

# COMMODORE 64C™

PERSONAL COMPUTER  
**system guide**  
*Learning to program in BASIC 2.0*







---

# COMMODORE 64C SYSTEM GUIDE

Learning to Program in BASIC 2.0

---



## USER'S MANUAL STATEMENT

### WARNING:

This equipment has been certified to comply with the limits for a Class B computing device, pursuant to subpart J of Part 15 of the Federal Communications Commission's rules, which are designed to provide reasonable protection against radio and television interference in a residential installation. If not installed properly, in strict accordance with the manufacturer's instructions, it may cause such interference. If you suspect interference, you can test this equipment by turning it off and on. If this equipment does cause interference, correct it by doing any of the following:

- Reorient the receiving antenna or AC plug.
- Change the relative positions of the computer and the receiver.
- Plug the computer into a different outlet so the computer and receiver are on different circuits.

**CAUTION:** Only peripherals with shield-grounded cables (computer input-output devices, terminals, printers, etc.), certified to comply with Class B limits, can be attached to this computer. Operation with non-certified peripherals is likely to result in communications interference.

Your house AC wall receptacle must be a three-pronged type (AC ground). If not, contact an electrician to install the proper receptacle. If a multi-connector box is used to connect the computer and peripherals to AC, the ground must be common to all units.

If necessary, consult your Commodore dealer or an experienced radio-television technician for additional suggestions. You may find the following FCC booklet helpful: "How to Identify and Resolve Radio-TV Interference Problems." The booklet is available from the U.S. Government Printing Office, Washington, D.C. 20402, stock no. 004-000-00345-4.

First Printing, April 1986  
Copyright © 1986 by Commodore Electronics Limited  
All rights reserved

This manual contains copyrighted and proprietary information. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of Commodore Electronics Limited.

**Commodore 64C** is a trademark of Commodore Electronics Limited.

**Commodore** and **Commodore 64** are registered trademarks of Commodore Electronics Limited.

**Commodore BASIC 7.0**

Copyright © 1986 by Commodore Electronics Limited  
All rights reserved

Copyright © 1977 by Microsoft Corp.  
All rights reserved



**TABLE OF  
CONTENTS**

<b>Chapter 1—Introduction</b>	<b>1</b>
<b>Chapter 2—Getting Started in BASIC</b>	<b>9</b>
<b>Chapter 3—Advanced BASIC Programming</b>	<b>39</b>
<b>Chapter 4—Graphics, Color and Sprites</b>	<b>63</b>
<b>Chapter 5—Sound and Music</b>	<b>95</b>
<b>Chapter 6—BASIC 2.0 Encyclopedia</b>	<b>107</b>
<b>Appendices</b>	<b>149</b>
A. BASIC 2.0 Error Messages	151
B. Connectors/Ports for Peripheral Equipment	155
C. Screen Display Codes	161
D. ASCII and CHR\$ Codes	163
E. Screen and Color Memory Maps	167
F. Derived Trigonometric Functions	169
G. Memory Map	171
H. BASIC 2.0 Abbreviations	173
I. Sprite Register Map	175
J. Sound and Music	177
<b>Glossary</b>	<b>181</b>
<b>Index</b>	<b>195</b>



CHAPTER

# Introduction

1

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

**CHAPTER 1**  
**Introduction**

**HOW TO USE THIS GUIDE**

**5**

**OVERVIEW OF THE COMMODORE 64C PERSONAL  
COMPUTER**

**6**



## How to Use This Guide

This Guide is designed to help you make full use of the advanced capabilities of the **Commodore 64C™** personal computer. Here's how to use the Guide:

1. Before you read any further in this book, make sure you've read the *COMMODORE 64C QUICK-CONNECT GUIDE*, which tells how to unpack and set up your new 64C computer and peripheral equipment. Also be sure to read the *COMMODORE 64C INTRODUCTORY GUIDE*, which contains important information on getting started with the Commodore 64C, including how to load and run prepackaged disk, cartridge and tape software. Both pieces of documentation come packed in the computer carton.
2. If you are interested mostly in learning the BASIC® language to create and run your own programs, you should first read Chapters 2 and 3 of this book. Chapter 2 gets you started quickly by introducing you to BASIC 2.0 concepts and providing numerous explanations and examples of commonly used commands and elementary programming techniques. Chapter 3 defines a number of more advanced BASIC commands and programming techniques, again giving explanations and examples of how to use them. Together, these two chapters provide a solid foundation from which you can move on to more specialized programming activities, such as graphics and sound.
3. If you want to learn how to add graphics and animation to your BASIC programs, read Chapter 4. This chapter tells how to program the 64C's powerful and varied graphics capabilities, which include eight sprites, 16 colors and a variety of animation techniques.
4. If you are interested in programming sound and music on the 64C, read Chapter 5, which describes the extensive sound and music features provided by the SID (Sound Interface Device). The SID is the 64C's versatile three-voice, six-octave sound synthesizer.
5. For more information on any facet of BASIC 2.0, read Chapter 6, *BASIC 2.0 ENCYCLOPEDIA*. This chapter defines all the elements of the BASIC 2.0 language and includes specific format and usage information on all BASIC 2.0 commands, statements and functions.
6. If, after reading Chapters 2 through 6, you are looking for additional technical information about a particular Commodore 64C topic, first check the APPENDICES to this book. These appendices contain a wide range of information, such as a complete list of BASIC error messages, ASCII and CHR\$ codes, screen and color memory maps, etc. See the GLOSSARY following the Appendices for definitions of common computer terms.

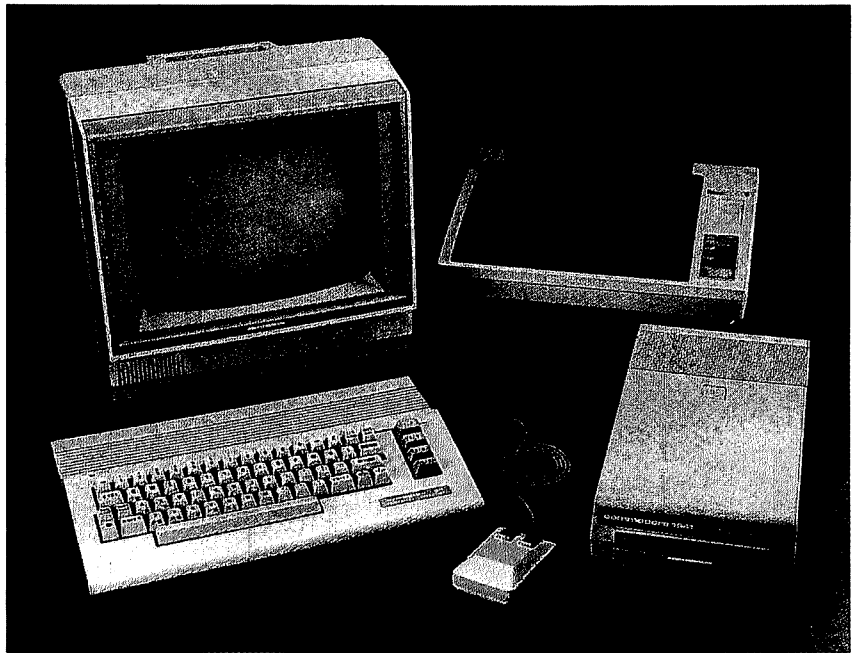


## Overview of the Commodore 64C Personal Computer

The features of the Commodore 64C are so many and so varied that—even in a book of this length—they can only be introduced. For additional information on all the technical features of the Commodore 64C, see the *COMMODORE 64® PROGRAMMER'S REFERENCE GUIDE*, available from your Commodore dealer or at most bookstores. For additional information on BASIC, see the complete three-part course on BASIC programming—*INTRODUCTION TO BASIC, PARTS I, II, and III*. This series of three books is also available from your Commodore dealer.

Key Commodore 64C features include:

- Fully compatible with Commodore 64 hardware, software and peripherals
- Versatile Commodore BASIC 2.0 programming language, offering more than 70 commands and functions
- 64K of RAM (Random Access Memory)
- 40 column screen output
- Ability to run thousands of off-the-shelf software programs for business, industry, science, education and home, including word processors, spreadsheets, databases, financial market software, telecommunications programs, etc.
- Ability to handle software packaged in disk, tape or cartridge formats
- Ability to work with a wide variety of peripheral devices, including video monitors, printers, modems, controllers (joysticks, mouse, etc.)
- Sophisticated graphics capabilities, including 8 individually programmable sprites and several animation modes
- Sixteen colors
- A professional style low-profile keyboard
- Ability to incorporate 6502 machine language data in BASIC programs
- Eight user-programmable function keys
- A three-voice, six-octave, synthesizer for sound and music



These Commodore 64C features can be translated into wide-ranging capabilities. The advanced software included with your new 64C incorporates icons, pulldown menus, a mouse and other sophisticated techniques, and is typical of the ever-expanding capabilities you can expect to exercise with your Commodore 64C.



CHAPTER

Getting Started  
in BASIC

2



**CHAPTER 2**  
**Getting Started**  
**in BASIC**

<b>BASIC PROGRAMMING LANGUAGE</b>	<b>13</b>
Direct Mode	13
Program Mode	13
<b>USING THE KEYBOARD</b>	<b>14</b>
Keyboard Character Sets	14
Using the Command Keys	14
Function Keys	19
Displaying Graphic Characters	20
Rules for Typing BASIC Language Programs	20
<b>GETTING STARTED—The PRINT Command</b>	<b>21</b>
Printing Numbers	21
Using the Question Mark to Abbreviate the PRINT Command	21
Printing Text	22
Printing in Different Colors	23
Using the Cursor Keys Inside Quotes with the PRINT Command	23
<b>BEGINNING TO PROGRAM</b>	<b>24</b>
What a Program Is	24
Line Numbers	24
Viewing your Program—The LIST Command	25
A Simple Loop—The GOTO Statement	25
Clearing the Computer's Memory—The NEW Command	26
Using Color in a Program	26
<b>EDITING YOUR PROGRAM</b>	<b>27</b>
Erasing a Line from a Program	27
Duplicating a Line	27
Replacing a Line	27
Changing a Line	28
<b>MATHEMATICAL OPERATIONS</b>	<b>28</b>
Addition and Subtraction	28
Multiplication and Division	29
Exponentiation	29
Order of Operations	29
Using Parentheses to Define the Order of Operations	30
<b>CONSTANTS, VARIABLES AND STRINGS</b>	<b>30</b>
Constants	30
Variables	31
Strings	32

<b>SAMPLE PROGRAM</b>	<b>33</b>
<b>STORING AND REUSING YOUR PROGRAMS</b>	<b>34</b>
<b>Formatting a Disk</b>	<b>34</b>
<b>SAVEing on Disk</b>	<b>35</b>
<b>SAVEing on Cassette</b>	<b>35</b>
<b>LOADing from Disk</b>	<b>36</b>
<b>LOADing from Cassette</b>	<b>36</b>
<b>Other Disk-Related Commands</b>	<b>37</b>



## **BASIC Programming Language**

The BASIC programming language is a special language that lets you communicate with your Commodore 64C. Using BASIC is one means by which you instruct your computer what to do.

BASIC has its own vocabulary (made up of **commands, statements and functions**) and its own rules of structure (called **syntax**). You can use the BASIC vocabulary and syntax to create a set of instructions called a **program**, which your computer can then perform or “**run.**”

Using BASIC, you can communicate with your Commodore 64C in two ways: within a program, or directly (outside a program).

### **Direct Mode**

Your Commodore 64C is ready to accept BASIC commands in **direct** mode as soon as you turn on the computer. In the direct mode, you type commands on the keyboard and enter them into the computer by pressing the RETURN key. The computer executes all direct mode commands immediately after you press the RETURN key. Most BASIC commands in your Commodore 64C can be used in direct mode as well as in a program.

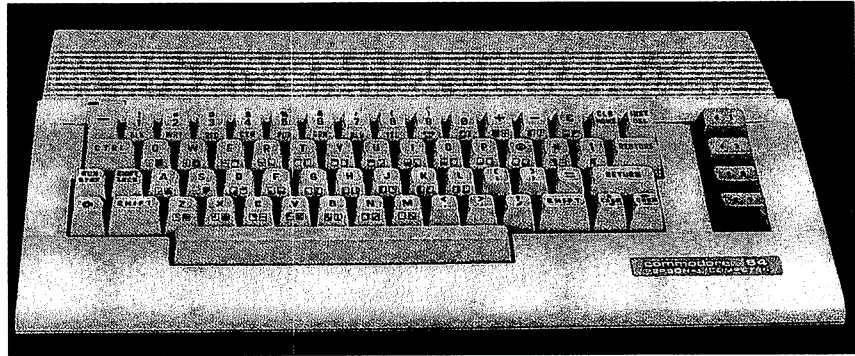
### **Program Mode**

In **program mode** you enter a set of instructions that perform a specific task. Each instruction is contained in a sequential **program** line. A statement in a program may be as long as 80 characters; this is equivalent to two full screen lines in 40-column format.

Once you have typed a program, you can use it immediately by typing the RUN command and pressing the RETURN key. You can also store the program on disk or tape by using the SAVE command. Then you can recall it from the disk or tape by using the LOAD command. This command copies the program from the disk or tape and places that program in the Commodore 64C's memory. You can then use or “execute” the program again by entering the RUN command. All these commands are explained later in this section. Most of the time you will be using your computer with programs, including programs you yourself write, and commercially available software packages. The only time you operate in direct mode is when you are manipulating or editing your programs with commands such as LIST, LOAD, SAVE and RUN. As a rule, the difference between direct mode and operation within a program is that direct mode commands have no line numbers.

## Using the Keyboard

Shown below is the keyboard of the Commodore 64C Personal Computer.



64C Keyboard

### Keyboard Character Sets

The Commodore 64C keyboard offers two different sets of characters:

- Upper-case letters and graphic characters
- Upper- and lower-case letters

You can use only one character set at a time.

When you turn on the Commodore 64C, the keyboard is normally using the upper-case/graphic character set. This means that everything you type is in capital letters. To switch back and forth between the two character sets, press the SHIFT key and the **C** key (the COMMODORE key) at the same time. To practice using the two character sets turn on your computer and press several letters or graphic characters. Then press the SHIFT key and the **C** (Commodore) key. Notice how the screen changes to upper- and lower-case characters. Press SHIFT and **C** again to return to the upper-case and graphic character set.

### Using the Command Keys

COMMAND keys are keys that send messages to the computer. Some command keys (such as RETURN) are used by themselves. Other command keys (such as SHIFT, CTRL, **C** and RESTORE) are used with other keys. The use of each of the command keys is explained below.

#### Return

When you press the RETURN key, what you have typed is sent to the Commodore 64C computer's

memory. Pressing the RETURN key also moves the cursor (the small flashing rectangle that marks where the next character you type will appear) to the beginning of the next line.

At times you may misspell a command or type in something the computer does not understand. Then, when you press the RETURN key, you probably will get a message like SYNTAX ERROR on the screen. This is called an "Error Message." Appendix A lists the error messages and tells how to correct the errors.

**NOTE:** In the examples given in this book, the following symbol indicates that you must press the RETURN key:

RETURN

## Shift

There are two SHIFT keys on the bottom row of the keyboard. One key is on the left and the other is on the right, just as on a standard typewriter keyboard.

The SHIFT key can be used in three ways:





1. With the upper/lower-case character set, the SHIFT key is used like the shift key on a regular typewriter. When the SHIFT key is held down, it lets you print capital letters or the top characters on double-character keys.
2. The SHIFT key can be used with some of the other command keys to perform special functions.
3. When the keyboard is set for the upper-case/graphic character set, you can use the SHIFT key to print the graphic symbols or characters that appear on the front face of certain keys. See the paragraphs entitled "Displaying Graphic Characters" later in this chapter for more details.

## Shift Lock

When you press this key down, it locks into place. Then, whatever you type will either be a capital letter, or the top character of a double-character key. To release the lock, press down on the SHIFT LOCK key again.

## Moving the Cursor Using the CRSR keys

You can use the two keys on the right side of the bottom row of the main keyboard to move the cursor:

- Pressing the  key alone moves the cursor **down**.
- Pressing the  and SHIFT keys together moves the cursor **up**.
- Pressing the  key alone moves the cursor **right**.
- Pressing the  and SHIFT keys together moves the cursor **left**.

You don't have to keep tapping a cursor key to move more than one space. Just hold the key down and the cursor continues to move until it reaches the position you want.

Notice that when the cursor reaches the right side of the screen, it "wraps", or starts again at the beginning of the next row. When moving left, the cursor will move along the line until it reaches the edge of the screen, then it will jump up to the end of the preceding line.

You should try to become very familiar with the cursor keys, because moving the cursor makes your programming much easier. With a little practice you will find that you can move the cursor almost without thinking about it.

## Inst/Del

This is a dual purpose key. INST stands for INSerT, and DEL for DELEte.

### **Inserting Characters**

You must use the SHIFT key with the INST/DEL key when you want to insert characters in a line. Suppose you left some characters out of a line, like this:

**WHILE U WERE OUT**

To insert the missing characters, first use the cursor keys to move the cursor back to the error, like this:

**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:

**WHILE ■ U WERE OUT**

Notice that INST doesn't move the cursor; it just adds space between the cursor and the character to its right. To make the correction, simply type in the missing "Y" and "O", like this:

**WHILE YOU WERE OUT**

### **Deleting Characters**

When you press the DEL key, the cursor moves one space to the left and erases the character that is there. This means that when you want to delete something, you move the cursor just to the right of the character you want to DELETE. Suppose you have made a mistake in typing, like this:

**PRINT "ERROER"**

You wanted to type the word ERROR, not ERROER. To delete the incorrect E that precedes the final R, position the cursor in the space where the final R is located. When you press the DEL key, the character to the right of the cursor (the R) automatically moves over one space to the left. You now have the correct wording like this:

**PRINT "ERROR"**

### Using INSerT and DELEte Together

You can use the INSerT and DELEte functions together to fix incorrect characters. First, move the cursor to the incorrect characters and press the INST/DEL key by itself to delete the characters. Next, press the SHIFT key and the INST/DEL key together to add any necessary space. Then type in the corrections. You can also type directly on top of undesired characters, then use INST to add any needed space.

### CTRL

The CTRL (Control) key is used with other keys to do special tasks called control functions. To perform a control function, hold down the CTRL key while you press some other key. Control functions are often used in prepackaged software such as a word processing system.

One control function that is used often is setting the character and cursor color. To select a color, hold down the CTRL key while you press a number key (1 through 8), on the top row of the keyboard. There are eight more colors available to you; these can be selected with the **C** key, as explained later.

Pressing CTRL during a screen printout will slow the display.

### Run/Stop

This is a dual function key. Under certain conditions you can use the RUN function of this key by pressing the SHIFT and RUN/STOP keys together. It is also possible to use the STOP function of this key to halt a program or a printout by pressing this key while the program is running. However, in most prepackaged programs, the STOP function of the RUN/STOP key is intentionally disabled (made unusable). This is done to prevent the user from trying to stop a program that is running before it reaches its normal end point. If the user were able to stop the program, valuable data could be lost.

### Restore

The RESTORE key is used with the RUN/STOP key to return the computer to its standard condition.

To do this, hold down the RUN/STOP key and press RESTORE.

Most prepackaged programs disable the RESTORE key for the same reason they disable the STOP function of the RUN/STOP key: to prevent losing valuable data.

### **CLR/Home**

CLR stands for CLear. HOME refers to the upper-left corner of the screen, which is called the HOME position. If you press this key by itself the cursor returns to the HOME position. When you use the SHIFT key with the CLR/HOME key, the screen CLearS and the cursor returns to the HOME position.

### **Commodore Key (C)**

The **C** key (known as the COMMODORE key) has a number of functions, including the following ones:

1. The **C** key lets you switch back and forth between the upper/lower-case character set (which displays the letters and characters on the top of the keys), and the upper-case/graphic display character set (which displays capital letters and the graphics symbols on the front face of the keys). To switch modes, press the **C** key and the SHIFT key at the same time.
2. The **C** key also lets you use a second set of eight colors for the cursor. To get these colors, you hold down the **C** key while you press a number key (1 through 8) in the top row.

### **Function Keys**

The four large keys located to the right of the main keyboard (marked F1, F3, F5 and F7 on the top and F2, F4, F6 and F8 on the front) are called **function keys**. These keys are often used by prepackaged software to allow you to perform a task with a single keystroke.




## Displaying Graphic Characters

To display the graphic symbol on the right front face of a key, hold down the SHIFT key while you press the key that has the graphic character you want to print. You can display the right side graphic characters only when the keyboard is in the upper-case/graphics character set (the normal character set usually available at power-up).

To display the graphic character on the left front face of a key, hold down the **⇧** key while you press the key that has the graphic character you want. You can display the left graphic character while the keyboard is in either character set.

## Rules for Typing BASIC Language Programs

You can type and use BASIC language programs even without knowing BASIC. You must type carefully, however, because a typing error may cause the computer to reject your information. The following guidelines will help minimize errors when typing or copying a program listing.

1. Spacing between words is usually not critical; e.g., typing FORT = 1TO10 is the same as typing FOR T = 1 TO 10. However, a BASIC keyword itself must not be broken up by spaces or connected to or contained in another word or connected to a letter. For example, typing TAND15 gives a SYNTAX ERROR message, while typing T AND 15 is valid. (See the BASIC 2.0 Encyclopedia in Chapter 6 for a list of BASIC keywords).
2. Any characters can be typed inside quotation marks. Some characters have special functions when placed inside quotation marks. These functions are explained later in this Guide.
3. Be careful with punctuation marks. Commas, colons and semicolons also have special properties, explained later in this chapter.
4. Always press the RETURN key (indicated in this Guide by ) after completing a numbered line.
5. Never type more than 80 characters in a program line. Remember, this is the same as two full screen lines.
6. Distinguish clearly between the letter I and the numeral 1 and between the letter O and the numeral 0.
7. The computer does not execute anything following the letters REM on a program line. REM stands for REMark. You can use the REM statement to put comments in your program that tell anyone listing the program what is happening at a specific point.

## Getting Started— The PRINT Command

The PRINT command tells the computer to display information on the screen. You can print both numbers and text (letters), but there are special rules for each case, described in the following paragraphs.

### Printing Numbers

To print numbers, use the PRINT command followed by the number(s) you want to print. Try typing this on your Commodore 64C:

```
PRINT 5
```

Then press the RETURN key. Notice the number 5 is now displayed on the screen.

Now type this and press RETURN:

```
PRINT 5,6
```

In this PRINT command, the comma tells the Commodore 64C that you want to print more than one number. When the computer finds commas in a string of numbers in a PRINT statement, each number that follows a comma is printed starting in either the 11th, 21st or 31st column on the screen, depending on the length of each number. If the previous number has more than 7 digits, the following number is moved to the next starting position, 10 columns to the right. The 64C always leaves at least two spaces plus one space for a sign between numbers which are separated by a comma. (For example, a negative number like - 4 will be preceded by two spaces when it follows a comma.) If you don't want all the extra spaces, use a semicolon (;) in your PRINT statement instead of a comma. The semicolon tells the computer not to add any spaces between strings. Numbers and numeric variables are printed with either a leading space or a minus sign, and a trailing space. Omitting a semicolon, a comma, or any separators acts the same as a semicolon. Type these examples and see what happens:

```
PRINT 5;6 RETURN
```

```
PRINT 100;200;300;400;500 RETURN
```

### Using the Question Mark to Abbreviate the PRINT Command

You can use a question mark (?) as an abbreviation for the PRINT command. Many of the examples in this section use the ? symbol in place of the word PRINT. In fact, most of the BASIC commands can be abbreviated. However, when you LIST a program, the keyword appears in the long version. The abbreviations for BASIC commands can be found in Appendix H of this Guide.

## Printing Text

To print text, first type the PRINT command (i.e., the word PRINT or a question mark). Then type quotation marks, followed by the words or characters you want to display, and another set of quotation marks. Then press the RETURN key. Remember that any words or characters you want to display must be typed on the screen with a quote symbol at each end of the string of characters. **String is the BASIC name for any set of characters surrounded by quotes.** The quote character is obtained by pressing SHIFT and the numeral 2 key on the top row of the keyboard. Try these examples:

```
? "COMMODORE 64C" RETURN
```

```
? "4*5" RETURN
```

Notice that when you press RETURN, the computer displays the characters within the quotes on the screen. Also note that the second example did not calculate  $4*5$  since it was treated as a string and not a mathematical calculation. If you want to calculate the result  $4*5$ , use the following command:

```
? 4*5 RETURN
```

You can PRINT any string you want by using the PRINT command and surrounding the printed characters with quotes. You can combine text and calculations in a single PRINT command like this:

```
? "4*5 = "4*5" RETURN
```

See how the computer PRINTS the characters in quotes, makes the calculation and PRINTS the result. It doesn't matter whether the text or calculation comes first. In fact, you can use both several times in one PRINT command. Type the following statement:

```
? 4*(2+3) " is the same as "4*5" RETURN
```

















Notice that even spaces inside the quotation marks are printed on the screen. Type:

```
? " OVER HERE" RETURN
```

## Printing in Different Colors

The Commodore 64C is capable of displaying 16 different colors on the screen. You can change colors easily. All you do is hold down the CTRL key and press a numbered key between 1 and 8 on the top row of the main keyboard. Notice that the cursor changes color according to the numbered key you pressed. All the succeeding characters are displayed in the color you selected. Hold down the **C** key and press a numbered key between 1 and 8, and eight additional colors are displayed on the screen.

The following table lists the colors available using the CTRL and **C** keys. The table also shows the key used to specify a given color, and the resulting control character that appears within the quotes of a PRINT statement.

KEYBOARD	COLOR	DISPLAY	KEYBOARD	COLOR	D:SPRAY
CTRL 1	BLACK		<b>C</b> 1	ORANGE	
CTRL 2	WHITE		<b>C</b> 2	BROWN	
CTRL 3	RED		<b>C</b> 3	LT. RED	
CTRL 4	CYAN		<b>C</b> 4	GRAY 1	
CTRL 5	PURPLE		<b>C</b> 5	GRAY 2	
CTRL 6	GREEN		<b>C</b> 6	LT. GREEN	
CTRL 7	BLUE		<b>C</b> 7	LT. BLUE	
CTRL 8	YELLOW		<b>C</b> 8	GRAY 3	

## Using the Cursor Keys Inside Quotes with the PRINT Command

When you type the cursor keys inside quotation marks, graphic characters are shown on the screen to represent the keys. These characters will NOT be printed on the screen when you press RETURN. Try typing a question mark (?), open quotes (SHIFTed 2 key); then press either of the down cursor keys 10 times, enter the words "DOWN HERE", and close the quotes. The line should look like this:

**“QQQQQQQQQQQ DOWN HERE”**

Now press RETURN. The Commodore 64C prints 10 blank lines, and on the eleventh line, it prints "DOWN HERE". As this example shows, you can tell the computer to print anywhere on your screen by using the cursor control keys inside quotation marks.

## Beginning to Program

So far most of the commands we have discussed have been performed in DIRECT mode. That is, the command was executed as soon as the RETURN key was pressed. However, most BASIC commands and functions can also be used in programs.

### What a Program Is

A program is a set of numbered BASIC instructions that tells your computer what you want it to do. These numbered instructions are referred to as **statements** or **lines**.

### Line Numbers

The lines of a program are numbered so that the computer knows in what order you want them executed or RUN. The computer executes the program lines in numerical order, unless the program instructs otherwise. You can use any whole number from 0 to 63999 for a line number. **Never** use a comma in a line number.

Many of the commands you have learned to use in DIRECT mode can be easily made into program statements. For example, type this:

```
10 ? "COMMODORE 64C" RETURN
```

Notice the computer did not display COMMODORE 64C when you pressed RETURN, as it would do if you were using the PRINT command in DIRECT mode. This is because the number, 10, that comes before the PRINT symbol (?) tells the computer that you are entering a BASIC program. The computer just stores the numbered statement and waits for the next input from you.

Now type RUN and press RETURN. The computer prints the words COMMODORE 64C. This is not the same as using the PRINT command in DIRECT mode. What has happened here is that **YOU HAVE JUST WRITTEN AND RUN YOUR FIRST BASIC PROGRAM**. The program is still in the computer's memory, so you can run it as many times as you want.

## Viewing Your Program—The LIST Command

Your one-line program is still in the 64C's memory. Now clear the screen by pressing the SHIFT and CLR/HOME keys together. The screen is empty. At this point you may want to see the program listing to be sure it is still in memory. The BASIC language is equipped with a command that lets you do just this—the LIST command.

Type LIST and press RETURN. The 64C responds with:

```
10 PRINT "COMMODORE 64C"  
READY.
```

Anytime you want to see all the lines in your program, type LIST. This is especially helpful if you make changes, because you can check to be sure the new lines have been registered in the computer's memory. In response to the command, the computer displays the changed version of the line, lines, or program. Here are the rules for using the LIST command. (Insert the line number you wish to see in place of the N.)

- To see line N only, type LIST N and press RETURN.
- To see from line N to the end of the program, type LIST N- and press RETURN.
- To see the lines from the beginning of the program to line N, type LIST-N and press RETURN.
- To see from line N1 to line N2 inclusive, type LIST N1-N2 and press RETURN.

## A Simple Loop—The GOTO Statement

The line numbers in a program have another purpose besides putting your commands in the proper order for the computer. They serve as a reference for the computer in case you want to execute the command in that line repetitively in your program. You use the GOTO command to tell the computer to go to a line and execute the command(s) in it. Now type:

```
20 GOTO 10
```

When you press RETURN after typing line 20, you add it to your program in the computer's memory.

Notice that we numbered the first line 10 and the second line 20. It is very helpful to number program lines in increments of 10 (that is, 10, 20, 30, 40, etc.) in case you want to go back and add lines in between later on. You can number such added lines by fives (15, 25 . . .) ones (1, 2 . . .)—in fact, by any whole number—to keep the lines in the proper order.

Type RUN and press RETURN, and watch the words COMMODORE 64C move down your screen. To stop the message from printing on the screen, press the RUN/STOP key on the left side of your keyboard.

The two lines that you have typed make up a simple program that repeats itself endlessly, because the second line keeps referring the computer back to the first line. The program will continue indefinitely unless you stop it or turn off the computer.

Now type LIST ~~RETURN~~. The screen should say:

```
10 PRINT "COMMODORE 64C"  
20 GOTO 10  
READY.
```

Your program is still in memory. You can RUN it again if you want to. This is an important difference between PROGRAM mode and DIRECT mode. Once a command is executed in DIRECT mode, it is no longer in the computer's memory.

Notice that even though you used the ? symbol for the PRINT statement, your computer has converted it into the full command. This happens when you LIST any command you have abbreviated in a program.

### **Clearing the Computer's Memory—The NEW Command**

Anytime you want to start all over again or erase a BASIC program in the computer's memory, just type NEW and press RETURN. This command clears out the computer's BASIC memory, the area where programs and data are stored.

### **Using Color in a Program**

To select color within a program, you must include the color selection information within a PRINT statement. For example, clear your computer's memory by typing NEW and pressing RETURN, then type the following, being sure to leave space between each letter:

```
10 PRINT " S P E C T R U M" RETURN
```

Now type line 10 again but this time hold down the CTRL key and press the 1 key directly after entering the first set of quote marks. Release the CTRL key and type the "S". Now hold down the CTRL again and press the 2 key. Release the CTRL key and type the "P". Next hold down the CTRL again and press the 3 key. Continue this process until you have typed all the letters in the word SPECTRUM and selected a color between



## Editing Your Program

each letter. Press the SHIFT and the 2 keys to type a set of closing quotation marks and press the RETURN key. Now type RUN and press the RETURN key. The computer displays the word SPECTRUM with each letter in a different color. Now type LIST and press the RETURN key. Notice the graphic characters that appear in the PRINT statement in line 10. These characters tell the computer what color you want for each printed letter. Note that these graphic characters do not appear when the Commodore 64C PRINTs the word SPECTRUM in different colors.

The color selection characters, known as control characters, in the PRINT statement in line 10 tell the Commodore 64C to change colors. The computer then prints the characters that follow in the new color until another color selection character is encountered. While characters enclosed in quotation marks are usually PRINTed exactly as they appear, control characters are only displayed within a program LISTing.

The following paragraphs will help you to type in your programs and make corrections and additions to them.

### Erasing a Line from a Program

Use the LIST command to display the program you typed previously. Now type 10 and press RETURN. You just erased line 10 from the program. LIST your program and see for yourself. If the old line 10 is still on the screen, move the cursor up so that it is blinking anywhere on that line. Now, if you press RETURN, line 10 is back in the computer's memory.

