	XY$	0 to 255 characters

X is a letter (A-Z), Y is a letter or number (0-9). Variable name can be more than 2 characters, but only the first two are recognized.

ARRAY VARIABLES

Type	Name
Single Dimension	XY(5)
Two-Dimension	XY(5,5)
Three-Dimension	XY(5,5,5)

Arrays of up to eleven elements (subscripts 0-10) can be used where needed. Arrays with more than eleven elements need to be DIMensioned.

ALGEBRAIC OPERATORS

- = Assign value to variable
- − Negation
- ↑ Exponentiation
- * Multiplication
- / Division
- + Addition
- − Subtraction

RELATIONAL AND LOGICAL OPERATORS

- = Equal to
- <> Not Equal to
- < Less than
- > Greater than
- <= Less than or Equal to
- >= Greater than or Equal to
- NOT Logical "Not"
- AND Logical "And"
- OR Logical "Or"

Expression equals 1 if true, 0 if false

SYSTEM COMMANDS

Command	Description
LOAD "NAME",8	Loads a program from disk
SAVE "NAME",8	Saves a program to disk
VERIFY "NAME",8	Verifies that program was SAVEd without errors
RUN	Executes a program
RUN xxx	Executes program starting at line xxx
STOP	Halts execution
END	Ends execution
CONT	Continues program execution from line where program was halted
PEEK(X)	Returns contents of memory location X
POKE X,Y	Changes contents of location X to value Y
SYS xxxxx	Jumps to execute a machine language program, starting at xxxxx
WAIT X,Y,Z	Program waits until contents of location X, when ORed with Z and ANDed with Y, is nonzero
USR(X)	Passes value of X to a machine language subroutine

EDITING AND FORMATTING COMMANDS

Command	Description
LIST	Lists entire program
LIST A-B	Lists from line A to line B
REM Message	Comment message can be listed but is ignored during program execution
TAB(X)	Used in PRINT statements. Spaces X positions on screen
SPC(X)	PRINTs X blanks on line

Command	Description
POS(X)	Return current cursor position
CLR/HOME	Positions cursor to left corner of screen
SHIFT CLR/HOME	Clears screen and places cursor in "Home" position
SHIFT INST/DEL	Inserts space at current cursor position
INST/DEL	Deletes character at current cursor position
CTRL	When used with numeric color key, selects that color. May be used in PRINT statement
CRSR Keys	Moves cursor up, down, left, right on screen
Commodore key	When used with SHIFT selects between upper/lower case and graphic display mode. When used with numeric color key selects optional text color

ARRAYS AND STRINGS

Command	Description
DIM A(X,Y,Z)	Sets maximum subscripts for A; reserves space for (X+1)*(Y+1)*(Z+1) elements starting at A(0,0,0)
LEN(X$)	Returns number of characters in X$
STR$(X)	Returns numeric value of X converted to a string
VAL(X$)	Returns numeric value of A$ up to first nonnumeric character
CHR$(X)	Returns ASCII character whose code is X
ASC(X$)	Returns ASCII code for first character of X$
LEFT$(A$,X)	Returns leftmost X characters of A$
RIGHT$(A$,X)	Returns rightmost X characters of A$
MID$(A$,X,Y)	Returns Y characters of A$ starting at character X

INPUT/OUTPUT COMMANDS

Command	Description
INPUT A$ OR A	PRINTs ? on screen and waits for user to enter a string or value
INPUT ABC A	PRINTs message and waits for user to enter value. Can also INPUT A$
GET A$ or A	Waits for user to type one-character value, no RETURN needed
DATA A,B,C	Initializes a set of values that can be used by READ statement
READ A$ or A	Assigns next DATA value to A$ or A
RESTORE	Resets data pointer to start READing the DATA list again
PRINT A= A	PRINTs string A= and value of A. Semicolon spaces, comma tabs data to next field

PROGRAM FLOW

Command	Description
GOTO x	Branches to line x
IF A=3 THEN 10	If assertion is true THEN execute following part of statement. If false execute next line number
FOR A=1 TO 10 STEP 2 ... NEXT	Executes all statements between FOR and corresponding NEXT with A going from 1 to 10 by 2. Step size is 1 unless specified
NEXT A	Defines end of loop. A is optional
GOSUB 2000	Branches to subroutine starting at line 2000
RETURN	Marks end of subroutine. Returns to statement following most recent GOSUB
ON x GOTO A,B	Branches to Xth line number on list. If X = 1 branches to A, etc.
ON X GOSUB A,B	Branches to subroutine at Xth line number in list

COMMODORE SALES CENTERS

Commodore Business Mochines, Inc.
1200 Wilson Drive
Westchester, PA 19380, U.S.A.

Commodore Business Mochines Ltd.
3370 Pharmacy Avenue, Agincourt,
Ontario, MIW 2K4, Canada

Commodore Business Mochines (UK) Ltd.
675 Ajax Avenue, Trading Estate,
Slough Berks, SL1 4BG, England

Commodore Bueromoshinen GmbH
Lyonerstrasse 38, PO BOX 710126
Frankfult 6000, West Germany

Commodore Itoliono S.R.L.
Via Conservatoria 22
Milano 20122, Italy

Commodore Informotion Center
5 Onion Road
Lane Cove 2066 NSW, Australia

Commodore Computer B.V.
Marksingel 2e4811 N.V. Breda
Postlrus 720, 480 3oS Breda, Netherlands

Commodore AG(Schweiz)
Aeschenvorstadt 57
4010 Basel, Switzerland