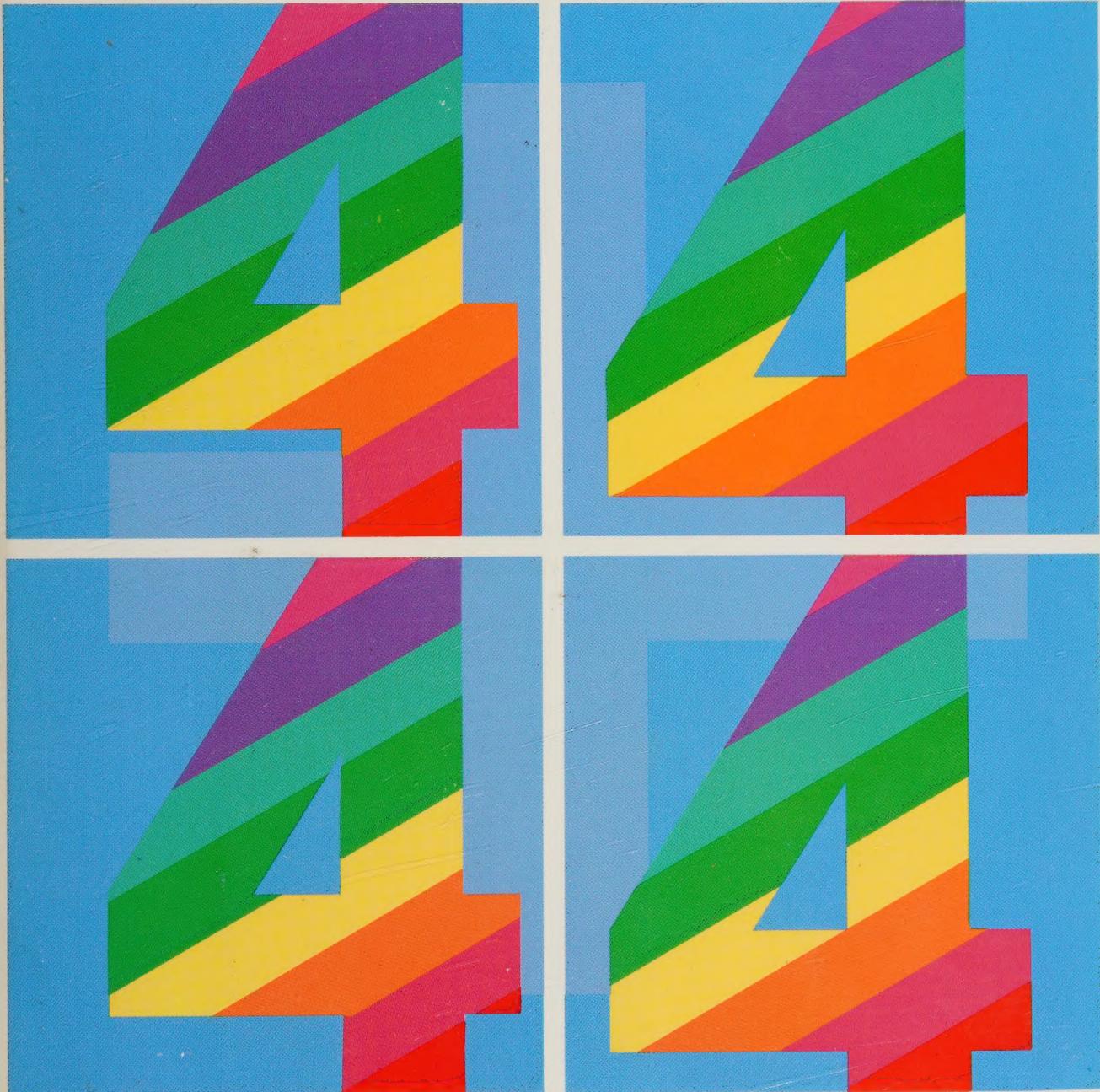


# **Programmer's Reference Guide for the Commodore Plus /4**



**Cyndie Merten • Sarah Meyer**

REF

QA  
76.8  
.C65  
M47  
1986

Business/Science/Technology  
Division ✓

DISCARD



Digitized by the Internet Archive  
in 2022 with funding from  
Kahle/Austin Foundation

<https://archive.org/details/programmersrefer0000mert>

CHICAGO PUBLIC LIBRARY



R00535 07598

# Programmer's Reference Guide for the Commodore Plus/4



---

# Programmer's Reference Guide for the Commodore Plus/4

Cyndie Merten  
Sarah Meyer

---



SCOTT, FORESMAN AND COMPANY  
Glenview, Illinois      London

Graphics characters that appear in Table 3-1 and Appendixes C and E are used with permission of Commodore Business Machines, Inc.

Copyright © 1986 Cyndie Merten and Sarah Meyer.

All Rights Reserved.

Printed in the United States of America.

ISBN 0-673-18249-5

**Library of Congress Cataloging-in-Publication Data**

Merten, Cyndie.

Programmer's reference guide for the Commodore Plus/4.

Includes index.

1. Commodore Plus/4 (Computer)—Programming.
2. BASIC (Computer program language) I. Meyer, Sarah C.

II. Title.

QA76.8.C65M47 1986 005.2'65 85-18409

ISBN 0-673-18249-5

1 2 3 4 5 6-RRC-90 89 88 87 86 85

The following are trademarks of Commodore Business Machines, Inc.: Commodore and the Commodore logo, Commodore Plus/4, Commodore 16, Commodore 64, VIC-20, VIC-1541, 1531 Datasette, C2N/1530 Datasette, Modem/300 Model 1660, MPS-801, Joystick T-1341, VIC-1526, VIC Modem 1600, Automodem 1650. The following is a registered trademark of Parker Brothers: Boggle.

**Notice of Liability**

The information in this book is distributed on an "As Is" basis, without warranty. Neither the author nor Scott, Foresman and Company shall have any liability to customer or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by the programs contained herein. This includes, but is not limited to, interruption of service, loss of data, loss of business or anticipatory profits, or consequential damages from the use of the programs.

# Preface

The Commodore Plus/4 represents an important advance in home computer design. The low-priced Plus/4, which Commodore refers to as its productivity computer, includes significant improvements over the phenomenally popular Commodore 64 and over any other computer in the home computer class. The built-in features include an expanded version of BASIC (Version 3.5), a machine-language monitor, graphic-drawing commands, improved disk- and error-handling commands, and integrated software that combines three programs:

- A word processor
- A spread sheet, with a graph generator linked to it
- A file manager

The built-in programs are accessed by a function key.

The Plus/4 has 64 K RAM built in, 60671 bytes of which are available for use in BASIC. The Plus/4 also has eight defined function keys that are easy to redefine with the KEY command. Escape key functions simplify screen editing tasks and let you create screen windows. The Plus/4 also has simple color settings that let you select from 121 different hues. In addition, the graphics modes let you use the drawing commands to draw pictures in high-resolution or multicolor modes. You can also select split-screen graphics modes that display regular text in a five-line screen window while the top of the screen is in a graphic mode. Graphic handling is much easier in BASIC Version 3.5 than it is in the Version 2.0 built into the Commodore 64.

Although the Plus/4 is superior to the Commodore 64, it does have one disadvantage: a full library of software is not yet available for the Plus/4. In addition, the Plus/4 does not have sprite graphics, which are available on the Commodore 64, and the Plus/4 music features are not as sophisticated, although music is easier to program.

When Commodore introduced the Plus/4, it was called the Commodore 264. The name was changed to Plus/4 when Commodore decided to include the built-in integrated software. At the time of the name change, Commodore also

announced the Commodore 16, which is compatible with the Plus/4. The Commodore 16 has only 16 K RAM and no built-in software. The two new computers are compatible, so all Commodore 16 software and peripherals are compatible with the Plus/4.

## **About This Book**

The Programmer's Reference Guide for the Commodore Plus/4 is a reference book for programmers of all levels. The book provides information for both BASIC and machine language programmers. The authors assume that readers are familiar with the general operations of the Plus/4 and understand all the keyboard functions. Neither BASIC nor machine language is taught in this book, but extensive information is provided on programming in both languages. Programmers of either BASIC or machine language will find the information they need to write programs for themselves or for commercial distribution.

The authors have written and tested all the programs in this book. (Please note that the programs are copyrighted and cannot be used for commercial purposes.) Cyndie Merten, programmer and mathematician, is a founding member of Dyadic Software Associates, a group of microcomputer consultants. Sarah Meyer is a free-lance technical writer who has published another book about the Plus/4. Together they have published several articles about Commodore computers. The authors combine their perspectives as programmer and writer to produce a book that is thorough, technically accurate, and clearly written. Please note that Commodore Business Machines, Inc. has not been involved in the preparation of this book. The authors bear responsibility for the accuracy of the material presented here.

The Programmer's Reference Guide for the Commodore Plus/4 is divided into six chapters. The chapters cover BASIC, the built-in software, programming techniques, machine language, graphics, and peripheral devices. Memory maps and other technical information are covered in the appendixes.

Chapter 1, The BASIC Language, provides complete descriptions of all 75+ commands, 36 functions, and the system variables that constitute BASIC 3.5. To simplify looking up BASIC keywords, the elements of BASIC 3.5 are presented in alphabetical order, with commands, functions, and system variables intermixed. For each keyword, the following information is given:

1. The abbreviation (when there is one).
2. A complete syntax, so you can quickly review the order of parameters.
3. A description of all uses for the command or function.
4. An explanation and range of possible values for each parameter.
5. Examples.

Graphics commands are given additional coverage in Chapter 4, Programming Graphics. Commands for controlling peripherals are also discussed in Chapter 6, Using Peripheral Devices. Chapter 3, Some Programming Techniques, also provides more information on BASIC commands.

The commands for use in the built-in programs are explained in Chapter 2, The Built-In Software. Chapter 2 is divided into four sections: word processor commands, commands for formatting printed output, spreadsheet commands (including commands for controlling the graph generator), and file manager commands. Within each section, commands are explained in alphabetical order. Examples are given where appropriate.

Chapter 3, Some Programming Techniques, is a collection of sections on diverse programming topics. Both BASIC and machine-language programming techniques are discussed. Sections include coverage of the following topics:

Editing the screen

Using the Escape key screen-editing functions

Using screen windows

Using text strings

Redefining the function keys

Using mathematical functions

Programming sound and music

Using arrays

Error handling

Chapter 4, Programming Graphics, explains the operations of the graphics modes in both BASIC and machine language. Color and screen control, drawing commands, and animation are among the topics discussed in this chapter. Many example programs are also provided.

Chapter 5, Machine Language on the Commodore Plus/4, explains the use of the built-in monitor commands and the application of 6502 machine-language programming on the Plus/4. This chapter does not teach machine language, but it does review the instruction set and describe the operating system for machine language programmers of all levels.

Chapter 6, Using Peripheral Devices, describes the operations of the disk drive, cassette recorder, printers, modem, and joystick in BASIC and machine language. Each peripheral, and the commands that control it, is explained in a separate section. Particular attention is given to disk-handling operations. Disk operating system (DOS) error messages are explained in Appendix A.

The appendixes are provided to explain additional technical information and to provide quick reference material. The six appendixes cover error messages for

BASIC and DOS errors, BASIC tokens, character string (CHR\$) codes, ASCII codes, screen display codes, a musical note chart, and memory and register maps.

The Programmer's Reference Guide for the Commodore Plus/4 also contains an extensive index that is designed to make finding information in this book quick and easy. We advise users to consult the index first when seeking specific information.

The authors have taken great care to ensure accuracy and thoroughness in the topics that are presented in this book. We cannot guarantee, however, that the book is error free. We have tried to make the book easy to use and understand, and we hope you find it helpful and instructive. We welcome your comments and corrections.

---

# Acknowledgments

The authors thank Bill Hindorff for reviewing this manual. We are grateful for his suggestions and constructive criticism.

We thank *COMMODORE Magazine* for publishing a Plus, 4 memory map in their November/December 1984 issue and Jim Butterfield for sharing his map in *Transactor* (Volume 5, Issue 5).

Also of great assistance in preparing the disk drive section of this manual was Richard Immers and Gerald G. Neufeld's book, *Inside Commodore DOS*.



---

# Contents

<b>1 The BASIC Language</b>	1
<b>2 The Built-In Software</b>	98
<b>3 Some Programming Techniques</b>	150
<b>4 Programming Graphics</b>	186
<b>5 Machine Language on the Commodore Plus/4</b>	227
<b>6 Using Peripheral Devices</b>	323
<b>APPENDIX A Error Messages</b>	395
<b>APPENDIX B BASIC Tokens</b>	406
<b>APPENDIX C CHR\$ Codes</b>	408
<b>APPENDIX D ASCII Codes</b>	416
<b>APPENDIX E Screen Display Codes</b>	418
<b>APPENDIX F Note Tables</b>	421
<b>APPENDIX G Plus/4 Memory Maps</b>	423
<b>INDEX</b>	440



# Programmer's Reference Guide for the Commodore Plus/4



---

# 1 The BASIC Language

---

This chapter contains information on each of the BASIC commands, functions, and system reserved variables. Other important details about BASIC are included in the beginning sections.

## The Elements of BASIC

The BASIC built into the Plus/4 is called Version 3.5. This version of BASIC is considerably more sophisticated than the Version 2.0 built into the Commodore 64. Version 3.5 contains about twice the number of BASIC commands and is easier to use.

This chapter explains each of the 75+ BASIC commands in Version 3.5. In addition, all BASIC functions are explained. The functions and the commands are explained together in alphabetical order. The possible parameters of all commands and functions are discussed. For some commands, such as the drawing commands, you must type a place-holder comma when you use the default value for a parameter. Be sure to note the requirements for each command.

BASIC lets you perform a large variety of tasks; despite this versatility, BASIC has very strict syntax rules. You must enter commands according to their formats and use only legal parameters. When you make a mistake, BASIC usually aborts the program and displays an error message. Appendix A explains the error messages that BASIC prints to help you diagnose your mistakes. The description of the HELP command explains how to use the HELP key to find errors in programming lines.

Note the following definitions if you are unsure of some terms:

**Keyword** A keyword is a word that is reserved as part of BASIC. Keywords include commands, parts of commands (such as TO, which is part of the FOR command), operators, function names, and certain reserved variables such as TI\$, a hardware timing value, and ER, an error-diagnosing variable. Keywords cannot be used as variable names or be embedded in variable names.

## 2 The BASIC Language

**Function** A function is a text string or numeric operation that returns a value. You can use any of the functions that are part of BASIC, and you can create your own with the DEF FN command.

**Operator** We use the term *operator* to mean a symbol or keyword (such as AND) that performs a mathematical task or compares two values. The types of operators available in BASIC are mathematical, comparison, and logical.

**Parameter** A parameter is a nonkeyword part of a BASIC command or function. Parameters usually have multiple possible values. You supply the parameter to define the way you want to use the BASIC command. Some parameters must be used in a command and many others are optional.

**Default** Some parameters have a default value, which means that a certain value is automatically used for that parameter if you do not specify some other value. To select the default value, you can generally just omit the parameter. In some commands, such as CIRCLE, you must type a placeholder comma for a default value if additional parameters follow the default. For example, to accept the default value for the color source in a CIRCLE command, type a comma in the color source position. The color source is the first parameter, so the command could look like this: CIRCLE, 160,100,60,50.

**Expression** Occasionally we will use the term *expression* to mean a number or string that can be a constant, variable, or function that results in an appropriate value.

### Constants and Variables

Constants are data values that you can use in a BASIC command. Variables are symbolic names that stand for one or more possible values in a BASIC command. For example, in the command PRINT "TOTAL:";T, the character string TOTAL is a constant and T is a variable that stands for the numeric value being printed.

BASIC 3.5 accepts three types of constants and variables:

1. Integer numbers (whole numbers)
2. Floating-point numbers (decimal numbers)
3. Character strings (text)

## Data Types

Floating-point numbers can be any type of number, whole or decimal (decimal numbers are also called *real numbers*), between 2.93873588E-39 and 1.70141183E+38, the negatives of those numbers, or zero. Floating-point numbers are stored in RAM using a 5-byte binary format.

Integer numbers can be any whole number between -32767 and 32767. (Note that you can use larger and smaller values for floating-point numbers.) Numbers with decimal parts are not accepted; they are truncated and ignored by BASIC. Integer numbers are stored in RAM in a 5-byte binary format. Numbers in integer arrays are stored as 2-byte binary numbers.

Character strings, or text strings, can be any characters in quotes, including numbers, blank spaces, and special symbols. The only keyboard character that cannot be directly included in a character string is a quotation mark. This is impossible because a quotation mark is used to begin and end strings. If you try to type a quotation mark in a string, BASIC assumes the quotation mark signifies the end of the string; any additional characters are assumed to be a variable name. For example, the command PRINT "HELLO" MOM prints HELLO 0. BASIC prints the 0 as the value for what it assumes is the variable MOM. However, a quotation mark may be used in a string with the help of the CHR\$ function. Note that a number in quotation marks is treated like any text and has no mathematical value.

BASIC discriminates between these three data types in variable form by the way you name the variable. The three variable types are shown in Table 1-1 with the symbols used to distinguish them.

Floating-point variables can stand for any type of number, whole or decimal, between 2.93873588E-39 and 1.70141183E+38, the negatives of those numbers, or zero. Integer variables can stand for any whole number between -32767 and 32767. (Note that you can use larger and smaller numbers for floating-point variables.) Numbers with decimal parts are not accepted. If you assign a decimal number to an integer variable, the decimal part of the number is ignored. For

TABLE 1-1. BASIC Variable Types

	<i>Floating Point</i>	<i>Integer</i>	<i>Character String</i>
<b>SYMBOL</b>	None	%	\$
<b>MEANING</b>	Decimal or whole numbers	Whole numbers only	Characters in quotes
<b>EXAMPLES</b>	X, X5, RX	X%, X5%, AGE%	\$\$, R5\$, NAME\$

example, if you assign 1.99 to X%, the value accepted for X% is 1. The decimal part is truncated, not rounded.

Character string, or text string, variables can stand for any characters in quotes, including numbers, blank spaces, and special symbols. The only keyboard character that cannot be directly included in a character string is a quotation mark. A number in quotation marks is treated like any text and has no mathematical value.

## Scientific Notation

Numbers can appear as simple numbers or in scientific notation. In scientific notation, a number is reduced to its simplest one-whole-digit form. The number of missing digits is shown in the exponent. The format for representing numbers in scientific notation is as follows:

**mantissa E sign exponent**

The mantissa is a floating-point number with one whole digit (e.g., 1.55). The E, which is the operator for scientific notation, stands for *times 10 raised to the following power*. The sign is a negative or positive sign; it indicates whether the exponent is negative or positive. The exponent is the absolute value of the power to which the number 10 is raised. This is always a whole number.

Both the mantissa and the exponent can be positive or negative numbers. The following examples show how the signs of each number affect the value of the number being represented.

Mantissa	Exponent	Number	Example
Positive	Positive	Positive	1E+03 = 1000
Positive	Negative	Positive fraction	1E-03 = .001
Negative	Positive	Negative	-1E+03 = -1000
Negative	Negative	Negative fraction	-1E-03 = -.001

BASIC automatically displays numbers with absolute value smaller than .01 or higher than 99999999 in scientific notation. If you enter a number outside this range without typing it in scientific notation, BASIC rounds the number. This rounding can cause a slightly inaccurate result if the number is used in a calculation. To avoid this distortion, always enter small or large numbers in scientific notation. In any case, BASIC can keep track of only about nine decimal digits in the mantissa.

## Variable Names

Variable names can be one letter followed by other letters or numbers, plus either % or \$ when appropriate. Note, however, that although longer variable names are accepted, BASIC reads only the first two characters (plus \$ or %) in any variable name. Additional characters are ignored; use them only to make your program more readable. Because BASIC reads only the first two characters, make sure all variables in a program have unique names for the first two characters. In other words, do not use COMPANY\$ and COUNTRY\$ as variables in the same program unless you want them to have the same value.

Also be sure that variable names do not contain any BASIC keywords. If this occurs, the program aborts in a SYNTAX ERROR. For example, do not use a variable such as WORDEF, which contains the keyword DEF. Keywords cannot appear in variable names even if they are not the first two characters.

## Using Variables in Parameters

Note that in most cases a variable can be used in place of a number or text string in a command parameter. The variable must, of course, be the right type of variable. You can generally use a calculation in place of a number or numeric variable in a command parameter. For example, any of the following forms is legal:

FOR X = 1 TO 5

FOR X = 1 TO A

FOR X = A TO B-1

## Arithmetic Operators

Table 1-2 shows the operators that are used for solving mathematical problems. Note that the multiplication symbol is an \*, not an x, and that the exponentiation symbol is an up arrow.

TABLE 1-2. Mathematical Operators

↑	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction and negation

BASIC solves compound mathematical problems in this order:

First Priority:	Exponentiation
Second Priority:	Multiplications and divisions
Third Priority:	Additions and subtractions
Fourth Priority:	Comparison operations
Fifth Priority:	Logical NOTs
Sixth Priority:	Logical ANDs
Seventh Priority:	Logical ORs

When a problem contains more than one calculation from each priority group, the problems of the same priority are solved left to right.

Parentheses override this priority scheme. BASIC solves parts of a problem that are enclosed in parentheses before any other parts of a calculation. Multiple problems within parentheses are solved according to the standard priority order. Problems can contain multiple sets of parentheses, but you must be sure that the number of left parentheses equals the number of right parentheses. When parentheses are nested within parentheses, the calculations in the innermost set of parentheses are solved first.

### Comparison, or Relational, Operators

BASIC recognizes six symbols that are used to compare two values. These symbols, which are called either comparison operators or relational operators, are described in Table 1-3. The comparison operators can be used to compare constants, variables, numbers, or text strings.

TABLE 1-3. Comparison Operators

>	The left-side value is greater than the right-side value.
<	The left-side value is less than the right-side value.
=	The values are equal.
<> or ><	The values are not equal.
=> or >=	The left-side value is equal to or greater than the right-side value.
<= or =<	The left-side value is less than or equal to the right-side value.

### Logical Operators

You can also use logical operators in calculations and in comparisons of values. There are three logical operators: AND, OR, and NOT. These operators are also

called Boolean operators. Their role is to check the truth value of two values, which may be constants, numeric variables, or calculations. A result of 0 is false, and any other value is considered true.

Numeric values (operands) on either side of a logical operator should be integer numbers, not floating-point numbers, so that they are between -32767 and 32767. If you use a floating-point number, it is converted to an integer number. The result of a logical operation is always an integer value.

You can also use the logical operators to AND or OR individual bits (binary digits) in two operands. You can use NOT to invert individual bits in a single operand.

The following chart shows how each of the logical operators provides a result after combining the truth values of two values. A value of -1 is used for a true result.

-1 AND -1 = -1	-1 OR -1 = -1	NOT -1 = 0	-1 XOR -1 = 0
-1 AND 0 = 0	-1 OR 0 = -1	NOT 0 = -1	-1 XOR 0 = -1
0 AND -1 = 0	0 OR -1 = -1		0 XOR -1 = -1
0 AND 0 = 0	0 OR 0 = 0		0 XOR 0 = 0

## Logical AND

AND requires both values to be true for the result of the ANDed expression to be true. Any other combination produces a false result. AND lets you set compound comparisons in a conditional command such as IF or WHILE. When you join a compound IF or WHILE command with AND, the result of the compound comparison is false if one or both of the conditions are false. For example

```

10 INPUT "AGE, ANNUAL INCOME"; X, Y
20 IF X>=60 AND Y<=10000 THEN PRINT "ELIGIBLE": ELSE PRINT
    "INELIGIBLE"
RUN
AGE, ANNUAL INCOME ? 60, 15000
INELIGIBLE
RUN
AGE, ANNUAL INCOME ? 65, 9900
ELIGIBLE

```

The IF command in the first execution is false because only one IF condition is true (X is greater than or equal to 60, but Y is not less than or equal to 10000). Therefore the THEN clause does not execute, and the ELSE clause does execute. In the second execution of the program, the IF command is true because both the first AND the second condition are true.

## Logical OR

OR requires only one of the two conditions to be met for the compound expression to be true. An ORed comparison is false only when both values are false. For example

```
10 INPUT "AGE, ANNUAL INCOME";X, Y
20 IF X=>60 OR Y<=10000 THEN PRINT "ELIGIBLE": ELSE PRINT
    "INELIGIBLE"
```

RUN

```
AGE, ANNUAL INCOME ? 60, 15000
ELIGIBLE
```

RUN

```
AGE, ANNUAL INCOME ? 65, 9900
ELIGIBLE
```

RUN

```
AGE, ANNUAL INCOME ? 55, 12000
INELIGIBLE
```

This modification of the previous program shows the difference between AND and OR. In the first program, the input 60 and 15000 makes the IF command false because both conditions must be met before the IF command is true. In the second program with OR in the IF command, the same input makes the IF command true because only one of the two conditions has to be met for the whole IF command to be true. The third execution shows that the only time ORed IF commands are false is when NEITHER condition is met.

## Logical NOT

NOT is somewhat different from AND and OR. NOT does not compare two values. Instead, NOT lets you negate any value or comparison operator. For example, we will add NOT to an IF command that compares a value to see if it is greater than another value: IF NOT X > Y. Without the NOT, this command checks to see if X is greater than Y. When NOT is added, this command checks to see if X is NOT greater than Y; in other words, if X is less than or equal to Y.

When you use NOT, you must type NOT before the values you are comparing. This may seem awkward because we would say “if X is NOT greater than Y,” but

you must put the NOT just before the value or comparison to negate or the command will cause a syntax error, which always stops a program. You might think of NOT as changing the meaning of  $X > Y$  to "unless X is greater than Y."

The following comparisons show how NOT affects comparison operators. The comparisons on the right are the same as those on the left:

$X > Y$	same as	$\text{NOT } X \leq Y$
$X \leq Y$	same as	$\text{NOT } X > Y$
$X \neq Y$	same as	$\text{NOT } X = Y$
$X = Y$	same as	$\text{NOT } X \neq Y$

The last NOT clause contains a double negative: NOT and  $\neq$  (not equal). Double negatives, though discouraged in most English applications, are acceptable in BASIC. But like double negatives in English, double negatives in BASIC cancel each other, so  $\text{NOT } X \neq Y$  is the same as  $X = Y$ .

This short program uses NOT to make the opposite of the comparison operator typed in the IF command:

```
10 INPUT "WHAT'S YOUR AGE"; A
20 IF NOT A => 21 THEN PRINT "USER IS A MINOR": ELSE PRINT
    "OK"
```

RUN

```
WHAT'S YOUR AGE ? 20
USER IS A MINOR
```

RUN

```
WHAT'S YOUR AGE ? 21
OK
```

The NOT makes the greater-than-or-equal-to symbol mean this: unless A is greater-than-or-equal-to 21, THEN print USER IS A MINOR. The comparison is the same as IF  $A < 21$ .

## Exclusive OR (XOR)

The exclusive OR, which is called XOR, is not a standard logical operator. XOR is used in machine language (EOR), and it is used in the WAIT command to invert the comparison of two bits. When both XORed bits have the same value,

either both 0 or both 1, the result of the comparison is 0. When the two XORed bits are not equal, the result of the comparison is 1.

## Comparing Text Strings

You can use the standard comparison operators to compare text strings. Strings are compared character by character; blanks are considered to be significant characters. So, for example, "WORD" does not equal "WORD ". Each character is evaluated according to its PET/CBM character set (CHR\$) number (see Appendix C). This character set gives a number value to every possible character. "A"(65) is less than "B"(66) is less than "C"(67), and so forth. A blank has a value of 32, so it is less than any letter, but significant nonetheless. "WORD" is less than "WORD " because the blank in "WORD " gives that string a greater value.

Consider the expression A\$=B\$. If all characters in all character positions in the two strings are equal, a truth result (-1) is returned. False comparisons produce a 0 result. The result of a string comparison is always an integer value (0 or -1), so you can use the result in a mathematical calculation. Note, however, that you cannot use a false result as a divisor because division by zero is illegal.

## BASIC Abbreviations

Most BASIC keywords can be abbreviated. These time-saving abbreviations are shown in Table 1-4. You can use abbreviations to "cheat" on the 88-character-per-command line limitation. But when a line containing abbreviations is LISTed, the abbreviations are converted into spelled-out keywords. You cannot edit and reenter such a line using the screen editor if it is more than 88 characters when LISTed. Only the first 88 characters will be accepted. Retype the line with the abbreviations instead.

The table shows some characters in uppercase and others in lowercase. You will no doubt usually enter programs in uppercase/graphic mode, so abbreviations will not appear in upper- and lowercase. Instead, the uppercase letters, which must be typed with the SHIFT key, appear as graphic symbols. We use uppercase and lowercase letters instead of uppercase and graphic symbols to make the table easier to read. Just remember to press SHIFT when you type the letters shown here in uppercase.

## Crunching Programs

When you want a program to use less memory, there are several crunching tricks you can use; they can be found on page 12.

TABLE 1-4. BASIC Abbreviations

<i>Keyword</i>	<i>Abbreviation</i>	<i>Keyword</i>	<i>Abbreviation</i>
ABS	aB	GRAPHIC	gR
AND	aN	GSHAPE	gS
ASC	aS	HEADER	heA
ATN	aT	HELP	heL
AUTO	aU	HEX\$	hE
BACKUP	bA	IF	—
BOX	bO	INPUT	—
CHAR	chA	INPUT#	iN
CHR\$	cH	INSTR	inS
CIRCLE	cI	INT	—
CLOSE	clO	JOY	jO
CLR	cL	KEY	kE
CMD	cM	LEFT\$	leF
COLLECT	colL	LEN	—
COLOR	coL	LET	lE
CONT	cO	LIST	li
COPY	coP	LOAD	lO
COS	—	LOCATE	loC
DATA	dA	LOG	—
DEC	—	LOOP	loO
DEF	dE	MID\$	mI
DELETE	deL	MONITOR	mO
DIM	dI	NEW	—
DIRECTORY	diR	NEXT	nE
DLOAD	dL	NOT	nO
DO	—	ON	—
DRAW	dR	OPEN	oP
DSAVE	dS	OR	—
ELSE	eL	PAINT	pA
END	eN	PEEK	pE
ERR\$	eR	POKE	pO
EXIT	exI	POS	—
EXP	eX	PRINT	?
FN	—	PRINT#	pR
FOR	fO	PUDEF	pU
FRE	fR	RCLR	rC
GET	gE	RDOT	rD
GO	—	READ	rE
GOSUB	goS	REM	—
GOTO	gO	RENAME	reN

TABLE 1-4. BASIC Abbreviations (continued)

<i>Keyword</i>	<i>Abbreviation</i>	<i>Keyword</i>	<i>Abbreviation</i>
RENUMBER	renU	STEP	stE
RESTORE	reS	STOP	sT
RESUME	resU	STR\$	stR
RETURN	reT	SYS	sY
RGR	rG	TAB(	tA
RIGHT\$	rI	TAN	—
RLUM	rL	THEN	tH
RND	rN	TO	—
RUN	rU	TRAP	tR
SAVE	sA	TROFF	troF
SCALE	scA	TRON	trO
SCNCLR	sC	UNTIL	uN
SCRATCH	scR	USING	usI
SGN	sG	USR	uS
SIN	sI	VAL	vA
SOUND	sO	VERIFY	vE
SPC(	sP	VOL	vO
SQR	sQ	WAIT	wA
SSHAPE	sS	WHILE	wh

- Use the lowest possible line numbers. References to large line numbers take up more memory than those to small line numbers. When you are writing the program, it is smart to leave gaps between line numbers so you can easily add lines. Once the program is finished, however, you can use the RENUMBER command to change all the line numbers to lower, closer-together numbers.
- Put multiple commands on a line. Separate commands on the same line with a colon. There is no need to put spaces between the commands. Remember, however, that each program line cannot exceed 88 characters in length.
- Delete spaces between characters in the program lines. Although spaces improve readability, they take up memory. Blanks are never required, so omit them if you need to.
- Remove REM statements if you need more room. Though useful for documenting a program, they do use up memory.
- Use variables in place of long numbers and calculations that are repeated in a program.
- Use arrays to hold groups of data. Arrays, which are explained elsewhere in this chapter, handle large groups of data as an organized list. If an array

represents integers that never go outside the range 32767 to 32767, then it should be defined as an integer array (with the % designation).

- Use DEF FN to define frequently used functions.
- Use READ and DATA commands to handle long lists of data whether or not the data items are related. DATA commands can be placed together at the end of the program and quickly accessed, data item by data item.
- Write subroutines to handle repeated tasks. Subroutines improve program organization, and they can save memory by omitting needlessly repeated commands.

**Note:** When BASIC searches for a program line to GOTO or GOSUB, it starts at the beginning of the program and looks sequentially. To speed execution, place DATA commands at the end of the program so that BASIC does not have to search through them when looking for a program line. Place frequently used subroutines near the beginning of the program so they are easy for BASIC to find.

You can save typing time (though not execution time) by defining function keys to print commands you use repeatedly. Function keys are easy to define, and you will save a lot of time if you can just press a key instead of typing the command. For example, if your program will have a lot of INPUT commands, define a function key to print INPUT.

Defining a function key to print a command is also useful when you are experimenting with a graphic-mode drawing. Define a key as one of the graphic mode commands (e.g., KEY 3,"GRAPHIC 2,1") so you can quickly switch to the drawing mode you want to use. The quickest way to get out of one of the drawing modes is to commit a syntax error. Just type a letter and press RETURN. A syntax error automatically cancels the current drawing mode and returns to text/graphic mode. The drawing in the graphic mode is unaffected by the syntax error. To get back to it, issue a GRAPHIC command without the ,1, which clears the graphic mode screen.

## BASIC Version 3.5 Commands, Functions, and System Variables

The rest of this chapter explains BASIC commands, functions, and reserved system variables together in alphabetical order.

**ABS**  
**ABS (number)**

**Abbr. aB**

ABS is the numeric function that finds the absolute value of the number enclosed in parentheses. The absolute value of a number is that number without

any sign, which means negative signs are removed from negative numbers. The absolute value of 0 is 0.

**Parameter:** any number, positive or negative, or a numeric expression

To display the absolute value for a number, put the ABS function in a PRINT command.

Examples: **PRINT ABS(35)**                  Displays the absolute value of 35.  
              35

**PRINT ABS(-35)**                  Displays the absolute value of -35.  
              35

**ASC**                                  **Abbr. aS**  
**ASC (string)**

ASC is the numeric function that finds the character-string code for the first character of the string inside parentheses. ASC is the opposite of the CHR\$(x) function, which finds the character for the character-string code number enclosed in parentheses.

**Parameter:** any character or key in quotation marks, or a string expression

If you type more than one character in an ASC function, the computer prints the code for only the first character in the string; all other characters are ignored.

To display the character-string code for a character, put the ASC function in a PRINT command.

Examples: **PRINT ASC("M")**                  Displays the CHR\$ code for M.  
              77

**PRINT ASC("█")**                  Displays the CHR\$ code for the shifted CLEAR key, which is printed as a reversed heart.  
              147

**PRINT ASC("MAP")**                  Displays the CHR\$ code for only the first letter in the string MAP.  
              77

**ATN**                                  **Abbr. aT**  
**ATN (number)**

ATN is the numeric function that finds the arctangent in radians of the number enclosed in parentheses. For more information, see the Mathematical Calculations section of Chapter 3.

**Parameter:** any numeric expression

Examples: PRINT ATN(1) Displays the arctangent of 1 in radians.  
.785398163

PRINT ATN(-2)\*180/ $\pi$  Displays the arctangent of -2 in degrees.  
-63.4349488

**AUTO** **Abbr. aU**  
**AUTO increment**

AUTO prints BASIC program line numbers automatically, which is useful when you are writing a long program. After you turn on automatic line numbering, type the first line in your program (using any line number) and press RETURN. Thereafter, AUTO prints the next line number as soon as you press RETURN at the end of each line. The increment between the line numbers is determined by the number you type in the AUTO command.

**Parameter:** increment number

The increment number can be any positive number that does not exceed 63999, which is the highest possible line number for a BASIC program. Entering a line number greater than 63999 creates a syntax error.

## Turning Off AUTO

You have to be in immediate mode to use the AUTO command. AUTO prints a line number every time you press RETURN on a program line containing more than the line number. Press RETURN on a line containing only the line number to stop the line numbering. Then issue an AUTO 0 or AUTO with no number to turn off automatic line numbering.

You can also issue a RUN command instead of AUTO 0 or AUTO, but note that you must issue one of these commands to turn off automatic line numbering.

Example: **AUTO 20** Automatically numbers lines in increments of 20.  
**50 INPUT "DATE"; D** Type any number for the first line number.  
**70 INPUT "TIME"; T** AUTO adds the increment value (20) and prints the next line number.

**BACKUP**                          **Abbr. bA**  
**BACKUP Ddrive TO Ddrive, ON Uunit**

Duplicates an entire disk in a dual disk drive. BACKUP does not let you copy just parts of disks or change the names of files or of the disk. Use the COPY command to duplicate individual files or change file names. This command does not work with single disk drives such as the 1541.

The disk you are copying is the “master” disk; you are copying from the master TO the blank disk.

**Cautions:**

1. BACKUP headers the recipient disk before copying files from the master disk. Since headering a disk erases all the information stored on the disk, do not BACKUP onto a disk that contains files you want to keep. Use a blank disk or a disk that contains information you no longer need.
2. BACKUP does not affect files on the master disk. However, since BACKUP does header the recipient disk, double check to be sure the master disk is in the drive you name as the master drive in the BACKUP command. To avoid accidentally backing up in the wrong direction, always put the master disk in drive 0.
3. BACKUP copies files indiscriminately—errors and all. For this reason, many programmers prefer to use the COPY command or a copy utility program to duplicate disks. If the master disk contains errors, do not use BACKUP.

**Parameters:** D disk drive number TO D disk drive number, U unit number

1. Drive numbers are either 0 or 1. No other numbers are allowed. The first disk drive number indicates which drive contains the master disk, whose contents you are copying. You should always put the master disk in drive 0.
2. TO is part of the command and must be included.
3. The second disk drive indicates which drive contains the blank disk onto which you are copying the information from the master disk. Always put the recipient disk in drive 1.
4. Unit number is an optional parameter that you should rarely if ever need. Use it only if you have more than one dual disk drive connected to your computer, and you are using a device other than unit 8 in the backup procedure. You can precede the unit number with ON, but ON is not required. The unit number must be between 8 and 11.

**Note:** The drive and unit number parameters can be specified with a variable or expression in parentheses.

**Examples:** BACKUP DO TO D1

Copies all the files on the disk in drive 0 onto the disk in drive 1.

**BACKUP DO TO D1, U9** Copies the disk in drive 0 of unit 9 onto the disk in drive 1 of unit 9.

**BOX****Abbr. bO****BOX color source, corner coordinate, corner coordinate, angle, fill**

Draws a rectangular shape in any of the four graphic drawing modes. You supply the column, row coordinates of two opposite corners. You can include a parameter to draw the rectangle at a tilted angle, and you can draw the box as an outline or as a solid shape.

BOX can be executed only in a graphic mode. For more information on the graphic modes and on the coordinates for the BOX command, see Chapter 4.

<i>Parameter</i>	<i>Values</i>	<i>Default</i>
Color source	0–3	1
First corner coordinate		
Column coordinate	0–319 (high-res modes) 0–159 (multicolor modes)	
Row coordinate	0–199	
Second corner coordinate		
Column Coordinate	0–319 (high-res modes) 0–159 (multicolor modes)	pixel cursor
Row coordinate	0–199	
Angle	0–360	0 (no angle)
Fill	0 (outline) or 1 (solid)	0

1. The color source indirectly selects the color for the drawing. There are five color sources, but color source 4 (the border color) cannot be used in drawing commands. Color sources 2 and 3 can be used only in multicolor modes.

- 0 screen background color
- 1 foreground color
- 2 multicolor mode extra color 1
- 3 multicolor mode extra color 2

The color source number you include in the BOX command tells the computer to draw in the current color for that source. For example, if you select 1, the computer draws the box in the current foreground color. If you want to use a color other than one of the current source values, you must first use the COLOR command to change one of the source values. Only sources 1 and 2 can be used to

draw with more than one color on the same screen. Sources 0 and 3 are global colors, which means that changing the colors for these sources affects all shapes previously drawn with them.

If you want to use the default value (1, the current foreground color), you do not have to type a number, but you must type a comma before the next parameter.

2. The first set of coordinates names the location of one corner of the box. The second set names the location of the opposite corner of the box. The second set of coordinates can be omitted. They will default to the pixel cursor. If you omit the coordinates, type one comma instead. These are the only coordinates you give.

You can name either of the opposite sets of corners, and you can enter them in any order (i.e., you do not have to enter the top corner first). If you name corners with the same row or column coordinate, you will draw a line instead of a rectangle.

3. After the box is drawn, the pixel cursor is at the location of the second set of coordinates.

4. You can draw the box tilted at any angle from 0 to 360. For example, a 45-degree angle draws a diamond shape. The default value is 0, no tilting. The tilting is done after the box is calculated. Therefore, the corners will not be at the specified coordinates. If you omit this parameter and use the Fill parameter, you must type a comma in place of the angle parameter.

5. You can draw the box as an outline or as a solid block. The default is 0, which draws an outline. If you want to draw a solid block, select 1 as the value for this parameter. No other values are legal. Since this is the last parameter, you do not need to type a comma to take its place if you do not use this parameter.

Examples:	<b>BOX, 60,50, 240,150</b>	Draws a rectangle in outline.
	<b>BOX, 80,50, 150,130, 45, 1</b>	Draws a solid rectangle tilted at 45 degrees.
	<b>10 GRAPHIC 2,1</b>	Enters split-screen high-resolution mode.
	<b>20 COLOR 1,5,4</b>	Changes the color of source 1, thereby indirectly changing the color used to draw the boxes.
	<b>30 FOR A=0 TO 360 STEP 10</b>	Sets up a loop to increment the value of the angle parameter in the BOX command.
	<b>40 BOX, 120,50, 200,100, A</b>	Draws a rectangle at the angle of A.
	<b>50 NEXT</b>	
	<b>CHAR</b>	<b>Abbr. chA</b>
	<b>CHAR color source, column coordinate, row coordinate, string, reverse mode</b>	

Displays a message at a specified screen location in any text or graphic mode. You give the column and row coordinates of the message in the CHAR command. You can also print the message in reversed-image mode.

CHAR is similar to the text-printing capabilities of PRINT, but CHAR also lets you easily position the message on the screen. In addition, CHAR can display messages in graphic modes, but PRINT cannot.

CHAR lets you print on top of, above, or below other messages. Because you can position each CHAR message, you can place messages anywhere in relation to each other.

CHAR has some slightly different features in the text and graphic modes. When you use CHAR in a text mode only, you can print in flashing mode, and you can include color changes and other special key commands that you can use in PRINT commands.

<i>Parameter</i>	<i>Values</i>	<i>Default</i>
Color source	0-3	1
Column coordinate	0-39	
Row coordinate	0-24	
Message	String expression	
Reverse mode	0 or 1	0 (off)

1. The color source indirectly selects the color for the drawing. There are five color sources, but color source 4 (the border color) cannot be used with drawing commands. Color sources 2 and 3 can be used only in multicolor modes.

- 0 screen background color
- 1 foreground color (default value)
- 2 multicolor mode extra color 1
- 3 multicolor mode extra color 2

The color source number you include in the CHAR command tells the computer to use the current color for that source to print the message. For example, if you select 1, the computer prints the message in the current foreground color. If you want to use a color other than one of the current source values, you must first use the COLOR command to change one of the source values. Only sources 1 and 2 can be used to draw with more than one color on the same screen. Sources 0 and 3 are global (whole screen) colors.

If you want to use the default value (1, the current foreground color), you do not have to type a number, but you must type a comma before the next parameter.

2. Even when you are displaying a message in a graphic mode, use the standard text column (0-39) and row (0-24) numbers. You do not place CHAR messages with the graphic mode 320-by-200 coordinates because you are placing whole letters, not small dots. Be careful; this can be confusing.

3. The message must be in quotes, just as a PRINT command message. You

can also use a text-string variable as the message parameter. You can concatenate strings to the message by adding a plus sign and the string. For example: CHAR,2,2,"HELLO, "+N\$+ HOW ARE YOU".

4. When you are using CHAR in a graphic mode, add a 1 as the final parameter when you want to display the message in reversed image. The default is 0, no reversed image.

When you are using CHAR in a text mode, do not use the Reversed Mode parameter. If you want to display the CHAR message in reversed image, use the CONTROL and RVS ON keys inside the quotes, just as you would in a PRINT command. This method does NOT work in a graphic mode.

When you are using CHAR in a text mode, you can use flashing mode and change character color by pressing CONTROL and FLASH ON, and CONTROL or C and the color key inside the quotes, just as you would in a PRINT command. You cannot use flashing mode in a graphic mode, and you cannot use this method to change foreground color in a graphic mode.

You can include special key commands such as the CLEAR key in a CHAR command in a text mode but not in a graphic mode. If you include a special key symbol in a graphic mode CHAR command, the computer prints the key's graphic symbol, but does not execute the key command. For example, if you include a CLEAR key inside the CHAR quotes in a graphic mode, the computer does not clear the screen, but it does print the heart symbol that stands for the CLEAR key in quote mode.

If you use the CHAR command in a split screen mode, the message will be printed on the graphic screen, not on the text screen. Even if the coordinates indicate that the message should be placed on the text area of the screen (bottom five lines), it will be plotted on the (unseen) graphic screen instead.

<b>Examples:</b>	10 GRAPHIC 1,1	
	20 CIRCLE, 160,100, 60,50,,,120	
	30 CHAR, 16,17, "ISOSCELES",1	Displays the message ISOSCELES at column 16, row 17 in reverse.
	40 CHAR, 16,18 "TRIANGLE" NEW	Displays TRIANGLE at column 16, row 18.
	10 GRAPHIC 0,1	Switches to text/graphic mode.
	20 INPUT "WHAT'S YOUR NAME"; A\$	
	30 CHAR, 10,20,"HELLO, "+A\$	
	<b>CHR\$</b>	<b>Abbr. cH</b>
	<b>CHR\$ (number)</b>	

Finds the keyboard definition represented by the character code in parentheses. Each key on the keyboard—including key combinations such as SHIFT

and CLEAR –has a unique character-string value that can be called by its CHR\$ code. You can use CHR\$ values to do anything to the screen output that you can do by pressing a key, such as changing character colors, turning on reversed-image mode, or deleting a character.

Printing the CHR\$ value to the screen has the same effect as pressing the key. For example, PRINT CHR\$(77) is the same as PRINT "M". This feature of CHR\$ is especially useful when you want to defer the "pressing" of a key. For example, in a BASIC program the only way to print a message that contains a quotation mark is to use the CHR\$ code for the quotation mark:

```
PRINT "IBM'S MOTTO IS ";CHR$(34);"THINK"; CHR$(34)  
IBM'S MOTTO IS "THINK"
```

If you actually press the quotation mark key when you type the line, the quotation mark opens or closes quote mode:

```
PRINT "IBM'S MOTTO IS "THINK""  
IBM'S MOTTO IS 0
```

In the second PRINT command example, the computer assumes the quote before THINK turns off quote mode. The computer also assumes that THINK is a variable name, which is why the 0 is printed. The only way to print the quotation mark as a character is to use its character code in a CHR\$ function.

The CHR\$ function is frequently used in function-key definitions to print a quotation mark or "press" a RETURN key at the end of the key definition.

Appendix C lists all the CHR\$ values. Appendix D contains the standard ASCII codes that are used by many computers for your reference. To find a CHR\$ value, you can use the ASC function, which finds the code for any key.

CHR\$ codes are used in I/O to devices other than the screen as well. The printable characters are generally the same, but the control functions will be different with a printer, for example, than with the screen.

Example:	PRINT CHR\$(28); A; CHR\$(129); B	Changes the character color to red, prints the value for A, changes the character color to orange, and prints B.
	<b>CIRCLE</b>	<b>Abbr. cl</b>
	<b>CIRCLE color source, center coordinates, x radius, y radius, start arc, end arc, angle, increment</b>	

This graphic mode command draws circles as well as a variety of other shapes. CIRCLE draws curved shapes such as arcs and ovals. CIRCLE also draws any

polygon with regular sides. For example, you can use CIRCLE to draw an isosceles triangle.

You can draw CIRCLE shapes tilted at any angle. If you want to draw solid shapes, you must use the PAINT command to fill in the CIRCLE outline. Unlike the BOX command, CIRCLE has no parameter for drawing a solid shape. See Chapter 4 for more information on CIRCLE coordinates.

<i>Parameter</i>	<i>Values</i>	<i>Default</i>
Color source	0–3	1
Center coordinates		Current pixel-cursor location
Column coordinate		
High-res modes	0–319	
Multicolor	0–159	
Row coordinate	0–199	
Column radius		
High-res modes	0–319	
Multicolor modes	0–159	
Row radius	0–199	Column radius value
Arc starting angle	0–360	0
Arc ending angle	0–360	360
Angle of tilt	0–360	0
Segment size	0–255	2

1. The color source indirectly selects the color for the drawing. There are five color sources, but color source 4 (the border color) cannot be used in drawing commands. Color sources 2 and 3 can be used only in multicolor modes.

- 0 screen background color
- 1 foreground color
- 2 multicolor mode extra color 1
- 3 multicolor mode extra color 2

The color source number you include in the CIRCLE command tells the computer to draw in the current color for that source. For example, if you select 1, the computer draws the shape in the current foreground color. If you want to use a color other than one of the current source values, you must first use the COLOR command to change one of the source values. Only sources 1 and 2 can be used to draw with more than one color on the same screen. Sources 0 and 3 are global (whole screen) colors.

If you want to use the default value (1, the current foreground color), you do not have to type a number, but you must type a comma before the next parameter.

2. The first coordinates name the center of the shape. The default center is the current location of the pixel cursor.
3. Horizontal radius is the distance from the center of the shape to the left and right sides of the shape.
4. Vertical radius is the distance from the center of the shape to the top and bottom of the shape. Because vertical dots on the high-resolution graphic screen are slightly farther apart than horizontal dots are, when the vertical and horizontal radii have the same value, the shape drawn is an oval, not a circle. To draw a circle, the vertical radius must be scaled to the horizontal radius (e.g., 50 horizontal, 47 vertical). For multicolor coordinates, 25 horizontal is about 47 vertical.
5. Use the starting angle only when you want to draw an arc. Zero degrees is at the top of the screen; 180 is at the bottom; 90 is to the right; and 270 is to the left.
6. The ending angle for an arc defaults to 360 if not specified.
7. You can draw a shape tilted at an angle from 0 to 360 degrees. The default is 0, which is no tilt.
8. You choose the shape of the CIRCLE drawing by choosing the number of degrees between the segments in the drawing. The default number of degrees between shape segments is 2 degrees, which draws a circle.

When you draw a circle on the screen, you are actually drawing a 180-sided polygon (360 divided by 2, the default segment value). The larger the increment of degrees between segments, the more angular the drawing. For example, a segment value of 120 draws a triangle, not a circle.

To draw a polygon with the CIRCLE command, divide 360 (the total number of degrees in a real circle) by the number of sides you want the shape to have. For example, to draw a hexagon, divide 360 by 6. Then use the result, 60, as the segment parameter.

**Note:** Although the segment-size parameter can have a value of up to 255, any value between 180 and 255 draws a straight line.

**Examples:** **10 GRAPHIC 2,1**

<b>20 CIRCLE, 140,80, 90,50, 180,320</b>	Draws an arc.
<b>30 CIRCLE, 160,100, 60,50,,,90</b>	Draws a diamond.
<b>40 CIRCLE, 160,100, 60,50,,,90,120</b>	Draws a triangle rotated 90 degrees.
<b>50 CIRCLE, 160,75, 72,60</b>	Draws a circle.

**CLOSE**

**Abbr. clo**

**CLOSE file number**

Closes access to a peripheral device or a data file on tape or disk. CLOSE is paired with the OPEN command, which gives access to a data file or peripheral device. You must CLOSE the file or device with the same logical file number you used to OPEN it.

Be sure to CLOSE files and devices when you finish accessing them; leaving

them OPEN leads to errors. Note that you can have only 10 files OPENed at a time.

**Parameter:** logical file number

The logical file number must be the same number you used to OPEN the file. The logical file number has no relation to the file itself; you can use any number between 1 and 255 as long as you use the same number in commands, such as OPEN and CLOSE, that refer to the file.

Example: 10 OPEN 4,4,7      Opens access to the printer.

20 INPUT X\$, Y\$

30 PRINT #4, X\$, Y\$

40 CLOSE 4

Clears the values of all variables without otherwise affecting the current program. All numeric variables are reset to zero, and all text-string variables are reset to a null string.

The CLR command is automatically executed when you issue a NEW or RUN command. CLR is also executed when you edit a program, which is why you cannot use CONT to resume program execution after you edit the program.

## CMD file number, output list

Interrupts the normal flow of output to the screen and sends the output to a different device, such as the printer or a data file. The device or file must first be accessed by an OPEN command.

**Parameters:** logical file number, list of output items

1. The logical file number must be the same number you used in the OPEN command to access the file.
  2. The list of output items is optional. It can include numeric and string expressions separated with commas or semicolons, just as a PRINT command.

Examples: 10 OPEN 4.4

Opens access to the printer.

#### 20 CMD 4."THIS IS MY PROGRAM"

Prints output to the printer.  
Directs output to printer and prints a heading.

30 LIST

Lists program to printer under heading.

**40 PRINT# 4:CLOSE 4**

Turns off the CMD command and closes access to the printer.

The following sequence can be used in immediate mode to LIST the current program to the printer:

**OPEN4,4:CMD4:LIST  
PRINT#4:CLOSE4**

**COLLECT**  
**COLLECT Ddrive, ON Uunit**

**Abbr. coll**

Cleans up a disk by clearing files that were improperly closed and are inaccessible. COLLECT removes improperly closed files from a disk and its directory so you can store files in disk space that was rendered unusable.

**Note:** Never use the COLLECT command on disks that contain information written with direct-access commands (see Chapter 6).

**Parameters:** drive number, U unit number

1. Drive numbers are either 0 or 1. No other numbers are allowed.
2. Unit number is an optional parameter. Use it only if you have more than one disk drive connected to your computer, and you are using a device other than unit 8 in the COLLECT procedure. You must precede the unit number with U, and the unit number must be between 8 and 11. You can type ON before U, but ON is not required.

**Note:** The drive and unit number parameters can be specified with a variable or expression in parentheses.

**Example:** **COLLECT DO**      Gets rid of all inaccessible files on drive 0.

**COLOR**                  **Abbr. col**  
**COLOR color source, color, luminance**

Lets you change the color of the screen, the characters, the border, and the multicolor sources. You can choose any of the 16 basic colors, and one of 8 shades of any color except black, which has no shades.

The COLOR command lets you indirectly choose the color to be used in drawing commands. Drawing commands such as CIRCLE do not have parameters for selecting color. You must first change the color with a COLOR command.

You can change character color with the color keys on the keyboard, which are pressed with the **Ctrl** or CONTROL keys. This method of selecting a character color has two shortcomings: you cannot select a luminance level, and the color change does not affect the value in the foreground-color source (source 1). This

means that although the new color affects the characters you type, it does not change the color of the drawings you do in graphic modes.

<i>Parameter</i>	<i>Values</i>	<i>Default</i>
Screen color source number	0–4	None
Color number	1–16	None
Luminance value	0–7	7 (lightest shade)

1. There are five color sources whose color you can change with the COLOR command:

- 0 screen background color
- 1 foreground color
- 2 multicolor mode extra color 1
- 3 multicolor mode extra color 2
- 4 screen border color

You cannot use the border color (source 4) in drawing commands. Color source 1, foreground color, determines the default color of characters and graphics on the screen. Changing the character color with the color keys changes only the color of characters printed in text mode. Changing the foreground color (source 1) changes both the default graphics drawing color and the color of characters printed in text mode.

2. You can choose any of the colors listed on the color keys by using the following numbers:

1 = black	9 = orange
2 = white	10 = brown
3 = red	11 = yellow-green
4 = cyan	12 = pink
5 = purple	13 = blue-green
6 = green	14 = light blue
7 = medium blue	15 = dark blue
8 = yellow	16 = light green

3. You can choose one of eight shades of a color by adding a luminance level. To choose the darkest shade, use 0; the shades are progressively lighter, 7 being the brightest.

The luminance setting is optional. The default value is 7, which selects the brightest shade of the color. When you use the color keys on the keyboard to change the character color, a preset luminance value is automatically used.

Black has no luminance values, although it is not an error to include this parameter when you are selecting black. The luminance settings 0 through 6 for white are shades of gray.

**Examples:** COLOR 0, 12, 3      Changes the screen background to medium pink.  
COLOR 1, 3, 0      Changes the foreground color to dark red.

**CONT**                  **Abbr. cO**

Lets you restart a BASIC program after you have interrupted its execution with the STOP key, the STOP command, or an END command in a program. The program resumes at the point in the program where execution was interrupted, and all variables retain their most-recent values.

You cannot resume execution with CONT if any of the following events have occurred between STOP and CONT:

- You change or add lines to the program.
- You move the cursor to a program line and press RETURN with or without changing the line.
- The program stopped because of an error (in which case an error message would have appeared on the screen).
- You do anything to cause an error after suspending the program.
- You execute a CLR command.

**COPY**                  **Abbr. coP**

**COPY Ddrive, old file name TO Ddrive, new file name, ON Uunit**

Makes a duplicate of a disk file or an entire disk. On a single disk drive, such as the 1541, you can copy a file only onto the same disk. You must give a copy on the same disk a different name from the original file. The COPY procedure does not affect the master file. In addition, you can copy from one drive to another if you have a dual disk drive. You cannot use the COPY command to copy from one single disk drive to another with different unit (device) numbers.

Unlike the BACKUP command, COPY does not header the recipient disk before duplicating the file(s). The advantage of this difference is that you can use COPY to add files to a disk that already contains files you want to keep. The only drawback is that you must take care not to COPY a group of files onto a disk that does not have enough room. Avoid this problem by checking the directories of both disks before you issue the COPY command. A 1541 disk can hold up to 664 blocks (256 bytes each) of information.

Also unlike the BACKUP command, COPY does not duplicate disk errors in a file. If a file you are COPYing contains a disk error, the file is not copied. The advantage of this difference is that you do not duplicate inaccessible files.

<i>Parameter</i>	<i>Values</i>	<i>Default</i>
drive number,	0 or 1	0
"master file"	any file name in quotes	
TO drive number,	0 or 1	0
"receiving file",	any file name in quotes	
U unit number	8-11	8

1. Both drive numbers can be omitted if you are making a duplicate of a file on the same disk.
2. The Master File is the name of the file you want to copy. The name must be in quotes.
3. TO is a necessary part of the COPY command. The drive number can be omitted, but TO cannot.
4. The Receiving File is the name of the file that will become the copy of the master file. The receiving file name can be the same as the master file, which is likely when you are copying from one drive to another. The name must be in quotes. The receiving file name must be different if you are copying a file onto the same disk.
5. Unit number is an optional parameter. Use it only if you have more than one disk drive connected to your computer and you are using a device other than unit 8 in the COPY procedure. You must precede the unit number with U, and the unit number must be between 8 and 11. You can type ON before the unit number, but ON is not required. Most people will never need the unit number option.

**Note:** The drive and unit number parameters and the file names can be specified with a variable or expression in parentheses.

Examples:	COPY DO, "ADDR" TO D1, "ADDR"	Copies the file ADDR from the disk in drive 0 to the disk in drive 1.
	COPY DO TO D1	Copies all the files on the disk in drive 0 to the disk in drive 1.
	COPY "MEMO1" TO "MEMO2"	Copies the file MEMO1 onto the same disk, renaming the file MEMO2. This does not affect MEMO1.

**COS****Abbr. none****COS (number)**

**COS** is the numeric function that finds the cosine of the angle in parentheses. The angle must be expressed in radians. For more information, see the Mathematical Calculations section of Chapter 3.

**Parameter:** any number or numeric expression

**Examples:** PRINT COS( $\pi$ )

-1

Prints the cosine of an angle of  $\pi$  radians (180 degrees).

PRINT COS(30\* $\pi$ /180)

.866025404

Prints the cosine of an angle of 30 degrees.

**DATA****Abbr. dA****DATA data list**

Contains a list of values that are available for assignment to variables by READ commands. DATA commands are complements to READ commands; neither command works without the other.

DATA items can be either numbers or text. Text data does not need to be enclosed in quotes unless it includes an embedded comma or colon, although the text items are treated as if they were in quotes. Since DATA commands always contain constant values, not variable names, the computer assumes that any nonnumeric DATA item is text.

DATA commands can contain any number of values as long as the list is no longer than 88 characters on the screen. READ commands can get data from DATA commands anywhere in the program.

You must have enough DATA values in a program to assign a value to every variable in the READ commands that are executed in the program. If there are not enough DATA values, the program is aborted and the error message OUT OF DATA is displayed.

When DATA items are READ, the computer keeps track of the last value read by marking its place with a data pointer. You can reREAD DATA items by using the RESTORE command, which resets the data pointer to the beginning of a DATA command.

**Parameter:** list of data values separated by commas

DATA items must be separated by commas. Text items do not have to be in quotes unless they contain commas or colons.

**Example:** 10 DATA 1,2,3,4

20 READ A,B

30 PRINT "A =";A;"B =";B

READs the first two values from the DATA list.

```
40 READ C,D
50 PRINT "C =" ; C; "D =" ; D
60 RESTORE
```

READs the next two values from the DATA list.

```
70 READ X,Y,Z,
90 PRINT "X =" ; X; "Y =" ; Y; "Z =" ; Z
```

Resets the data pointer to the beginning of the DATA list.

```
RUN
```

READs the first three values from the RESTOREd DATA list.

```
A = 1  B = 2
C = 3  D = 4
X = 1  Y = 2  Z = 3
```

**DEC**

**Abbr. none**

**DEC (string)**

Finds the decimal (base 10) value of a hexadecimal base 16) number. Hexadecimal base digits are 0 through F, which equals decimal 15. The hexadecimal number, which must be a string expression, must be between 0 and hexadecimal value \$FFFF, which is equal to decimal 65535. (The dollar sign preceding a number is used to indicate that the number is hexadecimal but should NOT be included in the string sent to the DEC function.) The DEC function returns the unsigned value of the hexadecimal number. To get the 16-bit two's complement, X, of a hexadecimal number, X\$, use

```
X = DEC(X$)+(DEC(X$)>32767)*65536
```

**Example:** PRINT DEC("1E"); DEC("10"); DEC("A")
30 16 10

**DEF FN**

**Abbr. dE fn**

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

Defines a calculation as a function. DEF FN saves time and errors by sparing you from having to reenter a calculation you will use more than once in a program. After the function is defined as a formula, you can use it to solve a specific problem. To do so, call the function and supply the value you want the formula to solve, with FN name (value).

**Parameters:** function name (variable) = calculation

1. The function name is any legal variable name. When you want to use the function later in the program, you give FN followed by the function name.
2. The (variable) is replaced by a value when you call the function you defined.

This replacement is how you use the generic formula you defined in the DEF FN to solve a specific calculation.

3. The calculation must follow the rules for calculations.

**Note:** If BASIC RAM is moved by a GRAPHIC command after defining a function, the function may not be evaluated properly. Enter (and immediately leave if necessary) the graphic mode before you define the function.

Example:	10 DEF FNX(Y) = INT(A * 2 + Y) 15 INPUT A 20 PRINT FNX(35.2); FNX(19.9)  RUN ? 5 45 39	Defines the formula for function X.  Calls function X to use its formula to solve for 35.2 and then for 19.9, which replace Y in the function formula.
----------	--	--

## DELETE

Abbr. deL

**DELETE** line number-line number

Deletes BASIC program lines. You can issue this command only in immediate mode, not in a BASIC program.

**Parameter:** line number(s)

You can delete one line at a time or a group of lines. To delete one line, just enter the line number after the word DELETE. To delete a group of lines, enter DELETE, then the first line number, a dash, and the final line number.

You can also delete all the lines from the beginning of the program up to a certain line by entering DELETE followed by a dash and the last line you want to delete. To delete all the lines from a certain line to the end of the program, enter DELETE, the first line you want to delete and a dash.

Examples: **DELETE 75** Deletes line 75.

**DELETE 150-250** Deletes lines 150 through (and including) 250.

**DELETE -90** Deletes all lines up to and including 90.

**DELETE 140-** Deletes line 140 and all following lines to the end of the program.

DIM

### Abbr. dI

**DIM array name (subscripts), array name (subscripts), etc.**

Defines an array, which is also called a matrix. An array is a table of related values that you can use as a unit or as individual data items. You can refer to any

element of the array by giving the array variable name and the subscripts in the array where the element is located.

The DIM command names the array and defines the number of elements in the array. An array can have one, two, or more dimensions. If you use an array element without first DIMensioning the array, the computer gives the array the default number of elements (11).

You cannot change the dimensions of an array after you have DIMensioned it, or after you have accepted the default dimensions. If you DIM the array after you have used it, or try to reDIM the array, the program is aborted and the error message REDIM'D ARRAY is displayed.

The first element in any dimension of an array is numbered 0, not 1. This means that an array dimensioned as (5,3) is actually 6 by 4. When you figure the number of elements in an array, add 1 to each dimension, then multiply the results of the additions. For example, if the array is dimensioned DIM K(2,4), the array contains  $(2 + 1) * (4 + 1) = 15$  elements.

**Parameters:** array name (subscripts), array name (subscripts), etc.

The default number of elements is 11 (0–10).

1. The array name is a variable that follows standard variable rules. Arrays containing text elements must have text-string variable names (e.g., A\$). Arrays containing numeric elements must have a numeric variable name.

2. The subscripts set the number of elements in each dimension of the array.

You can define more than one array in a DIM command. Separate multiple array dimensions with a comma.

You can use arrays with more than two dimensions by supplying additional subscripts in the dimension command. For example, to DIMension a four-dimensional array, you can use DIM A(2,2,3,2).

Examples:	10 DIM G(9)	Defines a one-dimensional array with ten elements.
	20 DIM G\$(3,5)	Defines a two-dimensional text array with 24 elements (3+1 rows times 5+1 columns).
	30 DIM H(2,3,4)	Defines a three-dimensional array with 60 elements (2+1 times 3+1 times 4+1).
	90 PRINT G\$(2,2)	Prints the element at row 2, column 2.
	100 INPUT A(3)	INPUTs a value for element 3 in array A. Since array A has not been defined in a DIM command, it is given the default number of elements (11).

**DIRECTORY**

Abbr. diR

**DIRECTORY Ddrive, Uunit, file name**

Displays the following information about the contents of a disk:

- Names of all files on the disk
- The length of each file in blocks
- How much storage space remains on the disk

Press CONTROL and S to suspend the display, and any key to resume display. Hold down **C** to slow the display.

Each 1541 disk can contain up to 664 blocks of information. You should check to see how many blocks remain free before you COPY files onto a disk. You should also check before you save a file if you think the disk is nearly full.

**Parameters:** D drive number, U unit number, "file names or prefixes"

1. Drive numbers are either 0 or 1. No other numbers are allowed. The drive number must be preceded by D (e.g., D0). You do not need this parameter if you are using a single drive such as the 1541, or if you are accessing drive 0 of a dual drive.

2. U unit number is an optional parameter. Use it only if you have more than one disk drive connected to your computer and you are accessing a device other than unit 8. You must precede the unit number with U. You can also type ON before U and the unit number, but ON is not required.

3. You can display a partial disk directory by specifying a file name. It is especially useful to use wild cards in the file name. For example, after you type DIRECTORY, add, in quotes, the beginning letters of the file names you want to list and then the \* sign. The \* sign stands for all the other letters in the file names you want to list. The command looks like this: DIRECTORY "beginning letters\*".

You can use the question mark as a wild card to stand for any single character in a file name.

**Note:** The drive and unit number parameters and the file name can be specified with a variable or expression in parentheses.

Examples:	DIRECTORY	Displays the complete list of files on the disk currently in the disk drive.
	DIRECTORY D1	Displays the directory for the disk in drive 1 of a dual drive.
	DIRECTORY U9, "LET*"	Displays a list of files whose names begin with the characters LET. Other files on the disk in unit 9 are not listed.
	DIRECTORY "TEST?"	Displays the files whose names are TEST and one additional character (e.g., TEST1, TESTX).