### Duplicating a Line

Hold down the SHIFT key and press the CLR/HOME key on the upper right side of your keyboard. This will clear your screen. Now LIST your program. Move the cursor up again so that it is blinking on the "0" in the line numbered 10. Now type a 5 and press RETURN. You have just duplicated (i.e., copied) line 10. The duplicate line is numbered 15. Type LIST and press RETURN to see the program with the duplicated lines.

### Replacing a Line

You can replace a whole line by typing in the old line number followed by the text of the new line, then pressing RETURN. The old version of the line will be erased from memory and replaced by the new line as soon as you press RETURN.

## Mathematical Operations

### Changing a Line

Suppose you want to add something in the middle of a line. Simply move the cursor to the character or space that immediately follows the spot where you want to insert the new material. Then hold down the SHIFT key and the INST/DEL key together until there is enough space to insert your new characters.

Try this example. Clear the computer's memory by typing NEW and pressing RETURN. Then type:

```
10 ? "MY 64C IS GREAT" RETURN
```

Let's say that you want to add the word COMMODORE in front of the number 64C. Just move the cursor so that it is blinking on the "6" in 64C. Hold down the SHIFT and INST/DEL keys until you have enough room to type in COMMODORE (don't forget to leave enough room for a space after the E). Then type in the word COMMODORE.

If you want to delete something in a line (including extra blank spaces), move the cursor to the character following the material you want to remove. Then hold down the INST/DEL key by itself. The cursor will move to the left, and characters or spaces will be deleted as long as you hold down the INST/DEL key.

You can use the PRINT command to perform calculations like addition, subtraction, multiplication, division and exponentiation. You type the calculation after the PRINT command.

### Addition and Subtraction

Try typing these examples:

```
PRINT 6 + 4 RETURN
```

```
PRINT 50 - 20 RETURN
```

```
PRINT 10 + 15 - 5 RETURN
```

```
PRINT 75 - 100 RETURN
```

```
PRINT 30 + 40,55 - 25 RETURN
```

```
PRINT 30 + 40;55 - 25 RETURN
```

Notice that the fourth calculation (75-100) resulted in a negative number. Also notice that you can tell the computer to make more than one calculation with a single PRINT command. You can use either a comma or a semicolon in your command, depending on whether or not you want spaces separating your results.

## Multiplication and Division

Find the asterisk key (\*) on the right side of your keyboard. This is the symbol that the Commodore 64C uses for multiplication. The slash (/) key, located next to the right SHIFT key, is used for division.

Try these examples:

```
PRINT 5*3 RETURN
```

```
PRINT 100/2 RETURN
```

## Exponentiation

Exponentiation means to raise a number to a power. The up arrow key ( $\uparrow$ ), located next to the asterisk on your keyboard, is used for exponentiation. If you want to raise a number to a power, use the PRINT command, followed by the number, the up arrow and the power, in that order. For example, to find out what 3 squared is, type:

```
PRINT 3↑2 RETURN
```

## Order of Operations

You have seen how you can combine addition and subtraction in the same PRINT command. If you combine multiplication or division with addition or subtraction operations, you may not get the result you expect. For example, type:

```
PRINT 4 + 6/2 RETURN
```

If you assumed you were dividing 10 by 2, you were probably surprised when the computer responded with the answer 7. The reason you got this answer is that multiplication and division operations are performed by the computer before addition or subtraction. Multiplication and division are said to take precedence over addition and subtraction. It doesn't matter in what order you type the operation. In computing, the order in which mathematical operations are performed is known as the order of operations.

Exponentiation, or raising a number to a power, takes precedence over the other four mathematical operations. For example, if you type:

```
PRINT 16/4↑2 RETURN
```

the Commodore 64C responds with a 1 because it squares the 4 before it divides 16.

## Using Parentheses to Define the Order of Operations

You can tell the Commodore 64C which mathematical operation you want performed first by enclosing that operation in parentheses in the PRINT command. For instance, in the first example above, if you want to tell the computer to add before dividing, type:

```
PRINT (4 + 6)/2 RETURN
```

This gives you the desired answer, 5.

If you want the computer to divide before squaring in the second example, type:

```
PRINT (16/4)↑2 RETURN
```

Now you have the expected answer, 16.

If you don't use parentheses, the computer performs the calculations according to the above rules. When all operations in a calculation have equal precedence, they are performed from left to right. For example, type:

```
PRINT 4*5/10*6 RETURN
```

Since the operations in this example are performed in order from left to right, the result is 12 ( $4*5 = 20 \dots 20/10 = 2 \dots 2*6 = 12$ ). If you want to divide  $4*5$  by  $10*6$  you type:

```
PRINT (4*5)/(10*6) RETURN
```

The answer is now .33333333.

## Constants, Variables and Strings

### Constants

Constants are numeric values that are permanent: that is, they do not change in value over the course of an equation or program. For example, the number 3 is a constant, as is any number. This statement illustrates how your computer uses constants:

```
10 PRINT 3
```

No matter how many times you execute this line, the answer will always be 3.

## Variables

Variables are values that can change over the course of an equation or program statement. There is a part of the computer's BASIC memory that is reserved for the characters (numbers, letters and symbols) you use in your program. Think of this memory as a number of storage compartments in the computer that store information about your program; this part of the computer's memory is referred to as variable storage. Type in this program:

```
10 X=5
20 ?X
```

Now RUN the program and see how the computer prints a 5 on your screen. You told the computer in line 10 that the letter X will represent the number 5 for the remainder of the program. The letter X is called a variable, because the value of X varies depending on the value to the right of the equals sign. We call this an assignment statement because now there is a storage compartment labeled X in the computer's memory, and the number 5 has been assigned to it. The = sign tells the computer that whatever comes to the right of it will be assigned to a storage compartment (a memory location) labeled with the letter X to the left of the equals sign.

The variable name on the left side of the = sign can be either one or two letters, or one letter and one number (the letter MUST come first). The names can be longer, but the computer only looks at the first two characters. This means the names PA and PART would refer to the same storage compartment. Also, the words used for BASIC commands (LOAD, RUN, LIST, etc.) or functions (INT, ABS, SQR, etc.) cannot be used as names in your programs. Refer to the BASIC Encyclopedia in Chapter 5 if you have any questions about whether a variable name is a BASIC keyword. Notice that the = in assignment statements is not the same as the mathematical symbol meaning "equals", but rather means allocate a variable (storage compartment) and assign a value to it.

In the sample program you just typed, the value of the variable X remains at 5 throughout. You can put calculations to the right of the = sign to assign the result to a variable. You can mix text with constants in a PRINT statement to identify them. Type NEW and press RETURN to clear the 64C's memory; then try this program:

```
10 A=3*100
20 B=3*200
30 ?"A IS EQUAL TO "A
40 ?"B IS EQUAL TO "B
```

Now there are two variables, labeled A and B, in the computer's memory, containing the numbers 300 and 600 respectively. If, later in the program, you want to change the value of a variable, just put another assignment statement in the program. Add these lines to the program above and RUN it again.

```
50 A = 900*30/10
60 B = 95 + 32 + 128
70 GOTO 30
```

You'll have to press the STOP key to halt the program.

Now LIST the program and trace the steps taken by the computer. First, it assigns the value to the right of the = sign in line 10 to the letter A. It does the same thing in line 20 for the letter B. Next, it prints the messages in lines 30 and 40 that give you the values of A and B. Finally, it assigns new values to A and B in lines 50 and 60. The old values are replaced and cannot be recovered unless the computer executes lines 10 and 20 again. When the computer is sent to line 30 to begin printing the values of A and B again, it prints the new values calculated in lines 50 and 60. Lines 50 and 60 reassign the same values to A and B and line 70 sends the computer back to line 30. This is called an endless loop, because lines 30 through 70 are executed over and over again until you press the RUN/STOP key to halt the program. Other methods of looping are discussed later in this and the following two sections.

### Strings

A string is a character or group of characters enclosed in quotes. These characters are stored in the computer's memory as a variable in much the same way numeric variables are stored. You can also use variable names to represent strings, just as you use them to represent numbers. When you put the dollar sign (\$) after the string variable name, it tells the computer that the name is for a string variable, and not a numeric variable.

Type NEW and press RETURN to clear your computer's memory, then type in the program below:

```
10 A$ = "COMMODORE "
20 X = 64C
30 B$ = " COMPUTER"
40 Y = 1
50 ? "THE "A$;X;B$" IS NUMBER "Y
```

## Sample Program

See how you can print numeric and string variables in the same statement? Try experimenting with variables in your own short programs.

You can print the value of a variable in DIRECT mode, after the program has been RUN. Type ?A\$;B\$;X;Y after running the program above and see that those four variable values are still in the computer's memory.

If you want to clear this area of BASIC memory but still leave your program intact, use the CLR command. Just type CLR <RETURN> and all constants, variables and strings are erased. But when you type LIST, you can see the program is still in memory. The NEW command discussed earlier erases both the program and the variables.

Here is a sample program incorporating many of the techniques and commands discussed in this section.

This program calculates the average of three numbers (X, Y and Z) and prints their values and their averages on the screen. You can edit the program and change the assignments in lines 10 through 30 to change the values of the variables. Line 40 adds the variables and divides by 3 to get the average. Note the use of parentheses to tell the computer to add the numbers before it divides.

**TIP:** Whenever you are using more than one set of parentheses in a statement, it's a good idea to count the number of left parentheses and right parentheses to make sure they are equal.

```
10 X = 46
20 Y = 72
30 Z = 114
40 A = (X + Y + Z)/3
60 ?"THE AVERAGE OF"X;Y;"AND "Z;"IS" A;
90 END
```

## Storing and Reusing Your Programs

Once you have created your program, you will probably want to store it permanently so you will be able to recall and use it at some later time. To do this, you'll need either a Commodore disk drive or a Commodore Datassette.

You will learn several commands that let you communicate between your computer and your disk drive or Datassette. These commands are constructed with the use of a command word followed by several parameters. Parameters are numbers, letters, words or symbols in a command that supply specific information to the computer, such as a filename, or a numeric variable that specifies a device number. Each command may have several parameters. For example, the parameters of the disk format command include a name for the disk and an identifying number or code, plus several other parameters. Parameters are used in almost every BASIC command; some are variables which change and others are constants. These are the parameters that supply disk information to the 64C and disk drive:

### Disk Handling Parameters

disk name—	arbitrary 16 character identifying name you supply.
file name—	arbitrary 16 character identifying name you supply.
i.d.—	arbitrary two-character identifier you supply
drive number—	must use 0 for a single disk drive, 0 or 1 in a dual drive.
device number—	a preassigned number for a peripheral device. For example, the device number for a Commodore disk drive is usually 8.

### Formatting a Disk

To store programs on a new (or blank) disk, you must first prepare the disk to receive data. This is called "formatting" the disk. **NOTE:** Make sure you turn on the disk drive before inserting any disk.

The formatting process divides the disk into sections called tracks and sectors. A table of contents, called a directory, is created. Each time you store a program on disk, the name you assign to that program will be added to the directory.



To format a blank disk type this command:

```
OPEN 15,8,15: PRINT# 15, "N.A$,B$ RETURN
```

In Place of A\$, type a disk name of your choice; you can use up to 16 characters to identify the disk. In place of B\$, type a two-character code of your choice (such as W2).

The cursor disappears for a second or so. When the cursor blinks again, seal the disk with the following command:

```
CLOSE 15 RETURN
```

The entire formatting process takes about a minute.

### **SAVEing on Disk**

You can store your program on disk by using the following command:

```
SAVE"PROGRAM NAME",8 RETURN
```

The program name can be any name you choose, up to 16 characters long. Be sure to enclose the program name in quotes. You cannot put two programs with the same name on the same disk. If you do, the second program will not be accepted; the disk will retain the first one. In the example, the 8 indicates that you are saving your program on device number 8.

### **SAVEing on Cassette**

If you are using a Datassette to store your program, insert a blank tape in the recorder, rewind the tape if necessary, and type:

```
SAVE "PROGRAM NAME" RETURN
```

You must type the word SAVE, followed by the program name. The program name can be any name you choose up to 16 characters.

**NOTE:** The screen will go blank while the program is being **SAVED**, but returns to normal when the process is completed.

Unlike disk, you can save two programs to tape under the same name. However when you load it back into the computer, the first program sequentially on the tape will be loaded, so avoid giving programs the same name.

Once a program has been **SAVED**, you can **LOAD** it back into the computer's memory and **RUN** it anytime you wish.

### **LOADing from Disk**

Loading a program simply copies the contents of the program from the disk into the computer's memory. If a **BASIC** program was already in memory before you issued the **LOAD** command, it is erased.

To load your **BASIC** program from a disk, use the following command:

**LOAD"PROGRAM NAME",8 ~~RETURN~~**

In the example, the 8 indicates to the computer that you are loading from device number 8. Be careful to type the program name exactly as you typed it when **SAVEing** the program, or the computer will respond "**FILE NOT FOUND.**"

Once the program is loaded, type **RUN** and press **RETURN** to execute.

### **LOADing from Cassette**

To **LOAD** your program from cassette tape, type:

**LOAD "PROGRAM NAME" ~~RETURN~~**

If you do not know the name of the program, you can type:

**LOAD ~~RETURN~~**

and the next program on the tape will be found. While the **Datassette** is searching for the program the screen is blank. When the program is found, the screen displays:

**FOUND PROGRAM NAME**

To actually load the program, you then press the **Commodore** key.

You can use the counter on the **Datassette** to identify the approximate starting position of the programs. Then, when you want to retrieve a program, simply wind the tape forward from 000 to the program's start location, and type:

**LOAD ~~RETURN~~**

In this case you don't have to specify the **PROGRAM NAME**; your program will load automatically because it is the next program on the tape.

## Other Disk-Related Commands

### Verifying a Program

To verify that a program has been correctly saved, use the following command:

```
VERIFY"PROGRAM NAME",8 RETURN
```

If the program in the computer is identical to the one on the disk, the screen display will respond with the letters "OK."

The VERIFY command also works for tape programs. You type:

```
VERIFY"PROGRAM NAME" RETURN
```

You do not enter the comma and a device number.

### Displaying Your Disk Directory

To see a list or *directory* of the programs on your disk, use the following command sequence:

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

The cursor disappears and the screen displays this message:

```
SEARCHING FOR $  
LOADING
```

When the directory (the \$ file) has been loaded, the READY message is displayed and the cursor reappears. You then type:

```
LIST RETURN
```

This lists the disk directory of files.

For further information on SAVEing and LOADing your programs, or other disk related information, refer to your Datassette or disk drive manual. Also consult the LOAD and SAVE command descriptions in Chapter 6, BASIC 2.0 Encyclopedia.

\*\*\*\*\*

*You now know something about the BASIC language and some elementary programming concepts. The next chapter builds on these concepts, introducing additional commands, functions and techniques that you can use to program in BASIC.*



CHAPTER

Advanced  
BASIC  
Programming

3



**CHAPTER 3**  
**Advanced BASIC**  
**Programming**

<b>COMPUTER DECISIONS—The IF-THEN Statement</b>	<b>43</b>
Using the Colon	44
<b>LOOPS—The FOR-NEXT Command</b>	<b>44</b>
Empty Loops—Inserting Delays in a Program	45
The STEP Command	46
<b>INPUTTING DATA</b>	<b>47</b>
The INPUT Command	47
Assigning a Value to a Variable	47
Prompt Messages	47
Sample Program	48
The GET Command	49
The READ-DATA Command	50
The RESTORE Command	52
Using Arrays	52
Subscripted Variables	53
Dimensioning Arrays	54
Sample Program	54
<b>PROGRAMMING SUBROUTINES</b>	<b>56</b>
The GOSUB-RETURN Command	56
The ON GOTO/GOSUB Command	56
<b>USING MEMORY LOCATION</b>	<b>57</b>
Using PEEK and POKE for RAM Access	57
Using PEEK	57
Using POKE	58
<b>BASIC FUNCTIONS</b>	<b>58</b>
What Is a Function?	58
The INTEGER Function (INT)	59
Generating Random Numbers—The RND Function	60
The ASC and CHR\$ Commands	60
Converting Strings and Numbers	61
The VAL Function	61
The STR\$ Function	61
The Square Root Function (SQR)	61
The Absolute Value Function (ABS)	61
<b>THE STOP AND CONT (CONTINUE) COMMANDS</b>	<b>62</b>





## Computer Decisions— The IF-THEN Statement

This chapter describes how to use a number of powerful BASIC commands, functions and programming techniques.

These commands and functions allow you to program repeated actions through looping and nesting techniques; handle tables of values; branch or jump to another section of a program, and return from that section; assign varying values to a quantity—and more. Examples and sample programs show just how these BASIC concepts work and interact.

In the preceding chapter you learned how to change the values of variables. The next step is to have the computer make decisions based on these updated values. You do this with the IF-THEN statement. You tell the computer to execute a command only IF a condition is true (e.g., IF X = 5). The command you want the computer to execute when the condition is true comes after the word THEN in the statement. Clear your computer's memory by typing NEW and pressing RETURN, then type this program:

```
10 J = 0
20 ? J, "COMMODORE 64C"
30 J = J + 1
40 IF J < 5 THEN 20
60 END
```

You no longer have to press the STOP key to break out of a looping program. The IF-THEN statement tells the computer to keep printing "COMMODORE 64C" and incrementing (increasing) J until J = 5 is true. When an IF condition is false, the computer jumps to the next line of the program, no matter what comes after the word THEN.

Notice the END command in line 60. It is good practice to put an END statement as the last line of your program. It tells the computer where to stop executing statements.

Below is a list of comparison symbols that may be used in the IF statement and their meanings:

SYMBOL	MEANING
=	EQUALS
>	GREATER THAN
<	LESS THAN
<>	NOT EQUAL TO
>=	GREATER THAN OR EQUAL TO
<=	LESS THAN OR EQUAL TO

These comparisons work in expected mathematical ways with numbers. There are different ways to determine if one string is greater than, less than, or equal to another. You can learn about these “string handling” functions by referring to Chapter 6, BASIC 2.0 Encyclopedia.

### Using the Colon

A very useful tool in programming is the colon (:). You can use the colon to separate two (or more) BASIC commands on the same line.

Statements after a colon on a line will be executed in order, from left to right. In one program line you can put as many statements as you can fit into 80 characters, including the line number. This is equivalent to two full screen lines in 40-column format. This provides an excellent opportunity to take advantage of the THEN part of the IF-THEN statement. You can tell the computer to execute several commands when your IF condition is true. Clear the computer’s memory, type in the following program and RUN it.

```
10 N = 0
15 N = N + 1
20 IF N < 5 THEN PRINT N; “LESS THAN 5”:GOTO 15
30 ? N; “GREATER THAN OR EQUAL TO 5”
40 END
```

Now change line 10 to read  $N = 20$ , and RUN the program again. Notice you can tell the computer to execute more than one statement when  $N$  is less than 5. You can put any statement(s) you want after the THEN command. Remember that the GOTO 15 will not be reached until  $N < 5$  is true. Any command that should be followed **whether or not the specified condition is met** should appear on a separate line.

### Loops—The FOR-NEXT Command

In the first RUN of the program used in the previous example, we made the computer print the variable  $N$  five times by telling it to increase or “increment” the variable  $N$  by units of one, until the value of  $N$  equalled five; then we ended the program. There is a simpler way to do this in BASIC. We can use a FOR-NEXT loop, like this:

```
10 FOR N = 1 TO 5
20 ?N; “IS LESS THAN OR EQUAL TO 5”
30 NEXT N
40 END
```

Type and RUN this program and compare the result with the result of the IF-THEN program—they are similar. In fact, the steps taken by the computer are almost identical for the two programs. The FOR-NEXT loop is a very powerful programming tool. You can specify the number of times the computer should repeat an action. Let's trace the computer's steps for the program above.

First, the computer assigns a value of 1 to the variable N. The 5 in the FOR statement in line 10 tells the computer to execute all statements between the FOR statement and the NEXT statement, until N is equal to 5. In this case there is just one statement—the PRINT statement.

This is how the computer interprets the inner workings of a FOR . . . NEXT loop—it operates in much the same way as the IF . . . THEN example on the previous page. First, the 64C assigns a value of 1 to the variable N. It then executes all instructions between the FOR and NEXT keywords. When the NEXT statement is encountered, it tells the computer to increment the counter variable N (in this case by 1), compare N to 5 and continue with another cycle through the FOR . . . NEXT loop if N = 5 is false. The increment defaults to 1 if no other increment is specified in the FOR statement. After five passes through the loop, and once N = 5 is true, the computer processes the statement which immediately follows the NEXT statement and resumes with the rest of the program. Since the computer does not compare the value of N to the start value of the loop variable until the NEXT statement is encountered, every loop is executed at least once.

### **Empty Loops—Inserting Delays in a Program**

Before you proceed any further, it will be helpful to understand about loops and some ways they are used to get the computer to do what you want. You can use a loop to slow down the computer (by now you have witnessed the speed with which the computer executes commands). See if you can predict what this program will do before you run it.

```
10 A$ = "COMMODORE 64C"  
20 FOR J = 1 TO 20  
30 PRINT  
40 FOR K = 1 TO 1500  
50 NEXT K  
60 PRINT A$  
70 NEXT J  
80 END
```

Did you get what you expected? The loop contained in lines 40 and 50 tells the computer to count to 1500 before executing the remainder of the program. This is known as a delay loop and is often useful. Because it is inside the main loop of the program, it is called a nested loop. Nested loops can be very useful when you want the computer to perform a number of tasks in a given order, and repeat the entire sequence of commands a certain number of times.

### The STEP Command

You can tell the computer to increment your counter by units (e.g. 10, 0.5 or any other number). You do this by using a STEP command with the FOR statement. For example, if you want the computer to count by tens to 100, type:

```
10 FOR X = 0 TO 100 STEP 10
20 ? X
30 NEXT
```

Notice that you do not need the X in the NEXT statement if you are only executing one loop at a time—NEXT refers to the most recent FOR statement. Also, note that you do not have to increase (or “increment”) your counter—you can decrease (or “decrement”) it as well. For example, change line 10 in the program above to read:

```
10 FOR X = 100 TO 0 STEP - 10
```

The computer will count backward from 100 to 0, in units of 10.

If you don't use a STEP command with a FOR statement, the computer will automatically increment the counter by units of 1.

The parts of the FOR-NEXT command in the original line 10 are:

- FOR — word used to indicate beginning of loop
- X — counter variable; any number variable can be used
- 0 — starting value; may be any number, positive or negative
- TO — connects starting value to ending value
- 100 — ending value; may be any number, positive or negative
- STEP — indicates an increment other than 1 will be used
- 10 — increment; can be any number, positive or negative

### The INPUT Command

#### Assigning a Value to a Variable

Clear the computer's memory by typing NEW and pressing RETURN, and then type and RUN this program.

```
10 K = 10
20 FOR I = 1 TO K
30 ? "COMMODORE"
40 NEXT
50 END
```

In this program you can change the value of K in line 10 to make the computer execute the loop as many times as you want it to. You have to do this when you are typing the program, before it is RUN. What if you wanted to be able to tell the computer how many times to execute the loop at the time the program is RUN?

In other words, you want to be able to change the value of the variable K each time you run the program, without having to change the program itself. We call this the ability to interact with the computer. You can have the computer ask you how many times you want it to execute the loop. To do this, use the INPUT command. For example, replace line 10 in the program with:

```
10 INPUT K
```

Now when you RUN the program, the computer responds with a ? to let you know it is waiting for you to enter what you want the value of K to be. Type 15 and press RETURN. The computer will execute the loop 15 times.

#### Prompt Messages

You can also make the computer print a message in an INPUT statement to tell you what variable it's waiting for. Replace line 10 with:

```
10 INPUT "PLEASE ENTER A VALUE
FOR K";K
```

Remember to enclose the message to be printed in quotes. This message is called a prompt. Also, notice that you must use a semicolon between the ending

quote marks of the prompt and the K. You may put any message you want in the prompt, but the INPUT statement (line number included) must fit within 80 characters, just as any BASIC command must.

The INPUT statement can also be used with string variables. The same rules that apply for numeric variables apply for strings. Don't forget to use the \$ to identify all your string variables. Clear your computer's memory by typing NEW and pressing RETURN. Then type in this program.

```
10 INPUT "WHAT IS YOUR NAME";N$
20 ? "HELLO ",N$
30 END
```

Now RUN the program. When the computer prompts "WHAT IS YOUR NAME?", type your name. Don't forget to press RETURN after you type your name.

Once the value of a variable (numeric or string) has been inserted into a program through the use of INPUT, you can refer to it by its variable name any time in the program. Type ?N\$ <RETURN>—your computer remembers your name.

### Sample Program

Now that you know how to use the FOR-NEXT loop and the INPUT command, clear the computer's memory by typing NEW ~~RETURN~~, then type the following program:

```
10 T=0
20 INPUT "HOW MANY NUMBERS";N
30 FOR J=1 TO N
40 INPUT "PLEASE ENTER A NUMBER ";X
50 T=T+X
60 NEXT
70 A=T/N
80 PRINT
90 ? "YOU HAVE";N"NUMBERS TOTALING";T
100 ? "AVERAGE =";A
110 END
```

This program lets you tell the computer how many numbers you want to average. You can change the numbers every time you run the program without having to change the program itself.

Let's see what the program does, line by line:

Line 10 assigns a value of 0 to T (which will be the running total of the numbers).

Line 20 lets you determine how many numbers to average, stored in variable N.

Line 30 tells the computer to execute a loop N times.

Line 40 lets you type in the actual numbers to be averaged.

Line 50 adds each number to the running total.

Line 60 tells the computer to go back to line 30, increment the counter (J) and start the loop again.

Line 70 divides the total by the amount of numbers you typed (N) after the loop has been executed N times.

Line 80 prints a blank line on the screen.

Line 90 prints the message that gives you the amount of numbers and their total.

Line 100 prints the average of the numbers.

Line 110 tells the computer that your program is finished.

### The GET Command

There are other BASIC commands you can use in your program to interact with the computer. One is the GET command which is similar to INPUT. To see how the GET command works, clear the computer's memory and type this program.

```
10 GET A$
20 IF A$ = "" THEN GOTO 10
30 ? A$
40 END
```

When you type RUN and press RETURN, nothing seems to happen. The reason is that the computer is waiting for you to press a key. The GET command, in effect, tells the computer to check the keyboard and find out what character or key is being pressed. The computer is satisfied with a null character (that is, no character). This is the reason for line 20. This line tells the computer that if it gets a null character, indicated by the two

double quotes with no space between them, it should go back to line 10 and try to GET another character. This loop continues until you press a key. The computer then assigns the character on that key to A\$.

The GET command is very important because you can use it, in effect, to program a key on your keyboard. The example below prints a message on the screen when Q is pressed. Type the program and RUN it. Then press Q and see what happens.

```
10 ?"PRESS Q TO VIEW MESSAGE"  
20 GET A$  
30 IF A$ = "" THEN GOTO 20  
40 IF A$ = "Q" THEN GOTO 60  
50 GOTO 20  
60 FOR I = 1 TO 25  
70 ? "NOW I CAN USE THE GET STATEMENT"  
80 NEXT  
90 END
```

Notice that if you try to press any key other than the Q, the computer will not display the message, but will go back to line 20 to GET another character.

### The READ-DATA Command

There is another powerful way to tell the computer what numbers or characters to use in your program. You can use the READ statement in your program to tell the computer to get a number or character(s) from the DATA statement. For example, if you want the computer to find the average of five numbers, you can use the READ and DATA statements this way:

```
10 T = 0  
20 FOR J = 1 TO 5  
30 READ X  
40 T = T + X  
50 NEXT  
60 A = T / 5  
70 ? "AVERAGE = "; A  
80 END  
90 DATA 5,12,1,34,18
```



When you run the program, the computer will print `AVERAGE = 14`. The program uses the variable `T` to keep a running total, and calculates the average in the same way as the `INPUT` average program. The `READ-DATA` average program, however, finds the numbers to average on a `DATA` line. Notice line 30, `READ X`. The `READ` command tells the computer there must be a `DATA` statement in the program. It finds the `DATA` line, and uses the first number as the current value for the variable `X`. The next time through the loop the second number in the `DATA` statement will be used as the value for `X`, and so on.

You can put any number you want in a `DATA` statement, but you cannot put calculations in a `DATA` statement. The `DATA` statement can be anywhere you want in the program—even after the `END` statement. This is because the computer never really executes the `DATA` statement; it just refers to it. Be sure to separate your data items with commas, but be sure not to put a comma between the word `DATA` and the first number in the list.

The computer uses an internal pointer to remind itself which piece of data was read last. After the computer reads the first number in the `DATA` statement, the pointer points to the second number. When the computer comes to the `READ` statement again, it assigns the second number to the variable name in the `READ` statement.

`DATA` statements can be placed anywhere in a program—at the beginning, in the middle, at the end, or interspersed throughout the program. For efficiency, `DATA` statements are usually placed at the end of program. If you have more than one `DATA` statement in your program, the internal `DATA` pointer will refer to the `DATA` statement containing the next unread `DATA` value.

You can use as many `READ` and `DATA` statements as you need in a program, but make sure there is enough data in the `DATA` statements for the computer to read. Remove one of the numbers from the `DATA` statement in the last program and run it again. The computer responds with `?OUT OF DATA ERROR IN 30`. What happened is that when the computer executed the loop for the fifth time, there was no data for it to read. That is what the error message is telling you. Putting too much into the `DATA` statement doesn't create a problem because the computer never realizes the extra data exists.

## The RESTORE Command

You can use the RESTORE command in a program to reset the data pointer to the first piece of data if you need to. Replace the END statement (line 80) in the program above with:

```
80 RESTORE
```

and add:

```
85 GOTO 10
```

Now RUN the program. The program will run continuously using the same DATA statement. NOTE: If the computer gives you an OUT OF DATA ERROR message, it is because you forgot to replace the number that you removed previously from the DATA statement, so the data is all used before the READ statement has been executed the specified number of times.

You can use DATA statements to assign values to string variables. The same rules apply as for numeric data. Clear the computer's memory and type the following program:

```
10 FOR J= 1 TO 3
20 READ A$
30 ? A$
40 NEXT
50 END
60 DATA COMMODORE,64C,COMPUTER
```

If the READ statement calls for a string variable, you can place letters or numbers in the DATA statement. Notice however, that since the computer is READING a string, numbers will be stored as a string of characters, not as a value which can be manipulated. Numbers stored as strings can be printed, but not used in calculations. Also, you cannot place letters in a DATA statement if the READ statement calls for a number variable.

## Using Arrays

You have seen how to use READ-DATA to provide many values for a variable. But what if you want the computer to remember all the data in the DATA statement instead of replacing the value of a variable with the new data? What if you want to be able to recall the third number, or the second string of characters?

Each time you assign a new value to a variable, the computer erases the old value in the variable's box in memory and stores the new value in its place. You can tell the computer to reserve a row of boxes in memory and store every value that you assign to that variable in your program. This row of boxes is called an array.

### **Subscripted Variables**

If the array contains all of the values assigned to the variable X in the READ-DATA example, it is called the X array. The first value assigned to X in the program is named X(1), the second value is X(2), and so on. These are called subscripted variables. The numbers in the parentheses are called subscripts. You can use a variable or a calculation as a subscript. The following is another version of the averaging program, this time using subscripted variables.

```
5 DIM X(5)
10 T = 0
20 FOR J = 1 TO 5
30 READ X(J)
40 T = T + X(J)
50 NEXT
60 A = T/5
70 ? "AVERAGE = ";A
80 END
90 DATA 5,12,1,34,18
```

Notice there are not many changes. Line 5 is the only new statement. It tells the computer to set aside six storage compartments [X(0) through X(5)] in memory for the X array. Line 30 has been changed so that each time the computer executes the loop, it assigns a value from the DATA statement to the position in the X array that corresponds to the loop counter (J). Line 40 calculates the total, just as it did before, but you must use a subscripted variable to do it.

After you run the program, if you want to recall the third number, type ?X(3)<RETURN>. The computer remembers every number in the array X. You can create string arrays to store the characters in string variables the same way. Try updating the

COMMODORE 64C COMPUTER READ-DATA program so the computer will remember the elements in the A\$ array.

```
5 DIM A$(3)
10 FOR J= 1 TO 3
20 READ A$(J)
30 ? A$(J)
40 NEXT
50 END
60 DATA COMMODORE,64C,COMPUTER
```

**TIP:** You do not need the DIM statement in your program unless the array uses values greater than A(10)—i.e., involves more than 11 elements. See DIMENSIONING ARRAYS.