Loads a disk program into memory. You cannot use DLOAD to load programs from tape.

**Parameters:** “file name”, D drive number, U unit number

1. You must include the name of the file. Enter the name in quotes. You can use a variable name in place of the file name, but the variable must have a value, and it must be in parentheses (not in quotes). The only time this is likely to be useful is when you load a program from within another program.
  2. Drive numbers are either 0 or 1. No other numbers are allowed. The default value is 0. You do not need this parameter if you are loading from a single disk drive.
  3. Unit number is an optional parameter. Use it only if you have more than one disk drive connected to your computer and you are using a device other than unit 8 in the loading procedure. You must precede the unit number with U. You can also type ON before U and the unit number, but ON is not required.

**Note:** The drive and unit number parameters and the file name can be specified with a variable or expression in parentheses.

Note: Only program-type files can be DLOADED.

**Note:** In program mode, a RUN command (with no CLR) is automatically issued following a DLOAD operation. This makes it possible to chain programs.

**Examples:** DLOAD "CIRCLE\$"  
90 DLOAD (X\$)  
Loads file CIRCLE\$ from disk.  
Loads a file whose name is the current value of X\$.  
File X\$ is loaded during the execution of the  
current program.

**DO . . . UNTIL/WHILE/EXIT . . . LOOP**      **Abbrs. do/uN/wH.exI/loO**

**DO UNTIL** logical value **WHILE** logical value

## commands

EXIT

### **Exit**

**LOOP UNTIL** logical value **WHILE** logical value

Repeats execution of the commands between DO and LOOP. The DO loop cannot stop itself unless you add commands or clauses that set conditions for terminating the loop. UNTIL, WHILE, and EXIT are optional clauses that can be included to terminate a DO loop.

UNTIL and WHILE clauses, which control the number of loop executions, contain conditional formulas that are evaluated each time the loop repeats. EXIT lets you abort the loop.

## Parameters

<i>Required</i>	<i>Optional</i>
DO	UNTIL conditional formula WHILE conditional formula
Commands to be executed by the loop	EXIT
LOOP	UNTIL conditional formula WHILE conditional formula

1. The UNTIL clause usually contains at least one variable that is compared with a value. The condition of this comparison is checked each time the DO loop executes. The loop continues repeating until the condition(s) is (are) met. Program control then passes to the command after the LOOP command.

You can set multiple conditions by linking them with AND or OR (e.g., UNTIL X = 5 OR Y > 10).

**Example:** 20 DO UNTIL X = 10      This DO loop executes until X equals 10. When  
 30 PRINT X                                    this condition is met, the loop ends.  
 40 X = X + 2  
 50 LOOP

2. The WHILE clause usually contains at least one variable that is compared with a value. The condition of this comparison is checked each time the DO loop executes. The loop continues repeating while the condition(s) is (are) met. Program control then passes to the command after the LOOP command.

You can set multiple conditions by linking them with AND or OR (e.g., WHILE X = 5 OR Y > 10).

**Example:** 20 DO WHILE X < 10      This DO loop executes until X is greater than or  
 30 PRINT X                                    equal to 10. When this condition occurs, the loop  
 40 X = X + 2  
 50 LOOP                                        ends.

**Notes:** The difference between UNTIL and WHILE is that UNTIL conditions start off not being met; the loop continues until they are. WHILE conditions start off being met; the loop continues until they are not met.

The conditions in UNTIL and WHILE commands are always either true (met) or false (not met). If you use more than one condition, join them with AND or OR. If you use AND, both conditions must be met; if you use OR, only one condition has to be met.

Both DO and LOOP can have UNTIL conditions or WHILE conditions, but not both.

You can have a conditional clause (WHILE or UNTIL) in both the DO and LOOP commands in one loop.

If you omit both UNTIL and WHILE clauses in the DO loop, the loop is an infinite loop: it continues executing without stopping. You must interrupt the program with the STOP key to terminate the loop.

3. EXIT lets you leave the loop before the UNTIL or WHILE conditions end the loop. You can, for example, use EXIT to check for unwanted values and end a loop if a particular value is encountered. After an EXIT command, program execution passes to the line following the LOOP command.

**Note:** Always use EXIT (never GOTO) to leave a loop prematurely.

Example:	5 DATA YES, NO, YES, NO, END	Lists DATA values.
	10 DO WHILE X < 50	Begins a loop that runs as long as X is less than 50.
	20    X = X + 1	Increments the counter for the WHILE clause.
	30    READ ANS\$	Reads data from line 5.
	40    IF ANS\$ = "END" THEN EXIT	Aborts the loop if ANS\$ = END.
	50 LOOP	Sends the loop back to DO.
	NEW	Clears the previous program.
	10 DO:PRINT "HALT!"	Begins a DO loop.
	20 X = X + 1	Adds 1 to X each time the loop executes.
	30 IF X = 25 THEN EXIT	Aborts the loop when X = 25.
	40 LOOP	Sends the loop back to DO.

4. LOOP works with DO to set conditions for a repeated sequence of program lines. LOOP works for DO as NEXT does for FOR: it marks the end of the loop and sends execution back to the beginning of the loop.

If you do not include an UNTIL or WHILE clause with the DO command, you can add one here. The UNTIL and WHILE commands can appear with either the DO command or the LOOP command, or both.

#### DRAW

Abbr. dR

DRAW color source, coordinates TO coordinates TO coordinates etc.

Draws dots, lines, and any angled shape. DRAW can be used only in one of the graphic modes. Though you can draw any polygon with DRAW, it is sometimes simpler to use CIRCLE to draw polygons with regular-length sides. See Chapter 4 for more information on DRAW coordinates.

<i>Parameter</i>	<i>Values</i>	<i>Default</i>
Color source	0–3	1
Coordinates		Pixel cursor
Column coordinate		
High-res modes	0–319	
Multicolor modes	0–159	
Row coordinate	0–199	
TO coordinates		
Column coordinate		
High-res modes	0–319	
Multicolor modes	0–159	
Row coordinate	0–199	
TO column, row, etc.		

1. The color source indirectly selects the color for the drawing. There are five color sources, but color source 4 (the border color) cannot be used in drawing commands. Color sources 2 and 3 can be used only in multicolor modes.

- 0 screen background color
- 1 foreground color
- 2 multicolor mode extra color 1
- 3 multicolor mode extra color 2

The color-source number you include in the DRAW command tells the computer to draw in the current color for that source. For example, if you select 1, the computer draws the shape in the current foreground color. If you want to use a color other than one of the current source values, you must first use the COLOR command to change one of the source values. Only sources 1 and 2 can be used to draw with more than one color on the same screen. Sources 0 and 3 are global (whole screen) colors.

If you want to use the default value (1, the current foreground color), you do not have to type a number, but you must type a comma before the next parameter.

2. The first coordinates name the starting point of the drawing. If you omit this parameter, the current location of the pixel cursor is used as the starting point. If you omit the first coordinates, no placeholder comma is required.

3. TO is a required part of the DRAW command unless you are just drawing a dot.

4. The second and subsequent coordinates name the ending points of the line segments. You can add more than one TO clause to draw complex designs. If you

do, the ending point of the first line segment becomes the starting point of the next line segment, etc.

**Examples:** 5 GRAPHIC 1,1  
10 DRAW, 30,25 TO 289,150

Draws a line from column 30, row 25 to column 289, row 150.

20 DRAW, 160,25 TO 310,50 TO  
240,100 TO 160,100 TO 80,50  
30 DRAW TO 319,50

Draws a four-sided open shape.

Draws a line from the current pixel-cursor location to column 319, row 50.

## DS

### Abbr. none

You can PRINT DS to display a reading of the disk drive error number, or you can examine DS in a program when you need to know the drive status. Use DS with DS\$ to find out why the red error light on the disk drive is blinking after a disk operation such as DLOAD. If no error occurred, DS is zero.

## DS\$

### Abbr. none

You can print DS\$ to display a message explaining the drive status. Use DS\$ with DS to find out why the red error light on the disk drive is blinking after a disk operation such as DLOAD. The error messages are listed in Appendix A.

**DSAVE**                            **Abbr. dS**  
**DSAVE file name, Ddrive, Uunit**

Stores the current program onto a disk. You cannot use DSAVE to store programs onto cassette tape.

**Parameters:** "file name", D drive number, U unit number

1. You must include the name of the file. Enter the name in quotes. You can use a variable name in place of the file name, but the variable must have a value, and it must be in parentheses (not in quotes).
2. Drive numbers are either 0 or 1. No other numbers are allowed. The default value is 0. You do not need this parameter if you are storing onto a single disk drive.
3. Unit number is an optional parameter. Use it only if you have more than one disk drive connected to your computer and you are using a device other than unit 8 in the loading procedure. You must precede the unit number with U. You can also type ON before U and the unit number, but ON is not required.

**Note:** The drive and unit number parameters and the file name can be specified with a variable or expression in parentheses.

**Examples:** DSAVE "BOXES"  
90 DSAVE (A\$), U9

Stores the program BOXES onto the disk.  
Stores a file onto the disk in drive unit 9. The name of the file is the current value of A\$. The program is saved during execution.

**EL****Abbr. none**

Determines the line number of the last BASIC error that occurred in a program. Use PRINT EL to display the line number. The reserved variable EL is often used in conjunction with the TRAP command, which isolates errors without interrupting program execution.

**ELSE****Abbr. eL**

An optional clause you can add to an IF . . . THEN . . . ELSE command. See IF . . . THEN . . . ELSE.

**END****Abbr. eN**

Ends a program with no message. You need to end a program with the END command when subroutines or trap routines follow the body of the program. You can also use END somewhere in the body of the program to terminate the program if some condition is met.

**Example:** 50 IF A\$="STOP" THEN END      Ends the program if A\$ equals STOP.

**ER****Abbr. none**

Determines the error number of the last BASIC error that occurred in a program. Use PRINT ER to display the error number. The reserved variable ER is often used in conjunction with the TRAP command, which isolates errors without interrupting program execution. The BASIC errors are listed in Appendix A.

**ERR\$****Abbr. eR****ERR\$ (number)**

Returns the error message describing the BASIC error number in parentheses. The function ERR\$ is often used in conjunction with the TRAP command, which isolates errors without interrupting program execution. To display the error message for the most recent error in the program, type PRINT ERR\$(ER).

**Parameter:** a numeric expression with value 1–36

The BASIC error messages are listed in Appendix A.

**EXIT****Abbr. exI**

Terminates a DO . . . LOOP conditional command sequence. You cannot use EXIT with other commands, including IF . . . THEN . . . ELSE sequences. See DO.

**EXP****Abbr. eX****EXP (number)**

Numeric function that finds the value of the mathematical constant  $e$  (approximately 2.71828183) raised to the power in parentheses. To find exponentials of other numbers, use the up arrow symbol.

**Parameter:** any numeric expression

**Example:** PRINT EXP(-1)      Displays the reciprocal of  $e$ .  
.367879441

**FOR . . . TO . . . STEP . . . NEXT      Abbrs. fO/to/stE/nE**

**FOR variable = start value TO end value STEP increment  
commands**

**NEXT variable, variable, etc.**

Creates a loop to repeatedly execute all commands between the FOR command and the NEXT command. The loop repeats until the counter variable in the FOR command equals or exceeds the value of the ending point.

If you omit the optional STEP command, the FOR counter is incremented by 1 each time the loop executes. You can use STEP to increment the counter by any number. For example, you can use a STEP increment of 25 to draw a shape repeatedly, each time tilted 25 degrees more. You can also use STEP to count backwards by specifying a negative number.

**Parameters:** FOR counter variable = starting point TO ending point STEP increment  
commands

NEXT counter variable

1. The FOR command contains a variable whose value is updated each time the loop is executed.

The FOR command also contains the starting and ending points for the number of times the loop will execute. The starting and ending points can have any value, including variables. If the starting point is higher than the ending point, you normally include a negative STEP value.

2. The STEP clause, which is optional, tells the computer how much to add to the current value of the counter each time the NEXT command sends execution back to the FOR command. The default value is 1.

A negative STEP value decreases the value of the counter variable each time the loop is executed. (Remember that adding a negative value is like subtracting.)

3. The commands that are to be repeated during loop execution appear between the FOR command and the NEXT command.

4. The NEXT command tells the computer to go back to the FOR command. When the FOR command is reached again, the STEP value is added to the value of the counter. When the counter passes the ending point, the loop terminates. Never leave a FOR . . . NEXT loop with a GOTO. You can always leave when NEXT is executed by setting the loop variable equal to its ending point.

**Note:** A FOR . . . NEXT loop is always executed once. The counter variable is updated before leaving the loop.

The NEXT command can contain the FOR counter variable, but it is not required. If you are nesting loops, use the counter variable in the NEXT commands to avoid errors. The NEXT command can also contain a number of counter variables for nested loops. The variables must be listed starting with the innermost loop variable and ending with the outermost loop variable, therefore, nesting rules are followed.

**Nesting Multiple FOR . . . NEXT Loops** You can nest up to 10 FOR . . . NEXT loops. Note the following when you nest loops:

- The inner loop becomes part of the outer loop.
- The inner loop must start and end between the beginning and ending of the outer loop.
- The inner loop executes a full cycle from starting to ending points each time the outer loop executes once.

Chapter 3 contains more information on using loops in BASIC programs.

Examples:

```
10 FOR X = 10 TO 50 STEP 15
20 PRINT "X ="; X
30 NEXT
40 PRINT "AT THE END OF THE LOOP, X ="; X
```

```
RUN
X = 10
X = 25
X = 40
AT THE END OF THE LOOP, X = 55
```

NEW

```

10 FOR Y = 1 TO 3
20 FOR Z = 6 TO 1 STEP -2
30 PRINT "Z ="; Z;
40 PRINT "Y ="; Y
50 NEXT Z: NEXT Y      This line could also read NEXT Z,Y.
RUN

```

Z = 6    Y = 1	While Y, which is part of the outer loop, executes once, Z, from the inner loop, executes all three times.
Z = 4    Y = 1	When Y makes its second execution, Z repeats another full set, and so on.
Z = 2    Y = 1	
Z = 6    Y = 2	
Z = 4    Y = 2	
Z = 2    Y = 2	
Z = 6    Y = 3	
Z = 4    Y = 3	
Z = 2    Y = 3	

**FRE**                          **Abbr. fR**  
**FRE (number)**

Examines the number of available bytes of RAM. Use PRINT FRE(0) to display the amount of memory available. The FRE function uses a dummy argument, which means that the number in parentheses is meaningless. You have to include the parameter anyway; just type FRE(0).

**GET**                          **Abbr. gE**  
**GET input list**

Like INPUT, GET accepts input from the keyboard during program execution. GET, however, accepts only a single character at a time as data entry. In addition, GET does not wait for input, but returns a null (empty) string if no key is pressed. This allows you to check repeatedly for keyboard entry while other operations continue. To force the computer to wait for input, use the GETKEY command.

**Parameter:** variable(s)

The variable is nearly always a string variable. It stands for the key to be typed in response to the GET command. Use of a numeric variable allows only the 0 through 9 keys to be entered. Any other key causes a type mismatch error, which aborts the program unless TRAPPED.

<b>Example:</b>	10 PRINT "PRESS A KEY TO STOP ME"	Prints message to the screen.
	20 GET A\$:IF A\$="" THEN 10	If no key is pressed, then go to 10.
	30 PRINT "WHAT A RELIEF"	Program continues normally.

**GETKEY****abbr. gEkE****GETKEY input list**

Like INPUT, GETKEY accepts input from the keyboard during program execution. GETKEY, however, accepts only a single key at a time as data entry. Unlike GET, GETKEY waits for input.

**Parameter:** variable(s)

The variable is nearly always a string variable. It stands for the key to be typed in response to the GETKEY command. Use of a numeric variable allows only the 0 through 9 keys to be entered. Any other key causes a type mismatch error, which aborts the program unless TRAPped.

**Example:** 10 ? "PRESS THE CURSOR KEY TO ANSWER"  
 20 ? "MY GROUP IS: <- RED OR BLUE ->"  
 30 GETKEY A\$

40 IF A\$ = CHR\$(157) THEN ? "RED GROUP,  
 DO ODD-NUMBERED EXERCISES"

50 IF A\$ = CHR\$(29) THEN ? "BLUE GROUP,  
 DO EVEN-NUMBERED EXERCISES"

Waits for you to  
 press a key; the key's  
 value is assigned  
 to A\$.

Checks key entered.  
 CHR\$(157) is cursor-  
 left key code.

Check key entered.  
 CHR\$(29) is cursor-  
 right key code.

**GET#****abbr. gE#****GET# file number, input list**

Retrieves data one character at a time from an OPENed device or file. GET# works like a GET command except that the GET command gets a character from the keyboard, while the GET# command gets a character from a device or a file. GET# works like INPUT# except that INPUT# gets a whole group of characters from the file, while GET# gets only one character at a time.

**Parameters:** file number, variable(s)

1. The file number is a logical file number that links the file or device to other commands, including the OPEN command that accesses the device or file before it can be used.
2. The variable is nearly always a string variable. Use of numeric variables allows only values from ASC("0") (48) to ASC("9") (57) to be read. Anything else causes a type mismatch error, which aborts the program unless TRAPped.

Example:	10 OPEN 8,8,2,"\$"	Opens communication to the disk directory file.
	20 DO UNTIL ST<>0	Repeats a loop until the status byte indicates an end-of-file or an error.
	30 GET#8 ,K\$	Assigns one character from the disk file to K\$.
	40 IF(ASC(K\$)AND127>31 THEN PRINT K\$	Prints noncontrol characters.
	50 LOOP	
	60 CLOSE 8	Closes disk file.

**GOSUB line number**

Branches the program to a subroutine, which is a group of program lines that performs a reusable task. You can reuse a subroutine as often as you like just by calling it with the GOSUB command. Subroutines are particularly useful in programs that repeat a task.

Subroutines can appear anywhere in the program. They are ended by the RETURN command, which sends program control back to the main body of the program. Do not exit subroutines with a GOTO command. The program continues at the line following the GOSUB command.

**Parameter:** starting line number for subroutine

Example:	10 INPUT "WHAT SHAPE DO YOU WANT TO DRAW"; SS
	20 INPUT "DO YOU WANT THE SHAPE TO BE RED, BLUE, OR GREEN"; CS
	30 IF SS <> "CIRCLE" THEN INPUT "HOW MANY SIDES DOES THE SHAPE HAVE"; X
	35 IF SS = "CIRCLE" THEN X = 180
	37 IF X = 0 THEN PRINT "ZERO IS NOT ALLOWED. REENTER":GOTO 30
	40 GOSUB 80
	45 REM AFTER SUBROUTINE, PROGRAM RESUMES AT LINE 50
	50 INPUT "WANT TO DRAW ANOTHER SHAPE"; AS
	60 IF LEFT\$(AS,1) = "N" THEN END: ELSE GOTO 10
	70 REM BEGIN SUBROUTINE TO DRAW SHAPE
	80 GRAPHIC 1,1
	85 REM USE INPUT FROM LINE 20 TO SET COLOR IN LINE 110
	90 IF LEFT\$(CS,1) = "R" THEN C = 3: GOTO 110
	100 IF LEFT\$(CS,1) = "B" THEN C = 7: ELSE C = 6
	110 COLOR 1,C,3
	105 REM USE INPUT FROM LINE 30 TO FIGURE NUMBER OF SIDES OF SHAPE
	120 CIRCLE, 160,100,60,50,,,360/X

```
130 PAINT, 160,100
140 CHAR,2,2, "YOU HAVE DRAWN A "+S$
145     REM USE DELAY LOOP TO PROLONG SHAPE DISPLAY
150 FOR Y = 1 TO 300: NEXT Y
155     REM SWITCH BACK TO TEXT MODE
160 GRAPHIC 0,1
165     REM END OF SUBROUTINE
170 RETURN
```

**GOTO or GO TO      Abbr. gO**  
**GOTO line number**

Tells the computer to branch to another line and continue execution there. You can also use GOTO in immediate mode to jump into a program and begin executing it. The difference between executing a program with a RUN command and an immediate mode GOTO is that the RUN command clears all variables before executing the program, and GOTO does not.

**Parameter:** line number

GOTO can send execution to a later or earlier line in the program. When GOTO sends the program back to a previously executed line, an infinite loop results unless you also include a statement or mechanism such as a counter to end the loop.

Examples: 10 INPUT"WHAT'S YOUR NAME"; N\$  
20 GOTO 10

NEW

```
10 INPUT "WANT TO REPEAT"; N$
20 IF LEFT$(N$,1) = "N" THEN END
30 GOTO 10
```

GOTO sends the program back to line 10 each time line 20 is executed; this causes an infinite loop.

GOTO sends the program back to line 10 unless you answer N to the INPUT question. Typing N ends the loop.

**GRAPHIC****GRAPHIC mode, clear****GRAPHICCLR****Abbr. gR**

Switches to one of the four graphic drawing modes, or from a graphic drawing mode to the text modes. You can also clear the screen or the bit-mapped memory area that is set aside for graphics when you enter one of the graphic drawing modes.

<i>Parameter</i>	<i>Values</i>	<i>Default</i>
Mode	0–4	No default
Clear screen option	0 or 1	0
CLR command option	CLR	No default

1. You can use the GRAPHIC command to switch to any of these modes:

- 0 text mode
- 1 high-resolution mode
- 2 split-screen high-resolution mode
- 3 multicolor mode
- 4 split-screen multicolor mode

GRAPHIC 0 switches from a drawing mode to text/graphic mode, in which letters are uppercase and you can print all the graphic symbols on the fronts of the keys. If you want to switch to the alternate text mode, in which letters can be both lowercase and uppercase and in which you can print only the left-side graphic-key symbols, first switch to text/graphic mode, and then press the SHIFT and **G** keys together or PRINT CHR\$(14).

2. If you want to include a screen-clearing option with the mode choice, add a comma and 1 after the mode number. If you do not add this option, the computer switches back to whatever was on the screen the last time it was used in the current computing session.

3. The GRAPHIC command can also release the 12K area devoted to graphic modes. The first time you access any one of the graphic drawing modes, a 12K area is set aside for the graphics screen and your BASIC program is moved above it. When you return to a text mode, this area remains set aside for graphic mode use unless you tell the computer to release it. Just issue a GRAPHIC CLR command to regain use of this 12K of memory.

**Note:** If you define a function with DEF FN before you execute a GRAPHIC command, the function subsequently may not execute properly (when FN is used). In other words, the function definition is not moved properly. Care must be taken when defining functions and using graphics in a single program.

**Note:** A graphics screen actually requires only 10K bytes of memory. The amount removed from BASIC RAM is 12K because the screen must be located on 8K boundaries. The screen is located from \$2000 to \$3FFF; its color/luminance memories are located from \$1800 to \$1FFF. This leaves the 2K from \$1000 to \$17FF unused, but unavailable for BASIC.

Examples:	<b>GRAPHIC 0,1</b>	Switches from a graphic mode to the text modes. The last text screen is cleared.
	<b>GRAPHIC CLR</b>	Releases the 12K bit-mapped graphic area.
	<b>GRAPHIC 2,1</b>	Switches to split-screen high-resolution mode and clears the graphic screen.
	<b>GRAPHIC 3</b>	Switches to full-screen multicolor mode without clearing the graphic mode screen.

**GSHAPE**                  **Abbr. gS****GSHAPE string variable, coordinates, mode**

**GSHAPE** (GetSHAPE), which is the opposite of the **SSHAPE** (SaveSHAPE) command, retrieves and displays a graphic screen area saved by an **SSHAPE** command. You can use **SSHAPE** and **GSHAPE** in any graphic mode to store and retrieve a rectangular section of the screen that is up to 255 text-sized characters long.

These graphic screen areas are saved as text-string values in memory. You use a text-string variable to identify the screen area, just as you use a variable to identify any type of value.

After you save a screen area with **SSHAPE**, you can display it anywhere on the graphic screen. When you retrieve the area in the **GSHAPE** command, you give the screen location where you want the area to be displayed.

**Parameters:** string variable, top corner coordinate, display mode

1. The string variable is the name assigned to the graphic screen area saved with the **SSHAPE** command. Retrieve the area by using the same string-variable name used in the **SSHAPE** command.

2. Display a copy of the saved graphic screen area anywhere on the screen by giving the coordinates of the top left corner of the screen area where you want the drawing to appear.

3. When you retrieve the saved area, you can choose one of the five options for displaying it.

- 0 display duplicate of saved area (default)
- 1 display saved area in reversed colors
- 2 OR saved area with current area
- 3 AND saved area with current area
- 4 XOR saved area with current area

Option 0, which is the default value for this parameter, draws the area as you saved it.

Option 1 inverts the color values, so the shape is drawn as a reversed image of the saved area.

Option 2 overlays the shape on the existing screen pattern.

Option 3 displays only that part of the shape that covers an existing screen pattern.

Option 4 inverts the part of the existing screen pattern covered by the shape.

Chapter 4 further explains the GSHPA and SSHAP commands.

**HEADER**                  Abbr. heA

**HEADER disk name, Idisk id, Ddrive, ON Uunit**

Before you can store information on a new, blank disk, you must prepare the disk by formatting it. Formatting, also called headering, puts the blank disk into the format required by your disk drive. The disk is divided into blocks and a directory for the disk is prepared. Headering is necessary because blank disks are manufactured to be used in any brand of disk drive, and you need to format the disk so that it is compatible with your disk drive.

You MUST header a new disk before you can save files on it, but use the HEADER command with great care because headering completely and permanently erases any files already on the disk. You can header a used disk if you are willing to erase its current contents.

**Parameters:** "disk name", Iid code, D drive number, U unit number

1. The disk name (in quotes) can be up to 16 characters long.
2. Give the disk a unique two-character code. Use two characters for an id code, not just one. Type an I before the id code.
3. Drive numbers are either 0 or 1. The order of the id code and drive number parameters can be reversed.
4. Unit number is an optional parameter. Use it only if you have more than one disk drive connected to your computer and you are using a device other than unit 8 in the header procedure. You must precede the unit number with U, and the unit number must be between 8 and 11. You can type ON before U, but ON is not required.

### Are You Sure?

When you issue a HEADER command and press RETURN in immediate mode, the command is not executed immediately. First, the computer displays the question ARE YOU SURE? This question gives you a chance to make sure the disk does not contain information you want to keep.

To proceed with the headering procedure, type Y or YES and press RETURN. To abort the header, just press RETURN. In program mode, the question is not asked.

## Partial Headering

You can also clear a disk directory on an old disk without formatting the disk. This procedure, which gives you an empty disk with the old id, is called a partial header. Omit the id code from the HEADER command to do a partial header.

**Note:** The drive and unit number parameters and the disk name can be specified with a variable or expression in parentheses.

Examples: HEADER "CIRCLES",DO,IG3

HEADER "INSURANCE",D1,IP5

HEADER "HOUSEFILES",DO

Performs a partial header.

HELP

Abbr. heL

Highlights an erroneous command in a BASIC program by putting the command in flashing mode. If you want to highlight the error in a line, use HELP after the computer displays an error message when you execute a program. The HELP function key is defined with this BASIC command.

HEX\$

Abbr. hE

HEX\$ (number)

Gets the hexadecimal value for the decimal number in parentheses as a four-character text string. The value of the number in parentheses must be between 0 and 65535 inclusive. Since the hexadecimal value is always printed as a four-character string, zeros are placed at the beginning of values that are less than four characters long.

**Note:** The HEX\$ function accepts only nonnegative input. To use 16-bit two's complement input, use

X\$ = HEX\$(X-(X<0)\*65536)

Example: PRINT HEX\$(45), HEX\$(2001)

002D        07D1

IF . . . GOTO . . . ELSE

Abbrs. if/gO/eL

IF logical value GOTO line number : ELSE commands

Branches the program based on the value of a conditional clause. IF is a compound conditional statement that checks the status of a condition in the command and then chooses one of two courses of action.

One of the two IF command options is stated in the GOTO clause, which is executed when the IF condition is true. When the IF condition is false, the GOTO

clause is ignored and execution passes to the next line in the program or to the ELSE clause if one is present.

The GOTO clause is like a GOTO command: it tells the computer to go straight to a specified line number and resume execution there. The line number can be anywhere in the program.

IF ... GOTO ... ELSE is a limited variation of IF ... THEN ... ELSE. Use IF ... GOTO ... ELSE instead of IF ... THEN ... ELSE when the THEN clause would contain a GOTO command anyway.

**Parameters:** true-false condition GOTO line number : ELSE clause

1. The conditions in the IF command can use comparison operators (=, <, >, <>, <=, >=) to compare values. The values can be any of the following:

- Numbers or text strings
- Any type of variable
- Variables on both sides of the comparison operator
- Mathematical formulas

2. The line number after GOTO tells the computer where to go when the IF condition(s) is (are) true.

3. The ELSE clause contains instructions that are followed only when the IF condition(s) is (are) false. The ELSE clause is always optional. It must be separated from the rest of the command by a colon.

**Example:**    10 INPUT X                          Line 20 compares the value input for X to 0.  
               20 IF X <= 0 GOTO 10                 The GOTO command executes only when it is  
               30 PRINT X                                 true that X is less than or equals 0.

**IF ... THEN ... ELSE                          Abbrs. if/tH/eL**

**IF logical value THEN commands : ELSE commands**

IF is a compound conditional statement that checks the status of a condition in the command and then chooses one of two courses of action.

One of the two IF command options is stated in the THEN clause. The other option can be stated in an ELSE clause. If no ELSE clause is present, execution continues with the next line in the program when the condition is false.

The status of the IF command condition determines whether the THEN clause or the alternative is to be executed. THEN is executed when the IF condition is true, or met; ELSE is executed when the IF condition is false, or not met. The computer executes either the THEN clause or the ELSE clause, but never both.

**Parameters:** true-false condition THEN clause : ELSE clause

1. The conditions in the IF command can use comparison operators (=, <, >, <>, <=, or >=) to compare values. The values can be any of the following:

- Numbers or text strings
- Any type of variable
- Variables on both sides of the comparison operator
- Mathematical formulas

2. The THEN clause contains commands that are executed only when the IF command condition(s) is (are) true. The THEN clause, which is always a required part of the IF command, can contain any legal commands. (If the THEN clause contains more than one command, they must be separated by colons.) The THEN clause must be typed on the same line as IF with no punctuation separating it from the keyword THEN. If the THEN clause is a GOTO, the keyword GOTO can be omitted.

Example: 50 IF X\$ = "HALT" THEN END

The program ends when X\$ does equal HALT. The THEN clause is not executed otherwise.

3. The ELSE clause contains a command that is executed only when the IF condition(s) is (are) false. The ELSE clause can contain any legal command. If the ELSE clause is a GOTO, the keyword GOTO can be omitted. The ELSE clause is always optional, but the THEN clause is always required, so an IF command cannot have an ELSE clause but no THEN clause.

ELSE must be separated from the THEN clause by a colon. ELSE is a clause, not an independent command; type THEN and ELSE clauses on the same lines as the IF command.

**Note:** IF commands can be “nested,” but their ELSE commands will not be. When an IF condition is found to be false, the next ELSE clause on the line is always executed.

Examples: 30 IF A = B THEN PRINT  
"EQUALITY": ELSE GOTO 100

When A equals B, the THEN clause executes and the ELSE clause does not. When the condition is false, the THEN clause does not execute, and the ELSE clause does.

40 IF A = 0 THEN IF B = 0  
THEN PRINT "BOTH 0": ELSE  
PRINT "ONE NONZERO"

Prints BOTH 0 if A and B are zero. Prints ONE NONZERO if either is not zero.

<b>INPUT</b>	<b>Abbr. none</b>
<b>INPUT string; input list</b>	

Accepts your input from the keyboard during program execution. The program waits for you to type the input and press the RETURN key before it continues. You can add a question to the INPUT command to help the user understand the type of input expected.

**Parameters:** “prompt question”; variable(s)

1. The prompt question, which is optional, must be in quotes. If you omit the prompt, do not put the semicolon before the variable.

Do not type a question mark at the end of the question. Whether or not you include a prompt question, INPUT displays a question mark to indicate that keyboard input is expected. If you add the prompt question, the automatic question mark is displayed at the end of the question.

**Note:** If you do not want a question mark to be displayed for keyboard input, OPEN the keyboard (device number 0) as a file. Then, PRINT your prompt (ending with a semicolon) and use INPUT# to read the keyboard.

2. The data values you input from the keyboard are assigned to the INPUT variable. Use text-string variables for text input. You can use more than one variable in an INPUT command. If you do, separate the variables with commas; the semicolon is used only to separate the prompt question (if there is one) from the variables. If you use more than one variable, you must enter a value for each (separated by commas) before the program can continue. Otherwise, a double question mark will prompt for the rest of the input.

**Example:**    10 INPUT "WHAT'S THE DESTINATION"; D\$  
              20 PRINT "PACKAGE TO ";D\$

RUN

WHAT'S THE DESTINATION? LONDON  
PACKAGE TO LONDON

<b>INPUT#</b>	<b>Abbr. iN</b>
<b>INPUT# file number, input list</b>	

Retrieves a data value from an OPEN file or device and assigns it to variables. INPUT# works like INPUT, but instead of getting data input from the keyboard, INPUT# gets data from a file or device. The file or device must have been opened using the same logical file number. See also GET#.

**Parameters:** file number, variable(s)

1. The file number is a logical file number that identifies the file or device and links it to other commands. The file or device must have been previously accessed by an OPEN command with the same logical file number.
  2. The variable type must match the type of value to be assigned (e.g., if you are assigning text values, you must use text-string variables). If the INPUT# command contains more than one variable, separate the variables with commas.

Example:	10 OPEN 8,8,15 20 INPUT#8, N, E\$, T, S	Accesses the disk drive error channel. Gets three numeric and one text-string values from the channel and assigns them to N, E\$, T, and S.
	30 PRINT N, E\$, T, S 40 CLOSE 8	Prints the values on the screen. Closes the disk channel

INSTR

Abbr. in S

**INSTR (master string, substring, start position)**

You can find the position of a text string within another text string by using the INSTR function. INSTR returns a number that represents the character position in the master string where the sought string begins. If the sought string is not present, a value of 0 is returned.

The INSTR function has an optional parameter that lets you begin the text-string search at any character location in the master string. Use this option if you have found one instance of the sought string and want to search for additional appearances of the sought string or if you want to begin the search after some known occurrence of the sought string. This option is the only way to find additional instances of the sought string.

**Parameters:** master string, sought string, starting position

1. The master string is the text string being searched. It can be any text string enclosed in quotes. You can also use a text-string variable or string expression as this parameter. Only text-string values are allowed.

Note that blanks and punctuation marks are counted as character positions.

2. The sought string is the text for which you are searching the master string. The sought string can be any text string enclosed in quotes. You can also use a text-string variable or string expression as this parameter. Only text-string values are allowed in the INSTR function.

3. The starting position, which is optional, is a number representing the character position in the master string where you want to begin the search. The default is the first position in the master string. Once you have found one instance of the sought string, you can search for another by issuing another INSTR command using the location of the found string + 1 as the starting position.

Examples: 10 A\$ = "THE LAST STRAW"  
 20 PRINT INSTR(A\$, "ST")

RUN

7

PRINT INSTR(A\$, "ST", 8)  
 10

The sought string is found starting at character position 7.

Using 7 + 1 as the starting location, another instance of the sought string is found at character position 10.

**INT**

**INT (number)**

**Abbr. none**

Truncates a number with decimal parts into a whole integer number. The INT function simply ignores the decimal parts of the number; INT does not round the number. This means that the result is always less than or equal to the original number. For example, INT(9.9) is 9, not 10.

When the number is negative, the result is also always less than or equal to the number. In the case of negative numbers with a decimal value greater than .0, INT returns the next lowest integer. For example, INT(-5.1) is -6.

The INT function is often used with the RND (random number) function to generate random whole numbers. See the RND function.

**Parameter:** number in parentheses

The number can be any number, positive or negative. You can also use a calculation or variable as the number.

**Note:** To round off a number, X, use INT (X+.5)

Examples: PRINT INT(-5.0)

-5

PRINT INT(2.2\*3)

6

**JOY**

**JOY (port number)**

**Abbr. jO**

Finds the status of either joystick. Use JOY(1) to examine the status of the joystick in joy port 1; use JOY(2) to examine the status of the joystick in joy port 2.

The JOY function reads nine different joystick positions, which are numbered 0 through 8. Nine additional readings, numbered 128 through 136, are displayed when the fire button is also being pressed. The readings are shown in Table I-5.

TABLE 1-5. Joystick readings

	<i>Left</i> & up	<i>Left</i> Left	<i>Left</i> & down	<i>Down</i>	<i>Right</i> & down	<i>Right</i> Right	<i>Right</i> & up	<i>Up</i>	<i>Middle</i>
NO FIRE									
BUTTON	8	7	6	5	4	3	2	1	0
WITH FIRE									
BUTTON	136	135	134	133	132	131	130	129	128

Examples: PRINT JOY(1)

4

Joystick 1 is positioned down and to the right. The fire button is not being pressed.

PRINT JOY(2)

134

Joystick 2 is positioned down and to the left. The fire button is being pressed.

KEY

Abbr. kE

KEY number, definition

Defines a function key and can also display an up-to-date list of the function key definitions.

Display a list of each function key definition by typing the command KEY and pressing the RETURN key. Do not add any parameters.

Redefine a function key by supplying values for the following parameters:

**Parameters:** key number, definition

- Type the key number of the key you are redefining. You must follow it with a comma. If you are just displaying a list of key definitions, omit this parameter.

- Type the key definition as a text string. You can use BASIC functions and any non-BASIC word in quotes. For a compound definition, join the strings with plus signs (+).

Put the command in quotes. Add +CHR\$(13) if you want an automatic RETURN at the end of the definition. Add +CHR\$(34) if you want to use quotation marks.

Examples: KEY

KEY 2,"GRAPHIC 2,1"+CHR\$(13)

Lists the current key definitions.

Defines key 2 to execute a GRAPHIC 2,1 command.

KEY 3,"INPUT"+CHR\$(34)

Defines key 3 to display INPUT" on the screen.

## Defining a Function Key for Program Input

The function key definition procedure can also be used in a program. INPUT can be used to accept function key definitions. Of course, the input must end with a RETURN character from the definition or the keyboard. GETKEY receives only the first letter of the definition. Also, if GETKEY is called a second time following the receipt of a multiple character function key definition, an error results.

To be able to use a function key in a GETKEY command, you must first redefine the key as a single CHR\$ code. This definition allows BASIC to consider the function key as a single key not a string of characters. Once the key is defined as a single key, you can press the key as input for a GETKEY command. Then you can use an IF command to see if the key pressed equals the CHR\$ code for the function key and use a THEN clause to perform the desired operation(s). The following example redefines function keys 1 and 2 as CHR\$ codes 133 and 137 (these are the CHR\$ codes used for the function keys on the Commodore 64).

Note that redefinitions written in a program are still in effect when the program ends. To restore the original definitions, press the reset button.

Examples:

```

5  REM DEFINE KEYS 1 AND 2 AS CHR$ CODES 133 and 134
10 KEY1,CHR$(133): KEY2,CHR$(134)
20 GETKEYZ$: REM PRESS F1 OR F2
25 REM USE ASC TO CHECK THE CHR$ CODE FOR THE PRESSED
    KEY
30 IFASC(Z$)=133 THEN PRINT"DRAW A CIRCLE":X=1
40 IFASC(Z$)=134 THEN PRINT"DRAW A DIAMOND":X=360/4
50 GRAPHIC1,1
60 CIRCLE,160,100,60,50,,,X

```

In this example keys 1 and 2 are redefined to be YES and NO and can be used as input in line 40.

```

10 REM DEFINE KEYS 1 AND 2 AS YES AND NO
20 KEY 1,"YES"+CHR$(13)
30 KEY 2,"NO"+CHR$(13)
40 INPUT "WANT TO SEE THE KEY DEFINITIONS";A$
50 IF A$ = "YES" THEN KEY
60 IF A$ = "NO" THEN PRINT "OKAY"

```

<b>LEFT\$</b>	<b>Abbr. leF</b>
<b>LEFT\$ (string,length)</b>	

Truncates the string in parentheses to the specified length. LEFT\$ is used frequently to check input, particularly to check just the first letter of the input.

**Parameters:** string being truncated, number of characters to use

1. The master string can be any text string, text-string variable, or string expression.
2. The LEFT\$ result always begins at the leftmost character in the master string. You can keep as many characters as you want. If the length specified is longer than the master string, the whole string is returned.

**Examples:** PRINT LEFT\$("GRADUAL",4)  
GRAD

PRINT LEFT\$("RED",4)  
RED

The string contains only three characters, so only three are printed.

10 INPUT "DO YOU WANT TO CONTINUE"; A\$  
20 IF LEFT\$(A\$,1)="Y" THEN GOSUB 70:  
ELSE END

Checks text string A\$, input in line 10, for the string Y.

**LEN**  
**LEN (string)**

**Abbr. none**

Counts the number of characters in a text string.

**Parameter:** master string

The master string can be any text string, text-string variable, or string expression. Blank spaces and punctuation count as characters.

**Examples:** PRINT LEN("HAYWIRE")

7

10 INPUT "WHAT'S YOUR LAST NAME"; L\$  
20 IF LEN(L\$) > 8 THEN L\$ = LEFT(L\$,8):  
PRINT "YOUR NAME HAS BEEN  
SHORTENED"

30 PRINT L\$

RUN

WHAT'S YOUR LAST NAME ? MACDONALDSON  
YOUR NAME HAS BEEN SHORTENED  
MACDONAL

Checks the length of L\$ and used only the eight leftmost characters if the length is over eight.

**LET****Abbr. IE****LET variable = expression**

Makes a variable equal to a value. The word LET may be (and usually is) omitted from the command. The LET command is unique in that its main keyword is optional.

**Parameters:** variable = value

1. The variable type must match the type of value being assigned (e.g., if the value is a text string, the variable must be a text-string variable).

If you want to assign more than one variable per line, separate the assignments with colons.

2. The value can be another variable ( $X = Y$ ), a calculation ( $X = X + 10$ ), or a constant value ( $X = 18$ ). A variable can be equal to a calculation, including a formula containing the variable itself and another value.

The value for a variable can change during the program.

Examples:	10 LET X = 4/2*Y	Assigns the value $4/2*Y$ to X.
	20 N\$ = "NAME"	Assigns the text string NAME to N\$.
	30 X% = X% + A	Gives X% the value of the answer to $X\% + A$ .
	40 A = 4:B = 5	Assigns the value 4 to A and the value 5 to B.

**LIST****Abbr. II****LIST line number-line number**

Displays a copy of a BASIC program or BASIC program lines.

**Parameters:** line number—line number

Line numbers are optional. If you omit them, the whole program is displayed. If you want to list just one line, type LIST and the line number. If you want to list just part of the program, type the first and last lines you want to display.

If you want to list the beginning of the program, type LIST followed by a dash and the last line number you want to display. If you want to list the end of the program, type LIST followed by the first line you want to see and then a dash; do not add any ending line.

Examples:	LIST	Displays all the lines in the current program.
	LIST 20	Displays line 20 from the current program.
	LIST - 100	Displays the beginning of the program up to line 100.
	LIST 50-	Displays the program from line 50 on.

**LOAD****Abbr. IO****LOAD file name, device, relocate**

Retrieves a program from a cassette tape or from a disk and loads it into memory. Use LOAD for tape programs and nonrelocated disk loads. Use DLOAD for loading BASIC programs from the disk. For more information see Chapter 6.

**LOADing a Tape Program**

After you issue a LOAD command for a tape program, the computer tells you to PRESS PLAY ON TAPE.

1. Insert the tape.
2. Press the REWIND button to rewind the tape completely when necessary. Press the STOP button when the tape is rewound.
3. Type LOAD "program name"; the program name is the name of the program you want to load. When you load the first program (after rewinding) or the next program on the tape, you do not have to include the program name; the computer automatically loads the next program on the tape.
4. Press the RETURN key. The message PRESS PLAY ON TAPE appears on the screen.
5. Press the PLAY button. The screen goes blank. When the program is found, the following message is displayed:

**FOUND** program name

6. Press the **C** key (or wait a moment). The screen goes blank. When the loading procedure is finished, the READY prompt is displayed.
7. Type RUN and press the RETURN key to execute the program.

**Note:** Be sure to press the right buttons on the cassette recorder. The computer knows when to wait for a button to be pressed but does not know which button was pressed. If you press the wrong button and the computer "freezes," eject the tape, press the computer's reset button, and repeat the loading procedure.

**Note:** For a nonrelocated LOAD, use LOAD "program name",1,1. Programs can be saved so that a nonrelocated LOAD is always performed. See SAVE.

See the VERIFY command for a quick method for searching a tape for a program.

**LOADing a Disk Program**

Although it is easier to load disk programs with the DLOAD command, you can also use LOAD. You must use LOAD to do a nonrelocated load from disk. When you use LOAD with disk programs, you must include the disk drive device number.

After you issue a LOAD command for a disk program, the computer displays the message OK SEARCHING. When the program is loaded, the message (program name) FOUND is displayed, with the program's name displayed. Type RUN to execute the program.

**Parameters:** "file name", device number, relocate flag

1. You must include the name of the file or use wild cards to get the first program whose name matches. Enter the name in quotes. You can use a variable name in place of the file name. The variable must have a value. It may be in parentheses (not in quotes). The only time this is likely to be useful is when you load a program from within another program.

2. Device number is 1 for cassette recorder, and 8 for disk drive. The default value is 1, so you can omit this parameter if you are loading from a cassette tape.

3. You are unlikely to use the relocate flag except for machine-language programs. A flag of 0 tells the computer to load the program at the beginning of the BASIC program area, and 1 loads the program at the memory location from which it was saved.

**Note:** For disks, only program-type files can be LOADED.

**Note:** In program mode, a RUN command (with no CLR) is automatically issued following a LOAD operation. For example, you may want to LOAD a machine language subroutine from BASIC.

**10 IF L=0 THEN L=1 : LOAD "file",8,1**

The LOAD is executed only once, and the program continues.

Examples:	<b>LOAD</b>	Loads the next program on tape.
	<b>LOAD "SHAPES3",8</b>	Loads file SHAPES3 from disk.
	<b>90 LOAD (Y\$)</b>	Loads a file from tape. The name of the file is the current value of Y\$.
<b>LOCATE</b>		<b>Abbr. loC</b>
<b>LOCATE coordinates</b>		

Repositions the pixel cursor on a graphic mode screen. The invisible pixel cursor marks the final point of the previous drawing and the default beginning point of the next drawing.

Parameter	Values
Coordinates	
Column coordinate	
High-res modes	0-319
Multicolor modes	0-159
Row coordinate	0-199

Give the coordinates of the point on the graphic screen where you want the pixel cursor to be moved. The next drawing will use this point as its starting point unless the drawing command gives some other starting point. For more information, see Chapter 4.

Example: 10 GRAPHIC 4,1  
20 LOCATE 30, 25  
30 DRAW TO 60,50

Puts the pixel cursor at column 30, row 25.

Draws a line from the current pixel-cursor location to column 60, row 50.

Finds the natural logarithm of a number. LOG returns the log base  $e$  ( $e =$  the mathematical constant, approximately 2.71828183) of the number in parentheses. Divide by LOG(10) to get the log base 10. For more information, see the Mathematical Calculations section of Chapter 3.

**Parameter:** any numeric expression with a positive value

Examples: PRINT LOG(2)  
              .693147181

Prints the natural logarithm of 2.

**PRINT LOG(2)/LOG(10)** Prints the logarithm base 10 of 2.  
.301029996

Works with DO to set conditions for a repeated sequence of program lines. See DO.

Gets a substring of the specified length within a master text string. MID\$ starts the substring at the character position specified. MID\$ can also be used to change part of a text string.

**Parameters:** master string, starting position, number of characters to use

1. The master string can be any text string, text-string variable, or string expression.
  2. The substring is begun at the starting position; characters that come before the starting position are not used. The starting position can be any character position in the master string. If it is greater than the length of the master string, a null string is returned.

3. The length of the substring can be any length. If it is greater than the number of characters after the start position in the master string, the entire rest of the string is returned. The length can be omitted. If it is omitted, all of the string after the start position is returned.

MID\$ can also be used on the left side of an equation to replace a substring of a given length within the master string.

**Examples:** PRINT MID\$("GRADUATE",6,3)  
ATE

```
10 INPUT "ENTER THE NEXT MODEL"; A$
20 IF MID$(A$,6,5)<>"WAGON" THEN END
30 MID$(A$,6,5)="SEDAN": PRINT A$
RUN
ENTER THE NEXT MODEL ? 4-DR WAGON
4-DR SEDAN
```

Examines five characters starting at character 6 for the string WAGON. WAGON is replaced by SEDAN.

```
PRINT MID$("ROCKETSHIP",7)
SHIP
```

## MONITOR

**Abbr. mO**

Leaves BASIC and goes to the built-in machine-language monitor. You can use the 13 machine-language monitor commands to write and execute programs in machine language. Return to BASIC from the monitor by typing X and pressing the RETURN key. See Chapter 5 for more information on machine language.

## NEW

**Abbr. none**

Erases the current program from memory. The program cannot be recalled unless it is saved on tape or disk. (If you execute a NEW accidentally and want to try to retrieve your program, see Chapter 3 for information on unNEWing.) Always issue a NEW command before you start writing a new program to be sure the program area of memory is clear. If you do not clear the memory, lines from the previous program will mix with your current program.

## NEXT

**Abbr. nE**

Marks the closing bracket of a FOR loop. See FOR.

**ON . . . GOSUB**

Abbrs. on/goS

**ON number GOSUB line number, line number, etc.**

Branches the program to one of a list of subroutines. The selection is based on the condition of the ON value and the position of the subroutine line numbers in the GOSUB list.

Each time ON . . . GOSUB executes, only one of the line numbers in the GOSUB list is used. When the ON value equals 1, the computer goes to the first subroutine in the GOSUB list. When the ON value equals 2, the computer goes to the second subroutine in the GOSUB list, and so on.

**Parameters:** ON value GOSUB subroutine line number list

1. The ON value can be a variable or a calculation. It cannot be a negative number. If it is equal to zero or a number that is greater than the number of subroutine line numbers in the GOSUB command, no subroutine is executed. If it is not a whole number, its truncated value is used. For example, if there are four subroutine line numbers in the GOSUB command (e.g., ON number GOSUB 40, 70, 100, 130), the number must be greater than or equal to 1 and less than 5 for a subroutine to be executed.

2. The ON value selects a subroutine line number from the GOSUB list based on its relative position in the GOSUB list.

**Example:**

```

10 INPUT "DO YOU WANT TO DRAW A TRIANGLE, SQUARE, OR PENTAGON"; SS
12 REM USE INPUT FROM LINE 10 TO SET NUMBER OF SIDES
15 SS = LEFT$(SS,1) : X = 3 : IF SS = "T" GOTO 20
16 X = 4 : IF SS = "S" GOTO 20
17 X = 5 : IF SS <> "P" GOTO 10
20 INPUT "DO YOU WANT THE SHAPE TO BE RED, BLUE, OR GREEN"; CS
30 REM USE INPUT FROM LINE 20 TO SET COLOR IN LINE 60
40 IF LEFT$(CS,1) = "R" THEN C = 3: GOTO 60
50 IF LEFT$(CS,1) = "B" THEN C = 7: ELSE C = 6
60 COLOR 1,C,3
80 GRAPHIC 2,1
85 REM 2 IS SUBTRACTED FROM NUMBER OF SIDES
86 REM WHEN X=3, 3-2=1, SO PROGRAM GOES TO FIRST SUBROUTINE, ETC.
90 ON X-2 GOSUB 140, 180, 210
100 REM AFTER SUBROUTINE, PROGRAM RESUMES AT LINE 110
110 INPUT "WANT TO DRAW ANOTHER SHAPE"; AS
120 IF LEFT$(AS,1) = "N" THEN GRAPHICCLR:END: ELSE 10
130 REM BEGIN SUBROUTINE TO DRAW TRIANGLE
140 CIRCLE, 160,100,60,50,,,120
150 PAINT, 160,100
160 RETURN
170 REM BEGIN SUBROUTINE TO DRAW SQUARE
180 BOX, 100,50,220,150,,1
190 RETURN
200 REM BEGIN SUBROUTINE TO DRAW PENTAGON
210 CIRCLE, 160,100,60,50,,,72
220 PAINT, 160,100
230 RETURN

```

**ON . . . GOTO**                          **Abbr. on/gO**  
**ON number GOTO line number, line number, etc.**

Branches the program to one of a list of line numbers. The selection is based on the condition of the ON value and the position of the line numbers in the GOTO list.

Each time ON . . . GOTO executes, only one of the line numbers in the GOTO list is used. When the ON value equals 1, the computer goes to the first line number in the GOTO list. When the ON value equals 2, the computer goes to the second line number in the GOTO list, and so on.

ON . . . GOTO is similar to IF . . . GOTO, but ON lets you include a series of GOTO lines while IF lets you include only one.

**Parameters:** ON value GOTO line number list

1. The ON value can be a variable or a calculation. It cannot be a negative number. If it is equal to zero or a number that is greater than the number of line numbers in the GOTO command, no GOTO is executed. If it is not a whole number, its truncated value is used. For example, if there are four line numbers in the GOTO command (e.g., ON number GOTO 40, 70, 100, 130), the number must be greater than or equal to 1 and less than 5 for a GOTO to be executed.
2. The ON value selects a line number from the GOTO list based on its relative position in the GOTO list.

**Example:**

```

10 TRAP 130
20 INPUT "WHAT YEAR (1985-1994)";Y
30 PRINT "NEW YEAR'S DAY FALLS ON ";
40 ON Y-1984 GOTO 70,80,90,100,120,60,70,80,100,120
50 PRINT"INVALID INPUT":GOTO20
60 PRINT"MONDAY":END
70 PRINT"TUESDAY":END
80 PRINT"WEDNESDAY":END
90 PRINT"THURSDAY":END
100 PRINT"FRIDAY":END
110 PRINT"SATURDAY":END
120 PRINT"SUNDAY":END
130 RESUME 50

```

**OPEN**                                  **Abbr. oP**  
**OPEN file number, device, secondary address, file name**

Opens access to a peripheral device or to a tape or disk file. Devices and files must be OPENed before you can issue other commands (such as INPUT# or PRINT#) to them. You do not have to use OPEN before you load or save a program.

**Parameters:** logical file number, device number, secondary address, “file name”

1. The logical file number can be from 1 to 255. Normally, use 1 to 127. For some devices, 0 is a valid logical file number. Logical file numbers greater than 127 cause a line feed character to be sent after the carriage return at the end of each line. Some non-Commodore printers or RS232 devices may require this.

The file number is not actually a part of the file or device you are opening. The file number is just a temporary number used until you CLOSE the file. It gives the computer a way to keep track of which device or file you are accessing. The file number is like a number you take at a deli counter or laundry—it is associated with you and your order only while your business is being transacted.

Once the device or file is OPEN, you must use the same file number for the device or file when you address other commands to it. These other commands are CLOSE, CMD, GET#, INPUT#, PRINT#, and PRINT# USING. Once the file is CLOSEd, the logical file number is no longer associated with the file and you do not have to use the same logical file number the next time you OPEN the file.

2. The device number identifies the other end (device or file) of the communication channel you are opening through the computer. If you are accessing a disk file, use the disk drive device number; if you are accessing a tape file, use the cassette recorder device number, and so on.

Use these device numbers:

- 0 keyboard
- 1 cassette recorder
- 2 RS232 port
- 3 screen
- 4–5 printer (default is 4)
- 8–11 disk drive (default is 8)

3. The meaning of the secondary address depends on the device you are accessing.

- For a cassette recorder, there are three: 0 (read from tape), 1 (write to tape and close with end-of-file marker), or 2 (write to tape and close with end-of-tape marker). The default is 0.
- For a printer, you can use secondary addresses to send commands. For more information, see Chapter 6 and the printer manual; these commands differ according to printer brand and type.
- For a disk drive, a secondary address names the channel being used. For more information, see Chapter 6 and the disk drive manual.

4. You can use an optional name for the file on disk or tape. The file name can be any 1 to 16 characters. If you intend to call the file by name, do it now because you will not be able to later.

5. For disk files, include in the quotation marks the optional type of file following a comma. The types are P (program file), S (sequential file), L (relative file), or U (user file). The default is sequential file. For more information, see Chapter 6.

6. For disk files, an optional disk file mode (R for read, or W for write) can follow the file type (still in quotation marks and separated from the type by a comma). The default is read.

**Examples:** **OPEN 4,4**

Opens communication to the printer so you can print directly onto it.

**OPEN 1,1,0,"BOXES"**

Opens a tape file for reading from tape.

**OPEN 8,8,15**

Opens the command channel to the disk drive.

**OPEN 1,8,4,"REC3,S,W"**

Opens a sequential file REC3 on a disk so you can write data records to the file.

**OPEN 4,4,0**

Opens printer in upper case/graphic mode.

**OPEN 4,4,7**

Opens printer in upper/lower case mode.

**PAINT**

**Abbr. pA**

**PAINT color source, coordinates, mode**

Used in any graphic drawing mode to make the outline of a shape solid. The shape is filled with color from the starting point until boundaries are met on all sides. See Chapter 4 for more information on coordinates for PAINT.

<i>Parameter</i>	<i>Values</i>	<i>Default</i>
Color source	0-3	1
Coordinates		pixel cursor
Column coordinate		
High-res modes	0-319	
Multicolor modes	0-159	
Row coordinate	0-199	
Boundary mode	0 or 1	0

1. The color source indirectly selects the painting color. There are five color sources, but color source 4 (the border color) cannot be used in drawing commands. Color sources 2 and 3 can be used only in multicolor modes.

0 screen background color

1 foreground color (default value)

- 2 multicolor mode extra color 1
- 3 multicolor mode extra color 2
- 4 screen border color

The color source number you include in the PAINT command tells the computer to draw in the current color for that source. For example, if you select 1, the computer paints in the current foreground color. If you want to use a color other than one of the current source values, you must first use the COLOR command to change one of the source values. Only sources 1 and 2 can be used to draw with more than one color on the same screen. Sources 0 and 3 are global (whole screen) colors.

If you want to use the default value (1, the current foreground color), you do not have to type a number, but you must type a comma before the next parameter.

2. The coordinates tell the computer where to start painting. You can tell the computer to start at any point within the space you want to paint. You never need to include ending coordinates. The computer stops painting when the border is reached.

3. Boundary mode lets you choose whether to have the painting stop when it reaches a border of the color source with which it is painting (select 0), or when it reaches any nonbackground color (select 1). The default is 0. The other choice is meaningful only in multicolor mode.

**Note:** If the specified starting point was previously colored with a boundary color, no painting will be done.

Example:	10 GRAPHIC 3,1 20 CIRCLE, 80,50,30,50,,,90 30 COLOR 3,12,4 40 PAINT 3, 80,50,1	Draws a diamond and fills it in with the color in color source 3.
----------	---	---

<b>PEEK</b> <b>PEEK (memory location)</b>	<b>Abbr. pE</b>
--	-----------------

Finds the contents of any RAM location. The location must be between 0 and 65535. PEEK and POKE are complements and can be used together to place and look at memory contents. Examining ROM using PEEK is not easy. See Chapter 4 (Copying the Standard Character Set) for one method.

Example:	POKE 3090,1 PRINT PEEK(3090) 1	Displays an A at screen location 3090. Prints the value of memory location 3090.
----------	--------------------------------------	---

**POKE**                          Abbr. pO  
**POKE memory location, number**

Places a single value directly into a specific RAM location, such as each position in the screen memory. Unless you are an advanced programmer, you are unlikely to use this command.

**Parameters:** memory location, value

1. Names the specific address of a memory location or input/output register. The possible values are 0 to 65535. You can find specific addresses on the memory map in Appendix G.
2. Gives the number (0–255) for the value you want to place in the memory location. See the screen display chart in Appendix E for values that can be poked to screen locations. These values are not the same as CHR\$ values.

**Example:**

```

10 SCNCLR
15 REM 3079 IS COL 7 OF ROW 1 ON SCREEN; 4071 IS BOTTOM
      RIGHT CORNER
20 FOR X=3079 TO 4071 STEP 41
30 POKE X,0
25 REM PRINT AN @ SIGN AT 41-SPACE INTERVALS
40 NEXT X
50 FOR Y=3103 TO 4071 STEP 39
55 REM PRINT AN @ SIGN AT 39-SPACE INTERVALS
60 POKE Y,0
70 NEXT Y

```

**POS**                          Abbr. none  
**POS (number)**

Finds the column in which the cursor currently resides. This is the column in which the next item will be displayed by a PRINT command. The column number is between 0 and 39. The number in parentheses is a dummy argument, which means it does not mean anything. It is nonetheless required, and you should just use POS(0). You can also find the column by using PEEK(202) and the row by using PEEK(205).

**PRINT**                          Abbr. ?
  
**PRINT output list**

Displays the following types of information on the screen:

- Text entered in quotes
- Solutions to calculations

- Values of variables or functions

Each PRINT command can contain one or more of these types of data. You can use commas or semicolons to separate multiple PRINT items and to determine the format for output.

**Parameters:** “text message”

variable  
calculation

All the parameters are optional. If no parameter is included, the PRINT command prints only a carriage return. In most cases this results in a blank line being PRINTed, but if the previous PRINT command ended in a semicolon, a following PRINT with no parameters just goes to the next line. See the later paragraph on using semicolons.

1. All text must be enclosed in quotes. Text messages are printed exactly as they are typed. Calculations and variables typed inside quotes are also printed as they are; PRINT does not attempt to find solutions for them when they are in quotes.

2. When you tell the computer to PRINT a variable, the computer actually prints the value the variable stands for, not the variable name. If no value has been assigned to a numeric variable, a 0 is printed; if no value has been assigned to a text-string variable, nothing is printed. Positive numeric values are printed with a leading blank space; negative numeric values display the minus sign in front of the number.

3. When you tell the computer to PRINT a calculation, the computer actually prints the solution to the calculation. If the calculation contains a variable for which no value has been assigned, the computer considers the variable to have a value of 0 and solves the calculation accordingly.

### Punctuation in PRINT Commands

You can separate multiple PRINT command items with commas or semicolons. If you want to link the output from more than one PRINT command, you can add a comma or a semicolon to the end of the first PRINT command. This trailing punctuation tells the computer to treat the output as if it were from one PRINT command instead of several.

Commas	Force separate items into separate output zones of 10 spaces each. Each new item begins in a new output zone, regardless of how much space the previous item takes.
Semicolons	Print separate items right next to each other. Numbers are printed with one space or a negative sign in front and one space behind. If you do not like this number format, see PRINT USING or STR\$ for other options.

**Examples:** PRINT "THE MEDIUM"  
THE MEDIUM

PRINT 3/2, 4+5, -2+2  
1.5        9        0

10 PRINT 3\*10.2; 4-8;  
15 REM SKIP TO THE NEXT LINE IF THE NEXT ITEM IS PAST COL 10  
20 IF POS(X)>10 THEN PRINT  
30 PRINT 5+67.3

RUN  
30.6 -4 72.3

10 INPUT "YOUR NAME AND AGE"; N\$,A  
20 PRINT N\$;" , YOU ARE"; A  
YOUR NAME AND AGE ? DAN  
?? 26  
DAN, YOU ARE 26

**PRINT#            Abbr. pR**  
**PRINT# file number, output list**

Puts data values in an OPEN file such as printing information on a printer. Use PRINT# to put values into tape or disk data files. To retrieve these data, use INPUT# or GET#. When you use PRINT# to tell the printer what to print, you can use semicolons and commas to separate multiple data items, and they have the same spacing effect as they have in PRINT commands.

When sent to a data file, a comma will cause a number of spaces to be sent and a semicolon will place data items right next to each other. The data bytes are stored in the file in the same format as they would be displayed on the screen in a PRINT command. The format of the bytes written to a data file must be designed with the method of retrieval in mind. If they are to be read one at a time by the GET# command, any format is alright. However, if they are to be read by the INPUT# command, care must be taken to store comma characters between values and carriage return characters between lines of input.

**Parameters:** logical file number, variable(s) or value(s)

1. The file number is a logical file number that links the file or device to other commands, including the OPEN command that accesses the device or file before it can be used. See OPEN for more information on logical file numbers.
2. The variable type must match the type of value to be written (e.g., if you are writing text values, you must use text-string variables). If the PRINT# command contains multiple variables, separate them with commas or semicolons.

**Example:**

```

10 OPEN 1,8,4,"SORT,S,W"           Opens a disk file.
30 INPUT "HOW MANY NAMES TO ENTER";T
40 FOR X=1 TO T
50 INPUT "NEXT NAME"; A$
60 PRINT#1,A$; ",";
70 NEXT X
80 CLOSE 1

NEW

10 INPUT "STAFF TO RECEIVE MEMO"; A$   Accesses printer.
20 OPEN 4,4                           Prints to printer.
30 PRINT#4,"MEMO"
40 PRINT#4,"TO ALL ";A$;" STAFF MEMBERS"
50 CLOSE 4

PRINT USING or PRINT# USING          Abbrs. ?usI/pR/usI
PRINT USING format; output list
PRINT# file number, USING format; output list

```

Allows you to design a format for any type of output—text or numbers. You can use up to nine symbols to define how you want printed material to appear. The PUDEF command lets you replace up to four of the PRINT USING symbols.

**Parameters:** #logical file number, “format description”; items to be printed

1. Include a file number and # sign if you are writing to an OPEN file or device. Omit this parameter if you are not writing to a file or device.
2. The format description can contain any of the following nine symbols. The format must be enclosed in quotes.

<i>Symbol</i>	<i>Meaning</i>
#	Represents any one character. If the item to be printed is longer than the number of #s in the format, and the item is numeric, an error occurs and *s are printed instead of numbers. If the item is text, only as many characters of text as there are #s are printed.
,	Prints a comma in numbers, which you cannot ordinarily do. Place the comma in the format in the same position in which it will appear in the number to be printed.
.	Prints a decimal point in numbers. Only one decimal point can appear per number to PRINT.
\$	Prints a dollar sign. If you want the \$ to appear right next to any

- number, place a # before the \$ in the format. Otherwise, the \$ does not “float” to a position next to the number (e.g., \$ 3.50 instead of \$3.50).
- +      Displays a plus sign at the beginning or end (but not both) of a number. If the number to be printed is negative, a minus sign is displayed in the place designed for the plus sign.
  - Displays a minus sign at the beginning or end (but not both) of a number. If the number to be printed is positive, no sign is displayed.
  - 
  - If you include no sign in the format, and the number is negative, a minus sign is displayed at the beginning of the number and before a dollar sign, if one is included. If the number is positive, no sign is displayed, but the space for the sign may be used to display an extra digit in the number.
  - ↑↑↑↑     Prints the number in scientific notation (e.g., 2E-04). The up arrows must be preceded by a number sign (#).
  - =      Centers text output in the format field (e.g., if the format field is "=#####", and the text to be printed is TEST, TEST is printed two characters to the right, centered in an eight-character field, with the = sign counting as a character space).
  - >      Right-justifies text output. If the text to be printed is shorter than the output format, the text is printed right-justified instead of left-justified.

**Note:** If numbers are longer than the format for numeric output, the number is not printed. Instead, \* symbols are printed. If a text string is longer than the format for text output, as many characters of the text are printed as there are spaces in the format (e.g., if there are six places in the format and the item to be printed is SCHOOLHOUSE, SCHOOL is all that is printed).

3. The items to be printed can be text or numeric, and they can be variables or formulas. List these items at the end of the PRINT USING command, separated from the format by a semicolon. Multiple items to be printed must be separated by commas.

### Using Trailing Semicolons to Control Output from Multiple Commands

If you want the output from the next PRINT or PRINT USING command to appear on the same line as the last output, put a semicolon at the end of the list of items to be printed. This trailing semicolon has the same effect on output as a trailing semicolon in a PRINT command has.