### Dimensioning Arrays

Arrays can be used with nested loops, so the computer can handle data in a more advanced way. What if you had a large chart with 10 rows and 5 numbers in each row. Suppose you wanted to find the average of the five numbers in each row. You could create 10 arrays and have the computer calculate the average of the five numbers in each one. This is not necessary, because you can put all the numbers in a two-dimensional array. This array would have the same dimensions as the chart of numbers you want to work with—10 rows by 5 columns. The DIM statement for this array (we will call it array X) should be:

```
10 DIM X(10,5)
```

This tells the computer to reserve space in its memory for a two-dimensional array named X. The computer reserves enough space for 66 numbers. You do not have to fill an array with as many numbers as you DIMENSIONED it for, but the computer will still reserve enough space for all of the positions in the array.

### Sample Program

Now it becomes very easy to refer to any number in the chart by its column and row position. Refer to the chart below. Find the third element in the tenth row (1500). You would refer to this number as X(10,3) in your program. The program at the bot-

tom of this page reads the numbers from the chart into a two-dimensional array (X) and calculates the average of the numbers in each row.

Row	Column				
	1	2	3	4	5
1	1	3	5	7	9
2	2	4	6	8	10
3	5	10	15	20	25
4	10	20	30	40	50
5	20	40	60	80	100
6	30	60	90	120	150
7	40	80	120	160	200
8	50	100	150	200	250
9	100	200	300	400	500
10	500	1000	1500	2000	2500

```

10 DIMX(10,5),A(10)
20 FOR R = 1 TO 10
30 T = 0
35 FOR C = 1 TO 5
40 READ X(R,C)
50 T = T + X(R,C)
60 NEXT C
70 A(R) = T/5
80 NEXT R
90 FOR R = 1 TO 10
100 PRINT "ROW #";R
110 FOR C = 1 TO 5
120 PRINT X(R,C);NEXT C
130 PRINT "AVERAGE = ";A(R)
140 FOR D = 1 TO 1000:NEXT
150 NEXT R
160 DATA 1,3,5,7,9
170 DATA 2,4,6,8,10
180 DATA 5,10,15,20,25
190 DATA 10,20,30,40,50
200 DATA 20,40,60,80,100
210 DATA 30,60,90,120,150
220 DATA 40,80,120,160,200
230 DATA 50,100,150,200,250
240 DATA 100,200,300,400,500
250 DATA 500,1000,1500,2000,2500
260 END

```

## The GOSUB-RETURN Command

Until now, the only method you have had to tell the computer to jump to another part of your program is to use the GOTO command. What if you want the computer to jump to another part of the program, execute the statements in that section, then return to the point it left off and continue executing the program?

The part of program that the computer jumps to and executes is called a **subroutine**. Clear your computer's memory and enter the program below.

```
10 A$ = "SUBROUTINE":B$ = "PROGRAM"  
20 FOR J = 1 TO 5  
30 INPUT "ENTER A NUMBER";X  
40 GOSUB 100  
50 PRINT B$:PRINT  
60 NEXT  
70 END  
100 PRINT A$:PRINT  
110 Z = X↑2:PRINT Z  
120 RETURN
```

This program will square the numbers you type and print the result. The other print messages tell you when the computer is executing the subroutine or the main program. Line 40 tells the computer to jump to line 100, execute it and the statements following it until it sees a RETURN command. The RETURN statement tells the computer to go back in the program to the statement following the GOSUB command and continue executing. The subroutine can be anywhere in the program—including after the END statement. Also, remember that the GOSUB and RETURN commands must always be used together in a program (like FOR-NEXT and IF-THEN), otherwise the computer will give an error message.

## The ON GOTO/GOSUB Command

There is another way to make the computer jump to another section of your program (called branching). Using the ON statement, you can have the computer decide what part of the program to branch to based on a calculation or keyboard input. The ON statement is used with either the GOTO or GOSUB-RETURN commands, depending on what you need the program to do. A variable or calculation should be after the ON command. After the GOTO or GOSUB command, there should be a list of line numbers. Type the program below to see how the ON command works.

```

10 ? "ENTER A NUMBER BETWEEN ONE AND FIVE"
20 INPUT X
30 ON X GOSUB 100,200,300,400,500
40 END
100 ? "YOUR NUMBER WAS ONE":RETURN
200 ? "YOUR NUMBER WAS TWO":RETURN
300 ? "YOUR NUMBER WAS THREE":RETURN
400 ? "YOUR NUMBER WAS FOUR":RETURN
500 ? "YOUR NUMBER WAS FIVE":RETURN

```

When the value of X is 1, the computer branches to the first line number in the list (100). When X is 2, the computer branches to the second number in the list (200), and so on.

## Using Memory Locations

### Using PEEK and POKE for RAM/ROM Access

Each area of the computer's memory has a special function. For instance, there is a very large area to store your programs and the variables associated with them. This part of memory, called RAM, is cleared when you use the NEW command. Other areas are not as large, but they have very specialized functions. For instance, there is an area of memory locations that controls the music features of the computer.

There are two BASIC commands—PEEK and POKE—that you can use to access and manipulate the computer's memory. Use of PEEK and POKE commands can be a powerful programming device because the contents of the computer's memory locations determine exactly what the computer should be doing at a specific time.

#### Using PEEK

PEEK can be used to make the computer tell you what value is being stored in a memory location (a memory location can store any value between 0 and 255). You can PEEK the value of any memory location (RAM or ROM) in DIRECT or PROGRAM mode. Type:

```

P = PEEK(650) RETURN
? P RETURN

```

The computer assigns the value in memory location 650 to the variable P when you press RETURN after the first line. Then it prints the value when you press RETURN after entering the ? P command. Memory location 650 determines whether or not keys like the spacebar and CRSR repeat when you hold them

down. A 0 in location 650 tells the computer to repeat these keys when you hold them down. Hold down the spacebar and watch the cursor move across the screen.

### Using POKE

To change the value stored in a RAM location, use the POKE command. Type:

**POKE 650,96** RETURN

The computer stores the value after the comma (96) in the memory location before the comma (650). A 96 in memory location 650 tells the computer not to repeat keys like the spacebar and CRSR keys when you hold them down. (A value of 128 in location 650 allows all keys to repeat.) Now hold down the spacebar and watch the cursor. The cursor moves one position to the right, but it does not repeat. To return your computer to its normal state, type:

**POKE 650,0** RETURN

You cannot alter the value of all the memory locations in the computer—the values in ROM can be read, but not changed.

**NOTE:** Refer to the Commodore 64 Programmer's Reference Guide for a complete memory map of the Commodore 64C computer, this map shows you the contents of all memory locations.

## BASIC Functions

### What Is a Function?

A function is a predefined operation of the BASIC language that generally provides you with a single value. When the function provides the value, it is said to "return" the value. For instance, the SQR function is a mathematical function that returns the square root of a specific number.

There are two kinds of functions:

**Numeric**—returns a result which is a single number. Numeric functions range from calculating mathematical values to specifying the numeric value of a memory location.

**String**—returns a result which is a character.



Following are descriptions of some of the more commonly used functions. For a complete list of BASIC 2.0 functions see Chapter 6, BASIC 2.0 Encyclopedia.

### The INTEGER Function (INT)

What if you want to round off a number to the nearest integer? You'll need to use INT, the integer function. The INT function takes away (truncates) everything after the decimal point (for positive numbers only). Try typing these examples:

```
? INT(4.25) RETURN
? INT(4.75) RETURN
? INT(SQR(50)) RETURN
```

If you want to round off to the nearest whole number, then the second example should return a value of 5. In fact, you should round up any number with a decimal of 0.5 and above. To do this, you have to add 0.5 to the number before using the INT function. In this way, numbers with decimal portions of 0.5 and above will be increased by 1 before being rounded down by the INT function. Try this:

```
? INT(4.75 + 0.5) RETURN
```

The computer added 0.5 to 4.75 before it executed the INT function, so that it rounded 5.25 down to 5 for the result. If you want to round off the result of a calculation, do this:

```
? INT((100/6) + 0.5) RETURN
```

You can substitute any calculation for the division shown in the inner parentheses.

What if you want to round off numbers to the nearest 0.01? Instead of adding 0.5 to your number, add 0.005, then multiply by 100. Let's say you want to round 2.876 to the nearest 0.01. Using this method, you start with:

```
? (2.876 + 0.005)*100 RETURN
```

Now use the INT function to get rid of everything after the decimal point (which moves two places to the right when you multiply by 100). You are left with:

```
? INT((2.876 + 0.005)*100) RETURN
```

which gives you a value of 288. All that's left to do is divide by 100 to get the value of 2.88, which is the answer you want. Using this technique, you can round off calculations like the following to the nearest 0.01:

```
? INT((2.876 + 1.29 + 16.1-9.534 + 0.005)*100)/100 RETURN
```

## Generating Random Numbers—The RND Function

The RND function tells the computer to generate a random number. This can be useful in simulating games of chance, and in creating interesting graphic or music programs. All random (RND) numbers are nine digits, in decimal form, between the values 0.000000001 and 0.999999999. Type:

```
? RND (0) RETURN
```

Multiplying the randomly generated number by six makes the range of generated numbers increase to greater than 0 and less than 6. In order to include 6 among the numbers generated, we add one to the result of  $\text{RND}(0)*6$ . This makes the range  $1 < X < 7$ . If we use the INT function to eliminate the decimal places, the command will generate whole numbers from 1 to 6. This process can be used to simulate the rolling of a die. Try this program:

```
10 R = INT(RND(0)*6 + 1)
20 ? R
30 GOTO 10
```

Each number generated represents one toss of a die. To simulate a pair of dice, use two commands of this nature. Each number is generated separately, and the sum of the two numbers represents the total of the dice.

## The ASC and CHR\$ Functions

Every character that the Commodore 64C can display (including graphic characters) has a number assigned to it. This number is called a character string code (CHR\$) and there are 256 of them in the Commodore 64C. There are two functions associated with this concept that are very useful. The first is the ASC function. Type:

```
?ASC("Q") RETURN
```

The computer responds with 81. 81 is the character string code for the Q key. Substitute any character for Q in the command above to find out the Commodore ASCII code number for any character.

The second function is the CHR\$ function. Type:

```
?CHR$(81) RETURN
```

The computer responds with Q. In effect, the CHR\$ function is the opposite of the ASC function. They both refer to the table of character string codes in the computer's memory. CHR\$ values can be used to program function keys. See Appendix E of this Guide for a full listing of ASC and CHR\$ codes.

## Converting Strings and Numbers

Sometimes you may need to perform calculations on numeric characters that are stored as string variables in your program. Other times, you may want to perform string operations on numbers. There are two BASIC functions you can use to convert your variables from numeric to string type and vice versa.

**The VAL Function** The VAL function returns a numeric value for a string argument. Clear the computer's memory and type this program:

```
10 A$ = "64"  
20 A = VAL(A$)  
30 ? "THE VALUE OF ";A$;" IS";A  
40 END
```

**The STR\$ Function** The STR\$ function returns the string representation of a numeric value. Clear the computer's memory and type this program.

```
10 A = 65  
20 A$ = STR$(A)  
30 ? A" IS THE VALUE OF ";A$
```

## The Square Root Function (SQR)

The square root function is SQR. For example, to find the square root of 50, type:

```
? SQR(50) RETURN
```

You can find the square root of any positive number in this way.

## The Absolute Value Function (ABS)

The absolute value function (ABS) is very useful in dealing with negative numbers. You can use this function to get the positive value of any number—positive or negative. Try these examples:

```
? ABS(-10) RETURN
```

```
? ABS(5)" IS EQUAL TO "ABS(-5) RETURN
```

## The STOP and CONT (Continue) Commands

You can make the computer stop a program, and resume running it when you are ready. The STOP command must be included in the program. You can put a STOP statement anywhere you want to in a program. When the computer “breaks” from the program (that is, stops running the program), you can use DIRECT mode commands to find out exactly what is going on in the program. For example, you can find the value of a loop counter or other variable. This is a powerful device when you are “debugging” or fixing your program. Clear the computer’s memory and type the program below.

```
10 X = INT(SQR(630))
20 Y = (.025*80)^2
30 Z = INT(X*Y)
40 STOP
45 ? "RESUME PROGRAMMING"
50 A = (X * Y) + Z
80 END
```

Now RUN the program. The computer responds with “BREAK IN 40”. At this point, the computer has calculated the values of X, Y and Z. If you want to be able to figure out what the rest of the program is supposed to do, tell the computer to PRINT X;Y;Z. Often when you are debugging a large program (or a complex small one), you’ll want to know the value of a variable at a certain point in the program.

Once you have all the information you need, you can type CONT (for CONTInue) and press RETURN assuming you have not edited anything on the screen. The computer then CONTInues with the program, starting with the statement after the STOP command.

\*\*\*\*\*

*This chapter and the preceding one have been designed to familiarize you with the BASIC programming language and some of its capabilities. Remember that more information on every command and programming technique in this book can be found in the Commodore 64 Programmer's Reference Guide. The syntax for all Commodore 2.0 commands is given in Chapter 6, BASIC 2.0 Encyclopedia.*

CHAPTER

**Graphics,  
Color  
and Sprites**

**4**



**CHAPTER 4**  
**Graphics, Color and**  
**Sprites**

<b>COLOR CHARACTER STRING CODES (CHR\$)</b>	<b>67</b>
<b>COLOR REGISTERS—CHANGING SCREEN, BORDER AND CHARACTER COLORS</b>	<b>68</b>
<b>SCREEN MEMORY</b>	<b>70</b>
<b>COLOR MEMORY</b>	<b>72</b>
<b>ANIMATION</b>	<b>73</b>
<b>SPRITE GRAPHICS</b>	<b>77</b>
<b>Sprite Concepts</b>	<b>77</b>
<b>Designing a Sprite Image</b>	<b>78</b>
<b>Converting Your Sprite Image Into Data</b>	<b>80</b>
<b>Controlling Sprites</b>	<b>83</b>
<b>Animating Your Sprites</b>	<b>86</b>
<b>Tying Your Sprite Program Together</b>	<b>88</b>
<b>GRAPHICS MODES</b>	<b>91</b>

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100



## Color Character String Codes (CHR\$)

Your Commodore 64C gives you exceptional graphics capabilities. The Commodore 64C offers sixteen colors, five graphics modes and program-mable animated objects called sprites. This chapter elaborates on the several powerful graphics features built into the Commodore 64C and how they are used.

Each character on the 64C keyboard has a number associated with it. When you press a key, the computer scans the keyboard and understands exactly which character is typed. A character code value is entered into memory each time a key is pressed. These codes are referred to as character string codes. Appendix D lists all the character string codes for the Commodore 64C.

Within a program, you can select colors using character string codes instead of holding down the CTRL key and pressing a numbered key. For instance, enter the following sample program:

```
10 PRINT CHR$ (5) RETURN  
20 PRINT "WHITE" RETURN
```

**NOTE:** In the remainder of this section, the RETURN symbol is shown only after DIRECT mode statements, not after program lines.

When you RUN this program, the character color changes from blue to white and the word "WHITE" is displayed. The other 15 colors also have a character string code assigned to them. The following is a list of all the colors available on the Commodore 64C and the corresponding character string codes:

Color	CHR\$ Code	Color	CHR\$ Code
White	CHR\$ (5)	Dk. Gray	CHR\$ (151)
Red	CHR\$ (28)	Gray	CHR\$ (152)
Green	CHR\$ (30)	Lt. Green	CHR\$ (153)
Blue	CHR\$ (31)	Lt. Blue	CHR\$ (154)
Orange	CHR\$ (129)	Lt. Gray	CHR\$ (155)
Black	CHR\$ (144)	Purple	CHR\$ (156)
Brown	CHR\$ (149)	Yellow	CHR\$ (158)
Lt. Red	CHR\$ (150)	Cyan	CHR\$ (159)

To select any of the 64C colors, PRINT the above character string codes according to the colors you want to display on the screen. The following program illustrates how to select colors within a program.

```

10 PRINTCHR$(5)
15 PRINT"WHITE"
20 PRINTCHR$(28)
25 PRINT"RED"
30 PRINTCHR$(30)
35 PRINT"GREEN"
40 PRINTCHR$(31)
45 PRINT"BLUE"
47 PRINTCHR$(129)
48 PRINT"ORANGE"
50 PRINTCHR$(144)
55 PRINT"BLACK"
60 PRINTCHR$(149)
65 PRINT"BROWN"
70 PRINTCHR$(150)
75 PRINT"LT. RED"
80 PRINTCHR$(151)
85 PRINT"DK. GRAY"
90 PRINTCHR$(152)
95 PRINT"GRAY"
100 PRINTCHR$(153)
110 PRINT"LT. GREEN"
120 PRINTCHR$(154)
130 PRINT"LT. BLUE"
140 PRINTCHR$(155)
150 PRINT"LT. GRAY"
200 PRINTCHR$(156)
210 PRINT"PURPLE"
220 PRINTCHR$(158)
230 PRINT"YELLOW"
240 PRINTCHR$(159)
250 PRINT"CYAN"

```

## Color Registers— Changing Screen, Border and Character Colors

Your Commodore 64C has 64K of memory. This means the 64C holds 64 times 1024 (65536) bytes of information. Think of the internal structure of your computer as 65536 storage compartments piled one on top of the other. They are labeled starting from the bottom at location zero (0) and continue upward to location 65535 on top. You can also refer to each byte as a register, so your 64C has 65536 registers.

Each byte inside your computer is used for a specific purpose. For instance, you have 38911 bytes available to program in BASIC. Your Commodore 64C tells you this as soon as you turn on the computer and read the opening screen. You may ask, what are all the rest of the bytes used for? They control the computer's brain, known as the operating system. The operating system registers control all the features of your Commodore 64C.

A portion of the operating system controls graphics and color. You can select different colors by changing the contents of the 64C color registers. There are three color registers which control the colors of the border, the

background and the characters. When you first turn on your 64C, the background color is dark blue and the character and border colors are light blue. You can change the background, border and character color registers with the BASIC POKE statement.

The POKE command modifies the contents of the specified location and places the newly specified value in that location. The format of the POKE command is:

**POKE memory location, value**

For example, type the following POKE command:

```
POKE 53280,0 RETURN
```

Did you notice what happened? The border color changed from light blue to black. Location 53280 is the border color register. Location 53281 is the background color register and location 646 is the character color register. Now change the background color from dark blue to black with the following command:

```
POKE 53281,0 RETURN
```

Now all you need to know is how to change the character color with a POKE command. You already learned the two other methods to change the character color in the last section, first with the CTRL key and second with character string codes (CHR\$). The following POKE changes the character color from light blue to white:

```
POKE 646,1 RETURN
```

Note that the character color changes to white, but the characters already on the screen remain the same color as before. All the characters you type from now on are displayed in white unless you change the character color again.

You're probably wondering what the values that are POKEd into the color registers mean. These values are the color information codes for the 16 colors available on the Commodore 64C. The following list contains all the Commodore 64C colors and the corresponding color codes:

0	Black	8	Orange
1	White	9	Brown
2	Red	10	Light Red
3	Cyan	11	Dark Gray
4	Purple	12	Gray
5	Green	13	Light Green
6	Blue	14	Light Blue
7	Yellow	15	Light Gray

Try the following program. It uses FOR . . . NEXT loops, which you learned in the last chapter.

5 PRINT "☐": REM Use shifted CLR/HOME key to produce heart symbol shown in parentheses

```
10 FORI=0TO15
15 POKE53280,I
16 FORJ=1TO500:NEXT
18 NEXT
19 POKE53280,0
20 FORI=0TO15
25 POKE53281,I
26 FORJ=1TO500:NEXT
28 NEXT
29 POKE53281,0
30 FORI=0TO15
35 POKE646,I
36 PRINT"COLOR"
37 FORJ=1TO500:NEXT
38 NEXT
39 POKE646,14
50 POKE53280,14:POKE646,14:POKE53281,6
```

This program changes the color code value of each of the color registers using a FOR . . . NEXT loop. Lines 10 through 18 POKE each color value from 0 (black) to 15 (light gray) into the border color register and displays each border color on the screen. Lines 20 through 28 POKE each color value into the background color register and display each background color on the screen. Lines 30 through 38 POKE each color value into the character color register and display each character color on the screen.

Lines 16, 26 and 37 are FOR . . . NEXT loops that slow down the program. They are empty FOR . . . NEXT loops that delay program execution so you can notice the color changes on the screen. Try the program without the delay loops and see how fast the Commodore 64C runs. Line 40 restores the original border, screen and character color registers.

## Screen Memory

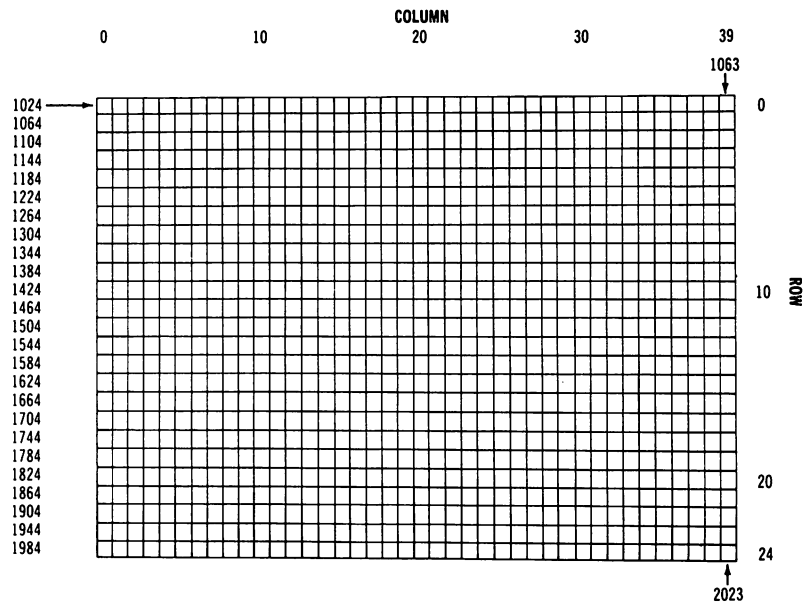
Since graphics is one of the Commodore 64C's strongest features, the screen is an important part of the computer. The 64C's screen has 1000 character positions—40 columns by 25 rows. Each character position uses one byte of memory, so the 64C needs 1000 bytes to store the information you see on the screen.

In the Color Register section, we referred to the memory of the Commodore 64C as 65536 storage compartments piled one on top of the other.

Screen memory uses part of those storage compartments starting at location 1024 and ending at location 2023. The screen appears as a grid having 40 X (horizontal) positions and 25 Y (vertical) positions. In memory, however the character positions are actually stored sequentially.

The top left character position on the screen, referred to as the HOME position, is stored at location 1024. The character position directly to the right of HOME is stored at location 1025 and so on. The character position at the top right corner of the screen is stored at location 1063, 40 locations past the beginning of screen memory. The last character position, located at the bottom right corner of the screen, is stored at location 2023, the end of screen memory. Examine Figure 4-1 to understand the correspondence between the way the screen looks and the way information is sequentially stored in memory.

**FIGURE 4-1. SCREEN MEMORY MAP**



Remember when you learned about character string codes in the Color Character String Code section? The Commodore 64C has a separate set of codes used only by screen memory to display characters on the screen. Instead of outputting characters to the screen in PRINT statements, you POKE a screen code value directly into a specific screen memory location. For example, enter the following line:

```
POKE 1024,1 RETURN
```

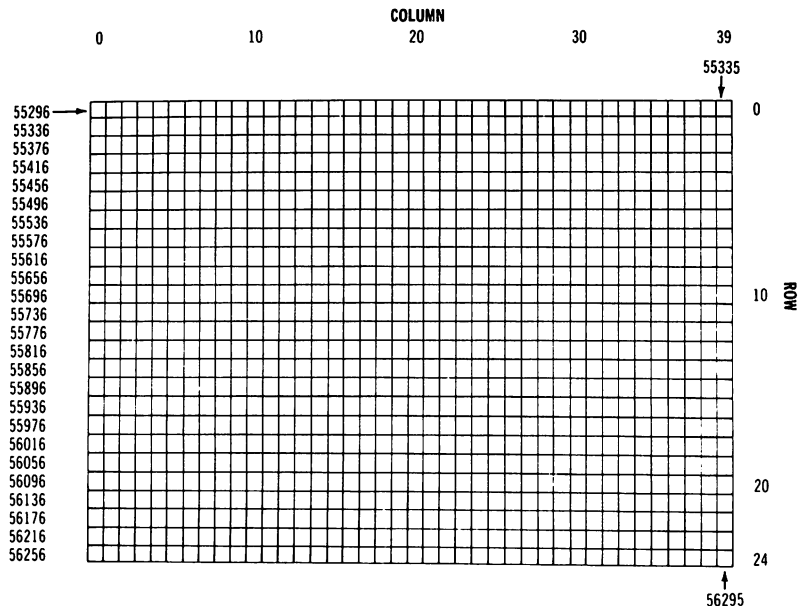
Did you notice what happened? The letter "A" is displayed in the upper left corner of the screen. Appendix C contains a list of the screen codes used in POKE statements to display characters on the screen. Notice that the screen codes in Appendix C and the character string codes in Appendix D are different. Appendix C contains screen codes that can only be POKEd directly into screen memory. Appendix D contains character string codes that are used more universally for inputting from and outputting to the screen, printer, disk drive and Datasette.

You can POKE any of the values in Appendix C into any of the screen locations between 1024 and 2023. Experiment with different characters and try displaying messages on the screen by POKEing a series of screen codes into consecutive screen memory locations. You can create character graphic images by POKEing different screen code graphic symbols in patterns that form picture images.

## Color Memory

Now that you have grasped the concept of screen memory, you need to know how to control the color of each character position on the screen. The Commodore 64C has a separate section of memory called COLOR MEMORY, that controls the color of information on the screen. The 64C uses 1000 bytes to store the color information for the 1000 character positions on the screen. Each screen memory location has a corresponding color memory location assigned to it. Compare Figure 4-1 with Figure 4-2, to understand the correspondence between screen memory and color memory and the way screen and color information are stored sequentially.

**FIGURE 4-2: COLOR MEMORY MAP**



Location 1024 in screen memory corresponds to location 55296 in color memory. Location 1063 corresponds to location 55335. Screen memory location 2023 corresponds to color memory location 56295. Remember, each screen location has a one to one correspondence to a color memory location that controls its color.

In the screen memory example you POKEd a 1 into location 1024 as follows:

```
POKE 1024,1 RETURN
```

This places the letter "A" in the HOME position on the screen. Now change the color of the letter "A" in the HOME position with the following POKE:

```
POKE 55296,1 RETURN
```

Did you notice the difference? The letter "A" in the HOME position changed from light blue to white. At this point you may wonder what the "1" means in POKE 55296,1. This time the "1" is not a screen code that represents a character. Instead it is the color code for white. Refer to the Color Registers section for the list of Commodore 64C colors and the corresponding color codes.

Remember, if you want to POKE a character to the screen, you actually need two POKES. First, POKE a screen code into screen memory to display a character. Second, POKE a color code into color memory to display the color of the character.

## Animation

The Commodore 64C is capable of animating objects on the screen. The idea behind computer animation is to display an image on the screen and simulate its motion through computer instructions.

Remember when you POKEd a character into screen memory and it was displayed on the screen? That's what you are going to do to animate a graphic character. To animate a graphic character on the screen, POKE its screen code into a screen memory location. Next, POKE the screen code for a blank (32) into the same screen location. Then POKE the graphic character screen code into a screen location next to the original one. Repeat the process with a series of adjacent screen memory locations. Since the computer is displaying and blanking out the graphic character in successive screen locations so quickly, the image appears to be moving. For example, type in the following program and RUN it.

```

10 PRINT"█"
20 FOR I=1024 TO 2023 STEP41
30 POKEI,81
35 POKE54272+I,7
40 FOR J=1TO45:NEXT
45 POKEI,32
50 NEXT
100 FOR I=2009TO1450 STEP-39
110 POKEI,81
120 POKE54272+I,7
130 FOR J=1TO45 :NEXT
140 POKEI,32
150 NEXT
160 GOTO20

```

This is your first taste of animation. You have just made a yellow ball bounce on the screen. Although the bouncing ball program is a simple example of animation, you are now on your way to programming sophisticated, animated graphics.

Here's an explanation of the program:

- Line 10 clears the screen. Loop 1, lines 10 through 50, displays and moves the ball from the top of the screen to the bottom. Line 20 begins a loop at the start of screen memory. Notice the FOR . . . NEXT statement has the words STEP 41. This tells the computer to increment the index variable I, by 41 locations at a time, starting at location 1024 and ending at location 2023. When STEP is not specified in a FOR . . . NEXT loop, your computer cycles through each index variable one at a time.
- Line 30 POKES screen code value 81 into the screen location according to the index variable I. The value 81 represents the screen code for the ball character that bounces on the screen. The first cycle of the loop POKES screen code 81 into location 1024. The second cycle POKES screen code 81 into screen location 1065 (1024 + 41). The third cycle POKES screen code 81 into screen location 1106 (1065 + 41) and so on. Each cycle through the loop skips 40 screen locations and POKES the ball 41 locations past the previous screen location.



- Line 35 POKES color code 7 (yellow) into the color memory location corresponding to the screen location that is POKED with the ball character. Remember, when you POKE a screen code value into screen memory, you also have to POKE a color code value into the corresponding color memory location. See Figure 7-1 and 7-2 to understand how each screen memory location corresponds to its own color memory location.
- In line 35, location  $54272 + I$  is the beginning of color memory during the first cycle of the loop ( $54272 + 1024 = 55296$ ). The loop increments the color memory location the same way as screen memory. The second cycle of the loop increments the index variable I, so the POKE statement in line 35 POKES the color code value into location 55337 ( $55296 + 41$ ). Color location 55337 corresponds to screen location 1065. As you can see, the loop takes care of POKING the screen location and corresponding color location so that the ball is always displayed correctly in yellow.
- Line 40 is an empty FOR . . . NEXT loop. It acts as a time delay to slow down the program so the animation appears smooth. Try the program without line 40. You'll notice the program becomes choppy.
- Line 45 POKES screen code value 32, the blank character, into the same screen location that was POKED with screen code 81 in line 30. This turns off the ball character. The ball character is turned on and off so quickly, it looks as though the ball is always on the screen.
- Line 50 is a NEXT statement. It updates the index variable I. The loop then cycles until the index variable equals 2023. At that point the program executes loop 2.
- Loop 2 bounces the ball upward and off the right side of the screen. Loops 1 and 2 both have the same statements, except different screen memory locations are decremented in line 100 instead of incremented as in line 20. The GOTO statement in line 60 tells the computer to go back to line 20 and execute everything again. The GOTO statement gives you a way to RUN your programs continuously. Stop the program by pressing the RUN/STOP key.

Here's another animation program that bounces the yellow ball off all four "walls" of the screen. This program is based on program three, but it has five loops instead of three. Each of the five loops is just like the two loops in the preceding program, except that the last three loops use different screen locations to control the three additional bounces of the ball.

```
10 PRINT"□"  
20 FOR I=1024 TO 2023 STEP41  
30 POKEI,81  
35 POKE54272+I,7  
40 FOR J=1TO45:NEXT  
45 POKEI,32  
50 NEXT  
100 FOR I=2009TO1450 STEP-39  
110 POKEI,81  
120 POKE54272+I,7  
130 FOR J=1TO45 :NEXT  
140 POKEI,32  
150 NEXT  
200 FOR I=1423TO1044 STEP-41  
210 POKEI,81  
220 POKE54272+I,7  
230 FOR J=1TO45 :NEXT  
240 POKEI,32  
250 NEXT  
300 FOR I=1050TO1554 STEP38  
310 POKEI,81  
320 POKE54272+I,7  
330 FOR J=1TO45 :NEXT  
340 POKEI,32  
350 NEXT  
400 FOR I=1544TO2009 STEP42  
410 POKEI,81  
420 POKE54272+I,7  
430 FOR J=1TO45 :NEXT  
440 POKEI,32  
450 NEXT  
490 GOTO100
```

Now that you can animate a simple graphic character, it's time to learn a much more sophisticated method called sprite animation.

### Sprite Concepts

You've learned how to control color with the CTRL key, with PRINT statements, and with character string codes. You now know how to PRINT alphanumeric and graphic characters on the screen within quotes, as character strings, and by POKEing values directly into screen memory. Animating existing character images, as described in the last section, has certain limitations. For true graphic animation, you need a way to create your own images, color those images and control their movement on the screen. That's where sprites come in.

Sprites are programmable movable objects. They are animated, high resolution images you can create into any shape. You can move these images anywhere on the screen and color them in 16 colors. The Commodore 64C has a set of sprite registers that control the color, movement and shape of the sprite. Sprites on the 64C provide you with true animation and sophisticated graphics capabilities.

A special chip inside the 64C, called the VIC (Video Interface Controller) chip, controls graphics modes and sprites. Border and screen color registers as well as the sprite registers are all part of the VIC chip. The VIC chip normally can control 8 sprites at once. Through advanced programming you can control more than eight sprites. The VIC chip can even determine if a sprite has moved in front of or behind another sprite. The size of each sprite can also be expanded both vertically and horizontally. You can use sprites in any mode: standard character, multi-color, standard and multi-color bit map and extended color modes. See the discussion of Graphics Modes later in this section for more information.

Let's begin by examining the properties of characters first, and then relate them to sprites. A character on the screen is an 8 by 8 dot grid. Since there are 40 columns by 25 lines on the screen, the entire screen has 320 (40 x 8 dots per character width) dots across times 200 (25 lines x 8 dots per character height) tall, which equals 64,000 total dots.

Each character pattern requires 8 bytes of storage in character memory. Each of the eight rows of dots in the 8 by 8 character grid require a byte of memory storage. In other words, each screen dot requires a bit of memory, so an 8 by 8 dot grid consists of 64 square dots and requires 64 bits (8 bytes) of memory.

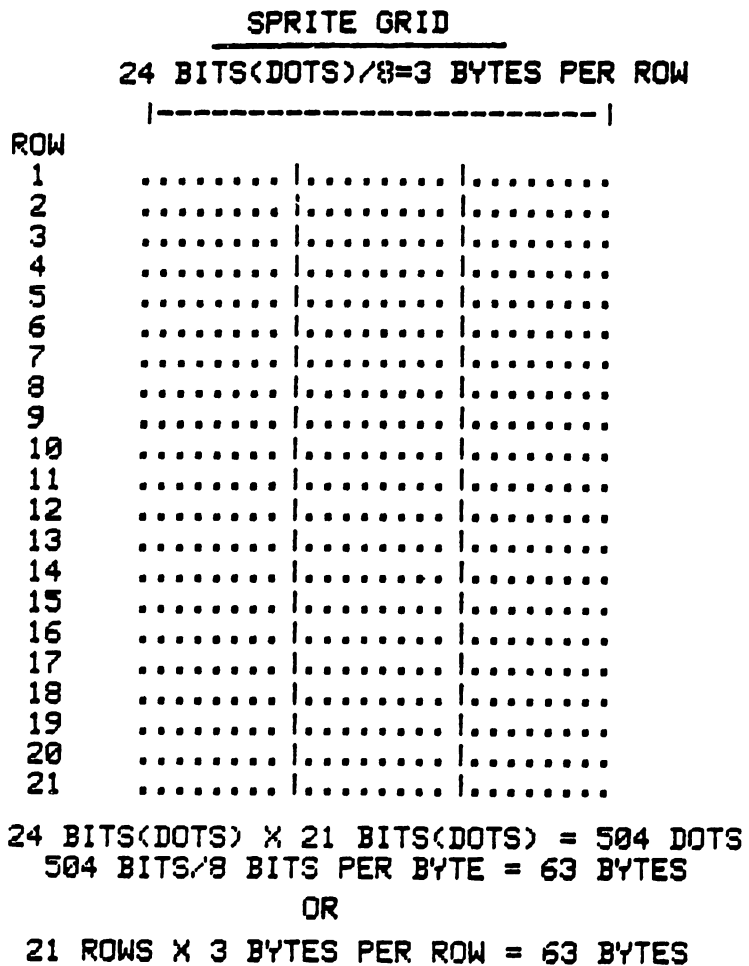
Each dot on the screen is called a *pixel*. *Pixel* is a computer term for picture element. A sprite is made up of a 24 by 21 pixel grid, compared to a character which is an 8 by 8 pixel grid. The width of a sprite is 24 pixels, which is equal to the width of three screen characters (bytes). Since a sprite is 21

rows of three bytes wide, a sprite requires 63 bytes (21 rows x 3 bytes per row) of storage. Figure 4-3 illustrates the layout and storage requirements of a sprite.

### Designing a Sprite Image

The first step in programming a sprite is designing the sprite image. For a beginner, the best way to design a sprite is on a piece of graph paper. Draw a box 24 blocks across by 21 blocks tall, just like Figure 4-3. The box you have just drawn is 504 (21 x 24) square blocks. Each block represents a bit in memory. If you divide 504 by 8 bits per byte, you'll see that the sprite uses up 63 bytes of memory.

FIGURE 4-3. SPRITE GRID

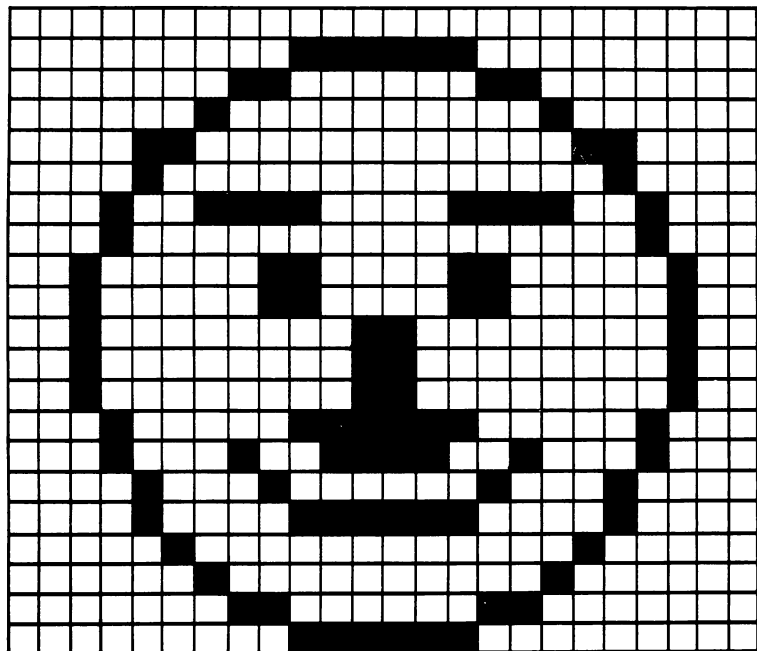


You can now start designing your sprite image. Keep in mind that each block within the box you have drawn represents one bit in the Commodore 64C's memory. As you probably know by now, a bit can take on one of two values, zero or one. That is why a bit is called a binary digit, since the root "bi" means two. A zero (0) means that a bit is "off" and a one (1) means that a bit is turned "on".

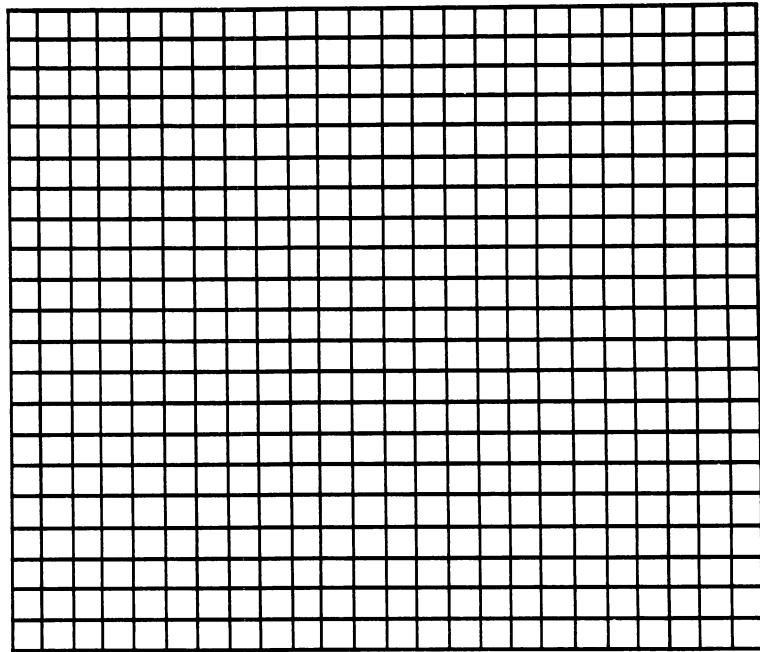
When you are designing your sprite on a piece of graph paper, darken a block if you want that bit to be on, and leave a block blank if you want that bit off. The combination of darkened blocks and blank blocks forms your sprite image. That is, if you want to turn on a dot in the sprite image, you must turn on a corresponding bit in memory where the sprite DATA is stored.

Refer to Figure 4-4 as an example of designing a sprite on a piece of graph paper. Remember, the darkened blocks are "on" bits and the blank blocks are "off" bits. The sprite image in Figure 4-4 represents a smiling face. Use the blank sprite-making grid in Figure 4-5 to create your own sprite images.

**FIGURE 4-4. SAMPLE SPRITE**



**FIGURE 4-5. SPRITE-MAKING GRID**



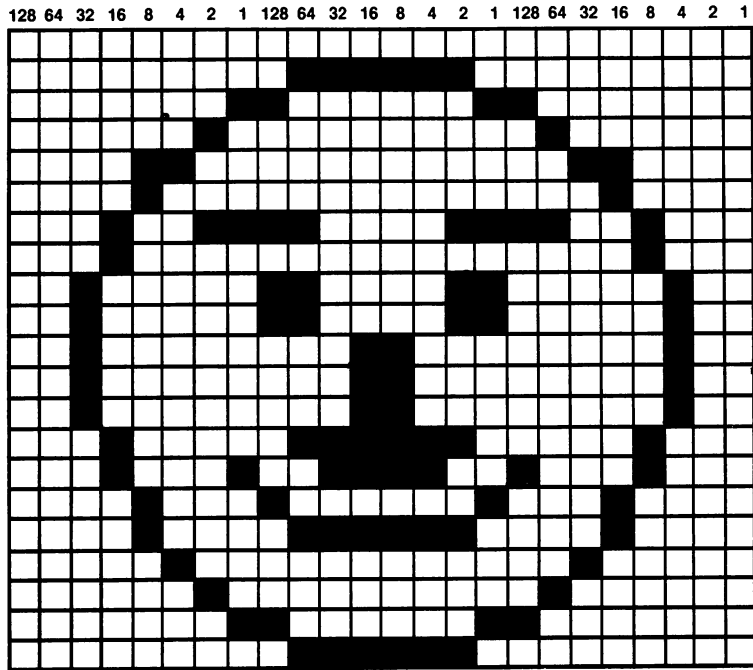
**Converting Your Sprite Image Into Data**

The next step in programming a sprite is coding the sprite image into data the computer can understand. On your sheet of graph paper, label the top of each column the same as in Figure 4-6.

Label the first eight columns as follows: 128, 64, 32, 16, 8, 4, 2, 1. Label the second and third set of eight columns the same way.

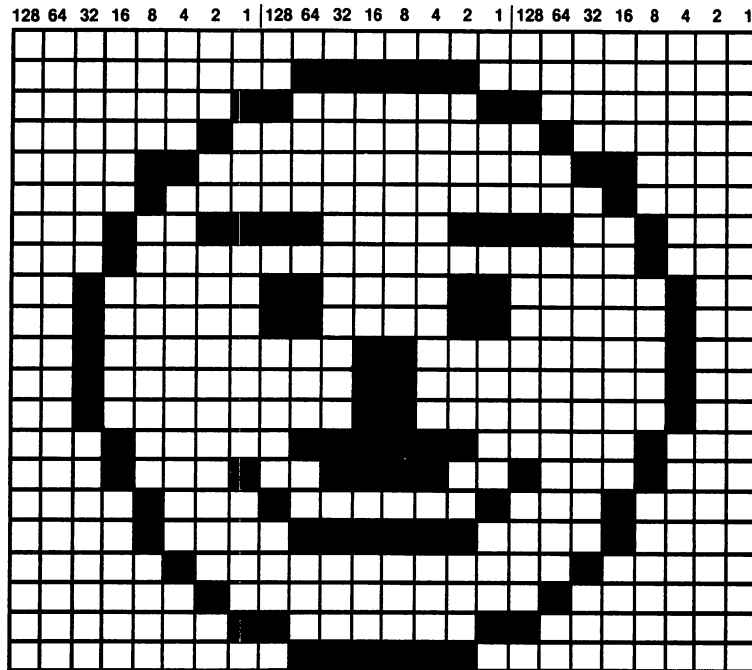
You now have three sub-sets (bytes) of eight columns (bits) per row, each labeled from 128 on the left to 1 on the right. Each 8 column sub-set represents 8 pixels that correspond to a byte of memory. Again, since there are 21 rows with three bytes each, the total amount of memory the sprite requires is 63 bytes.

FIGURE 4-6. LABELING THE SPRITE-MAKING GRID.



Now you have a way to convert the graph paper image to computer data. For each darkened square within an eight column sub-set (byte) add up the number at the top of the column. Do this for each of the three 8 column sub-sets per row or a total of 63 times. Do not add column values in which individual squares are blank since these represent "off" pixels. Only add up the column values for the darkened squares. Once you calculate all the byte values for each eight column sub-set, you have 63 pieces of data to define your sprite. These values must be READ by the 64C and stored in DATA statements within a program. Study Figure 4-7 to grasp the concept of converting a sprite picture on graph paper to data used by the 64C.

FIGURE 4-7. SPRITE-MAKING GRID WITH DATA VALUES



```

100 DATA 0,0,0
110 DATA 0,126,0
120 DATA 1,129,128
130 DATA 2,0,64
140 DATA 12,0,48
150 DATA 8,0,16
160 DATA 19,197,200
170 DATA 16,0,8
180 DATA 32,195,4
190 DATA 32,195,4
200 DATA 32,24,4
210 DATA 32,24,4
220 DATA 32,24,4
230 DATA 16,126,8
240 DATA 17,60,136
250 DATA 8,129,16
260 DATA 8,126,16
270 DATA 4,0,32
280 DATA 2,0,64
290 DATA 1,129,128
300 DATA 0,126,0
    
```

In the program shown in Figure 4-7, the DATA values in line 100 correspond to the three sub-sets of the first row of the sprite grid. All three pieces of DATA equal zero since all three sub-sets of the first row of the sprite grid are blank (off). Line 110 corresponds to the second row of the sprite grid. The first DATA value in line 110 equals zero, because again, no pixels are turned on in that sub-set. The second piece of DATA in line 110 equals 126, since the squares in the column positions labeled 64, 32, 16, 8, 4 and 2 in the middle sub-set are all turned on.



Again the third DATA value in line 110 is zero because none of the pixels in that 8 column sub-set is turned on. The DATA in line 120 represents the pixel values for the third row of the sprite grid, line 130 represents the values in the fourth row of the sprite grid, and so on. Line 300 corresponds to the last row of the sprite grid.

Now that you know how to design a sprite on a sheet of graph paper and code it into DATA that the Commodore 64C can understand, you are almost ready to write your first sprite program. But first let's examine the sprite registers and how they work.

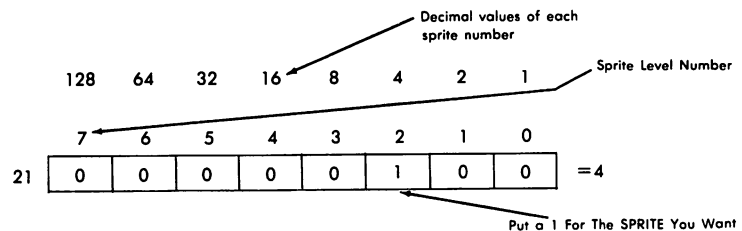
### Controlling Sprites

Special memory locations within the video chip, known as sprite registers, are set aside to control sprites. Each sprite register is assigned a specific task. The first register you need to set is the sprite enable register at location 53269. As the name implies, the sprite enable register turns on a sprite. You must POKE a value into the sprite enable register, depending on which sprite(s) you want to turn on. Here's a list of the POKE values that enable each sprite:

Sprite No.	POKE Value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

You may have noticed the POKE value for each sprite is equal to two, raised to the sprite number. For example, the POKE value for sprite seven is two raised to the seventh power, which equals 128. Figure 4-8 illustrates this concept.

**FIGURE 4-8. SPRITE POKE VALUES**



The POKE command to turn on sprite 7 is:

**POKE 53269,128**

If you want to enable more than one sprite, add the POKE values of the sprites you want to turn on, and POKE the sum into the sprite enable register.

Now you have to store the sprite DATA somewhere in the Commodore 64C's memory. Although you already converted your sprite picture into DATA as in lines 100 through 300 in Figure 4-7, you still have to READ that DATA and POKE it into memory. Before you can do that, you must tell the 64C where to store the DATA.

You point out where the DATA is stored using a sprite pointer. Each of the eight sprites has its own pointer. The following is a list of the sprite pointer memory locations:

Sprite No.	Memory Location
0	2040
1	2041
2	2042
3	2043
4	2044
5	2045
6	2046
7	2047

Now that you know what location to POKE for each sprite pointer, you need to know the value to POKE into these locations. Here's the formula:

1. Choose an available memory location that is not being used. For this example, choose location 12288.
2. Divide the chosen location by 64:  $12288/64 = 192$
3. POKE the sprite pointer of the sprite you previously enabled with the quotient from step 2. To continue our previous example, the following POKE command uses the seventh sprite pointer to point to sprite DATA starting at location 12288:

**POKE 2047, 192 ~~RETURN~~**

To determine other locations to store sprite DATA, consult the Commodore 64 Programmer's Reference Guide.

As mentioned before, the sprite DATA must be READ and then POKEd into memory once the sprite pointers tell the 64C where to store the DATA. The sprite pointer was set with the previous POKE command. Now you can READ the sprite DATA you converted from your sprite image and POKE it into memory starting at location 12288. POKEing the DATA into memory actually creates the sprite. The following program segment READs the DATA and POKEs it into memory starting at location 12288.

```
50 FOR N = 0 to 62
60 READ Q
70 POKE 12288 + N,Q
80 NEXT
```

So far you have enabled the sprite, set the sprite pointer to tell the 64C where to store the sprite DATA and POKEd the sprite into memory. All you need to do now is to assign a sprite color and control the sprite's movement on the screen, and your sprite program will be finished.

Each sprite has its own sprite color register. The following is a list of sprite color register locations:

Sprite No.	Memory Location
0	53287
1	53288
2	53289
3	53290
4	53291
5	53292
6	53293
7	53294

To assign a sprite color, POKE a sprite color register with a color code between 0 and 15. For example, if you enter:

```
POKE 53294,7 RETURN
```

sprite 7 is colored yellow. (For a list of color codes, see the Color Registers discussion given earlier in this section.)

## Animating Your Sprites

Animation is the last step before your program can RUN. The key behind animation is motion. Each of the eight sprites has two registers that control movement on the screen. One register is the sprite X position, which controls the horizontal sprite movement. The other is the sprite Y position, which controls the sprite's vertical movement. The following is a list of the sprite X and Y position registers for each sprite:

Sprite No.	Memory Location
0 - X pos	53248
0 - Y pos	53249
1 - X pos	53250
1 - Y pos	53251
2 - X pos	53252
2 - Y pos	53253
3 - X pos	53254
3 - Y pos	53255
4 - X pos	53256
4 - Y pos	53257
5 - X pos	53258
5 - Y pos	53259
6 - X pos	53260
6 - Y pos	53261
7 - X pos	53262
7 - Y pos	53263

The easiest way to control the vertical and horizontal coordinate values is within a FOR . . . NEXT loop. Set up a loop and POKE the index variable from the loop into the vertical and horizontal sprite position registers. For example, to move sprite 7 diagonally on the screen, use the following program segment:

```
85 FOR Z = 0 TO 200: REM Set up loop; index variable = z
90 POKE 53262,Z : REM Poke sprite 7 x pos. with index variable z
95 POKE 53263,Z : REM Poke sprite 7 y pos. with index variable z
98 NEXT : REM Update index variable position
```

Notice that the FOR . . . NEXT loop moves sprite 7 the maximum number of vertical values (200), but only moves horizontally 200 out of the 320 possible positions. That was done to keep the example program simple.

The sprite Y position register can store any of the 200 possible vertical position values. The sprite X position register cannot store all of the 320 horizontal position values because the sprite position register, like all other memory locations in the Commodore 64C, can only represent a value up to 255.

How do you position a sprite past the 255th horizontal screen position? The answer is, you have to borrow a bit from another register in order to represent a value greater than 255.

An extra bit is already set aside in the 64C's memory in case you want to move a sprite past the 255th horizontal location. Location 53264 controls sprite movement past position 255. Each of the 8 bits in 53264 controls a sprite. Bit 0 controls sprite 0, bit 1 controls sprite 1 and so on. For example, if bit 7 is on, sprite 7 can move past the 255th horizontal position.

Each time you want a sprite to move across the entire screen, turn on the borrowed bit in location 53264 when the sprite reaches horizontal position 255. Once the sprite moves off the right edge of the screen, turn off the borrowed bit so the sprite can move back onto the left edge of the screen. The following POKE command allows sprite seven to move past the 255th horizontal position:

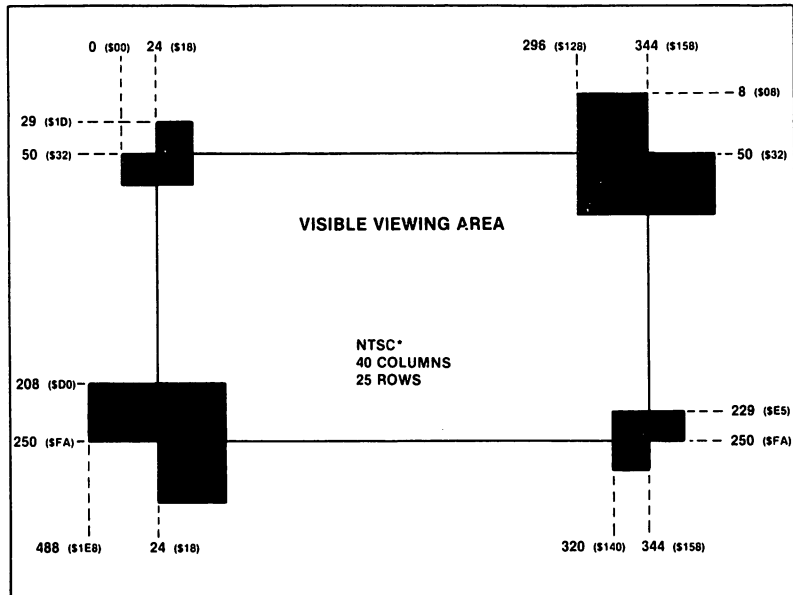
**POKE 53264,128 RETURN**

The number 128 is the resulting value from turning on bit 7. You arrive at this value by raising two to the seventh power. If you want to enable bit 5, raise two to the fifth power, which of course equals 32. The general rule is to raise two to the power of the sprite number that you want to move past the 255th horizontal screen position. Now you can borrow the extra bit you need to move a sprite all the way across the screen. To allow the sprite to reappear on the left side of the screen, turn off bit seven again, as follows:

**POKE 53264,0 RETURN**

Not all of the horizontal (X) and vertical (Y) positions are visible on the screen. Only vertical positions 50 through 249 and horizontal positions 24 through 342 are visible. In the example, when you moved sprite 7 on the screen, you started the sprite moving at horizontal location zero and vertical position zero. Location 0,0 is off the screen as is any horizontal location less than 24 and greater than 343. Any vertical location less than 50 and greater than 249 is also off the screen. The OFF-SCREEN locations are set aside so that an animated image can move smoothly onto and off of the screen. Study Figure 4-9 to understand the layout of the visible horizontal and vertical sprite positions.

FIGURE 4-9. VISIBLE SPRITE POSITIONS



\*North American television transmission standards for your home TV

### Tying Your Sprite Program Together

Now you are ready to tie all the sprite concepts together into a sprite program. Let's review the entire procedure. In order to program a sprite, you must:

1. Create the sprite image on a sheet of graph paper.
2. Convert the sprite image into DATA values the Commodore 64C can understand.
3. Enable the sprite.
4. Use a pointer to tell the Commodore 64C where to store the sprite DATA.
5. READ the sprite DATA and POKE it into memory, starting at the location indicated by the sprite pointer.
6. Color the sprite.
7. Control the sprite's movement on the screen.

The following program combines all the concepts, statements and program segments covered so far in this section. Type in the program, and press **RETURN** after each line. Once you've typed in the complete program, type RUN and press **RETURN**. You'll see a smiling face moving diagonally across the screen.

```

10 PRINT "☐"
20 POKE53269,128
30 POKE2047,192
50 FORN=0TO62
60 READ Q
70 POKE12288+N,Q
80 NEXT
85 FOR Z=1TO200
90 POKE53262,Z
95 POKE53263,Z
98 NEXT
100 DATA 0,0,0
110 DATA 0,126,0
120 DATA 1,129,128
130 DATA 2,0,64
140 DATA 12,0,48
150 DATA 8,0,16
160 DATA 19,197,200
170 DATA 16,0,8
180 DATA 32,195,4
190 DATA 32,195,4
200 DATA 32,24,4
210 DATA 32,24,4
220 DATA 32,24,4
230 DATA 16,126,8
240 DATA 17,60,136
250 DATA 8,129,16
260 DATA 8,126,16
270 DATA 4,0,32
280 DATA 2,0,64
290 DATA 1,129,128
300 DATA 0,126,0

```

Now add the following lines and RUN the program again.

```

55 POKE 53271,128
57 POKE 53277,128

```

Notice that the sprite now appears twice its original size. Location 53277 controls horizontal expansion and location 53271 controls vertical expansion of the sprite. The value POKEd into these locations is calculated according to which sprite you want to expand. The general rule is raise two to the power of the sprite number. For example, to expand sprite 7, the value 128 in lines 55 and 57 is calculated as two raised to the seventh power, or 128.

You have successfully written your first sprite program. Use this program as a basis and try adding other sprites to it. Notice lines 100 through 300 only contain three pieces of DATA each. The program is written this way to illustrate the correspondence between each DATA item and each eight column byte in Figure 4-7. When you become more familiar with sprite concepts you can shorten the program by including more DATA items in each DATA statement. Lines 100 through 300 are still stored as 80 character lines. The spaces are stored in memory just as visible characters, but they use memory needlessly. The process of shortening programs is called crunching. Later, when you become a more advanced programmer, you will realize the value of crunching your programs and using the Commodore 64C's memory more efficiently.

Change line 20 of the program as follows:

```
20 POKE 53269,224 : REM Enable sprites 7, 6 and 5
```

Add the following lines to the program and RUN it again. The REM statements are optional. You don't have to type them in. They document the program so you can follow each program step.

```
15 POKE 53280,1 :REM Change the border color to white
17 POKE 53281,1 :REM Change the background color to white
35 POKE 2046,192 :REM Set sprite 6 data pointer to 12288
37 POKE 2045,192 :REM Set sprite 5 data pointer to 12288
43 POKE 53293,6 :REM Color sprite 6 blue (6)
45 POKE 53292,2 :REM Color sprite 5 red (2)
92 POKE 53260,Z :REM Set sprite 6 horizontal (X) position
94 POKE 53258,100 :REM Set sprite 5 horizontal (X) position
96 POKE 53261,100 :REM Set sprite 6 vertical (Y) position
97 POKE 53259,Z :REM Set sprite 5 vertical (Y) position
99 GOTO 85 :REM Put the program into a continuous loop
```

Two more sprites appear on the screen, one from the left side of the screen and one from the top. Notice in the program, both sprites 5 and 6 use the same sprite DATA as sprite 7. That's why all three sprites look the same. If you want to change the way a sprite looks, design another sprite image on a piece of graph paper just as you did before. Then add another complete set of sprite DATA as in lines 100 through 300. In addition, READ the DATA and POKE it into a section of memory other than locations 12288 through 12351, since the other sprite DATA is already there. Finally, set the sprite DATA pointer to the starting location where the sprite DATA is POKED into memory.

All three sprites in the above program store their DATA starting at location 12288. That's why lines 30, 35 and 37 POKE the same value into each



## Graphics Modes

of the three respective sprite DATA pointers. If all eight sprites were enabled, each one could use the same DATA and you would have eight identical sprites on the screen.

Lines 43 and 45 color sprite 6 blue and sprite 5 red. Lines 92 through 97 control the movement of sprites 5 and 6. Line 99 puts the program into a continuous loop. If you want to stop it, press the RUN/STOP key. Notice the sprite remains on the screen. To clear the screen completely, hold down the RUN/STOP key and press the RESTORE key.

Up to now, you've programmed three sprites on the screen. Try using all eight. In a relatively short time you should be able to create your own sprites in several colors and animate them on the screen. You can then move on to explore the very sophisticated color, graphics and animation features available on the 64C. Consult the Commodore 64 Programmer's Reference Guide for more information on color graphics, sprites and animation.

The Commodore 64C can operate in five different graphics modes. They are divided into two groups known as character display modes and bit map modes. Character display modes, as the name implies, display an entire 8 x 8 dot character grid at a time. In character display modes, the smallest unit of information you can display is an 8 x 8 pixel grid which equals one character. Bit map modes allow you to display each pixel, one at a time. Bit map mode gives you absolute control over the screen image. Graphics performed in bit map mode are referred to as high resolution graphics.

Both groups of graphics modes can be divided into separate subdivisions. Character display modes are separated into these three subdivisions:

1. Standard Character Mode
2. Multi-Color Character Mode
3. Extended Background Color Mode

Bit map modes are separated into these two subdivisions:

1. Standard Bit Map Mode
2. Multi-Color Bit Map Mode

Each of the character display modes get character information from one of two places in the 64C's memory. Normally, character information is taken from character memory stored in a separate chip called a ROM (Read Only Memory). However, the 64C gives you the option of designing your own

characters and replacing the original Commodore 64C characters with your own. Your own programmable characters are stored in a portion of the 64K of RAM (Random Access Memory) available to you in the 64C.

When you first turn on the 64C, you are automatically in standard character mode. When you write programs, the 64C is also in standard character mode. Standard character mode displays characters in one of 16 colors on a background of one of 16 colors. All the information contained in this chapter operates in standard character mode except sprites. Sprites are classified separately from character display modes and bit map modes.

Multi-color character mode gives you more control over color than the standard graphics modes. Each screen dot within an 8 x 8 character grid can have one of four colors, compared to the standard modes which can only have one of two colors. Multi-color mode uses two additional background color registers. The three background color registers and the character color register together give you a choice of four colors for each dot within an 8 x 8 dot character grid.

Multi-color mode has one disadvantage. Each screen dot in multi-color mode is twice as wide as a dot in standard character mode and standard bit map mode. As a result, multi-color mode has only half the horizontal resolution (160 x 200) of the standard graphics modes. However, the increased control of color more than compensates for the loss in horizontal resolution.

Extended background color mode allows you to control the background color and foreground color of each character. Extended background color mode uses all four background color registers. In extended color mode, however, you can only use the first 64 characters of the screen code character set. The second set of 64 characters is the same as the first, but they are displayed in the color assigned to background color register 2. The same holds true for the third set of 64 characters and background color register 3, and the fourth set of 64 characters and background color register 4. The character color is controlled by color memory. For example, in extended color mode, you can display a purple character with a yellow background on a black screen.

Standard bit map mode allows you to control each screen dot in one of two colors. This gives you the ability to create detailed graphic images on the screen. Bit mapping is a technique that stores a bit in memory for each dot on the screen. If the bit in memory is turned off, the corresponding dot on the screen becomes the color of the background. If the bit in memory is turned on, the corresponding dot on the screen becomes the color of the foreground image. The series of 64,000 dots on the screen and 64,000 cor-

responding bits in memory control the image you see on the screen. Most of the finely detailed computer graphics you see in demonstrations and video games are bit mapped high resolution graphics.

Multi-color bit map mode is a combination of standard bit map mode and multi-color character mode. You can display each screen dot in one of four colors within an 8 x 8 character grid. Again, as in multi-color character mode, there is a tradeoff between the horizontal resolution and color control.