Note, however, that while a PRINT command can end in either a trailing semicolon or a trailing comma, a PRINT USING command can end only in a trailing semicolon. Also, while PRINT command items can be separated by either commas or semicolons, only commas are allowed as item separators in a PRINT USING command. The PRINT USING command allows more control over output because the format definition determines exactly how the output looks, so there is no need for semicolons as separators in a PRINT USING command.

```
Examples: 10 INPUT "ITEM"; I$  
20 INPUT "PRICE"; P  
25 REM PRINT UP TO 12 LETTERS  
30 PRINT USING "# #####"; I$;  
35 REM PRINT THE PRETAX PRICE  
40 PRINT USING "#$##,###.##"; P  
50 PRINT USING "#####"; "TOTAL";  
55 REM PRINT THE PRICE PLUS TAX  
60 PRINT USING "#$##,###.##"; P*1.06  
RUN  
ITEM? VIDEO EQUIPMENT  
PRICE? 1299.99  
VIDEO EQUIPM $1,299.99  
TOTAL $1,377.99
```

```
RUN  
ITEM? YACHT  
PRICE? 1000000.99  
YACHT *****  
TOTAL*****
```

NEW

```
10 INPUT "TITLE"; A$  
20 INPUT "NAME"; B$  
25 REM CENTER TITLE AND NAME  
30 PRINT USING "=#####";A$  
40 PRINT USING "=#####";B$  
RUN  
TITLE? TRAINING CATS  
NAME? JANE SMITH  
TRAINING CATS  
JANE SMITH
```

The # before the \$ forces the \$ to be printed right next to the first digit. Without the leading #, blanks appear between the \$ and any unused digits in the format.

VIDEO EQUIPMENT is longer than the 12 letters allowed by the format, so it is truncated.

On this run, the number of digits entered for the price is greater than the number accepted by the format, so the field is filled with \* symbols. Words that are too long are truncated; numbers that are too long are not printed at all.

**PUDEF**                    Abbr. pU  
**PUDEF** "one to four characters"

Lets you replace with any other symbol the symbols displayed by subsequent PRINT USING commands. You can replace blanks, commas, decimal points, and dollar signs from PRINT USING commands that follow the PUDEF command. You can issue as many PUDEF commands as necessary to print formatted data according to your special purposes.

Characters are replaced by their position in the PUDEF command. The default values for each position are used if you do not specify a different character in the appropriate position.

<i>Parameter</i>	<i>Values</i>	<i>Default</i>
First	Any character	Blank space
Second	Any character	Comma
Third	Any character	Decimal point
Fourth	Any character	Dollar sign

When you want to replace all the blank spaces in the output printed by a PRINT USING command, put the replacement character in the first position in the PUDEF command. When you want to replace commas, put the replacement character in the second position, and so on.

Because the computer recognizes replacement characters by position, you must type the default values in their positions if the following two conditions are true:

- You are not changing the default values.
- They appear in the PUDEF format before the characters you are changing.

If the default values appear after the changes, you can omit the defaults and just end the command. For example, if you want to leave blanks and commas when you change decimal points, you must place a blank and a comma in their positions in the PUDEF command and then type the decimal point replacement in the third position.

Examples: 90 PUDEF "/,"

Replaces blanks with slashes and decimal points with apostrophes. Commas and dollar signs are retained.

75 PUDEF ".bA"

Replaces commas with decimal points, decimal points with lowercase b's, and dollar signs with A's. Blanks are retained.

**RCLR****Abbr. rC****RCLR (color source)**

Finds the number of the color currently assigned to any of the five color sources:

- 0 screen background color
- 1 foreground color
- 2 multicolor mode extra color 1
- 3 multicolor mode extra color 2
- 4 screen border color

Type the number of the color source in parentheses. Only the color number is found. If you want to find the luminance, use the RLUM function.

**Examples:** PRINT RCLR(1)  
3 Prints the number for the current foreground color,  
which is 3, red.

PRINT RCLR(3)  
12 Prints the number for a current multicolor extra  
color, which is 12, pink.

**RDOT****Abbr. rD****RDOT (mode)**

Finds one of three pieces of information about the condition of the pixel cursor by typing one of the following values for mode:

- 0 returns the column coordinate
- 1 returns the row coordinate
- 2 returns the color source

The color source tells you with which of the four possible color sources the dot at the pixel cursor location is drawn:

- 0 screen background color
- 1 foreground color
- 2 multicolor mode extra color 1
- 3 multicolor mode extra color 2

Drawings cannot be made in the fifth color source, which determines the color

of the screen border. If you want to find the number of the color in the color source, use the RCLR function.

**READ****Abbr. rE****READ input list**

Always paired with DATA commands, one of several ways to assign data values within a program. READ contains a list of variables, and DATA contains a list of values. READ gets a value from a DATA command for each of its variables. You cannot input data from the keyboard for READ commands.

The program must contain enough DATA values for the READ variables. If there are not enough values, the program is aborted and an OUT OF DATA ERROR message is displayed. However, the total number of DATA values in all DATA commands is what counts, not the number of values per DATA command. When one DATA command runs out of values, READ automatically looks for the next DATA command in the program. DATA commands do not have to precede READ commands.

You can reREAD DATA values after you use the RESTORE command to reset the data pointer to the beginning of a DATA command. See RESTORE.

**Parameter:** variable(s)

The READ variables and the value types in the DATA commands must match (e.g., only text strings can be assigned to text-string variables). Variables must be separated by commas.

**Examples:**

```
10 DATA 55,44,33
20 READ A,B,C,D
40 PRINT A,B,C,D
RUN
?OUT OF DATA ERROR IN 20
```

The program must have at least as many DATA values as READ variables.

NEW

```
10 DATA MONDAY, MARCH, 18TH
20 READ A$,B$
30 READ C$,X
40 PRINT A$,B$,C$
50 DATA 1985
60 PRINT B$;X
RUN
```

```
MONDAY   MARCH   18TH
MARCH 1985
```

70 RESTORE

Resets the data pointer to beginning of line 10.

## 80 READ A

Reads first DATA item in line 10.

BUIN

Line 80 reads a text value for a numeric variable.

DEM

### Abbr. none

### REM remarks

Contains comments explaining program lines. REMarks make the program easier to understand when any user reads the program lines. If your program is longer than just a few lines, you should include REMarks so your program is well documented. Because REM statements are ignored by the computer, they can contain anything.

**Parameter:** remark

The remark does not need to be enclosed in quotation marks.

**Note:** In most BASIC lines, shifted characters are allowed only for abbreviations or in quotes. In a REM statement, shifted characters are fine until the program is LISTed. The computer treats a shifted character as a BASIC token and prints, not the character, but the BASIC keyword it corresponds to (see Appendix B for a token list). This is very annoying. If you want shifted characters in a REMark, put them inside quotes to avoid this problem.

Example:

```
10 REM PRINT A GREETING WITH THE USER'S NAME  
20 INPUT "WHAT'S YOUR NAME"; N$  
30 PRINT "HELLO, "; N$; ". WHAT'S NEW?"
```

## RENAME

### Abbr. reN

**RENAME Ddrive, old file name TO new file name, ON Unit**

Replaces the name of a disk file. The file itself is not affected.

**Parameters:** D drive number, "old name" TO "new name", U unit number

1. Give the number of the drive containing the disk whose file you want to rename. Drive numbers are either 0 or 1; no other numbers are allowed. The default value is 0. This parameter is optional. If you are using a single drive, leave out the drive number.
  2. Always list the file's current name first. Be sure to put it in quotes.
  3. TO is part of the RENAME command and must be included.
  4. Enclose in quotes the new name you want to give the file.
  5. U unit number is an optional parameter. Use it only if you have more than

one disk drive connected to your computer and you are using a device other than unit 8 in the RENAME procedure. You must precede the unit number with U, and the unit number must be between 8 and 11. You can type ON before U, but ON is not required.

**Note:** The drive and unit number parameters and the file names can be specified with a variable or expression in parentheses.

Examples: RENAME D1, "OLD" TO "NEW"      Changes the name of file OLD to NEW.

RENAME "TESTSORT" TO "SORT1"      Changes file TESTSORT to SORT1.

### RENUMBER

**Abbr. renU**

**RENUMBER new start line number, increment, old start line number**

Renumerates lines in the current BASIC program. This command can be executed in immediate mode only; you cannot include it in a program.

Parameter	Values	Default
New first line number	Any legal number*	10
Increment between lines	Any legal number*	10
First line number to be renumbered	Any line number in the program	First line in the program

\*Renumbering must not force line numbers to exceed the highest line number allowed, which is 63999, or strange results will occur.

1. Regardless of what numbering scheme is used in the current program, you can choose any new first line number. If you omit this parameter and thereby use the default, you must type a comma in place of the parameter in the command.

2. The increment tells the computer how many numbers to skip between line numbers. Even if the current line numbers are erratically spaced (e.g., 10, 15, 18, 20, 30, etc.), the RENumber command changes all line numbers so they are evenly spaced (e.g., 10, 20, 30, 40, 50, etc.). If you omit this parameter, type a comma in its place.

3. You can give any number in the program as the first line to be renumbered. This parameter lets you choose a line other than the first one in the program to begin renumbering.

Examples: RENumber 100

Makes 100 the new first line number. The defaults for increment and first line number to be renumbered are accepted.

```

1 REM POLYGON PROGRAM
10 INPUT "HOW MANY SHARES";T
15 COLOR 1,9,5
20 GRAPHIC 1,1
30 CIRCLE, 160,100,60,50,,,360/T
45 PAINT, 160, 100

```

**RENUMBER** 25, 20, 10

Begins at line 10 to renumber in increments of 20. The first new line number is 25.

**LIST**

```

1 REM POLYGON PROGRAM
25 INPUT "HOW MANY SHARES";T
45 COLOR 1,9,5
65 GRAPHIC 1,1
85 CIRCLE, 160,100,60,50,,,360/T
105 PAINT, 160, 100

```

**RESTORE**

**Abbr. reS**

**RESTORE line number**

Resets the data pointer that keeps track of the last item READ in a DATA command. Once the DATA command is reset, the values in the DATA command can be assigned again to READ variables. RESTORE does not affect READ commands or any other commands. You can reset to the beginning of the first DATA command in the program or to any DATA command in the program.

**Parameter:** line number

The line number parameter is optional. If you omit it, the pointer returns to the first DATA item in the first DATA command. If you specify a line number, the pointer is reset to the first DATA value in the first DATA command after that line. Data in previous DATA commands would not be reused.

**Example:**

```

10 DATA 1, 2, 3
20 DATA 7, 8, 9
30 READ A, B, C, D, E, F
40 PRINT A; B; C; D; E; F
50 RESTORE 20
60 READ A, B, C
70 PRINT A; B; C
RUN
1 2 3 7 8 9
7 8 9

```

The RESTORE command in line 50 resets the DATA pointer to the beginning of line 20. The READ command in line 60 rereads the DATA values in line 20.

<b>RESUME</b>	<b>Abbr. resU</b>
<b>RESUME line number</b>	
<b>RESUME NEXT</b>	

Works with the TRAP command, which catches program errors. Use RESUME to return to the current program after an error is found by a TRAP command. RESUME works only in conjunction with TRAP; RESUME cannot continue program execution unless the program is suspended by a TRAP command.

**Parameters:** line number or NEXT

1. If you issue a RESUME command with no parameter, execution resumes at the line where the error occurred. The computer will then try to execute this line again.
2. If you include a line number after the RESUME command, the program goes to that line and resumes execution there. Any line number that appears in the program can be used. If you use a line number, do not also type NEXT.
3. If you type NEXT, the program resumes execution at the line after the one that contained the error. The erroneous line is not reexecuted. If you use the NEXT parameter, do not also use a line number parameter.

**Examples:** **RESUME 250**      Restarts the program at line 250.

**RESUME NEXT**      Resumes execution at the line following the one that contains the TRAPped error.

<b>RETURN</b>	<b>Abbr. reT</b>
---------------	------------------

Ends a subroutine and returns program execution to the line after the last GOSUB command. RETURN is always paired with a GOSUB command. See GOSUB.

<b>RGR</b>	<b>Abbr. rG</b>
<b>RGR (number)</b>	

You can find the number of the current graphic mode:

- 0    text/graphic
- 1    high-resolution
- 2    split-screen high-resolution
- 3    multicolor
- 4    split-screen multicolor

The number in parentheses is a dummy argument, which means it does not mean anything. It is nonetheless required, and you should just type RGR(0).

Example: PRINT RGR(0)

2

The current graphic mode is split-screen high-res.

**RIGHT\$**

**Abbr. rI**

**RIGHT\$ (string, length)**

Returns a substring of the designated string with the specified length. RIGHT\$ is used frequently to check the end of an input string.

**Parameters:** master string, number of characters to use

1. The master string can be any text string, text-string variable, or string expression.
2. The RIGHT\$ substring always begins at the rightmost character in the master string. You can use as many characters as you want. If you specify more characters than the string contains, the entire string is returned.

Examples: PRINT RIGHT\$("GRADUATE",3)

ATE

PRINT RIGHT\$("RED",4)  
RED

The string contains only three characters so only three are printed.

```
10 INPUT "WHAT DAY IS IT"; D$  
20 INPUT "MORNING OR AFTERNOON"; T$  
30 IF RIGHT$(D$,4)="SDAY" THEN  
    PRINT "TUES/THURS SCHEDULE"  
RUN  
WHAT DAY IS IT? THURSDAY  
MORNING OR AFTERNOON? AFTERNOON  
TUES/THURS SCHEDULE
```

**RLUM**

**Abbr. rL**

**RLUM (color source)**

Find the number of the color luminance level assigned to one of the five color sources:

0 screen background color

1 foreground color

- 2      multicolor mode extra color 1
- 3      multicolor mode extra color 2
- 4      screen border color

To find the luminance level, type the number of the color source in parentheses. Only the luminance level is found.

The luminance level is stated in a range from 0 (darkest shade) to 7 (lightest shade).

If you want to find the color number, use the RCLR function.

<b>Examples:</b>	<b>PRINT RLUM(1)</b>	Prints the luminance level for the current foreground color. The level is found to be 3.
	<b>3</b>	
	<b>RND</b>	<b>Abbr. rN</b>

Finds a random number between 0 and 1. The numbers found are decimal values.

A negative number in parentheses reseeds the random number generator with that value. A zero reseeds the generator from the system clock. A positive number returns the next number in the current random number sequence. The generator should be reseeded only once in a program. The numbers in the sequence should be used thereafter.

Chapter 3 contains more information about mathematical functions, including RND.

<b>Example:</b>	<b>5 X = RND(0)</b>	Reseeds the generator.
	<b>10 X = INT (9 * RND(1)) + 1</b>	Generates a random whole number between 1 and 9.
	<b>20 INPUT "GUESS A NUMBER BETWEEN 1 &amp; 9";N</b>	You can choose to keep guessing until you get the right answer.
	<b>30 IF N &lt;&gt; X THEN PRINT "SORRY":</b>	
	<b>ELSE PRINT "RIGHT": GOTO 60</b>	
	<b>40 INPUT "GUESS AGAIN"; A\$</b>	
	<b>50 IF A\$ = "NO" THEN 60: ELSE</b>	
	<b>GOTO 20</b>	
	<b>60 INPUT "PLAY AGAIN";A\$</b>	To play again, goes to get a new random number.
	<b>70 IF A\$ = "NO" THEN END: ELSE 10</b>	
	<b>RUN</b>	
	<b>GUESS A NUMBER BETWEEN 1 &amp; 9? 8</b>	
	<b>SORRY</b>	
	<b>GUESS AGAIN? OK</b>	

GUESS A NUMBER BETWEEN 1 & 9? 6

RIGHT

PLAY AGAIN? NO

**RUN**

**Abbr. rU**

**RUN line number**

Executes the current BASIC program. Each time you issue a RUN command, all variables in the program are cleared (numeric variables to zero and string variables to nulls) because RUN contains an automatic CLR command.

**Parameter:** line number

Ordinarily you would use no parameter with the RUN command. But you can include a line number if you want program execution to start at a line other than the first one in the program. You might want to run just part of a program while you are still working it out.

**Examples:** **RUN**      Executes the current program.

**RUN 200**      Executes the current program from line 200. Preceding lines are not executed unless a branching command sends control back to a line before 200.

**SAVE**      **Abbr. sA**

**SAVE file name, device, end-of-tape flag**

Stores a BASIC program on tape or disk. Although SAVE can be used to save programs to cassette tape or disk, you should use this command to save to cassette tape and use DSAVE to save to disk.

**Parameters:** "file name", device number, end-of-tape marker

1. You should include the name of the file. Enter the name in quotes. You can use a variable name in place of the file name, but the variable must have a value. It may be in parentheses (not in quotes). The only time this is likely to be useful is when you store a program from within itself.

If you omit the file name in a SAVE to tape, the program is stored without a name, which is never a good idea.

2. Device number can be either 1 (for cassette tape recorder) or the disk drive number (8-11); no other numbers are allowed. The default value is 1, for cassette recorder, so you do not need this parameter if you are storing onto a cassette tape recorder.

3. If you are storing onto tape, you can add a final parameter to specify two

additional functions. If the final parameter is 1, the file cannot be relocated when subsequently LOADED. If the final parameter is 2, an end-of-tape marker (rather than an end-of-file) is written after the file on the tape. If the final parameter is 3, these two features are combined. If it is omitted, or is 0, neither feature is implemented.

**Note:** Files that cannot be relocated are LOADED into the memory locations from which they were SAVED, regardless of the status of the relocate flag in the LOAD command. The BASIC pointers at \$2D to \$32 (45-50) may be adversely altered by this event.

### Saving to Tape

When you issue the SAVE command, the computer displays the message

**PRESS PLAY AND RECORD ON TAPE**

Press the recorder buttons. Use the VERIFY command to make sure the program was stored accurately.

### Saving to Disk

When you issue the SAVE command, the disk light comes on. Do not remove the disk until the program is saved and the red light goes off. You can use the DIRECTORY command to view the disk directory and confirm that the file is saved. Use the VERIFY command to make sure the program was stored accurately.

Examples:	<b>SAVE "CIRCLE4"</b>	Stores file CIRCLE4 on tape.
	<b>SAVE "SHAPES9",8</b>	Stores file SHAPES9 on disk.
	<b>SAVE "GAME",1,2</b>	Stores file GAME on tape with an end-of-tape marker.
	<b>SCALE</b>	<b>Abbr. scA</b>
	<b>SCALE flag</b>	

Alters the scaling of the screen dots in graphic modes. Ordinarily the graphic modes have the following matrix of screen dots that you can control and use in drawings.

High-resolution modes	320 across and 200 down
Multicolor modes	160 across and 200 down

The SCALE command lets you change these values to 1024 logical dots both across and down in any mode.

In particular, this is useful when you are unsure whether a final program will run in high-resolution or in multicolor mode. If you do all the drawing with SCALE on, the figures will be the same size in either mode. No coordinate transformations are required to move between high resolution and multicolor SCALEd coordinates.

The SCALE command may be executed at any time and remains in effect until canceled.

After you turn on scaling, you must adapt the drawing commands to the new screen coordinates. For example, the center of the high-res screen is no longer 160,100. It is now 512,512.

To calculate SCALEd values from high resolution or multicolor coordinates, use these formulas:

High-res rows	5.12 * row coordinate
High-res columns	3.2 * column coordinate
Multicolor rows	5.12 * row coordinate
Multicolor columns	6.4 * column coordinate

For example, to get the same circle as drawn without SCALE by CIRCLE,160,100,60,50, you can use the following lines with SCALE:

```
5 SCALE 1
10 A = 3.2 * 160: B = 5.12 * 100
20 C = 3.2 * 60: D = 5.12 * 50
30 GRAPHIC 2,1
40 CIRCLE,A,B,C,D
```

**Parameter:** on or off.

Turn SCALEing on by using the parameter 1. Turn SCALEing off with the parameter 0.

**SCNCLR**                  **Abbr. scC**

Erases the screen and returns the cursor to the top of the screen or text area. Use SCNCLR to clear the screen in any mode, text or graphic.

**SCRATCH**                  **Abbr. scR**
**SCRATCH file name, Ddrive, Uunit**

Deletes a disk file permanently. References to the file are erased from the disk

and the file is flagged as SCRATCHed in the disk directory. The number of blocks occupied by the SCRATCHed file are freed for use.

Once overwritten, SCRATCHed files are lost permanently from the disk. When you issue a SCRATCH command in immediate mode, the computer gives you a chance to double check before the command is executed. The question ARE YOU SURE ? is displayed, and the computer does not proceed with the SCRATCH operation until you respond. Type Y to proceed. If you type anything else, the SCRATCH operation is aborted. In program mode, the question is not asked.

After you SCRATCH a file, you can verify that the file is deleted by displaying the disk directory. Note the difference in total blocks free now that the file is SCRATCHed.

**Note:** If a file is inadvertently SCRATCHed, it can sometimes be recovered using direct-access disk commands. See Chapter 6 for a sample program.

**Parameters:** "file name", D drive number, U unit number

1. Include the name of the file you want to SCRATCH. The name must be in quotes. It may contain wild cards, but take care to SCRATCH only files you no longer need. You can check which files would be deleted before you execute the SCRATCH by using the DIRECTORY command with the same file name parameter.

2. Drive numbers are either 0 or 1. No other numbers are allowed. You can omit this parameter if you are using a single disk drive, or if you are scratching a file in drive 0 of a dual drive.

3. Unit number is an optional parameter. Use it only if more than one disk drive is connected to your computer and you are using a device other than unit 8 in the SCRATCH procedure. You must precede the unit number with U, and the unit number must be between 8 and 11. You can also type ON before U, but ON is never required.

**Note:** The drive and unit number parameters and the file name can be specified with a variable or expression in parentheses.

**Examples:** SCRATCH "GAME", D1      Removes file GAME from the disk in drive 1 of a dual drive.

SCRATCH "PICKNUM"      Scratches file PICKNUM.

**SGN**                      **Abbr. sG**

**SGN (number)**

Finds the sign—positive, negative, or zero—of the number. PRINT SGN(X) displays one of the following three responses:

- 1 X is positive
- 0 X is zero
- 1 X is negative

These are the only possible outcomes; they reflect the number's sign, not its value.

**SIN** Abbr. sI  
**SIN (number)**

The numeric function that finds the sine of the angle in parentheses. The angle must be expressed in radians. For more information, see the Mathematical Calculations section of Chapter 3.

**Parameter:** any number or numeric expression

**Examples:** PRINT SIN( $\pi/2$ )  
1 Prints the sine of an angle of  $\pi$  over 2 radians  
(90 degrees).

PRINT SIN( $30 * \pi / 180$ )  
.5 Prints the sine of an angle of 30 degrees.

**SOUND** Abbr. sO  
**SOUND voice, frequency value, duration**

Plays a sound after the VOL command turns on the volume. There are two voices in the computer, so you can play two-voice harmonies. One of the voices can be set to make a range of nonmusical noise.

The SOUND command selects the voice, the note to be played, and the length of time the note will last.

**Parameters:** voice number, note value, sound duration

1. There are three possible values for voice number:

- 1 plays notes in voice 1
- 2 plays notes in voice 2
- 3 plays noise in voice 2

The notes played by voice settings 1 and 2 are the same set. If you play different notes in voices 1 and 2 together, the notes play simultaneously, producing harmony. If you play notes in the same voice, they play one at a time.

The voice setting 3 is actually part of voice 2. Voice 3 plays only noise, not musical notes.

2. The note value can be from 0 to 1023, although each value does not correspond to a true musical note. The following note chart shows the values for playing actual notes in four octaves. Middle C is 596. See Appendix F and Chapter 3 for more information.

<i>Low Octave</i>	<i>Middle Octave</i>	<i>High Octave</i>	<i>Highest Octave</i>
A      7	516	770	897
B      118	571	798	911
C      169	596	810	917
D      262	643	834	929
E      345	685	854	939
F      383	704	864	944
G      453	739	881	953

3. The duration tells the computer how long to play the sound. Duration can have a value from 0 to 65535. A value of 1 equals 1/60th of a second, so 60 equals one second. A value of 0 immediately cuts off the current sound for that voice.

Examples:

```

10 VOL 7
15 REM PLAY A TWO-VOICE HARMONY
20 SOUND 1, 516, 60
30 SOUND 2, 345, 60
40 SOUND 1, 643, 60
50 SOUND 2, 262, 60
60 SOUND 1, 739, 60
70 SOUND 2, 262, 60
NEW
10 VOL 7
15 REM PLAY A RANGE OF NOISE
20 FOR X = 600 TO 940 STEP 17
30 SOUND 3, X, 15
40 NEXT

```

<b>SPC</b>	<b>Abbr. sP</b>
<b>SPC (number)</b>	

Adds spaces to data output to a printer or to a file on disk or tape. SPC adds the specified number of spaces from the end of the previous PRINT item to the beginning of the next item. The number in parentheses can be from 0 to 255.

When you are printing on the printer and an SPC forces a space in the last character position on the line, a carriage return and line feed are automatically performed. When this occurs, no spaces are printed on the next line regardless of the number of spaces in the SPC function.

## SPC Compared with TAB

The difference between SPC and TAB is that TAB always counts spaces from the leftmost column while SPC counts spaces from the last PRINTed item. For example, in the following program, TAB forces WORD to be printed five spaces from the left side of the screen. SPC forces WORD to be printed five spaces from the end of the previous PRINTed item, which was 12345.

```
10 PRINT "12345" TAB(5) "WORD"  
20 PRINT "12345" SPC(5) "WORD"  
30 PRINT "1234512345"
```

RUN  
12345WORD  
12345 WORD  
1234512345

Finds the square root of the number in parentheses. The number cannot be a negative, though 0 is allowed.

**Parameter:** any nonnegative number or numeric expression

**SSHAPE** string variable, corner coordinates, corner coordinates

Saves small rectangular parts of graphic screens in any graphic mode. These graphic screen areas, which occupy the space of up to 255 characters, are saved as text-string values in memory. You use a text-string variable to identify the screen area, just as you use a variable to identify any type of value.

Use **SSHAPE** and **GSHAPE** when you want to repeat a pattern on a graphic mode screen or when you are using animation and want to move or erase the pattern. After you save a screen area with **SSHAPE**, you can use the **GSHAPE** command to display it anywhere on the graphic screen. When you retrieve the area with the **GSHAPE** command, you give the screen location where you want the area to be displayed.

Areas saved with **S\$SHAPE** are cleared when any CLR command occurs. See Chapter 4 for more information on **G\$SHAPE** and **S\$SHAPE**.

**Parameters:** string variable, corner coordinates, opposite corner coordinates

1. The string variable is the name assigned to the graphic screen area saved with the SSHAPE command. Retrieve the area with GSHPAGE by using the same string-variable name used in the SSHAPE command.
2. Give the coordinates for one corner of the shape you want to save.
3. Give the coordinates for the opposite (diagonal) corner.

**Example:**

```

10 GRAPHIC 1,1
20 FOR X = 0 TO 90STEP 10
30 CIRCLE, 160,100, 60,50,,X,120
40 NEXT
45 REM SAVE AN AREA FROM THE DRAWING
50 SSHAPE A$, 90,60, 200,72
55 REM DISPLAY THE SAVED AREA IN REVERSE AT THE TOP OF
     THE SCREEN
60 GSHPAGE A$, 0,5,1

```

**STatus**                  **Abbr. none**

Reserved variable name that contains a value representing the status of the most recent input/output operation. You can read the status of most peripheral devices. PRINT ST to display the status of the last operation using a peripheral; 0 usually means the operation was successful.

The following chart shows the status codes for I/O operations. See Chapter 6, which contains detailed information about input/output operations.

<i>Bit</i>	<i>Value</i>	<i>Tape I/O</i>	<i>Serial I/O</i>	<i>RS232 I/O</i>
0	1	—	Timeout write	Parity error
1	2	—	Timeout read	Framing error
2	4	Short block	—	Receiver buffer overrun
3	8	Long block	—	Receiver buffer empty
4	16	Read error	—	Clear to send missing
5	32	Checksum error	—	—
6	64	End of file	End or identify	Data set ready missing
7	128	End of tape	Device not present	Break detected

For the 1541 disk drive, the end or identify bit set usually means the end of file has been reached. Also, for a VERIFY, bit 4 set means a verify error was found.

**STEP                  Abbr. stE**

Tells the computer how much to add to the counter variable in a FOR . . . TO . . . NEXT loop. See FOR . . . NEXT.

**STOP                  Abbr. sT**

Suspends the execution of the current BASIC program and sends the computer to immediate mode so you can debug. Any OPEN files stay OPEN after a STOP, and variables retain the values they had when the program was interrupted.

When STOP executes, the computer displays the message BREAK IN LINE line number, just as if you pressed the STOP key. You can use the CONT command to restart the program at the line following the STOP command as long as you follow the CONT restrictions: do not make program changes. You can also use GOTO to restart the program. Type GOTO and the line number where you want to resume.

Use END when you do not want the BREAK message.

**STR\$                  Abbr. stR  
STR\$(number)**

Translates the number in parentheses to a text string. If the number is negative, a minus sign precedes it in the string. A leading blank space is generated if the number is positive. A trailing blank space is not generated in either case.

Example:

```
10 X = 67: Y = -22.4
20 X$ = STR$(X)
30 Y$ = STR$(Y)
40 PRINT X$, Y$
RUN
67-22.4
```

**SYS                  Abbr. sY  
SYS memory location**

Executes a machine-language program at the memory location named, which can be anywhere in RAM or ROM. For more information, see Chapter 5.

You can put a SYS command in a BASIC program to combine the BASIC program with a machine-language program. If you do this, the machine-language program is considered a subroutine, and you must end the machine-language program with an RTS (return from subroutine) instruction. This RTS instruction sends program control back to the BASIC program line that follows the SYS command.

**Parameter:** machine-language program address

The address can be a number between decimal 0 and 65535 (hexadecimal \$0000-\$FFFF), or a numeric expression standing for a number between 0 and 65535.

**Example:** SYS 1525      Turns on the built-in software. This is part of the definition for function key 1.

**TAB**                  **Abbr. tA**

**TAB (number)**

Used in a PRINT command to move the specified number of spaces to the right from the left side of the screen or page. A TAB function is similar to a tab key on a typewriter. The number in parentheses can be from 0 to 255. Values above 39 refer to subsequent lines.

TAB can appear anywhere in a PRINT command, including between two PRINT items. No punctuation is required to set off a TAB function. Regardless of where TAB appears in the PRINT command, TAB always counts spaces from the leftmost column of the current line.

### TAB Compared with SPC

The difference between TAB and SPC is that TAB always counts spaces from the leftmost column while SPC counts spaces from the last PRINTed item. For example, in the following program, TAB forces WORD to be printed five spaces from the left side of the screen. SPC forces WORD to be printed five spaces from the end of the previous PRINTed item, which was 12345.

**Example:**

```
10 PRINT "12345" TAB(5) "WORD"
20 PRINT "12345" SPC(5) "WORD"
30 PRINT "1234512345"
```

```
RUN
12345WORD
12345      WORD
1234512345
```

**TAN**                  **Abbr. none**
  
**TAN (number)**

Numeric function that finds the tangent of the angle in parentheses. The angle must be expressed in radians. If the error message DIVISION BY ZERO appears after a TAN function, TAN has overflowed.

For more information, see the Mathematical Calculations section of Chapter 3.

**Parameter:** any number or numeric expression

Examples: PRINT TAN( $\pi/3$ )  
1.73205081 Prints the tangent of an angle of  $\pi$  over 3 radians (60 degrees).

PRINT TAN(30\* $\pi/180$ )  
.577350269 Prints the tangent of an angle of 30 degrees.

**TI****Abbr. none**

Reserved system variable that represents the current value of the hardware interval timer, which is also called the jiffy clock. The interval clock starts at zero each time you turn on the computer, and it is updated every 1/60th of a second as long as the computer is on (up to 24 hours). TI is stated in 60ths of a second; to find out the time in seconds, divide TI by 60. For more information, see SETTIM in Chapter 5.

The interval clock is not on during many input and output operations.

You can use the value of TI (that is, -TI) to seed random number generation in the RND function.

**TI\$****Abbr. none**

Reserved system variable that represents the current value of the hardware interval timer, which is also called the jiffy clock. The interval clock starts at zero each time you turn on the computer, and it is updated every 1/60th of a second as long as the computer is on (up to 24 hours). TI\$ is stated in six digits: the first two represent the current hour, the next two represent the current minute, and the last two represent the current second.

The interval clock is not on during many input and output operations.

You can reassign the value of the clock by assigning a six-digit value to TI\$. When you reassign TI\$, you must use six digits in quotation marks. If you type any more or fewer digits, the command is rejected, and the message ILLEGAL QUANTITY ERROR is displayed.

The following little program shows that TI and TI\$ both hold the current value of the interval clock, each in its own way.

Example: 10 PRINT TI/60  
20 PRINT TI\$  
30 TI\$="000000"

**RUN**

106.833333  
000146 Shows the number of seconds the computer has been on. Shows the time the computer has been on in hours (00), minutes (01), and seconds (46).

Line 30 resets the clock. Note the changes in the values of TI and TI\$ when the program is run again just after resetting.

```
RUN
1.65
000001
```

**THEN**                  **Abbr. tH**

Tells the computer what to do when an IF condition is true. See IF . . . THEN . . . ELSE.

**TRAP**                  **Abbr. tR**  
**TRAP line number**

Detects errors during execution of a BASIC program so that execution is not aborted. When TRAP finds a program error, the error flag is set, and execution passes to the line named in the TRAP command. This line can contain a routine to help diagnose and solve the error.

After an error has been TRAPPED, you can examine or display the following information about the error:

<i>Information</i>	<i>Display Command</i>
Number of the line that contains the error	PRINT EL
Error message number	PRINT ER
Error message that identifies the error	PRINT ERR\$(ER)

These commands can be included in the program at the line you name in the TRAP command.

Return to normal program execution after a TRAP by issuing a RESUME command. Never return by using a GOTO.

**Parameter:** line number or 0

The line number tells the computer where in the program to go if an error is found. A value of 0 for a TRAP command turns off the error-trapping function in the program.

**TROFF**                  **Abbr. troF**

Turns off the built-in error-tracing functions that are turned on by the TRON command.

**TRON****Abbr. trO**

Turns on built-in error-tracing functions. The tracing function displays the line number on each line in the program as the program executes. This function helps you locate a line that is causing program error.

**UNTIL****Abbr. uN**

Sets a closing condition in a DO . . . LOOP sequence. When you include an UNTIL clause in a DO loop, the loop executes until a condition is met. See DO.

**USR****Abbr. uS****USR (number)**

This is a BASIC function (i.e., it is invoked by setting some variable equal to it). USR goes immediately to a user-callable machine language subroutine whose starting address is contained in memory locations 1281 and 1282. Before you can use the USR function to access a machine-language subroutine, you must POKE the subroutine address to memory locations 1281 and 1282. If you do not, the execution of a USR function aborts the program and displays an ILLEGAL QUANTITY ERROR message.

The number in parentheses is a variable or formula you are sending to be used in the machine-language subroutine. It is stored in floating-point accumulator 1. At the end of the subroutine, the value in floating-point accumulator 1 is returned to the BASIC program as the value of USR. Using a machine-language subroutine is much like using a BASIC user-defined function: you send a value to be used, and it sends back to the main program the resultant value. USR is a window between BASIC and machine language. See Chapter 5 for more information.

**VAL****Abbr. vA****VAL (string)**

Converts a text-string number to a numeric value when you have a number in a text-string variable instead of a numeric variable. For example, you might input a phone number as a text-string variable, and then want to use it as a number to dial your automodem.

If the text-string value contains characters that are not numbers or acceptable parts of numbers (e.g., minus signs, decimal points, or E, which connotes scientific notation), the rest of the string is not converted. If the first character is not an acceptable character, VAL returns a zero.

**Example:**

```
10 GETKEY A$  
20 IF VAL(A$)<>0 THEN GOSUB 80  
30 END  
80 PRINT "THE DIGIT IS ";A$  
90 RETURN
```

One key is gotten from the keyboard. If it is a digit from 1 to 9, the subroutine is executed.

**VERIFY****Abbr. vE****VERIFY file name, device, relocate**

Compares the current program to a program on tape or disk. The verification procedure assures you that the program you saved was stored accurately. Use VERIFY right after you save a program to confirm accurate storage. This is very important for saving to tape, but is not as important for disks since they are more reliable.

You can also use VERIFY as a shortcut in positioning a tape to after a given file. Just start at the beginning of the tape and VERIFY using the name of the file you want to follow. The computer finds the designated file, tells you the programs do not match, and stops just after it, ready for your next operation.

**Parameters:** "file name", device number, relocate flag

1. The file name is optional for tape files, required for disk files. If omitted, the computer verifies the next program on tape.
2. The device number is 1 for cassette tape recorder or the device number for the disk drive (8-11). The default is 1, so you can omit this parameter if you are using a cassette recorder. The device number for the disk drive is required.
3. The relocate flag can be 0 (verify the program at the beginning of the BASIC program area) or 1 (verify the program at the memory location where the program was saved). This parameter is usually used only with machine-language programs. Its value can be overridden if tape files are SAVED in a certain way. See SAVE.

**Examples:** **VERIFY**

Compares the current program to the next program on tape.

**VERIFY "BOXES",8**

Compares the current program to the program BOXES stored on disk.

**VOL****Abbr. vO****VOL level**

Sets the volume for sounds made by the SOUND command. A VOL command must be issued before any SOUND command can be audible. The volume you set affects all voices.

You do not have to set the volume for each SOUND command. The last volume setting is in effect until you issue another VOL command.

**Parameter:** volume level

The volume level setting can be from 0 to 8. The highest possible volume is 8. You can turn off the volume with a level setting of 0.

The volume set by the VOL command is relative only to other VOL commands. You can make one sound louder or quieter than another by changing the VOL setting in between. The overall volume is controlled by your monitor or TV.

**Example:** VOL 2 Sets a low volume

**WAIT** **Abbr. wA**  
**WAIT location, AND value, XOR value**

Pauses the execution of the BASIC program until the value of the specified bits in the given memory location equals a designated value.

**Parameters:** memory address, ANDed value to be checked, XORed value

1. The memory address is the location whose contents are to be checked.
2. The computer ANDs this value, which must be between 0 and 255, to the value in the memory location. The memory location is repeatedly checked until the operation yields a value that is not 0. When this happens, the program continues with the command following WAIT.
3. This parameter is optional. If it is present, this value, which must be between 0 and 255, is XORed with the memory location contents before the ANDing takes place. In other words, this parameter can be used to invert the comparison bits. The location checked by a WAIT command must be changed by some external event, such as a button on the tape recorder being pressed (see Chapter 6). Otherwise, this command results in an infinite loop that can be exited only with the reset button. Should this occur, remember to hold down the RUN/STOP key when pressing the reset button if you want to keep your program intact. Exit the monitor by typing X and RETURN.

**WHILE** **Abbr. wH**

Sets a condition for the continuation of a DO . . . LOOP. A WHILE clause in a DO loop makes the loop execute as long as the condition is met. See DO.

---

## 2

# The Built-In Software

---

This chapter explains all the commands for each built-in program. Separate sections cover the commands for the word processor, the spreadsheet, and the file manager. In addition, the instructions for formatting printed output are explained in a separate section of the chapter. Some commands can be used in more than one program. Next to each command is one or more abbreviations that indicate which programs accept the command. The abbreviations used are WP (word processor), SS (spreadsheet), and FM (file manager).

*Note:* Text that would be displayed in reverse mode on your screen is shown boxed in this book.

## Switching Between the Programs

When you first turn on the built-in software by pressing function key F1 and the RETURN key, the word processor comes up automatically. To switch to one of the other built-in programs, you must enter command mode and issue one of the following commands:

<i>Command</i>	<i>Destination</i>
tw	word processor
tf	file manager
tc	spreadsheet
gr	graph generator (accessed through the spreadsheet only)

## Command Mode

Many commands in each of the built-in programs are issued in command mode, which means you must enter command mode before you can issue the command. To enter command mode, press the **C** and **C** keys together. The status line

displays the > sign to indicate that you are in command mode.

Formatting instructions are entered in reverse mode, not command mode. Also, spreadsheet entries and text in the word processor are not made in command mode.

## Changing Screen Colors

When you first turn on the built-in software, the word processor work area displays pale yellow characters on a black screen. The current cursor position is white. If you want to use a different color combination, issue a COLOR command in the spreadsheet. You can change the color of the screen only from the spreadsheet.

Only the screen background color is selected with the COLOR command; other colors are selected automatically. To select a new color, use the numbers from Table 2-1.

1. Switch to the spreadsheet (press **C**, then **TC**, and press RETURN).
2. Enter command mode again (press **C** and **C**).
3. Issue the COLOR command (type COLOR and the number of the color you want to be the screen background. Type a semicolon at the end of the command).

When you switch to another program, the screen color change remains in effect.

TABLE 2-1. Screen Background Colors and Numbers

<i>Background</i>	<i>Characters</i>	<i>Number</i>	<i>Background</i>	<i>Characters</i>	<i>Number</i>
Black	Yellow/white	0	Orange	Yellow/white	8
Gray	Black/white	1	Brown	Yellow/white	9
Red	Yellow/white	2	Yellow-green	Black/white	10
Cyan	Black/white	3	Pink	Yellow/white	11
Purple	Black/white	4	Blue-green	Black/white	12
Green	Black/white	5	Light blue	Black/white	13
Blue	Cyan/white	6	Dark blue	Cyan/white	14
Yellow	Black/white	7	Light green	Black/white	15

## Formatting Disks

Work from any built-in program can be stored on a disk (graphs must be transferred to the word processor and stored as word processor files). Before you can use a disk for storage, however, the disk must be formatted. Formatting

prepares the disk for use by dividing it into sectors compatible with your disk drive and by establishing a disk directory.

There are two ways to format the disk:

1. You can format the disk from BASIC by using the HEADER command. Use HEADER if you have not yet turned on the built-in software and you know you will want to store your work on a disk. The HEADER command is explained in Chapter 1.
2. You can format the disk from within the built-in software by using the FORMAT command. The FORMAT command can be issued only from the spreadsheet program. The FORMAT command is explained in Section 3 of this chapter.

*Note:* Do not store file manager files on a disk that contains any other type of file, including word processor or spreadsheet files. Each file manager file should have its own disk.

## **Drawing Bar Graphs**

The built-in graph generator is actually a part of the spreadsheet. The graph generator has no commands of its own. You create the graphs by entering numbers in the spreadsheet, and you can transfer the graphs to the word processor to print them as part of a document.

To create bar graphs, follow these steps:

1. If you want to keep a copy of work in either work area, use the SF (Save File) command to store the work on a disk. Clear the word processor and spreadsheet work areas with the CM (Clear Memory) command.
2. Switch to the spreadsheet (**C** C, then **TC** and RETURN).
3. Type the numbers for the graph in the first row of the spreadsheet work area. Be sure to type numbers only in the top row.
4. Issue a MAP command to tell the computer you want to send the graph to the word processor.
5. Switch to the graph generator (**C** C, then **GR** and RETURN). The graph is automatically drawn using the numbers you entered in the spreadsheet work area.
6. Press RETURN to return to the spreadsheet.
7. Switch to the word processor (**C** C, then **TW** and RETURN). The graph is displayed at the top of the word processor work area. When the graph is in the word processor, you can make any needed changes to the graph.

## Drawing Point Graphs

The graph generator creates only bar graphs, but you can edit the graphs to make them point graphs. Point graphs show only the top value from each graph column whereas bar graphs show a solid bar for each column. You can create point graphs simply by erasing all the # signs in the bar graph except for the top one.

After you edit the graph to erase all but the top # sign, you may want to change the # signs to some other symbol, such as the \* sign or the % sign. To do this, use the RE (search and replace) command to substitute other symbols for the # sign.

## Section 1. Word Processor Commands

You can use the word processor to write any type of document. You can transfer data from any of the other programs and print this information as part of a word processor document. The word processor contains a variety of useful, time-saving features.

The word processor work area is 77 columns across and 99 lines long. If you are writing a document longer than 99 lines, you can link files together with the LINKFILE instruction and print the linked files as though they were one continuous file.

The following commands let you control the word processor work area and manipulate the text to suit your needs. Commands for formatting printed output are explained in Section 2.

### Key Commands

ESC and C	Enter command mode.
ESC and Q	Repeat the previous command.
ESC and @	Delete a RETURN key symbol.
CONTROL and 9	Turn on reverse mode.
CONTROL and 0	Turn off reverse mode.
CONTROL and =	Set a tab. Delete a tab when pressed in a tabbed column.
SHIFT and =	Move the cursor to the next tabbed column.

### Cursor Control Keys

F1	Moves to column 1 of the following line.
----	--

- F2            Moves to column 41 of the following line.
- HOME        Moves to line 1 in the current column.
- CLEAR       Moves to line 22 or the bottom line reached in the current session. The cursor remains in the current column.

## Command Mode Commands

### **CA       WP, FM, SS**

Displays a listing of all the files on the current disk. The file catalog includes the following information:

File names

File lengths (stated in blocks)

Blocks remaining on the disk

Spreadsheet files have a .c suffix appended to each file name. Word processor files have no suffix. File manager disks (each file should have its own) show only their name and blocks free. The total blocks free on the disk shows you how much space remains out of the blank disk total of 664.

The catalog information is displayed on a separate screen. The work area is not affected; you can display a disk catalog at any time. When you finish looking at the catalog and press RETURN, the intact work area is returned to the screen.

### **CB       WP**

Lets you create a text block of up to 16 lines. After a block is created, you can move it as a whole to any location in the work area. You can also erase the block.

To create a text block, follow these steps:

1. Move the cursor to the last line of the block.
2. Use the SP (Set Pointer) command to set the end of the block.
3. Move the cursor to the first line of the block.
4. Issue the CB (Create Block) command to create the block.

After the block is created, you can insert it elsewhere by moving the cursor to the new location and issuing an IB (Insert Block) command. You can erase a block with the DB (Delete Block) command.

If you want a text block to appear in more than one place in the work area, create a block, insert it where you want it to be repeated, and leave the original block intact at its original location. You can insert the block in as many locations as you like; you are never required to erase the original block.

### **CM WP, SS**

When issued in the word processor, clears the word processor work area, but does not affect the other work areas. After you issue a CM command, the word processor work is lost unless you save it first.

After you issue a CM command, the computer displays a question to double check your intentions. In the word processor, the question is

**CLEAR ALL Y/N?**

In the spreadsheet the question is

**ARE YOU SURE Y/N?**

The questions are the same: Are you sure you want to clear the current work area? Because all the information cleared from the work area will be lost unless you save it first, this question gives you a last chance to change your mind before the work area is erased.

### **CP WP**

Cancels all pointers set by the SP (Set Pointer) command. Whereas EP (Erase Pointer) deletes only one pointer at a time, CP (Clear Pointer) erases all pointers currently in the work area. The cursor can be anywhere in the work area when you issue a CP command.

### **CT WP**

Gets rid of all the tabs on the screen at once. After you issue a CT (Cancel Tab) command, all \* signs, which mark tabbed columns, are erased from the bottom line of the work area.

Tabs are set by the CONTROL and = keys. You can also use the CONTROL and = keys to cancel a single tab.

### **DB WP**

Lets you erase a block of text. After it is erased, the text beneath the block is moved up to fill the space left by the deleted block.

To set the boundaries of a block for deletion, follow these steps:

1. Move the cursor to the last line of the block.
2. Use the SP (Set Pointer) command to set the end of the block.
3. Move the cursor to the first line of the block.
4. Issue the DB (Delete Block) command to delete the block.

If you want to move the block elsewhere before you delete it, use the CB (Create Block) command to create the block, and the IB (Insert Block) command to insert the block at a new location. Then follow the preceding instructions to delete the block.

#### **DF      WP, SS**

Removes a file permanently from a disk. Use the DF (Delete File) command only when you no longer want to keep a copy of the file.

To delete a file you previously saved on disk, issue the DF command and type the name of the file when the prompt DELETE FILE: is displayed on the command line at the bottom of the work area. When you press RETURN, the disk takes a few seconds to remove the file from the disk. This command operates like the SCRATCH command in BASIC.

You can verify that the file has been deleted by viewing the disk catalog, which you access by issuing a CA command.

#### **DL      WP**

Lets you delete the line of text where the cursor is currently located. You can delete a line of text anywhere in the work area. After you delete a line, the lines beneath are moved up to fill in the space left by the deleted line.

To delete a line, move the cursor to the line you want to erase and issue the DL (Delete Line) command. You can delete more than one line by repeating the DL command, by issuing a **Q** command after a DL command, or by using the DB (Delete Block) command to erase a group of lines.

#### **EP      WP**

Cancels a pointer set by the SP (Set Pointer) command. Whereas CP (Clear Pointers) removes all pointers currently set in the work area, EP (Erase Pointer) erases only one.

To use EP, move the cursor to the line where the pointer you want to erase is located, then issue the EP command. The pointer on the current line is deleted, but other pointers in the work area are unaffected.

**FU WP, SS**

Cancels half-screen mode (see the HA command) and returns the screen to a full display of the word processor work area (when issued in the word processor). The FU (FULL screen) command has no effect unless the computer is in half-screen mode, which is turned on by the HA command. Issuing an FU command does not affect the contents of either the word processor or the spreadsheet.

To return to full-screen word processing, you must be in the word processor when you issue an FU command. If you issue the command in the spreadsheet, you will return to the full-screen spreadsheet. If this happens, just issue a TW command to switch to the word processor.

**HA WP, SS**

Divides the screen in half so that partial work areas from the spreadsheet and word processor programs can be displayed simultaneously. In half-screen mode, 12 lines from the word processor are displayed in the top half of the screen, and seven rows from the spreadsheet are shown in the bottom half of the screen.

Half-screen mode has several uses:

1. When you want to transfer data from the spreadsheet row by row with the MAP command. MAP lets you transfer data manually, and half-screen mode lets you see both work areas while you use MAP.
2. When you need to refer to spreadsheet data while you are writing a word processor document, but do not want to transfer the data.
3. When you need to get data from the word processor for use in the spreadsheet. You cannot transfer data from the word processor to the spreadsheet, but half-screen mode lets you view any part of the word processor work area while you are working in the spreadsheet.

When you first issue the HA command in the word processor, the screen is split in half, but the spreadsheet is not displayed in the work area. To bring up the spreadsheet, issue a TC command. The spreadsheet is then displayed on the bottom half of the screen, and the cursor is under spreadsheet control.

### Cursor Control

Although you can see partial work areas from both programs, you can use only one program at a time. The cursor can move around only half of the screen, and the keyboard is under the control of the current program. You cannot type any information into the other program. To switch between programs, you must still issue TC and TW commands. The status line at the bottom of the screen indicates which program currently controls the keyboard.

When you are in half-screen mode, you can still view any segment of the whole work areas of either program. Just use the cursor keys and other cursor-movement keys to display other parts of the work area on the half screen.

### **IB      WP**

Lets you insert a text block anywhere in the work area. To use the IB (Insert Block) command, you must first use the CB (Create Block) command to set the limits of the block. Follow these steps:

1. Move the cursor to the last line of the block.
2. Use the SP (Set Pointer) command to set the end of the block.
3. Move the cursor to the first line of the block.
4. Issue the CB command to create the block.
5. Move the cursor to the line where you want to insert the block.
6. Issue the IB command to insert the block.

While the block is being moved, the message WORKING is displayed on the status line at the bottom of the screen. When the block is inserted in the new location, the block also remains intact in its original location. Use the DB (Delete Block) command to erase the original block unless you want the block to appear in both places.

You can insert the block repeatedly when you want it to appear in several places. Just move the cursor to the next location and issue the IB command.

### **ID      WP, SS**

Initializes the current disk. Before you use the ID (Initialize Disk) command, make sure the drive is on and the disk is inserted.

You can initialize the disk at any time. Your current work is not affected, nor are any files on the disk. Whenever you change disks, you should issue the ID command after you insert the second disk so it is initialized.

### **IL      WP**

Lets you insert a blank line between lines of text anywhere in the work area. The new line is inserted above the line where the cursor is currently located. The lines beneath are not affected; they are moved down one line to make room for the new line.

To insert a line, move the cursor to the line where you want the blank line added and then issue the IL (Insert Line) command. You can insert more than one

line by repeating the IL command, by issuing a **C** Q command after an IL command, or by creating a block (CB command) of blank lines and using the IB (Insert Block) command to insert the block.

### LF WP, SS

Lets you bring a file stored on disk into the work area. When you issue the LF (Load File) command, the computer displays the message LOAD FILE: on the status line at the bottom of the screen. Type the name of the file you want to load.

When you load a file with the LF command, the work area is cleared before the file is displayed. The loaded file is displayed at the top of the work area regardless of the current location of the cursor. When you want to load the file at some other location in the work area, use the MF (Merge File) command.

Because the LF command always clears the work area before the file is loaded at the top of the work area, any document in the work area is lost unless you saved it before you issued the LF command. If you want to retain the document in the work area after a file is loaded, use the MF command to load the file.

### Error Messages

If the message FILE NOT FOUND >>> is displayed on the status line after you issue an LF command, the disk in the drive does not contain the file whose name you typed in response to LOAD FILE. Either the wrong disk is in the drive or you misspelled the file name when you typed it. Check the disk catalog to make sure the file exists on that disk; then reissue the command. If the file does not exist on the disk and you are sure you have the right disk, most likely the file was never saved. If you must switch disks, be sure to issue the ID command after doing so.

If the message NO FILE! is displayed on the status line, the disk drive is not connected, or is not turned on, or no disk is inserted in the drive.

### MF WP

Lets you bring a file stored on disk into the work area. The difference between LF (Load File) and MF (Merge File) is that LF clears the work area before bringing up the file, whereas MF does not affect the work area as long as you move the cursor past the work you want to keep.

When you issue the MF command, the computer displays the message LOAD FILE: on the status line at the bottom of the screen. Type the name of the file you want to load.

When you load a file with the MF command, the loaded file is displayed beginning at the current location of the cursor. Any work in the work area above the cursor location is not affected, which means you can combine your current document with a stored document. Any work below the cursor location is lost

and is replaced by the merged file. When you want to clear the work area and load the file at the top of the work area, use the LF command.

If you move the cursor to a line far enough down that there is not enough room in the work area to hold the current document and the file you want to load, only as many lines from the loaded file as can fit in the work area are loaded. The rest of the stored document is not loaded. No message is generated to inform you of the partial load.

### Error Messages

If the message FILE NOT FOUND >>> is displayed on the status line after you issue an MF command, the disk in the drive does not contain the file whose name you typed in response to LOAD FILE. Either the wrong disk is in the drive or you misspelled the file name when you typed it. Check the disk catalog to make sure the file exists on that disk; then reissue the command. If the file does not exist on the disk and you are sure you have the right disk, most likely the file was never saved. If you must switch disks, be sure to issue the ID command after doing so.

If the message NO FILE! is displayed on the status line, the disk drive is not connected, or is not turned on, or no disk is inserted in the drive.

### **\*P       WP**

Prints a copy of the document currently in the word processor work area. You can use \*P (Print) only to print single work area documents. Use PR to print linked files.

Before you issue a \*P command, make sure the printer is connected and turned on and that the paper is properly inserted.

*Note:* When you want to print data from the spreadsheet or the file manager, you must first transfer the data to the word processor.

*Note:* To abort a printout, press the RUN/STOP key until the PRESS RETURN prompt appears; then turn off your printer.

### **PR       WP**

Lets you print a copy of a linked series of word processor files. When documents contain the LINKFILE command, you can use the PR (PRint) command to print the linked files as an uninterrupted series. When you want to print only one file, use the \*P command, not the PR command. You cannot use the \*P command to print linked files.

The PR command is a complex command that performs several tasks; it saves, loads, and prints files. Linked files are loaded and printed automatically.

## How the PR Command Saves Files

When you issue a PR command, the document currently in the work area is saved automatically with the file name ..tw, which represents a Temporary Workspace. After the current document is saved, the message LOAD FILE: is displayed. You type the name of any file you want to print, and the file is loaded and printed automatically. You can load and print any word processor file.

After all the linked files have been printed, the ..tw file is reLOADED so you can continue editing.

## How the PR Command Prints Linked Files

When you use the PR command to print linked files, the computer considers the linked files as one file. Formatting commands in a file are also in effect in linked files unless you change the format.

When a LINKFILE command appears in a file being printed by the PR command, the computer automatically loads and prints the linked file as soon as the current file is printed. You do not have to issue any commands during the printout of linked files. You can link as many files as you like; the computer continues to print linked files until no LINKFILE command is found.

You can link files on different disks and print them together if you include a PAUSE command just above the LINKFILE command. PAUSE stops the printout until you press RETURN. When the file pauses, remove the disk, insert the second disk, and press RETURN to resume printing.

## RE WP

Searches the current document to find every instance of a group of characters you specify. The characters are replaced with other characters that you specify. For example, you can search a report for the phrase "due to the fact that" and replace it with the word "because" every time the phrase appears.

The search-and-replace operation is not entirely automatic. When you use the search-and-replace operation, the computer stops each time the searched string is found and gives you the option to abort the search. If you choose to continue, the computer gives you the option to replace the string with the replacement string you typed when you issued the command. This manual replacement protects you from inadvertently making replacement errors.

To use the search-and-replace feature, follow these steps:

1. Move the cursor to the top of the area you want to search. If you do not want to search the top of the document, just position the cursor past this part. If you fail to move the cursor above the area to be searched, the replacements cannot be made.

2. Issue the RE (REplace) command.
3. Type the characters you want to be replaced when the prompt SEARCH: is displayed on the status line. You can search and replace a single character or any string of characters up to 29 characters in length.
4. Press RETURN and type the characters you want to become the replacement string when the prompt BECOMES: is displayed on the status line. The replacement string can be any string of characters up to 28 characters in length. The replacement string does not have to be the same length as the string being replaced. You can replace a string with a blank.
5. Press RETURN. The computer will find the first instance of the searched string and highlight it in reverse. The message CONTINUE Y/N is displayed on the status line.
6. Press Y to continue the procedure. Pressing Y at this point does not replace the string. If you press Y, the message REPLACE Y/N? is displayed on the status line. If you press N, the search-and-replace operation is aborted.
7. Press Y to execute the replacement. The string is replaced and a pointer (<) is automatically set to mark the line. The computer moves to the next instance of the searched string and displays the message CONTINUE Y/N?. Repeat steps 6 and 7 until all instances of the string are found or until you abort the search-and-replace operation.

If you press N in response to the REPLACE prompt, the current instance of the searched string is not replaced. The computer moves to the next instance of the searched string and displays the message CONTINUE Y/N?. Repeat steps 6 and 7.

#### **SF      WP, SS**

Lets you store the current word processor document on a disk. To store a file, follow these steps:

1. Insert a formatted disk. Use the ID (Initialize Disk) command to initialize it.
2. Issue the SF (Save File) command.
3. Type a file name when the message SAVE FILE: is displayed on the status line. You must give each file a name. The file name must be between 2 and 16 characters long.
4. If a file by the given name already exists on the disk, REPLACE Y/N is displayed. Type Y in response to REPLACE Y/N unless you want to keep the old copy of the file. In this case, type N and save the current file with a different name.

If you do not have a formatted disk, insert a new disk, switch to the spreadsheet, and issue a FORMAT command before you issue the SF command.

You can verify that the file is stored by issuing a CA command, which displays a list of all files on the disk. Make sure the number of blocks assigned to the file is greater than zero. If the number of blocks is zero, the file was not properly saved. Press RETURN and issue the SF command again.

## SP WP

Defines the end of a block of text. Use a pointer to set the end of a text block for insertion or deletion or to prevent text beneath the pointer from moving when you use the INSERT or DELETE keys.

To set a pointer, move the cursor to the line where you want to locate the pointer and then issue the SP (Set Pointer) command. The pointer remains in effect until canceled. You can cancel a single pointer with the EP command, or cancel all the pointers in the work area with the CP command.

### Using a Pointer to Prevent Text from Shifting

When you move the cursor into the body of the document and then use the INSERT or DELETE keys, all following lines move too. Generally you would want only the rest of the paragraph to move when you press INSERT or DELETE. To prevent subsequent paragraphs from shifting, set a pointer at the end of the paragraph you want to work on. Then when you insert or delete within the paragraph, the rest of the document will not be affected.

## SR WP

Searches the current document to find every instance of a group of characters you specify. Unlike the RE command, the SR (SeaRch) command does not replace the characters that are being searched. Instead, the searched string is simply highlighted. Use SR to find a word quickly, to search for a string you want to replace with different strings each time it appears, or to look for misspellings.

When you use the search operation, the computer stops each time the searched string is found and gives you the option to abort the search.

To use the search feature, follow these steps:

1. Move the cursor to the top of the area you want to search. If you do not want to search the top of the document, just position the cursor past this part. If you fail to move the cursor above the area to be searched, the search cannot be made.
2. Issue the SR command.

3. Type the characters you want to search when the prompt SEARCH: is displayed on the status line. You can search for a single character or any string of characters up to 29 characters in length.
4. Press RETURN. The computer will find the first instance of the searched string and highlight it in reverse. The message CONTINUE Y/N is displayed on the status line.
5. Press Y to continue the procedure. You cannot change the string if you choose to continue. The computer will simply continue finding and highlighting instances of the searched string. If you want to make a change to the searched string, you can abort the search by pressing N. After you make the change, you can start the search again by reissuing the SR command.

**TC      WP, FM**

Lets you switch to the spreadsheet program. You can issue the TC (To the Calculator) command at any time. The word processor work area is not affected when you leave the program. Use the TW command to switch back to the word processor.

When you are using the half-screen mode, use TC to switch control to the spreadsheet half of the screen.

**TF      WP, SS**

Lets you switch to the file manager program. You can issue the TF (To the File) command at any time. The word processor work area is not affected when you leave the program. Use the TW command to switch back to the word processor.

## **Section 2. Instructions for Formatting Printed Documents**

The following instructions let you design the format of documents you print on a printer. These instructions do not affect the work as it is displayed on the screen. Although the instructions themselves are embedded in the document, they do not appear in the printed version of the work.

Formatting instructions are entered differently from commands typed in command mode. The formatting instructions are always typed in reverse mode, and they are always typed within the text, not on the status line. To enter a formatting instruction, follow these steps:

1. Turn on reverse mode by pressing CONTROL and the 9 key.

2. Type the formatting instruction in lowercase letters. Always type a semicolon (;) at the end of each formatting instruction. If you are typing more than one formatting instruction together, separate the instructions with a colon (:). If you include the colon, you must also include the semicolon. For example, if you use both an LMARG and an RMARG instruction on the same line, type them like this:

```
lmarg10;:rmarg65;
```

*Note:* Text that would be displayed in reverse mode on your screen is shown boxed in this book.

3. Turn off reverse mode by pressing CONTROL and the 0 key.

The formatting instructions are explained in alphabetical order. This section includes the six formatting instructions for printing data from file manager records.

## Format Defaults

Some formatting instructions have default values, but most do not. The following are default values for printed formats:

Paper size	66 lines (11 inches)
Page length	60 lines per page
Left margin	0
Right margin	77
Justification	Left only
Word wrap	On

## ASC

ASCII instruction lets you send a CHR\$ control code directly to the printer. You can tell a printer to print a word in boldface type, underline a heading, or print a special character. The CHR\$ control values depend on your printer, not on the computer. Your printer manual tells you the values for the CHR\$ control codes. See Chapter 6 for more information.

## CENTER

Tells the printer to print the current line centered on the page. Type the CENTER instruction at the beginning of the line you want to center, then type the text right next to the CENTER instruction. Do not try to center the text on the screen; the centering is done when the document is printed.

Example: **center;A HISTORY OF COAL MINING**      Prints the title centered on the page.

## EOF?

Lets you get data from more than one file manager record in a continuous print operation. EOF?(End Of File?) creates a conditional loop that goes through a file to get field data from a series of records. Then the document is reprinted for each record.

Use the EOF? instruction when you want to print multiple copies of the document with data from a sequential group of records. For example, you would use EOF? to print form letters, address lists, labels, and so on.

The EOF? instruction, which must be embedded at the end of the document, does two things:

1. Checks to see if the last record in the file has been read.
2. Forces the document to be printed again, using data from the next record, when more records remain in the file. When the last record has been read, EOF? ends the printing operation.

The EOF? instruction is the only indication in a document that more than one copy of the document will be printed. The RC instruction can be used to start at a record other than the first record, and the EOF? instruction forces the computer to the next record after the document is printed.

## FLD

Used to print the contents of a file manager field anywhere in a word processor document. You must include the field number in the FLD (FiLD) instruction. The printer gets the contents of the specified field from the current record in the current file manager file. Only the contents of the field are printed. If you want to print the field name, use the TTL instruction.

If the specified field for the current record is empty, a colon is printed in place of the field contents.

Examples:	<code>fld3;:fld5;:fld1;</code>	Prints the values in the current record for fields 3, 5, and 1.
The author is	<code>fld2;:fld1;</code>	Prints the values in the current record for fields 2 and 1. The fields are printed as part of a sentence.

## JUSTIFY

Forces text to be printed right justified, which means that the right side of the lines are printed flush against the right margin, just as the left sides of the lines are printed flush against the left margin. Text is ordinarily printed left justified with ragged right margins.

You can cancel a JUSTIFY instruction with the NOJUSTIFY instruction.

## LINKFILE

Lets you link multiple word processor files so they can be printed as one continuous document. You can link as many word processor files as you need, including files on other disks. (If you link files from other disks, include a PAUSE instruction just before the LINKFILE instruction to provide time to switch disks before the printing resumes.)

Linked files are treated as a continuous document when they are printed. This means the following:

1. Margin settings and other formatting instructions affect later files unless they are changed.
2. Page numbering continues throughout the series of linked files.
3. Text from linked files is printed without breaks on the page, so that you cannot tell from the printed pages where one file ends and the next begins.

To link a file, include the LINKFILE instruction at the end of the document. Beside LINKFILE, type the name of the next file, which must be enclosed in single quotes.

You must use the PR command (not \*P) to print linked files. When linked files are printed, the next file is automatically loaded and printed with no additional commands or other input from you.

Example:	<code>linkfile'section2'</code>	Links the file SECTION2 to the current file.
----------	---------------------------------	--

**LMARG**

Lets you set the left margin of the page when the document is printed. You can use any value for the left margin as long as it is a lower number than the right margin. The default left margin is 0.

Example:

```
lmarg 15;
```

**NOJUSTIFY**

Cancels a JUSTIFY instruction. NOJUSTIFY returns the document format to the default setting of left-justified output. Left justified means that the left side of the lines are printed flush against the left margin. The right side of the lines are printed with a ragged margin, which means that the ends of the lines are not flush against the margin.

**NEXTPAGE**

Forces the printer to go immediately to the next page and resume printing there regardless of where the printer is currently printing on the page. A NEXTPAGE instruction overrides the PAGELENgth setting.

You can put a NEXTPAGE instruction anywhere in the document to force a new page. For example, just before an EOF? instruction, you could include a NEXTPAGE instruction to force a new page each time a document is reprinted.

**NOWRAP**

Lets you print wider-than-usual text, such as spreadsheet data. Ordinarily the computer considers column 77 of the word processor to run continuously onto column 1 of the next line as if there were no end to a line. If a word is split up between column 77 and column 1 of the next line, the computer and the printer assume that the word is a single word. This continuity between column 77 and column 1 is called word wrap.

In contrast with the effects of word wrap in the word processor work area, the spreadsheet considers each row to be completely separate. When you transfer spreadsheet data to the word processor and print the combined material, you may have to compensate for the assumption of word wrap in the word processor and the assumption of no word wrap in the spreadsheet. You can make sure the spreadsheet lines are not run together by placing RETURNS at the end of each spreadsheet line or by issuing a NOWRAP instruction just above the spreadsheet data in the document.

You can cancel NOWRAP with the WRAPON instruction.

**NO#PAGE**

Cancels a #PAGE instruction. When you issue a NO#PAGE instruction, page numbers are no longer printed at the bottom of each page.

**OTHER**

Lets you tell the computer that the printer you are using is not made by Commodore. This printer-brand command is used to switch the special character set Commodore uses to the standard ASCII character set. If you use a non-Commodore printer and do not include an OTHER instruction in every printed document, some characters will not print correctly.

If you use a printer made by Commodore, there is no need ever to use the OTHER instruction.