\*\*\*\*\*

*This chapter has described a variety of color and graphics techniques based on advanced programming concepts. The full explanation of these concepts is beyond the scope of this Guide. If you want more details on graphics techniques and graphics programming, refer to the Commodore 64 Programmer's Reference Guide.*

*The next chapter completes your introduction to the Commodore 64C computer by outlining the 64C's varied sound and music capabilities.*



CHAPTER

Sound  
and  
Music

5



**CHAPTER 5**  
**Sound and Music**

<b>THE SID MICROPROCESSOR</b>	<b>99</b>
<b>MUSIC</b>	<b>99</b>
Playing From Sheet Music	99
Obtaining the Data	100
Writing the Program	101
<b>SOUND EFFECTS</b>	<b>104</b>
Program Notes	106





## The SID Microprocessor

### Music

A special microprocessor known as the SID (Sound Interface Device) provides the 64C with extraordinary capabilities in generating musical tones and sound effects. This chapter introduces you to these capabilities. For more details, see Appendix J of this book and consult the Commodore 64 Programmer's Reference Guide.

The Commodore 64C is capable of producing musical tones over a large range—a full six octaves for up to three separate voices (musical instruments) simultaneously. You can teach your 64C to play anything from *Happy Birthday* to Beethoven's Fifth Symphony.

By controlling a series of internal registers in the SID, you can program your 64C to play a variety of complex musical sounds. These sounds or notes have the qualities of a particular musical instrument and vary in pitch and duration.

### Playing From Sheet Music

In a musical score sheet you will find notes indicated by position and appearance. Compare these with Figure 5-1 for the note name and Figure 5-2 for note duration.

FIGURE 5-1. NOTE NAMES

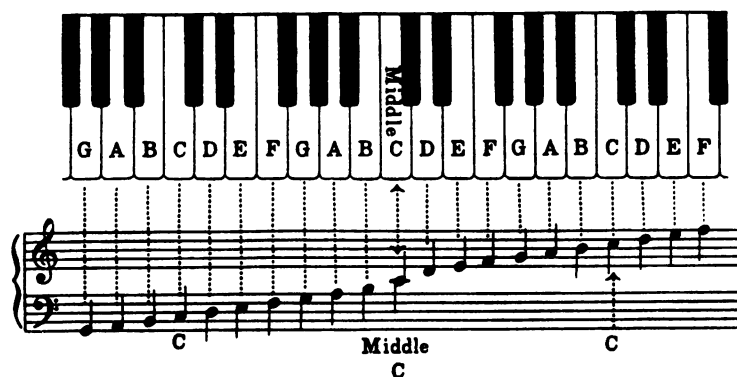


FIGURE 5-2. NOTE DURATION



To create these notes through the speakers of your monitor or TV, you must turn ON several registers in the SID microprocessor. There are seven registers for each of three voices. Each must be filled with a particular value. See Table 5-1 for the values of registers 2 through 6. Registers 0 and 1 are for sound frequency and are adjusted later in the program.

**Table 5-1. Sound Register Values**

Register number	2	3	4-ON	4-OFF	5	6
Musical instruments:						
Piano	225	0	65	64	9	0
Flute	0	0	17	16	96	0
Harpsichord	0	0	33	32	9	0
Xylophone	0	0	17	16	0	240
Accordian	0	0	17	16	102	0
Trumpet	0	0	33	32	96	0
Noise	0	0	129	128	-	-

### Obtaining the Data

To insert a musical score into your computer, follow each step in this example, which incorporates the music of the song "Tom Dooley":

**CHORUS:**

Hang down your head, Tom Doo - ley,

Hang down your head and cry. —

Hang down your head, Tom Doo - ley,

Poor boy, you're bound to die. —

1. Select the musical instrument and determine the register values from Table 5-1.  
Piano: register 2 is 255, register 3 is 0; register 4 is 65 for ON and 64 for OFF; register 5 is 9 and register 6 is 0.
2. Determine the name and value of each note; use Figures 5-1 and 5-2 as guides. Tabulate the results.  
Notes read: D (eighth), D (quarter), D (eighth), E (quarter), G (quarter), B (half), B (half), etc.
3. Convert each note into the proper register settings called N1 and N2 from the Note Table in Appendix J and the duration (DR), based upon the following note values:

Eighth note = 250  
 Quarter note = 500  
 Half note = 1000  
 Whole note = 2000  
 A note with a dot = DR \* 1.5

#### Tabulated Data

Note	Value	N1	N2	DR
D	1/8	18	104	250
D	1/4	18	104	500
D	1/8	18	104	250
E	1/4	20	169	500
G	1/4	24	146	500
B	1/2	30	245	1000
B	1/2	30	245	1000
etc.				

4. Write the program.

**NOTE:** Registers 2, 3, 4, 5 and 6 are set based on the musical instrument selected. Registers 0 and 1 are based upon each note and will vary. There is a register 24. It is the volume for all instruments and is always set to 15. The volume from your speaker is controlled by the TV or monitor volume control.

### Writing the Program

Playing music requires turning on the appropriate registers, reading the notes and turning the sound on and off. All the registers can be turned on early in the program except register 4, which is turned on only when the music is needed.

Selecting a register is done by the BASIC term POKE, followed by the register number plus 54272, a comma and the proper value.

1. Set all the registers to zero:  
10 S = 54272:FOR SW = S to S + 24: POKE SW,O:NEXT SW
2. Set the volume to the maximum of 15:  
20 POKE S + 24,15
3. Turn on registers 2, 3, 5 and 6, based upon the instrument you are using (in this case, the piano):  
30 POKE S + 2,255  
40 POKE S + 3,0  
50 POKE S + 5,9  
60 POKE S + 6,0
4. POKE a note into registers 1 and 0 from the table developed above. Since it will vary, represent the value with variable names N1 and N2.  
80 POKE S + 1,N1:POKE S,N2
5. Activate the sound with register 4, using the value for the proper instrument (65 for piano):  
90 POKE S + 4,65
6. Keep the sound on for the required time based on the value of DR in your table. Since this value is a variable, it is represented by its variable name, DR:  
100 FOR Z = 1 to DR: NEXT Z
7. Turn off the sound, using the proper value:  
110 POKE S + 4,64
8. Keep the sound off for a very short time—about a tenth of a second.  
120 FOR T = 1 to 50: NEXT T
9. Continue steps 4 through 8 with successive notes by using a READ statement and a loop.  
70 READ N1, N2, DR  
125 GOTO 70
10. Store the note information in DATA statements. For simplicity, each DATA statement below represents one note:

```
130 DATA 18,104,250
132 DATA 18,104,500
134 DATA 18,104,250
136 DATA 20,169,500
138 DATA 24,146,500
140 DATA 30,245,1000
142 DATA 30,245,1000
etc.
```

11. Include a means to stop the program:

```
75 IF N1 = 0 THEN END
200 DATA 0,0,0
```

Your sample program, when completed from sheet music, will look like this:

```
5 REM CHORUS FROM TOM DOOLEY
10 S = 54272:FOR SW = S TO S + 24:POKE SW,0:NEXT
20 POKE S + 24,15
30 POKE S + 2,255
40 POKE S + 3,0
50 POKE S + 5,9
60 POKE S + 6,0
70 READ N1,N2,DR
75 IF N1 = 0 THEN END
80 POKE S + 1,N1:POKE S,N2
90 POKE S + 4,65
100 FOR Z = 1 TO DR:NEXT Z
110 POKE S + 4,64
120 FOR T = 1 TO 50:NEXT T
125 GOTO 70
130 DATA 18,104,250,18,104,500,18,104,250,20,169,500,24,146,
500
140 DATA 30,245,1000,30,245,1000
150 DATA 18,104,250,18,104,500,18,104,250,20,169,500,24,146,
500
160 DATA 27,148,2000
170 DATA 18,104,250,18,104,500,18,104,250,20,169,500,24,146,
500
180 DATA 27,148,1000,27,148,1000
190 DATA 27,148,250,27,148,500,30,245,250,24,146,500,20,169,
500,24,146,1500
200 DATA 0,0,0
```

Be sure to raise the volume on your monitor when you run your program. To double the tempo, change line 100:

```
100 FOR T = 1 TO DR/2: NEXT T
```

To play a different song, change DATA statements to the appropriate values.

Now that you have created your first song, experiment with other instruments by varying the register values. You can also combine several voices to represent chords or other instruments by adding 7 or 14 to each of the register numbers (except register 24). Thus, registers 7 through 13 can control the second voice, and registers 14 through 20 the third voice.

## Sound Effects

Besides music, you can also create special sound effects by using the noise registers and varying the sound characteristics known as ADSR (Attack, Decay, Sustain and Release). These are combined in registers 5 and 6. A thorough explanation is provided in the *Commodore 64 Programmer's Reference Guide*. Below are register values for sample sound effects.

Sound Effects Register Values										
Registers	0	1	2	3	4-ON	4-OFF	5	6	*	24
Variable										
Names	N2	N1	P2	P1	W1	W2	AD	SR	DR	V
Sound effects:										
Police Siren	85	36	0	0	33	32	136	129	350	15
Crash	251	5	0	0	129	128	129	65	50	0
Rocket Blast-off	100	25	0	0	129	128	9	129	50	0
Machine Gun	75	34	0	0	129	128	8	1	50	15
Wailing	N2	40	0	0	65	64	15	0	1	15
Shooting	200	40	0	0	129	128	15	15	1	0

\*Not a register. Part of the timing loop.

The following program, called "Sound Effects", incorporates all these variables and can produce each of these sounds. The technique is identical to creating music, except generally only one note is needed; hence there are no data statements. For details, see the *Commodore 64 Programmer's Reference Guide*.

```

10 CLR: REM ** SOUND EFFECTS **
15 PRINT"WHICH SOUND EFFECT?" :PRINT "1.
   WAILING":PRINT "2. SHOOTING": PRINT "3. ";
16 PRINT"SIREN":PRINT"4. ROCKET":PRINT"5. CRASH":
   PRINT"6. MACHINE GUN"
17 INPUT X
20 S = 54272:FOR SW = S TO S + 24:POKESW,0:NEXT:K = - 1:
   T1$ = "000000"
21 ON X GOTO 23,24,25,26,27,28
23 V = 15:N1 = W1 = 65:W2 = 64:AD = 15:SR = 0:DR = 1:P1 = 9:
   P2 = 255:Q = 1:GOTO30:REM WAILING
24 N2 = 200:N1 = 40:W1 = 129:W2 = 128:AD = 15:SR = 15:
   DR = 1;GOTO30:REM SHOOTING
25 N2 = 85:N1 = 36:W1 = 33:W2 = 32:AD = 136:SR = 129:
   DR = 350:V = 15:Q = 2:GOTO30:REM SIREN
26 N2 = 100:N1 = 25:W1 = 129:W2 = 128:AD = 9:SR = 129:
   DR = 50:K = -.25:GOTO30:REM ROCKET
27 N2 = 251 :N1 = 5:W1 = 129:W2 = 128:AD = 129:SR = 65:
   DR = 50:GOTO30:REM CRASH
28 N2 = 75:N1 = 34:W1 = 129:W2 = 128:AD = 8:SR = 1:DR = 50:
   V = 15:REM MACHINE GUN
30 POKE S + 2,P2:POKE S + 3,P1:REM PULSE
40 POKE S + 5,AD:POKE S + 6,SR:REM ADSR
50 POKE S + 1,N1:POKE S,N2:REM NOTE
55 IF Q = 2 THEN Q = 3
56 IF Q = 2 THEN POKE S + 1,64:POKE S,188
60 POKE S + 4,W1:REM ON SWITCH
63 IF Q < > 1 GOTO70
65 FOR N2 = 200TO5 STEP-1:POKE S,N2:NEXTN2
68 FOR N2 = 150TO5 STEP-1:POKE S,N2:NEXTN2
70 FOR VL = 15 TO V STEP K:POKE S + 24,VL:REM VOLUME
80 FOR T = 1 TO DR:NEXT T:REM DURATION
90 NEXT VL
100 POKE S + 4,W2:REM SOUND OFF
110 IF T1$ > = "000005"THEN 10
115 IFQ = 3THENQ = 2:GOTO56
120 GOTO50

```

## Program Notes

The Sound Effects program contains six sound effects the user can pick from. Lines 10 through 21 clear all the variables and request a selection. The variable K in line 20 is necessary for the rocket sound. TI\$ sets the built-in timer to zero. Lines 23 through 28 establish the values of the register variables for each sound. Lines 30 through 50 enter these values into the proper registers. The variable Q in lines 55, 56 and 115 restricts those lines to the siren. The variable Q in line 63 restricts lines 65 and 68 for wailing only. Line 70 allows for a variable volume; where none was required, V was set to 15. Line 80 allows for a variable note duration; when not required, the variable DR was set to 1. Lines 60 and 100 are the main registers. Line 110 cuts off the sound after five seconds. You can then select another effect.

\*\*\*\*\*

*Although by now you have experienced first hand the versatility and power of the Commodore 64C computer, you probably realize that you have only begun to tap the potential of this extraordinary computer. The next chapter defines the format and use of all elements of the BASIC 2.0 programming language.*



CHAPTER

**BASIC 2.0**

**Encyclopedia**

6



**CHAPTER 6**  
**BASIC 2.0**  
**Encyclopedia**

<b>INTRODUCTION</b>	<b>111</b>
Organization of Encyclopedia	111
Definition Format	111
<b>BASIC COMMANDS AND STATEMENTS</b>	<b>113</b>
<b>BASIC FUNCTIONS</b>	<b>133</b>
<b>VARIABLES AND OPERATORS</b>	<b>142</b>
Variables	142
Operators	144
<b>RESERVED WORDS AND SYMBOLS</b>	<b>146</b>
Reserved System Words (Keywords)	146
Reserved System Symbols	147



## Introduction

## Organization of Encyclopedia

This chapter lists BASIC 2.0 language elements. It gives a complete list of the rules (syntax) of Commodore BASIC 2.0, along with a concise description of each.

The different types of BASIC operations are listed in individual sections, as follows:

1. **COMMANDS and STATEMENTS:** the commands used to edit, store and erase programs; and the BASIC program statements used in the numbered lines of a program.
2. **FUNCTIONS:** the string, numeric and print functions.
3. **VARIABLES AND OPERATORS:** the different types of variables, legal variable names, arithmetic operators and logical operators.
4. **RESERVED WORDS AND SYMBOLS:** the words and symbols reserved for use by the BASIC 2.0 language, which cannot be used for any other purpose.

## Definition Format

The definitions in this encyclopedia are arranged in the following format:

<p><i>Command name</i> ↓ <b>CLOSE</b></p> <p><i>Format</i> →</p> <p><i>Discussion of format and use</i> →</p> <p><i>Example(s)</i> →</p>	<p><i>Brief definition</i> ↓ —Close logical file</p> <p><b>CLOSE file number</b></p> <p>This statement closes any files used by the OPEN statement. The number or variable following the word CLOSE is the file number to be closed</p> <p><b>EXAMPLE:</b></p> <p><b>CLOSE 2</b>      Logical file 2 is closed.</p>
--	---

In a typical definition, the boldface line that defines the format consists of the following elements:

```
LOAD "filename" [,device number] [,relocate flag]
  ↑            ↑            ↑
keyword   argument   additional arguments
                   (possibly optional)
```

The parts of the command or statement that must be typed exactly as shown are in capital letters. Words not capitalized indicate words that the user supplies, such as the name of a program.

When quote marks (“ ”) appear (usually around a program name or filename), the user must include them in the appropriate place, according to the format example.

**KEYWORDS**, also called reserved words, appear in upper-case letters. Keywords are words that are part of the BASIC language. They are the central part of a command or statement, and they tell the computer what kind of action to take. These words cannot be used as variable names. A complete list of reserved words and symbols is given in Section 20.

Keywords may be typed using the full word or the approved abbreviation. (A full list of abbreviations is given in Appendix K). The keyword or abbreviation must be entered correctly or an error will result. The BASIC and DOS error messages are defined in Appendices A and B, respectively.

**ARGUMENTS**, also called parameters, appear in lower-case letters. Arguments complement keywords by providing specific information to the command or statement. For example, the keyword load tells the computer to load a program while the argument tells the computer which specific program to load. A second argument specifies from which drive to load the program. Arguments include filenames, variables, line numbers, etc.

**SQUARE BRACKETS** [ ] show optional arguments. The user selects any or none of the arguments listed, depending on requirements.

**ANGLE BRACKETS** < > indicate the user **MUST** choose one of the arguments listed.

A **VERTICAL BAR** | separates items in a list of arguments when the choices are limited to those arguments listed. When the vertical bar appears in a list enclosed in **SQUARE BRACKETS**, the choices are limited to the items in the list, but the user still has the option not to use any arguments. If a vertical bar appears within angle brackets, the user must choose one of the listed arguments.

**ELLIPSIS** ... a sequence of three dots means an option or argument can be repeated more than once.

**QUOTATION MARKS** “ ” enclose character strings, filenames and other expressions. When arguments are enclosed in quotation marks, the quotation marks must be included in the command or statement. Quotation marks are not conventions used to describe formats; they are required parts of a command or statement.

**PARENTHESES** () When arguments are enclosed in parentheses, they *must* be included in the command or statement. Parentheses are not conventions used to describe formats; they are required parts of a command or statement.

**VARIABLE** refers to any valid BASIC variable names, such as X, A\$, T%, etc.

**EXPRESSION** refers to any valid BASIC expressions, such as  $A + B + 2$ ,  $.5*(X + 3)$ , etc.

## **BASIC Commands and Statements**

### **CLOSE**

—Close logical file

#### **CLOSE file number**

This statement closes any files used by the OPEN statement. The number or variable following the word CLOSE is the file number to be closed.

#### **EXAMPLE:**

**CLOSE 2**                      Logical file 2 is closed.

### **CLR**

—Clear program variables

#### **CLR**

This statement restores default I/O channels, clears (not closes) I/O channels, resets DATA statement pointer, resets stack pointer, and resets variable pointers, but leaves the program intact. This statement is automatically executed when a RUN or NEW command is given.

## CMD

—Redirect screen output

### **CMD logical file number [,write list]**

This command sends the output, which normally goes to the screen (i.e., PRINT statement, LIST, but not POKES into the screen) to another device, such as a disk data file or printer. This device or file must be OPENed first. The CMD command must be followed by a number or numeric variable referring to the file. The write list can be any alpha-numeric string or variable. This command is useful for printing headings at the top of program listings.

#### **EXAMPLE:**

<b>OPEN 1,4</b>	OPENS device #4, which is the printer.
<b>CMD 1</b>	All normal output now goes to the printer.
<b>LIST</b>	The LISTing goes to the printer, not the screen—even the word READY.
<b>PRINT#1</b>	Sends output back to the screen.
<b>CLOSE 1</b>	Closes the file.

## CONT

—Continue program execution

### **CONT**

This command is used to restart a program that has been stopped by either using the STOP key, a STOP statement, or an END statement. The program resumes execution where it left off. CONT will not resume the program execution if any editing of the program has been performed during the pause. If the program stopped due to an error; or if you have caused an error before trying to restart the program, CONT will not work. The error message in this case is CANT CONTINUE ERROR.

## DATA

—Define data to be used by a program

### **DATA list of constants**

This statement is followed by a list of data items to be input into the computer's variable memory by READ statements. The items may be numeric or string and are separated by commas. String data need not be inside quote marks, unless they contain any of the following characters: space, colon, or comma. If two commas have nothing between them, the value is input as a zero if numeric, or as an empty string. Also see the RESTORE statement, which allows the Commodore 64C to reread data.



**EXAMPLE:**

DATA 100, 200, FRED, "HELLO, MOM", , 3, 14, ABC123

**DEF FN**

—Define a user function

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

This statement allows the definition of an arithmetic calculation as a function. In the case of a long formula that is used several times within a program, use of a function can save valuable program space. The name given to the function begins with the letters FN, followed by any alphanumeric name beginning with a letter. First, define the function by using the statement DEF, followed by the name given to the function. Following the name is a set of parentheses () with a dummy numeric variable name (in this case, X) enclosed. Next is an equal sign, followed by the formula to be defined. The function can be performed by substituting any number for X, using the format shown in line 20 of the example below:

**EXAMPLE:**

```
10 DEF FNA(X) = 12*(34.75-X/.3) + X
20 PRINT FNA(7)
```

The number 7 is inserted each place X is located in the formula given in the DEF statement. In the example above, the answer returned is 144.

**DIM**

—Declare number of elements in an array

**DIM variable (subscripts) [,variable(subscripts)] . . .**

Before arrays of variables can be used, the program must first execute a DIM statement to establish DIMensions of the array (unless there are 11 or fewer elements in the array). The DIM statement is followed by the name of the array, which may be any legal variable name. Then, enclosed in parentheses, put the number (or numeric variable) of elements in each dimension. An array with more than one dimension is called a matrix. Any number of dimensions may be used, but keep in mind the whole list of variables being created takes up space in memory, and it is easy to run out of memory if too many are used. Here's how to calculate the amount of memory used by an array:

- 5 bytes for the array name
- 2 bytes for each dimension
- 2 bytes/element for integer variables
- 5 bytes/element for normal numeric variables
- 3 bytes/element for string variables
- 1 byte for each character in each string element

Integer arrays take up two-fifths the space of floating-point arrays (e.g., DIM A% (100) requires 209 bytes; DIM A (100) requires 512 bytes.)

Array elements are numbered 0 to N, where N is the maximum value specified in the DIM statement. Thus, X(0) through X(10) indicates 11 elements.

More than one array can be dimensioned in a DIM statement by separating the array variable names by commas. If the program executes a DIM statement for any array more than once, the message "RE'DIMed ARRAY ERROR" is posted. It is good programming practice to place DIM statements near the beginning of the program.

**EXAMPLE:**

```
10 DIM A$(40),B7(15),CC%(4,4,4)
```

Dimensions three arrays, where arrays A\$, B7, and CC% have, respectively, 41 elements, 16 elements and 125 elements

**END**

—Define the end of program execution

**END**

When the program encounters the END statement, it stops RUNNING immediately. The CONT command can be used to restart the program at the next statement (if any) following the END statement. END is not required to terminate a program.

**FOR/TO/STEP/  
NEXT**

—Define a repetitive program loop structure.

```
FOR variable = start value TO end value [STEP increment]  
NEXT [variable]
```

The FOR . . . NEXT statement sets up a section of the program (i.e., a loop) that repeats for a set number of times. This is useful when something needs to be counted or something must be done a certain number of times (such as printing).

This statement executes all the commands enclosed between the FOR and NEXT statements repetitively, according to the start and end values. The start value and the end value are the beginning and ending counts for the loop variable. The loop variable is added to or subtracted from during the FOR/NEXT loop.

The logic of the FOR/NEXT statement is as follows. First, the loop variable is set to the start value. When the program reaches a program line con-

taining the NEXT statement, it adds the STEP increment (default = 1) to the value of the loop variable and checks to see if it is higher than the end value of the loop. If the loop variable is less than or equal to the end value, the loop is executed again, starting with the statement immediately following the FOR statement. If the loop variable is greater than the end value, the loop terminates and the program resumes immediately following the NEXT statement. The opposite is true if the step size is negative.

**EXAMPLE:**

```
10 FOR L = 1 TO 10
20 PRINT L
30 NEXT L
40 PRINT "I'M DONE! L = "L
```

This program prints the numbers from one to 10 followed by the message I'M DONE! L = 11.

The end value of the loop may be followed by the word STEP and another number or variable. In this case, the value following the STEP is added each time instead of one. This allows counting backwards, by fractions, or in increments other than one.

The user can set up loops inside one another. These are known as nested loops. Care must be taken when nesting loops so the last loop to start is the first one to end. NEXT without a variable name completes the last executed FOR loop.

**EXAMPLE:**

```
10 FOR L = 1 TO 100
20 FOR A = 5 TO 11 STEP .5
30 NEXT A
40 NEXT L
```

The FOR . . . NEXT loop in lines 20 and 30 is nested inside the one in line 10 and 40. Using a STEP increment of .5 is used to illustrate the fact that floating point indices are valid.

**GET**

—Receive input from the keyboard, one character at a time, without waiting for a key to be pressed

**GET variable list**

The GET statement is a way to receive data from the keyboard, one character at a time. When GET is encountered in a program, the character that is typed is stored in the 64C's memory. If no character is typed, a null (empty) character is returned, and the program continues without waiting

for a key. There is no need to hit the RETURN key. The word GET is followed by a variable name, either numeric or string.

If the program intends to GET a numeric key and a key besides a number is pressed, the program stops and an error message is displayed. The GET statement may also be put into a loop, checking for an empty result. The GET statement can be executed only within a program. Otherwise an ILLEGAL DIRECT ERROR occurs.

**EXAMPLE:**

```
10 GETA$:IF A$(<)"A"THEN 10
```

This line waits for the A key to be pressed to continue.

```
20 GET B, C, D
```

GET numeric variables B,C and D from the keyboard without waiting for a key to be pressed.

**GET#**

—Receive input data from an input device

**GET# file number, variable list**

This statement inputs one character at a time from a previously opened file. In accepting keyboard input, the GET# statement works like the GET statement. The GET# statement can be executed only within a program.

**EXAMPLE:**

```
10 GET#1,A$
```

This example receives one character, which is stored in the variable A\$, from file number 1. This example assumes that file 1 was previously opened. See the OPEN statement.

**GOSUB**

—Call a subroutine from the specified line number

**GOSUB line number**

This statement is similar to the GOTO statement, in that the statement directs the computer to jump to a specified line and continue program execution at that line. However, a GOSUB statement must eventually encounter a RETURN statement. When the RETURN statement is encountered, the program jumps back to the statement immediately following the GOSUB statement.

The target of a GOSUB statement is called a subroutine. A subroutine is useful if a task is repeated several times within a program. Instead of duplicating the section of program over and over, set up a subroutine, and GOSUB to it at the appropriate time in the program. See also the RETURN statement.

**EXAMPLE:**

```
20 GOSUB 800      This example calls the subroutine beginning at
                  line 800 and executes it. All subroutines must
                  terminate with a RETURN statement.
                  :
                  :
800 PRINT "HI THERE": RETURN
```

**GOTO/GO TO**

—Transfer program execution to the specified line number

**GOTO line number**

After a GOTO statement is encountered in a program, the computer executes the statement specified by the line number in the GOTO statement. When used in direct mode, GOTO executes (RUNs) the program starting at the specified line number without clearing the variables or clearing disk channels, etc.

**EXAMPLES:**

```
10 PRINT"COMMODORE" The GOTO in line 20 makes line 10
20 GOTO 10           repeat continuously until RUN/STOP is
                    pressed.

GOTO 100            Starts (RUNs) the program starting at
                    line 100, without clearing the variable
                    storage area.
```

**IF/THEN**

—Evaluate a conditional expression and execute portions of a program depending on the outcome of the expression

**IF expression THEN [clause]**

The IF . . . THEN statement evaluates a BASIC expression and takes one of two possible courses of action depending upon the outcome of the expression. If the expression is true, the clause following THEN is executed. This can be any BASIC statement. If the expression is false, the program resumes with the program line immediately following the program line

containing the IF statement. The entire IF . . . THEN statement must be contained within 80 characters (two screen lines).

The IF . . . THEN statement can take two additional forms:

**IF expression THEN line number**

or:

**IF expression GOTO line number**

These forms transfer program execution to the specified line number if the expression is true. Otherwise, the program resumes with the program line number immediately following the line containing the IF statement. Consider the following example:

```
50 IF X > 0 THEN PRINT "OK"
```

This line checks the value of X. If X is greater than 0, the statement immediately following the keyword THEN (PRINT "OK") is executed. If X is less than or equal to 0, the program goes to the next line.

**EXAMPLE:**

```
:
10 IF X = 10 THEN 100
20 PRINT "X DOES NOT EQUAL 10"
:
99 STOP
100 PRINT "X EQUALS 10"
```

This example evaluates the value of X. IF X equals 10, the program control is transferred to line 100 and the message "X EQUALS 10" is printed. IF X does not equal 10, the program resumes with line 20, the 64C prints the message "X DOES NOT EQUAL 10" and the program stops.

## INPUT

—Receive a data string or a number from the keyboard and wait for the user to press RETURN

### **INPUT ["prompt string";] variable list**

The INPUT statement asks for data from the user while the program is RUNNING and places the data into a variable or variables. The program stops, prints a question mark (?) on the screen, and waits for the user to type the answer and hit the RETURN key. The word INPUT is followed by a prompt string and a variable name or list of variable names separated by commas. The message in the prompt string inside quotes suggests (prompts) the information the user should enter. If this message is present, there must be a semicolon (;) after the closing quote of the prompt.

When more than one variable is INPUT, separate them by commas. The computer asks for the remaining values by printing two question marks (??). If the RETURN key is pressed without INPUTting a value, the INPUT variable retains its previous value. The INPUT statement can be executed only within a program.

**EXAMPLE:**

```
10 INPUT "PLEASE TYPE A NUMBER";A
20 INPUT "AND YOUR NAME";A$
30 PRINT A$ " YOU TYPED THE NUMBER";A
```

**INPUT#**

—Inputs data from an I/O channel into a string or numeric variable

**INPUT# channel number, variable list**

This statement works like INPUT, but takes the data from a previously OPENed channel, usually on a disk or tape instead of the keyboard. No prompt string is used. This statement can be used only within a program.

**EXAMPLE:**

```
10 OPEN 2,8,2
20 INPUT#2, A$, C, D$
```

This statement INPUTs the data stored in variables A\$, C and D\$ from the disk channel number 2, which was OPENed in line 10.

**LET**

—Assigns a value to a variable

**[LET] variable = expression**

The word LET is rarely used in programs, since it is not necessary. Whenever a variable is defined or given a value, LET is always implied. The variable name that receives the result of a calculation is on the left side of the equal sign. The number, string or formula is on the right side. You can only assign one value with each (implied) LET statement.

**EXAMPLE:**

```
10 LET A = 5      Assign the value 5 to numeric variable A.
20 B = 6          Assign the value 6 to numeric variable B.
30 C = A * B + 3  Assign the numeric variable C, the value resulting
                  from 5 times 6 plus 3.
40 D$ = "HELLO"  Assign the string "HELLO" to string variable D$.
```

## LIST

—List the BASIC program currently in memory

### **LIST [first line] [ – last line]**

The LIST command displays a BASIC program that has been typed or LOADED into the Commodore 64C's memory so you can read and edit it. When LIST is used alone (without numbers following it), the Commodore 64C gives a complete LISTing of the program on the screen. The listing process may be slowed down by holding down the CTRL key, or stopped by hitting the RUN/STOP key. If LIST is followed by a line number, the Commodore 64C shows only that line. If LIST is typed followed by a number and just a dash, the Commodore 64C shows all lines from that number to the end of the program. If LIST is typed with a dash followed by a number, all lines from the beginning of the program to that line number are LISTed. If LIST is typed with two numbers separated by a dash, all lines from the first to the second line number inclusive are displayed. By using these variations, any portion of a program can be examined or brought to the screen for modification. LIST can be used in a program. Program execution will reset after the LIST is performed.

#### **EXAMPLES:**

<b>LIST</b>	Shows entire program.
<b>LIST 10</b>	Shows only line 10.
<b>LIST 100 –</b>	Shows from line 100 until the end of the program.
<b>LIST – 100</b>	Shows all lines from the beginning through line 100.
<b>LIST 10-200</b>	Shows lines from 10 to 200, inclusive.

## LOAD

—Load a program from a peripheral device such as a disk drive or Datassette

### **LOAD “filename” [,device number] [,relocate flag]**

This is the command used to recall a program stored on disk or cassette tape. Here, the filename is a program name up to 16 characters long, in quotes. The name can be followed by a comma (outside the quotes) and a device number to determine where the program is stored (disk or tape). If no number is supplied, the Commodore 64C assumes device number 1 (the Datassette tape recorder).

The relocate flag is a number (0 or 1) that determines where a program is loaded in memory. A relocate flag of 0 tells the Commodore 64C to load the program at the start of the BASIC program area. A flag of 1 tells the



computer to LOAD from the point where it was SAVED. The default value of the relocate flag is 0. A value of 1 is generally used when loading machine language programs or bit-map screens.

The device most commonly used with the LOAD command is the disk drive. This is device number 8.

If LOAD is typed with no arguments, followed by RETURN, the 64C assumes you are loading from tape and you are prompted to "PRESS PLAY ON TAPE". If you press PLAY, the 64C starts looking for a program on tape. When the program is found, the 64C prints FOUND"filename", where the filename is the name of the first file which the Datassette finds on the tape. Press the Commodore key or spacebar to LOAD the found filename. (If you press no key, after about 10 seconds the file is loaded automatically.) Once the program is LOADED, it can be RUN, LISTed or modified.