**#PAGE**

Lets you print the page number at the bottom of each page. You can precede the #PAGE instruction with a SET#PG instruction to start the first page number to any number. The #PAGE instruction automatically increments the page numbers as each page is printed.

Use the NO#PAGE instruction to turn off the #PAGE instruction.

**PAGELEN**

Lets you limit the number of lines that will be printed on each page. The value of PAGELEN (PAGELENgth) can be any positive number that is less than the value of PAPERSIZE. The default value of PAGELEN is 60 lines.

**PAGEPAUSE**

Stops the printout at the end of every page. If you are printing on single sheets of paper (not connected fanfold paper), use PAGEPAUSE to give yourself time to insert a new piece of paper after each page prints. The printing does not resume until you press the RETURN key.

You might also, for example, use PAGEPAUSE to be sure the paper is properly aligned as each page prints.

**PAPERSIZE**

Tells the printer you are using nonstandard length paper. The printer assumes the paper is 11 inches long, which equals 66 lines on most printers, the default

setting for PAPERSIZE. You do not need to include this instruction when you use 11-inch long paper.

The PAPERSIZE is stated in lines, not in inches. Be sure to use the proper PAPERSIZE; if you do not, page tops will be positioned incorrectly. When you change the PAPERSIZE, you will most likely also change the PAGELEN. The value of PAGELEN must be less than the value of PAPERSIZE.

Example: **pagesize84;:pagelen75;**

Tells the printer you are printing on 14-inch legal-sized paper, and you want each page to contain no more than 75 lines.

## PAUSE

Can appear anywhere in a word processor document and stops the printout immediately. If you are printing linked files that are stored on different disks, use PAUSE to give yourself time to insert the other disk. The printing does not resume until you press the RETURN key.

You might also, for example, use PAUSE to make sure the paper is properly aligned during the printout.

## RC

Can be used to tell the computer to start at a specific record number when accessing file manager data. Use RC (ReCord) when you are using file manager data in multiple printed copies of a document and you want to start at a record other than record 1. When the RC instruction is used, the first execution of the EOF? instruction goes to the record after the one named in the RC instruction.

Type the RC instruction just before the tf;:rc; instruction. Include the number of the record you want to be accessed first. If you omit the RC instruction when you are printing multiple file manager records, the first record accessed will automatically be record 1.

Example: **rc10;** Starts accessing file manager data at record 10. Records 1 through 9 are skipped.

## #RC

Can be used to print the record number of the current record when you are printing file manager data. #RC (ReCord number) prints only the record number, not any information about the record contents.

## RMARG

Lets you set the right margin of the page when the document is printed. You can use any value for the right margin, as long as it is a higher number than the left margin. The default right margin is 77.

Examples:

**rmarg75;**

Sets a right margin of 75.

**lmarg20;:rmarg65;**

Sets a left margin of 20 and a right margin of 65.

## SET#PG

Lets you begin page numbering at a number other than 1. You can precede the #PAGE instruction with a SET#PG instruction to start the first page number at any number. The #PAGE instruction automatically increments the page numbers as each page is printed.

A SET#PG instruction can appear anywhere in the word processor document. You can use SET#PG to change the sequence of page numbers or to print a page with any page number. SET#PG is also useful when you are reprinting selected pages from a long document.

## TF;;RC

Use together at the beginning of a word processor document that will access file manager data. The TF;;RC (To the File manager and Record) instruction is required to tell the computer to get data from the file manager. You can type a separate RC instruction with a record number above the TF;;RC instruction when you want to begin accessing records at a record other than record 1.

Example:

**rc10;  
tf;;rc;**

Tells the computer to get file manager data starting at record 10.

## TTL

Use when accessing file manager records. TTL prints the field name for the field specified in the instruction. TTL prints only the field name, not any data stored in the field. Use TTL and FLD together when you want to print both the name of the field and the contents of that field in a particular record.

Example:

**ttl5;:fld5;**

Prints the field name of field 5 and the field contents from the current record.

## WRAPON

Cancels the NOWRAP instruction and returns to normal word wrap conditions. Ordinarily the computer considers column 77 of the word processor to run continuously onto column 1 of the next line as if there were no end to a line. If a word is split up between column 77 and column 1 of the next line, the computer and the printer assume that the word is a single word. This continuity between column 77 and column 1 is called *word wrap*.

Turn word wrap off with the NOWRAP instruction when you are printing spreadsheet data and you do not want the rows of data to be considered a continuous line. You can also make sure spreadsheet rows are not run together by placing RETURNS at the end of each spreadsheet line.

## Section 3. Spreadsheet Commands

The spreadsheet lets you keep track of any type of tabular information. You can use numbers, words, or formulas as spreadsheet entries. The spreadsheet work area is organized into numbered rows and columns. Each row-and-column position in the work area is called a cell. A cell is identified by its row;column numbers.

### Key Commands

Cursor-down arrow	Moves the cell cursor down.
Cursor-up arrow	Moves the cell cursor up.
F2 or <b>C</b> and R	Moves the cell cursor to the right.
F1 or <b>C</b> and L	Moves the cell cursor to the left.
<b>C</b> and T keys	Prepare for text entry into a cell.
<b>C</b> and N keys	Prepare for numeric entry into a cell. Necessary only when the cell was previously reserved for text or formula entries.
<b>C</b> and F keys	Prepare for formula entry into a cell. Also redisplays a formula used in a cell.
<b>C</b> and Q keys	Repeat the previous command.

### Mathematical Operators Used in Spreadsheet Formulas

#	Precedes a constant number in a formula.	#100 + 2;1
+	Addition.	6;6 + 6;7

.	Subtraction.	8;2 - 7;2
*	Multiplication.	2;6 * 3;1
/	Division.	9;2 / 4;4
↑	Exponentiation.	4;4 ↑ 2;2
ABS	Absolute (positive) value.	ABS 3;2
ATN	Arctangent in radians.	ATN 3;2
COS	Cosine of an angle in radians.	COS 5;8
DIV	Divides a row or column series of cells.	DIV 4;4 TO 9;4
EXP	Finds an exponential of the constant <i>e</i> (approximately 2.71828183).	EXP 4;12
IFTRUE	Makes an entry only if a clause is true.	3;1=#100iftrue3;2←#200
LOG	Logarithm base <i>e</i> .	LOG 12;4
MAX	Displays the highest number in a row or column series of numbers.	MAX 7;1 TO 7;9
MIN	Displays the lowest number in a row or column series of numbers.	MIN 7;1 TO 7;9
MLT	Multiplies a row or column series of cell entries.	MLT 5;2 TO 5;9
NOTIFTRUE	Makes an entry only if a conditional clause is false.	3;1>2;1notiftrue3;1←2;1
SIN	Sine of an angle in radians.	SIN 4;4
SUB	Subtracts a row or column series of cell entries.	SUB 5;2 TO 9;2
SUM	Adds a row or column series of cell entries.	SUM 5;2 TO 9;2
TAN	Tangent of angle in radians.	TAN 12;3
←	Transfers the contents of one cell or a number into another cell in an IFTRUE or NOTIFTRUE formula.	3;1 ← 2;1

## Command Mode Commands

### **AUTO    SS**

Turns on AUTomatic calculation mode. When auto mode is on, formulas entered with the FIT command are automatically calculated. In addition, when you are in auto mode and change a cell entry that was used in a formula, any other cell entries that are affected by the change are automatically recalculated. Auto mode changes, however, do not affect cells frozen with the FRE command.

When auto mode is off, calculations are performed in manual mode. In manual mode, which is turned on with the MAN command, FITted formulas are not solved, and formulas affected by cell changes are not recalculated. You must enter auto mode to solve these calculations.

The default calculation mode is manual mode. The current calculation mode is indicated by MANU. or AUTO., which is displayed at the far right on the status line.

### **BLKMAP    SS**

Lets you transfer spreadsheet data to the word processor in blocks of up to 7 columns by 50 rows. The width of the block is limited to 7 columns because 7 columns of 11-column cells fill the 77-column word processor work area.

The word processor work area does not need to be empty when you transfer spreadsheet data, but be sure there is room in the 99-line word processor work area to hold all the spreadsheet data you are transferring. If there is not enough room and you are moving spreadsheet data to the bottom of the word processor work area, only as much data as can fit will be transferred.

When you use the BLKMAP (BLockMAP) command, spreadsheet data are transferred without the row and column numbers. The data are placed in the word processor work area starting at the location of the word processor cursor. If the word processor cursor is placed so that work in the word processor is already occupying some of the lines where the spreadsheet data will go, those lines of the word processor work are cleared and the spreadsheet data are written in its place. Make sure the word processor cursor is placed so that overwriting will not occur.

To use BLKMAP, follow these steps:

1. Switch to the word processor (**C C**, then **TW** and **RETURN**).
2. Move the cursor in the word processor to the position where you want the spreadsheet data to be moved. The spreadsheet data will be displayed over any word processor text that appears in the area where the data are to be moved, so be sure there are enough blank lines at the transfer location.
3. Switch back to the spreadsheet (**C C**, then **TC** and **RETURN**).

4. Put the cell cursor in the cell that is to be the top left corner of the block you are moving to the word processor.
5. Enter command mode and type BLKMAP and the cell number for the bottom right corner of the block. For example, type BLKMAP 7;3 to move data from the current cell cursor position to cell 7;3.  
The message WORKING is displayed briefly while the transfer is taking place.
6. Switch to the word processor (**C** C, then **TW** and **RETURN**).

After the spreadsheet data have been moved, they become part of the word processor document and can be manipulated by all the word processor commands. But the data are now text and can no longer be updated by the spreadsheet commands.

You can also use the MAP command to transfer spreadsheet data to the word processor. MAP transfers data row by row, not in blocks. Use the MAP command when you want to transfer longer cell entries that do not appear in the cells but are held in memory. BLKMAP transfers only the 11 characters per cell that appear in the spreadsheet work area.

**Example:** **BLKMAP23;7** Transfers the block of data from the current cell cursor location to cell 23;7.

### **CA WP, FM, SS**

Displays a listing of all the files on the current disk. The file CAatalog includes the following information:

- File names.
- File lengths (stated in blocks).
- Blocks remaining on the disk.

Spreadsheet files have a .c suffix appended to each file name. Word processor files have no suffix. File manager disks (each file should have its own) show only their name and blocks free.

The total blocks free on the disk shows you how much space remains out of the blank disk total of 664.

The catalog information is displayed on a separate screen. The work area is not affected; you can display a disk catalog at any time. When you finish looking at the catalog and press RETURN, the intact work area is returned to the screen.

**Example:** **ca** Lists a catalog of files on the current disk.

**CCO      SS**

Lets you copy the cell entries from one column into another. CCO (Column COpy) writes a duplicate of all cell entries from the column you are copying into the column where the cell cursor is currently located. The cells are copied into the same row positions. The cells in the column that is being copied are not affected; this is a duplication, not a transfer.

To use the CCO command, move the cell cursor into any cell in the column that is the destination of the copied column. Then type the CCO command followed by the column number of the column whose cells you are copying. Type a semicolon at the end of the command and press RETURN. The message WORKING is displayed on the status line while the procedure is being executed. The cells from the copied column then appear in the same row positions in the new column. The cells in the old column are not affected.

Several precautions must be taken with the CCO command:

1. The CCO command overwrites any cell entries that are already present in the cells into which you are copying. This means that those cell entries are lost. You can protect a cell against overwriting by freezing the value with the FRE command. A frozen cell remains intact when a column is copied into the column where the frozen cell is located.
2. The CCO command does not adjust formulas when the column is copied. If any of the cells is used in a formula, you must change the formula manually. It is best to use CCO only to copy data into a column past all the columns you have already entered. Copying columns into the midst of filled columns can lead to errors and oversights even if you use the CINS command to insert a blank column before you copy. Like CCO, CINS does not adjust the formulas in the affected columns.
3. Be sure to type a semicolon at end of the CCO command or the command will not work.

**Example:** cco3;      Copies all the entries from column 3 into the column where the cell cursor is currently located.

**CDEL      SS**

Deletes all the cell entries in an entire column. If there are any columns to the right of the deleted column, they are moved one column to the left to fill in the deleted column. Be careful when you use the CDEL (Column DElete) command: CDEL does not adjust any formulas that are affected by a column deletion. Formulas that refer to cells to the right of a deleted column will most likely be affected, and you will have to change the formulas manually.

**Example:** cdel      Deletes the column where the cursor is currently located.

**CINS SS**

Inserts a blank column between filled columns. Columns to the right of the inserted column are moved one column to the right to make room for the inserted column.

Be careful when you use the CINS (Column INSert) command: CINS does not adjust any formulas that are affected by a column insertion. Formulas that refer to cells to the right of an inserted column will most likely be affected, and you will have to change the formulas manually.

**CM WP, SS**

When issued in the spreadsheet, clears the spreadsheet work area, but does not affect the other work areas. Spreadsheet work is lost after you issue a CM command, unless you save the work first.

After you issue a CM command, the computer displays a question to double check your intentions. In the word processor the question is: CLEAR ALL Y/N? In the spreadsheet the question is: ARE YOU SURE Y/N? The questions are the same: Are you sure you want to clear the current work area? Because all information cleared from the work area is lost unless you save it, this question gives you a last chance to change your mind before the work area is erased.

**COLOR SS**

Lets you select a new color for the screen background. When you first turn on the built-in software, the word processor work area displays a white cursor and pale yellow characters on a black screen. If you want to use a different color combination, issue a COLOR command in the spreadsheet. You can change the color of the screen only from the spreadsheet.

You can select only the screen background color with the COLOR command; other colors are selected automatically. To select a new color, use the numbers from Table 2-2.

TABLE 2-2. Screen Background Colors and Numbers

<i>Background</i>	<i>Characters</i>	<i>Number</i>	<i>Background</i>	<i>Characters</i>	<i>Number</i>
Black	Yellow/white	0	Orange	Yellow/white	8
Gray	Black/white	1	Brown	Yellow/white	9
Red	Yellow/white	2	Yellow-green	Black/white	10
Cyan	Black/white	3	Pink	Yellow/white	11
Purple	Black/white	4	Blue-green	Black/white	12
Green	Black/white	5	Light blue	Black/white	13
Blue	Cyan/white	6	Dark blue	Cyan/white	14
Yellow	Black/white	7	Light green	Black/white	15

To issue a COLOR command, enter command mode and type COLOR and the number of the color you want for the screen background. Type a semicolon at the end of the command.

When you switch to another program, the screen color change remains in effect.

When you select a new color, cell entries that were made previously are not all automatically changed to the new character color (if there is a new character color). To change the cell colors, just pass the cell cursor up and down one column. All the cells in each row are changed to the new character color as the cell cursor passes down the column.

All the characters in the word processor work area, however, are automatically changed to a new character color as soon as you switch to the word processor.

### **COPY      SS**

Lets you copy a cell entry into the cell where the cell cursor is currently located. If there is already a value in the recipient cell, it is overwritten unless it is protected by the FRE (frozen cell) command. The cell that is being copied is not affected by the COPY command; this is a duplication, not a transfer.

To use the COPY command, move the cell cursor into the cell where you want the duplicate entry to appear. Then type the COPY command followed by the cell number of the cell you are copying.

The COPY command does not adjust formulas when the cell is copied. If the duplicate cell is used in a formula that needs adjustment, you must change the formula manually.

**Example:** `copy3;4`      Copies the entry from cell 3;4 into the cell where the cell cursor is currently located.

### **DF      WP, SS**

Removes a file permanently from a disk. Use the DF (Delete File) command only when you no longer want to keep a copy of the file.

To delete a file you previously saved on disk, issue the DF command and type the name of the file when the prompt DELETE FILE: is displayed on the command line at the bottom of the work area. When you press RETURN, the disk takes a few seconds to remove the file from the disk. This command operates like the SCRATCH command in BASIC.

You can verify that the file has been deleted by viewing the disk catalog, which you access by issuing a CA command.

**FIT SS**

Copies a formula from a cell into another cell and adjusts the formula to fit the new cell. To use FIT, move the cell cursor to the cell where you want the adjusted formula to appear. Then type FIT and the cell number of the cell whose formula you want to adapt. The FIT command automatically changes the cell numbers in the formula to fit the new cell.

When you use the FIT command in manual calculation mode, the solution to the FITted formula is not calculated. Instead, the solution to the original formula is displayed in the new cell. You must enter auto calculation mode to solve FITted formulas. As soon as you enter auto mode, all FITted formulas in the work area are solved and the answers are displayed in the appropriate cells.

If you are already in auto mode when you enter a FITted formula, the correct solution to the FITted formula is displayed automatically in the new cell.

For example, if the formula in cell 3;6 is SUM 3;2 TO 3;5, and you move the cursor to cell 4;6 and issue the command FIT 3;6, the computer changes the formula to SUM 4;2 TO 4;5.

The difference between the formulas is that the row number has been adjusted to match the difference between the row of the original formula (row 3) and the row of the FITted formula (row 4). When the FITted formula is adjusted, the computer measures the difference between the row and column locations where the original formula appeared and the row and column coordinates of the cell where you issued the FIT command. Row and column number differences are then used to adjust the formula in the new cell.

**Example:** With the cell cursor in cell 7;5, issue the command FIT4;3. Cell 4;3 contains this formula; sum 1;3 to 3;3. The formula is adjusted to be sum 4;5 to 6;5.

The difference between cells 7;5 and 4;3 is three rows and two columns. To adjust the formula from cell 4;3, the computer adds three rows to each row and two columns to each column in the formula. The resulting cell numbers in the formula are calculated 1+3;3+2 TO 3+3;3+2, or sum 4;5 to 6;5.

**FL SS**

Returns numeric cell entries to the default floating point format. Floating point numbers have a variable number of decimal places. The decimal point “floats” to the appropriate place in the number.

Use the FL (FLoating point) command when you want to cancel an IN (integer format) or \$\$ (two-decimal-place dollar format) command.

Current cell entries are not affected when you change the numeric display format. If you change the format and want to adjust current cell entries, move the

cell cursor to each cell and press RETURN. The numeric entries will be converted to the current numeric display format.

## FORMAT

Prepares a NEW disk for file storage. Formatting divides the disk into sectors compatible with your disk drive and establishes a disk directory. You can format a used disk, but formatting erases everything on a disk, so do not format a used disk unless you are willing to erase all the information stored on it.

You can format the disk in two ways:

1. From BASIC by using the HEADER command. Use HEADER if you have not yet accessed the built-in software and you know you will want to store your work on a disk. The HEADER command is explained in Chapter 1.
2. From within the built-in software by using the FORMAT command. The FORMAT command can be issued only from the spreadsheet program. If you are using a different program and want to format a disk, switch to the spreadsheet. DO NOT switch back to BASIC to use the HEADER command because you will lose your work in the built-in program.

Follow these steps to format a disk from the spreadsheet:

1. Insert a new disk into the disk drive. If you have ever used the disk before, DO NOT format it unless you are certain you do not want to keep anything on the disk. Formatting erases any information already stored on the disk. Do not format disks that contain commercial software either.
2. Enter the command mode, type FORMAT, and press RETURN.
3. Type Y in response to ARE YOU SURE? if you are indeed sure you want to format the disk. This prompt question gives you a last chance to bail out before the format executes. Be sure the disk is blank or expendable before you type Y. Type N if you change your mind.
4. In response to the prompt, type in a disk name (up to 16 characters) followed by a comma and a 2-character identification for the disk. Every disk should be given a unique identification.

After a short period of time, the disk is formatted and you can save files on it.

*Note:* Do not store file manager files on a disk that contains any other type of file, including word processor or spreadsheet files. Each file manager file should have its own disk.

**FRE SS**

Protects a cell entry from being changed by an inserted, deleted, or copied cell or by any change in the numeric display format or a formula. The value of a frozen cell cannot be changed at all until you cancel the freeze with the THAW command.

To FReeze a cell, move the cell cursor to the cell whose entry you want frozen. Issue the FRE command.

When the cell cursor enters a frozen cell, an asterisk is displayed on the status line to indicate that the cell value is frozen.

Example: **FRE**      Freezes the entry in the current cell so it cannot be changed.

**FU WP, SS**

Cancels half-screen mode (see the HA command) and returns the screen to a full display of the spreadsheet work area. The FU (FUll screen) command has no effect unless the computer is in half-screen mode, which is turned on by the HA command. Issuing an FU command does not affect the contents of either the word processor or the spreadsheet.

To return to the full-screen spreadsheet, you must be in the spreadsheet when you issue an FU command. If you issue the command in the word processor, you will return to the full-screen word processor. If this happens, just issue a TC command to switch to the spreadsheet.

**GOTO SS**

Moves quickly and directly to a specified cell without using the cursor movement keys. To use GOTO, type GOTO and the number of the cell that is your destination.

Example: **goto 12;10**      Sends the cell cursor directly to cell 12;10.

**HA WP, SS**

Divides the screen in half so that partial work areas from the spreadsheet and word processor programs can be displayed simultaneously. In half-screen mode, 12 lines from the word processor are displayed in the top half of the screen, and 7 rows from the spreadsheet are shown in the bottom half of the screen.

Half-screen mode has several uses:

1. When you want to transfer data from the spreadsheet row by row with the MAP command. MAP lets you transfer data manually, and half-screen mode lets you see both work areas while you use MAP.
2. When you need to refer to spreadsheet data while you are writing a word processor document, but do not want to transfer the data.
3. When you need to get data from the word processor for use in the spreadsheet. You cannot transfer data from the word processor to the spreadsheet, but half-screen mode lets you view any part of the word processor work area while you are working in the spreadsheet.

When you first issue the HA command in the word processor, the screen is split in half, but the spreadsheet is not displayed in the work area. When you first issue the HA command in the spreadsheet, the spreadsheet is displayed in the bottom half of the screen, and the word processor work area is not visible. To bring up the other program, issue a TC or TW command. The other program is then displayed on the other half of the screen.

### **Cursor Control**

Although you can see partial work areas from both programs, you can use only one program at a time. The cursor can move around only half of the screen, and the keyboard is under the control of the current program. You cannot type any information into the other program. To switch between programs, you must still issue TC and TW commands. The status line at the bottom of the screen indicates which program currently controls the keyboard.

When you are in half-screen mode, you can still view any segment of the whole work areas of either program. Just use the cursor keys and other cursor movement keys to display other parts of the work area on the half screen.

#### **HOME      SS**

Moves the cell cursor directly and quickly to cell 1;1.

**Example:** **home**      Moves the cell cursor directly to cell 1;1.

#### **ID      WP, SS**

Initializes the current disk. Before you use the ID (Initialize Disk) command, make sure the drive is on and the disk is inserted.

You can initialize the disk at any time. Your current work is not affected, nor are any files on the disk. Whenever you change disks, you should issue the ID command after you insert the second disk so that it is initialized.

Example: id      Initializes the disk currently in the drive.

### IFTRUE

Checks the condition of part of a formula. IFTRUE lets you make cell entries based on conditions in other cells. For example, if you are figuring a budget, you can put an amount of money into the savings column IF it is TRUE that income minus expenses for the month is a positive number.

The IFTRUE formula is a compound command consisting of the following:

1. The first part contains the calculation whose outcome is checked by IFTRUE.
2. The IFTRUE command is typed next.
3. The final part is the assignment of a value to a cell, which is executed ONLY if the condition in the first part is met.

The first part of the formula usually contains a comparison operator. Four comparison operators you can use are described in Table 2-3.

The third part of the IFTRUE formula uses the left-pointing arrow to assign a value to a cell, which occurs only when the first part of the formula is true.

The IFTRUE formula can be used while the cell cursor is in any cell. However, when the IFTRUE formula is executed, a 0 (when false) or 1 (when true) is placed in the current cell if that cell is not also the destination for the IFTRUE formula. So do not issue an IFTRUE when the current cell contains a value you want to protect.

You can also use the NOTIFTRUE command to check conditions in a formula. NOTIFTRUE assigns the value in part 3 of the formula only when the calculation in the first part of the formula is not true.

Examples: 10;3 > 11;3 iftrue 12;3 ← 10;3

Assigns the value of cell 10;3 to cell 12;3 if the entry in 10;3 is greater than the entry in 11;3.

**TABLE 2-3. Comparison Operators**

<i>Symbol</i>	<i>Meaning</i>
=	equal
nte	not equal
>	greater than
<	less than

**12;10 + #100 = 4;10 iftrue 3;10 ← #250**

Assigns the number 250 to cell 3;10 if the value of 12;10 plus 100 is equal to the entry in cell 4;10.

## **IN      SS**

Displays numeric cell entries as whole numbers only, regardless of whether or not the numbers were entered with decimal parts. Decimal parts of numbers entered are simply chopped off; they are not rounded. For example, if you are using the INteger format and enter 12.9, the spreadsheet will display 12 in the cell.

Current cell entries are not affected when you change the numeric display format. If you change the format and want to adjust current cell entries, move the cell cursor to each cell and press RETURN. The numeric entries will be converted to the current numeric display format.

Use the FL command to cancel an IN command.

Although integer format truncates decimal parts, the spreadsheet remembers the entire value you entered. If you go back to floating point format, send the cell cursor back into the cell and press RETURN, the decimal part will then be displayed.

**Example:** **in** Changes the format of numeric entries so only whole numbers are displayed.

## **LEFTJ      SS**

Displays numeric entries LEFT Justified in their cells. Ordinarily numeric entries are right justified, which means they are flush against the right margin of the cell. Left justification displays numeric entries flush against the left margin of the cell. Text entries are automatically left justified.

The cell cursor can be anywhere when you enter a LEFTJ command. Only numeric entries that you make after the command is issued are affected. You can, however, also change previous entries by moving the cell cursor back to each cell and pressing RETURN.

Cancel LEFTJ with the RIGHTJ command.

**Example:** **leftj** Forces numeric entries to be displayed flush against the left margin of the cell.

## **LF      WP, SS**

Lets you bring a file stored on disk into the work area. When you issue the LF (Load File) command, the computer displays the message LOAD FILE: on the status line at the bottom of the screen. Type the name of the file you want to load.

When you load a file with the LF command, the work area is cleared before the file is displayed. The loaded file is displayed at the top of the work area regardless of the current location of the cursor. Because the LF command always clears the work area before the file is loaded at the top of the work area, any work in the work area will be lost unless you saved it before you issued the LF command.

### Error Messages

If the message FILE NOT FOUND >>> is displayed on the status line after you issue an LF command, the disk in the drive does not contain the file whose name you typed in response to LOAD FILE. Either the wrong disk is in the drive or you misspelled the file name when you typed it. Check the disk catalog to make sure the file exists on that disk; then reissue the command. If the file does not exist on the disk and you are sure you have the right disk, most likely the file was never saved. If you must switch disks, be sure to issue the ID command after doing so.

If the message NO FILE! is displayed on the status line, the disk drive is not connected, is not turned on, or no disk is inserted in the drive.

**Example:** **lf** After you type lf, the computer displays LOAD  
**LOAD FILE: budget** FILE. Type the name of the file you want to load.

**MAN SS**

Cancels automatic calculation mode. When MANual mode is on, formulas are calculated only when you enter the formula cell and press RETURN. Formulas entered with the FIT command are not calculated. In addition, when you are in manual mode and change a cell entry that was used in a formula, any other cell entries that are affected by the change are not recalculated. To perform these calculations, you must use the AUTO command to enter auto mode.

The default calculation mode is manual mode. The current calculation mode is indicated by MANU. or AUTO., which is displayed at the far right on the status line.

**Example:** **MAN** Cancels auto mode and returns to manual calculation mode.

**MAP SS**

Lets you transfer cells of spreadsheet data by rows to the word processor. Transferring data with the MAP command is manual: you move the cell cursor into each cell you want to transfer. Because you select each cell, you should use MAP in half-screen mode so that you can see both work areas simultaneously.

Use MAP when you want to transfer only selected cells from the spreadsheet. The spreadsheet data are MAPped to the current word processor cursor location.

MAPped data overwrites word processor text, so be sure the cursor is beneath all the text or create enough blank spaces within the text to avoid overwriting.

Use the IL (Insert Line) command to open up blank lines. You can also create a block of blank lines with the SP (Set Pointer), CB (Create Block), and IB (Insert Block) commands. To create a block of blank lines, move the cursor to a blank area on the word processor screen, set a pointer at the bottom of the appropriate number of blank lines, and create a block. Then move the cursor to the line in the text where you want to locate the blank block, and insert the block.

When you use MAP, you set a top left corner of the area to be transferred. You cannot MAP above or to the left of this corner. No bottom right corner is set, and you can leave out any cell to the right of the leftmost column just by not putting the cell cursor into a cell.

Every time you enter a row, whether you have been in it before or not, you must move the cell cursor to the leftmost cell on that row. Otherwise you will not be able to MAP any cells in the row. If you have previously MAPped the leftmost cell, you still have to return to it to MAP any more cells when you reenter the row.

After you MAP the leftmost cell in a row, you do not have to MAP cells in any particular order. You can move more cells in the row or go immediately to the next row. You can go back up to rows, skip cells, or skip down to other rows, as long as you visit the leftmost cell each time you enter the row.

To use MAP, follow these steps:

1. Turn on the half-screen mode (**C C**, then **HA** and **RETURN**).
2. Switch to the word processor (**C C**, then **TW** and **RETURN**).
3. Move the word processor cursor to the place on the screen where you want to place the MAPped data. Note that MAPped data overwrite word processor text, so move the word processor cursor under all the text, or open up enough blank space within the text to prevent text loss.
4. Switch back to the spreadsheet (**C C**, then **TC** and **RETURN**).
5. Put the cell cursor in the cell that is to be the upper left corner of the cells MAPped to the word processor.
6. Issue the command **MAP** and press **RETURN**.
7. Move the cell cursor into every cell you want to transfer.

You can move to the next row and continue MAPping without issuing another MAP command. However, once the first row establishes the leftmost column of a MAPped row, subsequent rows must also begin in that column. Use the F1, F2, and cursor up and down keys to move across rows to transfer data.

## Using MAP to Transfer Long Cell Entries

When you first enter data into a cell, you can enter up to 36 characters per cell, but only 11 characters are displayed. Extra characters are held in memory and are displayed on the command line when you enter the cell. MAP sends these extra characters, including long formulas. In contrast, BLKMAP transfers only 11 characters per cell.

### Turning OFF the MAP Command

The OFF command lets you stop the MAPping procedure. You can also terminate MAP by switching to the word processor.

### Issuing Other Commands During a MAPping Procedure

Unless you terminate MAP by using the OFF command or by leaving the spreadsheet, MAPping remains in effect. You can issue other spreadsheet commands during MAPping without turning MAP off. You can also make changes in cell entries during a MAPping procedure without interrupting the transfer process. The changed cell entry is MAPped to the word processor.

- Example:
- map      Turns on cell MAPping.
  - off      Turns off cell MAPping.

## NOTIFTRUE

Checks the condition of part of a formula. NOTIFTRUE lets you make cell entries based on conditions in other cells. For example, if you are figuring a budget, you can put an amount of money into the savings column IF it is NOT TRUE that income minus expenses for the month is a negative number.

The NOTIFTRUE formula is a compound command consisting of the following:

1. The first part contains the calculation whose outcome is checked by NOTIFTRUE.
2. The NOTIFTRUE command is typed next.
3. The final part is the assignment of a value to a cell, which is executed ONLY when the condition in the first part is not met.

The first part of the formula usually contains a comparison operator. Four comparison operators you can use are shown in Table 2-4.

TABLE 2-4. Comparison Operators

<i>Symbol</i>	<i>Meaning</i>
=	equal
nne	not equal
>	greater than
<	less than

The third part of the NOTIFTRUE formula uses the left-pointing arrow to assign a value to a cell, which occurs only when the first part of the formula is not true.

The NOTIFTRUE formula can be used while the cell cursor is in any cell. However, when the NOTIFTRUE formula is executed, a 0 (when false) or 1 (when true) is placed in the current cell if that cell is not also the destination for the NOTIFTRUE formula. Therefore, do not issue a NOTIFTRUE when the current cell contains a value you want to protect.

You can also use the IFTRUE command to check conditions in a formula. IFTRUE assigns the value in part 3 of the formula only when the calculation in the first part of the formula is true.

Examples: 8;6 = 9;6 notiftrue 10;6 ← 7;6

Assigns the value of cell 7;6 to cell 10;6 if the entry in 8;6 does not equal the entry in 9;6.

2;1 \* #.6 < 4;1 notiftrue 3;1 ← #7.5

Assigns the number 7.5 to cell 3;1 if the value of cell 2;1 times 0.6 is not less than the entry in cell 4;1.

## OFF      SS

Turns off the MAP command when you are finished transferring rows of spreadsheet data to the word processor.

Example: off      Cancels a MAP command.

## RCO      SS

Lets you copy the cell entries from one row into another. RCO (Row COpy) writes a duplicate of all cell entries from the row you are copying into the row

where the cell cursor is currently located. The cells are copied into the same column positions. The cells in the row that is being copied are not affected; this is a duplication, not a transfer.

To use the RCO command, move the cell cursor into any cell in the row that is the destination of the copied row. Then type the RCO command followed by the row number of the row whose cells you are copying. Type a semicolon at the end of the command and press RETURN. The message WORKING is displayed on the status line while the procedure is being executed. Then the cells from the copied row appear in the same row positions in the new row. The cells in the old row are not affected.

Several precautions must be taken with the RCO command:

1. The RCO command overwrites any cell entries that are already present in the cell into which you are copying. This means that those cell entries are lost. You can protect a cell against overwriting by freezing the value with the FRE command. A frozen cell remains intact when a row is copied into the row where the frozen cell is located.
2. The RCO command does not adjust formulas when the row is copied. If any of the cells is used in a formula, you must change the formula manually. It is best to use RCO only to copy data into a row below all the rows you have already entered. Copying rows into the midst of filled rows can lead to errors and oversights even if you use the RINS command to insert a blank row before you copy. Like RCO, RINS does not adjust the formulas in the affected rows.
3. Be sure to type a semicolon at the end of the RCO command or the command will not work.

**Example:** rco3; Copies all the entries from row 3 into the row where the cell cursor is currently located.

#### RDEL . SS

Deletes all the cell entries in an entire row. If there are any rows below the deleted row, they are moved up one row to fill in the deleted row. Be careful when you use the Row DElete command: RDEL does not adjust any formulas that are affected by a row deletion. Any formulas that refer to cells below a deleted row will most likely be affected, and you will have to change the formulas manually.

**Example:** rdel Deletes the current row.

#### RESET SS

Clears the work areas of all the built-in programs and starts the built-in software over as if you had just turned the software on. The computer goes back

to the software title screen, not to BASIC, so you do not press the F1 key to get back to the software.

After you issue a RESET command, the computer displays the message ARE YOU SURE Y/N. If you are certain you are willing to erase all your work areas and return the built-in programs to a just-turned-on condition, type Y. If you decide against resetting the built-in programs, type N.

The difference between the RESET command and the RESET button, which is located on the side of the computer, is that the RESET command clears the built-in software and returns the software to a just-turned-on condition while the RESET button cancels the built-in software and returns the computer to BASIC, where the initial power-on message is displayed.

The RESET command can be issued only from the spreadsheet.

**Example:** **reset**      Clears all software work areas and returns them to a just-turned-on condition.

#### **RIGHTJ      SS**

Cancels the LEFT Justify command. RIGHTJ (RIGHT Justify), which is the default condition, displays numeric entries right justified, which means they are flush against the right margin of the cell. Left justification displays numeric entries flush against the left margin of the cell. Text entries are automatically left justified.

The cell cursor can be anywhere when you enter a RIGHTJ command. Only numeric entries you make after the command is issued are affected. You can, however, also change previous entries by moving the cell cursor back to each cell and pressing return.

**Example:** **rightj**      Cancels LEFTJ and forces numeric entries to be displayed flush against the right margin of the cell.

#### **RINS      SS**

Inserts a blank row between filled rows. Rows below the inserted row are moved one row down to make room for the inserted row.

Be careful when you use the RINS (Row INSert) command: RINS does not adjust any formulas that are affected by a row insertion. Any formulas that refer to cells below an inserted row will most likely be affected, and you will have to change the formulas manually.

**Example:** **rins**      Inserts a blank row at the location of the cell cursor.

SF WP, SS

Lets you store the current spreadsheet work area on a disk. To store a file, follow these steps:

1. Insert a formatted disk. Use the ID command to initialize it.
  2. Issue the SF command.
  3. Type a file name when the message SAVE FILE: is displayed on the status line. You must give each file a name. The file name must be between 2 and 16 characters long.
  4. If a file by the given name already exists on the disk, REPLACE Y/N is displayed. Type Y in response to REPLACE Y/N unless you want to keep the old copy of the file. In this case, type N and save the current file with a different name.

If you do not have a formatted disk, insert a new disk and issue a **FORMAT** command before the **SF** command.

You can verify that the file is stored by issuing a CA command, which displays a list of all the files on the disk. Make sure the number of blocks assigned to the file is greater than zero. If the number of blocks is zero, the file was not saved. Press RETURN and issue the SF command again.

**TF**      **WP, SS**

Lets you switch to the file manager program. You can issue the TF (To the File manager) command at any time. The spreadsheet work area is not affected when you leave the program. Use the TC command to switch back to the spreadsheet.

Example: **tf**      Switches to the file manager program.

**TW      SS, FM**

Lets you switch to the word processor program. You can issue the TW (To the Word processor) command at any time. The spreadsheet work area is not affected when you leave the program. Use the TC command to switch back to the spreadsheet.

Example: tw      Switches to the word processor program.

**THAW      SS**

Lets you cancel a FRE command, which protects a cell entry from being changed by an inserted, deleted, or copied cell or by any change in the numeric display format or a formula. The value of a frozen cell cannot be changed at all until you cancel the freeze with the THAW command.

Once a frozen cell is THAWed, it is subject to changes like any other cell.

**Example:** **thaw** Cancels the freeze on a cell value.

**TRANSFER      SS**

Lets you copy the contents of one cell into another cell. You can also use TRANSFER, which is always represented by a left-pointing arrow, to put a number into a cell. TRANSFER is never issued by itself; it is used as part of IFTRUE and NOTIFTRUE compound formulas.

**Example:** **3;1=#55 iftrue 5;1 ← #33** Transfer the number 33 into cell 5;1 if cell 3;1 equals 55.

**\$\$      SS**

Displays numeric cell entries with two decimal points, regardless of whether or not the numbers were entered with decimal parts. If a number contains more than two decimal places, the remaining numbers are simply chopped off; they are not rounded. For example, if you are using the \$\$ (dollar) format and enter 12.9999, the spreadsheet displays 12.99 in the cell. If you enter number 55, the spreadsheet displays 55.00.

Current cell entries are not affected when you change the numeric display format. If you change the format and want to adjust current cell entries, move the cell cursor to each cell and press RETURN. The numeric entries will be converted to the current numeric display format.

Use the FL command to cancel a \$\$ format command.

Although the \$\$ format truncates decimal numbers with more than two decimal places, the spreadsheet remembers the entire value you entered. If you go back to floating-point format, send the cell cursor back into the cell and press RETURN, the full decimal part will then be displayed.

**Example:** **\$\$** Changes the format of numeric entries so that all numbers are displayed with two decimal places.

## **Section 4. The File Manager**

The file manager lets you keep records of many types of information. You can store addresses, product information, bibliographies, and any other sort of information that can be adapted to a standard form.

The file manager is organized into files, records, and fields. Each record in a file contains a number of individual fields, such as name, state, or phone number. You design the file yourself by setting the number, name, and length of fields. You enter information into each field for each record. Then you store the records in the file.

After file manager records are stored, you can display them again, sort them by any field, search the records for a specific piece of information, create subfiles, and send field information to the word processor for incorporation into a printed document. The format instructions used for sending field information to the word processor are explained in Section 2 of this chapter.

## File Manager Commands

### **CA      WP, FM, SS**

Displays a listing of all the files on the current disk. The file CAtalog includes the following information:

- File names
- File lengths (stated in blocks)
- Blocks remaining on the disk

Spreadsheet files have a .c suffix appended to each file name. Word processor files have no suffix. File manager disks (each file should have its own) show only their name and blocks free. The total blocks free on the disk shows you how many of the blank disk total of 664 blocks remain available.

The catalog information is displayed on a separate screen. The work area is not affected; you can display a disk catalog at any time. When you finish looking at the catalog and press RETURN, the intact work area is returned to the screen.

Example: **ca**      Lists a catalog of files on the current disk.

### **DS      FM**

Lets you sort a file by any of up to three fields. The sort creates a temporary subfile of records reorganized by the sort criteria. You can use the subfile in searches, in record reviews, or for printing multiple copies of a word processor document. The DS (Disk Sort) command does not affect record numbers or contents, but just the order in which records are organized.

When you want to sort by two or three fields, type the field numbers in the order of sorting priority. To sort first by state and then by name, type the state field number and then the name field number. Separate multiple field numbers with semicolons (e.g., ds7;9;3;).

When you sort a file, the word processor work area is automatically cleared. If you have work in the word processor work area, save it before you issue the DS command. To perform a disk sort, follow these steps:

1. Insert the disk that contains the file you want to sort. Use the ID command to initialize it.
2. Type the DS command and the field number of the fields you want to use to sort the file (e.g., ds5;3; sorts by the fifth field and then by the third field).

After you issue the DS command, the message SYSTEM WILL CLEAR WORD PROCESSOR TO SORT Y/N? is displayed. Type Y to proceed or N to abort the sort. While the sort is executing, the file manager displays the message BEGIN DISK SORT ON with the field numbers of the sort. When the sort is finished, the file name, number of records, and number of the last record stored are displayed.

Records are sorted using the CHR\$ code values of the contents of the sort field(s). This means that fields containing only letters are sorted alphabetically whereas fields containing only numbers are sorted numerically. Fields containing a mixture of letters and numbers or punctuation marks (including blanks) may not be sorted as you would expect. Refer to the CHR\$ code list in Appendix C for the values used.

The sort creates a temporary subfile of the old file records rearranged. To review the sorted file, use the RV (ReView) command. Records are displayed in the sorted order, and the actual record number of each record is displayed at the bottom of screen during the review. You can display individual records with the RC (ReCord) command. Use the old record numbers, which are not changed when the file is sorted.

The subfile is destroyed when you turn off the computer, re-sort the file, or load another file manager file. The subfile is not lost, however, when you switch to another program.

You can also terminate the subfile with the RESETLIST command, which returns the file to its normal condition.

Examples:    **ds2;**              Sorts the file by the second field.

**ds4;7;1;**      Sorts the file first by field 4, then by field 7, and finally by field 1.

## HIGHRC     FM

Sets an upper limit on the records to be included in a subfile. The record number in the HIGHRC (HIGH ReRecord) command prevents all those records with higher record numbers from being accessed until the subfile is disbanded.

To use the HIGHRC command, type HIGHRC and the record number you want to be the last in the subfile. Type a semicolon at the end of the command.

The subfile is disbanded when you turn off the computer, load another file manager file, re-sort the file, or issue a RESETLIST command.

**Example:** highrc75; Sets record number 75 as the highest record that can be accessed.

### **NEWTF      FM**

Lets you design a new file. In the design process, you give the file a name, set the total number of fields for the file, and give the names and maximum lengths for each field. The field information you enter in the file design is used to prompt you when you enter information for each record.

When you design a new file, have a new, blank disk ready. The disk need not be formatted. Any information that is currently on the disk will be lost during the new file construction. Each new file should be stored on its own disk. Never store a file manager file on the same disk with other types of files, including files from the other built-in programs. The file manager creates a large "file" using direct-access disk commands. The file does not appear in the disk directory. Do not use the BASIC COLLECT command on file manager disks.

When you first switch to the file manager, the message TYPE TF OR NEWTF asks if you will be using an old file (TYPE TF) or creating a new one (TYPE NEWTF). When you respond with NEWTF (NEW To File), the computer displays the following prompts:

#### **ENTER FILE NAME (1..16)**

You give the file a name up to 16 characters long. This is used as the name for the disk during formatting.

#### **ENTER NUMBER OF FIELDS 1..17 01;**

You enter the total number of fields the file will contain. The default is 1 and the maximum is 17. Just type over the 01 to select a number. The trailing semicolon is required.

#### **ENTER FIELD NAME (1..35) FIELD # 01;**

You enter the name of the first field. The name can be up to 35 characters long.

#### **ENTER FIELD LENGTH 1..38 FIELD # 01; 01;**

You enter the maximum number of characters to be entered in field 1. The default is 1 and the maximum is 38. Just type over the 01 to select a number. Again, the trailing semicolon is required.

The third and fourth prompts are repeated for each field until the total number of fields (entered at the second prompt) are defined.

When you finish entering all the fields, the screen clears and a review of the file design is displayed. A message telling you how many records you can store (999) for the file is also displayed.

The computer also asks OK TO FORMAT DISK? Y/N. You should have a new, blank disk ready. Insert it in the disk and type Y in response to the prompt question. The formatting procedure takes a short time. When the computer reports that the disk is ready, your new file design is stored on it. You are then ready to enter records.

**Example:** newtf      Tells the file manager that you are ready to design a new file.

#### **NR      FM**

Enters information into records. Unlike entering records with the RC (ReCord) command, the NR (Next Record) command automatically moves to the next record after each record is entered. The NR command saves time, and you should use it instead of RC when you are entering a series of records.

To use the NR command, follow these steps:

1. Issue a TF command to display the record number of the last record entered if you are adding records to a previously stored file.
2. Issue an RC command to display the first record you are going to enter. If you are entering records into a new file, the first record would be RC1;; and if you are adding records to an old file, the first record would be 1 plus the record number named in the TF command as the last record entered.
3. Store the first record with the UD (UpDate record) command.
4. Issue an NR command to enter subsequent records. NR displays the next record automatically.
5. Store each record with the UD (UpDate record) command.
6. Terminate the NR command by entering command mode and issuing a command.

**Example:** nr      Displays a series of records so you can quickly enter information.

**PI      FM**

Lets you set limited criteria for the creation of a subfile. You can pick alphabetic or numeric ranges that limit the records included in a subfile. The range applies to one field, which you specify when you issue the PI (PIck) command. For example, you can PIck a limited range of area codes from a phone number field.

After the subfile is PIcked, you can search, sort, review, or transfer the records to the word processor. For example, you can limit a sorted file to only those records whose name field starts with A through F.

Note that while PI creates a subfile, it does not reorganize the records in the subfile. The records in a PIcked subfile are still arranged in the order the records were entered. If you issue a HIGHRC command before you issue a PI command, the PI command will use only the records whose numbers are less than the high record.

The PI command looks for exact matches between the range limits you enter and the entries in the records. The PI command distinguishes between upper and lower case letters, so be sure to note this difference when you enter the range limits.

To use the PI command, follow these steps:

1. Type pi and the number of the field you will use to pick a limited range of records. End the PI command with a semicolon.
2. Type the low end of the PIcked range in response to the prompt BOTTOM: and press RETURN. You can use one or more letters or numbers, a word, or any phrase up to 38 characters long. For example, to PIck a subfile of zip codes that begin with 190 through 194, you would type 190 as the BOTTOM of the PIcked range.
3. Type the high end of the PIcked range in response to the prompt TOP: and press RETURN. Again, you can use up to 38 characters as the range limit.

The PI command creates the limited subfile by searching all the records in the file. While the PI command is executing, a left-pointing arrow is displayed for each record that is put in the subfile.

To review the PIcked subfile, issue an rv1; command. You can also issue commands to sort or search the PIcked subfile.

Use RESETLIST to delete a PIck subfile.

Example:	pi5; BOTTOM: My	Uses field number 5 as the basis for the PIcked subfile. Sets My as the bottom of the PIcked range.
----------	--------------------	--

**TOP: Q**

Sets an uppercase Q as the high end of the Picked range. With these limits, the subfile will contain entries from field 5 that start with My through Qy.

**RC      FM**

Displays any record. You can also use the RC (ReCord) command to enter information into a record. The RC command displays the record whose record number you give in the command. You can display a filled record or an empty record. Unlike the RV (ReView records) command, which displays all the records in a series, the RC command displays only one specific record. Once the record is displayed, you can make changes if you like.

When you display an empty record, you can enter information into it. Unlike entering records with the NR (Next Record) command, the RC command does not automatically move to the next record after each record is entered. Use RC when you are entering just a few records at a time, and use NR when you are entering a series of records.

To use the RC command to display a record, type RC and the record number of the record you want to display followed by a semicolon.

To use the RC command to enter a record, follow these steps:

1. Issue a TF command to display the record number of the last record entered if you are adding records to a previously stored file.
2. Issue an RC command to display the first record you are going to enter. If you are entering records into a new file, the first record would be RCI;, and if you are adding records to an old file, the first record would be 1 plus the record number named in the TF command as the last record entered.
3. Store the first record with the UD (UpDate record) command.
4. Issue another RC command to enter the next record. You must enter another RC command for each record you enter.
5. Store each record with the UD (UpDate record) command.

**Example:**    **rc3;**    Displays record number 3, which may or may not be a filled record. You can view it or change it if it is filled, or you can enter information if it is not filled.

**RESETLIST      FM**

Cancels any type of subfile. The RESETLIST command disbands sorted subfiles, searched subfiles, and subfiles set by the HIGHRC or PI (PICK) commands. RESETLIST has no effect on the contents of the records; it is NOT like the spreadsheet RESET command.

Because RESETLIST restores the file to its original organization, you should use it before you create a subfile when you want to be sure a previous subfile does not affect your current project.

**Example:** **resetlist** Restores the file to its original organization by canceling a subfile.

## **RV FM**

Displays the records in order. If the file has been reorganized into a subfile, the records are displayed in subfile organization with the actual record number also displayed on the screen. The RV (ReView) command starts with the record whose number you give in the command and continues through the file, displaying each record quickly:

To speed up the review, press the space bar. To slow it down, press the S key. To end the review, press the Q key.

Use the RV command to make a quick review of a new file or a newly created subfile or to scan for a particular record whose number you do not remember. To use the RV command, type rv and the number of the first record you want to review, followed by a semicolon.

**Examples:** **rv1;** Quickly displays all the records in the file or subfile, starting at record 1.

**rv25;** Starts the review at record 25.

## **SR FM**

Looks through records to find those that contain data you specify. You can search for any letters or numbers up to 38 characters in length. The SR (SeACh) command searches every field; the search is for a character match rather than for the contents of a specific field. In other words, the SR command does not search just one field in each record. The search criteria cannot spread into two fields. For example, if you are searching for New York and the words New and York appear in separate fields in a record, this is not considered a match.

To use the SR command, type sr and press RETURN. When the prompt SEARCH is displayed, type the characters you want to find in the records. Unlike the PI (PIck) command, the SR command does not distinguish between upper and lower case letters, so Computer and computer are considered equal.

If there is no subfile, the SR command begins its search with record 1 and sequentially investigates every record in the field. If there is a subfile, the SR command searches records according to the organization of the subfile.

As a search executes, each record that contains a match is displayed. The message CONTINUE Y/N is also displayed. If you want to continue searching for more instances of a match, type Y. If you want to abort the search, type N.

**Example:** **sr**                   Initiates a search that will look through each field in  
**SEARCH: 1919**        each record to find the numbers 1919.

**TC      WP, FM**

Lets you switch to the spreadsheet program. You can issue the TC (To the Calculator) command at any time. Use the TF command to switch back to the file manager.

**Example:** **tc**        Switches to the spreadsheet program.

**TF      WP, SS, FM**

In the file manager, the TF command tells the computer you want to use an already stored file. Enter TF to use an old file when the file manager displays the message TYPE TF OR NEWTF. You can issue a TF command at any time to display the record number of the last record entered. This information is useful when you are about to add more records to a file.

The TF command displays the following information about the current file:

The file name

The number of records used of the original 999

The record number of the last record entered

If a subfile has been created and is still in effect, the top of the status report displays the number of records in the subfile. The last record entered is displayed on the second line of the status report.

**Example:** **tf**        Displays information about a stored file and about a subfile if one is present.

**TW      SS, FM**

Lets you switch to the word processor program. You can issue the TW command at any time. Use the TF command to switch back to the file manager.

**Example:** **tw**        Switches to the word processor program.

**UD      FM**

Stores a record on the disk. Use the UD (UpDate) command after you enter or change a record with either the RC (ReCord) or NR (Next Record) command.

The UD command stores the current record with the record number currently displayed unless you specify some other record number in the UD command.

You can also use UD as a shortcut to make duplicates of a record or when you want to store a record that is almost identical to the current record. To store a record with a record number other than the one displayed on the screen, just include a record number in the UD command. For example, if you are storing records of a stamp collection and have two similar stamps, enter and store the first stamp as record 1. Then change the record while record 1 is still displayed on the screen. Save this version of the record with the command ud2;.

Examples: ud Stores the current record.

ud37; Stores the record information currently on the screen as record number 37.

---

## **3** Some Programming Techniques

---

The BASIC built into the Commodore Plus/4, Version 3.5, is the most powerful and versatile version of BASIC that Commodore has ever used in a computer. This chapter explains some of the major programming techniques you can use in writing BASIC programs as well as a few machine language techniques. These include the following topics:

- Using the screen editor
- Using the Escape key screen editing functions
- Using screen windows
- Using text strings
- Redefining the function keys
- Using mathematical functions
- Programming sound and music
- Using arrays
- UnNEWing programs
- Using the built-in error-trapping routines

Each of these topics is covered in a separate section of the chapter. Sample programs are used to illustrate the use of each technique.

For explanations of all BASIC commands, see Chapter 1. For extensive information on programming graphics, see Chapter 4. For in-depth descriptions of commands for handling disk drives and other peripherals, see Chapter 6.

## Using the Screen

The computer screen is 40 columns by 25 lines, which means it can display 1000 characters at a time. These 1000 character places have their own locations in memory in what is called the Screen Memory Map.

The top left corner of the screen has a memory address of 3072 (\$0C00). The character just to the right of that location has an address of 3073. The bottom right corner of the screen is at address 4071 (\$0FE7). Each character position on the screen has a specific address.

Each time you type a character or the computer displays one, the computer updates the screen memory at the character position where the new character appears. For example, if you type SCNCLR, the computer clears the screen and displays the READY. prompt on the second line of the screen. At this point, screen memory address 3112 contains an R, address 3113 contains an E, and so on. All other screen memory addresses contain a blank. The values stored in screen memory to display characters are not the same as CHR\$ code values. See Appendix E for a list of screen display codes.

A screen memory location is updated every time that character position gets a new value. When a line moves up because the screen scrolls up, the characters in the line are removed from the old memory locations and are registered in the new locations. Screen memory has only one task: to keep track of each character position on the screen. It does not evaluate the text on the screen for errors; that is done by the computer when you press the RETURN key.

## POKEing and PEEKing

You can use the POKE command to put a specific character at a specific screen location. Use the screen memory locations and the screen display codes listed in Appendix E (not the CHR\$ codes) in POKE commands.

You can also POKE a color into the color memory location corresponding to a character position. The color of each character position is registered separately from the character itself. The color memory map, which is similar to the screen memory map, begins at memory location 2048 (\$0800) (the top left corner of the screen) and ends at location 3047 (\$08E7) (the bottom right corner of the screen).

Examples:

```
10 INPUT"PLAYER'S SUIT";S$  
20 IF S$ = "HEARTS"THEN  
POKE 3441,83: POKE 2417,2+16*4
```

Puts a red (color = 3 1; luminance = 4) heart symbol at column 9, row 9 (columns and rows counted from 0).

You can find out what value is at a memory location by using the PEEK function. PEEK returns the code that stands for the current occupant of the memory location into which you are PEEKing. For example:

PRINT PEEK(3441)      Displays the screen display code for the character  
83                        at screen memory location 3441.

You can POKE and PEEK values at memory locations other than just the screen memory.

## Program Lines

Each program line can take up to 88 characters, which is just over two lines on the screen. If a line is longer than 88 characters, the computer rejects the line and displays the ?STRING TOO LONG ERROR message as soon as you press RETURN.

The computer requires a RETURN key press for every program line. When you press RETURN, the computer interprets the BASIC and stores it in the program area of memory as tokenized BASIC. If there is already a line with that line number in memory, the old line is replaced by the new line. When you return to a line to make changes, you must press RETURN to register the changes, and you can press RETURN anywhere in the line.

Using the abbreviations for BASIC keywords can allow you to fit extra commands on a line. When the line is printed, the keywords are spelled out. This means you cannot cursor up to the line and reenter it using the RETURN key. You must retype such long lines to change them.

## Copying Program Lines without Retyping

When you enter program lines, the information is stored in program memory and (as long as the lines appear on the screen) in screen memory. These memory areas operate independently, so you can change one area without necessarily changing the other. For example, if you go back to a line and make some changes, those changes are immediately updated in screen memory. The changes are not updated in program memory, however, unless you press the RETURN key while you are somewhere on that program line. If you do not press RETURN and just move off the line with a cursor key, the changes are not entered into program memory even though they are registered in screen memory. The contents of screen memory last only as long as the information appears on the screen.

When you are typing in a program, you can save time by taking advantage of the computer's screen editing features. If you are typing a line that is similar to one already on the screen, you can cursor to that line, change the line number and

anything else in the line, and then press RETURN. The computer accepts the line with the new number and retains the old line, too, as long as you remember to change the line number. If you forget to change the line number, the modified version of the line will replace the original version.

The program area of memory is not the same as screen memory, so even though the old line has been overwritten in screen memory, both lines can be intact in program memory. You can verify that both copies of the line are in program memory by LISTing the program. For example, type this line and press RETURN:

```
10 INPUT"WHAT'S YOUR NAME";N$
```

Now cursor back to the line number. Change it to 20, cursor to YOUR NAME and change it to THE DATE, delete the E from NAME, change N\$ to D\$, and press RETURN. The screen should look like this:

```
20 INPUT"WHAT'S THE DATE";D$
```

Line 10 no longer appears on the screen, but it is still in program memory. Issue a LIST command to display the program, which should look like this:

```
10 INPUT"WHAT'S YOUR NAME";N$  
20 INPUT"WHAT'S THE DATE";D$
```

Remember to press RETURN after you make changes to a line (if you want to keep the changes, that is). It is a good idea to LIST a program after you make changes so that you can verify that the changes were made in program memory. Even experienced programmers sometimes forget to press RETURN.

## Quote Mode

When you type a quotation mark, quote mode is turned on and everything you type is subject to quote-mode rules. Quote mode is turned off when you type a second quotation mark, when you press the RETURN key, or when you issue an ESCAPE O sequence.

The following rules define quote mode:

1. The computer does not interpret any characters typed inside quotes, so you can type anything in quotes without getting a SYNTAX ERROR message when you execute the command. All non-BASIC characters can and must be enclosed in quotes except information appearing in REM or DATA statements.
2. Commands do not execute in quote mode.

3. Many key functions, such as cursor-control and color-change keys, do not execute immediately. When you press one of these keys, a special reversed-image symbol is displayed. These symbols stand for keyboard-controlled functions that do not execute until the quote mode is turned off and the command is run.

## Insert Mode

When you press the INSERT key, you enter insert mode. While you are in insert mode, some key functions, such as cursor control and color change keys, do not execute immediately. When you press one of these keys, a special reversed-image symbol is displayed. These symbols stand for keyboard-controlled functions that do not execute in insert mode. They will not execute at all unless they are also in quote mode. In fact, you end up with a syntax error if you leave one of these function symbols in a command, so be sure to delete them unless they are in quotes and you want them to remain.

Insert mode ends when you type as many characters as the number of times you pressed the INSERT key, when you press RETURN, or when you issue the ESCAPE O sequence. If you want to get rid of the inserted spaces without pressing RETURN or typing characters, press the DELETE key until the inserted spaces are deleted. The DELETE key does not actually start deleting right away. A reversed T is printed in the inserted spaces until they are all filled.

Note that these deferred restrictions do NOT apply when you are in automatic insert mode, which you enter by pressing ESCape and A and cancel with ESCape C.

Table 3-1 shows the keys that do not execute in quote mode. This table also shows the one-character symbols that represent these keys.

You can directly embed the symbols for most of these functions in quote mode by pressing the indicated keys. The exceptions are the REVERSEd codes. REVERSEd H, I, and N can be entered in quote mode by pressing CONTROL H, I, and N. For REVERSEd SHIFTed N and M, leave a blank space in the quote mode text string where you want one of these functions to appear. Then exit quote mode and cursor back to the blank space in the text string. Once you are in position, turn on reverse mode (with CONTROL 9) and press the appropriate SHIFTed key. The symbol from the chart is displayed in the text string, and the function is deferred until you execute the command that contains the text string.

## Clearing the Screen During Program Execution

You can use the CLEAR key to clear the screen during program execution. To use this key, press SHIFT/CLEAR in quotes in a command such as PRINT or INPUT, or use the CHR\$ code for the CLEAR key, which is 147.

TABLE 3-1. Special Quote Mode and Insert Mode Characters

<i>Key</i>	<i>Displays</i>	<i>Function Embedded</i>
CURSOR UP	█	Move the cursor up a line.
CURSOR DOWN	█	Move the cursor down a line.
CURSOR LEFT	█	Move the cursor left a space.
CURSOR RIGHT	█	Move the cursor right a space.
INSERT	█	Prepare to insert a character.
DELETE	█	Delete a character.
CLEAR	█	Clear the screen.
HOME	█	Send the cursor to upper left.
CONTROL BLACK	█	Make character color black.
CONTROL WHITE	█	Make character color white.
CONTROL RED	£	Make character color red.
CONTROL CYAN	█	Make character color cyan.
CONTROL PURPLE	█	Make character color purple.
CONTROL GREEN	█	Make character color green.
CONTROL BLUE	█	Make character color blue.
CONTROL YELLOW	█	Make character color yellow.
█ ORANGE	█	Make character color orange.
█ BROWN	█	Make character color brown.
█ YELLOW-GREEN	█	Make character color yellow-green.
█ PINK	█	Make character color pink.
█ BLUE-GREEN	█	Make character color blue-green.
█ LIGHT BLUE	█	Make character color light blue.
█ DARK BLUE	█	Make character color dark blue.
█ LIGHT GREEN	█	Make character color light green.
CONTROL RVS ON	R	Turn on reversed mode.
CONTROL RVS OFF	█	Turn off reversed mode.
CONTROL FLASH ON	█	Turn on flashing.
CONTROL FLASH OFF	█	Turn off flashing.
REVERSED H	H	Disable █ SHIFT keys.
REVERSED I	I	Enable █ SHIFT keys.
REVERSED SHIFT M	█	Disabled RETURN character.
REVERSED N	N	Switch to upper/lower case.
REVERSED SHIFT N	█	Switch to upper case/graphics.

*Note:* The uppercase characters shown boxed above appear reversed on your screen.

Example: 10 INPUT "PROJECT NAME";P\$  
 20 PRINT CHR\$(147); "A REPORT ON ";P\$

Instead of using the CHR\$ code for the CLEAR key, you can type the opening quote, press the CLEAR key (with SHIFT), and then type the rest of the PRINT message. The CLEAR key appears in quotes as a reversed heart.

20 PRINT "  A REPORT ON ";P\$

You can use the SCNCLR command in a program line to clear the screen during program execution. SCNCLR has no parameters.

Example: 10 INPUT "PROJECT NAME";P\$  
 15 SCNCLR  
 20 PRINT "A REPORT ON ";P\$

## Clearing Graphic Mode Screens

To clear a graphic mode screen while you are in a graphic mode, use SCNCLR. When you issue SCNCLR in a graphic mode, only the graphic mode screen is cleared. When you issue a SCNCLR command in a text mode, the text screen is cleared, but the graphic mode screen is not cleared. When you issue a SCNCLR in a split-screen graphics mode, both the text and graphics mode screens are cleared.

If you are just issuing the GRAPHIC command to enter a graphic mode, clear the graphic mode screen by adding a ,1 to the end of the GRAPHIC command (e.g., GRAPHIC 2,1).

You can clear the text screen in a split-screen graphic mode screen by pressing the CLEAR key (with SHIFT), but this is not the best way to clear this area. When you press CLEAR, the cursor goes to the cursor-home position—the very top of the screen, which is not visible in split-screen mode. You have to cursor back manually to the text window at the bottom of the screen. It is much simpler just to scroll the text out of the text window by pressing the cursor down key five times. If you do a lot of work in a split-screen mode, you could set a screen window consisting of the bottom five (text) lines of the screen. Then, a SCNCLR command or the CLEAR key would clear only the window and leave the cursor at the top left of the text area of the screen.

## Using the Escape Key Functions to Control the Screen

The Plus/4 has 17 ESCape functions that you can use to edit or otherwise control the screen. These functions, which are explained in subsequent sections, include the following types of operations:

- Scrolling controls
- Cursor controls
- Deletion and insertion operations
- Screen-size reduction
- Screen windowing

The ESCape key functions are described briefly in Table 3-2.

ESCAPE functions are a key sequence of the ESCape key and one other key. To use any of the ESCape functions, press the ESCape key, release it, and then press the other key. Be sure to release the ESCape key before you press the second key.

Four ESCape functions have continuous operation after you turn them on: automatic insert mode, screen window settings, scrolling on and off, and screen display-size reduction. The other escape functions execute only once; to repeat them, you must repeat the ESCape key sequence. The four continuous ESCape functions have their own cancel-function sequences. The one-time-only ESCape functions cannot actually be canceled because they occur only once, and their

TABLE 3-2. ESCape Key Functions

<i>Second Key</i>	<i>Escape Function</i>
A	Turn on automatic insert mode.
B	Set screen window bottom right corner.
C	Cancel ESCape A, automatic insert mode.
D	Delete the current line.
I	Insert a blank line.
J	Move the cursor to the beginning of the current line.
K	Move the cursor to the end of the current line.
L	Turn on normal scrolling.
M	Cancel normal scrolling.
N	Cancel ESCape R, so normal screen size is reset.
O	Cancel manual insert, quote, reverse, and flashing modes.
P	Erase all characters from the beginning of the current line to the current cursor position.
Q	Erase all characters from the current cursor position to the end of the current line.
R	Reduce normal screen display size.
T	Set screen window top left corner.
V	Scroll up one line.
W	Scroll down one line.

functions cannot be reversed by a cancellation (although in most cases there is another ESCape key function that has the opposite effect).

*Note:* If you press the ESCape key and then decide not to press the second key in a function sequence, the cursor is temporarily frozen until you press any one key. Obviously you should not press one of the 17 keys that activate an ESCape function. You also should not press a function key or the CONTROL or **C** keys because they have no effect. Press the space bar or one of the cursor keys to thaw the cursor and resume normal operations.

## Canceling Insert, Quote, Reversed-Image, and Flashing Modes

You can quickly cancel insert mode, quote mode, reversed-image mode, or flashing mode by issuing the key sequence ESCape O. Although each of these modes can be terminated by other means, ESCape O is a convenient alternative. ESCape O is especially useful for turning off quote mode when you need to make corrections inside quotes.

Note that ESCape O cancels insert mode only when you turn this mode on with the INSERT key. ESCape O does not cancel automatic insert mode, which is turned on with ESCape A and canceled with ESCape C.

## Cursor-Control ESCape Functions

You can quickly move the cursor to the beginning or the end of the current line by using the ESCape J and ESCape K functions. ESCape J moves the cursor to the first column of the current line. ESCape K moves the cursor to the last character (not the last column) in the current line. If the cursor is already on or past the last displayed character on the line, ESCape K has no effect on the cursor. ESCape J always moves the cursor to column 1 regardless of whether or not there is a character displayed in that column.

If you use ESCape J or K while the cursor is on a BASIC command that takes up two lines, ESCape J moves the cursor to the first column of the first line even if the cursor is somewhere on the second line. ESCape K moves the cursor to the last character of the second line regardless of whether the cursor is on the first or second line of the two-line command.

## Scrolling-Control ESCape Functions

Under normal operating conditions, the screen display scrolls continuously. You can turn off normal scrolling so that the cursor stops scrolling when it reaches the current bottom line on the screen. To turn off scrolling, issue the ESCape M sequence. When scrolling is off, the cursor returns to the top of the screen (in the

same column) when you cursor past the bottom line. Text on the screen does not move.

To turn normal scrolling back on, issue the ESCape L sequence.

You can make the screen display scroll up or down one line at a time with the ESCape V and W sequences. ESCape V moves the screen display up one line; ESCape W moves it down one line. The cursor can be located anywhere on the screen; it does not have to be at the top or the bottom line. ESCape W always displays a blank line at the top of the screen as text lines are moved down a line at a time. ESCape V always displays a blank line at the bottom of the screen as text lines are moved up a line at a time.