#### EXAMPLES:

LOAD	Reads in the next program from tape.
LOAD "HELLO"	Searches tape for a program called HELLO, and LOADs it if found.
LOAD A\$,8	LOADs the program from disk whose name is stored in the variable A\$.
LOAD"HELLO",8	Looks for the program called HELLO on disk drive number 8, drive 0.
LOAD"MACHLANG",8,1	LOADs the machine language program called "MACHLANG" into the location from which it was SAVED.

The LOAD command can be used within a BASIC program to find and RUN the next program on a tape or disk. This is called *chaining*.

## NEW

—Clear program and variable storage

## NEW

This command in effect "erases" the entire program in memory. NEW invokes an automatic CLR command, so that it restores default I/O channels, clears (but does not close) I/O channels, resets DATA statement pointer, resets stack pointer and resets variable pointers. Unless the program was stored on disk or tape, it is lost. Be careful with the use of this command. The NEW command also can be used as a statement in a BASIC program. However, when the Commodore 64C gets to this line, the program is erased and everything stops.

## ON

—Conditionally branch to a specified program line or call a subroutine according to the results of the specified expression

**ON expression <GOTO/GOSUB> line #1 [, line #2, . . . ]**

The word ON is followed by a mathematical expression, then either of the keywords GOTO or GOSUB and a list of line numbers separated by commas. If the integer result of the expression is 1, the first line in the list is executed. If the result is 2, the second line number is executed and so on. If the result is 0, or larger than the number of line numbers in the list, the program resumes with the line immediately following the ON statement. If the number is negative, an ILLEGAL QUANTITY ERROR results.

### EXAMPLE:

```
10 INPUT X:IF X<0 THEN 10
```

```
20 ON X GOTO 30, 40, 50, 60
```

```
25 STOP
```

When X = 1, ON sends control to the first line number in the list (30)

When X = 2, ON sends control to the second line (40), etc

```
30 PRINT "X = 1"
```

```
40 PRINT "X = 2"
```

```
50 PRINT "X = 3"
```

```
60 PRINT "X = 4"
```

If X = 0 or X > 4 the program terminates (breaks) at line 25.

## OPEN

—Open an input or output channel

**OPEN logical file number, device number [,secondary address] [, "filename, filetype, mode"] [,cmd string]**

The OPEN statement allows the Commodore 64C to access files within devices such as a disk drive, a Datassette cassette recorder, a printer or even the screen of the Commodore 64C. The word OPEN is followed by a logical file number, which is the number to which all other BASIC input/output statements will refer, such as PRINT#(write), INPUT#(read), etc. This number is from 0 to 255, but for most uses it should be from 1 to 127. The number zero and the numbers over 127 are reserved for special use.

The second number, called the device number, follows the logical file number. Device number 0 is the Commodore 64C keyboard; 1 is the cassette recorder; 2 is RS-232; 3 is the Commodore 64C screen, 4-7 are usually for printers; and 8-11 are usually for disk drives. It is often a good idea to use the same file number as the device number because it makes it easy to remember which is which. Valid device numbers are 0 to 30, of which the values from 4 to 30 are assumed to be serial bus devices.

Following the device number may be a third parameter called the secondary address. In the case of the cassette, this can be 0 for read, 1 for write and 2 for write with END-OF-TAPE marker at the end. In the case of the disk, the number refers to the channel number. See your disk drive manual for more information on channels and channel numbers. For the printer, the secondary addresses are used to select certain programming functions.

There may also be a filename specified for disk or tape OR a string following the secondary address, which could be a command to the disk/tape drive or the name of the file on tape or disk. If the filename is specified, the type and mode refer to disk files only. Disk file types currently include PROGRAM, SEQUENTIAL, RELATIVE and USER; modes are READ and WRITE.

**EXAMPLES:**

- |                                    |   |
|------------------------------------|---|
| 10 OPEN 3,3                        | OPENS the screen as file number 3.  |
| 20 OPEN 1,0                        | OPENS the keyboard as file number 1.  |
| 30 OPEN 1,1,0,"DOT"                | OPENS the cassette for reading, as file number 1, using "DOT" as the filename.  |
| OPEN 4,4                           | OPENS the printer as file number 4.   |
| OPEN 15,8,15                       | OPENS the command channel on the disk as file 15, with secondary address 15. Secondary address 15 is reserved for the disk drive command/error channel. |
| 5 OPEN 8,8,12,"TESTFILE,SEQ,WRITE" | OPENS a sequential disk file for writing called TESTFILE as file number 8, with secondary address 12.   |

See also: CLOSE, CMD, GET#, INPUT#, and PRINT# statements and system variable ST.

## POKE

—Change the contents of a memory location

### **POKE address, value**

The POKE statement allows changing of any value in the Commodore 64C RAM, and allows modification of many of the Commodore 64C's Input/Output registers. The keyword POKE is always followed by two parameters. The first is a location inside the Commodore 64C memory. This can be a value from 0 to 65535. The second parameter is a value from 0 to 255, which is placed in the location, replacing any value that was there

previously. The value of the memory location determines the bit pattern of the memory location.

**EXAMPLE:**

10 POKE 53280,1                      Changes screen border color

**NOTE:** PEEK, a function related to POKE, returns the contents of the specified memory location, is listed under FUNCTIONS.

**PRINT**

—Output to the text screen

**PRINT [print list]**

The PRINT statement is the major output statement in BASIC. While the PRINT statement is the first BASIC statement most people learn to use, there are many variations of this statement. The word PRINT can be followed by any of the following:

<b>Characters inside quotes</b>	("text")
<b>Variable names</b>	A, B, A\$, X\$
<b>Functions</b>	SIN(23), ABS(33)
<b>Expressions</b>	2 + 2, A + 3, A = B
<b>Punctuation marks</b>	; ,

The characters inside quotes are often called literals because they are printed literally, exactly as they appear. Variable names have the value they contain (either a number or a string) printed. Functions also have their number values printed.

Punctuation marks are used to help format the data neatly on the screen. The comma separates printed output by 10 spaces, while for numeric output only the semicolon causes the numbers to be preceded by a space or minus sign and followed by a cursor right. When used with text the semicolon adds no spaces. Either punctuation mark can be used as the last symbol in the statement. This results in the next PRINT statement acting as if it is continuing the previous PRINT statement.

## PRINT#

EXAMPLES:	RESULTS
10 PRINT "HELLO"	HELLO
20 A\$ = "THERE":PRINT "HELLO ";A\$	HELLO THERE
30 A = 4:B = 2:?A + B	6
40 J = 41:PRINT J::PRINT J - 1	41 40
50 PRINT A;B::D = A + B:PRINT D;A-B	4 2 6 2

See also POS, SPC and TAB functions.

—Output data to files

### **PRINT# logical channel number, [print list]**

PRINT# is followed by a number which refers to the data channel previously OPENed. The number is followed by a comma and a list of items to be output to the channel, which can be strings, numeric or string variables or numeric data. The comma and semicolon act in the same manner for spacing with printers as they do in the PRINT statement. Some devices may not work with TAB and SPC.

#### **EXAMPLE:**

10 OPEN 4,4	Outputs the data "HELLO
20 PRINT#4,"HELLO THERE!",A\$,B\$	THERE" and the variables A\$ and B\$ to the printer.
10 OPEN 2,8,2	Outputs the data variables
20 PRINT#2,A,B\$,C,D	A, B\$, C and D to the disk file number 2.

**NOTE:** After a CMD command has been used, the PRINT# command is used by itself to "unlisten" a device (e.g., close the channel to the printer) before closing the file, as shown in this example:

```
10 OPEN 4,4
20 CMD 4
30 PRINT#4,"PRINT WORDS"
40 PRINT#4
50 CLOSE 4
```

## READ

—Read data from DATA statements and input it into variable memory

### READ variable list

This statement inputs information from DATA statements and stores it in variables. The READ statement variable list may contain both strings and numbers. Be careful to avoid reading strings where the READ statement expects a number and vice versa. This produces a TYPE MISMATCH ERROR message.

The data in the DATA statements are READ in sequential order. Each READ statement can read one or more data items. Every variable in the READ statement requires a data item. If one is not supplied, an OUT OF DATA ERROR occurs. See the DATA statement.

In a program, you can READ the data and then re-read it by issuing the RESTORE statement. The RESTORE sets the sequential data pointer back to the beginning, where the data can be read again. See the RESTORE statement.

#### EXAMPLES:

```
10 READ A, B, C
20 DATA 3, 4, 5
```

READ the first three numeric variables.

```
10 READ A$, B$, C$
20 DATA JOHN, PAUL, GEORGE
```

READ the first three string variables.

```
10 READ A, B$, C
20 DATA 1200, NANCY, 345
```

READ (and input into the 64C's memory) a numeric variable, a string variable and another numeric variable.

## REM

—Comments or remarks about the operation of a program line

### REM message

The REMark statement is a note to whoever is reading a listing of the program. REM may explain a section of the program, give information about the author, etc. REM statements do not affect the operation of the program, except to add length to it (and therefore use more memory). Nothing to the right of the keyword REM is interpreted by the computer as an executable instruction. (However, LIST *will* interpret graphic characters as tokens.) Therefore, no other executable statement can follow a REM on the same line.

#### EXAMPLE:

```
10 NEXT X:REM THIS LINE INCREMENTS X.
```

## RESTORE

—Reset DATA pointer so the DATA can be reREAD

### RESTORE [line #]

BASIC maintains an internal pointer to the next DATA constant to be READ. This pointer can be reset to the beginning of the program with RESTORE. When the RESTORE statement is executed in a program, the DATA pointer is reset to the first item in the first DATA statement of the program. This provides the capability to reREAD the data.

#### EXAMPLES:

10 FOR I = 1 TO 3	This example READs the data
20 READ X	in line 70 and stores it in
30 GROSS = X + GROSS	numeric variable X. It adds
40 NEXT	the total of all the numeric
50 RESTORE	data items. Once all the data
60 GOTO 10	has been READ, three cycles through
70 DATA 10,20,30	the loop, the READ pointer is
	RESTOREd to the beginning of the
	program and it returns to line 10 and
	performs repetitively.

10 READ A,B,C	This example RESTORES the DATA
20 DATA 100,500,750	pointer to the beginning data
30 READ X,Y,Z	item in line 20. When line 60
40 DATA 36,24,38	is executed, it will READ the
50 RESTORE	DATA 100,500,750.
60 READ S,P,Q	

## RETURN

—Return from subroutine

### RETURN

This statement is always paired with the GOSUB statement. When the program encounters a RETURN statement, it goes to the statement immediately following the last GOSUB command executed. If no GOSUB was previously issued, then a RETURN WITHOUT GOSUB ERROR message is displayed and the program stops. All subroutines end with a RETURN statement.

### EXAMPLE:

```
10 PRINT "ENTER SUBROUTINE"  
20 GOSUB 100  
30 PRINT "BACK FROM SUBROUTINE"  
.  
.  
.  
90 STOP  
100 PRINT "SUBROUTINE 1"  
110 RETURN
```

This example calls the subroutine at line 100 which prints the message "SUBROUTINE 1" and RETURNS to line 30, the rest of the program.

## RUN

—Execute BASIC program

### 1) RUN [line #]

Once a program has been typed into memory or LOADED, the RUN command executes it. Before starting program execution, RUN clears all variables, resets DATA statement pointer, clears (but does not close) I/O channels, and restores default I/O channels. If there is a number following the RUN command, execution starts at that line number.

### EXAMPLES:

RUN	Starts execution from the beginning of the program.
RUN 100	Starts program execution at line 100.

## SAVE

—Store the program in memory to disk or tape

### SAVE ["filename"][,device number][,EOT flag]

This command stores a program currently in memory onto a cassette tape or disk. If the word SAVE is typed alone followed by RETURN, the Commodore 64C assumes that the program is to be stored on cassette tape. It has no way of checking if there is already a program on the tape in that location, so make sure you do not record over valuable information on your tape. If SAVE is followed by a filename in quotes or a string variable name, the Commodore 64C gives the program that name, so it may be located easily and retrieved in the future. If a device number is specified for the SAVE, follow the name with a comma (after the quotes) and a number or numeric variable. Device number 1 is the tape drive, and number 8 is the disk drive. After the device number on a tape command, there can be a



comma and a second number or *secondary address*. If this number is 0, a normal BASIC SAVE occurs. If the number is 1, the 64C saves the current starting address in the tape header for use in subsequent LOAD operations. If the number is 2, the Commodore 64C puts an END-OF-TAPE marker (EOT flag) after the program. If the number 3 is encountered, the 64C saves the current starting address in the tape header and an EOT marker is set. If, in trying to LOAD a program, the Commodore 64C finds one of these markers, the program is not loaded and a FILE NOT FOUND ERROR is reported.

**EXAMPLES:**

- |                           |  |
|---------------------------|--|
| <b>SAVE</b>               | Stores program on tape, without a name.  |
| <b>SAVE "HELLO"</b>       | Stores a program on tape, under the name HELLO.                                      |
| <b>SAVE A\$,8</b>         | Stores on disk, with the name stored in variable A\$.                                |
| <b>SAVE "HELLO", 8</b>    | Stores on disk, with name HELLO (equivalent to DSAVE "HELLO").                       |
| <b>SAVE "HELLO", 1, 2</b> | Stores on tape, with name HELLO, and places an END-OF TAPE marker after the program. |

**STOP**

—Halt program execution

**STOP**

This statement halts the program. A message, BREAK IN LINE XXX, occurs (only in program mode), where XXX is the line number containing the STOP command. The program can be restarted at the statement following STOP if the CONT command is used immediately, without any editing occurring in the listing. The STOP statement is often used while debugging a program.

**SYS**

—Call and execute a machine language subroutine at the specified address

**SYS address**

This statement performs a call to a subroutine at a given address. The address range is 0 to 65535. The program begins executing the machine-language program starting at that memory location.

**EXAMPLE:**

- |                  |  |
|------------------|--|
| <b>SYS 40960</b> | Calls and executes the machine-language routine at location 40960. |
|------------------|--|

## VERIFY

—Verify program in memory against one saved to disk or tape

### **VERIFY “filename” [,device number] [,relocate flag]**

This command causes the Commodore 64C to check the program on tape or disk against the one in memory, to determine if the program was **SAVED**. This command is also very useful for positioning a tape so that the Commodore 64C writes after the last program on the tape. It will do so, and inform the user that the programs don't match. The tape is then positioned properly, and the next program can be stored without fear of erasing the previous one.

**VERIFY**, with no arguments after the command, causes the Commodore 64C to check the next program on tape, regardless of its name, against the program now in memory. **VERIFY**, followed by a program name in quotes or a string variable in parentheses, searches the tape for that program and then checks it against the program in memory when found. **VERIFY**, followed by a name, a comma and a number, checks the program on the device with that number (1 for tape, 8 for disk). The relocate flag is the same as in the **LOAD** command. It verifies the program from the memory location from which it was **SAVED**.

#### **EXAMPLES:**

<b>VERIFY</b>	Checks the next program on the tape.
<b>VERIFY “HELLO”</b>	Searches for <b>HELLO</b> on tape, checks it against memory.
<b>VERIFY “HELLO”, 8,1</b>	Searches for <b>HELLO</b> on disk, then checks it against memory.

## WAIT

—Pause program execution until a data condition is satisfied

### **WAIT <location>, <mask-1> [,mask-2]**

The **WAIT** statement causes program execution to be suspended until a given memory address recognizes a specified bit pattern or value. In other words, **WAIT** can be used to halt the program until some external event has occurred. This is done by monitoring the status of bits in the Input/Output registers. The data items used with the **WAIT** can be values in the range 0–65535 for *location* and 0–255 for *masks*. For most programmers, this statement should never be used. It causes the program to halt until a specific memory location's bits change in a specific way. This is useful for certain I/O operations. The **WAIT** statement takes the value in the memory location and performs a logical **AND** operation with the value in *mask-1*. If *mask-2* is specified, the result of the first operation is exclusively **ORed** with *mask-2*. In other words, *mask-1* “filters out” any bits not to be tested. Where the bit is 0 in *mask-1*, the corresponding bit in the result will

always be 0. The mask-2 value flips any bits, so that an off condition can be tested for as well as an on condition. Any bits being tested for a 0 should have a 1 in the corresponding position in mask-2. If corresponding bits of the <mask-1> and <mask-2> operands differ, the exclusive-OR operation gives a bit result of 1. If the corresponding bits get the same result the bit is 0. It is possible to enter an infinite pause with the WAIT statement, in which case the RUN/STOP and RESTORE keys can be used to recover.

The first example below WAITs until a key is pressed on the tape unit to continue with the program. The second example will WAIT until a sprite collides with the screen background.

**EXAMPLES:**

**WAIT 1, 32, 32**

**WAIT 53273, 6, 6**

**BASIC Functions**

**Function Format**

The format of the function descriptions in the following pages is:

**FUNCTION (argument)**

where the argument can be a numeric value, variable or string.

Each function description is followed by an EXAMPLE. The lines appearing in bold face in the examples are what *you* type. The line without bold is the computer's response.

**ABS**

—Return absolute value

**ABS (X)**

The absolute value function returns the unsigned value of the argument X.

**EXAMPLE:**

**PRINT ABS (7\*(- 5) )**

35

## ASC

—Return CBM ASCII code for character

### ASC(X\$)

This function returns the Commodore ASCII code of the first character of X\$. You must append CHR\$(0) to a null string, or else an ILLEGAL QUANTITY ERROR is issued.

#### EXAMPLE:

```
X$ = "CBM":PRINT ASC (X$)
```

```
67
```

## ATN

Compute arctangent, in radians, of X

—Return angle whose tangent is X radians

### ATN (X)

This function computes the arctangent, measured in radians, of X. The value returned is in the range  $-\pi/2$  through  $\pi/2$ .

#### EXAMPLE:

```
PRINT ATN (3)
```

```
1.24904577
```

## CHR\$

—Return ASCII character for specified CBM ASCII code

### CHR\$(X)

This is the opposite of ASC and returns the string character whose CBM ASCII code is the integer value of X. X must be 0-255. Refer to Appendix D for a table of CHR\$ codes.

#### EXAMPLES:

```
PRINT CHR$ (65) Prints the A character.
```

```
A
```

```
PRINT CHR$ (147) Clears the text screen.
```

## COS

—Return cosine for angle of X radians

### COS(X)

This function returns the value of the cosine of X, where X is an angle measured in radians. The value returned is in the range  $-1$  to  $1$ .

**EXAMPLE:**

```
PRINT COS ( $\pi/3$ )  
.50000001
```

**EXP**

—Return value of e (2.71828183) raised to the X power

**EXP(X)**

This function returns a value of e (2.71828183) raised to the power of X.

**EXAMPLE:**

```
PRINT EXP(1)  
2.71828183
```

**FNxx**

—Return value from user defined function

**FNxx(x)**

This function returns the value from the user-defined function xx created in a DEF FNxx statement.

**EXAMPLE:**

```
10 DEF FNAA(X)=(X-32)*5/9  
20 INPUT X  
30 PRINT FNAA(X)  
RUN  
? 40 (? is input prompt)  
4.44444445
```

**FRE**

—Return number of available bytes in memory

**FRE (X)**

where X a dummy argument. The 64C returns the number of bytes as a signed 16-bit value. To get the actual number of bytes, use:

```
PRINT FRE (0)<0* - 65536 + FRE(0)
```

**EXAMPLE:**

```
PRINT FRE (0) Returns the current number of free bytes for  
BASIC programs and variables.
```

## INT

—Return integer form (truncated) of a floating point value

### INT(X)

This function returns the integer value of the expression. If the expression is positive, the fractional part is left out. Any fraction causes the next lower integer to be returned.

#### EXAMPLES:

```
PRINT INT(3.14)
```

```
3
```

```
PRINT INT(- 3.14)
```

```
-4
```

## LEFT\$

—Return the leftmost characters of string

### LEFT\$(string,integer)

This function returns a string consisting of the number of leftmost characters of the string determined by the specified integer. The integer argument must be in the range 0 to 255. If the integer is greater than the length of the string, the entire string is returned. If an integer value of zero is used, then a null string (of zero length) is returned.

#### EXAMPLE:

```
PRINT LEFT$ ("COMMODORE",5)
```

```
COMMO
```

## LEN

—Return the length of a string

### LEN(string)

This function returns the number of characters in the string expression. Non-printed characters and blanks are included.

#### EXAMPLE:

```
PRINT LEN ("COMMODORE")
```

```
9
```

## LOG

—Return natural log of X

### LOG(X)

This function returns the natural log of X, where  $X > 0$ . The natural log is log to the base e (see EXP(X)). To convert to log base 10, divide by LOG(10).

#### EXAMPLE:

```
PRINT LOG (37/5)
2.00148
```

## MID\$

—Return a substring from a larger string

### MID\$(string,starting position[,length])

This function returns a substring specified by the LENGTH, starting at the character specified by the starting position. The starting position of the substring defines the first character where the substring begins. The length of the substring is specified by the length argument. The starting position value can range from 1 to 255; the length value can range from 0 to 255. If the starting position value is greater than the length of the string, or if the length value is zero, then MID\$ returns a null string value. If the length argument is left out, all characters to the right of and including the starting position are returned.

#### EXAMPLE:

```
PRINT MID$("COMMODORE 64C",3,5)
MMODO
```

## PEEK

—Return contents of a specified memory location

### PEEK(X)

This function returns the contents of memory location X, where X is located in the range 0 to 65535, returning a result between 0 and 255. This is the counterpart of the POKE statement.

#### EXAMPLE:

```
PRINT PEEK (650)
0
```

In this example, the 0 indicates that keyboard is in normal operating mode with regard to which keys repeat when held down.

## $\pi$

—Return the value of pi (3.14159265)

### $\pi$

**EXAMPLE:**

```
PRINT  $\pi$  This returns the result 3.14159265.
```

## POS

—Return the current cursor column position on the screen

### POS(X)

The POS function indicates in which screen column the cursor is currently located. X is a dummy argument, which must be specified, but the value is ignored.

**EXAMPLE:**

```
10 PRINT"CURSOR IS IN COLUMN";  
20 PRINTPOS(0)
```

When you run this program, the screen displays this:

```
CURSOR IS IN COLUMN 19
```

This means that, after displaying the words CURSOR IS IN COLUMN, the cursor was in column 19.

## RIGHT\$

—Return sub-string from rightmost end of string

### RIGHT\$(string, integer)

This function returns a sub-string taken from the rightmost characters of the string argument. The length of the sub-string is defined by the length argument which can be any integer in the range of 0 to 255. If the value of the numeric expression is zero, a null string (zero length) is returned. If the value given in the length argument is greater than the length of the string, the entire string is returned. Also see the LEFT\$ and MID\$ functions.

**EXAMPLE:**

```
PRINT RIGHT$("BASEBALL",5)  
EBALL
```



## RND

—Return a random number

### RND (X)

This function returns a random number between 0 and 1. This is useful in games, to simulate dice roll and other elements of chance. It is also used in some statistical applications.

- If X = 0 Returns a random number based on the hardware clock.
- IF X > 0 Generates a reproducible random number based on the seed value below.
- IF X < 0 Produces a random number which is used as a base called a seed. Starts a repeatable sequence.

To simulate the rolling of a die, use the formula  $\text{INT}(\text{RND}(1)*6 + 1)$ . First the random number from 0 to 1 is multiplied by 6, which expands the range to 0-6 (actually, greater than zero and less than six). Then 1 is added, making the range greater than 1 and less than 7. The INT function truncates all the decimal places, leaving the result as a digit from 1 to 6.

#### EXAMPLES:

```
PRINT RND(0)           Displays a random number
.507824123             between 0 and 1.
PRINT INT(RND(1)*100 + 1) Displays a random number
89                    between 1 and 100.
```

## SGN

—Return sign of argument X

### SGN(X)

This function returns the sign,(positive, negative or zero) of X. The result is + 1 if X > 0, 0 if X = 0, and - 1 if X < 0.

#### EXAMPLE:

```
PRINT SGN(4.5);SGN(0);SGN(- 2.3)
1 0 -1
```

## SIN

—Return sine of argument

### SIN(X)

This is the trigonometric sine function. The result is the sine of X. X is measured in radians.

#### EXAMPLE:

```
PRINT SIN ( $\pi/3$ )
.866025404
```

## SPC

—Skip spaces on print output

### SPC (X)

This function is used in PRINT or PRINT# commands to skip over X numbers from the current character position. Note that characters passed over are not erased. See also the TAB function, which advances to a fixed column position.

#### EXAMPLE

```
PRINT "COMMODORE";SPC(3);"64C"  
COMMODORE 64C
```

## SQR

—Return square root of argument

### SQR (X)

This function returns the value of the SQuare Root of X, where X is a positive number or 0. The value of the argument must not be negative, or the BASIC error message ILLEGAL QUANTITY is displayed.

#### EXAMPLE:

```
PRINT SQR(25)  
5
```

## STR\$

—Return string representation of number

### STR\$ (X)

This function returns the STRing representation of the numeric value of the argument X. The string characters are the same as those that would be printed. That is, positive numbers and zero are preceded by a space, while negative numbers are preceded by a minus sign. The counterpart of the STR\$ function is the VAL function.

#### EXAMPLE

```
PRINT STR$(123.45)  
123.45  
  
PRINT STR$(-89.03)  
-89.03  
  
PRINT STR$(1E20)  
1E+20
```

## TAB

—Move cursor to tab position in present statement

### TAB(X)

The TAB function is used in PRINT and PRINT# commands to conditionally skip to a specified column position. TAB operates differently with screens than with printers or disk files. For printers or disk output, TAB acts exactly as SPC does (see the SPC description). For screen output, if column X is to the right of the current column position, then X becomes the current column position. If X is at the same position as or left of the current column position, TAB has no effect. Characters passed over are not erased.

#### EXAMPLE:

```
10 PRINT"COMMODORE"TAB(25)"64C"  
COMMODORE 64C
```

## TAN

—Return tangent of argument

### TAN(X)

This function returns the tangent of X, where X is an angle in radians.

#### EXAMPLE:

```
PRINT TAN(.785398163)  
1
```

## USR

—Call user-defined subprogram

### USR(X)

When this function is used, BASIC puts the value of X into the Floating Accumulator (FAC) in locations \$0061 through \$0066 (97 through 102) and calls the USR vector. You must put your machine language routine's address at \$0311(785) and \$0312(786) (low/high bytes). Since USR is a function, it returns a real value. Whatever is in the FAC when your machine language routine returns is passed. The USR vector defaults to an ILLEGAL QUANTITY ERROR routine.

#### EXAMPLE:

```
10 POKE 785,0  
20 POKE 786,192  
30 A = USR(X)  
40 PRINT A
```

Place starting location (\$C000 = 49152:\$00 = 0:\$C0 = 192) of machine language routine in location 785 and 786. Line 30 stores the returning value from the floating point accumulator.

## VAL

—Return the numeric value of a number string

### VAL(X\$)

This function converts the string X\$ into a number which is the inverse operation of STR\$. The string is examined from the left-most character to the right, for as many characters as are in recognizable number format. If the Commodore 64C finds illegal characters, only the portion of the string up to that point is converted. Acceptable numeric characters are:

0-9 spaces		+ , -	Preceding a number or following E only
decimal point (one only)	E		Exponential location (one only)

If no numeric characters are present, VAL returns a 0.

#### EXAMPLE:

```
10 A$ = "120"  
20 B$ = "365"  
30 PRINT VAL (A$) + VAL (B$)  
RUN  
485
```

## Variables and Operators

### Variables

The Commodore 64C uses three types of variables in BASIC. These are: normal numeric, integer numeric and string (alphanumeric).

Normal NUMERIC VARIABLES, also called floating point variables, can have any value from **\*\*superscript\*\*** - 10 to **\*\*superscript\*\*** + 10, with up to nine digits of accuracy. When a number becomes larger than nine digits can show, as in + 10 or - 10, the computer displays it in scientific notation form, with the number normalized to one digit and eight decimal places, followed by the letter E and the power of 10 by which the number is multiplied. For example, the number 12345678901 is displayed as 1.23456789E + 10.

INTEGER VARIABLES can be used when the number is from + 32767 to - 32768, and with no fractional portion. An integer variable is a number like 5, 10 or - 100. Integers take up less space than floating point variables, particularly when used in an array.

STRING VARIABLES are those used for character data, which may contain numbers, letters and any other characters the Commodore 64C can display. An example of a string variable is "Commodore 64C."

VARIABLE NAMES may consist of a single letter, a letter followed by a number or two letters. Variable names may be longer than two characters, but only the first two are significant. An integer is specified by using the percent sign (%) after the variable name. String variables have a dollar sign (\$) after their names.

**EXAMPLES:**

**Numeric Variable Names:** A, A5, BZ

**Integer Variable Names:** A%, A5%, BZ%

**String Variable Names:** A\$, A5\$, BZ\$

ARRAYS are lists of variables with the same name, using an extra number (or numbers) to specify an element of the array. Arrays are defined using the DIM statement and may be floating point, integer or string variable arrays. The array variable name is followed by a set of parentheses () enclosing the number of the variable in the list.

**EXAMPLE:**

A(7), BZ%(11), A\$(87)

Arrays can have more than one dimension. A two-dimensional array may be viewed as having rows and columns, with the first number identifying the row and the second number identifying the column (as if specifying a certain grid on a map).

**EXAMPLE:**

A(7,2), BZ%(2,3,4), Z\$(3,2)

RESERVED VARIABLE NAMES are names reserved for use by the Commodore 64C, and may not be used for another purpose. These are the variables ST, TI and TI\$. Words such as TO and IF or any other names that contain keywords, such as RUN, NEW or LOAD, cannot be used.

ST is a status variable for input and output (except normal screen/ keyboard operations). The value of ST depends on the results of the last I/O operation. In general, if the value of ST is 0, then the operation was successful.

TI and TI\$ are variables that relate to the real time clock built into the Commodore 64C. The system clock is updated every 1/60th of a second. It starts at 0 when the Commodore 64C is turned on, and is reset only by changing the value of TI\$. The variable TI gives the current value of the

clock in 1/60th of a second. TI\$ is a string that reads the value of the real time clock as a 24-hour clock. The first two characters of TI\$ contain the hour, the third and fourth characters are minutes and the fifth and sixth characters are seconds. This variable can be set to any value (so long as all characters are numbers) and will be updated automatically as a 24-hour clock.

**EXAMPLE:**

TI\$ = "101530" Sets the clock to 10:15 and 30 seconds (AM).

The value of the clock is lost when the Commodore 64C is turned off. It starts at zero when the Commodore 64C is turned on, and is reset to zero when the value of the clock exceeds 235959 (23 hours, 59 minutes and 59 seconds).

### **Operators**

The BASIC operators include ARITHMETIC, RELATIONAL and LOGICAL 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. If several operators are used together, the computer assigns priorities as follows: First, exponentiation, then multiplication and division, and last, addition and subtraction. If two operators have the same priority, then calculations are performed in order from left to right. If these operations are to occur in a different order, Commodore 64C BASIC allows giving a calculation a higher priority by placing parentheses around it. Operations enclosed in parentheses will be calculated before any other operation. Make sure the equations have the same number of left and right parentheses, or a SYNTAX ERROR message is posted when the program is run.

There are also operators for equalities and inequalities, called RELATIONAL operators. Arithmetic operators always take priority over relational operators.

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

Finally, there are three LOGICAL operators, with lower priority than both arithmetic and relational operators:

AND  
OR  
NOT

These are most often used to join multiple formulas in IF . . . THEN statements. When they are used with arithmetic operators, they are evaluated last (i.e., after + and -). If the relationship stated in the expression is true, the result is assigned an integer value of -1. If false, a value of 0 is assigned.

**EXAMPLES:**

IF A = B AND C = D THEN 100	Requires both A = B and C = D to be true.
IF A = B OR C = D THEN 100	Allows either A = B or C = D to be true.
A = 5:B = 4:PRINT A = B	Displays a value of 0.
A = 5:B = 4:PRINT A > 3	Displays a value of -1.
PRINT 123 AND 15:PRINT 5 OR 7	Displays 11 and 7.

## Reserved Words and Symbols

## Reserved System Words (Keywords)

This section lists the words and symbols used to make up the BASIC 2.0 language. These words and symbols cannot be used within a program as other than a component of the BASIC language. The only exception is that they may be used within quotes in a PRINT statement.