Both ESCape W and ESCape V functions work when scrolling is turned off by ESCape M. In fact, ESCape V is the only way to scroll the top line off the screen when normal scrolling is turned off.

## Screen Editing

Previous sections in this chapter on screen editing have demonstrated some of the techniques you can use to correct errors, save typing time, clear the screen (or just delete some lines from it), and change the screen colors. Besides the features described in these sections, you can also use the Escape functions to edit the screen.

1. Inserting and deleting characters and lines on the screen.
2. Changing the screen size.

Insertion/deletion and changing screen size are especially useful when you are working on programs.

### Automatic Insert Mode

The INSERT key lets you add as many characters as the number of times you press the INSERT key. We might call this method of character insertion a manual insert mode. Automatic insert mode, when engaged, lets you insert as many characters as you like. Automatic insert mode saves time when you want to insert more than one or two characters.

To turn on automatic insert mode, press the ESCape key and then the A key. From then until you cancel the mode, all the characters you type are in insert mode; regular insert mode rules are in effect. The cursor can be located anywhere on the screen when you turn on automatic insert mode.

When you issue a RUN command while you are still in automatic insert mode and there is information on the screen beneath the line where you issued RUN,

the output of the execution of the program is inserted above the old data. The old data are pushed ahead of the execution output.

### Canceling Automatic Insert Mode

Cancel automatic insert mode by pressing the ESCape key and then the C key. As soon as you issue this key sequence, normal conditions return. Note that ESCape O cancels manual insert mode but not automatic insert mode.

### Deleting the Current Line

The ESCape D function lets you erase a line on the screen. To use ESCape D, move the cursor to the line you want to erase; then press the ESCape key and then the D key. The current line is deleted, and any following lines are moved up one line to fill in the gap. You can erase additional lines by repeating the key sequence. If the line is part of a BASIC line that is longer than one screen line, the entire BASIC line is deleted from the screen display.

The ESCape D function edits only what is on the screen, not what is in the program area of memory. This means that lines deleted by the ESCape D sequence are erased from the screen but not from the program. If you LIST the program after you use ESCape D to delete a line, you will see that the line is still in the program. You can use the DELETE command to remove a line from a program.

### Inserting a Blank Line

The ESCape I function lets you insert a blank line between two lines anywhere on the screen. To use ESCape I, move the cursor to the line where you want to insert a blank line; then press the ESCape key and then the I key. A blank line is inserted, and any following lines are moved down one line to make room for the new line. You can add additional blank lines by repeating the key sequence.

### Erasing Partial Lines

ESCAPE P and ESCAPE Q erase partial lines. ESCAPE P erases all characters that precede the cursor and the character under the cursor. ESCAPE Q erases all characters that follow the cursor and the character under the cursor. The cursor does not move when you execute either of these sequences.

If you are using ESCape P or ESCape Q to erase part of a program line and

that line is longer than 40 characters (i.e., it extends onto the next screen line), the entire program line is affected, not just the current screen line.

If you press RETURN after you erase a partial line, the part of the line you erased is deleted from program memory as well as from the screen. If you do not press RETURN before you leave the line, the original line remains intact in program memory. If the part of the line remaining would create a syntax error if it were executed, be sure to correct the line before you run the program. In many cases after you issue an ESCape P, the remaining characters will create a syntax error because you have erased the BASIC keyword.

**Example:** 10 PRINT "THE SECRETARY WILL DISAVOW AN  
Y KNOWLEDGE OF YOUR ACTIVITIES"

If the cursor is on the T in SECRETARY and you issue an ESCape P, the line will look like this (the cursor remains where the T was):

ARY WILL DISAVOW AN  
Y KNOWLEDGE OF YOUR ACTIVITIES"

If the cursor is on the T in SECRETARY and you issue an ESCape Q, the line will look like this (the cursor remains where the T was):

10 PRINT "THE SECRE

## Reducing the Screen Display Size

You can slightly reduce the size of the screen display with ESCape R. The normal screen is 40 columns by 25 lines. The reduced screen is 38 columns by 23 columns. The size is not optional; if you want to reduce the screen display further, you must create a screen window. The purpose of the function is to accommodate certain TV sets that cannot fully display the entire 40-by-25 screen.

As soon as you switch between normal and reduced screen display, the screen is cleared and the cursor is displayed at the cursor-home position at the top left corner of the screen. The cursor-home position in reduced screen display mode is at the column 2, line 2 position of normal screen display mode.

To cancel the reduced screen and return to the normal-sized screen, issue an ESCape N sequence. The screen clears and the cursor returns to the normal cursor-home position. Note that ESCape N cancels screen windows. Pressing the HOME key twice also cancels the reduced screen mode.

*Note:* The ESCape R sequence changes only the logical size of the screen. A bit on the graphics chip can be used to bring the border in over the unused row and columns. See Chapter 4.

## Setting a Screen Window

You can create a screen window of any size and in any part of the screen. When you set a screen window, all new text appears in the window work area. The rest of the screen contents remain unaffected, so you can view other material while you use the window work area. If you set the window over any characters already on the screen, the old characters are typed over.

To set a screen window do the following:

1. Move the cursor to the line and column you want to be the top left corner of the screen window.
2. Press the ESCAPE key and then the T key.
3. Move the cursor to the line and column you want to be the bottom right corner of the window.
4. Press the ESCAPE key and then the B key.

After the screen window is set, all text is displayed in the window. The rest of the screen remains as it was when you set the window. Screen windows are particularly useful for debugging programs and for working out parts of a program.

## Releasing a Screen Window

To return to full screen display, press the HOME key twice. The cursor appears in the cursor-home position of a full screen (a reduced screen is forgotten). The screen is not cleared when you create or release a window.

## Issuing an ESCape Function in a Program

There are several ways to include an ESCape function in a BASIC program. For example, you can type an ESCape sequence in response to an INPUT command. You can use the CHR\$ code for the ESCape key (27), and then the GETKEY command to input the second key in the sequence:

```
80 GETKEY A$  
90 PRINTCHR$(27)+A$
```

You can include specific ESCape sequences by using the CHR\$ codes for both keys. In the following example, CHR\$(65) stands for A, which turns on automatic insert mode, and CHR\$(87) stands for W, which scrolls down one line,

thereby moving the cursor up one line. Line 200 goes back to the message printed by line 120 and alters the message if X is less than zero after line 130.

```
100 INPUT X
120 PRINT"ACCEPTABLE CONDITIONS"
130 X=X-4
140 IF X >0 THEN 100
160 X$=CHR$(27)+CHR$(65)
180 Y$=CHR$(27)+CHR$(87)
200 PRINT Y$+X$+"WARNING! UN";
RUN
? 3
WARNING! UNACCEPTABLE CONDITIONS
```

Note that if you set a screen window, turn on automatic insert mode, turn off scrolling, or reduce the screen display size during a program, these continuous ESCape functions will remain in effect after the program has finished running.

## Using Text Strings

A text string can contain up to 255 characters, including blank spaces. Any character can appear within a literal constant text string except for a quotation mark. A quotation mark delineates the opening or closing of a literal constant text string and cannot appear as a character within a string. If you want to include a quotation mark within a text string, you must use the CHR\$ value for the quotation mark, which is 34.

Literal constant text strings must be typed in quotation marks except when they appear as constants in a DATA list. Quotation marks are optional for strings in a DATA list unless they contain colons or commas.

### Text-String Variables

A text-string variable can represent any text string. Text-string variables have a \$ sign as the final character in the variable name (e.g., X\$, W2\$).

When you assign a text string to a variable, you must use a text-string variable. A TYPE MISMATCH error occurs if you assign a text-string constant to a different type of variable or you assign a number to a text-string variable. If a number is included in a text string, the number is considered to be part of the text string, has no mathematical value, and cannot be used in any mathematical operation.

An empty text-string variable is called a null string. It has length zero and can

be created with a literal string constant consisting of only a pair of quotation marks (with nothing between them).

Input from keys on the keyboard is considered to be a text string, so you must use a text-string variable to read a key. For example, you must use a string variable in the GETKEY command to tell the computer to accept a single pressed key as input.

*Note:* It is possible to use a numeric variable with a GETKEY command. Pressing a digit (0–9) results in that single digit's value being assigned to the numeric variable. Pressing any other key results in an error. The error aborts the program unless it is TRAPPED.

## Combining String Values

You can use the plus sign to concatenate multiple text strings, including string variables and string functions. When you are combining text strings, you must type the plus sign outside the quotes (e.g., "DATA" + "BASE").

Concatenated strings are compressed into one value. No spaces are added between two strings (e.g., "DATA" + "BASE" equals "DATABASE"). The strings are concatenated from left to right.

Use the plus sign in compound function key definitions. For example, to define a function key to list and then run a program, you can use the following definition:

```
KEY 3, "LIST" + CHR$(13) + "RUN" + CHR$(13)
```

The plus sign is the only arithmetic operator you can use with text strings. You cannot use the minus sign to remove characters from a string; instead, use the RIGHTS\$, LEFT\$, and MID\$ functions to get substrings.

## Comparing String Values

You can use text strings in comparisons just as you use numbers. All six comparison operators (=, <>, >, <, <=, and >=) can be used to compare text strings.

You cannot compare a text string or text-string variable with a numeric value. If this illegal comparison is attempted, a TYPE MISMATCH error aborts the program unless it is TRAPPED.

When you use the equal or not equal signs to compare text strings, the computer reads the strings character by character, checking for an exact match, including blank spaces. For example, "STRING" <> "STRING" is true because the second string contains a blank space that is not present in the first string.

When you use the other comparison operators to compare text strings, the computer reads the strings character by character, checking for which string's current character is greater or less based on the character's character code number (CHR\$ value). A has the lowest value, Z the highest, so characters are checked for standard alphabetical order, although the comparison is actually done by numeric values for each letter of the alphabet. Shifted characters are always greater than unshifted characters. See the CHR\$ value list in Appendix C.

When numbers in text strings are compared, the computer treats them as a character (CHR\$) code number. All numbers have lower character codes than any letter, so 9 is less than A.

#### Examples of Text String Comparisons

```
10 INPUT T
20 A$ = "BIT" + STR$(T)
30 PRINT "A$ ="; A$
RUN
? 2
A$ = BIT 2
```

Makes A\$ equal to the string "BIT" plus the value of T converted to a string value by the STR\$ function.

NEW

```
10 DO:INPUT K$
20 PRINT "GO":LOOP WHILE K$ <> "STOP"
```

Compares the value input for K\$ to the string "STOP". The loop continues while K\$ is not equal to "STOP".

NEW

```
10 INPUT X$
20 IF LEFT$(X$,1) < "L" THEN
    GOSUB 100: ELSE PRINT "PAST RANGE"
RUN
? MIDDLE
PAST RANGE
```

Checks the first character of X\$, input in line 10. If the character is less than L (A-K), the program branches to a subroutine. Otherwise the PAST RANGE is printed.

## Using the String Functions

BASIC Version 3.5 contains 14 functions that operate using text strings:

- ASC returns the character (CHR\$) code of the first character in the string.
- CHR\$ returns the character string represented by its character code.

- DEC returns the decimal value of a hexadecimal string.
- DS\$ returns the contents of the disk drive error channel (see Appendix A).
- ERR\$ returns a message that describes an error condition.
- HEX\$ returns a hexadecimal text string for a decimal value.
- INSTR finds a string embedded within another string and returns its start position.
- LEFT\$ returns the leftmost characters in a string.
- LEN returns the number of characters in a string.
- MID\$ returns a character string within another string or replaces a substring in a character string.
- RIGHT\$ returns the rightmost characters in a string.
- STR\$ converts numeric values into text strings.
- TI\$ returns the current value of the system clock.
- VAL converts numbers in a string into a numeric value.

Each of these functions is described in Chapter 1. The following sections of this chapter provide additional information about some of the functions. More information about ERR\$ appears under the section about error-trapping techniques.

When any of these functions contains a text string as a parameter in parentheses, you can use any of the following forms of text-string representation:

- A literal constant text string
- A text-string variable
- A text-string array variable
- A character code in the form CHR\$(code)
- A concatenation (using +) of any of the above

Any numeric parameter in a string function can be in any of these forms:

- A number
- A numeric variable
- A numeric array variable
- A mathematical formula

## The Substring Functions: LEFT\$, RIGHT\$, and MID\$

The functions LEFT\$, RIGHT\$, and MID\$ can be used to form a substring of a text string. Use these functions to check input or to assign part of a string to a string variable. These functions are frequently used in conditional statements. For example, you can use LEFT\$ to check the first letter of the input.

Each of these functions can return up to 255 characters, which is the maximum length of a string. If the substring is longer than the master string, the computer returns the entire master string (starting from the function starting point).

The parameters for each function are enclosed in parentheses. The first parameter for each of these functions is the master string, which can be any legal string. The second parameter in the LEFT\$ and RIGHT\$ functions is the length of the substring. The second MID\$ parameter is the starting location for the substring. The third MID\$ parameter is the substring length.

The LEFT\$ substring begins at the leftmost character in the master string and continues for the specified number of characters. The RIGHT\$ substring has the specified length and ends at the rightmost character in the master string. The MID\$ substring can begin at any character position in the master string and continues for the specified number of characters.

Examples: 10 Y\$ = "REDWHITEBLUE"

20 X\$ = MID\$(Y\$,4,5)

X\$ is the word WHITE.

PRINT LEFT\$("RED",4)

RED

The string contains only three characters, so only three are printed.

10 INPUT "DO YOU WANT TO CONTINUE"; A\$ Searches text string A\$,

20 IF LEFT\$(A\$,1)="Y" THEN GOSUB 70:

input in line 10, for the string Y.

ELSE END

## Finding the Length of a String: the LEN(X\$) Function

LEN counts the total number of characters in a text string. Any text string can be counted. Blank spaces and punctuation marks count as characters in the string. For example "HO HO!" has six characters. You can use any string expression, even a text-string array element, as the LEN parameter.

The following example uses array elements as string function parameters:

```
5 DIM F$(49),L$(49)
10 FOR X = 0 TO 49
20 INPUT "NEXT NAME";L$(X)
30 FE=INSTR(L$(X)," ")
```

```

40 IF FE>0 THEN F$(X)=LEFT$(L$(X),FE-1);
   L$(X)=RIGHT$(L$(X),LEN(L$(X))-FE)
50 PRINT "LAST NAME: ";L$(X);;
   IF LEN(F$(X))>0 THEN PRINT " FIRST NAME: ";F$(X);
60 PRINT:NEXT

```

LEN can be used to figure a parameter in a LEFT\$, RIGHT\$, or MID\$ function. For example, in the following program, the starting position in the MID\$ function in line 30 uses the LENGTH of S\$:

```

10 INPUT "MONTH/DAY/YEAR"; S$
30 PRINT"THIS IS DAY ";MID$(S$,LEN(S$)-4,2);" OF MONTH ";
   LEFT$(S$,2)
RUN
MONTH/DAY/YEAR? 12/25/84
THIS IS DAY 25 OF MONTH 12

```

You can also use the string functions to search input for characters that are not acceptable for your application. For example, suppose an error will occur in your program if information being input contains numbers. You can use the string functions to search each piece of input for a number.

```

10 INPUT "FULL NAME"; S$
20 FOR I=48 TO 57
30 Y = INSTR(S$,CHR$(I))
40 IF Y>0 THEN PRINT "ILLEGAL INPUT; CHARACTER";Y;
   "IS A NUMBER"
50 NEXT

```

The INSTR (IN STRing) function finds the starting position of a string embedded within another string. INSTR searches a text string from left to right and returns a number that tells you the character position of the first character in the sought string.

Like the LEN function, INSTR is a numeric function, which means it returns a number, not a text string. Also like LEN, INSTR works only on a text string expression, not on a number.

## Converting Strings and Numeric Values: STR\$ and VAL

The STR\$ function converts numeric values into text strings. You would use this conversion when you want to use a string function to search the number. There are no numeric functions comparable to LEFT\$, RIGHT\$, MID\$, and LEN. So

when you want to use such a function on a number, first use STR\$ to convert the number to a text string.

Example: `T$ = STR$(T): IF RIGHT$(T$,3)  
= ".99" THEN T = T + .01`

The STR\$ function converts the numeric value of T into a string called T\$. The RIGHT\$ function reads the three rightmost characters of T\$. If these characters equal .99, .01 is added to the value of T.

After you convert a number to a text string, the number loses its numeric properties, which means you cannot use the number in a calculation. Instead, the computer treats the stringified number the same way it treats any text string. After you finish using the number as a text string, you can change it back to a numeric value by using the VAL function.

The VAL function converts numbers in a string into a numeric value. You can use VAL to reverse a STR\$ function or to extract numbers from any text-string expression.

When you use VAL on a text string that contains both numbers and non-numeric characters, the computer converts only the numbers up to the first nonnumeric character. For example, if you issue the command PRINT VAL("34R5"), the computer displays only 34; the 5 after the R is not displayed because the first nonnumeric character turns off the VAL function.

## Redefining the Function Keys

You can redefine a function key at any time in immediate mode or within a program. Any definitions you write are erased from computer memory when you turn off or reset the computer (unless you hold down the RUN/STOP key during the reset). The KEY command lets you write a definition for a function key. The KEY command also displays the current definitions of the function keys; all redefinitions written during the current computing session are displayed in this list.

Follow these steps to redefine a function key:

**STEP 1** Type KEY and the key number (to define the HELP key, type an 8) followed by a comma.

**STEP 2** Type the text string for the key definition:

You can define the key to perform multiple tasks in BASIC. Link multiple BASIC commands and/or functions with plus signs.

If a literal constant string is used for a key definition, it must be in quotes.

Use CHR\$ codes in the definition to use quotation marks or a key such as a return or the ESCape key.

STEP 3 Verify the new definition by issuing a KEY command, which displays a list of current key definitions.

Examples: KEY 1,"INPUT"+CHR\$(34)

KEY 6,"LIST"+CHR\$(13)+"RUN"+CHR\$(13)

Displays INPUT".

Issues a LIST command and a RUN command.

RUN executes as soon as the program is listed.

Creates a screen window, whose top left corner is the current cursor location and whose bottom is the lower right corner of the screen. After the window is set, a LIST command is issued.

KEY 3,CHR\$(27)+"TLIST"+CHR\$(13)

## Calling a Function Key During Program Execution

The function key definition procedure for defining keys in immediate mode can also be used in a program. INPUT can be used to accept function key definitions. Of course, the input must end with a RETURN character from the definition or the keyboard. GETKEY receives only the first letter of the definition. Also, if GETKEY is called a second time following the receipt of a multiple character function key definition, an error results.

To be able to use a function key in a GETKEY command, you must first redefine the key as a single CHR\$ code. This definition allows BASIC to consider the function key to be a single key, not a string of characters. Once the key is defined as a single key, you can press the key as input for a GETKEY command. Then you can use an IF command to see if the key pressed equals the CHR\$ code for the function key and use a THEN clause to perform the desired operation(s). The following example redefines function keys 2 and 3 to change screen colors, switch to graphic mode 1, and draw a painted shape.

Note that redefinitions written in a program are still in effect when the program ends. To restore the original definitions, press the reset button.

```
10 REM DEFINE KEYS 2 AND 3 AS CHR$ CODES 134 AND 135
20 KEY2,CHR$(134): KEY3,CHR$(135)
30 PRINT "PRESS F2 TO DRAW THE MOON. PRESS F3 TO DRAW
THE SUN."
```

```

40 GETKEYZ$: REM PRESS F2 OR F3
50 REM USE ASC TO CHECK THE CHR$ CODE FOR THE PRESSED
   KEY
60 IFASC(Z$)=134 THEN
   COLOR0,1:COLOR1,2:GRAPHIC1,1:CIRCLE,160,100,60,50:
   PAINT,160,100
80 IFASC(Z$)=135 THEN
   COLOR0,7:COLOR1,8:GRAPHIC1,1:CIRCLE,160,100,60,50:
   PAINT,160,100

```

## Changing the Function Key Definitions in Machine Language

The function key definitions are stored in RAM and can be altered in machine language. The lengths of each function key definition are stored in \$055F - \$0566. The definitions themselves (in CHR\$ codes) are stored in \$0567-\$05E6. To change a definition, not only must the length for the key be changed and its definition be altered but the data for all of the function keys beyond it must be moved up or down to meet the new definition. The keys are stored in the following order:

<i>Key</i>	<i>Length Address</i>
F1	\$055F
F2	\$0560
F3	\$0561
F4	\$0562
F5	\$0563
F6	\$0564
F7	\$0565
HELP	\$0566

When a function key is pressed, the SCNKEY routine (called by the system interrupt service routine) places this information in memory separate from the normal keyboard queue. The information is processed by keyboard read routines BASIN (\$FFCF) and GETIN (\$FFE4).

This example changes the definition of the HELP key to whatever the user types in. The HELP key is the easiest to change because no other definitions are affected.

- Example:
- . 2000 A9 0D      LDA #\$0D      Carriage return character.
  - . 2002 20 D2 FF    JSR \$FFD2      Send to screen.
  - . 2005 A2 00      LDX #\$00      .X points to the definition area.
  - . 2007 20 CF FF    JSR \$FFCF      Get a character from the keyboard.

- . 200A 9D 9F 05 STA \$059F,X Store in the HELP key's definition area.
- . 200D C9 0D CMP #\$0D Look for a carriage return.
- . 200F F0 07 BEQ \$2018 When found, quit.
- . 2011 E8 INX Increment the pointer.
- . 2012 E0 48 CPX #\$48 Compare to the maximum allowed.
- . 2014 90 F1 BCC \$2007 If not there yet, go on.
- . 2016 B0 01 BCS \$2019 If there, quit.
- . 2018 E8 INX Increment the pointer to get a count.
- . 2019 8E 66 05 STX \$0566 Store count in HELP key length.
- . 201C 00 BRK Stop processing.

*Note:* There is an “unofficial” ROM subroutine that redefines a function key. Store the key to redefine (0 to 7) in \$76, the address of the new definition is \$22-\$23, load .A with the length of the definition and call the subroutine at \$FF49.

## Mathematical Calculations

This section briefly discusses a few important concepts for using your computer for calculating.

**Number Storage** In BASIC there are two numeric variable types. The more straightforward is the integer variable (signified by attaching a % to the variable name). An integer variable can have values from -32767 to +32767. Theoretically, a value of -32768 is allowed. Try the following example program:

```
10 N%=32768
20 PRINT N%
```

The result will be a -32768. So much for theory.

The second variable type is floating point. The format of floating point number storage is examined in the section on USR in Chapter 5. The largest magnitude of a floating point number is 1.70141183E+38, and the smallest magnitude distinguishable from zero is 2.93873588E-39. The floating point format allows about nine decimal digits of accuracy.

**Speeding Up Calculations** The first rule to speed calculations is to do as few as possible. In particular, unless variable space is at a premium, do not calculate the same quantity twice. Calculate it once, and save the value in a variable. The exponentiation function (up arrow) is slow, so avoid it if possible. In particular,

square a number by multiplying it by itself, get a square root by using the SQR function, and calculate a reciprocal by dividing 1 by the number.

It is somewhat faster to add a number to itself than to multiply by 2. It is somewhat faster to divide by 2 than to multiply by 0.5. Subtraction and addition take virtually the same amount of time.

**Logarithms and Exponentials** LOG(X) returns the logarithm base  $e$  (the natural logarithm) of X. The logarithm with respect to a different base, B, can be found by dividing LOG(X) by LOG(B). The inverse of the logarithm base  $e$  ( $e$  raised to a power) can be calculated with EXP. The inverse of the logarithm base B can be calculated by using the up arrow to raise B to the power.

**Trigonometric Calculations** An approximation of pi is available by using the pi key (the  $\text{C}$  and equal keys pressed together). This is particularly useful when the values of trigonometric ratios are desired and the angle is measured in degrees. The SIN, COS, and TAN functions are available, but each requires the specified angle to be measured in radians. To translate from degrees to radians, multiply the angle by pi, and divide by 180.

To calculate the values of the remaining trigonometric ratios, recall that

$$\csc(X) = 1/\text{SIN}(X)$$

$$\sec(X) = 1/\text{COS}(X)$$

$$\cot(X) = 1/\text{TAN}(X)$$

The only inverse trigonometric ratio available is the arctangent (ATN). The value is returned in radians. To change to degrees, multiply by 180, and divide by pi. To calculate the values of the remaining inverse functions, recall that

$$\arcsin(A) = \text{ATN}(A/\text{SQR}(1-A^2))$$

$$\arccos(A) = \text{ATN}(\text{SQR}(1-A^2)/A)$$

$$\text{arccsc}(A) = \text{ATN}(1/(A\text{SQR}(1-1/(A^2))))$$

$$\text{arcsec}(A) = \text{ATN}(A\text{SQR}(1-1/(A^2)))$$

$$\text{arccot}(A) = \text{ATN}(1/A)$$

**Rounding Off Numbers** When asked to display a number, the PRINT (or PRINT#) command prints nine digits of precision. If this is not desired, the PRINT USING command may be used. Or, recall that the number X rounded to N decimal places is given by

$$\text{INT}(10^N * X + .5)/10^N$$

**Random Numbers** BASIC has a built-in random number function that returns a floating point number between 0 and 1. When called with a negative argument, the RND function reseeds the random number generator with the value specified and returns the first value from this seed. When called with a zero argument, the RND function reseeds the random number generator from a hardware clock and returns the first value from this seed. When called with a positive argument, the next value in the random sequence is returned. To have a different random number sequence every time the program is run, call RND at zero first, then at 1 (or any positive number) thereafter. To have the same sequence each time, call RND at a constant negative seed value first, then at 1 (or any positive number) thereafter.

When a random floating point number between L and H is needed, use

$$(H-L)*RND(1)+L$$

When a random integer between L% and H% (inclusive) is needed, use

$$INT((H\%-L\%+1)*RND(1)+L\%)$$

## Programming Sound and Music

The computer has two voices that can play music and a voice setting that can create noise. Only two commands, VOL and SOUND, are required to play music and sound effects.

### The VOL Command

The VOL command sets the volume level for tones played by the SOUND command. The volume level can be from 0 (off) to 8 (highest volume). You must be sure that the volume selector on your TV or monitor is turned up. The level of volume set by the VOL command is relative to other VOL settings. The absolute sound level is set by the volume selector on your TV or monitor.

The volume level remains at the last level set at the end of a program. If you do not turn the volume off at the end of the program or reset its value, the next SOUND command plays at the volume last set, even if the next SOUND command is in a different program.

You must execute a VOL command before the SOUND command, or you will not be able to hear any sound.

### The SOUND Command

The SOUND command selects the tone to be played. The SOUND command has three parameters:

## SOUND voice, tone frequency, duration

1. Voice selects the voice in which the tone is to be played. There are two voices and three voice settings (the second voice has an alternative setting):

Voice setting 1 plays 1024 tones.

Voice setting 2 plays the same 1024 sounds.

Voice setting 3 plays 1024 settings of noise.

You can play sounds from voices 1 and 2 simultaneously. Because voice setting 3 is an alternative setting for voice 2, you cannot simultaneously play sounds with voice settings 2 and 3.

2. Tone frequency selects the frequency of the sound to be played. This setting can be from 0 to 1023. Table 3-3 lists the numerical values for five octaves of musical notes. Other values play tones but not musical notes. Noise can be played in voice setting 3.

Note that, although the tones go from low to high, tone value 1023 is the lowest note and 1022 is the highest. Tone value 0 is virtually as low as 1023. You probably cannot hear a tone value between 1016 and 1022.

TABLE 3-3. Numerical Values for Five Octaves of Notes

Note	Octave 1 Frequency	Octave 2 Frequency	Octave 3 Frequency	Octave 4 Frequency	Octave 5 Frequency
A	7	516	770	897	960
A#	64	544	784	904	964
B	118	571	798	911	967
C	169	596*	810	917	971
C#	217	620	822	923	974
D	262	643	834	929	976
D#	305	664	844	934	979
E	345	685	854	939	982
F	383	704	864	944	984
F#	419	722	873	948	986
G	453	739	881	953	988
G#	485	755	889	957	990

\*596 is the setting for middle C.

*Note:* Use the following formula to calculate a frequency value for some other output frequency (FO):

$$\text{frequency} = 1024 - \text{INT}(111860.781/\text{FO})$$

The lowest possible frequency is about 109 Hz and the highest is above the audible range (over 20 KHz).

3. Duration selects how long the note is played. The values in the duration position can be from 1, which equals 1/60th of a second, to 65535, which is more than 18 minutes. You can also use decimal numbers, variables, or calculations as duration values but only their integer part will be used.

Use this formula to figure duration value:

$$\text{Duration value} = \text{time in seconds} \times 60$$

When you are programming a song, the durations of all notes are relative to the whole-note duration you choose. After you select the duration value for the whole note, other note duration values are determined by fractions of the whole-note value.

In this program, voices 1 and 2 are used simultaneously to play two-voice harmony. Although a frequency value of zero does generate a sound, this program uses a value of zero to mean a rest (no sound).

**Example:**

```

10 VOL8
20 DIMN1%(66),N2%(66),D1%(66),D2%(66)
30 I=0
40 READN1%(I),D1%(I):IFN1%(I)<0THEN60
50 I=I+1:GOTO40
60 T1=I:I=0
70 READN2%(I),D2%(I):IFN2%(I)<0THEN90
80 I=I+1:GOTO70
90 I1=-1:I2=-1
100 IFD1>0THEN130:ELSE SOUND1,N1,0
110 I1=I1+1:IFI1<T1THEN D1%(I1):N1=N1%(I1):ELSE180
120 IFN1>0THEN SOUND1,N1,300
130 IFD2>0THEN160:ELSE SOUND2,N2,0
140 I2=I2+1:D2=D2%(I2):N2=N2%(I2)
150 IFN2>0THEN SOUND2,N2,300
160 D1=D1-1:D2=D2-1
170 FOR I=0 TO 80:NEXT:GOTO100
180 VOL0
190 DATA 0,1,685,1,770,1,810,1
200 DATA 798,1,685,1,798,1,834,1
210 DATA 810,2,854,2,755,2,854,2
220 DATA 770,1,685,1,770,1,810,1
230 DATA 798,1,685,1,798,1,834,1
240 DATA 810,2,770,2,0,4
250 DATA 0,1,854,1,810,1,854,1
260 DATA 770,1,810,1,685,1,739,1
270 DATA 704,2,770,2,834,2,864,2
280 DATA 864,1,834,1,798,1,834,1
290 DATA 739,1,798,1,643,1,704,1
300 DATA 685,2,739,2,810,2,854,2
310 DATA 854,1,810,1,770,1,810,1
320 DATA 704,2,834,2,834,1,798,1
330 DATA 739,1,798,1,685,2,810,2

```

```
340 DATA810,1,770,1,704,1,770,1
350 DATA643,2,798,2,810,6
360 DATA-1,-1
370 DATA7,2,516,4,485,2
380 DATA516,1,345,1,516,1,596,1
390 DATA571,1,345,1,571,1,643,1
400 DATA596,2,516,2,485,2,345,2
410 DATA516,1,345,1,516,1,596,1
420 DATA571,1,345,1,571,1,643,1
430 DATA596,2,516,2,596,2,516,2
440 DATA643,1,516,1,383,1,516,1
450 DATA262,1,383,1,7,1,169,1
460 DATA118,2,262,2,453,2,571,2
470 DATA571,1,453,1,345,1,453,1
480 DATA169,1,345,1,118,1,118,1
490 DATA7,2,169,2,262,1,383,1
500 DATA118,1,262,1,118,2,118,2
510 DATA169,1,345,1,7,1,169,1
520 DATA7,2,7,2
530 DATA118,1,453,1,383,1,453,1
540 DATA596,6
550 DATA-1,-1
```

### Line-by-Line Explanation

- 10 Set volume to maximum.
- 20 Prepare data arrays for notes and durations.
- 30 I counts the number of notes.
- 40 Read a note and duration for voice 1; a negative number means done.
- 50 Increment counter and continue.
- 60 T1 is the total number of notes for voice 1. Start I over again at 0.
- 70 Read a note and duration for voice 2; a negative number means done.
- 80 Increment counter and continue.
- 90 I1 and I2 are pointers to the data arrays.
- 100 If voice 1 is not finished, go on to line 130. Otherwise, stop voice 1.
- 110 Increment voice 1 pointer. If done, quit. Otherwise, set up note and duration.
- 120 If not a rest, start the note.
- 130 If voice 2 is not finished, go on to line 160. Otherwise, stop voice 2.
- 140 Increment voice 2 pointer and set up note and duration.
- 150 If not a rest, start the note.

- 160 Decrement the durations.
- 170 Wait briefly. Change the value here to make all the notes longer or shorter.
- 180 Turn off the volume.
- 190–350 Data for voice 1.
- 360 End of data for voice 1.
- 370–540 Data for voice 2.
- 550 End of data for voice 2.

## Sound in Machine Language

In machine language, sound is generated by accessing the graphics chip (which also handles sound) directly. The relevant registers are as follows:

	<i>Bit(s)</i>	<i>Function</i>
\$FF0E	0–7	Low byte of frequency for voice 1
\$FF0F	0–7	Low byte of frequency for voice 2
\$FF10	0–1	High 2 bits of frequency for voice 2
\$FF11	0–3	Volume
	4	Select voice 1 (0 = off, 1 = on)
	5	Select voice 2 (0 = off, 1 = on)
	6	Select noise for voice 2 (0 = off, 1 = on)
	7	Sound switch (0 = on, 1 = off)
\$FF12	0–1	High 2 bits of frequency for voice 1
	2–7	Nonsound uses

To generate a sound, first select the voice or voices to use and the volume level with register \$FF11. Normally, it is appropriate to set bit 7 to 1 at this time, to keep the sound silent for the moment. Note that voice 1 is on or off, but voice 2 can be on for tone, on for noise, or off. If bit 5 is set to 1, voice 2 generates tones, regardless of the setting of bit 6. Next, set the frequencies in the appropriate registers. Be careful when setting the two high bits of voice 1 to leave the remaining bits of register \$FF12 unchanged. To start the sound, clear bit 7 of register \$FF11. To stop the sound, set bit 7 of register \$FF11 (or deselect the voices or set the volume to zero).

Table 3–4 shows the hexadecimal note values.

The following program plays the first few notes of Scott Joplin's "The Entertainer." Up to 255 bytes of data are stored at \$2100 in the format, high byte of frequency, low byte of frequency, duration.

TABLE 3-4. Hexadecimal Musical Notes

Note	Octave 1	Octave 2	Octave 3	Octave 4	Octave 5
A	7	204	302	381	3C0
A#	40	220	310	388	3C4
B	76	23B	31E	38F	3C7
C	A9	254	32A	395	3CB
C#	D9	26C	336	39B	3CE
D	106	283	342	3A1	3D0
D#	131	298	346	3A6	3D3
E	159	2AD	356	3AB	3D6
F	17F	2C0	360	3B0	3D8
F#	1A3	2D2	369	3B4	3DA
G	1C5	2E3	371	3B9	3DC
G#	1E5	2F3	379	3BD	3DE

- Example:
- . 2000 A9 9F LDA #\$9F Select voice 1 with maximum volume.
  - . 2002 8D 11 FF STA \$FF11 Store in sound selection register.
  - . 2005 A2 00 LDX #\$00 .X points to the data.
  - . 2007 BD 01 21 LDA \$2101,X Get the low byte of the frequency.
  - . 200A 8D 0E FF STA \$FF0E Store in low byte of frequency for voice 1.
  - . 200D AD 12 FF LDA \$FF12 Get high byte register.
  - . 2010 29 FC AND #\$FC Mask off low two bits.
  - . 2012 1D 00 21 ORA \$2100,X OR in the high bits of frequency for voice 1.
  - . 2015 8D 12 FF STA \$FF12 Store in high byte of frequency for voice 1.
  - . 2018 AD 11 FF LDA \$FF11 Get sound selection register.
  - . 201B 29 7F AND #\$7F Turn on sound.
  - . 201D 8D 11 FF STA \$FF11 Store sound selection register.
  - . 2020 BD 02 21 LDA \$2102,X Get duration of note.
  - . 2023 85 D8 STA \$D8 Store in temporary variable.
  - . 2025 A5 A5 LDA \$A5 Get low byte of clock.
  - . 2027 29 02 AND #\$02 Look at bit 1.
  - . 2029 F0 FA BEQ \$2025 Wait until it is set.
  - . 202B A5 A5 LDA \$A5 Get low byte of clock.
  - . 202D 29 02 AND #\$02 Look at bit 1.
  - . 202F D0 FA BNE \$202B Wait until it is clear.
  - . 2031 C6 D8 DEC \$D8 Decrement duration.
  - . 2033 D0 F0 BNE \$2025 If not done, wait again.
  - . 2035 AD 11 FF LDA \$FF11 Get sound selection register.
  - . 2038 09 80 ORA #\$80 Turn off sound.
  - . 203A 8D 11 FF STA \$FF11 Store sound selection register.
  - . 203D A5 A5 LDA \$A5 Get low byte of clock.
  - . 203F 29 02 AND #\$02 Look at bit 1.
  - . 2041 F0 FA BEQ \$203D Wait until it is set.
  - . 2043 E8 INX Increment .X by three to point at next note.
  - . 2044 E8 INX

- . 2045 E8 INX
- . 2046 E0 33 CPX #\$33 Compare to position following last datum.
- . 2048 90 BD BCC \$2007 If not done, continue.
- . 204A 00 BRK Stop processing.

Here is some example data:

```
>2100 03 42 02 03 4C 02 03 56
>2108 02 03 95 04 03 56 02 03
>2110 95 04 03 56 04 03 95 0C
>2118 03 A1 02 03 A6 02 03 AB
>2120 02 03 95 02 03 A1 04 03
>2128 AB 02 03 95 02 03 A1 04
>2130 03 95 0C 00 00 00 00 00
```

## Using Arrays to Handle Groups of Data

An array, which is also called a matrix, is a set of related values. A two-dimensional array is organized into numbered “rows” and “columns.” The name of the array and the number of elements the array can contain are established in an array-DIMensioning command. After an array is defined in a DIM command, you can use values in the array as individual data items. You refer to any element of a two-dimensional array by giving the array variable name and the row and column number in the array where the element is located. This row-and-column address is called a subscript of the array.

An array can have one, two, or more dimensions. A one-dimensional array has only a row of data. A two-dimensional array has rows and columns of data. Arrays with more than two dimensions have more complex data configurations. You need not be able to visualize multidimensional arrays to work with them effectively.

You should always DIMension an array before you use it. If you use an array element without first DIMensioning the array, the computer gives the array the default number of elements (11). You cannot change the dimensions of an array after you have DIMensioned it or after you have accepted the default dimensions. If you DIM the array after you have used it, or try to reDIM the array, the program aborts and the error message REDIM'D ARRAY is displayed.

The array name is a variable that follows standard variable rules. Arrays containing text elements must have text-string variable names. Arrays containing numeric elements must have a numeric variable name.

The subscripts set the number of rows and columns in a two-dimensional array. Rows are listed first. If you are using a one-dimensional array, there is no column number.

The first element in an array is numbered 0, not 1. This means that an array dimensioned as (5,3) actually has 6 rows and 4 columns, or 24 elements. When you figure the number of elements in an array, add 1 to each dimension, then multiply the results of the additions. For example, if the array is dimensioned DIM K(2,4), the array contains  $(2 + 1) * (4 + 1) = 15$  elements.

The following example uses a two-dimensional array to input and print data. The printed data in line 80 shows how array elements are assigned in a two-dimensional array.

```

10 DIMA(2,3)
20 FORY=0TO2
30 FORX=0TO3
40 INPUT"NEXT ELEMENT";A(Y,X)
50 NEXTX,Y
60 FORY=0TO2
70 FORX=0TO3
80 PRINT"ROW";Y;"COLUMN";X;"EQUALS";A(Y,X)
90 NEXT X,Y

```

The following example uses a one-dimensional array and a two-dimensional array to keep track of data. The program creates a 4-by-4 letter grid like the one used in the game Boggle. Array B contains numbers for the current 16 letters to be used in the grid. Array N\$ contains the 50 letters that can become part of the grid.

```

10 DIMB(3,3):DIMN$(49)
15 DATA A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z
20 DATA A,E,I,O,U,A,E,I,U,O,A,E,I,O,R,S,T,N,M,R,S,T,N,M
25 REM ASSIGN NUMBERS TO LETTERS
30 FOR Y=0TO49
40 READ N$(Y)
50 NEXTY
52 X = RND(0):REM SEED RANDOM NUMBER GENERATOR
53 PRINT
55 REM RANDOMLY ASSIGN NUMBERS TO SQUARES
60 FOR X=0TO3
70 FOR W=0TO3
80 B(X,W)=INT(RND(1)*50)
100 PRINTN$(B(X,W));
110 NEXTW:PRINT:NEXT X

```

The FOR . . . NEXT loop in lines 30–50 assigns a letter value to the elements of array N\$ so that array element N\$(0) equals A, N\$(1) equals B, and so on. We use 50 elements instead of just 26, so we can assign common letters to more than

one number, which increases the chances that the common letters are chosen as 1 of the 16 letters in the grid.

Lines 60–110 randomly generate 16 numbers between 0 and 49. In line 100 the chosen element of array N\$ is printed. The nested loops are used to force the computer to print four letters next to each other on a single row. The PRINT in line 110 forces the computer to print the next four letters on a new line.

The next example uses two one-dimensional arrays to sort a list of names into alphabetical order. Array N\$ contains the names that are to be sorted. Array K is the key file that contains the numbers used to produce and print a sorted list of names. This sorting technique, which is called a bubble sort, uses the key file to expedite the sorting procedure. The key file is sorted as the names are compared; the name file itself is not actually sorted. As soon as the loop in lines 180–260 determines which of the two current names is first, the next name is retrieved from the N\$ array. Elements of array K are sorted in line 240 when name N\$(K(Y)) is greater than the next name, N\$(K(Y+1)).

Line 300 prints the unsorted array N\$ and the sorted names, which are accessed in the sorted order by using the sorted key file, array K.

```

5 DATA JOE,JAY,ANNE,JIM,DAN,CATHERINE,SARAH,GLENN,GERRY,TERRY,BRUCE,JOHN
10 DATA BOB,LIB,OLIVER,STELLA,MARCIA,DAVE,MARY LOU,CATHY,JACK,END
15 REM COUNT HOW MANY NAMES ARE TO BE SORTED
20 READN$:IFN$="END"THEN 60
40 N=N+1
50 GOTO 20
60 RESTORE
70 REM DIMENSION ARRAYS WITH THE NUMBER OF NAMES TO BE SORTED
71 REM N$ HOLDS NAMES; K IS A KEY FILE THAT ASSOCIATES A NUMBER WITH A
NAME
80 DIMN$(N),K(N)
90 REM READ NAMES FROM THE DATA LIST
100 FORX=1TON
120 READN$(X)
130 REM ASSIGN ARRAY N$ ELEMENT NUMBER TO KEY FILE
140 K(X)=X
160 NEXTX
180 FORX=NTO1STEP-1
200 FORY=1TOX-1
210 REM SEE WHICH OF TWO NAMES COMES FIRST ALPHABETICALLY
220 IFN$(K(Y))<=N$(K(Y+1))THEN 260:REM THEN GO GET NEXT NAME
230 REM IF THE 1ST NAME IS > THE 2ND, ELEMENTS IN KEY FILE ARE REVERSED
235 REM THIS IS KEY TO SORT: ONLY KEY FILE IS SORTED;
ITS SORT IS BASED ON VALUES TAKEN FROM ARRAY N$
240 T=K(Y):K(Y)=K(Y+1):K(Y+1)=T
260 NEXTY,X
270 REM PRINT UNSORTED AND SORTED LISTS
280 PRINTCHR$(18)"UNSORTED",TAB(10)"SORTED"
290 REM N$(X) PRINTS UNSORTED NAMES; KEY FILE IS USED TO PRINT SORTED
NAMES
300 FORX=1TON:PRINTN$(X),TAB(10)N$(K(X)):NEXT

```

## UnNEWing Programs

If you issue a NEW command and immediately regret it, you can try to restore the program. If you or one of your programs executed a GRAPHIC command during the current session and no GRAPHICCLR command has been executed since, use \$4000 instead of \$1000 in the following steps:

1. Enter the monitor by issuing the MONITOR command.
2. Examine memory starting at \$1000 by typing M 1000.
3. Scan along the resulting memory dump to find the end of your first program line. If you are lucky, you can recognize it from the text you used. If not, check the token list in Appendix B. The end of the line is marked with a zero byte (but this may not be the first zero byte you encounter). Note the address of the location following the zero byte (the first location of the next line). Cursor to the location \$1001 in the dump and enter the low byte of the address there, and the high byte in \$1002 (and press RETURN).
4. Continue looking at the memory dump (issue additional M commands if needed) until you find the end of your program. It is marked by three zero bytes. Note the address of the location following the three zero bytes (the first unused location). Display locations \$2D and \$2E by typing M 2D 2D. Enter the low byte of the address in location \$2D and the high byte in \$2E (and press RETURN).
5. Exit the monitor by typing X. Your program should be back.

**Example:** Suppose you had typed in the following program (and not yet saved it to tape or disk).

```
10 PRINT "I AM A LITTLE KITTY"  
20 PRINT "MY NAME IS NICKY"  
30 DO WHILE A$=""  
40 PRINT "MEOW"  
50 GETA$  
60 LOOP
```

Then you typed:

NEW

Here is what would happen when you unNEWed the program:

MONITOR

MONITOR

```

PC  SR  AC  XR  YR  SP
;  0000 00 00 00 00 F8
M 1000
>1000 00 00 00 0A 00 99 22 49 : ....."I"
>1008 20 41 4D 20 41 20 4C 49 : AM A LI
>1010 54 54 4C 45 20 4B 49 54 : TTLE KIT
>1018 54 59 22 00 34 10 14 00 : TY".4...
>1020 99 22 4D 59 20 4E 41 4D : ."MY NAM
>1028 45 20 49 53 20 4E 49 43 : E IS NIC
>1030 4B 59 22 00 42 10 1E 00 : KY".b...
>1038 EB 20 FD 20 41 24 B2 22 : A$2"
>1040 22 00 4E 10 28 00 99 22 : ".n.(..
>1048 4D 45 4F 57 22 00 56 10 : MEOW".v.
>1050 32 00 A1 41 24 00 5C 10 : 2.IA$. .
>1058 3C 00 EC 00 00 00 00 00 : <.....

```

Note that the first line ends at \$101A and that the location following the zero is \$101C. So cursor up to the first line of the memory dump and change \$1001 and \$1002. Now the first line of the dump is:

```
>1000 00 1C 10 0A 00 99 22 49 : ....."I"
```

Note that the last line ends at \$105A (the token for LOOP is \$EC) and that the location following the three zeroes is \$105E.

```
M 2D 2D
>002D 03 10 03 10 03 10 00 FD : .....
```

Change \$2D and \$2E so that the dump is

```

>002D 5E 10 03 10 03 10 00 FD : .....
X
READY.
LIST
10 PRINT"I AM A LITTLE KITTY"
20 PRINT"MY NAME IS NICKY"
30 DO WHILE A$=""
40 PRINT"MEOW"
50 GETA$
60 LOOP

```

READY.

Your program is restored.

## Using the Built-In Error-Trapping Routine

The TRAP command lets you prevent a program from being aborted because of any BASIC error condition except an UNDEFINED STATEMENT error. TRAP catches the error and branches to the line number named as the TRAP parameter. At this line number, you can write any sort of error-handling and/or error-reporting routine. You should use RESUME to resume execution after completing your error routine. Using a GOTO to leave a trap routine causes BASIC to think it is still in the TRAP routine (unless it receives another TRAP command). Because errors inside TRAP routines cannot be trapped, it is best never to leave a trap routine through a GOTO. RESUME reexecutes the line in which the error occurred. RESUME NEXT resumes executing at the statement following the error. RESUME *linenumber* resumes execution at the specified line number. Error trapping can be turned off by using a TRAP command with no line number. Error trapping is also turned off by a CLR command.

TRAP does not trap disk drive errors (read from the disk error channel) or other errors not generated by BASIC. For a list of all the BASIC errors. See Appendix A.

**Example:** In this program, a DIVISION BY ZERO error can occur if 0 is input as a value for X. The error is trapped, and the line number (EL), error number (ER), and error message (ERR\$(ER)) are printed before execution resumes with the INPUT statement. The program prints out 10 valid results. An input resulting in an error is not counted because execution resumes at 30.

```
10 TRAP60
20 FOR I=1 TO 10
30 INPUT Y,X
40 PRINT Y/X
50 NEXT:END
60 PRINT "LINE";EL;"ERROR";ER;ERR$(ER)
70 PRINT X:RESUME 30
```

### Line-by-Line Explanation

- 10 Turn on error trapping with trap routine starting at line 60.
- 20 Count 10 valid results.
- 30 Get values for X and Y.
- 40 Print Y divided by X.
- 50 Go on to next result, and quit when done.
- 60 Print out line number, error number, and error message,
- 70 Print out the value of X and resume program execution at 30.

---

# 4 Programming Graphics

---

The graphics statements built into BASIC on the Plus/4 make graphics treatment in BASIC almost entirely different from that of machine language. No detailed knowledge of the graphics chip itself is required to do sophisticated high-resolution and multicolor graphics on the Plus/4. But, of course, most machine language programmers will need to control the chip's functions directly. This chapter is therefore divided into two sections: the first primarily for the BASIC programmer and the second primarily for the machine language programmer.

## **Graphics Programming in BASIC**

All the BASIC statements used in this chapter are detailed in Chapter 1. When you need additional information on the use and parameters of a BASIC statement, refer to that chapter.

### **Color and Luminance**

Your Commodore Plus/4 is capable of producing 16 different colors. Each of these colors may be modified into eight shades (the eight shades of black are indistinguishable). This means that the Plus/4 can produce 121 different colors. In BASIC there are five sources for the colors on the screen. The color for each of these sources is chosen with the COLOR statement.

**COLOR *source,color,luminance***

Depending on which graphic mode the Plus/4 is in, different color sources are available for text and graphics. The uses of color in each mode are explained in this chapter. The value for *color* chooses the color:

<i>Value</i>	<i>Color</i>	<i>Value</i>	<i>Color</i>
1	black	9	orange
2	white	10	brown
3	red	11	yellow-green
4	cyan	12	pink
5	purple	13	blue-green
6	green	14	light blue
7	blue	15	dark blue
8	yellow	16	light green

The value of luminance is optional and can be 0 (dark) through 7 (light). The COLOR statement uses a luminance of 7 if you do not specify a value.

The following example program displays all of the colors of your Plus/4 on the screen.

### Color and Luminance Example Program

```

10 COLOR0,2,6
20 GRAPHIC1,1
30 X=0
40 FORC=1TO16
50 Y=0
60 FORL=0TO7
70 COLOR1,C,L
80 BOX1,X,Y,X+16,Y+16,,1
90 Y=Y+16
100 NEXTL
110 X=X+20
120 NEXTC
130 COLOR1,1,0:CHAR1,16,20,"HIT KEY"
140 GETKEYK$:GRAPHICO

```

### Line-by-Line Explanation

- 10 First change the background color to a light gray.
- 20 Put the Plus/4 into high-resolution graphics mode.
- 30 Start the x-coordinate of the display at zero.
- 40 Go through the 16 available colors.

- 50 Start the y-coordinate of each column at zero.
- 60 Go through the eight available shades for each color.
- 70 Set the foreground color to the current color and shade.
- 80 Draw a box with the foreground color and fill it with color.
- 90 Adjust the y-coordinate for the next box.
- 100 Use the next luminance value.
- 110 Adjust the x-coordinate for the next column.
- 120 Use the next color value.
- 130 Change the foreground color to black and write "HIT KEY" on the screen.
- 140 Wait for a keypress and return to text mode.

## Text Mode

The normal mode of the Plus/4's operation is text mode. In text mode, you can PRINT alphanumeric and graphic characters onto the Plus/4 screen. When you need to return to text mode after using another graphic mode, use

### **GRAPHIC 0,*clear flag***

The 0 tells the Plus/4 to return to text mode. The *clear flag* is an optional parameter. If it is 1, the text screen is automatically cleared. If it is not present or is 0, any text previously placed on the text screen remains there. You can also return to text mode by using the GRAPHICCLR command. This command also frees the 12K reserved for the graphics screen for use by your program.

In text mode there are three possible color sources. The background and border of the screen are colored with

### **COLOR 0,*color,luminance***

for the background, and

### **COLOR 4,*color,luminance***

for the border. The color of the characters themselves can be controlled with

### **COLOR 1,*color,luminance***

Characters can be placed on the screen with the PRINT (and PRINT USING) and CHAR statements. Each character has its own foreground color. That means you can PRINT every character in a different color. Using the COLOR statement to change colors that often is cumbersome, so BASIC has color control characters you can include in a text string to change the color. There are only 16 of these control characters, so you can choose only 16 colors by this method. Their default values are shown in Table 4-1. The color and luminance assigned to each color key are normally determined by the contents of 275–290 (\$0113–\$0122). These locations can be altered, thereby altering the meanings of the color keys. Also, the Plus/4 can be instructed to get the values of the color keys from ROM by setting the high bit of 2041 (\$07F9) to 1, in which case Table 4-1 applies.

If you press the color keys in quote mode (see Chapter 3), a graphics character is included in the string you create. The color changes when the string is printed. If you press the keys at any other time, the color of the characters you type from then on is changed. If you include the CHR\$ code of the color in a string you print, the color changes when the string is printed.

Other control characters can be included in text strings.

TABLE 4-1. Color Keys

<i>CHR\$</i>	<i>Keys</i>	<i>Color</i>	<i>Luminance</i>	
144	Control 1	1	0	Black
5	Control 2	2	7	White
28	Control 3	3	3	Red
159	Control 4	4	6	Cyan
156	Control 5	5	4	Purple
30	Control 6	6	3	Dark green
31	Control 7	7	4	Blue
158	Control 8	8	7	Yellow
129	█ 1	9	4	Orange
149	█ 2	10	2	Dark brown
150	█ 3	11	5	Yellow-green
151	█ 4	12	6	Pink
152	█ 5	13	5	Blue-green
153	█ 6	14	6	Light blue
154	█ 7	15	2	Dark blue
155	█ 8	16	5	Light green

<i>CHR\$</i>	<i>Keys</i>	<i>Function</i>
18	CONTROL 9	Turns on reverse printing.
146	CONTROL 0	Turns off reverse printing.
130	CONTROL ,	Turns on flashing.
132	CONTROL .	Turns off flashing.
8	CONTROL H	Disables <b>█</b> SHIFT.
9	CONTROL I	Enables <b>█</b> SHIFT.
14	CONTROL N	Switches (whole screen) to upper/lower case.
142	Reverse SHIFT N	Switches (whole screen) to upper case/graphics.
17	Cursor Down	Moves down one line.
145	Cursor Up	Moves up one line.
29	Cursor Right	Moves one space to the right.
157	Cursor Left	Moves one space to the left.
19	HOME	Moves to top left of screen.
147	SHIFT HOME	Clears screen and moves to top left of screen.

Again, if you press the designated keys in the quote mode, these functions occur when the string is printed. The reverse SHIFT N cannot be entered directly into a string. You must type in the string leaving blanks where it is to go. Then, when you are out of the quote mode, cursor back to those blanks, get into reverse printing (Control 9) and type in the SHIFT N. You can also include the CHR\$ code to perform the function when the string is printed.

The PRINT (or PRINT USING) statement may be used to place text on the screen wherever the cursor is. When PRINT or PRINT USING is used, you can control the position on the screen by placing the cursor where you want the text to be. Generally, you must start by printing a CLEAR/HOME character (CHR\$(147)), so that the cursor always begins in the upper left corner of the screen, and then using cursor control characters to reach the desired position.

The CHAR statement may also be used to place text on the screen. You may still want to clear the screen using CLEAR/HOME, but you do not need to be concerned with cursor positioning. The CHAR statement requires designating an x-coordinate and y-coordinate for the placement of the text. The following example program illustrates the use of some of these capabilities.

### Text Mode Example Program

```

10 COLOR0,5,5:COLOR4,5,4:COLOR1,7,3
20 PRINT"♥[QQQQQQQQ"
30 PRINTSPC(14)"INTRODUCING"
40 PRINTSPC(14)"YOUR "CHR$(130)"NEW"
50 COLOR1,16,1

```

```
60 PRINTSPC(14)"COMMODORE"
70 PRINTSPC(14)"COMPUTER"
80 CHAR,12,7,"█[R] "
90 FORY=8TO13
100 CHAR,12,Y," [ ] " "
110 NEXT
120 CHAR,12,14," "
130 PRINT" [QQQ] "
140 PRINTCHR$(144)SPC(15)"HIT KEY"
150 GETKEYK$:COLOR0,2,7:COLOR4,15,6
```

### Notes

- 20 In quotes is a reversed heart followed by eight reversed upper case Q's. These are printed with the SHIFT CLEAR/HOME and cursor-down keys, respectively.
- 80 In quotes is a reversed upper-right-corner graphic (CHR\$(127)) followed by a reversed upper-case R and 15 spaces. These are obtained with CONTROL CYAN, CONTROL REVERSE ON, and the space bar, respectively.
- 100 In quotes is a space followed by 13 reversed right square brackets, and one more space. The reversed right square bracket is printed with the cursor-right key.
- 130 In quotes are three reversed upper case Q's, which are printed with the cursor-down key.

### Line-by-Line Explanation

- 10 Change the background color to light purple, the border to darker purple, and the character color to blue.
- 20 Clear the screen and cursor down near the middle.
- 30 Space over and print INTRODUCING.
- 40 On the next line space over and print YOUR and a flashing NEW.
- 50 Change the character color to green.
- 60 Space over and print COMMODORE.
- 70 On the next line space over and print COMPUTER.
- 80 On the eighth line in the thirteenth column, print a color change to cyan, and 15 reversed spaces.
- 90 Do the next statement for lines 9 through 14.

- 100 In the thirteenth column, print a reversed space, 13 cursor rights, and another reversed space.
- 110 Do next line.
- 120 On the fifteenth line in the thirteenth column print 15 reversed spaces.
- 130 Print three cursor down commands.
- 140 Print a color change to black, space over, and print HIT KEY.
- 150 Wait for a key. Then change background and border colors to normal values.

## High-Resolution Mode

The graphics mode that provides maximum resolution on the Plus/4 is accessed with the command:

**GRAPHIC 1,*clear flag***

The *clear flag* is optional (1 clears the high-resolution screen, 0 or absence leaves the screen intact).

This statement creates a bit-mapped screen on which the programmer can use the Plus/4's graphics statements: BOX, CIRCLE, DRAW, and PAINT. A section of memory (12K bytes) is set aside for this use and is therefore not available for the BASIC program. The text screen area of memory is left intact. Therefore, you can PRINT on the text screen behind the graphics screen, and the PRINTed data are revealed when you return to text mode.

***The Split-Screen*** You can create the bit map and view the upper portion of it and the lower five lines of the text screen simultaneously with the command:

**GRAPHIC 2,*clear flag***

Again, both the text and the bit-map areas of memory are reserved. The use of this statement merely reveals the lower five lines of the text screen and conceals the corresponding portion of the bit map. When the clear flag is set to 1, both the graphics and text screens are cleared and the text cursor is placed at the left of the first visible text line.

***The Coordinate System*** The high-resolution screen is normally addressed with a 320 by 200 coordinate matrix. The horizontal (or x) coordinate ranges from 0 at

the left of the screen to 319 at the right. The vertical (or y) coordinate ranges from 0 at the top of the screen to 199 at the bottom. Only the CHAR statement (which plots characters on the high-resolution screen) uses the text rows and columns for its coordinates. The CHAR x and y are related to the high-resolution x and y by

$$\text{CHAR coordinate} = \text{INT}(high\text{-resolution coordinate}/8)$$

**Colors in High Resolution** The background and border colors of the screen can be changed as usual with the commands:

`COLOR 0,color,luminance`

and

`COLOR 4,color,luminance`

The other color available in high resolution is foreground. The foreground color is set with

`COLOR 1,color,luminance`

In each of the drawing statements, you can specify the use of foreground or background color.

Although you can choose to set (to foreground color) or clear (to background color) each pixel on the screen in this mode, the colors of some pixels are not independent. The background color is globally defined. Whenever you change COLOR 0, every background colored pixel changes color. Also, each 8-pixel-by-8-pixel character cell on the screen is assigned a single foreground color. This means that every pixel set to foreground color within a character cell is the same color. If you change foreground colors between two drawings to the same character cell, the color of pixels drawn both times will be the color used last. However, each of the 1000 character cells has an independent color (and luminance), allowing for a great deal of creative color usage, even in high-resolution mode.

Whenever you draw a pixel, whether in foreground or background color, the foreground color for its character cell is set to the current foreground color. This is normally the desired effect when you are drawing in foreground color. But when drawing in background color, one would expect the foreground colors to be unaffected. Unfortunately, this is not the case. If you have drawn in foreground color in a character cell, and you return to draw in background color, the foreground color of that cell is updated to your current foreground color. If you

have changed the foreground color since drawing in that cell, the foreground colored pixels will change color, even though you are drawing in background color.

## High-Resolution Colors Example Program

This example program illustrates the color limitations of high-resolution graphics mode. Note that the coordinates of the boxes in the first series are incremented by 10. The character boundaries are therefore frequently crossed and a “bleeding” of colors is observed. The coordinates of the boxes in the second series are incremented by eight (and start on a character boundary). Therefore, no character boundaries are crossed and the colors remain true.

```

10 GRAPHIC1,1
20 FORX=0TO190STEP10
30 COLOR1,((X/10)AND15)+1,5
40 BOX,X,X,X+16,X+10,,1
50 NEXT
60 FORX=0TO192STEP8
70 COLOR1,((X/8)AND15)+1,5
80 BOX,X+64,X,X+80,X+8,,1
90 NEXT
100 COLOR1,1,0:CHAR,5,20,"HIT KEY"
110 GETKEYK$:GRAPHICO

```

### Line-by-Line Explanation

- 10 Enter high-resolution graphics mode (and clear graphics screen).
- 20 First series of box coordinates are incremented by 10.
- 30 Choose a new foreground color for each box.
- 40 Draw a box 16 by 10 at the current coordinates with foreground color and fill it in.
- 50 Get the next set of coordinates.
- 60 Second series of box coordinates are incremented by 8.
- 70 Choose a new foreground color for each box.
- 80 Draw a box 16 by 8 at the current coordinates with foreground color and fill it in.
- 90 Get the next set of coordinates.
- 100 Change foreground color to black and plot "HIT KEY".
- 110 Wait for a key, then return to text mode.

## Multicolor Graphic Mode

Multicolor graphic mode is normally used when the color restrictions of high resolution are unacceptable. It is accessed with

### GRAPHIC 3,*clear flag*

The *clear flag* is optional (1 clears the multicolor graphic screen, 0 or absence leaves the screen intact).

This statement creates a multicolor bit-mapped screen on which the programmer can draw graphics with the Plus/4's graphics statements: BOX, CIRCLE, DRAW, and PAINT. A section of memory (12K bytes) is set aside for this use and is therefore not available for the BASIC program. The text screen area of memory is left intact. Therefore, you can PRINT on the text screen behind the graphics screen, and the PRINTed data are revealed when you return to text mode.

**The Split-Screen** You can create the multicolor bit map and view the upper portion of it and the lower five lines of the text screen simultaneously with the command:

### GRAPHIC 4,*clear flag*

Again, both the text and the multicolor bit map areas of memory are reserved. The use of this statement merely reveals the lower five lines of the text screen and conceals the corresponding portion of the multicolor bit map. When the *clear flag* is set to 1, both the graphics and text screen are cleared and the text cursor is placed at the left of the first visible text line.

**The Coordinate System** The multicolor graphic screen is normally addressed by a 160 by 200 coordinate matrix. The horizontal (or *x*) coordinate ranges from 0 at the left of the screen to 159 at the right. The vertical (or *y*) coordinate ranges from 0 at the top of the screen to 199 at the bottom. The *x* coordinates used on the multicolor graphic screen are exactly one-half of the corresponding *x* coordinates on the high-resolution screen. Only the CHAR statement (which plots characters on the multicolor graphic screen) uses the text rows and columns for its coordinates. The CHAR *x* and *y* are related to the multicolor graphic *x* and *y* by

$$\text{CHAR } y\text{-coordinate} = \text{INT}(\text{multicolor graphic } y\text{-coordinate}/8)$$

and

$$\text{CHAR } x\text{-coordinate} = \text{INT}(\text{multicolor graphic } x\text{-coordinate}/4)$$

By the way, characters plotted with CHAR in multicolor mode do not appear the same as in text mode or high resolution because pairs of bits (not single bits) in the character pattern determine the color.

**Colors in Multicolor Graphic Mode** The background and border colors of the screen can be changed as usual with the commands

**COLOR 0,color,luminance**

and

**COLOR 4,color,luminance**

Three other colors are available in multicolor mode. They are all nonbackground colors. The foreground color is set with

**COLOR 1,color,luminance**

Multicolor 1 is set with

**COLOR 2,color,luminance**

Multicolor 2 is set with

**COLOR 3,color,luminance**

With each of the drawing statements, you can specify which color source to use.

Color management in multicolor graphics is somewhat involved. First, the background and multicolor 2 are global colors. That is, whenever you change COLOR 0, everything drawn in background color changes color. Similarly, whenever you change COLOR 3, everything drawn in multicolor 2 changes color. Second, each 8-pixel-by-8-pixel (or 4 by 8, using multicolor coordinates) character cell has its own set of colors and luminances for foreground color and multicolor 1. This means that all the pixels within a character cell drawn in foreground color are the same color. Also, all the pixels within a character cell drawn in multicolor 1 are the same color. But the two colors can be different and background and multicolor 2 can be used at the same time, allowing up to four colors within a character cell. And each of the 1000 character cells has its own independent pair of foreground color and multicolor 1.

Whenever you draw, the foreground color and multicolor 1 for the character cell you are drawing in are updated to their current values. This means that if you have changed either of these colors since drawing with them in this character cell, the pixels you drew before will change color even if you are not now drawing with their color source.

## Multicolor Example Program

This program draws three sets of circles, one in each of the three nonbackground colors. The program demonstrates that these three colors are independent, but drawing over a color replaces it. At the end, the foreground color is changed to black to plot the characters in "HIT KEY". After the user hits a key, a line is drawn in background color through the circles. Even though the line is drawn in background color, the foreground color of the character cells through which it passes is changed to black, spoiling the circles drawn in foreground colors.

```
10 GRAPHIC3,1
20 COLOR1,2,6:COLOR2,5,5,:COLOR3,13,4
30 FORC=3TO1STEP-1
40 FORA=0TO180STEP10
50 CIRCLEC,40+C*20,100,10,50,,A,20
60 NEXTA,C
70 COLOR1,1,0:CHAR1,5,20,"HIT KEY"
80 GETKEYK$:DRAW0,0,100TO159,100
90 GETKEYK$:GRAPHICO
```

### Line-by-Line Explanation

- 10 Get into multicolor graphic mode.
- 20 Set up the foreground color, multicolor 1, and multicolor 2.
- 30 Draw in each of the three nonbackground colors.
- 40 Draw ovals at a series of angles.
- 50 Draw an oval.
- 60 Do the next angle and the next color.
- 70 Change the foreground color to black.
- 80 Wait for a key press. Then, draw a line in background color through the ovals.
- 90 Wait for a key press. Then, return to text mode.

## The Pixel Cursor and Relative Coordinates

Each of the BASIC drawing statements needs at least one set of coordinates. If you review those statements (BOX, CIRCLE, DRAW, PAINT, SSHAPE, and GSHPAE) in the BASIC language section, you will notice that in many cases the coordinates have a default value of the location of the pixel cursor. The pixel cursor is an invisible set of coordinates that BASIC keeps track of at all times.

You can change the pixel cursor location with the LOCATE statement, and BASIC may change the pixel cursor location when a drawing is done.

When you want to use the default value of the pixel cursor's coordinates in a drawing statement, but need to specify one of the later parameters, only one empty position must be left (NOT one for *x* and one for *y*). For example, the BOX statement:

**BOX color,x1,y1,x2,y2,angle,paint flag**

defaults the second set of coordinates to the pixel cursor. If you want to do this, but also want your box painted, the following can be used:

**BOX 1,10,10,,45,1**

This statement draws a filled-in box in foreground color between the absolute coordinates (10,10) and the pixel cursor at an angle of 45 degrees.

The pixel cursor may be used as a reference for relative coordinates. That is, you can specify coordinates relative to the pixel cursor's current coordinates. This is useful when you want to be able to execute the same series of drawing steps at various different locations on the screen.

Two types of relative coordinates are available. The first is rectangular relative coordinates. Instead of specifying an absolute *x* and *y* coordinate, you specify changes in *x* and *y* from the pixel cursor's current location. This is signaled by the use of a plus (+) sign for a positive change or a minus (-) sign for a negative change. For example, the statement

**CIRCLE 1,+10,-30,20**

draws a circle of radius 20 with its center 10 pixels to the right and 30 pixels above the pixel cursor's location.

The second type of relative coordinate is polar. Instead of specifying an absolute *x* and *y* coordinate, you specify a distance and an angle from the pixel cursor's current location. The distance is specified first, then separated from the angle (in degrees) by a semicolon. For example, the statement

**CIRCLE 1,15;45,25**

draws a circle of radius 25 with its center 15 pixels from the pixel cursor's location at an angle of 45 degrees.

## **Relative Coordinates Example Program**

```
10 GRAPHIC1,1
20 READX,Y,C,L:IFX<0THEN40
```

```
30 GOSUB140:GOTO20
40 COLOR1,1,0:CHAR,5,19,"HIT KEY"
50 GETKEYK$:GRAPHICO:END
60 DATA100,100,4,5
70 DATA50,60,3,4
80 DATA10,20,5,6
90 DATA120,40,6,5
100 DATA140,70,7,5
110 DATA200,110,8,6
120 DATA240,50,9,6
130 DATA-1,-1,-1,-1
140 COLOR1,C,L
150 LOCATEX,Y
160 FORA=0TO180STEP45
170 BOX,+30,+6,,A
180 NEXT
190 PAINT,15;100
200 RETURN
```

### Line-by-Line Explanation

- 10 Enter high-resolution mode and clear screen.
- 20 Read a set of coordinates, color, and luminance for a flower. If the last flower is done, go to 40.
- 30 Call flower-drawing subroutine at 140 and return to 20 for next flower.
- 40 Change the foreground color to black and plot "HIT KEY".
- 50 Wait for a key. Then return to text mode and stop the program.
- 60–130 Data for the flowers.
- 140 Change the foreground color for this flower.
- 150 Locate the pixel cursor at the *x* and *y* for this flower.
- 160 Draw boxes at angles from 0 to 180 degrees.
- 170 Draw a box with a corner 30 pixels below and 6 pixels to the right of the pixel cursor location and the other corner at the pixel cursor.
- 180 Do the next angle.
- 190 Fill in the area in the middle of the flower.
- 200 Return from subroutine.

## Custom Character Sets

The characters you see on your computer monitor or TV are formed by a pattern of dots. Each character occupies a cell eight dots (pixels) wide and eight dots high. Each dot is either on or off. Because each dot is individually controlled, the graphics chip is said to be in high-resolution mode. The pattern of dots that are on form the character you see. The patterns for the built-in characters are stored in your computer on a permanent memory chip (the character ROM).

Each dot in the character's pattern is represented by one bit in the character ROM. Since there are 8 bits in one byte (or memory location), it takes 8 bytes to specify each character's pattern. Following is a diagram illustrating the character C's  $8 \times 8$  cell.

BIT VALUES

128	64	32	16	8	4	2	1	
		X	X	X	X			byte 0
X	X				X	X		byte 1
X	X							byte 2
X	X							byte 3
X	X							byte 4
X	X				X	X		byte 5
	X	X	X	X	X			byte 6
								byte 7

The character C is represented by 8 bytes of data calculated by adding up the bit values of the bits it has on. For example, byte 0 is  $32 + 16 + 8 + 4 = 60$  (\$3C).

There are two sets of built-in characters. The first is upper case/graphics and the second is upper/lower case. You can switch between the two by pressing the SHIFT and **C** keys simultaneously.

When the built-in characters are not sufficient, you can set up a custom character set. That is, the graphics chip can be instructed to get its character patterns not from the character ROM, but from an area of user memory (RAM). A character set consists of 128 characters (8 bytes each) and resides in 1K (1024 bytes) of memory. (See the machine language section of this chapter for information on expanding the character set to 256 characters.)

Two locations control where the character patterns come from: 65298 (\$FF12), which controls whether the patterns come from ROM or RAM, and 65299 (\$FF13), which controls what locations in memory the patterns come from. When you switch between upper case/graphics and upper/lower case with the SHIFT and **C** keys, you are changing the value of 65299, which controls where the patterns come from. You can disable the **C** and SHIFT key combination by POKEing the value 128 into the memory location 1351, or by PRINTing a CHR\$(8). To reenable this feature, POKE a 0 into 1351, or PRINT a CHR\$(9).

**Copying the Standard Set** If you want to add your own special characters to the built-in characters and still use some of the standard characters, you need to copy the patterns stored in the character ROM to an area of user memory. This situation is somewhat complicated in BASIC. The PEEK function automatically reads RAM. This means that if you PEEK the area of memory containing the character ROM, you do not get the ROM patterns at all, but the contents of the RAM "underneath" the ROM. (See the banking section in Chapter 5.) You can defeat the BASIC switch to RAM with the following POKE:

**POKE 1177,62**

You can then copy the upper case/graphic ROM patterns to your RAM area (at *characters*) with

**FOR I = 0 TO 1023:POKE *characters*+I,PEEK(53248+I):NEXT I**

or copy the upper/lower case ROM patterns with

**FOR I = 0 TO 1023:POKE *characters*+I,PEEK(54272+I):NEXT I**

You must then restore BASIC's subroutine with

**POKE 1177,63**

Because using this method actually changes the operation of BASIC while its subroutine is altered, restore the switch to RAM before you perform any other operations.

**High-Resolution Characters** You can create custom character patterns in RAM and use them instead of the ROM characters in your program. First, you need to define your new characters. For each custom character use an  $8 \times 8$  grid similar to the one shown in the beginning of this section. The top row of the grid represents the top row of your character. Add up the bit values for the dots you want on in this row. This sum is the first of eight data values for your character. Do the same for each of the remaining rows.

Since your custom characters must appear in RAM, you must switch the graphics chip to look at RAM. This is accomplished by clearing bit 2 of 65298 (\$FF12), that is, ANDing its current value with 251.

**POKE65298,PEEK(65298)AND251**

To reselect ROM, you must set this bit, that is, OR its current value with 4.

**POKE65298,PEEK(65298)OR4**

You need to decide where in RAM your characters will be stored. They must be stored in an area of RAM not otherwise used by your program. (See Chapter 5 on moving BASIC and where BASIC programs reside.) Also, the graphics chip always considers a 1K (1024-byte) section of memory (room for 128 characters) to be its current character set. Therefore, your custom character set must begin on a 1K boundary. The boundary used is controlled by the upper 6 bits of 65299 (\$FF13). To specify the location of your character set, you can use the following BASIC line:

POKE65299,(PEEK(65299)AND3)ORx

where the value of  $x$  is determined from Table 4-2.

TABLE 4-2. Custom Character Set Locations

X Characters				X Characters				X Characters			
Hex	Dec	Hex	Decimal	Hex	Dec	Hex	Decimal	Hex	Dec	Hex	Decimal
\$00	0	\$0000	0	\$60	96	\$6000	24576	\$C0	192	\$C000	49152
\$04	4	\$0400	1024	\$64	100	\$6400	25600	\$C4	196	\$C400	50176
\$08	8	\$0800	2048	\$68	104	\$6800	26624	\$C8	200	\$C800	51200
\$0C	12	\$0C00	3072	\$6C	108	\$6C00	27648	\$CC	204	\$CC00	52224
\$10	16	\$1000	4096	\$70	112	\$7000	28672	\$D0	208	\$D000	53248
\$14	20	\$1400	5120	\$74	116	\$7400	29696	\$D4	212	\$D400	54272
\$18	24	\$1800	6144	\$78	120	\$7800	30720	\$D8	216	\$D800	55296
\$1C	28	\$1C00	7168	\$7C	124	\$7C00	31744	\$DC	220	\$DC00	56320
\$20	32	\$2000	8192	\$80	128	\$8000	32768	\$E0	224	\$E000	57344
\$24	36	\$2400	9216	\$84	130	\$8400	33792	\$E4	228	\$E400	58368
\$28	40	\$2800	10240	\$88	134	\$8800	34816	\$E8	232	\$E800	59392
\$2C	44	\$2C00	11264	\$8C	138	\$8C00	35840	\$EC	236	\$EC00	60416
\$30	48	\$3000	12288	\$90	142	\$9000	36864	\$F0	240	\$F000	61440
\$34	52	\$3400	13312	\$94	146	\$9400	37888	\$F4	244	\$F400	62464
\$38	56	\$3800	14336	\$98	150	\$9800	38912	\$F8	248	\$F800	63488
\$3C	60	\$3C00	15360	\$9C	154	\$9C00	39936	\$FC	252	\$FC00	64512
\$40	64	\$4000	16384	\$A0	160	\$A000	40960				
\$44	68	\$4400	17408	\$A4	164	\$A400	41984				
\$48	72	\$4800	18432	\$A8	168	\$A800	43008				
\$4C	76	\$4C00	19456	\$AC	172	\$AC00	44032				
\$50	80	\$5000	20480	\$B0	176	\$B000	45056				
\$54	84	\$5400	21504	\$B4	180	\$B400	46080				
\$58	88	\$5800	22528	\$B8	184	\$B800	47104				
\$5C	92	\$5C00	23552	\$BC	188	\$BC00	48128				

The value in the table for the character base address (*characters*) tells you where to copy the ROM patterns to if you want to use some of the standard characters. If you want to know where a specific character's patterns begin, take its screen code (see Appendix E), multiplied by 8, and add it to the character base address. To copy the pattern of a specific character in the character ROM, use the following:

```
FOR I=0 TO 7  
POKE characters+new code*8+I,PEEK(ROM+old code*8+I)  
NEXT I
```

where *characters* is the new character set base address, *new code* is the new screen code for the character, *ROM* is the old base address (53248 for an upper case/graphic character, or 54272 for an upper/lower case character), and *old code* is the original screen code for the character.

## High-Resolution Custom Characters Example Program

In this example, type the small letters inside the quotes without shifting, and type the capital letters inside the quotes shifted.

```
10 POKE1177,62  
20 FORI=0TO1023  
30 POKE8192+I,PEEK(54272+I)  
40 NEXT  
50 POKE1177,63  
60 POKE65299,(PEEK(65299)AND3)OR32  
70 POKE65298,PEEK(65298)AND251  
80 POKE1351,128  
90 FORI=0TO7:READA:POKE8192+I,A:NEXT  
100 PRINT"The Temperature is 25@C"  
110 PRINT"Hit a key to restore standard characters"  
120 GETKEYA$  
130 POKE65299,(PEEK(65299)AND3)OR208  
140 POKE65298,PEEK(65298)OR4  
150 POKE1351,0  
160 END  
170 DATA24,36,36,24,0,0,0,0
```

### Line-by-Line Explanation

10 The BASIC function PEEK is altered to look at ROM.

- 20 The entire 128 character set is copied ( $8 \times 128 = 1024$ ).
- 30 The upper/lower case characters are copied from ROM to RAM at 8192 (\$2000).
- 40 Get the next byte.
- 50 The BASIC function PEEK is restored to normal operation.
- 60 The graphic chip character base address is set to 8192 (\$2000).
- 70 The graphic chip reads character patterns from RAM.
- 80 The **C** and SHIFT key combination does not switch the character set base address.
- 90 A new custom character is inserted into the first character (@) position.
- 100 A message including the new character is printed to the screen.
- 110 A prompt instructs the user of the program.
- 120 The program waits for keyboard input.
- 130 The graphic chip character base address is restored to 53248 (\$D000).
- 140 The graphic chip reads character patterns from the character ROM.
- 150 The **C** and SHIFT key combination is reenabled.
- 160 Program execution ends.
- 170 The data for the custom character (a degree symbol).

**Multicolor Characters** One of the major reasons for creating a custom character set is to use more than a single foreground color and the background color in a character cell. In multicolor text mode, a pair of bits determine which of four colors a pair of pixels will be. The character pattern is organized in a parallel fashion with a high-resolution character, except that 2 bits together determine the color of the corresponding pair of pixels on the screen. Each byte in the character pattern determines the appearance of 8 pixels on the screen. The first 2 bits are the first 2 pixels, the second 2 the next 2, and so on.

The bit patterns are assigned as follows:

<i>Bit Pair</i>	<i>Color Source</i>
00	background 0 (65301)
01	background 1 (65302)
10	background 2 (65303)
11	foreground

As you can see, the first three bit patterns use global color sources. This means you have three global (whole screen) colors. The last bit pattern uses the normal color memory to get its color (and luminance), so that each character can have an independent foreground color.

Using the bits in pairs this way limits the resolution of your character in the horizontal direction. Sometimes this is too restrictive. The Plus/4 allows you to mix high-resolution and multicolor characters on the same screen. The Plus/4 must be in multicolor mode. This is accomplished with

**POKE 65287,PEEK(65287)OR16**

To put the chip back in high-resolution mode use

**POKE 65287,PEEK(65287)AND239**

When the Plus/4 is in multicolor mode, the color (not luminance) chosen for a given character cell determines whether the character occupying it is high resolution or multicolor. If the color associated with a given cell is greater than 8 (yellow), then the cell is multicolor. The actual foreground color of the cell is not the color chosen (*color*) but the color given by

*color*-8

If the color associated with the cell is less than or equal to 8, then the cell is high resolution and its actual color is the color chosen. This means that, when using multicolor text mode, only the first eight colors are available for use as foreground colors. However, all eight luminances are available for each of these colors.

The background registers 65301-65303 used for coloring the nonforeground pixel pairs can be assigned any of the 16 colors and 8 luminances. To calculate the value needed, use

*luminance\*16+color-1*

where the *luminance* is 0 through 7 and the color is chosen from the available colors (1-16).

*Note:* In multicolor mode, the cursor is invisible. It is a good idea to return to high-resolution character mode when exiting a program.

## Multicolor Custom Characters Example Program

In this example, type the small letters inside the quotes without shifting, and type the capital letters inside the quotes shifted.

```

10 POKE1177,62
20 FORI=0TO1023
30 POKE8192+I,PEEK(54272+I)
40 NEXT
50 POKE1177,63
60 POKE65299,(PEEK(65299)AND3)OR32
70 POKE65298,PEEK(65298)AND251
80 POKE65287,PEEK(65287)OR16
90 POKE1351,128
100 POKE65301,65
110 POKE65302,0:POKE65303,93
120 FORI=0TO15:READA:POKE8512+I,A:NEXT
130 PRINTCHR$(154)"Butterflies Fly Free"
140 PRINTSPC(10)()"SPC(10)CHR$(153)()"
150 PRINTCHR$(144)"Hit a key to restore standard characters"
160 GETKEYAS
170 POKE65299,(PEEK(65299)AND3)OR208
180 POKE65298,PEEK(65298)OR4
190 POKE65287,PEEK(65287)AND239
200 POKE1351,0
210 END
220 DATA196,241,237,237,253,253,241,193
230 DATA76,60,236,236,252,252,60,12

```

### Line-by-Line Explanation

- 10 Disable the switch to RAM of the subroutine called by PEEK.
- 20 Start loop to copy entire 1K character set.
- 30 Move upper/lower case character set to RAM, starting at location 8192.
- 40 Go back for the next byte.
- 50 Restore the subroutine called by PEEK.
- 60 Change the character base address of graphics chip to 8192.
- 70 Make graphics chip get the character patterns from RAM.
- 80 Put the graphics chip into multicolor mode.
- 90 Disable **Shift** keys.
- 100 Set screen background color 0 to white with luminance 4 (gray).  
 $4*16+2-1=65$
- 110 Set background color 1 to black.  
 $0*16+1-1=0$   
And, set background color 2 to light blue with luminance 5.  
 $5*16+14-1=93$

- 120 Read the multicolor characters in the DATA statements, and replace the parentheses characters (screen codes 40 and 41) with them.  
 $8192+40*8=8512$
- 130 PRINT a CHR\$(154) to be in multicolor with foreground color blue, followed by the sentence. When you run this program, note the appearance of the standard characters PRINTed in multicolor. This is because the bits are being interpreted as multicolor pairs.
- 140 PRINT a multicolor blue butterfly. Then change to multicolor green (CHR\$(153)) and PRINT a green butterfly. Note that in both butterflies the spot is light blue and the body is black. These areas use the shared background color registers for their color information.
- 150 PRINT a CHR\$(144) for high resolution with foreground color black, followed by the message. Note that the characters now look normal because their bits are being interpreted one at a time as high-resolution on or off messages.
- 160 Wait for a key.
- 170 Restore graphics chip looking at the location of the character ROM.
- 180 Restore graphics chip looking at ROM.
- 190 Get out of multicolor mode.
- 200 Restore **C** SHIFT keys.
- 210 End of execution.
- 220–230 Data for the two halves of the butterfly.

**Extended Color Mode** In this mode each dot is individually controlled, as they are in high-resolution characters; the difference is that you can specify any of four background colors (of the bits that are 0) for each character. You can also still choose an individual foreground color (of the bits which are 1) for each character cell. In extended color mode, the number of characters available is cut to 64 (representing settings for 6 bits). The setting of the two high bits in the screen code for a character determines which background color it will use.

Screen Code	Character Pattern	Background Color Register
0 – 63	0 – 63	65301
64 – 127	0 – 63	65302
128 – 191	0 – 63	65303
192 – 255	0 – 63	65304

The background color registers are all global color sources. This means you have four global (whole screen) colors. The foreground bits use the normal color memory to get their color (and luminance), so each character can have an independent foreground color.

To enter extended color mode, use

**POKE 65286,PEEK(65286)OR64**

To put the chip back in high-resolution mode, use

**POKE 65286,PEEK(65286)AND191**

The background registers 65301–65304, which are used for coloring the background pixels, can be assigned any of the 16 colors and 8 luminances. To calculate the value needed, use

*luminance\*16+color-1*

where the *luminance* is 0 through 7, and the *color* is chosen from the available colors (1–16). The color assigned to background registers 63501 and 63502 can also be controlled by the COLOR command for sources 0 and 3, respectively. The remaining two color registers cannot be changed with the COLOR command.

*Note:* In extended background color mode, the cursor is invisible. It is a good idea to return to high-resolution character mode when exiting a program.

Example:

```

10 SCNCLR
20 POKE65286,PEEK(65286)OR64
30 FORI=1TO4
40 PRINT"COLOR, LUMINANCE FOR";I;:INPUTC,L
50 POKE65300+I,L*16+C-1:NEXT
60 FORI=0TO63
70 POKE3072+I,I
80 POKE3136+I,I+64
90 POKE3200+I,I+128
100 POKE3264+I,I+192
110 NEXT
120 I=0
130 FORC=0TO15:FORL=0TO7
140 POKE2048+I,L*16+C
150 POKE2176+I,L*16+C
160 I=I+1
170 NEXTL,C
180 PRINT"HIT KEY":GETKEY$:POKE65286,PEEK(65286)AND191
190 COLOR0,2,7

```

## Line-by-Line Explanation

- 10 Clear the screen.
- 20 Turn on extended color mode.
- 30 For each background color register:
  - 40 Get a color and luminance.
  - 50 Put the selection into the register and continue.
- 60 For each possible character:
  - 70 Put up a set with background color 1.
  - 80 Put up a set with background color 2.
  - 90 Put up a set with background color 3.
  - 100 Put up a set with background color 4.
- 110 Next character.
- 120 I points to a spot in color memory.
- 130 For each color and luminance:
  - 140 Put color and luminance into color memory.
  - 150 Put color and luminance into color memory.
- 160 Increment pointer.
- 170 Next luminance and color.
- 180 Wait for a key press, then turn off the extended color mode.
- 190 Return the background color to white.

## Smooth Scrolling

When the last line of the Plus/4 screen is filled with text, the screen is automatically scrolled. That is, the top line is discarded, the rest of the lines are moved up one line, and the last line is cleared. Thus, the text moves up a whole line at a time. In some applications it is desirable to scroll the text (or bit map) smoothly, that is, one pixel at a time, either vertically or horizontally.

Vertical scrolling is handled by memory location 65286 (\$FF06). Normally, the first step is to shrink the screen to 24 lines. The effect of this is to expand the border to cover one character line. The information to be added at the top or

bottom of the screen can then be placed invisibly on the line and scrolled onto the screen. To shrink the screen, bit 3 of the scroll register must be cleared with

**POKE 65286,PEEK(65286)AND247**

To restore the screen later, use

**POKE 65286,PEEK(65286)OR8**

The low three bits of the scroll register determine the scrolling position; normally it is 3. To set the scrolling position, use

**POKE 65286,(PEEK(65286)AND248)OR*scrolling position***

When the scrolling position is set to 7, the bottom line is completely invisible and the top line is completely visible. Then, as the scrolling position is decreased to 0, more and more of the bottom line is revealed, and the top line is concealed. When the scrolling position is 0, all but the bottom pixel of the bottom line is visible, and all but the bottom pixel of the top line is invisible.

If the existing lines are each moved up one, and a new line is placed at the bottom while the scrolling position is simultaneously reset to 7, a smooth upward scroll is achieved. To accomplish this, the operation should be done in machine language and timed with the raster beam (see Chapter 5). But upward scrolling can be done fairly well in BASIC. The example program shows an upward scroll. Downward scrolling is accomplished by reversing the operations.

Horizontal scrolling is handled by memory location 65287 (\$FF07). The first step is normally to shrink the screen to 38 columns. The effect of this is to expand the border to cover two character columns. Then, the information to be added at the left or right of the screen can be placed invisibly on the line and scrolled onto the screen. To shrink the screen, bit 3 of the scroll register must be cleared with

**POKE 65287,PEEK(65287)AND247**

To restore the screen later, use

**POKE 65287,PEEK(65287)OR8**

The low three bits of the scroll register determine the scrolling position; normally it is 0. To set the scrolling position, use

**POKE 65287,(PEEK(65287)AND248)OR*scrolling position***

When the scrolling position is set to 0, both the left and right columns are completely invisible. Then, as the scrolling position is increased to 7, more and

more of the left column is revealed, and the second column from the right is concealed. When the scrolling position is 7, all but the left pixel of the left column is visible, and all but the right pixel of the second column from the right is invisible.

If the information in each column is moved right, and a new column is added on the left while the scrolling position is simultaneously cleared to 0, a smooth right scroll is obtained. This is best done in machine code. A left scroll is obtained by reversing the procedure.

### Smooth Scrolling Example Program

```
10 SCNCLR:FORI=0TO24:GOSUB120:NEXT
20 SR=65286
30 FORT=1TO20:NEXT
40 POKESR,(PEEK(SR)AND240)OR7
50 GOSUB120
60 FORI=6TO0STEP-1
70 FORT=1TO60:NEXT
80 POKESR,(PEEK(SR)AND240)OR1
90 NEXT
100 GETK$:IFK$=""THENGOTO30
110 POKESR,(PEEK(SR)AND240)OR11:END
120 PRINTCHR$(13) "HIT A KEY TO STOP SCROLLING";:RETURN
```

#### Line-by-Line Explanation

- 10 Clear the screen and fill it with the message.
- 20 Set SR to scroll register address.
- 30 Wait loop.
- 40 Set to 24 lines and set the scroll register value to 7.
- 50 Put the message on the hidden line.
- 60 Count down for scroll register value.
- 70 Wait loop.
- 80 Set scroll register value.
- 90 Go back for next value.
- 100 See if a key has been pressed. If not, return to line 30.
- 110 Set to 25 lines and the default scroll register value (3).
- 120 Subroutine to print carriage return followed by message.

## Animation

Computer animation consists of rapidly displaying a series of graphics, each changed somewhat from the previous one, to create the illusion of motion. There are a number of methods for doing this. The more advanced can be accomplished only through direct control of the graphics chip. Also, to attain the speeds required to fool the human eye, machine language programming is usually necessary.

Only a couple examples of straightforward methods using BASIC are presented here. To get more involved with animation, study the concepts of directly controlling the chip from machine language in the next section and experiment with them.

**Animation Using Characters** Character animation can be extremely convincing, even in BASIC. The pattern on the screen is defined by only 1 byte (a screen code placed in the proper character cell), or possibly 2, when color changes are occurring. So the pattern can be changed just by changing the value of that screen code. This can be done very quickly.

For serious animation, direct access to the screen and color memories (with POKE statements) is usually used. For information on these memories, see the summary of memory maps later in this chapter. This example uses the CHAR statement to place multicolor characters on the screen.

The animation consists of three positions (frames) of the two-character-wide multicolor butterfly presented earlier. The first (frame 1) shows its wings wide open. The second (frame 2) shows them partially closed. The last (frame 3) shows them totally closed. To animate the butterfly, the frames must be shown in the sequence 1,2,3,2,1,2,3,2,1, . . . . For the butterfly to appear to move, each successive frame must be shown in a different (adjacent) pair of character cells from the previous frame. So, we must erase the previous frame in addition to showing the next frame.

## Character Animation Example Program

In this example, type the small letters inside the quotes without shifting, and type the capital letters inside the quotes shifted.

```

10 POKE1177,62
20 FORI=0TO727
30 POKE8192+I,PEEK(54272+I)
40 NEXT
50 POKE1177,63
60 POKE65299,(PEEK(65299)AND3)OR32
70 POKE65298,PEEK(65298)AND251
80 POKE65287,PEEK(65287)OR16

```

```

90 POKE1351,128
100 POKE65301,65
110 POKE65302,0:POKE65303,93
120 FORI=0TO47:READA:POKE8512+I,A:NEXT
130 PRINTCHR$(147)CHR$(154)" Butterflies Fly Free"
140 PRINTCHR$(13);CHR$(144)" Hit key for standard characters"
150 Y=24:Y1=24
160 GETKS:IFKS<>""THENPRINTCHR$(144):GOTO220
170 A$="()":GOSUB270
180 A$="*+":GOSUB270
190 A$="-":GOSUB270
200 A$="*+":GOSUB270
210 GOTO160
220 POKE65299,(PEEK(65299)AND3)OR208
230 POKE65298,PEEK(65298)OR4
240 POKE65287,PEEK(65287)AND239
250 POKE1351,0
260 END
270 CHAR1,12,Y1," "
280 CHAR1,12,Y,CHR$(154)+A$
290 CHAR1,22,Y1," "
300 CHAR1,22,Y,CHR$(153)+A$
310 Y1=Y:Y=Y-1:IFY<0THENY=24
320 RETURN
330 DATA196,241,237,237,253,253,241,193
340 DATA76,60,236,236,252,252,60,12
350 DATA52,49,57,61,61,49,49,49
360 DATA112,48,176,240,240,48,48,48
370 DATA4,3,3,3,3,3,3,3
380 DATA64,0,0,0,0,0,0,0

```

## Notes

- 130 There is a space between the opening quotation mark and the shifted B.  
There are two spaces between the words *Butterflies* and *Fly*.
- 140 There is a space between the opening quotation mark and the shifted H.  
There are two spaces between the words *for* and *standard*, and between the words *standard* and *characters*.
- 270 There are two spaces between the pair of quotation marks.
- 290 There are two spaces between the pair of quotation marks.

## Line-by-Line Explanation

- 10 Disable switch to RAM of subroutine called by PEEK.
- 20 Start loop to copy first 90 characters.
- 30 Move upper/lower case characters to RAM starting at location 8192.

- 40 Go back for next byte.
- 50 Restore subroutine called by PEEK.
- 60 Change character base address of graphics chip to 8192.
- 70 Make graphics chip get character patterns from RAM.
- 80 Put graphics chip into multicolor mode.
- 90 Disable SHIFT keys.
- 100 Set screen background color 0 to white with luminance 4 (gray).  
 $4*16+2-1=65$
- 110 Set background color 1 to black.  
 $0*16+1-1=0$
- And set background color 2 to light blue with luminance 5.  
 $5*16+14-1=93$
- 120 Read the multicolor characters in the DATA statements, and replace the characters with screen codes 40 through 45.  
 $8192+40*8=8512$
- 130 PRINT a CHR\$(154) to be in multicolor with foreground color blue, followed by the sentence.
- 140 PRINT a carriage return (CHR\$(13)), a CHR\$(144) to be in high resolution with foreground color black, and the message.
- 150 Start the butterflies at the bottom of the screen.
- 160 Look to see if a key was hit. If so, change the foreground color to black, and go on to finish the program at line 220.
- 170 Set up for frame 1 and call output subroutine at 270.
- 180 Set up for frame 2 and call output subroutine.
- 190 Set up for frame 3 and call output subroutine.
- 200 Set up for frame 2 and call output subroutine.
- 210 Return to line 160 to go again.
- 220 Restore graphics chip looking at the location of the character ROM.
- 230 Restore graphics chip looking at ROM.
- 240 Get out of multicolor mode.

- 250 Restore **G SHIFT** keys.
  - 260 End of program execution.
  - 270 Put two space characters in the position of the previous frame for the rightmost butterfly (saved in Y1).
  - 280 Set color to multicolor blue and put up the current frame for the rightmost butterfly.
  - 290 Put two space characters in the position of the previous frame for the leftmost butterfly.
  - 300 Set color to multicolor green and put up the current frame for the leftmost butterfly.
  - 310 Save the current position in Y1. Change Y to plot the butterflies one character line higher on the next frame. But if they fly off the screen, restart them at the bottom.
  - 320 End of subroutine.
- 330–380 Data for the three frames of the butterfly.

**Animation in Multicolor Graphics** When using a bit-mapped screen for animation, you can actually have the moving objects appear to move in front of a stationary background. Usually this is done in machine language because considerable speed is required to avoid flicker. But, an example program can be done in BASIC using the SSHAPE and GSHAPE statements.

The example uses two frames of a pogo stick jumper. First, the background information that will be under the jumper is saved (SSHAPE). Then, he is plotted to the screen (GSHAPE). To prepare for the next frame, the jumper is erased (by replacing the background saved before), and the process starts again.

The SSHAPE statement allows you to save a portion of the screen in a string variable. Because a single string can be at most 255 bytes long, only a limited amount of the screen can be saved at one time. If you try to save too large a portion, a STRING TOO LONG ERROR results. SSHAPE saves only the pattern of bits in the area, not the color information.

The GSHAPE statement places such a string anywhere on the screen. The colors used are the current colors. If they have changed since the shape was saved, the new ones are used. The last parameter of GSHAPE lets you decide how the shape is to be placed on the screen:

0 = shape replaces background

1 = the inverted shape replaces background

2 = the shape is ORed with the background

3 = the shape is ANDed with the background

4 = the shape is XORed with the background

The effect of this parameter depends on the actual bit patterns used in the shape and on the background. In multicolor mode, the bit patterns are determined by which color sources (0-3) are used to draw the shape. Tables 4-3 through 4-7 show the results of using each parameter value.

TABLE 4-3. Resulting Color Source Using a Parameter Value of 0

COLOR SOURCE USED IN SHAPE	<i>Color Source Used on Screen</i>			
	0	1	2	3
0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3

TABLE 4-4. Resulting Color Source Using a Parameter Value of 1

COLOR SOURCE USED IN SHAPE	<i>Color Source Used on Screen</i>			
	0	1	2	3
0	3	3	3	3
1	2	2	2	2
2	1	1	1	1
3	0	0	0	0

TABLE 4-5. Resulting Color Source Using a Parameter Value of 2

COLOR SOURCE USED IN SHAPE	<i>Color Source Used on Screen</i>			
	0	1	2	3
0	0	1	2	3
1	1	1	3	3
2	2	3	2	3
3	3	3	3	3

TABLE 4-6. Resulting Color Source Using a Parameter Value of 3

COLOR SOURCE USED IN SHAPE	<i>Color Source Used on Screen</i>			
	0	1	2	3
0	0	0	0	0
1	0	1	0	1
2	0	0	2	2
3	0	1	2	3

TABLE 4-7. Resulting Color Source Using a Parameter Value of 4

COLOR SOURCE USED IN SHAPE	<i>Color Source Used on Screen</i>			
	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

The example program places a shape drawn with color source 2 on a background screen drawn with color source 1. Both the shape and the background screen also use background color (source 0). Using a parameter value of 2, the background "shows through" the shape wherever it uses color source 0. Where it uses color source 2, the resulting color source is 2 (where the background color source is 0) or 3 (where the background color source is 1). To keep the colors the same, color sources 2 and 3 must be set to the same color.

### Multicolor Graphics Animation Example Program

```

10 GRAPHIC3,1
20 COLOR2,5,5:COLOR3,5,5
30 DRAW2,50,50TO50,73
40 CIRCLE2,46,45,2:PAINT2,46,45
50 DRAW2,46,50TO46,55TO50,53
60 DRAW2,46,55TO46,58TO50,70
70 SSHAPEA$(0),44,41,50,73
80 DRAW2,70,50TO70,75
90 CIRCLE2,68,40,2:PAINT2,68,40
100 DRAW2,68,44TO68,49TO70,53
110 DRAW2,68,49TO68,52TO70,70
120 SSHAPEA$(1),66,36,70,75
130 SCNCLR
140 DRAW1,0,95TO159,95
150 DRAW1,0,115TO159,115

```

```

160 FORI=1TO6:READX,Y,W
170 BOX1,X,Y,X+W,94,,1
180 NEXT
190 SSHAPEB$,6,61,10,100:X1=0
200 FORX=0TO154STEP12
210 GSHAPEB$,X1+6,61
220 SSHAPEB$,X,78,X+6,110
230 GSHAPEA$(0),X,78,2
235 FORT=1TO20:NEXT
240 GSHAPEB$,X,78
250 SSHAPEB$,X+6,61,X+10,100
260 GSHAPEA$(1),X+6,61,2
265 FORT=1TO20:NEXT
270 X1=X
280 NEXTX
290 GSHAPEB$,X1+6,61
300 CHAR1,5,20,"HIT KEY"
310 GETKEYK$:GRAPHIC0
320 DATA10,20,30,50,40,10
330 DATA65,20,15,85,20,10
340 DATA105,30,15,140,50,20

```

### Line-by-Line Explanation

- 10 Enter multicolor graphic mode and clear the screen.
- 20 Set color sources 2 and 3 to purple.
- 30 Draw the first pogo stick.
- 40 Draw the first jumper's head.
- 50 Draw the first jumper's neck and arms.
- 60 Complete the first jumper.
- 70 Save the first frame in A\$(0).
- 80 Draw the second pogo stick.
- 90 Draw the second jumper's head.
- 100 Draw the second jumper's neck and arms.
- 110 Complete the second jumper.
- 120 Save the second frame in A\$(1).
- 130 Clear the screen to prepare for animation.
- 140 Draw the first side of the street.
- 150 Draw the second side of the street.
- 160 For six buildings, read their coordinates and width.

- 170 Draw a building.
  - 180 Go back for the next building.
  - 190 Save information under first frame.
  - 200 Do the following for  $x$ -coordinates, stepping across the screen by 12.
  - 210 Replace background over the previous frame.
  - 220 Save background under this frame.
  - 230 Get the first frame of jumper.
  - 235 Wait briefly. This is optional. The longer you wait, the slower the jumper goes, but the smoother the movement seems.
  - 240 Replace background over the previous frame.
  - 250 Save background under this frame.
  - 260 Get the second frame of jumper.
  - 265 Wait briefly as in line 235.
  - 270 Save the coordinate for the next time through.
  - 280 Go back for the next  $x$  value.
  - 290 Erase the final frame of the jumper.
  - 300 Write "HIT KEY".
  - 310 Wait for a key. Then return to text mode.
- 320–340 The coordinates and widths for the six buildings.

## Graphics Programming in Machine Language

A great deal of the graphics capacity of your Plus/4 is available from BASIC. However, there is frequently no substitute for being able to control the graphics chip directly, especially because it does have capabilities that are not utilized in the built-in BASIC graphics statements.

The graphics chip operates in one of two modes, either character mode or bit-map mode. The split screen available from BASIC is not a hardware feature; it is part of the BASIC language software supplied on ROM inside your computer.

When operating in character mode, you can choose high-resolution characters, multicolor characters, or extended color mode. When operating a bit-map mode, you can choose high resolution (320 by 200) or multicolor (160 by 200). Each of these possibilities is explored in this section.

## Characters

In addition to being used for most applications involving text, characters are frequently the most efficient means of displaying many other types of graphic information on the screen. The character set can be expanded to include 256 completely different characters. The entire screen is controlled through the use of only 1000 memory locations for screen codes (and 1000 for color information). This makes access to the screen extremely fast.

**Locating a Character Set in Memory** The standard built-in upper case/graphics character set is located in ROM at \$D000. The upper/lower case set is located in ROM at \$D400. If you are using a custom character set, it is usually located in RAM. To tell the graphics chip to look at RAM, you must clear bit 2 of \$FF12. The character set is located by setting the high 6 bits of \$FF13 to the high 6 bits of the character set location. Care should be taken to preserve the 2 low bits of \$FF13 because they are used for other purposes.

Normally, a character set consists of 128 eight-byte patterns (1K). Hence a character set location must be on a 1K boundary. Any screen code greater than 127 will produce a reversed screen image in high-resolution mode. To expand the character set to 256 separate patterns, turn off the hardware reverse by setting bit 7 of \$FF07 to 1. When this is done, your character set must be located on a 2K boundary.

**Locating Screen and Color Memory** A character is placed on the screen by placing its screen code into a location of screen memory. The screen code (multiplied by 8) is used by the graphics chip to look up the character's pattern in the character set. The color and luminance associated with a particular screen location are stored in the corresponding location in color memory.

These two sections of memory are each 1K (1000 locations for the 25 rows of 40 columns and 24 unused locations). They are treated as a single 2K block. Color memory is located by setting the high 5 bits of \$FF14 to the high 5 bits of the desired color memory location. Screen memory is always located in the 1K immediately following color memory. Color memory must be located on a 2K boundary. Normally, color memory is located at \$0800 (2048) and screen memory at \$0C00 (3072).

**High-Resolution Characters** In high-resolution character mode, each dot in an 8×8 character cell is controlled by a single bit in the character definition. The screen code stored in screen memory (multiplied by 8 and added to the character set base address) points at the character definition. The character definition consists of 8 bytes. The first byte corresponds to the top row of the character and

the last to the bottom row. The high bit of each byte corresponds to the leftmost dot in the character cell and the low bit to the rightmost dot. When a bit is off, the color of the corresponding dot comes from the background color register \$FF15. When a bit is on, the color of its dot comes from the byte of color memory associated with the character cell. Therefore, each character cell on the screen has its own independent color and a common background color.

Normally, if the screen code in screen memory is greater than 127, the chip subtracts 128 to determine where the character's pattern is stored. But, when the chip retrieves the pattern, it reverses the roles of on and off bits. Thus, only 128 patterns need to be stored to obtain 256 characters. The second 128 characters are exactly the reversed image of the first 128. If you do not need this reversing and would like to have 256 different characters, set bit 7 of \$FF07 to 1, as described in a previous section.

The color memory controls three functions. When bit 7 is set, the character is flashed. The next three bits determine the luminance of the color chosen with the low 4 bits.

**Extended Color Mode** Extended color mode is chosen by setting bit 6 of \$FF06 to 1. This is a high-resolution mode in the sense that each dot is individually controlled by the character pattern. The character set is only 64 characters, which must be located on a 2K boundary. The hardware reverse bit in \$FF07 is ignored, and flashing is disabled. The dots in the character with corresponding bits on get their color from color memory just as they do in high-resolution mode. However, the background color for each character cell is determined independently by the screen code used.

Screen Code	Character Pattern	Background Color Register
\$00-\$3F	\$00-\$3F	\$FF15
\$40-\$7F	\$00-\$3F	\$FF16
\$80-\$BF	\$00-\$3F	\$FF17
\$C0-\$FF	\$00-\$3F	\$FF18

Therefore, each character cell has its own independent foreground color and the choice of one of four different background colors.

**Multicolor Characters** Choose multicolor mode by setting bit 4 of \$FF07. The resolution of characters in multicolor is half that of high resolution in the horizontal direction. This is because, rather than each bit controlling the color of a single dot, each pair of bits controls the color of a pair of dots.

<i>Bit Pair</i>	<i>Color Source</i>
00	\$FF15
01	\$FF16
10	\$FF17
11	color memory

The bits of color memory are interpreted differently. The high (flashing) bit is ignored. The next three bits determine luminance as usual. But the foreground color is determined only by the lower 3 bits of the low nybble. The high bit from the low nybble determines whether this character is interpreted as high resolution or multicolor. Thus, high-resolution characters and multicolor characters may be mixed, but only the first eight colors are available for both types.

There is no hardware reverse, but that bit (bit 7 of \$FF07) still determines the size of the character set. If it is clear, screen codes from 128 to 255 result in the same characters as 0 through 127. When it is set, the full 256 codes correspond to separate character definitions.

## Bit Maps

In bit-map mode, every dot (or pair of dots) on the screen is individually controlled by a bit (or pair of bits) in memory. To enter bit-map mode, bit 5 of \$FF06 must be set to 1. Normally, your bit map will come from RAM, so bit 2 of \$FF12 must be cleared to make the graphics chip look at RAM. The location of the bit map itself is also set with \$FF12. Bits 5 through 3 correspond to bits 15 through 13 of the address of the bit map. It must be on an 8K boundary. Luminance for your bit map is located in the area of memory pointed to by \$FF14. Bits 7 through 3 of \$FF14 correspond to bits 15 through 11 in the address of the luminance memory (which occupies the position of color memory in character mode). It must be on a 2K boundary. The color memory for your bit map (which occupies the position of screen memory in character mode) is always located 1K above the luminance memory.

The screen organization for a bit map is oriented toward character cells. The first byte represents the upper left corner of the screen. That and the next 7 bytes fill the upper left character cell. The eighth byte begins the character cell immediately to the right of the first one. Page 223 has a diagram of the screen layout.

**High-Resolution Bit Maps** In a high-resolution bit map, every dot on the screen is individually controlled by a bit. The screen is organized as shown. Within each of the 8000 bytes that make up the bit map, the high bit determines the status of the leftmost dot on the screen, and the low bit determines the status of the rightmost dot on the screen. The resolution is 320 by 200 (a total of 64000 individual dots).

## Number of Bytes from the Base of the Bit Map

Byte 0	Byte 8	Byte 16	...	Byte 312
Byte 1	Byte 9	Byte 17	...	Byte 313
Byte 2	Byte 10	Byte 18	...	Byte 314
Byte 3	Byte 11	Byte 19	...	Byte 315
Byte 4	Byte 12	Byte 20	...	Byte 316
Byte 5	Byte 13	Byte 21	...	Byte 317
Byte 6	Byte 14	Byte 22	...	Byte 318
Byte 7	Byte 15	Byte 23	...	Byte 319
Byte 320	Byte 328	Byte 336	...	Byte 632
Byte 321	Byte 329	Byte 337	...	Byte 633
Byte 322	Byte 330	Byte 338	...	Byte 634
Byte 323	Byte 331	Byte 339	...	Byte 635
Byte 324	Byte 332	Byte 340	...	Byte 636
Byte 325	Byte 333	Byte 341	...	Byte 637
Byte 326	Byte 334	Byte 342	...	Byte 638
Byte 327	Byte 335	Byte 343	...	Byte 639
Byte 7680	Byte 7688	Byte 7696	...	Byte 7992
Byte 7681	Byte 7689	Byte 7697	...	Byte 7993
Byte 7682	Byte 7690	Byte 7698	...	Byte 7994
Byte 7683	Byte 7691	Byte 7699	...	Byte 7995
Byte 7684	Byte 7692	Byte 7700	...	Byte 7996
Byte 7685	Byte 7693	Byte 7701	...	Byte 7997
Byte 7686	Byte 7694	Byte 7702	...	Byte 7998
Byte 7687	Byte 7695	Byte 7703	...	Byte 7999

Each byte in the luminance memory corresponds to a character cell on the bit map. The luminance byte is treated as 2 nybbles. The high nibble determines the luminance of bits that are off (0) in that character cell, and the low nibble determines the luminance of bits that are on (1) in that character cell.

In a similar way, the colors of each dot are controlled by the color memory in which each byte corresponds to a character cell. But here the high nibble determines the color of bits that are on (1) in that character cell, whereas the low nibble determines the color of bits that are off (0) in that character cell.

*Note:* The nybbles are reversed from luminance memory.

**Multicolor Bit Maps** By setting bit 4 of \$FF07, the graphics chip is put into multicolor mode. If this is done in addition to the steps for a bit map, a multicolor bit map results. In a multicolor bit map, the screen is organized exactly as a

high-resolution bit map, except that each pair of dots on the screen is controlled by a pair of bits in memory. This cuts the resolution to 160 by 200 (32000 dots).

But the color flexibility is increased by the addition of two global colors. Within each byte of the bit map, the high 2 bits determine the color of the leftmost pair of dots on the screen. The 2 low bits determine the color of the rightmost pair of dots on the screen. The colors are determined as follows:

<i>Bit Pair</i>	<i>Luminance Source</i>	<i>Color Source</i>
00	\$FF15	\$FF15
01	low nybble of luminance	high nybble of color
10	high nybble of luminance	low nybble of color
11	\$FF16	\$FF16

## Summary of Memory Map Considerations

This section is organized by each type of graphics mode and details the use of memory. It is intended for quick reference. The previous section explains the actual operation of each mode.

### High-Resolution Characters

- \$FF07 Bit 7 determines hardware reverse on and 128 characters (0), or reverse off and 256 characters (1).
- \$FF12 Bit 2 determines RAM (0) or ROM (1).
- \$FF13 Bits 7–2 determine location of character set (bits 15–10).
- \$FF14 Bits 7–3 determine location of color/screen memory (bits 15–11).
- \$FF15 Background luminance and color.
- \$FF19 Border luminance and color.

Character color, luminance, and flashing are determined by color memory (normally \$0800–\$0BE7). Characters displayed are determined by screen memory (normally \$0C00–\$0FE7).

### Extended Color Characters

- \$FF06 Set bit 6 for extended color mode.
- \$FF07 Bit 7 determines 128 characters (0), or 256 characters (1).

- \$FF12 Bit 2 determines RAM (0) or ROM (1).
- \$FF13 Bits 7–2 determine location of character set (bits 15–10).
- \$FF14 Bits 7–3 determine location of color/screen memory (bits 15–11).
- \$FF15 Background 0 luminance and color.
- \$FF16 Background 1 luminance and color.
- \$FF17 Background 2 luminance and color.
- \$FF18 Background 3 luminance and color.
- \$FF19 Border luminance and color.

Character color and luminance are determined by color memory (normally \$0800–\$0BE7). Characters displayed and their background color are determined by screen memory (normally \$0C00–\$0FE7).

## Multicolor Characters

- \$FF07 Set bit 4 for multicolor mode. Bit 7 determines 128 characters (0) or 256 characters (1).
- \$FF12 Bit 2 determines RAM (0) or ROM (1).
- \$FF13 Bits 7–2 determine location of character set (bits 15–10).
- \$FF14 Bits 7–3 determine location of color/screen memory (bits 15–11).
- \$FF15 Background 0 luminance and color.
- \$FF16 Background 1 luminance and color.
- \$FF17 Background 2 luminance and color.
- \$FF19 Border luminance and color.

Character color, luminance, and high-resolution or multicolor status are determined by color memory (normally \$0800–\$0BE7). Characters displayed are determined by screen memory (normally \$0C00–\$0FE7).

## High-Resolution Bit Map

- \$FF06 Set bit 5 for bit-map mode.
- \$FF12 Bit 2 determines RAM (0) or ROM (1). Bits 5–3 determine location of bit map (bits 15–13).

- \$FF14 Bits 7–3 determine location of luminance/color memory (bits 15–11).  
\$FF19 Border luminance and color.

The bit map displayed is determined by bit-map memory. The luminances of both colors are determined by color memory (normally \$0800–\$0BE7). Color for both colors is determined by screen memory (normally \$0C00–\$0FE7).

### Multicolor Bit Map

- \$FF06 Set bit 5 for bit-map mode.  
\$FF07 Set bit 4 for multicolor mode.  
\$FF12 Bit 2 determines RAM (0) or ROM (1).  
Bits 5–3 determine location of bit map (bits 15–13).  
\$FF14 Bits 7–3 determine location of luminance/color memory (bits 15–11).  
\$FF15 Background 0 luminance and color.  
\$FF16 Background 1 luminance and color.  
\$FF19 Border luminance and color.

The bit map displayed is determined by bit-map memory. The luminances of both colors are determined by color memory (normally \$0800–\$0BE7). Color for both local colors is determined by screen memory (normally \$0C00–\$0FE7).

---

# 5 Machine Language on the Commodore Plus/4

---

A computer consists of a central processing unit (CPU), some memory, and input/output (I/O) circuitry that communicates with the outside world. When the CPU is a microprocessor, the computer is a microcomputer. The CPU of the Commodore Plus/4 is a 7501 microprocessor. A computer program consists of a list of instructions to the CPU that command it to perform operations on the memory and the I/O units. Programming done at this level is called machine-language programming. Higher-level languages such as BASIC are written in machine language. An assembler may be used to allow a programmer to use mnemonics for instructions and labels for memory locations. The term *assembly language* refers to machine-language programs that are created with the use of an assembler.

The 7501 microprocessor understands exactly the same instructions as the 6502 microprocessor. Because this instruction set originated with the 6502, it is referred to as 6502 machine language.

This chapter should be sufficient for 6502 programmers to transfer their skills to the Commodore Plus/4. It should also allow programmers of other microprocessors to begin programming in 6502. Programmers new to machine language of any kind may want to refer to additional material on machine-language programming. Many such books are available.

## Introduction to Using Machine Language

Programming in machine language (or machine code) puts your program in complete control of what the computer is doing. To get this control, some of the simplicity of programming in BASIC or another high-level language is sacrificed. In general, programs in machine language are faster and more memory efficient than programs in BASIC. When these qualities are required, the sacrifice is worthwhile.

The Commodore Plus/4 is equipped with a built-in machine-language monitor. The monitor is accessed from BASIC with the MONITOR command. Using this monitor, machine-language programs may be entered and executed. Several other functions are also provided (see the next section), making it possible to experiment in machine language without purchasing any additional software. The monitor is also very useful for debugging machine-language programs created using an assembler.

Machine-language programming generally requires a more detailed knowledge of the hardware devices used in the computer than BASIC programming does. Information on using the various chips in the Plus/4 is provided in the appropriate chapter in this book. Some of the subroutines in the ROM of the Plus/4 are available to the machine-language programmer. Information on these subroutines is included in this chapter. These subroutines are particularly useful in performing I/O operations for which many programmers do not want to write their own code.

## **Machine-Language Monitor Commands**

The monitor is entered by issuing the MONITOR command from BASIC. When the monitor is entered, the current contents of the program counter (PC), status register (SR), accumulator (AC), X register (XR), Y register (YR), and stack pointer (SP) are displayed in hexadecimal.

### **MONITOR**

```
PC  SR  AC  XR  YR  SP  
;  0000 00  00  00  00  F9
```

The screen editor is fully functional while you are in the monitor. That is, whenever a RETURN is entered, the line on which the cursor resides is processed, and the screen scrolls normally.

Whenever the monitor encounters an error in a line it is attempting to process, a question mark is displayed and no action is taken. For a list of monitor input/output errors, see Appendix A.

*Note:* The monitor normally accesses RAM up to \$8000 and ROM thereafter. If you want to access all RAM, change location \$07F8 to \$80 (see M command). To switch back to ROM, change it back to \$00.

## **Entering a Program—The A or . Command**

The A (Assemble) command is used to enter a line of machine code. The syntax is as follows:

*A address opcode mnemonic operand*

or

*. address opcode mnemonic operand*

The *address* is the hexadecimal address at which the line of machine code is placed.

The *opcode mnemonic* is a valid 6502 mnemonic (see section on 6502 instruction set).

The *operand* is a valid 6502 operand (see section on 6502 instruction set and addressing modes).

When such a line and a RETURN are entered, the monitor translates the mnemonic and operand into hexadecimal and outputs them. Then it automatically calculates the next available address and waits for the next line of code. If only a RETURN is entered following the *A address* prompt, the monitor exits assembly mode. If an incorrect line is entered, the translation to hexadecimal is not done, and a question mark is displayed. Also, the monitor does not attempt to calculate a next available address.

Example: **A 2000 LDA #\$05**

will result in the following display:

```
A 2000 A9 05    LDA #$05  
A 2002
```

The monitor is asked to assemble a load accumulator with the number 5 instruction at the address \$2000. It translates this into the hexadecimal codes A9 and 05. It then prompts for the next line of code at address \$2002.

## Comparing Two Sections of Memory—The C Command

The C (Compare) command is used to find the differences in content between two sections of memory. The syntax is as follows:

**C start address 1 end address 1 start address 2**

The *start address 1* is the hexadecimal address at which the first section of memory begins. The *end address 1* is the last hexadecimal address of the first section of memory. The *start address 2* is the hexadecimal address of the section of memory to compare with the previously defined section. No end address is

needed for the second section of memory; it is assumed to be the same length as the first section. The specified end and start points are included in the Compare.

The monitor reports in ascending order all addresses from the first section that do not have the same contents as the corresponding address in the second section. If the two sections of memory are found to be identical, only a RETURN is output.

Example: C 3000 3004 2000  
3003 3004

The monitor is asked to compare the contents of the section of memory from \$3000 to \$3004, inclusive, with the contents of the section of memory from \$2000 to \$2004 inclusive. It reports that the contents of \$3003 do not match those of \$2003 and that the contents of \$3004 do not match those of \$2004.

### Examining a Program—The D Command

The D (Disassemble) command is used to view a line or lines of machine code. The syntax is as follows:

*D start address end address*

The *start address* is the optional hexadecimal address at which the disassembly starts. The *end address* is the optional hexadecimal address at which the disassembly stops.

The monitor attempts to translate the contents of the designated section of memory into opcode mnemonics and operands. When an illegal opcode is encountered, question marks are displayed. The specified end address is the last location disassembled unless it occurs in the middle of an instruction, in which case the entire instruction is disassembled. If the end address is not specified, 21 memory locations are disassembled. If no addresses are specified, disassembly begins one location beyond the last location accessed. Thus, code is continually disassembled 21 locations at a time when you enter the D command repeatedly.

The disassembly is displayed on the screen preceded by a period and the address at which the instruction starts. Since the period is the equivalent of an A (Assemble) command, cursoring to a disassembled line, changing the mnemonic opcode or operand, and entering a RETURN is a good way to correct instructions.

*Note:* You cannot correct instructions by changing the hexadecimal dump before the opcode mnemonic.

Example: D F2A4 F2A8  
. F2A4 A2 FF     LDX #\$FF

- |           |     |
|-----------|-----|
| . F2A6 78 | SEI |
| . F2A7 9A | TXS |
| . F2A8 D8 | CLD |

The monitor is asked to disassemble the contents of ROM locations \$F2A4 through \$F2A8 inclusive. It displays the contents in hexadecimal and in mnemonic opcodes and operands.

### Filling Memory with a Specified Byte—The F Command

The F (Fill) command is used to insert a single specified data byte into every location of a section of memory. The syntax is as follows:

**F start address end address data byte**

The *start address* is the hexadecimal address at which the section of memory begins. The *end address* is the last hexadecimal address of the section of memory.

The *data byte* is the hexadecimal number (one or two digits) to be inserted into each memory location.

Every location from the starting address through the ending address is set equal to the specified data byte. If you attempt to alter ROM locations in this way, the RAM under the ROM is filled with the specified data byte.

**Example: F 3000 3004 FF**

The monitor is asked to insert the value \$FF into every memory location from \$3000 to \$3004 inclusive.

### Executing a Program—The G Command

The G (Go) command is used to begin execution of a machine code program. The syntax is as follows:

**G address**

The *address* is the optional hexadecimal address at which the execution begins.

The monitor begins executing instructions at the address specified. If no address is specified, execution begins at the current value of the program counter. To view the current value of the program counter, use the R command. To return to the monitor after execution, the machine language program must end with a BRK instruction. The value of the program counter following the execution of a BRK instruction is not necessarily the address of the instruction following the BRK. Care must be taken when you do not specify the starting address.

Example: G 2000

The monitor is asked to begin executing with the instruction at location \$2000.

### Searching Memory for Specified Bytes—The H Command

The H (Hunt) command is used to search a section of memory for occurrences of specified data byte(s). The syntax is as follows:

**H start address end address data**

The *start address* is the hexadecimal address at which the section of memory begins.

The *end address* is the last hexadecimal address of the section of memory.

The *data* are the one or more hexadecimal bytes (one or two digits) separated by spaces, or a character string preceded by a single quote.

Every location from the starting address through the ending address is examined and compared with the first byte of the *data*. When a match is found, the next location is compared with the second byte, and so on until a difference is found. If no difference is found, the starting location of the data in memory is displayed.

If the *data* are specified by the use of a character string, the bytes that match are the corresponding CHR\$ codes (see Appendix C) of the characters.

Example: H 8000 FFFF 43 42 4D  
8007 FC56

The monitor is asked to search every memory location from \$8000 to \$FFFF inclusive for the start of the sequence \$43, \$42, \$4D. The monitor finds this exact sequence twice, starting at \$8007 and starting at \$FC56.

Example: H 8000 FFFF 'COMMODORE  
80CF E3A7

The monitor is asked to search every memory location from \$8000 through \$FFFF for the start of the sequence of bytes given by the CHR\$ codes of each letter in the word “COMMODORE.” It finds this sequence twice, starting at \$80CF and at \$E3A7.

### Loading from Cassette or Disk—The L Command

The L (Load) command is used to load a machine-language program previously saved (see the section on the S command) on cassette or disk. The syntax is as follows:

**L *filename,device***

The *filename* is the optional Plus/4 file name surrounded by quotation marks. The *device* is the optional Plus/4 device number (see Chapter 6).

The monitor loads the specified program file into memory. The file must be a program-type file (see Chapter 6) because these have the start address for loading stored in the first 2 bytes of the file. These bytes are created by the save (S) command and represent the first location that was saved into the file. Hence, a program is always loaded into the same section of memory from which it was saved.

If the file name is omitted, the device must be the cassette. If the device is omitted, it is assumed to be the cassette.

**Example:** L"TEST",8  
SEARCHING FOR TEST  
LOADING

The monitor is asked to load the program "TEST" from the disk drive (device 8). It reports that it is searching for the file, and when the file is found, the LOAD-ING message appears.

**Example:** L  
PRESS PLAY ON TAPE  
OK  
SEARCHING  
FOUND  
LOADING

The monitor is asked to load the next program on the cassette. It reports when the file is found and when loading begins.

## Examining Memory—The M Command

The M (Memory) command is used to examine a section of memory. The syntax is as follows:

**M *start address end address***

The *start address* is the optional hexadecimal address at which the memory dump starts. The *end address* is the optional hexadecimal address at which the memory dump stops.

The monitor outputs the contents of the specified section of memory to the screen 8 bytes per line. The specified end address is the last location dumped unless it occurs in the middle of a set of 8 bytes, in which case the entire set is

displayed. If the end address is not specified, 96 memory locations are displayed. If no addresses are specified, the memory dump begins one location beyond the last location accessed. Thus, memory is continually dumped 96 locations at a time when you enter the M command repeatedly.

The memory dump is displayed on the screen preceded by a greater-than sign and by the address at which the line starts. Because the greater-than sign is used to change memory locations, cursoring to a dumped line, changing the desired memory location(s), and entering a RETURN is a good way of altering the contents of memory. If you attempt to alter ROM locations in this way, the contents of the ROM are redisplayed. The RAM under the ROM will be altered.

**Example:** M 80CF 80D7

```
>80CF 43 4F 4D 4D 4F 44 4F 52 : COMMODORE
>80D7 45 20 42 41 53 49 43 20 : E BASIC
```

*Note:* Text that would be displayed in reverse mode on your screen is shown boxed in this book.

The monitor is asked to display the contents of memory locations \$80CF through \$80D7. It displays the contents in hexadecimal and as characters.

### Changing Memory—The > Command

The > (greater-than sign) command is used to change the contents of memory. The syntax is as follows:

*> address data*

The *address* is the starting address for the memory change.

The *data* are 1 to 8 hexadecimal bytes of data separated by spaces to be placed in memory.

The monitor places the specified data bytes into memory starting at the specified address. If more than 8 data bytes are specified, the excess is ignored. After placing the bytes in memory, the monitor displays the contents of the memory as in the M (Memory) command. If ROM locations are specified the data are placed in the RAM underneath, but the contents of the ROM are displayed.

**Example:** >2000 43 59 4E 44 49 45 20 4D

results in the following display:

```
>2000 43 59 4E 44 49 45 20 4D : CYNDIE M
```

The monitor is asked to place the bytes \$43, \$59, \$4E, \$44, \$49, \$45, \$20, and \$4D in memory starting at \$2000. It does so and displays the resulting contents of \$2000 through \$2007.

## Examining Registers—The R Command

The R (Register) command is used to display the contents of the 6502 registers. The syntax is as follows:

R

The monitor displays the contents of the 6502 registers. Listed will be the program counter (PC), status register (SR), accumulator (A), X register (XR), Y register (YR), and stack pointer (SP). The register dump is displayed on the screen preceded by a semicolon. Since the semicolon is used to change register contents, cursoring up to that line, changing the desired register(s), and entering a RETURN is a good way of altering the contents of the registers.

Example:

R

```
PC SR AC XR YR SP  
; 0000 00 00 00 00 F9
```

The monitor is asked to display the contents of the 6502 registers and does so.

## Changing Registers—The ; Command

The ; (semicolon) command is used to change the contents of the 6502 registers. The syntax is as follows:

```
 ; PC SR AC XR YR SP
```

The *PC* is the new program counter in hexadecimal.

The *SR* is the new status register in hexadecimal.

The *AC* is the new accumulator in hexadecimal.

The *XR* is the new X register in hexadecimal.

The *YR* is the new Y register in hexadecimal.

The *SP* is the new stack pointer in hexadecimal.

The monitor places the specified data bytes into the 6502 registers. This command is normally executed following an R (Register) command by cursoring back to its output line, changing the appropriate numbers, and entering a RETURN. Only the register(s) to be changed and values for those displayed to the left of them must be entered.

Example: ; 2000 00 05

The monitor is asked to change the program counter to \$2000, the status register to \$00, and the accumulator to \$05. The remaining registers are unchanged. Performing an R command at this point verifies the changes.

R

PC	SR	AC	XR	YR	SP
; 2000 00 05 00 00 F9					

### Saving on Cassette or Disk—The S Command

The S (Save) command is used to save a section of memory on cassette or disk. The syntax is as follows:

**S *filename,device,start address,end address***

The *filename* is the Plus/4 file name surrounded by quotation marks.

The *device* is the Plus/4 device number (see Chapter 6).

The *start address* is the address of the first location to save.

The *end address* is the address FOLLOWING the last location to save.

The monitor saves the specified section of memory in a program file with the specified name. It automatically saves the start address in the first two bytes of the file, so that it is loaded into the proper memory location by the L (Load) command. It is important to note carefully that the memory range specified is NOT saved inclusively; rather, the start address is included and the end address is not.

Example: S"TEST",8,2000,3000  
SAVING TEST

The monitor is asked to save the contents of memory from \$2000 through \$2FFF as the program “TEST” on the disk drive (device 8). It reports that it is saving the file.

Example: S"TEST",1,2000,2100  
PRESS PLAY & RECORD ON TAPE  
OK  
SAVING TEST

The monitor is asked to save the contents of memory from \$2000 through \$20FF as the program “TEST” on the cassette (device 1).

## Copying a Section of Memory—The T Command

The T (Transfer) command is used to copy the contents of a section of memory into another section of memory. The syntax is as follows:

**T start address 1 end address 1 start address 2**

The *start address 1* is the hexadecimal address at which the section of memory to be copied begins.

The *end address 1* is the last hexadecimal address of the section of memory to be copied.

The *start address 2* is the hexadecimal address of the destination section of memory.

No end address is needed for the destination section of memory. It is assumed to be the same length as the section of memory to be copied. The specified end and start points are copied.

The monitor places the contents of the specified section of memory into the destination section of memory. The contents of the first section are not altered, and the previous contents of the destination section are lost. A transfer from a ROM section copies the contents of the ROM. A transfer to a ROM area places the copied data in the RAM underneath.

A section of memory may be successfully transferred to an overlapping destination section if the destination section begins at an address lower than the original section. Attempts to transfer to an overlapping section beginning in the middle of the original section do not result in copying the information correctly.

**Example:** **T 2000 20FF 3000**

The contents of \$2000 through \$20FF are copied into \$3000 through \$30FF.

## Verifying a File on Cassette or Disk—The V Command

The V (Verify) command is used to compare the contents of a file saved on cassette or disk with the contents of memory. The syntax is as follows:

**V filename,device**

The *filename* is the optional Plus/4 file name surrounded by quotation marks.

The *device* is the optional Plus/4 device number (see Chapter 6).

The monitor compares the specified program file with memory. The file must be a program-type file (see Chapter 6), as these have the start address for comparing stored in the first two bytes of the file. These bytes are created by the

Save (S) command and represent the first location that was saved to the file. Hence, a program is always compared with the same section of memory from which it was saved. If the file name is omitted, the device must be the cassette. If the device is omitted, it is assumed to be the cassette.

If any differences between the contents of the file and the memory from which it was saved are found, a verifying error is reported. If the memory has not been changed since the save, the file should be saved again, possibly on a different tape or disk.

**Example:**

```
V"TEST",8
SEARCHING FOR TEST
VERIFYING ERROR
```

The monitor is asked to compare the program “TEST” from the disk drive (device 8) with the section of memory from which it was saved. A difference is found and reported as an error.

**Example:**

```
V
PRESS PLAY ON TAPE
OK
SEARCHING
FOUND
VERIFYING
```

The monitor is asked to compare the first program on the cassette with the memory from which it was saved. No differences are found.

### Exiting to BASIC from the Monitor—The X Command

The X (Exit) command is used to return to BASIC from the monitor. The syntax is as follows:

```
X
```

Control is returned to the BASIC language interpreter. If the stack pointer has been altered while in the monitor, the BASIC CLR command should be executed when you return to BASIC.

**Example:**

```
X
READY.
CLR

READY.
```

## The 6502 Microprocessor

The actual microprocessor chip used in the Plus/4 is a 7501. However, for programming purposes, this chip is the same as a 6502. In this section, the operation of the microprocessor is reviewed. Programmers familiar with any machine language should be able to adapt to the 6502 after studying this section.

The examples in this section are designed to show the usage of the 6502 instruction set and can be entered (and saved on cassette or disk) using the built-in machine-language monitor (see previous section).

### Registers

The 6502 is equipped with six internal registers. These registers are accessed with special instructions, which are not the same as instructions that access memory locations. Three of these registers are user registers. It is usually faster to access the registers than to access memory, so using them efficiently can speed up a machine-language program.

***The Program Counter*** The program counter (PC) is a 16-bit register that stores the address of the next instruction to be executed. When an instruction is executed, the PC is incremented so as to point to the following instruction. When a branch or jump instruction is executed, the PC is changed accordingly.

***The Status Register*** The status register (SR) is an 8-bit register containing the status flags for the microprocessor. Only 7 bits are actually used, and they are assigned as follows:

Bit	Label	Name	Usage
7	N	Negative	This bit is set to 1 when the last result had the high (sign) bit set to 1 and cleared to 0 when the last result had the high bit cleared to 0.
6	V	Overflow	This bit is set to 1 when the last operation resulted in a two's complement arithmetic overflow. That is, two numbers with the same sign were added, resulting in a number of the opposite sign; a positive number was subtracted from a negative number yielding a positive number; or a negative number was subtracted from a positive number yielding a negative number. If no overflow occurred, it is cleared to 0.