ABS	FN	LIST	READ	PRINT
AND	FOR	LOAD	REM	PRINT#
ASC	FRE	LOG	RESTORE	STR\$
ATN	GET	MID\$	RETURN	SYS
CHR\$	GET#	NEW	RIGHT\$	TAB
CLOSE	GOSUB	NEXT	RND	TAN
CLR	GO	NOT	RUN	THEN
CMD	GOTO	ON	SAVE	TI
CONT	IF	OPEN	SGN	TIME
COS	INPUT	OR	SIN	TIMES\$
DATA	INPUT#	PEEK	SPC	TI\$
DEF	INT	POKE	SQR	TO
DIM	LEFT\$	POS	ST	USR
END	LEN	PRINT	STEP	VAL
EXP	LET	PRINT#	STOP	VERIFY
				WAIT



## Reserved System Symbols

The following characters are reserved system symbols.

Symbol	Use(s)	
+	Plus sign movement	Arithmetic addition; string concatenation
-	Minus sign movement	Arithmetic subtraction; negative number; unary minus
*	Asterisk	Arithmetic multiplication
/	Slash	Arithmetic division
↑	Up arrow	Arithmetic exponentiation
	Blank space	Separate keywords and variable names
=	Equal sign	Value assignment; relationship testing
<	Less than	Relationship testing
>	Greater than	Relationship testing
,	Comma	Format output in variable lists; command/statement function parameters
.	Period	Decimal point in floating point constants
;	Semicolon	Format output in variable lists
:	Colon	Separate multiple BASIC statements on a program line
“”	Quotation mark	Enclose string constants
?	Question mark	Abbreviation for the keyword PRINT
(	Left parenthesis	Expression evaluation and functions
)	Right parenthesis	Expression evaluation and functions
%	Percent	Declare a variable name as integer
#	Number	Precede the logical file number in input/output statements
\$	Dollar sign	Declare a variable name as a string
π	Pi	Declare the numeric constant 3.14159265



# APPENDICES

- APPENDIX A — BASIC 2.0 ERROR MESSAGES
- APPENDIX B — CONNECTORS/PORTS FOR PERIPHERAL  
EQUIPMENT
- APPENDIX C — SCREEN DISPLAY CODES
- APPENDIX D — ASCII AND CHR\$ CODES
- APPENDIX E — SCREEN AND COLOR MEMORY MAPS
- APPENDIX F — DERIVED TRIGONOMETRIC FUNCTIONS
- APPENDIX G — MEMORY MAP
- APPENDIX H — BASIC 2.0 ABBREVIATIONS
- APPENDIX I — SPRITE REGISTER MAP
- APPENDIX J — SOUND AND MUSIC



## APPENDIX A

### BASIC 2.0 ERROR MESSAGES

### ERROR MESSAGES

MESSAGE	What the Problem Is	What to Do
BAD DATA	String data was received from an open file, but the program was expecting numeric data.	Make sure data was saved with a separator between each item.
BAD SUBSCRIPT	The program was trying to reference an element of an array whose number is outside the range specified in the DIM statement.	Verify you have dimensioned the array properly. In direct mode, have the 64C print the value of the subscript as a clue.
BREAK	Program execution was stopped because you hit the STOP key.	Use the CONT command to proceed or reRUN the program.
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.	You probably made a correction; reRUN the program.
DEVICE NOT PRESENT	The required I/O device not available for an OPEN, CLOSE, CMD, PRINT #, INPUT #, or GET #.	Verify the peripheral you are calling for is on and proper OPEN statement is used.
DIVISION BY ZERO	Division by zero is a mathematical oddity and not allowed.	Command the 64C to print the suspect variables to determine which one became a zero.
EXTRA IGNORED	Too many items of data were typed in response to an INPUT statement. Only the first few items were accepted.	Check your punctuation.
FILE NOT FOUND	No file with that name exists.	Verify you have the correct tape or disk and you spelled the name correctly; note especially spacing and upper-case characters.
FILE NOT OPEN	The file specified in a CLOSE, CMD, PRINT #, INPUT #, or GET #, must first be OPENed.	Open file. Verify you used proper file number.
FILE OPEN	An attempt was made to open a file using the number of an already open file.	Close file first or use new file number.
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.	Use smaller strings. Reduce the number of parentheses.
ILLEGAL DEVICE NUMBER	Occurs when you try to access a device illegally (e.g., LOADING from keyboard, screen or RS-232C).	Use correct device number.

MESSAGE	What the Problem Is	What to Do
ILLEGAL DIRECT	The INPUT statement can only be used within a program, and not in direct mode.	Use another command.
ILLEGAL QUANTITY	A number used as the argument of a function or statement is out of the allowable range.	Use direct mode to determine the value of the variables at the moment. Correct negative subscripts. Verify dimensions are large enough.
LOAD	There is a problem with the program on disk.	Reload.
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.	Verify the loop has a starting and ending point. Do not jump into the middle of a loop.
NOT INPUT FILE	An attempt was made to INPUT or GET data from a file which specified to be for output only.	Correct the OPEN statement's secondary address.
NOT OUTPUT FILE	An attempt was made to PRINT data to a file which was specified as input only.	Correct the OPEN statement's secondary address.
OUT OF DATA	A READ statement was executed but there is no data left unREAD in a DATA statement.	Verify data was not missed; add more data if necessary.
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.	Reduce the quantity of GOSUBs and FOR NEXT loops operating at once. Reuse loop variables where possible to prevent too many unfinished loops. Clean up the memory using FRE(X) function.
OVERFLOW	The result of a computation is larger than the largest number allowed, which is 1.70141884E + 38.	Check your computation steps.
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.	If the array was identified early it was automatically dimensioned to 10. Locate the DIM statement before using the variable.

MESSAGE	What the Problem Is	What to Do
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.	Provide the proper INPUT response.
RETURN WITHOUT GOSUB	A return statement was encountered, and no GOSUB command has been issued.	Verify the program ends before coming to subroutines tagged at program's end.
STRING TOO LONG	A string can contain up to 255 characters.	Keep strings to 255 characters and any single INPUT to 80 characters.
?SYNTAX ERROR	A statement is unrecognizable by the Commodore 64C. A missing or extra parenthesis, misspelled keywords, etc.	Look for spelling or grammar errors or words not in the BASIC vocabulary.
TYPE MISMATCH	This error occurs when a number is used in place of a string, or vice-versa.	Verify \$ signs were typed where they belong.
UNDEF'D FUNCTION	A user defined function was referenced, but it has never been defined using the DEFFN statement.	Define the function with DEF within the program.
UNDEF'D STATEMENT	An attempt was made to GOTO or GOSUB or RUN a line number that doesn't exist.	Make sure line numbers exist.
VERIFY	The program on tape or disk does not match the program currently in memory.	Save the program again, under another name.

**NOTE:** A common error is to type a 41-character line, not hit RETURN and type a second line as if it were a new line. RETURN will then treat both lines as one. To find this type of error, list your program and continue hitting RETURN. Watch the cursor jump to the beginning of each instruction line. A skipped line means it was tagged onto the line above it. Retype these lines.

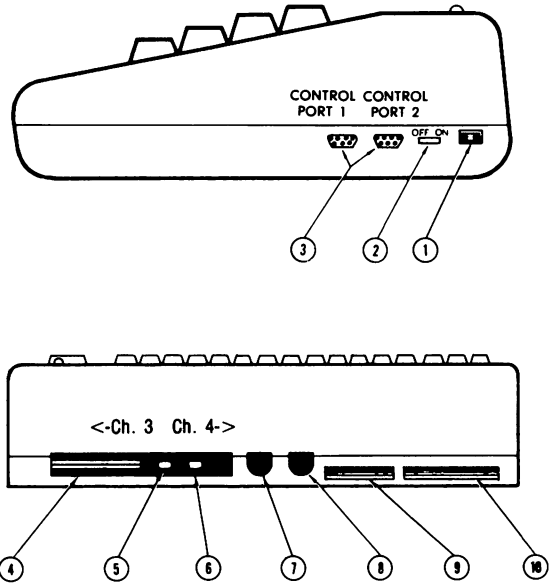
CCCCCCCCCCCCCCCCCC



## APPENDIX B

### CONNECTORS/ PORTS FOR PERIPHERAL EQUIPMENT

## COMMODORE CONNECTIONS FOR PERIPHERALS



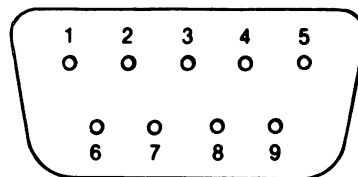
- |                     |                      |
|---------------------|----------------------|
| 1. Power Socket     | 6. RF (TV) Connector |
| 2. Power Switch     | 7. Video Port        |
| 3. Control Ports    | 8. Serial Port       |
| 4. Expansion Port   | 9. Cassette Port     |
| 5. Channel Selector | 10. User Port        |

## Side Panel Connections

1. Power Socket—The free end of the cable from the power supply is attached here.
2. Power Switch—Turns on power from the transformer.
3. Control Ports—There are two Control ports, numbered 1 and 2. Each Control port can accept a joystick or game controller paddle. A light pen or mouse can be plugged only into port 1, the port closest to the front of the computer. Use the ports as instructed with the software.

### Control Port 1

Pin	Type	Note
1	JOYA0	
2	JOYA1	
3	JOYA2	
4	JOYA3	
5	POT AY	
6	BUTTON A/LP	
7	+5V	MAX. 50mA
8	GND	
9	POT AX	



(front view of port)

### 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	

## Rear Connections

4. Expansion Port—This rectangular slot is a parallel port that accepts program or game cartridges as well as special interfaces.

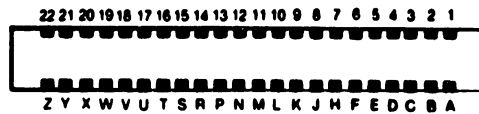
### Cartridge Expansion Slot

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

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

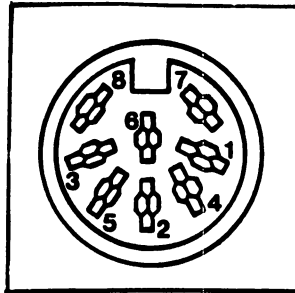
Pin	Type
A	GND
B	ROMH
C	RESET
D	NMI
E	S 02
F	A15
H	A14
J	A13
K	A12
L	A11
M	A10



(view of port while facing the rear of the 64C)

5. Channel Selector—Use this switch to select which TV channel (L = channel 3, H = channel 4) the computer's picture will be displayed on when using a television instead of a monitor.
6. RF Connector—This connector supplies both picture and sound to your television set.

7. Video Port—This DIN connector supplies direct audio and composite video signals. These can be connected to the Commodore monitor or used with separate components.



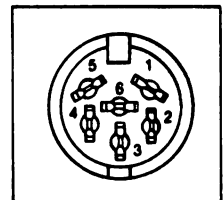
(view of port while facing the rear of the 64C)

Pin	Type	Note
1	LUM/SYNC	Luminance/SYNC output
2	GND	
3	AUDIO OUT	
4	VIDEO OUT	Composite signal output
5	AUDIO IN	
6	COLOR OUT	Chroma signal output
7	NC	No connection
8	NC	No connection

8. Serial Port—A Commodore serial printer or disk drive can be attached directly to the Commodore 64C through this port.

**Serial I/O**

Pin	Type
1	SERIAL SRQIN
2	GND
3	SERIAL ATN IN/OUT
4	SERIAL CLK IN/OUT
5	SERIAL DATA IN/OUT
6	RESET

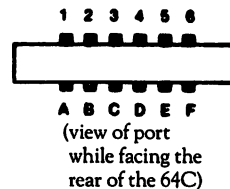


(view of port while facing the rear of the 64C)

9. **Cassette Port**—A 1530 Datasette recorder can be attached here to store programs and information.

**Cassette**

Pin	Type
A-1	GND
B-2	+5V
C-3	CASSETTE MOTOR
D-4	CASSETTE READ
E-5	CASSETTE WRITE
F-6	CASSETTE SENSE

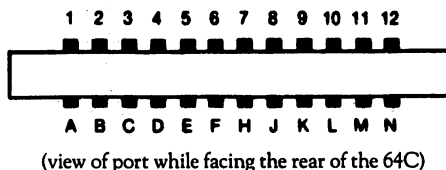


10. **User Port**—Various interface devices can be attached here, including a Commodore modem.

**User I/O**

Pin	Type	Note
1	GND	
2	+5V	MAX. 100 mA
3	RESET	
4	CNT1	
5	SP1	
6	CNT2	
7	SP2	
8	PC2	
9	SER. ATN IN	
10	9 VAC	MAX. 100 mA
11	9 VAC	MAX. 100 mA
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  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100

## APPENDIX C

### SCREEN DISPLAY CODES

The following chart lists all of the characters built into the Commodore screen character sets. It shows which numbers should be POKEd into the VIC chip (40 column) screen memory (location 1024 to 2023) to get a desired character on the 40-column screen. (Remember, to set color memory, use locations 55296 to 56295.) Also shown is which character corresponds to a number PEEKed from the screen.

Two character sets are available, but only one is available at a time. The sets are switched by holding down the SHIFT and **☐** (Commodore) keys simultaneously. The entire screen of characters changes to the selected character set.

From BASIC, PRINT CHR\$(142) will switch to upper-case/graphics mode and PRINT CHR\$(14) will switch to upper/lower-case mode.

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.

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	Q	q	17	"		34
A	a	1	R	r	18	#		35
B	b	2	S	s	19	\$		36
C	c	3	T	t	20	%		37
D	d	4	U	u	21	&		38
E	e	5	V	v	22	,		39
F	f	6	W	w	23	(		40
G	g	7	X	x	24	)		41
H	h	8	Y	y	25	.		42
I	i	9	Z	z	26	+		43
J	j	10	[		27	,		44
K	k	11	£		28	-		45
L	l	12	]		29	.		46
M	m	13	↑		30	/		47
N	n	14	←		31	0		48
O	o	15	SPACE		32	1		49
P	p	16	!		33	2		50

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
3		51		M	77			103
4		52		N	78			104
5		53		O	79			105
6		54		P	80			106
7		55		Q	81			107
8		56		R	82			108
9		57		S	83			109
:		58		T	84			110
:		59		U	85			111
<		60		V	86			112
=		61		W	87			113
>		62		X	88			114
?		63		Y	89			115
		64		Z	90			116
	A	65			91			117
	B	66			92			118
	C	67			93			119
	D	68			94			120
	E	69			95			121
	F	70	<b>SPACE</b>		96			122
	G	71			97			123
	H	72			98			124
	I	73			99			125
	J	74			100			126
	K	75			101			127
	L	76			102			






















Codes from 128-255 are reversed images of codes 0-127.











APPENDIX D

ASCII AND CHR\$ CODES

This appendix shows you what characters will appear if you PRINT CHR\$(X), for all possible values of X. It also shows the values obtained by typing PRINT ASC("X"), where X is any character that can be displayed. This is useful in evaluating the character received in a GET statement, converting upper to 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		26	4	52	N	78
	1		27	5	53	O	79
	2		28	6	54	P	80
	3		29	7	55	Q	81
	4		30	8	56	R	82
	5		31	9	57	S	83
	6		32	:	58	T	84
	7	!	33	;	59	U	85
DISABLES  8	8	"	34	<	60	V	86
ENABLES  9	9	#	35	=	61	W	87
	10	\$	36	>	62	X	88
	11	%	37	?	63	Y	89
	12	&	38	@	64	Z	90
	13	.	39	A	65	[	91
	14	(	40	B	66	\	92
	15	)	41	C	67	]	93
	16	*	42	D	68	^	94
	17	+	43	E	69	_	95
	18	,	44	F	70	~	96
	19	-	45	G	71		97
	20	.	46	H	72		98
	21	/	47	I	73		99
	22	0	48	J	74		100
	23	1	49	K	75		101
	24	2	50	L	76		102
	25	3	51	M	77		

PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS
	103		124		145		166
	104		125		146		167
	105		126		147		168
	106		127		148		169
	107		128	Brown	149		170
	108	Orange	129	Lt. Red	150		171
	109		130	Dk. Gray	151		172
	110		131	Gray	152		173
	111		132	Lt. Green	153		174
	112	f1	133	Lt. Blue	154		175
	113	f3	134	Lt. Gray	155		176
	114	f5	135		156		177
	115	f7	136		157		178
	116	f2	137		158		179
	117	f4	138		159		180
	118	f6	139		160		181
	119	f8	140		161		182
	120		141		162		183
	121		142		163		
	122		143		164		
	123		144		165		

PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS
	184		186		188		190
	185		187		189		191

<b>CODES</b>	<b>192-223</b>	<b>SAME AS</b>	<b>96-127</b>
<b>CODES</b>	<b>224-254</b>	<b>SAME AS</b>	<b>160-190</b>
<b>CODE</b>	<b>255</b>	<b>SAME AS</b>	<b>126</b>



# APPENDIX E

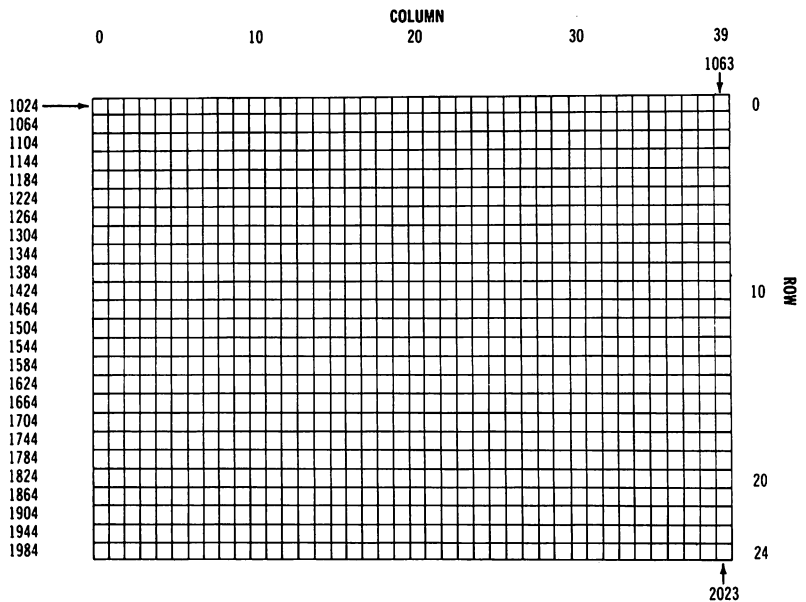
## SCREEN AND COLOR MEMORY MAPS

### Screen Memory Map

The following maps display the memory locations used in specifying the placement and color of characters on the screen. Each map is separately controlled and consists of 1,000 positions (25 lines of 40 characters each).

The characters displayed on the maps can be controlled directly with the POKE command. (Remember to POKE the colors to the color map as well.)

### VIC SCREEN MEMORY MAP



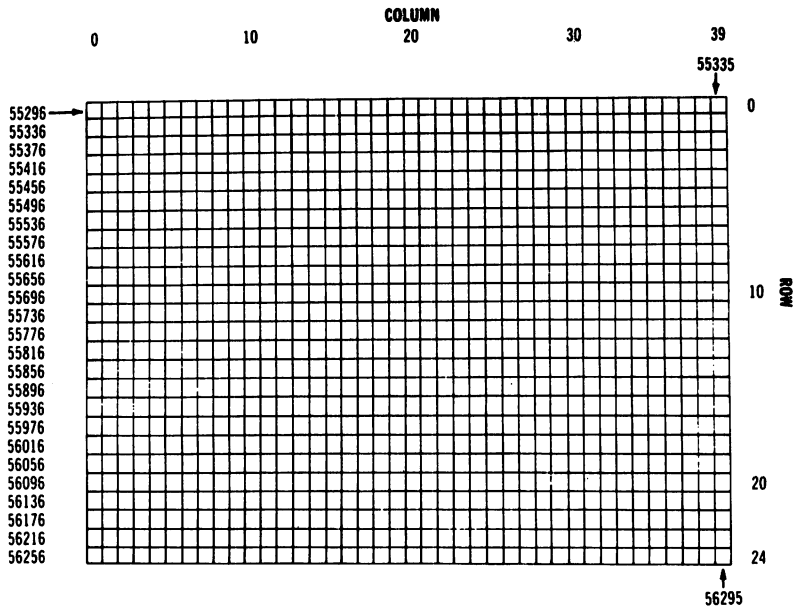
The Screen Map is POKEd with a Screen Display Code value (see Appendix C). For example:

**POKE 1024, 13**

will display the letter M in the upper-left corner of the screen.

## Color Memory Map

## VIC COLOR MEMORY MAP



The color RAM appears in this range in I/O space. If the color map is POKEd with a color value; this changes the character color. For example:

**POKE 55296, 1**

will change the letter M inserted above from light green to white.

Note: Only the lower nybble (4 bits) is used. If you PEEK color RAM, do this to determine color at location X:

**C = PEEK(X) and 15**

### Color Codes

0 Black	8 Orange
1 White	9 Brown
2 Red	10 Light Red
3 Cyan	11 Dark Gray
4 Purple	12 Medium Gray
5 Green	13 Light Green
6 Blue	14 Light Blue
7 Yellow	15 Light Gray

Border Control Memory 53280

Background Control Memory 53281

## APPENDIX F

### DERIVED TRIGONOMETRIC FUNCTIONS

FUNCTION	BASIC EQUIVALENT
SECANT	$\text{SEC}(X) = 1/\text{COS}(X)$
COSECANT	$\text{CSC}(X) = 1/\text{SIN}(X)$
COTANGENT	$\text{COT}(X) = 1/\text{TAN}(X)$
INVERSE SINE	$\text{ARCSIN}(X) = \text{ATN}(X/\text{SQR}(-X^2 + 1))$
INVERSE COSINE	$\text{ARCCOS}(X) = -\text{ATN}(X/\text{SQR}(-X^2 + 1)) + \pi/2$
INVERSE SECANT	$\text{ARCSEC}(X) = \text{ATN}(X/\text{SQR}(X^2 - 1))$
INVERSE COSECANT	$\text{ARCCSC}(X) = \text{ATN}(X/\text{SQR}(X^2 - 1)) + (\text{SGN}(X) - 1) * \pi/2$
INVERSE COTANGENT	$\text{ARCOT}(X) = \text{ATN}(X) + \pi/2$
HYPERBOLIC SINE	$\text{SINH}(X) = (\text{EXP}(X) - \text{EXP}(-X))/2$
HYPERBOLIC COSINE	$\text{COSH}(X) = (\text{EXP}(X) + \text{EXP}(-X))/2$
HYPERBOLIC TANGENT	$\text{TANH}(X) = \text{EXP}(-X)/(\text{EXP}(X) + \text{EXP}(-X))^2 + 1$
HYPERBOLIC SECANT	$\text{SECH}(X) = 2/(\text{EXP}(X) + \text{EXP}(-X))$
HYPERBOLIC COSECANT	$\text{CSCH}(X) = 2/(\text{EXP}(X) - \text{EXP}(-X))$
HYPERBOLIC COTANGENT	$\text{COTH}(X) = \text{EXP}(-X)/(\text{EXP}(X) - \text{EXP}(-X))^2 + 1$
INVERSE HYPERBOLIC SINE	$\text{ARCSINH}(X) = \text{LOG}(X + \text{SQR}(X^2 + 1))$
INVERSE HYPERBOLIC COSINE	$\text{ARCCOSH}(X) = \text{LOG}(X + \text{SQR}(X^2 - 1))$
INVERSE HYPERBOLIC TANGENT	$\text{ARCTANH}(X) = \text{LOG}((1 + X)/(1 - X))/2$
INVERSE HYPERBOLIC SECANT	$\text{ARCSECH}(X) = \text{LOG}((\text{SQR}(-X^2 + 1) + 1)/X)$
INVERSE HYPERBOLIC COSECANT	$\text{ARCCSCH}(X) = \text{LOG}((\text{SGN}(X) * \text{SQR}(X^2 + 1)/X)$
INVERSE HYPERBOLIC COTANGENT	$\text{ARCCOTH}(X) = \text{LOG}((X + 1)/(X - 1))/2$

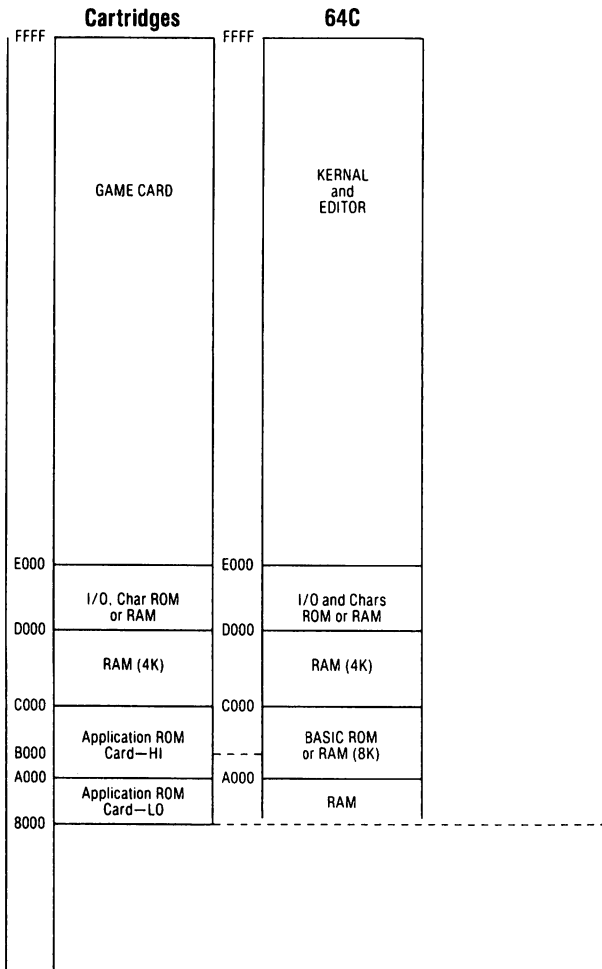
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100



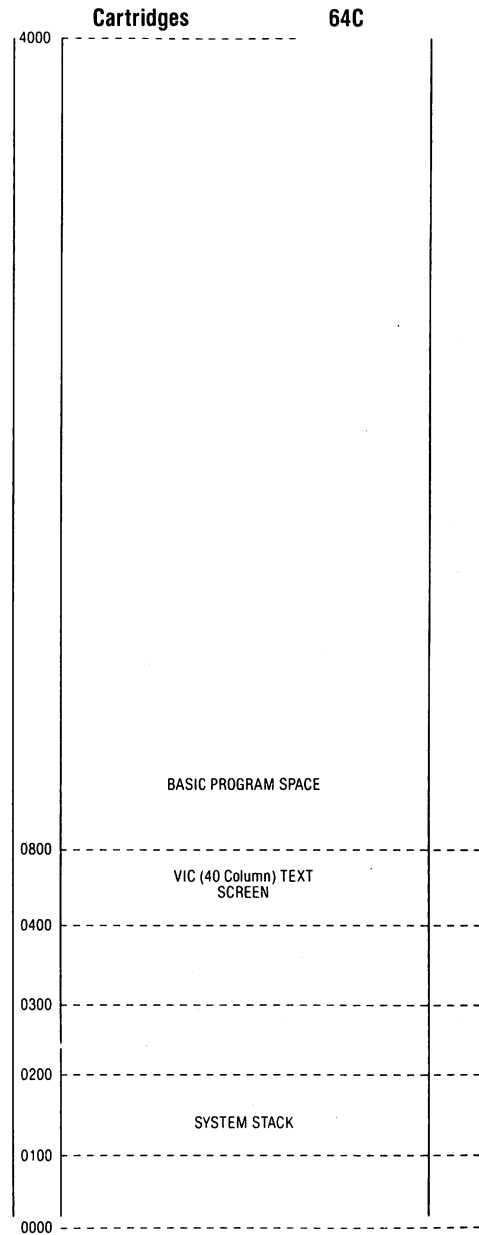
# APPENDIX G

## MEMORY MAP

### COMMODORE 64C MEMORY MAP



### COMMODORE 64C MEMORY MAP





APPENDIX H  
 BASIC 2.0  
 ABBREVIATIONS

Note: The abbreviations below operate in uppercase/graphics mode. Press the letter key(s) indicated, then hold down the SHIFT key and press the letter key following the word SHIFT.

Command	Abbreviation	Looks like this on screen	Command	Abbreviation	Looks like this on screen
ABS	A <b>SHIFT</b> B	A	LEFT\$	LE <b>SHIFT</b> F	LE
AND	A <b>SHIFT</b> N	A	LEN	NONE	LEN
ASC	A <b>SHIFT</b> S	A	LET	L <b>SHIFT</b> E	L
ATN	A <b>SHIFT</b> T	A	LIST	L <b>SHIFT</b> I	L
CHR\$	C <b>SHIFT</b> H	C	LOAD	L <b>SHIFT</b> O	L
CLOSE	CL <b>SHIFT</b> O	CL	LOG	NONE	LOG
CLR	C <b>SHIFT</b> L	C	MID\$	M <b>SHIFT</b> I	M
CMD	C <b>SHIFT</b> M	C	NEW	NONE	NEW
CONT	C <b>SHIFT</b> O	C	NEXT	N <b>SHIFT</b> E	N
COS	NONE	COS	NOT	N <b>SHIFT</b> O	N
DATA	D <b>SHIFT</b> A	D	ON	NONE	ON
DEF	D <b>SHIFT</b> E	D	OPEN	O <b>SHIFT</b> P	O
DIM	D <b>SHIFT</b> I	D	OR	NONE	OR
END	E <b>SHIFT</b> N	E	PEEK	P <b>SHIFT</b> E	P
EXP	E <b>SHIFT</b> X	E	POKE	P <b>SHIFT</b> O	P
FN	NONE	FN	POS	NONE	POS
FOR	F <b>SHIFT</b> O	F	PRINT	?	?
FRE	F <b>SHIFT</b> R	F	PRINT#	P <b>SHIFT</b> R	P
GET	G <b>SHIFT</b> E	G	READ	R <b>SHIFT</b> E	R
GET#	NONE	GET#	REM	NONE	REM
GOSUB	GO <b>SHIFT</b> S	GO	RESTORE	RE <b>SHIFT</b> S	RE
GOTO	G <b>SHIFT</b> O	G	RETURN	RE <b>SHIFT</b> T	RE
IF	NONE	IF	RIGHT\$	R <b>SHIFT</b> I	R
INPUT	NONE	INPUT	RND	R <b>SHIFT</b> N	R
INPUT#	I <b>SHIFT</b> N	I	RUN	R <b>SHIFT</b> U	R
INT	NONE	INT	SAVE	S <b>SHIFT</b> A	S

Com-mand	Abbrevi- ation	Looks like this on screen	Com-mand	Abbrevi- ation	Looks like this on screen
SGN	S <b>SHIFT</b> G	S	TAB(	T <b>SHIFT</b> A	T
SIN	S <b>SHIFT</b> I	S	TAN	NONE	TAN
SPC(	S <b>SHIFT</b> P	S	THEN	T <b>SHIFT</b> H	T
SQR	S <b>SHIFT</b> Q	S	TIME	TI	TI
STATUS	ST	ST	TIME\$	TI\$	TI\$
STEP	ST <b>SHIFT</b> E	ST	USR	U <b>SHIFT</b> S	U
STOP	S <b>SHIFT</b> T	S	VAL	V <b>SHIFT</b> A	V
STR\$	ST <b>SHIFT</b> R	ST	VERIFY	V <b>SHIFT</b> E	V
SYS	S <b>SHIFT</b> Y	S	WAIT	W <b>SHIFT</b> A	W

# APPENDIX I

## SPRITE REGISTER MAP

Register #		DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
Dec	Hex									
0	0	S0X7							S0X0	SPRITE 0 X Component
1	1	S0Y7							S0Y0	SPRITE 0 Y Component
2	2	S1X7							S1X0	SPRITE 1 X
3	3	S1Y7							S1Y0	SPRITE 1 Y
4	4	S2X7							S2X0	SPRITE 2 X
5	5	S2Y7							S2Y0	SPRITE 2 Y
6	6	S3X7							S3X0	SPRITE 3 X
7	7	S3Y7							S3Y0	SPRITE 3 Y
8	8	S4X7							S4X0	SPRITE 4 X
9	9	S4Y7							S4Y0	SPRITE 4 Y
10	A	S5X7							S5X0	SPRITE 5 X
11	B	S5Y7							S5Y0	SPRITE 5 Y
12	C	S6X7							S6X0	SPRITE 6 X
13	D	S6Y7							S6Y0	SPRITE 6 Y
14	E	S7X7							S7X0	SPRITE 7 X Component
15	F	S7Y7							S7Y0	SPRITE 7 Y Component
16	10	S7X8	S6X8	S5X8	S4X8	S3X8	S2X8	S1X8	S0X8	MSB of X COORD.
17	11	RC8	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	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
21	15	SE7							SE0	SPRITE ENABLE (ON/OFF)
22	16	N.C.	N.C.	RST	MCM	CSEL	XSCL2	XSCL1	XSCLO	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.	LPIRQ	ISSC	ISBC	RIRQ	Interrupt Requests
26	1A	N.C.	N.C.	N.C.	N.C.	MLPI	MISSC	MISBC	MRIRQ	Interrupt Request MASKS
27	1B	BSP7							BSP0	Background- Sprite PRIORITY
28	1C	SCM7							SCM0	MULTICOLOR SPRITE SELECT
29	1D	SEXX7							SEXX0	SPRITE EXPAND X
30	1E	SSC7							SSC0	Sprite-Sprite COLLISION
31	1F	SBC7							SBC0	Sprite- Background COLLISION

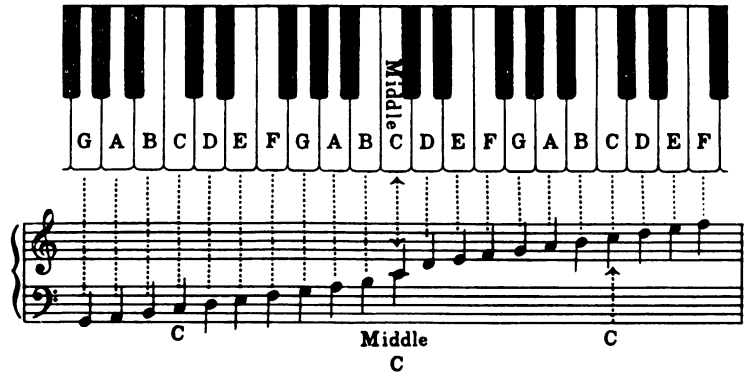
Register # Dec	Hex	Color
32	20	BORDER COLOR
33	21	BACKGROUND COLOR 0
34	22	BACKGROUND COLOR 1
35	23	BACKGROUND COLOR 2
36	24	BACKGROUND COLOR 3
37	25	SPRITE MULTICOLOR 0
38	26	SPRITE MULTICOLOR 1

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

**APPENDIX J**  
**SOUND AND**  
**MUSIC**

**Music Note Table**

Note values are POKEd into two memory locations 54272 and 54273, also known as registers or switches 0 and 1 respectively.  
 POKE the value N1 (the HIGH value) into Register 1 (location 54273) and the value N2 (the LOW value) into Register 0 (location 54272).  
 The list below covers three octaves of notes for the Bass and Treble Clefs. For the full list of note values, see the *Commodore 64 Programmer's Reference Guide*.