<i>Bit</i>	<i>Label</i>	<i>Name</i>	<i>Usage</i>
5		Not Used	
4	B	Break	This bit is set to 1 only when the last instruction was a BRK, and is cleared to 0 otherwise.
3	D	Decimal	This bit is set to 1 when the microprocessor is operating in decimal mode and is cleared to 0 otherwise.
2	I	Interrupt	When this bit is set to 1, maskable interrupts are not performed. When it is cleared to 0 interrupts are restored.
1	Z	Zero	This bit is set to 1 when the last result was zero and cleared to 0 when the last result was nonzero.
0	C	Carry	The carry is set to 1 if the last addition resulted in a carry or the last subtraction did not require a borrow. Otherwise it is cleared to 0.

***The Accumulator*** The accumulator (.A) is an 8-bit general purpose register. It is generally used as the main communication register between segments of code.

***The X Register*** The X register (.X) is an 8-bit index register. It may be used as an index (see sections on indexed and indexed indirect addressing) or as temporary storage for intermediate results.

***The Y Register*** The Y register (.Y) is an 8-bit index register. It may be used as an index (see sections on indexed and indirect indexed addressing) or as temporary storage for intermediate results.

***The Stack Pointer*** The stack pointer (SP) is an 8-bit register that contains the current pointer to the stack on page 1 (see the next section on the stack).

## The Stack

The 6502 maintains a 256-byte stack that is always located on page 1 (memory locations \$0100 through \$01FF). The stack pointer register (SP) maintains the current position within the stack. It is initially set by the operating system at \$FF to point to the first position (\$01FF). When an item is pushed onto the stack, it is

placed at the position indicated by the SP, and the SP is decremented. When an item is pulled from the stack, the SP is incremented, and the value is gotten from the position indicated by this new value for the SP. In addition to being affected when a push onto or pull from the stack is performed, the SP value can be altered with the TXS instruction or put into the X register with the TSX instruction.

The stack is used by the processor to store the return address for a return from subroutine. When a JSR instruction is executed, the address of the last byte of the JSR instruction is pushed onto the stack (using two stack locations). When an RTS instruction is executed, the 2 bytes are pulled from the stack and placed in the program counter. The PC is then incremented and points to the instruction following the JSR instruction.

The processor uses the stack in a similar fashion when it receives an interrupt. In this case, however, not only the return address but also the current value of the status register are pushed onto the stack before the interrupt is processed. When an RTI instruction is executed, the status register value and the 2 return address bytes are pulled off the stack and restored.

Four additional instructions affect the stack: PHA, PHP, PLA, and PLP. PHA and PHP push the accumulator and status register, respectively, onto the stack. PLA and PLP pull a value from the stack and place it in the accumulator and status register, respectively.

Care must always be taken to maintain the stack properly; that is, do not perform a JSR without a corresponding RTS, and vice versa. Do not perform a PHA or PHP without a corresponding PLA or PLP. Failure to properly match the pushes with pulls from the stack may have bizarre and unpredictable results. It is possible to overflow the stack. Usually this occurs because of improper nesting rather than an actual need for more space. If a program overflows the stack from a need for more space, it should be reorganized to use the stack more sparingly.

## Instruction Set

The instructions available in the 6502 microprocessor are presented in alphabetical order. A short explanation of the instruction, and which registers and flags it affects, introduces the instruction. The following table shows the available addressing modes for the instruction (see the next section for an explanation of all the addressing modes), the proper syntax for using the instruction from the built-in machine-language monitor, the hexadecimal opcode, the number of bytes the instruction occupies, and the number of machine cycles it takes to execute. At the end of this section, all of this information is briefly summarized.

Each instruction is accompanied by short example programs. These programs may be entered through the machine-language monitor (see the previous section

on the monitor). The examples may use concepts that are fully explained elsewhere in this book in addition to the instruction they are listed with.

Following is a list of the notation used in this section:

A	Accumulator
B	Break flag
C	Carry flag
D	Decimal mode flag
h	Hexadecimal digit (0 – F)
I	Interrupt disable flag
M	Memory location
N	Negative flag
V	Overflow flag
X	X register
Y	Y register
Z	Zero flag
$\wedge$	Logical AND
$\vee$	Logical OR
$\vee\!\vee$	Logical exclusive OR
$\sim$	Logical inverse
$\downarrow$	Push onto stack
$\uparrow$	Pull from stack
$\rightarrow$	Transfer
$\leftarrow$	Transfer

### ADC—Add Memory to Accumulator with Carry

The value currently in the accumulator plus the carry is added to the specified operand, and the result is placed in the accumulator. Normally the carry is cleared (see CLC) prior to an addition. When in decimal mode, the Z flag is not valid; check the accumulator for a zero result.

Operation: $A + M + C \rightarrow A, C$		Flags Affected: N, Z, C, V		
Addressing Mode	Syntax	Opcode	Bytes	Cycles
Immediate	ADC #\$hh	69	2	2
Zero page	ADC \$hh	65	2	3
Zero page, X	ADC \$hh,X	75	2	4
Absolute	ADC \$hhhh	6D	3	4
Absolute, X	ADC \$hhhh,X	7D	3	4*
Absolute, Y	ADC \$hhhh,Y	79	3	4*
(Indirect, X)	ADC (\$hh,X)	61	2	6
(Indirect), Y	ADC (\$hh),Y	71	2	5*

\* Add 1 when a page boundary is crossed.

The status of the carry following an addition reflects the result. If a carry was generated, the carry is set to 1. If no carry was generated, the carry is cleared to 0. This fact allows multiple precision additions, as shown in the example. If the result is negative, the N flag is set to 1; otherwise, it is cleared to 0. If the result is zero, the Z flag is set to 1; otherwise it is cleared to 0. If the result exceeds +127 or -128, the overflow flag is set to 1; otherwise it is cleared to 0.

**Example:** Frequently it is necessary to have a greater precision in calculations than the 256 possible values for a single byte. This is possible by treating a group of two or more bytes as a single number. In this example, the two 32-bit numbers stored at \$2100-\$2103 and \$2104-\$2107 (high byte to low byte) are added, and the result is stored in \$2108-\$210B.

- . 2000 18 CLC Clear the carry prior to the first addition.
- . 2001 A2 03 LDX #\$03 .X will index through the 4 bytes.
- . 2003 BD 00 21 LDA \$2100,X Get a byte of the first number.
- . 2006 7D 04 21 ADC \$2104,X Add corresponding byte from the second number.
- . 2009 9D 08 21 STA \$2108,X Store the result in destination.
- . 200C CA DEX Decrement .X to point at the next byte.
- . 200D 10 F4 BPL \$2003 Continue processing until all 4 bytes are done.
- . 200F 00 BRK Stop processing.

Use the M (Memory) command to examine and modify the contents of \$2100-\$2107 before executing the program (using G 2000). Then check the

contents of \$2100-\$210B after the program executes. The values in \$2108-\$210B will be the sum of \$2100-\$2103 and \$2104-\$2107.

The carry flag is used to transmit the carry information between the bytes. Before the first ADC, the carry is cleared, so the first result is correct. If a carry is produced, it is automatically added into the next byte because the program does not clear the carry before performing the next ADC.

## AND—AND Memory with Accumulator

The value currently in the accumulator is logically ANDed to the specified operand, and the result is placed in the accumulator.

<i>Operation: A <math>\wedge</math> M <math>\rightarrow</math> A</i>		<i>Flags Affected: N, Z</i>		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Immediate	AND #\$hh	29	2	2
Zero page	AND \$hh	25	2	3
Zero page, X	AND \$hh,X	35	2	4
Absolute	AND \$hhhh	2D	3	4
Absolute, X	AND \$hhhh,X	3D	3	4*
Absolute, Y	AND \$hhhh,Y	39	3	4*
(Indirect, X)	AND (\$hh,X)	21	2	6
(Indirect), Y	AND (\$hh),Y	31	2	5*

\* Add 1 when a page boundary is crossed.

Those bits that are set to 1 in both the value in the accumulator and the value in the operand are set to 1 in the result. Bits that are cleared to 0 in either value are cleared to 0 in the result. If the result has the high bit set to 1, the N flag is set to 1; otherwise, it is cleared to 0. If the result is zero, the Z bit is set to 1; otherwise, it is cleared to 0.

**Example:** The AND instruction is useful when one or more bits of a byte must be cleared to 0 while the other bits remain unchanged. This example increments the luminance of the screen without affecting the color and times itself by looking at a single bit in the jiffy clock timer on zero page.

- . 2000 A9 00 LDA #\$00 Start at zero luminance.
- . 2002 85 FF STA \$FF Save current luminance at \$FF.
- . 2004 AD 15 FF LDA \$FF15 Get value of background register.
- . 2007 29 OF AND #\$0F Do not change the color bits.
- . 2009 05 FF ORA \$FF Put in luminance.
- . 200B 8D 15 FF STA \$FF15 Put value into background register.

. 200E A5 A5	LDA \$A5	Get low byte of jiffy clock.
. 2010 29 08	AND #\$08	Wait until bit 3 is on.
. 2012 F0 FA	BEQ \$200E	If it is still off, keep waiting.
. 2014 A5 A5	LDA \$A5	Get low byte of jiffy clock.
. 2016 29 08	AND #\$08	Wait until bit 3 is off.
. 2018 D0 FA	BNE \$2014	If it is still on, keep waiting.
. 201A 18	CLC	Get ready to calculate new luminance.
. 201B A5 FF	LDA \$FF	Get current value.
. 201D 69 10	ADC #\$10	Add \$10 to it to go up one luminance level.
. 201F 85 FF	STA \$FF	Save it.
. 2021 10 E1	BPL \$2004	If not at \$80, go on.
. 2023 00	BRK	Stop execution.

The program is executed using G 2000.

## ASL—Shift Left One Bit

Each bit in the specified operand is shifted one bit to the left. The high bit is shifted into the carry flag, and a 0 is shifted into the low bit.

Operation: $C \leftarrow [7]$ Addressing Mode	$O7 \leftarrow 0$ Syntax	Flags Affected: N, Z, C		
		Opcode	Bytes	Cycles
Accumulator	ASL	0A	1	2
Zero page	ASL \$hh	06	2	5
Zero page, X	ASL \$hh,X	16	2	6
Absolute	ASL \$hhhh	0E	3	6
Absolute, X	ASL \$hhhh,X	1E	3	7

Bit 7 of the operand is shifted into the carry flag. Bits 6 through 0 are shifted into bits 7 through 1, respectively. A 0 is shifted into the low bit. If the result is negative, the N flag is set to 1; otherwise it is cleared to 0. If the result is zero, the Z bit is set to 1; otherwise it is cleared to 0.

**Example:** This program uses the ASL instruction to move luminances (0 through 7) into the high nybble of a byte so that they can be placed in the background register.

. 2000 A2 00	LDX #\$00	.X will store the current luminance.
. 2002 86 FF	STX \$FF	\$FF is a temporary location for the shifting.
. 2004 06 FF	ASL \$FF	Shift the luminance left four times
. 2006 06 FF	ASL \$FF	to get it into correct position

. 2008 06 FF	ASL \$FF	for putting in the background color register.
. 200A 06 FF	ASL \$FF	Get value in background register.
. 200C AD 15 FF	LDA \$FF15	Clear the luminance bits.
. 200F 29 0F	AND #\$0F	Put in the new lumiance information.
. 2011 05 FF	ORA \$FF	Put value in background register.
. 2013 8D 15 FF	STA \$FF15	Get low byte of jiffy clock timer.
. 2016 A5 A5	LDA \$A5	Look at bit 3.
. 2018 29 08	AND #\$08	If still zero, look again.
. 201A F0 FA	BEQ \$2016	Get low byte of jiffy clock timer.
. 201C A5 A5	LDA \$A5	Look at bit 3.
. 201E 29 08	AND #\$08	If still one, look again.
. 2020 D0 FA	BNE \$201C	Increment the current luminance.
. 2022 E8	INX	See if finished.
. 2023 E0 08	CPX #\$08	If not, go on.
. 2025 D0 DB	BNE \$2002	Stop execution.
. 2027 00	BRK	

Execute the program with G 2000.

### BCC—Branch If Carry Flag Is Clear

This instruction examines the current status of the carry bit. If it is cleared to 0, the branch occurs. If it is 1, the branch does not occur, and execution continues with the following instruction.

Operation: Branch on C = 0	Addressing Mode	Syntax	Flags Affected: None	Opcode	Bytes	Cycles
Relative		BCC \$hhhh		90	2	2*

\* Add 1 cycle when the branch is taken. Add 2 cycles when it is taken across a page boundary.

**Example:** When adding a 1-byte number to a 2-byte number, you can use standard double precision addition, but the operation may be speeded up by using a BCC to determine when to increment the high byte. In this example, the 8-bit number stored at \$2100 is added to the 16-bit number stored at \$2101-\$2102 (high byte to low byte), and the result is stored in \$2101-\$2102.

. 2000 18	CLC	Clear carry flag prior to addition.
. 2001 AD 00 21	LDA \$2100	Get single byte to add.
. 2004 6D 02 21	ADC \$2102	Add to low byte of \$2101-\$2102.
. 2007 8D 02 21	STA \$2102	Store result.
. 200A 90 03	BCC \$200F	If carry is clear, branch to finish.
. 200C EE 01 21	INC \$2101	Otherwise, increment the high byte.
. 200F 00	BRK	Stop execution.

Use the M (Memory) command to examine and modify the contents of \$2100–\$2102 before executing the program (using G 2000). Then check the contents of \$2100–\$2102 after the program executes. The values in \$2101–\$2102 are the sum of their earlier values and the value in \$2100.

## BCS—Branch If Carry Flag Is Set

This instruction examines the current status of the carry bit. If it is set to 1, the branch occurs. If it is 0, the branch does not occur, and execution continues with the following instruction.

<i>Operation: Branch on C = 1</i>	<i>Flags Affected: None</i>			
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Relative	BCS \$hhhh	B0	2	2*

\* Add 1 cycle when the branch is taken. Add 2 cycles when it is taken across a page boundary.

**Example:** When subtracting a 1-byte number from a 2-byte number, you can use standard double precision subtraction, but the operation may be speeded up by using a BCS to determine when to decrement the high byte. In this example, the 8-bit number stored at \$2100 is subtracted from the 16-bit number stored at \$2101–\$2102 (high byte to low byte), and the result is stored in \$2101–\$2102.

- . 2000 38 SEC Set carry flag prior to subtraction.
- . 2001 AD 02 21 LDA \$2102 Get low byte of \$2101–\$2102.
- . 2004 ED 00 21 SBC \$2100 Subtract single byte.
- . 2007 8D 02 21 STA \$2102 Store the result.
- . 200A B0 03 BCS \$200F If carry set, branch to finish.
- . 200C CE 01 21 DEC \$2101 Otherwise, decrement the high byte.
- . 200F 00 BRK Stop execution.

Use the M (Memory) command to examine and modify the contents of \$2100–\$2102 before executing the program (using G 2000). Then check the contents of \$2100–\$2102 after the program executes. The values in \$2101–\$2102 are the difference between their earlier values and the value in \$2100.

## BEQ—Branch If Zero Flag Is Set

This instruction examines the current status of the zero flag bit. If it is set to 1, the branch occurs. If it is 0, the branch does not occur, and execution continues with the following instruction.

Operation: Branch on Z = 1		Flags Affected: None		
Addressing Mode	Syntax	Opcode	Bytes	Cycles
Relative	BEQ \$hhhh	F0	2	2*

\* Add 1 cycle when the branch is taken. Add 2 cycles when it is taken across a page boundary.

**Example:** In this example, the BEQ instruction is used to cause the program to continue looping until a key is hit.

- . 2000 20 E4 FF JSR \$FFE4 Call the ROM subroutine to look for input.
- . 2003 F0 FB BEQ \$2000 If zero is returned, then there is no input, so look again.
- . 2005 00 BRK Stop processing.

## BIT—Test Bits in Memory with Accumulator

Bits 6 and 7 of the operand are transferred to the V and N flags. The values in the accumulator and the operand are logically ANDed, and a zero result is indicated in the Z flag. Neither the value in the accumulator nor in the operand is altered.

Operation: A $\wedge$ M		Flags Affected: N, Z, V		
$M7 \rightarrow N$				
$M6 \rightarrow V$				
Addressing Mode	Syntax	Opcode	Bytes	Cycles
Zero page	BIT \$hh	24	2	3
Absolute	BIT \$hhhh	2C	3	4

If bit 7 is set to 1 in the specified memory location, then the N flag is set to 1; otherwise it is cleared to 0. If bit 6 is set to 1 in the specified memory location, then the V flag is set to 1; otherwise it is cleared to 0. If the result of ANDing the value in the accumulator with the value in the specified memory location is 0, the Z flag is set to 1; otherwise it is cleared to 0. The values stored in the accumulator and the memory location are not changed.

**Example:** The BIT instruction is particularly useful when information is required about some memory location, but the value in the accumulator must be preserved. The example counts the number of bits set to 1 in the value stored at \$2100. The accumulator keeps track of which bit it is on.

- . 2000 A2 00 LDX #\$00 .X will contain the bit count.

. 2002 A9 01	LDA #\$01	Start looking at bit 0.
. 2004 2C 00 21	BIT \$2100	See if this bit is set in \$2100.
. 2007 F0 01	BEQ \$200A	If not, go on to next bit.
. 2009 E8	INX	Increment the bit count.
. 200A 0A	ASL	Shift left to the next bit.
. 200B 90 F7	BCC \$2004	When carry is set, then program is done.
. 200D 00	BRK	Stop execution.

The number of bits set to 1 in \$2100 is in the X register when the BRK instruction is executed. The contents of the registers are displayed when a BRK is executed by the BRK instruction processor.

### BMI—Branch If Negative Flag Is Set

This instruction examines the current status of the negative flag bit. If it is set to 1, the branch occurs. If it is 0, the branch does not occur, and execution continues with the following instruction.

<i>Operation: Branch on N = 1</i> <i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: None</i>		
		<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Relative	BMI \$hhhh	30	2	2*

\* Add 1 cycle when the branch is taken. Add 2 cycles when it is taken across a page boundary.

**Example:** In this program, the locations from \$2100 to \$217F that hold a nonnegative number are counted. The BMI instruction is used to determine when a negative number is encountered (it is not counted).

. 2000 A0 00	LDY #\$00	.Y counts the nonnegative numbers.
. 2002 A2 80	LDX #\$80	.X contains the pointer to the current location.
. 2004 BD 80 20	LDA \$2080,X	Get value of next location.
. 2007 30 01	BMI \$200A	If it is negative, do not count it.
. 2009 C8	INY	Count a nonnegative location.
. 200A E8	INX	Bump the pointer.
. 200B D0 F7	BNE \$2004	If not finished, keep going.
. 200D 00	BRK	Stop processing.

The count is displayed as the contents of the Y register when the BRK instruction is executed.

## BNE—Branch If Zero Flag Is Clear

This instruction examines the current status of the zero flag bit. If it is cleared to 0, the branch occurs. If it is 1, the branch does not occur, and execution continues with the following instruction.

<i>Operation: Branch on Z = 0</i>		<i>Flags Affected: None</i>		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Relative	BNE \$hhhh	D0	2	2*

\* Add 1 cycle when the branch is taken. Add 2 cycles when it is taken across a page boundary.

**Example:** In this program, the contents of \$2100-\$21FF are copied into \$2200-\$22FF. The BNE instruction is used to continue looping until the X register reaches zero.

- . 2000 A2 00 LDX #\$00 Start counter at zero.
- . 2002 BD 00 21 LDA \$2100,X Get value from original area.
- . 2005 9D 00 22 STA \$2200,X Put value in destination area.
- . 2008 CA DEX Decrement counter.
- . 2009 D0 F7 BNE \$2002 If not done, go back.
- . 200B 00 BRK Stop processing.

Use the M (Memory) command to examine or change the contents of the memory areas.

## BPL—Branch If Negative Flag Is Clear

This instruction examines the current status of the negative flag bit. If it is cleared to 0, the branch occurs. If it is 1, the branch does not occur, and execution continues with the following instruction.

<i>Operation: Branch on N = 0</i>		<i>Flags Affected: None</i>		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Relative	BPL \$hhhh	10	2	2*

\* Add 1 cycle when the branch is taken. Add 2 cycles when it is taken across a page boundary.

**Example:** This program copies the contents of the color registers (\$FF15-\$FF19) into memory at \$2100-\$2104. The BPL instruction is used to determine when the process is complete.

- . 2000 A2 04 LDX #\$04 Start counter at highest byte to copy.
- . 2002 BD 15 FF LDA \$FF15,X Get register value.
- . 2005 9D 00 21 STA \$2100,X Store in memory.
- . 2008 CA DEX Decrement the counter.
- . 2009 10 F7 BPL \$2002 If not done, continue.
- . 200B 00 BRK Stop processing.

## BRK—Force an Interrupt

An interrupt occurs, the values of the PC and SR are pushed on the stack, and processing continues through the IRQ vector.

<i>Operation:</i> PC + 2 ↓		<i>Flags Affected:</i> B		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	BRK	00	1	7

The B flag is set to 1 before the status register is pushed onto the stack, and then is cleared to 0. Hence, an interrupt service routine may examine the contents of the stack to determine if the interrupt was caused by a BRK instruction. The built-in interrupt service routine pushes the three user registers (.A, .X, and .Y in that order) onto the stack, checks for a break instruction, and indirectly jumps through the BRK instruction vector at \$0316–\$0317 when a BRK is executed. Trapping a BRK instruction is accomplished most easily by altering this vector to point to your routine.

It should be noted that the interrupt caused by a BRK instruction can NOT be masked by setting the interrupt disable (I) flag.

**Example:** This example alters the BRK instruction vector. Before executing it, check the default contents of \$0316–\$0317 with the M (Memory) command. The low byte of the default address for BRK processing is located in \$0316, and the high byte in \$0317. If this address is not \$F44C, replace the \$F44C in the program with it.

- . 2000 A9 0B LDA #\$0B Load the low byte of the address of new BRK processor.
- . 2002 8D 16 03 STA \$0316 Store in vector.
- . 2005 A9 20 LDA #\$20 Load the high byte of the address of new BRK processor.
- . 2007 8D 17 03 STA \$0317 Store in vector.

- . 200A 00 BRK Execute a BRK instruction (and stop processing).
- . 200B EE 19 FF INC \$FF19 Increment the border color.
- . 200E 4C 4C F4 JMP \$F44C Then jump to normal BRK processor.

After you execute this program (by typing G 2000), the execution of a BRK instruction will increment the border color before proceeding normally. When you are finished with this example, reset the computer by typing G FFF6 so as to restore the BRK instruction vector to its default value.

### BVC—Branch If Overflow Flag Is Clear

This instruction examines the current status of the overflow flag bit. If it is cleared to 0, the branch occurs. If it is 1, the branch does not occur, and execution continues with the following instruction.

<i>Operation: Branch on V = 0</i>	<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Flags Affected: None</i>	<i>Cycles</i>
Relative		BVC \$hhhh	50	2		2*

\* Add 1 cycle when the branch is taken. Add 2 cycles when it is taken across a page boundary.

**Example:** This program adds the contents of \$2100 to the contents of \$2101 and stores the result in \$2102. If a two's complement overflow occurs, the border color is incremented.

- . 2000 18 CLC Prepare for addition.
- . 2001 AD 00 21 LDA \$2100 Get the first value.
- . 2004 6D 01 21 ADC \$2101 Add the second value.
- . 2007 8D 02 21 STA \$2102 Store the result.
- . 200A 50 03 BVC \$200F Branch if no overflow occurred.
- . 200C EE 19 FF INC \$FF19 Increment the border color.
- . 200F 00 BRK Stop processing.

Examine and change the contents of \$2100–\$2102 to experiment with two's complement overflows.

### BVS—Branch If Overflow Flag Is Set

This instruction examines the current status of the overflow flag bit. If it is set to 1, the branch occurs. If it is 0, the branch does not occur, and execution continues with the following instruction.

<i>Operation: Branch on V = 1</i>	<i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: None</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Relative		BVS \$hhhh		70	2	2*

\* Add 1 cycle when the branch is taken. Add 2 cycles when it is taken across a page boundary.

**Example:** This example uses the BIT instruction to transfer the status of bit 6 of the low byte of the jiffy clock to the overflow flag. First, the program waits until this bit is clear, then until it is set. An increment of the border color signals the end of the wait.

- . 2000 24 A5      BIT \$A5      Transfer bit 6 of jiffy clock to overflow flag.
- . 2002 70 FC      BVS \$2000      Continue waiting if the bit is set.
- . 2004 EE 19 FF    INC \$FF19      Increment the border color.
- . 2007 24 A5      BIT \$A5      Transfer bit 6 of jiffy clock to overflow flag.
- . 2009 50 FC      BVC \$2007      Continue waiting if the bit is clear.
- . 200B EE 19 FF    INC \$FF19      Increment the border color.
- . 200E 00           BRK           Stop processing.

## CLC—Clear the Carry Flag

The carry flag is cleared to 0.

<i>Operation: 0 → C</i>	<i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: C</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied		CLC		18	1	2

**Example:** This program adds the contents of \$2100 to the contents of \$2101 and stores the result in \$2102. Because the ADC instruction adds the value of the carry to the sum of the accumulator and the operand, the CLC instruction is used to clear the carry flag before the addition. This ensures that the result is exactly the first value plus the second value.

- . 2000 18           CLC           Clear the carry flag to prepare to add.
- . 2001 AD 00 21    LDA \$2100      Get the first value.
- . 2004 6D 01 21    ADC \$2101      Add the second value.
- . 2007 8D 02 21    STA \$2102      Store the result.
- . 200A 00           BRK           Stop processing.

Use the M (Memory) command to examine and change the contents of \$2100–\$2102.

## CLD—Clear Decimal Mode

The processor is put in normal hexadecimal mode.

<i>Operation: 0 → D</i>			<i>Flags Affected: D</i>	
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	CLD	D8	1	2

See the section on decimal mode.

**Example:** This program places the processor into decimal mode and adds the contents of \$2100 and \$2101. The result is placed in \$2102. Decimal mode is terminated using the CLD instruction.

- |                 |            |                                      |
|-----------------|------------|--------------------------------------|
| . 2000 F8       | SED        | Put the processor into decimal mode. |
| . 2001 18       | CLC        | Prepare to add.                      |
| . 2002 AD 00 21 | LDA \$2100 | Get the first decimal value.         |
| . 2005 6D 01 21 | ADC \$2101 | Add the second decimal value.        |
| . 2008 8D 02 21 | STA \$2102 | Store the result.                    |
| . 200B D8       | CLD        | Exit decimal mode.                   |
| . 200C 00       | BRK        | Stop processing.                     |

Use the M (Memory) command to examine and change the contents of \$2100–\$2102 to experiment with decimal mode addition.

## CLI—Clear Interrupt Disable Flag

The interrupt disable flag is cleared, which allows interrupts to occur.

<i>Operation: 0 → I</i>			<i>Flags Affected: I</i>	
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	CLI	58	1	2

**Example:** This example alters the IRQ vector. Before executing it, check the default contents of \$0314 \$0315 with the M (Memory) command. The low byte of the default address for IRQ processing is located in \$0314 and the high byte in \$0315. If this address is not \$CE0E, replace the \$CE0E in the program with it.

- |           |     |                          |
|-----------|-----|--------------------------|
| . 2000 78 | SEI | Disables the interrupts. |
|-----------|-----|--------------------------|

. 2001 A9 0D	LDA #\$0D	Get the low byte of the address of the new IRQ processor.
. 2003 8D 14 03	STA \$0314	Store in the vector.
. 2006 A9 20	LDA #\$20	Get the high byte of the address of the new IRQ processor.
. 2008 8D 15 03	STA \$0315	Store in the vector.
. 200B 58	CLI	Reenable the interrupts.
. 200C 00	BRK	Stop processing.
. 200D EE 19 FF	INC \$FF19	Increment the border color.
. 2010 4C 0E CE	JMP \$CEOE	Jump to normal IRQ processing.

After you execute this program (by typing G 2000), the processing of an interrupt increments the border color before proceeding normally. When you are finished with this example, reset the computer by typing G FFF6 so as to restore the IRQ vector to its default value.

### CLV—Clear Overflow Flag

The overflow flag is cleared to 0.

<i>Operation: 0 → V</i>	<i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: V</i>		
			<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied		CLV	B8	1	2

**Example:** This program uses the overflow flag to perform an unconditional branch. Such a branch may be used instead of a jump instruction when code must be relocatable.

. 2000 B8	CLV	Clear the overflow flag.
. 2001 50 01	BVC \$2004	This branch is always taken.
. 2003 00	BRK	This is never executed.
. 2004 EE 19 FF	INC \$FF19	Increment the border color.
. 2007 00	BRK	Stop processing.

Unless a JMP or branch instruction elsewhere starts processing at \$2001, the branch there is unconditional (always taken).

### CMP—CMP Memory with Accumulator

The value currently in the accumulator is compared with the specified operand and the appropriate flags are set.

Operation: A - M Addressing Mode	Syntax	Flags Affected: N, Z, C		
		Opcode	Bytes	Cycles
Immediate	CMP #\$hh	C9	2	2
Zero page	CMP \$hh	C5	2	3
Zero page, X	CMP \$hh,X	D5	2	4
Absolute	CMP \$hhhh	CD	3	4
Absolute, X	CMP \$hhhh,X	DD	3	4*
Absolute, Y	CMP \$hhhh,Y	D9	3	4*
(Indirect, X)	CMP (\$hh,X)	C1	2	6
(Indirect), Y	CMP (\$hh),Y	D1	2	5*

\* Add 1 when a page boundary is crossed.

The value in memory is subtracted from the value in the accumulator, but neither value is altered. The following table shows the effect on each of the flags. The N flag should be used when a two's complement compare is required (e.g., \$FF is minus 1 and less than \$01). The C flag should be used when an unsigned compare is required (e.g., \$FF is 255 and greater than \$01).

Condition	N Flag	Z Flag	C Flag
A < memory	1	0	0
A = memory	0	1	1
A > memory	0	0	1

**Example:** In this program, the CMP instruction is used to compare a value in the accumulator with the current raster line (register \$FF1D). When a match is found, the background color is changed. The routine is executed 256 times by using the X register to count.

- . 2000 A2 00      LDX #\$00      Start the counter at zero.
- . 2002 A9 30      LDA #\$30      Set the accumulator to the upper raster value desired.
- . 2004 CD 1D FF    CMP \$FF1D    Compare with current raster line.
- . 2007 DO FB        BNE \$2004    Go back if it is not equal.
- . 2009 A9 70        LDA #\$70      Get the value for the color black.
- . 200B 8D 15 FF    STA \$FF15      Store it in the background color register.
- . 200E A9 50        LDA #\$50      Set the accumulator to the lower raster value desired.
- . 2010 CD 1D FF    CMP \$FF1D    Compare with current raster line.
- . 2013 DO FB        BNE \$2010    Go back if it is not equal.

- 2015 EE 15 FF INC \$FF15 Increment the background color to white.
- 2018 CA DEX Decrement the counter.
- 2019 DO E7 BNE \$2002 If not zero, do it again.
- 201B OO BRK Stop processing.

The values in \$2003 and \$200F may be changed to allow experimentation with the raster line values.

### CPX—Compare Memory with X Register

The value currently in the X register is compared with the specified operand and the appropriate flags are set.

<i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: N, Z, C</i>		
		<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Immediate	CPX #\$hh	E0	2	2
Zero page	CPX \$hh	E4	2	3
Absolute	CPX \$hhhh	EC	3	4

The value in memory is subtracted from the value in the X register, but neither value is altered. The following table shows the effect on each of the flags. The N flag should be used when a two's complement compare is required (e.g., \$FF is minus 1 and less than \$01). The C flag should be used when an unsigned compare is required (e.g., \$FF is 255 and greater than \$01).

<i>Condition</i>	<i>N Flag</i>	<i>Z Flag</i>	<i>C Flag</i>
X < memory	1	0	0
X = memory	0	1	1
X > memory	0	0	1

**Example:** This program copies the contents of \$2100–\$210F to \$2110–\$211F. The X register is initialized to \$10 and incremented until it reaches \$1F. The CPX instruction is used to determine this.

- 2000 A2 10 LDX #\$10 Initialize .X to \$10.
- 2002 BD F0 20 LDA \$20F0,X Get the value in originating location.
- 2005 9D 00 21 STA \$2100,X Store the value in destination location.

- . 2008 E8 INX Increment .X by 1.
- . 2009 E0 20 CPX #\$20 Compare .X to \$20.
- . 200B 90 F5 BCC \$2002 If .X is less than \$20, go back.
- . 200D 00 BRK Stop processing.

Use the M (Memory) command to examine and change the values in \$2100–\$211F.

### CPY—Compare Memory with Y Register

The value currently in the Y register is compared with the specified operand and the appropriate flags are set.

<i>Operation: Y - M Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: N, Z, C</i>		
		<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Immediate	CPY #\$hh	C0	2	2
Zero page	CPY \$hh	C4	2	3
Absolute	CPY \$hhhh	CC	3	4

The value in memory is subtracted from the value in the Y register, but neither value is altered. The following table shows the effect on each of the flags. The N flag should be used when a two's complement compare is required (e.g., \$FF is minus 1 and less than \$01). The C flag should be used when an unsigned compare is required (e.g., \$FF is 255 and greater than \$01).

<i>Condition</i>	<i>N Flag</i>	<i>Z Flag</i>	<i>C Flag</i>
Y < memory	1	0	0
Y = memory	0	1	1
Y > memory	0	0	1

**Example:** This program copies the contents of \$2100–\$210F to \$2110–\$211F. The Y register is used as an indirect index and an index. The CPY instruction is used to determine when to stop copying.

- . 2000 A9 00 LDA #\$00 Load .A with zero.
- . 2002 A8 TAY Load .Y with zero.
- . 2003 85 D8 STA \$D8 Store zero in the low indirect address.
- . 2005 A9 21 LDA #\$21 Load .A with \$21.
- . 2007 85 D9 STA \$D9 Store \$21 in the high indirect address.

- . 2009 B1 D8      LDA (\$D8),Y      Get the contents of the originating location.
- . 200B 99 10 21 STA \$2110,Y      Store in the destination location.
- . 200E C8      INY      Increment .Y by 1.
- . 200F C0 10      CPY #\$10      Compare .Y with \$10.
- . 2011 90 F6      BCC \$2009      If .Y is less than \$10, go back.
- . 2013 00      BRK      Stop processing.

Use the M (Memory) command to examine and change the values in \$2100–\$211F.

## DEC—Decrement Memory by 1

The value currently in the operand is decremented by 1.

<i>Operation: M - 1 → M</i> <i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: N, Z</i>		
		<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Zero page	DEC \$hh	C6	2	5
Zero page, X	DEC \$hh,X	D6	2	6
Absolute	DEC \$hhhh	CE	3	6
Absolute, X	DEC \$hhhh,X	DE	3	7

The value stored in the operand is decreased by 1. If the result is negative, the N flag is set to 1; otherwise it is cleared to 0. If the result is zero, the Z flag is set to 1; otherwise it is cleared to 0. It should be noted that the carry flag is NOT affected. That is, a decrement from \$00 to \$FF does NOT result in the carry flag being cleared.

**Example:** This program searches the operating system ROM for the last occurrence of an \$EA (the opcode for a NOP). Indirect indexed addressing is used, and the DEC instruction updates the indirect address.

- . 2000 A9 00      LDA #\$00      Load .A with a zero.
- . 2002 85 D8      STA \$D8      Store zero in the low byte of indirect address.
- . 2004 A8      TAY      Store zero in .Y.
- . 2005 A9 FF      LDA #\$FF      Load .A with \$FF.
- . 2007 85 D9      STA \$D9      Store \$FF in the high byte of indirect address.
- . 2009 A9 EA      LDA #\$EA      .A is the value to look for.
- . 200B D1 D8      CMP (\$D8),Y      Examine the current location.

. 200D F0 08	BEQ \$2017	If the desired value is found, quit.
. 200F C6 D8	DEC \$D8	Decrement the low byte of the indirect address by 1.
. 2011 DO F8	BNE \$200B	If not zero, go back.
. 2013 C6 D9	DEC \$D9	When the low byte is zero, decrement the high byte.
. 2015 DO F4	BNE \$200B	If not zero, go back.
. 2017 00	BRK	Stop processing.

The address of the last occurrence of an \$EA is stored in \$D8–\$D9 following the execution of this program.

## DEX—Decrement the X Register by 1

The value currently in the X register is decremented by 1.

<i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: N, Z</i>		
		<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	DEX	CA	1	2

The value stored in the X register is decreased by 1. If the result is negative, the N flag is set to 1; otherwise it is cleared to 0. If the result is zero, the Z flag is set to 1; otherwise it is cleared to 0.

**Example:** This program copies contents of zero page into \$2100–\$21FF. The X register is used as the index. The DEX instruction is used to update the index value.

. 2000 A2 00	LDX #\$00	Initialize .X to zero.
. 2002 B5 00	LDA \$00,X	Get a value from zero page.
. 2004 9D 00 21	STA \$2100,X	Store the value in \$2100–\$21FF.
. 2007 CA	DEX	Decrement the index by 1.
. 2008 DO F8	BNE \$2002	If not equal to zero, go back.
. 200A 00	BRK	Stop processing.

The operating system uses many zero page locations, so using an M (Memory) command to examine zero page from the monitor may not reveal the contents of zero page while a program is executing. A routine similar to the preceding one can be used in a program to transfer the contents of zero page to a safe location prior to returning to the monitor.

## DEY—Decrement the Y Register by 1

The value currently in the Y register is decremented by 1.

<i>Operation: Y - 1 → Y</i>	<i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: N, Z</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied		DEY		88	1	2

The value stored in the Y register is decreased by 1. If the result is negative, the N flag is set to 1; otherwise it is cleared to 0. If the result is zero, the Z flag is set to 1; otherwise it is cleared to 0.

**Example:** In this program the contents of \$2100-\$21FF are copied onto the screen at \$0C00-\$0CFF. The Y register is used for indirect indexed addressing, and the DEY instruction updates the index value.

. 2000 A9 00	LDA #\$00	Load .A with zero.
. 2002 85 D8	STA \$D8	Store zero in the low byte of origination address.
. 2004 85 DA	STA \$DA	Store zero in the low byte of destination address.
. 2006 A8	TAY	Initialize .Y at zero.
. 2007 A9 21	LDA #\$21	Load .A with \$21.
. 2009 85 D9	STA \$D9	Store \$21 in the high byte of origination address.
. 200B A9 OC	LDA #\$0C	Load .A with \$0C.
. 200D 85 DB	STA \$DB	Store \$0C in the high byte of destination address.
. 200F B1 D8	LDA (\$D8),Y	Get the value of the originating byte.
. 2011 91 DA	STA (\$DA),Y	Store in the destination byte.
. 2013 88	DEY	Decrement .Y by 1.
. 2014 D0 F9	BNE \$200F	If not zero, continue.
. 2016 00	BRK	Stop processing.

Use the M (Memory) command to put the desired screen code values in \$2100-\$21FF (see Appendix E for screen codes).

## EOR—Exclusive-OR Memory with Accumulator

The value currently in the accumulator is logically exclusive-ORed to the specified operand, and the result is placed in the accumulator.

Operation: $A \vee M \rightarrow A$		Flags Affected: N, Z		
Addressing Mode	Syntax	Opcode	Bytes	Cycles
Immediate	EOR #\$hh	49	2	2
Zero page	EOR \$hh	45	2	3
Zero page, X	EOR \$hh,X	55	2	4
Absolute	EOR \$hhh	4D	3	4
Absolute, X	EOR \$hhh,X	5D	3	4*
Absolute, Y	EOR \$hhh,Y	59	3	4*
(Indirect, X)	EOR (\$hh,X)	41	2	6
(Indirect), Y	EOR (\$hh),Y	51	2	5*

\* Add 1 when a page boundary is crossed.

Those bits that are set to 1 in the value in the accumulator or set to 1 in the operand, but not both, are set to 1 in the result. Bits that are cleared to 0 in both values or set to 1 in both values are cleared to 0 in the result. If the result has the high bit set to 1, the N flag is set to 1; otherwise it is cleared to 0. If the result is zero, the Z bit is set to 1; otherwise it is cleared to 0.

**Example:** This program flashes the upper left character on the screen by alternating between the character residing there and its reverse. The EOR instruction alternately sets and clears the high bit to produce this result.

- . 2000 A2 00      LDX #\$00      .X counts 256 flashes.
- . 2002 AD 00 OC    LDA \$OC00      Get the character in the upper left corner.
- . 2005 49 80      EOR #\$80      Toggle the high bit.
- . 2007 8D 00 OC    STA \$OC00      Put the new character in the upper left corner.
- . 200A A5 A5      LDA \$A5      Now wait. Get the low byte of jiffy clock.
- . 200C 29 02      AND #\$02      Look at bit 1.
- . 200E D0 FA      BNE \$200A      Wait until it is off.
- . 2010 A5 A5      LDA \$A5      Get the low byte of jiffy clock.
- . 2012 29 02      AND #\$02      Look at bit 1.
- . 2014 F0 FA      BEQ \$2010      Wait until it is on.
- . 2016 CA          DEX          Decrement the flash counter.
- . 2017 D0 E9      BNE \$2002      If not done, continue.
- . 2019 00          BRK          Stop processing.

The speed of the flashing can be changed by changing which bit of the jiffy clock is examined at \$200C and \$2012.

## INC—Increment Memory by 1

The value currently in the operand is incremented by 1.

<i>Addressing Mode</i>	<i>Operation: M + 1 → M</i>	<i>Flags Affected: N, Z</i>		
	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Zero page	INC \$hh	E6	2	5
Zero page, X	INC \$hh,X	F6	2	6
Absolute	INC \$hhhh	EE	3	6
Absolute, X	INC \$hhhh,X	FE	3	7

The value stored in the operand is increased by 1. If the result is negative, the N flag is set to 1; otherwise it is cleared to 0. If the result is zero, the Z flag is set to 1; otherwise it is cleared to 0. It should be noted that the carry flag is NOT affected. That is, an increment from \$FF to \$00 does NOT result in the carry flag being set.

**Example:** The upper left screen location is cycled through all of the available screen codes. The INC instruction is used to update the character.

- . 2000 A2 00      LDX #\$00      .X counts the 256 screen codes.
- . 2002 EE 00 OC    INC \$OC00      Increment the upper left screen code by 1.
- . 2005 A5 A5      LDA \$A5      Now wait. Get the value in the jiffy clock.
- . 2007 29 02      AND #\$02      Look at bit 1.
- . 2009 D0 FA      BNE \$2005      Wait until it is off.
- . 200B A5 A5      LDA \$A5      Get the value in the jiffy clock.
- . 200D 29 02      AND #\$02      Look at bit 1.
- . 200F F0 FA      BEQ \$200B      Wait until it is on.
- . 2011 CA          DEX          Decrement the counter.
- . 2012 D0 EE      BNE \$2002      If not done, continue.
- . 2014 00          BRK          Stop processing.

The speed of the updating can be changed by changing which bit of the jiffy clock is examined at \$2007 and \$200D.

**INX—Increment the X Register by 1**

The value currently in the X register is incremented by 1.

<i>Operation:</i> $X + 1 \rightarrow X$	<i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected:</i> N, Z	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied		INX		E8	1	2

The value stored in the X register is increased by 1. If the result is negative, the N flag is set to 1; otherwise it is cleared to 0. If the result is zero, the Z flag is set to 1; otherwise it is cleared to 0.

**Example:** This program counts the number of times the value \$EA (the opcode for a NOP) occurs in the operating system ROM. The X register is used to count and the INX instruction is used to update it.

. 2000 A9 00	LDA #\$00	Load .A with zero.
. 2002 85 D8	STA \$D8	Store zero in the low byte of the address.
. 2004 AA	TAX	Store zero in .X, the count register.
. 2005 A8	TAY	Store zero in .Y, the index register.
. 2006 A9 80	LDA #\$80	Load .A with \$80.
. 2008 85 D9	STA \$D9	Store \$80 in the high byte of the address.
. 200A B1 D8	LDA (\$D8),Y	Get the value at the current address.
. 200C C9 EA	CMP #\$EA	Compare with \$EA.
. 200E D0 01	BNE \$2011	If not equal, go on.
. 2010 E8	INX	Increment the count register.
. 2011 E6 D8	INC \$D8	Increment the low byte of the address.
. 2013 D0 F5	BNE \$200A	If not zero, go back for the next byte.
. 2015 E6 D9	INC \$D9	Increment the high byte of the address.
. 2017 D0 F1	BNE \$200A	If not zero, go back for the next byte.
. 2019 00	BRK	Stop processing.

When the BRK instruction is executed, the count in the X register is displayed.

## INY—Increment the Y Register by 1

The value currently in the Y register is incremented by 1.

<i>Operation: <math>Y + 1 \rightarrow Y</math></i>	<i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: N, Z</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied		INY		C8	1	2

The value stored in the Y register is increased by 1. If the result is negative, the N flag is set to 1; otherwise it is cleared to 0. If the result is zero, the Z flag is set to 1; otherwise it is cleared to 0.

**Example:** In this example, the first byte in \$8000–\$80FF containing a value of \$EA is located. The Y register is used as the index, and the INY instruction updates it.

- . 2000 A0 00 LDY #\$00 Initialize .Y to zero.
- . 2002 A9 EA LDA #\$EA Load .A with the value to look for.
- . 2004 D9 00 80 CMP \$8000,Y Compare with the current location.
- . 2007 F0 03 BEQ \$200C If equal, quit.
- . 2009 C8 INY Update the index register.
- . 200A DO F8 BNE \$2004 If there are more locations to search, go back.
- . 200C 00 BRK Stop processing.

When the BRK instruction is executed, the value of the Y register when an \$EA was found (or a zero if none was found) is displayed.

## JMP—Jump to a New Location

The address specified in the operand is transferred to the program counter, and processing continues with the instruction located at that address.

<i>Operation: <math>(PC + 1) \rightarrow PCL</math></i>	<i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: None</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
	Absolute	JMP \$hhhh		4C	3	3
	Indirect	JMP (\$hhhh)		6C	3	5

Without affecting any of the flags, the processing control is passed to the specified address. This is the only instruction that has an indirect addressing mode. In this mode, the value at the specified address is used as the low byte of the destination address, and the value stored in the next location is used as the high byte of the destination address.

**Example:** In this program the background color is cycled through every possible value. The JMP instruction is used to jump to the reset routine. The reset routine restores the background color to its default value (and returns to BASIC).

. 2000 A9 00	LDA #\$00	Load .A with a zero.
. 2002 8D 15 FF	STA \$FF15	Store zero in the background color register.
. 2005 EE 15 FF	INC \$FF15	Increment the background color register.
. 2008 F0 OE	BEQ \$2018	If it has cycled around to zero, quit.
. 200A A5 A5	LDA \$A5	Now wait. Load the value in the jiffy clock.
. 200C 29 02	AND #\$02	Look at bit 1.
. 200E D0 FA	BNE \$200A	Wait until it is off.
. 2010 A5 A5	LDA \$A5	Load the value of the jiffy clock.
. 2012 29 02	AND #\$02	Look at bit 1.
. 2014 F0 FA	BEQ \$2010	Wait until it is on.
. 2016 D0 ED	BNE \$2005	Done waiting, go back.
. 2018 4C F6 FF	JMP \$FFF6	Transfer control to the reset routine.

## JSR—Jump to Subroutine

The current value of the program counter (plus 2) is pushed onto the stack for use as the return address. The address specified with the instruction is transferred to the program counter, and processing continues with the instruction located at that address.

Addressing Mode	Syntax	Opcode	Bytes	Cycles
Absolute	JSR \$hhhh	20	3	6

The processor saves the return address on the stack. The RTS instruction pulls these bytes off the stack. A JSR instruction should have a corresponding RTS.

None of the flags is affected. Hence, they may be used to transfer information between the main program and the subroutine.

**Example:** This program cycles the two upper left character locations on the screen through all possible values. They are changed alternately, with a brief wait in between. The wait routine is a subroutine and is called using the JSR instruction.

. 2000 A2 00	LDX #\$00	.X counts all the possible values.
. 2002 EE 00 OC	INC \$0C00	Next character for the first location.
. 2005 20 12 20	JSR \$2012	Call the wait subroutine.
. 2008 EE 01 OC	INC \$0C01	Next character for the second location.
. 200B 20 12 20	JSR \$2012	Call the wait subroutine.
. 200E CA	DEX	Decrement the counter.
. 200F DO F1	BNE \$2002	If not done, go back.
. 2011 00	BRK	Stop processing.
. 2012 A5 A5	LDA \$A5	Start of subroutine. Load the jiffy clock.
. 2014 29 08	AND #\$08	Look at bit 3.
. 2016 DO FA	BNE \$2012	Wait until it is off.
. 2018 A5 A5	LDA \$A5	Load the value of the jiffy clock.
. 201A 29 08	AND #\$08	Look at bit 3.
. 201C FO FA	BEQ \$2018	Wait until it is on.
. 201E 60	RTS	Return from the subroutine.

### LDA—Load Accumulator with Value from Memory

The value currently in the location specified by the operand is placed in the accumulator.

Operation: $M \rightarrow A$ Addressing Mode	Syntax	Flags Affected: N, Z		
		Opcode	Bytes	Cycles
Immediate	LDA #\$hh	A9	2	2
Zero page	LDA \$hh	A5	2	3
Zero page, X	LDA \$hh,X	B5	2	4
Absolute	LDA \$hhhh	AD	3	4
Absolute, X	LDA \$hhhh,X	BD	3	4*
Absolute, Y	LDA \$hhhh,Y	B9	3	4*
(Indirect, X)	LDA (\$hh,X)	A1	2	6
(Indirect), Y	LDA (\$hh),Y	B1	2	5*

\* Add 1 when a page boundary is crossed.

If the value transferred is negative, the N flag is set to 1; otherwise it is cleared to 0. If the value transferred is zero, the Z flag is set to 1; otherwise it is cleared to 0. The value in the operand is not altered.

**Example:** The characters on the top of the line of the screen are copied to the second line of the screen. Each value is transferred by loading it into the accumulator with the LDA instruction and storing it to the destination.

- . 2000 A2 27      LDX #\$27      .X indexes the line.
- . 2002 BD 00 OC    LDA \$OC00,X   Load the accumulator with the next character.
- . 2005 9D 28 OC    STA \$OC28,X   Store on the second line.
- . 2008 CA           DEX             Decrement the index register.
- . 2009 10 F7        BPL \$2002    If there is more to do, go back.
- . 200B 00            BRK            Stop processing.

If the BRK instruction causes the screen to scroll, the result of the program is not seen. To avoid this, clear the screen before you execute the program.

### LDX—Load X Register with Value from Memory

The value currently in the location specified by the operand is placed in the X register.

<i>Operation: M → X</i> <i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: N, Z</i>		
		<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Immediate	LDX #\$hh	A2	2	2
Zero page	LDX \$hh	A6	2	3
Zero page, Y	LDX \$hh,Y	B6	2	4
Absolute	LDX \$hhhh	AE	3	4
Absolute, Y	LDX \$hhhh,Y	BE	3	4*

\* Add 1 when a page boundary is crossed.

If the value transferred is negative, the N flag is set to 1; otherwise it is cleared to 0. If the value transferred is zero, the Z flag is set to 1; otherwise it is cleared to 0. The value in the operand is not altered.

**Example:** This program copies each character on the first line of the screen one location to the right. The X register is used as the index and initialized using the LDX instruction.

- . 2000 A2 27      LDX #\$27      Initialize .X for indexing.
- . 2002 BD 00 OC    LDA \$0C00,X    Get the next character.
- . 2005 9D 01 OC    STA \$0C01,X    Store 1 byte to the right.
- . 2008 CA           DEX             Decrement the index register.
- . 2009 10 F7        BPL \$2002     If not done, go back.
- . 200B 00            BRK            Stop processing.

If the screen scrolls upon execution of the BRK instruction, the effect of the program will not be seen. To avoid this, clear the screen before executing the program.

### LDY—Load Y Register with Value from Memory

The value currently in the location specified by the operand is placed in the Y register.

<i>Operation: M → Y</i>		<i>Flags Affected: N, Z</i>		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Immediate	LDY #\$hh	A0	2	2
Zero page	LDY \$hh	A4	2	3
Zero page, X	LDY \$hh,X	B4	2	4
Absolute	LDY \$hhhh	AC	3	4
Absolute, X	LDY \$hhhh,X	BC	3	4*

\* Add 1 when a page boundary is crossed.

If the value transferred is negative, the N flag is set to 1; otherwise it is cleared to 0. If the value transferred is zero, the Z flag is set to 1; otherwise it is cleared to 0. The value in the operand is not altered.

**Example:** This program copies the characters on the first line of the screen to the second. The Y register is used as the index and initialized with the LDY instruction.

- . 2000 A9 OC      LDA #\$0C      Load .A with the high byte of the address.
- . 2002 85 D9      STA \$D9      Store in the high byte of the originating address.
- . 2004 85 DB      STA \$DB      Store in the high byte of the destination address.
- . 2006 A9 00      LDA #\$00      Load .A with the low byte of the first line address.

. 2008 85 D8	STA \$D8	Store in the low byte of the originating address.
. 200A A9 28	LDA #\$28	Load .A with the low byte of the second line address.
. 200C 85 DA	STA \$DA	Store in the low byte of the destination address.
. 200E A0 27	LDY #\$27	Initialize Y to move one line.
. 2010 B1 D8	LDA (\$D8),Y	Get the character from the first line.
. 2012 91 DA	STA (\$DA),Y	Put on the second line.
. 2014 88	DEY	Decrement the index register.
. 2015 10 F9	BPL \$2010	If not done, go on.
. 2017 00	BRK	Stop processing.

If the screen scrolls upon execution of the BRK instruction, the effect of the program will not be seen. To avoid this, clear the screen before executing the program.

### LSR—Shift Right One Bit

Each bit in the specified operand is shifted one bit to the right. The low bit is shifted into the carry flag, and a 0 is shifted into the high bit.

Operation: $0 \rightarrow [7]$ Addressing Mode	$0] \rightarrow C$ Syntax	Flags Affected: N, Z, C, Opcode Bytes Cycles
Accumulator	LSR	4A 1 2
Zero page	LSR \$hh	46 2 5
Zero page, X	LSR \$hh,X	56 2 6
Absolute	LSR \$hhhh	4E 3 6
Absolute, X	LSR \$hhhh,X	5E 3 7

Bit 0 of the operand is shifted into the carry flag. Bits 7 through 1 are shifted into bits 6 through 0, respectively. Since a 0 is shifted into the high bit, the result can never be negative. Hence, the N flag is always cleared to 0. If the result is zero, the Z bit is set to 1; otherwise it is cleared to 0.

**Example:** This program counts the number of bits set to 1 in memory location \$2100. The LSR instruction is used to shift the bits one by one into the carry bit for testing.

. 2000 A0 07	LDY #\$07	.Y counts the 8 bits.
. 2002 A2 00	LDX #\$00	.X counts the number of bits set to 1.
. 2004 AD 00 21	LDA \$2100	Get the byte to count.
. 2007 4A	LSR	Shift bit 0 into the carry.

. 2008 90 01	BCC \$200B	If clear, skip the next instruction.
. 200A E8	INX	Increment the count of bits set to 1.
. 200B 88	DEY	Decrement the bit count.
. 200C 10 F9	BPL \$2007	If not done, continue.
. 200E 00	BRK	Stop processing.

The number of bits set to 1 is displayed in the X register when the BRK instruction is executed.

## NOP—No Operation

No operation is performed.

<i>Operation: None</i>	<i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: None</i>		
			<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied		NOP	EA	1	2

The processor performs no operation. None of the flags are affected. This instruction is generally used as a placeholder.

**Example:** In this program a loop executes until a certain memory location is zero. In this case, a zero page location (the low byte of the jiffy clock) is used. A NOP instruction is placed just after the LDA instruction. This leaves room for testing a location not on zero page in a subsequent execution of the program.

. 2000 A5 A5	LDA \$A5
. 2002 EA	NOP
. 2003 D0 FB	BNE \$2000
. 2005 00	BRK

## ORA—OR Memory with Accumulator

The value currently in the accumulator is logically ORed with the specified operand, and the result is placed in the accumulator.

<i>Operation: A <math>\vee</math> M <math>\rightarrow</math> A</i>	<i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: N, Z</i>		
			<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Immediate		ORA #\$hh	09	2	2
Zero page		ORA \$hh	05	2	3
Zero page, X		ORA \$hh,X	15	2	4

Operation: $A \vee M \rightarrow A$		Flags Affected: N, Z		
Addressing Mode	Syntax	Opcode	Bytes	Cycles
Absolute	ORA \$hhhh	0D	3	4
Absolute, X	ORA \$hhhh,X	1D	3	4*
Absolute, Y	ORA \$hhhh,Y	19	3	4*
(Indirect, X)	ORA (\$hh,X)	01	2	6
(Indirect), Y	ORA (\$hh),Y	11	2	5*

\* Add 1 when a page boundary is crossed.

Those bits that are set to 1 in the value in the accumulator or set to 1 in the operand, or both, are set to 1 in the result. Bits that are cleared to 0 in both values are cleared to 0 in the result. If the result has the high bit set to 1, the N flag is set to 1; otherwise it is cleared to 0. If the result is zero, the Z bit is set to 1; otherwise it is cleared to 0.

**Example:** The first 256 locations of color memory are set to multicolor \$08. Then the ORA instruction is used to place the graphics chip into multicolor mode. The program waits for a key to be hit and then returns to normal mode.

. 2000 A0 00	LDY #\$00	.Y indexes color memory.
. 2002 A9 08	LDA #\$08	.A is color to fill with.
. 2004 99 00 08	STA \$0800,Y	Store color in the next color location.
. 2007 88	DEY	Decrement the index register.
. 2008 D0 FA	BNE \$2004	If not done, go on.
. 200A AD 07 FF	LDA \$FF07	Get the current value of register 7.
. 200D 09 10	ORA #\$10	Set bit 4 to 1 to turn on multicolor.
. 200F 8D 07 FF	STA \$FF07	Store in register 7.
. 2012 20 E4 FF	JSR \$FFE4	Look for a key hit.
. 2015 F0 FB	BEQ \$2012	If none, go back.
. 2017 AD 07 FF	LDA \$FF07	Get the current value of register 7.
. 201A 29 EF	AND #\$EF	Clear bit 4 to 0 to turn off multicolor.
. 201C 8D 07 FF	STA \$FF07	Store in register 7.
. 201F 00	BRK	Stop processing.

## PHA—Push Accumulator onto the Stack

The value currently in the accumulator is pushed onto the stack.

Operation: $A \downarrow$		Flags Affected: None		
Addressing Mode	Syntax	Opcode	Bytes	Cycles
Implied	PHA	48	1	3

The value in the accumulator is stored in the location indicated by the stack pointer, and the stack pointer is decremented.

**Example:** This program continuously changes the border color until a key is hit. The PHA instruction is used to save the original color of the border on the stack for restoration after the key hit.

- . 2000 AD 19 FF LDA \$FF19 Get the current border color.
- . 2003 48 PHA Save on the stack.
- . 2004 20 OC 20 JSR \$200C Execute the border color change subroutine.
- . 2007 68 PLA Retrieve the original border color.
- . 2008 8D 19 FF STA \$FF19 Restore the original border color.
- . 200B 00 BRK Stop processing.
- . 200C EE 19 FF INC \$FF19 Increment the border color.
- . 200F 20 E4 FF JSR \$FFE4 Look for a key hit.
- . 2012 F0 F8 BEQ \$200C If no key, then go back.
- . 2014 60 RTS Exit the subroutine.

### PHP—Push Processor Status Register onto the Stack

The value currently in the processor status register is pushed onto the stack.

<i>Operation: SR ↓</i>		<i>Flags Affected: None</i>		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	PHP	08	1	3

The value in the processor status register is stored in the location indicated by the stack pointer, and the stack pointer is decremented.

**Example:** This example shows BASIC preparing to return from a SYS command. The value of the status register returned from the user routine is saved in \$07F5.

- . A7CF 08 PHP Push the returned status onto the stack.
- . A7D0 8D F2 07 STA \$07F2 Save the returned .A.
- . A7D3 8E F3 07 STX \$07F3 Save the returned .X.
- . A7D6 8C F4 07 STY \$07F4 Save the returned .Y.
- . A7D9 68 PLA Pull the returned status from the stack.

- . A7DA 8D F5 07 STA \$07F5 Save the returned status register.
- . A7DD 60 RTS Return from SYS.

## PLA—Pull Accumulator from the Stack

A value is pulled from the stack and placed in the accumulator.

<i>Operation: A ↑</i>			<i>Flags Affected: N, Z</i>
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes Cycles</i>
Implied	PLA	68	1 4

The stack pointer is incremented, and the value it points to is placed in the accumulator.

**Example:** This program continuously changes the border color until a key is pressed. The original color of the border is saved on the stack and retrieved using the PLA instruction after a key is hit.

- . 2000 AD 19 FF LDA \$FF19 Get the current border color.
- . 2003 48 PHA Save on the stack.
- . 2004 20 OC 20 JSR \$200C Execute the border color change subroutine.
- . 2007 68 PLA Retrieve the original border color from the stack.
- . 2008 8D 19 FF STA \$FF19 Restore the original border color.
- . 200B 00 BRK Stop processing.
- . 200C EE 19 FF INC \$FF19 Increment the border color.
- . 200F 20 E4 FF JSR \$FFE4 Look for a key hit.
- . 2012 F0 F8 BEQ \$200C If no key, then go back.
- . 2014 60 RTS Exit the subroutine.

## PLP—Pull Processor Status Register from the Stack

A value is pulled from the stack and placed in the processor status register.

<i>Operation: SR ↑</i>			<i>Flags Affected: All</i>
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes Cycles</i>
Implied	PLP	28	1 4

The stack pointer is incremented, and the value it points to is placed in the processor status register.

**Example:** This example shows BASIC setting up for a SYS command. The value to send in the status register is stored in \$07F5.

- A7BE AD F5 07 LDA \$07F5      Get the status register value from memory.
- A7C1 48            PHA              Push the value onto the stack.
- A7C2 AD F2 07 LDA \$07F2      Set up .A.
- A7C5 AE F3 07 LDX \$07F3      Set up .X.
- A7C8 AC F4 07 LDY \$07F4      Set up .Y.
- A7CB 2B            PLP              Pull the status register from the stack.
- A7CC 6C 14 00 JMP (\$0014)      Jump to the user routine.

## ROL—Rotate Left One Bit

Each bit in the specified operand is rotated one bit to the left. The high bit is rotated into the carry flag, and the carry flag is rotated into the low bit.

*Operation: C ← [7      0] ←*      *Flags Affected: N, Z, C*

<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Accumulator	ROL	2A	1	2
Zero page	ROL \$hh	26	2	5
Zero page, X	ROL \$hh,X	36	2	6
Absolute	ROL \$hhhh	2E	3	6
Absolute, X	ROL \$hhhh,X	3E	3	7

Bit 7 of the operand is rotated into the carry flag. Bits 6 through 0 are rotated into bits 7 through 1, respectively. The carry flag is rotated into bit 0. If bit 7 of the result is set to 1, the N flag is set to 1; otherwise it is cleared to 0. If the result is zero, the Z bit is set to 1; otherwise it is cleared to 0.

**Example:** This program is a 4-byte shift left (or multiplication by 2). If an overflow occurs, the border color is incremented. The ROL instruction is used to shift left and transmit the information between bytes by means of the carry bit.

- 2000 0E 03 21 ASL \$2103      Shift the low byte to the left.
- 2003 2E 02 21 ROL \$2102      Rotate the next byte to the left.
- 2006 2E 01 21 ROL \$2101      Rotate the next byte to the left.

- . 2009 2E 00 21 ROL \$2100      Rotate the next byte to the left.
- . 200C 90 03        BCC \$2011      If no overflow, skip the next instruction.
- . 200E EE 19 FF INC \$FF19      Increment the border color.
- . 2011 00            BRK              Stop processing.

Use the M (Memory) command to examine and change the values in \$2100–\$2103.

## ROR—Rotate Right One Bit

Each bit in the specified operand is rotated one bit to the right. The low bit is rotated into the carry flag, and the carry flag is rotated into the high bit.

<i>Operation:</i>	$\boxed{[7 \quad 0] \rightarrow C}$	<i>Flags Affected: N, Z, C</i>		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Accumulator	ROR	6A	1	2
Zero page	ROR \$hh	66	2	5
Zero page, X	ROR \$hh,X	76	2	6
Absolute	ROR \$hhhh	6E	3	6
Absolute, X	ROR \$hhhh,X	7E	3	7

Bit 0 of the operand is rotated into the carry flag. Bits 7 through 1 are rotated into bits 6 through 0, respectively. The carry flag is rotated into bit 7. If the high bit of the result is set to 1, the N flag is set to 1; otherwise it is cleared to 0. If the result is zero, the Z bit is set to 1; otherwise it is cleared to 0.

**Example:** This is a 4-byte shift right (division by 2). The ROR instruction is used to shift the bits right and transmit information between bytes by means of the carry bit.

- . 2000 4E 00 21 LSR \$2100      Shift the high byte to the right.
- . 2003 6E 01 21 ROR \$2101      Rotate the next byte to the right.
- . 2006 6E 02 21 ROR \$2102      Rotate the next byte to the right.
- . 2009 6E 03 21 ROR \$2103      Rotate the next byte to the right.
- . 200C 00            BRK              Stop processing.

Use the M (Memory) command to examine and change the values in \$2100–\$2103.

## RTI—Return from an Interrupt

The values of the SR and PC are pulled from the stack, and processing continues.

<i>Operation:</i>	<i>SR ↑</i>	<i>PC ↑</i>	<i>PC + I → PC</i>	<i>Flags Affected:</i>	<i>All</i>
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>	
Implied	RTI	40	1	6	

This instruction is used to end the processing of an interrupt. The status register is restored from the stack. Because an interrupt can occur only when the I flag is clear, this status register value will have the I flag cleared. The program counter is also pulled from the stack and incremented, and processing continues at that point. No other internal registers are affected. See the section on interrupts for more information.

**Example:** This example alters the IRQ vector. The normal interrupt processing is suspended. The substitute service routine simply increments the border color. The program must be halted by pressing the reset button on the right side of the computer.

- . 2000 78            SEI            Disable maskable interrupts.
- . 2001 A9 OF        LDA #\$OF      Get the low byte of the address of the new IRQ processor.
- . 2003 8D 14 03     STA \$0314     Store in the vector.
- . 2006 A9 20        LDA #\$20      Get the high byte of the address of the new IRQ processor.
- . 2008 8D 15 03     STA \$0315     Store in the vector.
- . 200B 58            CLI            Reenable maskable interrupts.
- . 200C 4C OC 20     JMP \$200C     Infinite loop.
- . 200F EE 19 FF     INC \$FF19     Increment the border color.
- . 2012 68            PLA            Retrieve the value of .Y from the stack.
- . 2013 A8            TAY            Restore .Y to the preinterrupt value.
- . 2014 68            PLA            Retrieve the value of .X from the stack.
- . 2015 AA            TAX            Restore .X to the preinterrupt value.
- . 2016 68            PLA            Restore the value of .A from the stack.
- . 2017 40            RTI            Return from the interrupt.

## RTS—Return from a Subroutine

The value of the PC is pulled from the stack, and processing continues from that point.

<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	RTS	60	1	6

This instruction is used to end the processing of a subroutine. The return address is pulled from the stack, incremented, and stored in the PC. None of the flags is affected. Hence, the flags may be used to communicate between the main program and the subroutine.

**Example:** This program cycles the two upper left character locations on the screen through all possible values. They are changed alternately, with a brief wait between. The wait routine is a subroutine that is terminated by an RTS instruction.

. 2000 A2 00	LDX #\$00	.X counts all the possible values.
. 2002 EE 00	OC INC \$OC00	Next character for the first location.
. 2005 20 12	20 JSR \$2012	Call the wait subroutine.
. 2008 EE 01	OC INC \$OC01	Next character for the second location.
. 200B 20 12	20 JSR \$2012	Call the wait subroutine.
. 200E CA	DEX	Decrement the counter.
. 200F DO F1	BNE \$2002	If not done, go back.
. 2011 00	BRK	Stop processing.
. 2012 A5 A5	LDA \$A5	Start of the subroutine. Load the jiffy clock.
. 2014 29 08	AND #\$08	Look at bit 3.
. 2016 DO FA	BNE \$2012	Wait until it is off.
. 2018 A5 A5	LDA \$A5	Load the value of the jiffy clock.
. 201A 29 08	AND #\$08	Look at bit 3.
. 201C F0 FA	BEQ \$2018	Wait until it is on.
. 201E 60	RTS	Return from the subroutine.