**TABLE OF NOTE VALUES**

NOTE	N1	N2
G	6	36
G#	6	130
A	6	228
A#	7	77
B	7	189
C	8	50
C#	8	175
D	9	51
D#	9	191
E	10	84
F	10	241
F#	11	152
G	12	73
G#	13	4
A	13	201
A#	14	156
B	15	122
*C	16	101
C#	17	96
D	18	104
D#	19	128
E	20	169

\* MIDDLE C

F	21	227
F#	23	49
G	24	146
G#	26	8
A	27	148
A#	29	57
B	30	245
C	32	204
C#	34	192
D	36	208
D#	39	1
E	41	83
F	43	200
F#	46	99

## Sound Control Settings

Each sound parameter is POKEd into a register of the specialized sound generating chip. Each register is a memory location (called byte address) starting with 54272.

Each sound has a characteristic ADSR consisting of the following four parameters: Attack, Decay, Sustain, Release.

**Attack** is the rate sound rises to maximum volume. It can vary from a 2-millisecond cycle to an 8-second cycle. The corresponding register value is 0 to 15.

**Decay** is the rate sound falls from maximum volume to sustain level. This varies from a 6-millisecond cycle to 24 seconds, corresponding with 0 to 15.

The values of Attack and Decay are POKEd together into register 5 by a single number derived by multiplying the ATTACK value by 16 and adding the DECAY value.

**Sustain** is the amplitude level at which the sound is held, varying from 0% to 100% of maximum level corresponding to register values of 0 to 15.

**Release** is the rate at which volume falls from the sustain level to zero; similar in timing to the decay rate.

Sustain and Release are POKEd into register 6 together as one number derived by multiplying SUSTAIN by 16 and adding the RELEASE value.

**Waveform** is the shape of the sound wave produced. The waveforms called Triangle, Sawtooth and Pulse are related to the sound of musical instruments. Noise is a randomized waveform. Only specific register values will activate this characteristic of sound.

**Pulse** is the tonal quality of the Pulse waveform. Thus, whenever register 4 is activated with a 65, a value other than zero must be POKEd into either register 2 or 3 for the Pulse Rate.



**Frequency** is the vibratory level of sound which distinguishes one note from another. Concert A is 440 cycles per second. Registers 0 and 1 are required to define the frequency. 256 times the value in Register 1 plus the value of Register 0 is the sound generator's oscillator frequency. This is directly proportional to the sound frequency.

Following is a table of values which can be POKEd into these registers. The actual memory location is 54272 plus the register number.

Voice 1	Register	Voice 2	Voice 3	Description	Range of Values
0	7	14	14	frequency	0 to 255
1	8	15	15	frequency	0 to 255
2	9	16	16	pulse	0 to 255
3	10	17	17	pulse	0 to 15
4	11	18	18	Waveform	16,32,64,128 17,33,65,129
5	12	19	19	Attack/Decay	0 to 255
6	13	20	20	Sustain/Release	0 to 255
All voices					
		21		Filter-low cutoff	0 to 7
		22		Filter-high cutoff	0 to 255
		23		Resonance	16,32,64,128 or any sum
		23		Filter switch/ voice	1,2 or 4
		24		Volume	0 to 15

See the *Commodore 64 Programmer's Reference Guide* for more details on the SID sound chip.



# GLOSSARY

## GLOSSARY

This glossary provides brief definitions of frequently used computing terms.

**Acoustic Coupler or Acoustic Modem:** A device that converts digital signals to audible tones for transmission over telephone lines. Speed is limited to about 1,200 baud, or bits per second (bps). Compare direct-connect modem.

**Address:** The label or number identifying the register or memory location where a unit of information is stored.

**Alphanumeric:** Letters, numbers and special symbols found on the keyboard, excluding graphic characters.

**ALU:** Arithmetic Logic Unit. The part of a Central Processing Unit (CPU) where binary data is acted upon.

**Animation:** The use of computer instructions to simulate motion of an object on the screen through gradual, progressive movements.

**Array:** A data-storage structure in which a series of related constants or variables are stored in consecutive memory locations. Each constant or variable contained in an array is referred to as an element. An element is accessed using a subscript. See Subscript.

**ASCII:** Acronym for American Standard Code for Information Interchange. A seven-bit code used to represent alphanumeric characters. It is useful for such things as sending information from a keyboard to the computer, and from one computer to another. See Character String Code.

**Assembler:** A program that translates assembly-language instructions into machine-language instructions.

- Assembly Language:** A machine-oriented language in which mnemonics are used to represent each machine-language instruction. Each CPU has its own specific assembly language. See CPU and machine language.
- Assignment Statement:** A BASIC statement that sets a variable, constant or array element to a specific numeric or string value.
- Asynchronous Transmission:** A scheme in which data characters are sent at random time intervals. Limits phone-line transmission to about 2,400 baud (bps). See Synchronous Transmission.
- Attack:** The rate at which the volume of a musical note rises from zero to peak volume.
- Background Color:** The color of the portion of the screen that the characters are placed upon.
- BASIC:** Acronym for Beginner's All-purpose Symbolic Instruction Code.
- Baud:** Serial-data transmission speed. Originally a telegraph term, 300 baud is approximately equal to a transmission speed of 30 bytes or characters per second.
- Binary:** A base-2 number system. All numbers are represented as a sequence of zeros and ones.
- Bit:** The abbreviation for Binary digIT. A bit is the smallest unit in a computer. Each binary digit can have one of two values, zero or one. A bit is referred to as enabled or "on" if it equals one. A bit is disabled or "off" if it equals zero.
- Bit Control:** A means of transmitting serial data in which each bit has a significant meaning and a single character is surrounded with start and stop bits.
- Bit Map Mode:** An advanced graphic mode in the Commodore 128 in which you can control every dot on the screen.
- Border Color:** The color of the edges around the screen.
- Branch:** To jump to a section of a program and execute it. GOTO and GOSUB are examples of BASIC branch instructions.

- Bus:** Parallel or serial lines used to transfer signals between devices. Computers are often described by their bus structure.
- Bus Network:** A system in which all stations or computer devices communicate by using a common distribution channel or bus.
- Byte:** A group of eight bits that make up the smallest unit of addressable storage in a computer. Each memory location in the Commodore 64C contains one byte of information. One byte is the unit of storage needed to represent one character in memory. See Bit.
- Carrier Frequency:** A constant signal transmitted between communicating devices that is modulated to encode binary information.
- Character:** Any symbol on the computer keyboard that is printed on the screen. Characters include numbers, letters, punctuation and graphic symbols.
- Character Memory:** The area in the Commodore 64C's memory which stores the encoded character patterns that are displayed on the screen.
- Character Set:** A group of related characters. The Commodore 64C character sets consist of: upper-case letters, lower-case letters and graphic characters.
- Character String Code:** The numeric value assigned to represent a Commodore 64C character in the computer's memory.
- Chip:** A miniature electronic circuit that performs a computer operation such as graphics, sound and input/output.
- Clock:** The timing circuit for a microprocessor.
- Clocking:** A technique used to synchronize a sending and a receiving data-communications device that is modulated to encode binary information.
- Coaxial Cable:** A transmission medium, usually employed in local networks.
- Collision Detection:** Determination of occurrence of collision between two or more sprites, or between sprites and data.

- Color Memory:** The area in the Commodore 64C's memory that controls the color of each location in screen memory.
- Command:** A BASIC instruction used in direct mode to perform an action. See Direct Mode.
- Compiler:** A program that translates a high-level language, such as BASIC, into machine language.
- Composite Monitor:** A device used to provide a 40-column video display.
- Computer:** An electronic, digital device that stores and processes information.
- Condition:** Expression(s) between the words IF and THEN, evaluated as either true or false in an IF . . . THEN statement. The condition IF . . . THEN statement gives the computer the ability to make decisions.
- Coordinate:** A single point on a grid having vertical (Y) and horizontal (X) values.
- Counter:** A variable used to keep track of the number of times an event has occurred in a program.
- CPU:** Acronym for Central Processing Unit. The part of the computer containing the circuits that control and perform the execution of computer instructions.
- Crunch:** To minimize the amount of computer memory used to store a program.
- Cursor:** The flashing square that marks the current location on the screen.
- Data:** Numbers, letters or symbols that are input into the computer to be processed.
- Data Base:** A large amount of data stored in a well-organized manner. A data-base management system is a program that allows access to the information.
- Data Rate or Data Transfer Rate:** The speed at which data is sent to a receiving computer—given in baud, or bits per second (bps).

**Datassette:** A Commodore device used to store programs and data files sequentially on tape.

**Debug:** To correct errors in a program.

**Decay:** The rate at which the volume of a musical note decreases from its peak value to a mid-range volume called the sustain level. See Sustain.

**Decrement:** To decrease an index variable or counter by a specific value.

**Delay Loop:** An empty FOR . . . NEXT loop that slows the execution of a program.

**Digital:** Of or relating to the technology of computers and data communications where all information is encoded as bits of 1s or 0s that represent on or off states.

**Dimension:** The property of an array that specifies the direction along an axis in which the array elements are stored. For example, a two-dimensional array has an X-axis for rows and a Y-axis for columns. See Array.

**Direct Connect Modem:** A device that converts digital signals from a computer into electronic impulses for transmission over telephone lines. Contrast with Acoustic Coupler.

**Direct Mode:** The mode of operation that executes BASIC commands immediately after the RETURN key is pressed. Also called Immediate Mode. See Command.

**Disable:** To turn off a bit, byte or specific operation of the computer.

**Disk Drive:** A random access, mass-storage device that saves and loads files to and from a floppy diskette.

**Disk Operating System:** Program used to transfer information to and from a disk. Often referred to as a DOS.

**Duration:** The length of time a musical note is played.

**Electronic Mail or E-Mail:** A communications service for computer users where textual messages are sent to a central computer, or electronic "mail box," and later retrieved by the addressee.

**Enable:** To turn on a bit, byte or specific operation of the computer.

**Envelope Generator:** Portion of the Commodore 64C that produces specific waveforms (sawtooth, triangle, pulse width and noise) for musical notes. See *Waveform*.

**EPROM:** A PROM that can be erased by the user, usually by exposing it to ultraviolet light. See *PROM*.

**Error Checking or Error Detection:** Software routines that identify, and often correct, erroneous data.

**Execute:** To perform the specified instructions in a command or program statement.

**Expression:** A combination of constants, variables or array elements acted upon by logical, mathematical or relational operators that return a numeric value.

**File:** A program or collection of data treated as a unit and stored on disk or tape.

**Firmware:** Computer instructions stored in ROM, as in a game cartridge.

**Frequency:** The number of sound waves per second of a tone. The frequency corresponds to the pitch of the audible tone.

**Full-Duplex Mode:** Allows two computers to transmit and receive data at the same time.

**Function:** A predefined operation that returns a single value.

**Function Keys:** The four keys on the far right of the Commodore 64C keyboard. Each key can be programmed to execute a series of instructions. Since the keys can be SHIFTed, you can create eight different sets of instructions.

**GCR Format:** The abbreviation for Group Code Recording, a method of storing information on a disk in CP/M mode.

**Graphics:** Visual screen images representing computer data in memory (i.e., characters, symbols and pictures).

**Graphic Characters:** Non-alphanumeric characters on the computer's keyboard.



**Grid:** A two-dimensional matrix divided into rows and columns. Grids are used to design sprites and programmable characters.

**Half-Duplex Mode:** Allows transmission in only one direction at a time; if one device is sending, the other must simply receive data until it's time for it to transmit.

**Hardware:** Physical components in a computer system such as keyboard, disk drive and printer.

**Hexadecimal:** Refers to the base-16 number system. Machine language programs are often written in hexadecimal notation.

**Home:** The upper-left corner of the screen.

**IC:** Integrated Circuit. A silicon chip containing an electric circuit made up of components such as transistors, diodes, resistors and capacitors. Integrated circuits are smaller, faster and more efficient than the individual circuits used in older computers.

**Increment:** To increase an index variable or counter with a specified value.

**Index:** The variable counter within a FOR . . .NEXT loop.

**Input:** Data fed into the computer to be processed. Input sources include the keyboard, disk drive, Datassette or modem.

**Integer:** A whole number (i.e., a number containing no fractional part), such as 0, 1, 2, etc.

**Interface:** The point of meeting between a computer and an external entity, whether an operator, a peripheral device or a communications medium. An interface may be physical, involving a connector, or logical, involving software.

**I/O:** Input/output. Refers to the process of entering data into the computer, or transferring data from the computer to a disk drive, printer or storage medium.

**Keyboard:** Input component of a computer system.

**Kilobyte (K):** 1,024 bytes.

**Loop:** A program segment executed repetitively a specified number of times.

**Machine Language:** The lowest level language the computer understands. The computer converts all high-level languages, such as BASIC, into machine language before executing any statements. Machine language is written in binary form that a computer can execute directly. Also called machine code or object code.

**Matrix:** A two-dimensional rectangle with row and column values.

**Memory:** Storage locations inside the computer. ROM and RAM are two different types of memory.

**Memory Location:** A specific storage address in the computer. There are 65,536 memory locations (0-65535) in the Commodore 64C.

**MFM:** The abbreviation for Modified Frequency Modulation, a method of storing information on disks. There are a number of different MFM formats used for CP/M programs. The Commodore 1571 disk drive can read and write to many MFM formats.

**Microprocessor:** A CPU that is contained on a single integrated circuit (IC). Microprocessors used in Commodore personal computers include the 6510, the 8502 and the Z80.

**Mode:** A state of operation.

**Modem:** Acronym for MODulator/DEModulator. A device that transforms digital signals from the computer into electrical impulses for transmission over telephone lines, and does the reverse for reception.

**Monitor:** A display device resembling a television set but with a higher-resolution (sharper) image on the video screen.

**Motherboard:** In a bus-oriented system, the board that contains the bus lines and edge connectors to accommodate the other boards in the system.

**Multi-Color Character Mode:** A graphic mode that allows you to display four different colors within an 8 × 8 character grid.

**Multi-Color Bit Map Mode:** A graphic mode that allows you to display one of four colors for each pixel within an 8 × 8 character grid. See Pixel.

**Multiple-Access Network:** A flexible system by which every station can have access to the network at all times; provisions are made for times when two computers decide to transmit at the same time.

**Null String:** An empty character (""). A character that is not yet assigned a character string code.

**Octave:** One full series of eight notes on the musical scale.

**Operating System:** A built-in program that controls everything your computer does.

**Operator:** A symbol that tells the computer to perform a mathematical, logical or relational operation on the specified variables, constants or array elements in the expression. The mathematical operators are +, -, \*, / and  $\uparrow$ . The relational operators are <, =, >, < =, > = and < >. The logical operators are AND, OR NOT, and XOR.

**Order of Operations:** Sequence in which computations are performed in a mathematical expression. Also called Hierarchy of Operations.

**Parallel Port:** A port used for transmission of data one byte at a time over multiple wires.

**Parity Bit:** A 1 or 0 added to a group of bits that identifies the sum of the bits as odd or even.

**Peripheral:** Any accessory device attached to the computer such as a disk drive, printer, modem or joystick.

**Pitch:** The highness or lowness of a tone that is determined by the frequency of the sound wave. See Frequency.

**Pixel:** Computer term for picture element. Each dot on the screen that makes up an image is called a pixel. Each character on the screen is displayed within an  $8 \times 8$  grid of pixels. The entire screen is composed of a  $320 \times 200$  pixel grid. In bit-map mode, each pixel corresponds to one bit in the computer's memory.

**Pointer:** A register used to indicate the address of a location in memory.

- Polling:** A communications control method used by some computer/terminal systems whereby a “master” station asks many devices attached to a common transmission medium, in turn, whether they have information to send.
- Port:** A channel through which data is transferred to and from the CPU.
- Printer:** Peripheral device that outputs the contents of the computer’s memory onto a sheet of paper. This paper is referred to as a hard copy.
- Program:** A series of instructions that direct the computer to perform a specific task. Programs can be stored on diskette or cassette, reside in the computer’s memory, or be listed on a printer.
- Programmable:** Capable of being processed with computer instructions.
- Program Line:** A statement or series of statements preceded by a line number in a program. The maximum length of a program line on the Commodore 64C is 80 characters.
- PROM:** Acronym for Programmable Read Only Memory. A semiconductor memory whose contents cannot be changed.
- Protocol:** The rules under which computers exchange information, including the organization of the units of data to be transferred.
- Random Access Memory (RAM):** The programmable area of the computer’s memory that can be read from and written to (changed). All RAM locations are equally accessible at any time in any order. The contents of RAM are erased when the computer is turned off.
- Random Number:** A nine-digit decimal number from 0.000000001 to 0.999999999 generated by the RaNDom (RND) function.
- Read Only Memory (ROM):** The permanent portion of the computer’s memory. The contents of ROM locations can be read, but not changed. The ROM in the Commodore 64C contains the BASIC language interpreter, character-image patterns and portions of the operating system.
- Register:** Any memory location in RAM (usually referenced to an I/O device or the microprocessor itself). Each register stores one byte.

**Release:** The rate at which the volume of a musical note decreases from the sustain level to zero.

**Remark:** Comments used to document a program. Remarks are not executed by the computer, but are displayed in the program listing.

**Resolution:** The number of addressable pixels on the screen; determines the fineness of detail of a displayed image.

**RGBI Monitor:** Red/Green/Blue/Intensity. A high-resolution display device necessary to produce an 80-column screen format.

**Ribbon Cable:** A group of attached parallel wires.

**RS-232:** A recommended standard for electronic and mechanical specifications of serial transmission ports. The Commodore 64C parallel user port can be treated as a serial port if accessed through software, sometimes with the addition of an interface device.

**Screen:** Video display unit which can be either a television or video monitor.

**Screen Code:** The number assigned to represent a character in screen memory. When you type a key on the keyboard, the screen code for that character is entered into screen memory automatically. You can also display a character by storing its screen code directly into screen memory with the POKE command.

**Screen Memory:** The area of the Commodore 64C's memory that contains the information displayed on the video screen.

**Serial Port:** A port used for serial transmission of data; bits are transmitted one bit after the other over a single wire.

**Serial Transmission:** The sending of sequentially ordered data bits.

**Software:** Computer programs (sets of instructions) stored on disk, tape or cartridge that can be loaded into random access memory. Software, in essence, tells the computer what to do.

**Sound Interface Device (SID):** The sound synthesizer chip responsible for all the audio features of the Commodore 64C. See the Commodore 64 Programmer's Reference Guide for chip specifications.

- Source Code:** A non-executable program written in a high-level language. A compiler or assembler must translate the source code into an object code (machine language) that the computer can understand.
- Sprite:** A programmable, movable, high-resolution graphic image. Also called a Movable Object Block (MOB).
- Standard Character Mode:** The mode the Commodore 64C operates in when you turn it on and when you write programs.
- Start Bit:** A bit or group of bits that identifies the beginning of a data word.
- Statement:** A BASIC instruction contained in a program line.
- Stop Bit:** A bit or group of bits that identifies the end of a data word and defines the space between data words.
- String:** An alphanumeric character or series of characters surrounded by quotation marks.
- Subroutine:** An independent program segment separate from the main program that performs a specific task. Subroutines are called from the main program with the GOSUB statement and must end with a RETURN statement.
- Subscript:** A variable or constant that refers to a specific element in an array by its position within the array.
- Sustain:** The midranged volume of a musical note.
- Synchronous Transmission:** Data communications using a synchronizing, or clocking signal between sending and receiving devices.
- Syntax:** The grammatical rules of a programming language.
- Tone:** An audible sound of specific pitch, amplitude and waveform.
- Transparent:** Describes a computer operation that does not require user intervention.
- Variable:** A unit of storage representing a changing string or numeric value. Variable names can be any length, but only the first two characters are stored by the Commodore 64C. The first character must be a letter.

**Video Interface Controller (VIC):** The chip responsible for the 40-column graphics features of the Commodore 64C. See the Commodore 64 Programmer's Reference Guide for chip specifications.

**Voice:** A sound-producing component inside the SID chip. There are three voices within the SID chip so the Commodore 64C can produce three different sounds simultaneously. Each voice consists of a tone oscillator/waveform generator, an envelope generator and an amplitude modulator.

**Waveform:** A graphic representation of the shape of a sound wave. The waveform determines some of the physical characteristics of the sound.

**Word:** Number of bits treated as a single unit by the CPU. In an eight-bit machine, the word length is eight bits.

CCCCCCCCCCCCCCCCCC



# INDEX

## A

Abbreviations—BASIC, 173-174  
ABSolute function, 61, 133  
Accessories, 5  
Addition, 28, 147  
ADSR, 178  
Animation, 73, 86  
Arrays, 52, 143  
ASC function, 60, 134, 163-165  
ASCII character codes, 163-165  
Asterisk key, 29, 147  
Attack, 178  
ATN function, 134

## B

BASIC  
    abbreviations, 173-174  
    commands, 111-133  
    language, 13, 107-147  
    math functions, 28-30, 133-141  
    numeric functions, 61, 169  
    operators, 28-30, 144, 147  
    statements, 111-133  
    string functions, 61, 140  
    variables, 31, 142  
Bit Map mode, 91-93

## C

Cartridge slot, 157  
Cassette tape recorder, 159  
Channel selector, 157  
Character Display mode, 159  
CHR\$ codes, 67, 163-165  
CHR\$ function, 60, 134  
CLR statement, 113  
CLR/HOME key, 19  
Clock, 143-144  
CLOSE statement, 35, 113  
CMD, 114  
Colon, 44  
Color  
    code display, 67  
    CHR\$ codes, 67  
    keys, 23  
    memory map, 72  
    screen and border registers, 68  
    screen codes, 69

Comma, 21  
Commodore key, 19  
Connections, 135-159  
    constants, 30  
CONT command, 62, 114  
ConTRoL key, 18  
COSine function, 169  
CuRSoR keys, 16, 23

## D

DATA statement, 50, 128  
Decay, 104  
DEFine statement, 115  
Delay loop, 70  
DELeTe key, 16  
DIMension statement, 54, 115  
Direct mode, 13  
Disk commands, 37  
Disk Directory, 37  
Disk Programs, 34-37  
Division, 29  
Displaying Graphics characters, 20  
Dollar sign, 37  
Duration, 99

## E

Editing programs, 27-28  
END statement, 43, 116  
Error messages, 20, 131  
EXPOnent function, 135  
Extended background color, 91-92

## F

File, 124  
FN function, 135  
FOR . . . NEXT statement, 44-46,  
    116  
Formatting disks, 34-35  
FRE function, 135  
Frequency, 179  
Function keys, 19

## G

Game controls and ports, 156  
GET statement, 48-49, 117  
GET# statement, 118

GOSUB statement, 56-57, 118  
GOTO statement, 25-26, 119  
Graphic keys, 20  
Graphic modes, 91-93

## H

High resolution mode, 91-93  
HOME key, 19  
Hyperbolic functions, 169

## I

IF . . . THEN statement, 43-44, 119  
INPUT statement, 47-49, 120  
INPUT#, 121  
INSErT key, 16-18  
INTEger function, 59, 136  
Integer variable, 142

## J

Joystick ports, 156  
Joysticks, 156

## K

Keyboard, 14-20

## L

LEFT\$ function, 136  
LENGth function, 136  
LET statement, 121  
LIST command, 25, 122  
LOAD command, 36  
LOADing cassette software, 36  
LOADing disk software, 36  
LOGarithm function, 137  
Loops, 44-46, 116

## M

Machine language, 131, 141  
Memory, 57-58, 70-72  
Memory maps, 71, 72, 167, 168, 171  
MID\$ function, 137  
Modem, 5, 6  
Multicolors, 23, 67-70, 91-93  
Multiplication, 29, 144, 147

Music programs, 101-104  
Musical notes, 101, 177  
Musical scale, 99, 177

## N

NEW command, 26, 123  
NEXT statement, 44-46, 116  
Noise, 178  
Null string, 49, 191  
Numeric variables, 31

## O

ON statement, 56, 124  
OPEN statement, 35, 124-125  
Operators, 144  
    arithmetic, 28, 144  
    logical, 145  
    order of, 29  
    relational, 145

## P

Paddle, 156  
Parentheses, 30, 147  
PEEK function, 57, 137  
Peripherals, 155-159  
Pi, 138  
Pixel, 77  
POKE statement, 57, 58-125  
Ports, 155-159  
POS function, 138  
PRINT statement, 21-23, 126  
Printers, 6  
PRINT#, 127  
Program, 24  
    line numbering, 24  
    mode, 13  
    music, 101  
    viewing, 25  
Programmable keys, 19  
Programmer's Reference Guide, 6  
Pulse, 178

## Q

Question mark, 21  
Quotation marks, 20, 22  
Quote mode, 20, 22, 23

## R

RAM, 171  
RaNDom function, 60, 139  
Random numbers, 60, 139  
READ statement, 50-51, 128  
Registers, 78, 100, 177, 179  
Release, 178  
REMArk statement, 128  
Reserved variables, 143  
RESTORE key, 18  
RESTORE statement, 52  
RETURN key, 14  
RETURN statement, 56  
RIGHT\$ function, 138  
ROM, 171  
RUN command, 24  
RUN/STOP key, 18

## S

SAVE command, 35, 130  
Saving programs (tape), 35  
Saving programs (disk), 35  
Screen codes, 71, 161, 162  
Screen memory map, 71, 167  
Semicolon, 21  
serial port, 158  
SGN function, 139  
Shift key, 15  
Shift lock key, 16  
SID chip, 5, 99-106, 177-179  
SINe function, 139  
Slash key, 29, 147  
Software programs,  
    loading, 36  
    saving, 34-35  
Sound effects, 104, 106  
Sound registers, 100, 177-179  
SPC function, 140  
Sprite control, 80, 91  
Sprite programming, 88  
Sprite Register Map, 175  
Sprite viewing area, 87, 88  
Sprites, 77, 91  
SQuaRe function, 61, 140  
STEP, 46, 116  
STOP statement, 62, 131

STOP key, 18  
Storing Programs, 34, 35  
String variables, 32, 61  
Strings, 32, 61  
STR\$ function, 61, 140  
Subroutine, 56  
Subscripts, 53  
Subtraction, 28, 147  
Sustain, 178  
Syntax, 111, 113  
Syntax error, 153  
SYS statement, 131

## T

TAB function, 141  
TAN function, 141  
THEN, 43, 119  
TI variable, 143-144  
TI\$ variable, 143-144

## U

Up arrow key, 29  
Upper case/graphic mode, 14  
Upper/lower case mode, 14  
User port, 159  
USR function, 141

## V

VALue function, 61, 142  
Variables, 142  
    array, 52  
    dimensions, 54  
    floating point, 142  
    integer, 142  
    numeric, 31  
    string (\$), 32  
VERIFY command, 37, 132  
VIC chip, 77  
Voice, 99-104, 177-179

## W

WAIT command, 132  
Waveform, 198

## T

TAB function,  
TAN function,  
Telecommunications,  
THEN,  
TI variable,  
TI\$ variable,  
Trackball,  
Troubleshooting chart,

## U

Up arrow key,  
Upper case/graphic mode,  
Upper/Lower Case mode,  
User groups,  
User port,  
USR function,

## V

VALue function,  
Variables  
    array,  
    dimensions,  
    floating point,  
    integer,  
    numeric,  
    string (\$),  
VERIFY command,  
VIC chip,  
Voice,

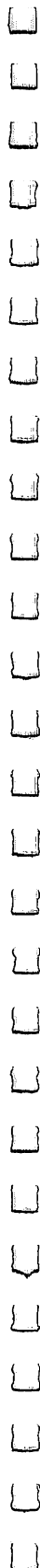
## W

WAIT command,  
Waveform,



NOTES

---





NOTES

---







NOTES

---



# GET FREE SOFTWARE WHEN YOU SUBSCRIBE TO COMMODORE MAGAZINE.

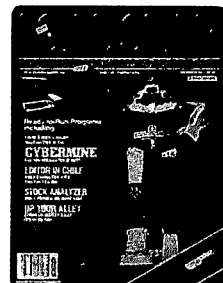
You'll find unbiased in-depth reviews of the best new software and hardware . . . discover how to make your own programs . . . learn practical home and business applications. And much, much more.

Plus, in every issue, you'll receive free programs—both games and practical applications—you can type in and use right away!

And, if you act now we'll send you a FREE "Best of Loadstar" disk . . . full of great games, practical programs, plus utilities, graphics, music, tutorials and much more!

Subscribe or renew your subscription now at the low rate of \$26.95 and we'll send you a full year of *Commodore Magazine* (12 issues, total) PLUS your FREE "Best of Loadstar" disk (\$6.95 value).

To order call toll free 800-345-8112. In Pennsylvania call 800-662-2444.



**Yes,** *I want to save 10% off the basic subscription rate of \$30.00 for Commodore Magazine and receive the "Best of Loadstar" Disk, FREE.*

Name \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip \_\_\_\_\_

**METHOD OF PAYMENT**

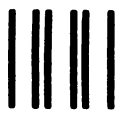
- Enclosed is my check or money order for \$26.95 (Make payable to COMMODORE PUBLICATIONS)
- Bill me
- Charge my VISA or MasterCard Card number

\_\_\_\_\_

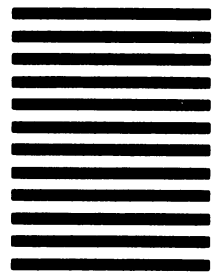
Signature \_\_\_\_\_

Expiration Date \_\_\_\_\_





NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES



**BUSINESS REPLY MAIL**  
FIRST CLASS PERMIT NO. 251 HOLMES, PA  
POSTAGE WILL BE PAID BY ADDRESSEE

**Commodore Publications**  
Magazine Subscription Department  
Box 651  
Holmes, PA 19043



**COMMODORE** 

Commodore Business Machines, Inc.  
1200 Wilson Drive • West Chester, PA 19380

Commodore Business Machines, Ltd.  
3470 Pharmacy Avenue • Agincourt, Ontario, M1W 3G3

327974-04

U-1

Printed in USA