## SBC—Subtract Memory from Accumulator with Carry

The operand is subtracted from the value in the accumulator minus the inverse of the carry, and the result is placed in the accumulator. Normally the carry is set (see SEC) prior to a subtraction. When in decimal mode, the Z flag is not valid; check the accumulator for a zero result.

Operation: A - M - ~ C → A, C		Flags Affected: N, Z, C, V		
Addressing Mode	Syntax	Opcode	Bytes	Cycles
Immediate	SBC #\$hh	E9	2	2
Zero page	SBC \$hh	E5	2	3
Zero page, X	SBC \$hh,X	F5	2	4
Absolute	SBC \$hhhh	ED	3	4
Absolute, X	SBC \$hhhh,X	FD	3	4*
Absolute, Y	SBC \$hhhh,Y	F9	3	4*
(Indirect, X)	SBC (\$hh,X)	E1	2	6
(Indirect), Y	SBC (\$hh),Y	F1	2	5*

\* Add 1 when a page boundary is crossed.

The microprocessor does two's complement subtraction by taking the one's complement of the operand and performing an addition with carry. Thus, it is necessary to set the carry prior to the subtraction to get a valid two's complement result. The status of the carry following a subtraction reflects the result. If no borrow was required, the carry is set to 1. If a borrow was required, the carry is cleared to 0. This fact may be used to perform multiple precision subtractions. If the result is negative, the N flag is set to 1; otherwise, it is cleared to 0. If the result is zero, the Z flag is set to 1; otherwise it is cleared to 0. If the result exceeds +127 or -128, the overflow flag is set to 1; otherwise it is cleared to 0.

**Example:** Frequently, it is necessary to have a greater precision in calculations than the 256 possible values for a single byte. This is possible by treating a group of 2 or more bytes as a single number. In this example, the 32-bit number stored at \$2104-\$2107 (high byte to low byte) is subtracted from the 32-bit number at \$2100-\$2103, and the result stored in \$2108-\$210B.

- . 2000 38 SEC Set the carry prior to the first subtraction.
- . 2001 A2 03 LDX #\$03 .X will index through the 4 bytes.
- . 2003 BD 00 21 LDA \$2100,X Get a byte of the first number.
- . 2006 FD 04 21 SBC \$2104,X Subtract the corresponding byte from the second number.
- . 2009 9D 08 21 STA \$2108,X Store the result in the destination.
- . 200C CA DEX Decrement .X to point at the next byte.
- . 200D 10 F4 BPL \$2003 Continue processing until all 4 bytes are done.
- . 200F 00 BRK Stop processing.

Use the M (Memory) command to examine and modify the contents of \$2100-\$2107 before executing the program (using G 2000). Then check the contents of \$2100-\$210B after the program executes. The values in \$2108-\$210B will be the difference of \$2100-\$2103 and \$2104-\$2107.

The carry flag is used to transmit the borrow information between the bytes. Before the first SBC, the carry is set, so the first result is correct. If a borrow is produced (i.e., the carry is cleared), it is automatically accounted for in the next byte because the program does not set the carry before performing the next SBC.

### SEC—Set the Carry Flag

The carry flag is set to 1.

<i>Operation: I → C</i>			<i>Flags Affected: C</i>	
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	SEC	38	1	2

**Example:** This program subtracts the contents of \$2101 from the contents of \$2100 and stores the result in \$2102. Since the SBC instruction adds the value of the carry to the sum of the accumulator and the one's complement of the operand, the SEC instruction is used to set the carry flag prior to the subtraction. This ensures that the result is exactly the first value minus the second value.

- |                 |            |  |
|-----------------|------------|--|
| . 2000 38       | SEC        | Set the carry flag to prepare to subtract. |
| . 2001 AD 00 21 | LDA \$2100 | Get the first value.                       |
| . 2004 ED 01 21 | SBC \$2101 | Subtract the second value.                 |
| . 2007 8D 02 21 | STA \$2102 | Store the result.                          |
| . 200A 00       | BRK        | Stop processing.                           |

Use the M (Memory) command to examine and change the contents of \$2100–\$2102.

### SED—Set Decimal Mode

The processor is put into decimal mode.

<i>Operation: I → D</i>			<i>Flags Affected: D</i>	
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	SED	F8	1	2

See the section on decimal mode.

**Example:** This program places the processor into decimal mode using the SED instruction and adds the contents of \$2100 and \$2101. The result is placed in \$2102, and decimal mode is terminated.

. 2000 F8	SED	Put the processor into decimal mode.
. 2001 18	CLC	Prepare to add.
. 2002 AD 00 21	LDA \$2100	Get the first decimal value.
. 2005 6D 01 21	ADC \$2101	Add the second decimal value.
. 2008 8D 02 21	STA \$2102	Store the result.
. 200B D8	CLD	Exit decimal mode.
. 200C 00	BRK	Stop processing.

Use the M (Memory) command to examine and change the contents of \$2100–\$2102 to experiment with decimal mode addition.

### SEI—Set Interrupt Disable Flag

The interrupt disable flag is set to prevent interrupts from occurring.

<i>Operation: I → I</i> <i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: I</i>		
		<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	SEI	78	1	2

This instruction sets the I flag of the microprocessor status register. When this flag is set, maskable interrupts do not occur. The interrupt caused by the BRK instruction cannot be prevented in this way.

**Example:** This example alters the IRQ vector. Before executing it, check the default contents of \$0314–\$0315 with the M (Memory) command. The low byte of the default address for IRQ processing is located in \$0314 and the high byte in \$0315. If this address is not \$CE0E, replace the \$CE0E in the program with it.

. 2000 78	SEI	Disable maskable interrupts.
. 2001 A9 0D	LDA #\$0D	Get the low byte of the address of the new IRQ processor.
. 2003 8D 14 03	STA \$0314	Store in the vector.
. 2006 A9 20	LDA #\$20	Get the high byte of the address of the new IRQ processor.
. 2008 8D 15 03	STA \$0315	Store in the vector.
. 200B 58	CLI	Reenable maskable interrupts.

- . 200C 00 BRK Stop processing.
- . 200D EE 19 FF INC \$FF19 Increment the border color.
- . 2010 4C 0E CE JMP \$CEOE Jump to normal IRQ processing.

After you execute this program (by typing G 2000), the processing of an interrupt increments the border color before proceeding normally. When you are finished with this example, reset the computer by typing G FFF6 to restore the IRQ vector to its default value.

### STA—Store Value in Accumulator into Memory

The value currently in the accumulator is placed in the location specified by the operand.

<i>Operation: A → M</i>		<i>Flags Affected: None</i>		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Zero page	STA \$hh	85	2	3
Zero page, X	STA \$hh,X	95	2	4
Absolute	STA \$hhhh	8D	3	4
Absolute, X	STA \$hhhh,X	9D	3	5
Absolute, Y	STA \$hhhh,Y	99	3	5
(Indirect, X)	STA (\$hh,X)	81	2	6
(Indirect), Y	STA (\$hh),Y	91	2	6

The value of the accumulator is not altered, and the flags are not altered.

**Example:** The characters on the top line of the screen are copied to the second line of the screen. Each value is transferred by loading it into the accumulator and storing it to the destination with the STA instruction.

- . 2000 A2 27 LDX #\$27 .X indexes the line.
- . 2002 BD 00 OC LDA \$0C00,X Load the accumulator with next character.
- . 2005 9D 28 OC STA \$0C28,X Store the value in the accumulator on the second line.
- . 2008 CA DEX Decrement the index register.
- . 2009 10 F7 BPL \$2002 If there is more to do, go back.
- . 200B 00 BRK Stop processing.

If executing the BRK instruction causes the screen to scroll, the result of the program will not be seen. To avoid this, clear the screen before executing the program.

## STX—Store Value in X Register into Memory

The value currently in the X register is placed in the location specified by the operand.

<i>Operation: X → M</i>		<i>Flags Affected: None</i>		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Zero page	STX \$hh	86	2	3
Zero page, Y	STX \$hh,Y	96	2	4
Absolute	STX \$hhhh	8E	3	4

The value of the X register is not altered, and the flags are not altered.

**Example:** This program counts the number of locations containing the value \$EA (the opcode for a NOP) in \$DA00–\$DAFF. The X register is used as the counter, and the final count is stored in \$2100 using the STX instruction.

- 2000 A2 00      LDX #\$00      Start .X count at zero.
- 2002 A0 00      LDY #\$00      .Y is the index register.
- 2004 A9 EA      LDA #\$EA      Load .A with the value to look for.
- 2006 D9 00 DA    CMP \$DA00,Y   Check the next byte.
- 2009 D0 01      BNE \$200C     If not equal, then skip the next instruction.
- 200B E8          INX          Increment the count.
- 200C 88          DEY          Decrement the index register.
- 200D D0 F7      BNE \$2006     If not done, go back.
- 200F 8E 00 21    STX \$2100    Store the final count at \$2100.
- 2012 00          BRK          Stop processing.

## STY—Store Value in Y Register into Memory

The value currently in the Y register is placed in the location specified by the operand.

<i>Operation: Y → M</i>		<i>Flags Affected: None</i>		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Zero page	STY \$hh	84	2	3
Zero page, X	STY \$hh,X	94	2	4
Absolute	STY \$hhhh	8C	3	4

The value of the Y register is not altered, and the flags are not altered.

**Example:** This program looks through \$8000–\$80FF until it finds an occurrence of the value \$EA (the opcode for a NOP). The location of the \$EA is then stored at \$2100 using the STY instruction.

. 2000 A0 00	LDY #\$00	Start .Y at zero.
. 2002 A9 EA	LDA #\$EA	Load .A with the value to look for.
. 2004 D9 00 80	CMP \$8000,Y	Compare with the next byte.
. 2007 F0 03	BEQ \$200C	If equal, then done.
. 2009 C8	INY	Increment to the next byte.
. 200A D0 F8	BNE \$2004	If there is more to search, then go on.
. 200C 8C 00 21	STY \$2100	Store the location in \$2100.
. 200F 00	BRK	Stop processing.

## TAX—Transfer Value in Accumulator into X Register

The value currently in the accumulator is placed in the X register.

<i>Operation: A → X</i> <i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: N, Z</i>		
		<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	TAX	AA	1	2

The value in the accumulator is not altered. If the value transferred is negative, the N flag is set to 1; otherwise it is cleared to 0. If the value transferred is zero, the Z flag is set to 1; otherwise it is cleared to 0.

**Example:** In this program a value for the X register is gotten from a location specified by an indirect index. Because the indirect indexed addressing mode is not available for the LDX instruction, the value is loaded into the accumulator and then transferred to the X register with the TAX instruction.

. 2000 A9 00	LDA #\$00	Load .A with a zero.
. 2002 85 D8	STA \$D8	Store in the low byte of the address.
. 2004 A9 21	LDA #\$21	Load .A with a \$21.
. 2006 85 D9	STA \$D9	Store in the high byte of the address.
. 2008 A0 00	LDY #\$00	Load .Y with a zero.
. 200A B1 D8	LDA (\$D8),Y	Get the index for the data retrieval.
. 200C AA	TAX	Transfer the index to .X.
. 200D BD 00 21	LDA \$2100,X	Load .A with the data byte.
. 2010 8D 00 0C	STA \$0C00	Store on the screen.
. 2013 00	BRK	Stop processing.

## TAY—Transfer Value in Accumulator into Y Register

The value currently in the accumulator is placed in the Y register.

<i>Operation: A → Y</i>		<i>Flags Affected: N, Z</i>		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	TAY	A8	1	2

The value in the accumulator is not altered. If the value transferred is negative, the N flag is set to 1; otherwise it is cleared to 0. If the value transferred is zero, the Z flag is set to 1; otherwise it is cleared to 0.

**Example:** In the following program a data byte must be multiplied by 2 to be used as an index. Since shifts may not be performed on the Y register, the shift left is performed in the accumulator, and the result is copied into the Y register using the TAY instruction.

- . 2000 AD 00 21 LDA \$2100 Get the data byte.
- . 2003 OA ASL Multiply by 2.
- . 2004 A8 TAY Transfer the index value into .Y.
- . 2005 B9 00 22 LDA \$2200,Y Load the referenced data.
- . 2008 00 BRK Stop processing.

## TSX—Transfer Value in Stack Pointer into X Register

The value currently in the stack pointer is placed in the X register.

<i>Operation: SP → X</i>		<i>Flags Affected: N, Z</i>		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	TSX	BA	1	2

The value in the stack pointer is not altered. If the value transferred is negative, the N flag is set to 1; otherwise it is cleared to 0. If the value transferred is zero, the Z flag is set to 1; otherwise it is cleared to 0.

**Example:** This program illustrates the functioning of the stack. The TSX instruction is used to copy the stack pointer into the X register. The value of the stack pointer when the program begins executing is stored at \$2100. Then the accumulator is pushed onto the stack, and the resulting stack pointer is stored at \$2101. The accumulator is then pulled from the stack, and the new stack pointer is stored at \$2102.

. 2000 BA	TSX	Transfer the initial SP value to .X.
. 2001 8E 00 21	STX \$2100	Store at \$2100.
. 2004 48	PHA	Push .A onto the stack.
. 2005 BA	TSX	Transfer the new SP value to .X.
. 2006 8E 01 21	STX \$2101	Store at \$2101.
. 2009 68	PLA	Pull .A from the stack.
. 200A BA	TSX	Transfer the new SP value to .X.
. 200B 8E 02 21	STX \$2102	Store at \$2101.
. 200E 00	BRK	Stop processing.

The M (Memory) command may be used to examine \$2100–\$2102 following the execution of this example program. See the section on the stack for more information.

### TXA—Transfer Value in X Register into Accumulator

The value currently in the X register is placed in the accumulator.

<i>Addressing Mode</i>	<i>Syntax</i>	<i>Flags Affected: N, Z</i>		
		<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	TXA	8A	1	2

The value in the X register is not altered. If the value transferred is negative, the N flag is set to 1; otherwise it is cleared to 0. If the value transferred is zero, the Z flag is set to 1; otherwise it is cleared to 0.

**Example:** In this example the bits set to 1 in the value of \$2100 are counted. This is done in the X register because the accumulator cannot be incremented. The value obtained is then transferred to the accumulator using the TXA instruction.

. 2000 A0 07	LDY #\$07	.Y keeps track of which bit.
. 2002 A2 00	LDX #\$00	.X counts bits set to 1.
. 2004 AD 00 21	LDA \$2100	Get the value to count.
. 2007 4A	LSR	Shift the next bit into carry.
. 2008 90 01	BCC \$200B	If clear, skip the next instruction.
. 200A E8	INX	Increment the count of bits set to 1.
. 200B 88	DEY	Decrement which bit.
. 200C 10 F9	BPL \$2007	If not done, go on.
. 200E 8A	TXA	Transfer the count to .A for future use.
. 200F 00	BRK	Stop processing.

The value of \$2100 may be altered using the M (Memory) command.

## TXS—Transfer Value in X Register into Stack Pointer

The value currently in the X register is placed in the stack pointer.

<i>Operation: X → SP</i>		<i>Flags Affected: None</i>		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	TXS	9A	1	2

The value in the X register is not altered. The information stored in the stack area (page 1) is not changed by this instruction. Only which byte the stack pointer points to is changed.

**Example:** When the computer is reset, part of the process is to reset the stack pointer to its initial value, \$FF. This example illustrates how that would be done using the TXS instruction.

- . 2000 A2 FF      LDX #\$FF      Load .X with the desired pointer value.
- . 2002 9A          TXS          Transfer to SP.
- . 2003 00          BRK          Stop processing.

## TYA—Transfer Value in Y Register into Accumulator

The value currently in the Y register is placed in the accumulator.

<i>Operation: Y → A</i>		<i>Flags Affected: N, Z</i>		
<i>Addressing Mode</i>	<i>Syntax</i>	<i>Opcode</i>	<i>Bytes</i>	<i>Cycles</i>
Implied	TYA	98	1	2

The value in the Y register is not altered. If the value transferred is negative, the N flag is set to 1; otherwise it is cleared to 0. If the value transferred is zero, the Z flag is set to 1; otherwise it is cleared to 0.

**Example:** This program finds the offset from \$8020 of the first location containing the value \$EA in \$8020-\$811F by incrementing the Y register. Then the offset from \$8000 is calculated using the TYA instruction to transfer the contents of the Y register to the accumulator and adding. Note that the status of the carry bit at \$2010 indicates whether the first \$EA occurs in \$8020-\$80FF (carry clear) or \$8100-\$811F (carry set).

- . 2000 A0 00      LDY #\$00      Start .Y at zero.
- . 2002 A9 EA      LDA #\$EA      Load the accumulator with the byte to search for.

- . 2004 D9 20 80 CMP \$8020,Y Examine the next memory location.
- . 2007 F0 03 BEQ \$200C If equal, then done.
- . 2009 C8 INY Increment the index.
- . 200A D0 F8 BNE \$2004 If not finished, go back.
- . 200C 18 CLC Prepare to add.
- . 200D 98 TYA Copy the index into the accumulator.
- . 200E 69 20 ADC #\$20 Add \$20.
- . 2010 8D 00 21 STA \$2100 Store the result in \$2100.
- . 2013 00 BRK Stop processing.

## Summary of Instruction Set

### Opcodes and Number of Bytes

INSTR.	#Imm (2)	Abs (3)	ZP (2)	Acc (1)	Imp (1)	Addressing Modes (Bytes)							
						(,X) (2)	(,)Y (2)	ZP,X (2)	Ab,X (3)	Ab,Y (3)	Rel (2)	Ind (3)	ZP,Y (2)
ADC	69	6D	65			61	71	75	7D	79			
AND	29	2D	25				21	31	35	3D	39		
ASL		0E	06	0A				16	1E				90
BCC													B0
BCS													F0
BEQ													
BIT		2C	24										
BMI													30
BNE													D0
BPL													10
BRK						00							
BVC													50
BVS													70
CLC						18							
CLD						D8							
CLI						58							
CLV						B8							
CMP	C9	CD	C5				C1	D1	D5	DD	D9		
CPX	E0	EC	E4										
CPY	C0	CC	C4										
DEC		CE	C6						D6	DE			
DEX						CA							
DEY						88							
EOR	49	4D	45				41	51	55	5D	59		
INC		EE	E6						F6	FE			
INX						E8							
INY						C8							

### Execution Times (in clock cycles)

## *Addressing Modes*

\* Add 1 cycle if page boundary is crossed.

\*\* Add 1 cycle if the branch occurs to the same page. Add 2 cycles if the branch occurs to a different page.

## Opcodes

Least Significant Nybble													
	0	1	2	4	5	6	8	9	A	C	D	E	
0	BRK	ORA (,X)			ORA ZP	ASL ZP	PHP	ORA Imm	ASL Acc		ORA Abs	ASL Abs	
1	BPL	ORA (,Y)			ORA ZP,X	ASL ZP,X	CLC	ORA Ab,Y			ORA Ab,X	ASL Ab,X	
2	JSR	AND (,X)	BIT ZP	AND ZP	ROL ZP	PLP	AND Imm	ROL Acc	BIT Abs	AND Abs	ROL Abs		
3	BMI	AND (,Y)		AND ZP,X	ROL ZP,X	SEC	AND Ab,Y			AND Ab,X	ROL Ab,X		
M o s t	4	RTI	EOR (,X)		EOR ZP	LSR ZP	PHA	EOR Imm	LSR Acc	JMP Abs	EOR Abs	LSR Abs	
S i g n	5	BVC	EOR (,Y)		EOR ZP,X	LSR ZP,X	CLI	EOR Abs,Y			EOR Ab,X	LSR Ab,X	
	6	RTS	ADC (,X)		ADC ZP	ROR ZP	PLA	ADC Imm	ROR Acc	JMP Ind	ADC Abs	ROR Abs	

	Least Significant Nybble												
	0	1	2	4	5	6	8	9	A	C	D	E	
i	7	BVS	ADC 0,Y		ADC ZP,X	ROR ZP,X	SEI	ADC Ab,Y			ADC Ab,X	ROR Ab,X	
f													
i	8		STA (,X)	STY ZP	STA ZP	STX ZP	DEY		TXA	STY Abs	STA Abs	STX Abs	
c													
a	9	BCC	STA 0,Y	STY ZP,X	STA ZP,X	STX ZP,Y	TYA	STA Ab,Y	TXS		STA Ab,X		
n													
N	A	LDY	LDA Imm	LDX (,X)	LDY Imm	LDA ZP	LDX ZP	TAY	LDA Imm	TAX	LDY Abs	LDA Abs	LDX Abs
y													
b	B	BCS	LDA 0,Y		LDY ZP,X	LDA ZP,X	LDX ZP,Y	CLV	LDA Ab,Y	TSX	LDY Ab,X	LDA Ab,X	LDX Ab,Y
b													
l	C	CPY	CMP Imm		CPY ZP	CMP ZP	DEC ZP	INY	CMP Imm	DEX	CPY Abs	CMP Abs	DEC Abs
e													
	D	BNE	CMP 0,Y		CMP ZP,X	DEC ZP,X	CLD	CMP Ab,Y			CMP Ab,X	DEC Ab,X	
	E	CPX	SBC Imm		CPX ZP	SBC ZP	INC ZP	INX	SBC Imm	NOP	CPX Abs	SBC Abs	INC Abs
	F	BEQ	SBC 0,Y		SBC ZP,X	INC ZP,X	SED	SBC Ab,Y			SBC Ab,X	INC Ab,X	

## Decimal Mode

The 6502 is equipped with a decimal adder that can be used to process data stored in binary coded decimal (BCD). Two decimal digits are stored in a byte by storing a 0 through 9 in each of the low and high nybbles. When addition is performed, a result above 9 in the low nybble causes a carry into the high nybble, and a result above 9 in the high nybble causes a carry set condition. Subtraction performs in an analogous way.

Decimal mode is enabled by setting the decimal mode flag in the status register to 1 (see SED). It is disabled by clearing the flag to 0 (see CLD). One of the initialization steps when the Plus/4 is first turned on (or reset) is to issue a CLD instruction. To use decimal mode, issue a SED instruction.

Example:	. 2000 F8	SED	Set decimal mode.
	. 2001 18	CLC	Clear the carry for adding.
	. 2002 A9 98	LDA #\$98	Load .A with 98 (BCD).
	. 2004 69 12	ADC #\$12	Add 12 (BCD).
	. 2006 D8	CLD	Clear decimal mode.
	. 2007 00	BRK	Stop processing.

The result (in .A and the status register) will be 10 (BCD) with a carry.

*Note:* The zero flag is not valid following a decimal mode operation. Use a compare to check for zero.

*Note:* The system interrupt service routine does not clear decimal mode. If it is set in the main program, it is set during IRQ service unless the IRQ routine clears decimal mode. Note that clearing decimal mode in the IRQ routine does not affect its setting in the main program because the status register is restored during the RTI.

## Addressing Modes

Some 6502 instructions must be accompanied by an operand address containing the data with which the instruction will be performed. There are a number of ways to specify the operand address. Basically, 6502 addressing modes fall into two categories, indexed and nonindexed. The nonindexed modes are immediate, absolute, zero page, relative, and indirect. The indexed modes are absolute indexed, zero page indexed, indirect indexed, and indexed indirect.

Many instructions do not require an operand. Their addressing mode is referred to as implied and they require only 1 byte. In the machine-language monitor, an implied mode instruction appears in this format:

*mnemonic*

where *mnemonic* is an instruction mnemonic.

The following instructions are available in implied mode: BRK, CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, PHA, PHP, PLA, PLP, RTI, RTS, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, and TYA.

A few instructions may be performed on the data in the accumulator. Their addressing mode is referred to as accumulator and they require only 1 byte. In the machine-language monitor, an accumulator mode instruction appears in this format:

*mnemonic*

where *mnemonic* is an instruction mnemonic.

The following instructions are available in accumulator mode: ASL, LSR, ROL, and ROR.

**Immediate Mode** An instruction in immediate mode consists of 2 bytes. The first is the opcode for the desired instruction. The second is the data for use with the instruction. In the machine-language monitor, an immediate mode instruction appears in this form:

*mnemonic #\$hh*

where *mnemonic* is an instruction mnemonic and h represents a hexadecimal digit.

The following instructions are available in immediate mode: ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, and SBC.

**Absolute Mode** An instruction in absolute mode consists of 3 bytes. The first is the opcode for the desired instruction. The second is the low byte of the address of the data to be used with the instruction. The third is the high byte of the address of the data to be used with the instruction. In the machine-language monitor, an absolute mode instruction appears in this form:

*mnemonic \$hhhh*

where *mnemonic* is an instruction mnemonic and h represents a hexadecimal digit.

The following instructions are available in absolute mode: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, and STY.

**Zero Page Mode** An instruction in zero page mode consists of 2 bytes. The first is the opcode for the desired instruction. The second is the low byte of the address of the data to be used with the instruction. The high byte of the address is assumed to be zero and is not specified. In the machine-language monitor, a zero page mode instruction appears in this form:

*mnemonic \$hh*

where *mnemonic* is an instruction mnemonic and h represents a hexadecimal digit.

The following instructions are available in zero page mode: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, and STY.

**Relative Mode** Only branch instructions are available in relative mode. They consist of 2 bytes. The first is the opcode for the desired instruction. The second is

a 1-byte two's complement offset to the address to be branched to. When the offset is used, the program counter points to the byte immediately following the current instruction. Therefore, an offset of zero does not affect program flow. The maximum branch forward is achieved with an offset of \$7F (127). The maximum branch backward is achieved with an offset of \$80 (-128). In the machine-language monitor, a relative mode instruction appears in this form:

*mnemonic* \$hhhh

where *mnemonic* is an instruction mnemonic, h represents a hexadecimal digit, and \$hhhh is the address to branch to. The monitor calculates the offset and stores it in the appropriate location. If the specified address is too far away (more than 129 bytes forward or more than 126 bytes backward from the beginning of the branch instruction), a question mark will be displayed indicating an error in the line.

The following instructions are available in relative mode: BCC, BCS, BEQ, BMI, BNE, BPL, BVC, and BVS.

**Indirect Mode** Only the jump instruction is available in indirect mode. It consists of 3 bytes. The first is the opcode for the jump instruction. The second is the low byte of the address of the jump vector. The third is the high byte of the address of the jump vector. In the machine-language monitor, an indirect mode jump instruction appears in this form:

JMP (\$hhhh)

where h represents a hexadecimal digit and \$hhhh is the address of the jump vector in which the new program counter is stored. The jump vector consists of 2 bytes: the first is the low byte and the second is the high byte of the new program counter.

The JMP instruction is available in indirect mode.

**Absolute Indexed Mode** In this mode, either the X register or the Y register is used as an index. The address of the operand is calculated by adding the value of the index to the specified base address. An instruction in absolute indexed mode consists of 3 bytes. The first is the opcode for the desired instruction. The second is the low byte of the base address. The third is the high byte of the base address. In the machine-language monitor, an absolute indexed mode instruction using the X register appears in this form:

*mnemonic* \$hhhh,X

where *mnemonic* is an instruction mnemonic and h represents a hexadecimal digit.

The following instructions are available in absolute indexed mode using the X register: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, and STA.

In the machine-language monitor, an absolute indexed mode instruction using the Y register appears in this form:

*mnemonic \$hhhh,Y*

where *mnemonic* is an instruction mnemonic and h represents a hexadecimal digit.

The following instructions are available in absolute indexed mode using the Y register: ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC, and STA.

**Zero Page Indexed Mode** In this mode, either the X register or the Y register is used as an index. The address of the operand is calculated by adding the value of the index to the specified base address. An instruction in zero page indexed mode consists of 2 bytes. The first is the opcode for the desired instruction. The second is the low byte of the base address. The high byte of the base address is assumed to be zero and is not specified. In the machine-language monitor, a zero page indexed mode instruction using the X register appears in this form:

*mnemonic \$hh,X*

where *mnemonic* is an instruction mnemonic and h represents a hexadecimal digit.

The following instructions are available in zero page indexed mode using the X register: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA, and STY.

In the machine-language monitor, a zero page indexed mode instruction using the Y register appears in this form:

*mnemonic \$hh,Y*

where *mnemonic* is an instruction mnemonic and h represents a hexadecimal digit.

The following instructions are available in zero page indexed mode using the Y register: LDX and STX.

*Note:* In zero page indexed mode, the address of the operand is always on zero page. For example, if the X register contains \$04 and the instruction LDA \$FE,X is executed, the value found in location \$02 will be loaded into the accumulator.

**Indirect Indexed Mode** In this mode the Y register is used as the index. The base address is found indirectly in the zero page location specified with the instruction.

The address of the data is found by adding the index to the base address. An instruction in indirect indexed mode consists of 2 bytes. The first is the opcode for the desired instruction. The second is the low byte of the location at which the base address is found. The high byte of the location at which the base address is found is assumed to be zero and is not specified. In the machine-language monitor, an indirect indexed mode instruction appears in this form:

*mnemonic (\$hh),Y*

where *mnemonic* is an instruction mnemonic, h represents a hexadecimal digit, and \$hh is the zero page location of the low byte of the base address. The zero page location following the specified zero page location must contain the high byte of the base address.

The following instructions are available in indirect indexed mode: ADC, AND, CMP, EOR, LDA, ORA, SBC, and STA.

**Indexed Indirect Mode** In this mode the X register is used as the index. The zero page base address is specified with the instruction. The address of the data is found at the address created by adding the index to the base address. An instruction in indexed indirect mode consists of 2 bytes. The first is the opcode for the desired instruction. The second is the low byte of the base address. The high byte of the base address is assumed to be zero and is not specified. In the machine-language monitor, an indexed indirect mode instruction appears in this form:

*mnemonic (\$hh,X)*

where *mnemonic* is an instruction mnemonic, h represents a hexadecimal digit, and \$hh is the zero page base address. The value of the index is added to the base address. The location thus pointed to must contain the low byte of the address of the data; the following location must contain the high byte of the address of the data.

The following instructions are available in indexed indirect mode: ADC, AND, CMP, EOR, LDA, ORA, SBC, and STA.

## Using Interrupts on the Plus/4

When a task such as the incrementing of a real-time clock must be performed on a regular basis, it is frequently impractical to use straight-line code. To alleviate this problem, the 6502 is equipped with processor interrupt capability. The Plus/4 implementation of interrupts is described in this section.

The 6502 has an IRQ (interrupt) line. When this line is activated by an

interrupt, the processor performs a number of tasks. First, the current instruction is completed. Next, the program counter and processor status register are pushed onto the stack. Program control is then transferred through the IRQ vector at \$FFFE-\$FFFF to an interrupt service routine.

The IRQ vector at \$FFFE-\$FFFF is located in the operating system ROM. It points to a routine in the operating system ROM that initiates interrupt processing. First, all three user registers are pushed onto the stack (accumulator, then X register, then Y register). Then it checks for the BRK instruction (B) flag in the saved processor status register. Finding that the interrupt was not caused by a BRK instruction, a jump indirect is performed through a vector at \$0314-\$0315. This is normally the location of programmer intervention. To have a special function performed, this vector can be changed to point at a user routine. To complete the interrupt processing, either jump to the normal operating system routine (the address of which can be found in \$0314-\$0315 on power-up), or pull all three registers and return as shown here:

```
PLA
TAY
PLA
TAX
PLA
RTI
```

The RTI instruction restores the processor status register and program counter from the stack. Hence, all of the processor registers are returned to their preinterrupt state, and the main program continues executing.

The SEI instruction can be used to disable interrupts. This sets the I bit in the processor status register. While this bit is set, the processor cannot be interrupted by a maskable (and non-BRK instruction) interrupt. The CLI instruction is used to reenable interrupts.

The Plus/4 is equipped with an interrupt enable register (\$FF0A). Setting the appropriate bit in this register will cause the corresponding interrupt to occur (when the I bit is clear). The Plus/4 also has an interrupt status register (\$FF09). The high bit of this register is set to 1 when an interrupt occurs. The remaining bits are set to 1 when the corresponding device has an interrupt condition. These bits are set regardless of the interrupt enable status of the device and therefore can be used for timing or other functions with or without actually interrupting the processor. All of the bits in this register are cleared to 0 by writing a 1 to them. The bit corresponding to a device interrupting the processor must be cleared before it can interrupt again.

## Raster Interrupts

The Plus/4 is capable of interrupting the processor in response to the vertical position of the raster beam on the TV or monitor. This capability can be used, for example, to update information for the screen display while it is not seen, thus eliminating flicker.

To enable raster interrupts, bit 1 of the interrupt enable register (\$FF0A) must be set to 1. The low 8 bits of the raster value at which the interrupt is to occur (a 9-bit number) must be placed in the raster compare register (\$FF0B). The high bit is the low bit of the interrupt enable register (\$FF0A) and can be set to the appropriate value at the same time as the enabling. The raster is on the usable screen (not the border) from about \$04 through \$CB.

When a raster interrupt occurs, bit 1 of the raster interrupt status register (\$FF09) is set to 1. Also, bit 7 of this register is set to indicate that an interrupt occurred.

### Example:

In this example program, raster interrupts are used to create a band of changed background color on the screen. First, an interrupt occurs near the top of the screen. The processing of this interrupt includes decrementing the background color and setting up the next interrupt. The second interrupt occurs a few lines lower. The processing of this interrupt includes incrementing the background color back to its original value and setting up the first interrupt again. This process is repeated forever.

. 2000 78 SEI	Disable interrupts for set up.
. 2001 A9 19 LDA #\$19	Low byte of address of service routine.
. 2003 8D 14 03 STA \$0314	Store in low byte of IRQ vector.
. 2006 A9 20 LDA #\$20	High byte of address of service routine.
. 2008 8D 15 03 STA \$0315	Store in high byte of IRQ vector.
. 200B A9 02 LDA #\$02	Bit 1 set to 1, bit 0 cleared to 0.
. 200D 8D 0A FF STA \$FF0A	Enable raster interrupts and clear high bit of compare value.
. 2010 A9 30 LDA #\$30	Compare value is \$30.
. 2012 8D 0B FF STA \$FF0B	Store in low bits of compare value.
. 2015 58 CLI	Start interrupts going.
. 2016 4C 16 20 JMP \$2016	Infinite loop.
. 2019 AD 09 FF LDA \$FF09	Get interrupt status register.
. 201C 8D 09 FF STA \$FF09	Clear all bits that were set.
. 201F AD 0B FF LDA \$FF0B	Get raster compare value.
. 2022 C9 50 CMP #\$50	Compare to \$50 (the bottom interrupt).
. 2024 90 0B BCC \$2031	If less than \$50 (the top interrupt), branch.
. 2026 A9 30 LDA #\$30	Compare value for top interrupt.
. 2028 8D 0B FF STA \$FF0B	Store in compare value.
. 202B EE 15 FF INC \$FF15	Increment the screen color.

. 202E	4C 39 20	JMP \$2039	Jump to exit.
. 2031	A9 50	LDA #\$50	Compare value for bottom interrupt.
. 2033	8D 0B FF	STA \$FF0B	Store in compare value.
. 2036	CE 15 FF	DEC \$FF15	Decrement the screen color.
. 2039	68	PLA	Exit. Pull .A.
. 203A	A8	TAY	Transfer to .Y.
. 203B	68	PLA	Pull .A.
. 203C	AA	TAX	Transfer to .X.
. 203D	68	PLA	Pull .A.
. 203E	40	RTI	Return from interrupt.

Try changing the values used in \$2010 and \$2026 to change the position of the first interrupt or the values used in \$2022 and \$2031 for the second interrupt. To regain control of the computer, push the reset button (and hold down RUN/STOP if you do not want to return to BASIC).

## Timer Interrupts

The Plus/4 graphics chip is equipped with three timers that can be used to interrupt the processor. Interrupts from the first timer are enabled by setting bit 3 of the interrupt enable register (\$FF0A) to 1. This timer has a 16-bit reload value that is stored in two 8-bit registers. The low byte is stored in \$FF00 and the high byte in \$FF01. A load from these registers reads the current value of the timer. A store to these registers sets the reload value. In addition, a store to the low byte (\$FF00) stops the counter, and a store to the high byte (\$FF01) starts it. Hence, the appropriate order for storing a new reload value is always low byte, then high byte.

Once started, this timer counts down from the reload value to zero. Upon reaching zero, it sets bit 3 of the interrupt status register (\$FF09) to 1; if it has been interrupt enabled, bit 7 of this register is also set to 1 and an interrupt occurs. This timer is then reloaded with the reload value and proceeds to count down to zero, repeating the process. For interrupts to continue to occur properly, bit 3 of the interrupt status register (\$FF09) must be cleared by storing a 1 to it, after detecting a timer 1 interrupt.

The second and third timers are interrupt enabled by setting bits 4 and 6, respectively, of the interrupt enable register (\$FF0A) to 1. Unlike the first timer, these timers do not have reload registers. Each has a 16-bit start value that is stored into two 8-bit registers (\$FF02-\$FF03 for the second, and \$FF04-\$FF05 for the third). A load from these registers reads the current value of the timer. A store to these registers sets the start value. As before, a store to a low byte (\$FF02 or \$FF04) stops the associated timer, and a store to a high byte (\$FF03 or \$FF05) starts it. Again, the appropriate order for storing a new start value is low byte, then high byte.

Once started, these timers count down from the start value to zero. Upon reaching zero, they set bit 4 (for the second timer) or 6 (for the third timer) in the

interrupt status register (\$FF09) to 1; if interrupt enabled, bit 7 of this register is also set to 1 and an interrupt occurs. These timers then continue to count down from zero to \$FFFF and from there back to zero unless start values are once again stored for them. When they reach zero, the process repeats. For interrupts to continue to occur properly, bit 4 or 6 (as the case may be) of the interrupt status register (\$FF09) must be cleared by storing a 1 to it, after detecting a timer 2 or 3 interrupt.

**Example:** This example program uses timers 2 and 3 to generate interrupts. Each timer controls the updating of a separate and independent counter on the screen. The main program reads the keyboard and outputs the characters to the screen. This program illustrates the use of interrupts to update information on a regular basis while response to user input is maintained.

The keyboard is read using an operating system ROM subroutine. It expects the keyboard to have been scanned by the normal operating system's raster interrupt. Hence, this program maintains the system interrupt and jumps to the system interrupt service routine following completion of the counter updates. Before executing this program, check the default contents of \$0314 \$0315 with the M (Memory) command. The low byte of the default address for IRQ processing is located in \$0314 and the high byte in \$0315. If this address is not \$CE0E, replace the \$CE0E at \$2096 in the program with it.

- . 2000 78 SEI Disable interrupts for set up.
- . 2001 A9 42 LDA #\$42 Get low byte of interrupt routine address.
- . 2003 8D 14 03 STA \$0314 Store in low byte of interrupt vector.
- . 2006 A9 20 LDA #\$20 Get high byte of interrupt routine address.
- . 2008 8D 15 03 STA \$0315 Store in high byte of interrupt vector.
- . 200B A9 00 LDA #\$00 Get low byte of start value for timer two.
- . 200D 8D 02 FF STA \$FF02 Store in start value register; stop timer two.
- . 2010 A9 00 LDA #\$00 Get low byte of start value for timer three.
- . 2012 8D 04 FF STA \$FF04 Store in start value register; stop timer three.
- . 2015 A9 40 LDA #\$40 Get high byte of start value for timer two.
- . 2017 8D 03 FF STA \$FF03 Store in start value register; start timer two.
- . 201A A9 80 LDA #\$80 Get high byte of start value for timer three.
- . 201C 8D 05 FF STA \$FF05 Store in start value register; start timer three.
- . 201F A9 50 LDA #\$50 Get value with bits 4 and 6 set to 1.
- . 2021 8D 09 FF STA \$FF09 Clear interrupt status register for timers two and three.
- . 2024 A9 52 LDA #\$52 Get value with bits 1, 4, and 6 set to 1.

- . 2026 8D 0A FF STA \$FF0A      Interrupt enable timers two and three and the raster interrupts used by the operating system.
- . 2029 A2 02      LDX #\$02      .X indexes the counters on the screen.
- . 202B A9 30      LDA #\$30      .A contains the screen code for a zero.
- . 202D 9D 00 0C STA \$0C00,X      Store zeroes to left counter.
- . 2030 9D 25 0C STA \$0C25,X      Store zeroes to right counter.
- . 2033 CA      DEX      Next counter location.
- . 2034 10 F7      BPL \$202D      If not done, go back.
- . 2036 58      CLI      Start the interrupts going.
- . 2037 20 E4 FF JSR \$FFE4      Get a character from keyboard queue.
- . 203A F0 FB      BEQ \$2037      If none, look again.
- . 203C 20 D2 FF JSR \$FFD2      Put character to screen.
- . 203F 4C 37 20 JMP \$2037      Look at keyboard queue again.
- . 2042 AD 09 FF LDA \$FF09      Interrupt service. Get interrupt status register.
- . 2045 29 40      AND #\$40      Mask off all but timer three bit.
- . 2047 F0 23      BEQ \$206C      If not a timer three interrupt, go on.
- . 2049 A9 00      LDA #\$00      Get low byte of start value for timer three.
- . 204B 8D 04 FF STA \$FF04      Store in start value register; stop timer three.
- . 204E A9 80      LDA #\$80      Get high byte of start value for timer three.
- . 2050 8D 05 FF STA \$FF05      Store in start value register; start timer three.
- . 2053 A9 40      LDA #\$40      Get value with bit 6 set to 1.
- . 2055 8D 09 FF STA \$FF09      Clear interrupt status bit for timer three.
- . 2058 A2 02      LDX #\$02      .X indexes the counter on the screen.
- . 205A FE 00 0C INC \$0C00,X      Increment left counter location.
- . 205D BD 00 0C LDA \$0C00,X      Load value.
- . 2060 C9 3A      CMP #\$3A      Compare to one more than screen code for nine.
- . 2062 90 08      BCC \$206C      If valid number code, go on.
- . 2064 A9 30      LDA #\$30      Get screen code for zero.
- . 2066 9D 00 0C STA \$0C00,X      Store in counter location.
- . 2069 CA      DEX      Go on to next location.
- . 206A 10 EE      BPL \$205A      If more digits, go back.
- . 206C AD 09 FF LDA \$FF09      Get interrupt status register.
- . 206F 29 10      AND #\$10      Mask off all but timer two bit.
- . 2071 F0 23      BEQ \$2096      If not a timer two interrupt, go on.
- . 2073 A9 00      LDA #\$00      Get low byte of start value for timer two.
- . 2075 8D 02 FF STA \$FF02      Store in start value register; stop timer two.
- . 2078 A9 40      LDA #\$40      Get high byte of start value for timer two.
- . 207A 8D 03 FF STA \$FF03      Store in start value register; start timer two.
- . 207D A9 10      LDA #\$10      Get value with bit 4 set to 1.
- . 207F 8D 09 FF STA \$FF09      Clear interrupt status bit for timer two.
- . 2082 A2 02      LDX #\$02      .X indexes the counter on the screen.

. 2084	FE 25 0C	INC \$0C25,X	Increment right counter location.
. 2087	BD 25 0C	LDA \$0C25,X	Load value.
. 208A	C9 3A	CMP #\$3A	Compare to one more than screen code for nine.
. 208C	90 08	BCC \$2096	If valid number, go on.
. 208E	A9 30	LDA #\$30	Get screen code for zero.
. 2090	9D 25 0C	STA \$0C25,X	Store in counter location.
. 2093	CA	DEX	Go on to next location.
. 2094	10 EE	BPL \$2084	If more digits, go back.
. 2096	4C 0E CE	JMP SCE0E	Jump to operating system interrupt service routine.

Try changing the values used in \$200B, \$2015, \$2073, and \$2078 to change the start value for timer 2 or the values used in \$2010, \$201A, \$2049, and \$204E to change the start value for timer 3. To regain control of the computer, push the reset button (and hold down RUN/STOP if you do not want to return to BASIC).

## The Operating System

The operating system is a program that is built into the Plus/4. It resides in ROM so that it is not erased, even when the computer is turned off. The operating system oversees the operation of the computer system. When the computer is first turned on, the 6502 microprocessor automatically looks at locations \$FFFC-\$FFFD for the initial value of the program counter. Control is passed through this vector to that address. At that address begins an operating system routine that initializes all the registers and memory to be used subsequently to appropriate values. Then, the operating system directs BASIC (or an external program cartridge) to begin executing. BASIC makes frequent use of operating system routines. Some of these routines are available to the machine-language programmer; as outlined below.

## Banking on the Plus/4

The Plus/4 is equipped with 64K of RAM, which is the maximum amount it can address (using 16 bits). However, the operating system and the BASIC language (as well as the built-in software) must reside somewhere. The solution is the ability to "bank." This means that certain registers have the function of determining which memory (RAM or ROM) is addressed.

It is important to distinguish banking for the 6502 microprocessor, which is discussed here, from choosing to have the graphics chip to look at ROM or RAM, which is discussed in Chapter 4. These are two independent choices that do not directly affect each other.

The Plus/4 is designed to allow the microprocessor to look at ROM or RAM in the address spaces \$8000-\$FCFF and \$FF40-\$FFFF. The intervening address space is occupied by I/O and the graphics chip. When the Plus/4 is first turned on, the microprocessor is initialized to look at ROM. Logically it must be so, or the computer would have no program instructions to operate with. To switch to looking at RAM, a store is done to location \$FF3F. To switch back to ROM, a store is done to location \$FF3E.

When the ROM is banked in, a load from its area of memory results in reading the ROM. A store to its area of memory stores the byte into the RAM underneath the ROM. This makes it possible, for example, to store a new character set in the RAM underneath the character ROM without banking in the RAM. Then the graphics chip can be told to look at it.

Considerable care must be taken when banking out the operating system ROM to operate in RAM. For example, the interrupt service vector at \$FFFE-\$FFFF, which is set correctly in the operating system ROM, may not be set in RAM. And, when setting it in RAM, remember that the interrupt service routine it normally points to resides in the operating system ROM and is no longer present.

For use of the upper 32K of RAM for data, storing data to RAM does not require banking the RAM in. When loading the data, it is possible to simply stop interrupts (using the SEI instruction) before banking in RAM and fetching data. ROM can then be banked back in and interrupts reenabled (using the CLI instruction). This is the method used by BASIC to allow 60671 bytes free.

In addition to allowing switching between ROM and RAM, the Plus/4 allows switching between several different ROMs in the address spaces \$8000-\$FBFF and \$FF40-\$FFFF. This is accomplished by means of the cartridge bank port at \$FDD0-\$FDDF, each location of which could switch in a different ROM when stored to. Routines to facilitate the exchange of information between the operating system and a cartridge program are located from \$FC00-\$FCFF and are present in every ROM configuration. The Plus/4 built-in software resides in cartridge bank 5.

## ROM Subroutines

The operating system ROM in the Plus/4 contains a jump table to various operating system subroutines. It is important to call operating system subroutines only through the jump table when writing software designed to run on any Plus/4 computer. The reason for this is that the manufacturer occasionally makes changes in the operating system that result in a change in the location of an operating system routine. But the address of the subroutine call in the jump table remains unchanged.

Before using operating system routines in a program, it is usually a good idea to experiment with them; some have so many possible variations that it is difficult to analyze how they function in every possible circumstance.

Alphabetic List of Operating System Subroutines

<i>Routine</i>	<i>Call Address</i>
ACPTR	\$FFA5
BASIN	\$FFCF
BSOUT	\$FFD2
CHKIN	\$FFC6
CHOUT	\$FFC9
CINT	\$FF81
CIOUT	\$FFA8
CLALL	\$FFE7
CLOSE	\$FFC3
CLRCH	\$FFCC
GETIN	\$FFE4
IOBASE	\$FFF3
IOINIT	\$FF84
LISTN	\$FFB1
LOADSP	\$FFD5
MEMBOT	\$FF9C
MEMTOP	\$FF99
OPEN	\$FFC0
PLOT	\$FFF0
RAMTAS	\$FF87
RDTIM	\$FFDE
READSS	\$FFB7
RESET	\$FFF6
RESTOR	\$FF8A
SAVESP	\$FFD8
SCNKEY	\$FF9F
SCRORG	\$FFED
SECND	\$FF93
SETLFS	\$FFBA
SETHOOK	\$FF90
SETNAM	\$FFBD
SETTIM	\$FFDB
SETTMO	\$FFA2
STOP	\$FFE1
TALK	\$FFB4
TKSA	\$FF96
UDTIM	\$FFEA
UNLSN	\$FFAE
UNTLK	\$FFAB
VECTOR	\$FF8D

## Operating System Subroutine Descriptions

**\$FF81      CINT**  
**Registers Altered: .A, .X, .Y**

Initializes the screen editor. This subroutine performs such functions as setting up default I/O devices (keyboard and screen), the text window, and the current character color. It clears the screen to all blanks with character color black.

**\$FF84      IOINIT**  
**Registers Altered: .A, .X**

Initializes the I/O devices. This subroutine performs such functions as setting up the graphics chip and the DMA disk.

**\$FF87      RAMTAS**  
**Registers Altered: .A, .X, .Y**

Performs a RAM test. This subroutine performs such functions as clearing zero page and pages 2, 3, 4, and 7, setting the top and bottom of memory, and defining the function keys. Call it before any of these areas are initialized, or otherwise they will be overwritten.

**\$FF8A      RESTOR**  
**Registers Altered: .A, .X, .Y**

Restores the vectors to their initial values. This subroutine sets up the vectors at \$0312-\$0331 from ROM, performing such functions as directing IRQ and BRK service to the appropriate addresses.

**\$FF8D      VECTOR**  
**Registers Altered: .A, .Y**

When called with the carry clear, loads the vectors at \$0312-\$0331 from a designated section of memory. Before calling this subroutine, set .X to the low address to load the vectors from, and .Y to the high address to load the vectors from. Be certain none of the vectors will be used during the loading process (e.g., disable interrupts). When called with the carry set, this subroutine reads the vectors and stores their values in the designated section of memory. Before calling it, set .X to the low address to store the vectors to, and .Y to the high address to store the vectors to.

**\$FF90****SETMSG****Registers Altered: .A**

Sets the system message flag to control output of messages. The value in .A when the routine is called is stored in the message flag (\$9A). A value of \$00 means a program is running (and presumably handles message output). A value of \$80 flags output of direct BASIC mode messages. A value of \$C0 flags output of MONITOR messages.

**\$FF93****SECND****Registers Altered: .A**

Sends the value in .A as a secondary address to a device following a call to LISTN (\$FFB1). The valid secondary addresses depend on the device.

The secondary address must be ORed with \$60 before SECND is called. For the 1541 disk operating system, the low nybble determines the channel (\$0 is reserved for LOAD, \$1 for SAVE, and \$F is the command channel). If the high nybble is \$F, a CLOSE is indicated. If the high nybble is \$E, an OPEN is indicated.

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

**\$FF96****TKSA****Registers Altered: .A**

Sends the value in .A as a secondary address to a device following a call to TALK (\$FFB4). The valid secondary addresses depend on the device.

The secondary address must be ORed with \$60 before TKSA is called. For the 1541 disk operating system, the low nybble determines the channel (\$0 is reserved for LOAD, \$1 for SAVE, and \$F is the command channel). If the high nybble is \$F, a CLOSE is indicated. If the high nybble is \$E, an OPEN is indicated.

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

**\$FF99****MEMTOP****Registers Altered: .X, .Y**

When called with the carry set, returns with .X equal to the low byte of the top of memory, and .Y equal to the high byte of the top of memory. When called with the carry clear, the top of memory (\$0533-\$0534) is set. The low byte is from .X and the high byte from .Y. The top of memory is the address plus one of the end of a contiguous section of RAM for use by BASIC. It is initially set during the power-on sequence.

**\$FF9C      MEMBOT****Registers Altered: .X, .Y**

When called with the carry set, returns with .X equal to the low byte of the bottom of memory, and .Y equal to the high byte of the bottom of memory. When called with the carry clear, the bottom of memory (\$0531-\$0532) is set. The low byte is from .X and the high byte from .Y. Only the high byte of the bottom of memory is initialized during the power-on sequence. BASIC does not use this information.

**\$FF9F      SCNKEY****Registers Altered: .A, .X, .Y**

Scans the keyboard and sets up the keyboard queue and the function key index register for the GETIN (\$FFE4) and BASIN (\$FFCF) routines. This subroutine is normally called by the system IRQ service routine.

**\$FFA2      SETTMO****Registers Altered: none**

The value in .A is stored in the timeout flag (\$0535). This location is not used by the built-in Plus/4 operating system. It is designed for use with add-on hardware and software.

**\$FFA5      ACPTR****Registers Altered: .A**

Returns 1 byte of data in .A from the serial bus or DMA disk using handshaking. A device must be instructed to talk using the routine TALK (\$FFB4) before this routine is called.

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

**\$FFA8      CIOUT****Registers Altered: none**

Sends the byte in .A to the serial bus or DMA disk using handshaking. Normally, one or more devices will have been instructed to listen using the routine LISTN (\$FFB1) before this routine is called.

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

**\$FFAB****UNTLK****Registers Altered:** .A

Commands all devices on the serial bus or the DMA disk to stop talking (see TALK, \$FFB4).

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

**\$FFAE****UNLSN****Registers Altered:** .A

Commands all devices on the serial bus or the DMA disk to stop listening (see LISTN, \$FFB1).

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

**\$FFB1****LISTN****Registers Altered:** .A

Commands a device on the serial bus or the DMA disk to listen. The device number (\$04-\$1F) must be in .A. Also, if it has not previously been set, the device number must be stored in the current device number (\$AE) before this routine is called.

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

**\$FFB4****TALK****Registers Altered:** .A

Commands a device on the serial bus or the DMA disk to talk. The device number (\$04-\$1F) must be in .A. Also, if it has not previously been set, the device number must be stored in the current device number (\$AE) before this routine is called.

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

**\$FFB7****READSS****Registers Altered:** .A

Returns the current value of the I/O status byte (\$90) in .A. The bits of the status byte have various meanings depending on the device accessed. In general, a value of zero indicates no error. Some of the meanings of the status bits are outlined in the following table.

<i>Bit</i>	<i>Tape I/O</i>	<i>Serial I/O</i>	<i>RS232 I/O</i>
0	—	timeout write	parity error
1	—	timeout read	framing error
2	short block	—	receiver buffer overrun
3	long block	—	receiver buffer empty
4	read error	—	clear to send missing
5	checksum error	—	—
6	end of file	end or identify	data set ready missing
7	end of tape	device not present	break detected

For the 1541 disk drive, the end or identify bit set usually means the end of file has been reached. Also, for a LOADSP (\$FFD5) with the verify flag set, bit 4 set means that a verify error was found.

**\$FFBA SETLFS**  
**Registers Altered:** none

Prepares for a call to OPEN (\$FFC0), LOADSP (\$FFD5), or SAVESP (\$FFD8). Before calling the routine, set .A to the logical file number to be associated with the file (needed for OPEN only), .X to the device number, and .Y to the secondary address to be sent to the device for OPEN; for LOADSP, a zero for this secondary address causes a relocated load. If no secondary address is needed, set .Y to \$FF.

**\$FFBD SETNAM**  
**Registers Altered:** none

Prepares for a call to OPEN (\$FFC0), LOADSP (\$FFD5), or SAVESP (\$FFD8). Before calling this routine, set .A to the length of the file name, .X to the low byte of the address of the file name, and .Y to the high byte of the address of the file name. If no file name is needed, set .A to \$00.

**\$FFC0 OPEN**  
**Registers Altered:** .A, .X, .Y

This routine is vectored through \$0318-\$0319. It opens the logical file specified in calls to SETLFS (\$FFBA) and SETNAM (\$FFBD).

If an error occurs, the message flag (see SETMSG, \$FF90) is consulted to determine what message (if any) to output. The carry is returned set, and .A contains the error number. Possible errors are as follows:

\$01 = The logical file table is full.

\$02 = The specified logical file is already open.

\$04 = The specified file name is not found on the specified device.

\$05 = The specified device is not present.

The system I/O status byte may also be checked for errors. It is read using READSS (\$FFB7).

### **\$FFC3 CLOSE**

**Registers Altered: .A, .X**

This routine is vectored through \$031A-\$031B. This routine closes the logical file specified by the value in .A.

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

### **\$FFC6 CHKIN**

**Registers Altered: .A, .X**

This routine is vectored through \$031C-\$031D. The logical file specified by the value of .X, which must have been previously opened using OPEN (\$FFC0), is designated as an input channel.

If an error occurs, the message flag (see SETMSG, \$FF90) is consulted to determine what message (if any) to output. The carry is returned set, and .A contains the error number. The possible errors are as follows:

\$03 = The specified logical file is not open.

\$05 = The specified device is not present.

\$06 = The specified device is not an input device.

The system I/O status byte may also be checked for errors. It is read using READSS (\$FFB7).

### **\$FFC9 CHOUT**

**Registers Altered: .A, .X**

This routine is vectored through \$031E-\$031F. The logical file specified by the value of .X, which must have been previously opened using OPEN (\$FFC0), is designated as an output channel.

If an error occurs, the message flag (see SETMSG, \$FF90) is consulted to determine what message (if any) to output. The carry is returned set, and .A contains the error number. The possible errors are as follows:

\$03 = The specified logical file is not open.

\$05 = The specified device is not present.

\$07 = The specified device is not an output device.

The system I/O status byte may also be checked for errors. It is read using READSS (\$FFB7).

**\$FFCC      CLRCH**

**Registers Altered: .A, .X**

This routine is vectored through \$0320-\$0321. The input and output channels are cleared. If the serial bus or DMA disk was in use, an unlisten or untalk command is sent. The input and output devices are reset to the keyboard and screen.

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

**\$FFCF      BASIN**

**Registers Altered: .A**

This routine is vectored through \$0322-\$0323. This subroutine returns 1 byte in .A from the current input channel. For all devices except the keyboard, calls to OPEN (\$FFC0) and CHKIN (\$FFC6) must precede this call. The channel remains open following the call to this routine.

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

For the keyboard, the first call to this routine turns on the cursor at its current location and receives input (including function key definitions) that is echoed to the screen until a carriage return is detected. A full screen of characters may be entered. When the carriage return is received, the routine returns with the first character in .A. Each subsequent call returns the next character. The input is complete when a carriage return (\$0D) is returned.

**\$FFD2      BSOUT**

**Registers Altered: none\***

This routine is vectored through \$0324-\$0325. The byte in .A is sent to the output channel. For all devices except the screen, calls to OPEN (\$FFC0) and CHOUT (\$FFC9) must precede this call. The channel remains open following a call to this routine.

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

\*When used to send data to device 1 (the cassette) or 2 (RS-232), this routine may alter the value of .A on return.

**\$FFD5****LOADSP****Registers Altered: .A, .X, .Y**

Loads a file into memory or verifies a file against memory. For a relocated load, .X contains the low byte of the load address and .Y contains the high byte. After storing this load address, the routine vectors through \$032E-\$032F. For a load, .A should be \$00. For a verify operation in which the designated memory is compared with the file but not changed, .A should be \$01 (or another nonzero value). The call to this subroutine must be preceded by calls to SETLFS (\$FFBA) and SETNAM (\$FFBD). If the secondary address sent with SETLFS is \$00, then the load is relocated to the address given in .X and .Y and the first 2 bytes of the file are ignored; otherwise the first 2 bytes of the file specify the load address. In any case, .X returns with the low byte of the last address loaded (+1) and .Y with the high byte of the last address loaded (+1). For a verify, the status byte is \$00 for a valid compare and \$10 when a difference is found.

If an error occurs, the message flag (see SETMSG \$FF90) is consulted to determine what message (if any) to output. The carry is returned set, and .A contains the error number. The possible errors are as follows:

- \$00 = The routine was terminated by the STOP key.
- \$04 = The specified file name is not found on the specified device.
- \$05 = The specified device is not present.
- \$08 = The file name was missing.
- \$09 = The specified device is illegal for this purpose.

The system I/O status byte may also be checked for errors. It is read using READSS (\$FFB7).

**\$FFD8****SAVESP****Registers Altered: .A, .X, .Y**

Saves memory into a file. The address to start the save must be stored on zero page, low byte followed by high byte. This zero page location must then be placed in .A. The low byte of the address to stop the save (plus 1) must be placed in .X and the high byte in .Y. After these addresses are stored, the routine vectors through \$0330-\$0331. The call to this subroutine must be preceded by calls to SETLFS (\$FFBA) and SETNAM (\$FFBD).

If an error occurs, the message flag (see SETMGS, \$FF90) is consulted to determine what message (if any) to output. The carry is returned set, and .A contains the error number. The possible errors are as follows:

\$00 = The routine was terminated by the STOP key.

\$05 = The specified device is not present.

\$08 = The file name was missing.

\$09 = The specified device is illegal for this purpose.

The system I/O status byte may also be checked for errors. It is read using READSS (\$FFB7).

### **\$FFDB      SETTIM**

**Registers Altered:** none

Sets the system clock. This clock consists of 3 bytes (24 bits) on zero page (\$A3-\$A5). The low byte of the desired setting must be in .A, the middle byte in .X, and the high byte in .Y. The clock is updated by the normal system IRQ service routine. If the system IRQ service is not done, calls to UDTIM (\$FFEA) may be used to update it. The normal system interrupt is a raster interrupt that occurs every 1/60th of a second (on NTSC systems). The clock is incremented by 1 until it reaches \$4F1A00, which is 24 hours; then it is reset to zero. During some I/O operations, interrupts are disabled. This affects the clock.

### **\$FFDE      RDTIM**

**Registers Altered:** .A, .X, .Y

Reads the system clock. The low byte of the clock setting is returned in .A, the middle byte in .X, and the high byte in .Y. See SETTIM (\$FFDB) for a description of the system clock.

### **\$FFE1      STOP**

**Registers Altered:** .A, .X

This routine is vectored through \$0326-\$0327. This routine looks at a zero page location (\$91) to determine if the STOP key was pressed. This location is updated by the system IRQ service routine or UDTIM (\$FFEA). If the STOP key was detected, this routine calls CLRCH (\$FFCC) and clears the keyboard queue. If the Z flag is set on exit, the STOP key was detected; if it is clear, the STOP key was not detected.

### **\$FFE4      GETIN**

**Registers Altered:** .A, .X, .Y

This routine is vectored through \$0328-\$0329. This subroutine returns 1 byte in .A from the current input channel. For all devices except the keyboard, calls to

OPEN (\$FFC0) and CHKIN (\$FFC6) must precede this call. The channel remains open following the call to this routine.

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

For the keyboard, this routine gets 1 byte from the queue. If the queue is empty, a zero is returned in .A.

**\$FFE7 CLALL**

**Registers Altered: .A, .X**

This routine is vectored through \$032A-\$032B. This routine clears all channels with a call to CLRCH (\$FFCC) and closes all logical files.

The system I/O status byte may be checked for errors. It is read using READSS (\$FFB7).

**\$FFEA UDTIM**

**Registers Altered: .A, .X**

Increments the system clock (\$A3-\$A5) and compares it with \$4F1A01 to determine when to reset to zero. Also, this routine checks for the STOP key and sets the STOP key flag (\$91), which is used by STOP (\$FFE1). This routine is normally called by the system IRQ service routine. See SETTIM (\$FFDB) for a description of the system clock.

**\$FFED SCRORG**

**Registers Altered: .X, .Y**

Returns \$28 (the screen width) in .X and \$19 (the screen height) in .Y to indicate the physical size of the Plus/4 screen. This subroutine is useful only for programs intended to run on other Commodore computers with a different screen size (e.g., the VIC-20).

**\$FFF0 PLOT**

**Registers Altered: .A, .X, .Y**

When called with the carry flag clear, sets the cursor position. The new cursor line number should be in .X, and the new cursor column should be in .Y. The screen window is returned to its default (the whole screen). When called with the carry flag set, this routine returns the current cursor position. The cursor line number is returned in .X and the cursor column in .Y.

**\$FFF3      IOBASE**  
**Registers Altered: .X, .Y**

Returns \$00 (the low byte of the I/O base address) in .X and \$FD (the high byte of the I/O base address) in .Y. This subroutine is useful only for programs intended to run on other Commodore computers.

**\$FFF6      RESET**  
**Registers Altered: all**

Does a warm reset of the Plus/4. If the STOP key is detected, this routine goes to the MONITOR, leaving the contents of memory intact. If STOP is not detected, the Plus/4 goes to BASIC with the contents of memory reinitialized.

## Interfacing with BASIC

Sometimes it is desirable to combine BASIC and machine language to get the advantages of both. This can be done in a number of ways, including changing the operating system vectors (\$0300–\$0331). The most common methods include the USR function, the SYS command, and adding a BASIC wedge (a custom command set). These methods are discussed in this section.

### The USR Function

USR is a floating point function. The syntax is

**USR(x)**

where *x* is a numeric expression. The USR function places the value of its argument in floating point accumulator #1 (\$61–\$66) and then passes control of processing to the address contained in the USR vector at \$0501–\$0502 (1281–1282). These locations are initialized so as to cause an ILLEGAL QUANTITY error if the USR function is called without changing their values. The USR function returns to the BASIC program the value in floating point accumulator #1 when its execution is complete. The machine code routine pointed to by the USR vector must be terminated with an RTS to return control to BASIC.

***The Floating Point Accumulator*** Locations \$61–\$66 comprise the value contained in floating point accumulator #1. The number is stored in exponential form. The absolute value resides in \$61–\$65. Bit 7 (the sign bit) of location \$66 indicates the sign of the number (if bit 7 is set to 1, the number is negative; otherwise it is nonnegative). A value of zero is indicated by setting \$61 (which is

the exponent) to zero. The nonzero absolute value of the floating point accumulator can be calculated using the following formula:

$$[2^{-8} * (\$62) + 2^{-16} * (\$63) + 2^{-24} * (\$64) + 2^{-32} * (\$65)] * 2^{[(\$61)-128]}$$

where (\$62) means the contents of memory location \$62.

**Example:** In this example, the USR function is defined as a routine that returns the current contents of the specified register of the graphics chip. BASIC program is as follows:

```
10 POKE1281,0:POKE1282,32
20 PRINT"WHICH REGISTER IS DESIRED";:INPUTN%
30 M=USR(N%):IFM<0THENPRINT"ILLEGAL REGISTER
    NUMBER":GOTO20
40 PRINT"REGISTER";N%;"CONTAINS";M
```

### Line-by-Line Explanation

- 10 These two POKEs change the USR vector to point to the new routine.
- 20 The user is asked for an integer register number.
- 30 The USR function performs error checking, returning a negative value if an illegal register number was requested. An illegal request causes control to return to line 20.
- 40 The contents of the register are displayed.

The machine code subroutine to process the USR function can be entered with the machine-language monitor. This subroutine converts the register number in the floating point accumulator to a 1-byte index, checks it for validity, and gets the appropriate contents of the graphics chip register. The subroutine then converts this value into a floating point number and places it in the floating point accumulator for return. A negative number is returned to indicate an error.

. 2000	A5 66	LDA \$66	Look at sign of input register number.
. 2002	30 3A	BMI \$203E	If negative, it is invalid, so return.
. 2004	A5 61	LDA \$61	Look at the exponent of input register number.
. 2006	F0 0E	BEQ \$2016	If zero, it is zero, so skip conversion.
. 2008	38 *	SEC	Prepare to subtract.
. 2009	A9 88	LDA #\$88	Load accumulator with 136.
. 200B	E5 61	SBC \$61	Subtract the exponent.
. 200D	A8	TAY	This is the number of times to divide by 2.
. 200E	A5 62	LDA \$62	Get the mantissa.
. 2010	88	DEY	Decrement number of times to divide.
. 2011	30 03	BMI \$2016	If negative, then done, so go on.
. 2013	4A	LSR	Divide by 2.

• 2014 D0 FA	BNE \$2010	This is always taken.
• 2016 AA	TAX	Put converted register number in .X.
• 2017 E0 20	CPX #\$20	Compare to highest valid register number (+1).
• 2019 B0 1F	BCS \$203A	If greater than or equal, exit with error.
• 201B A0 08	LDY #\$08	This keeps track of multiplications by 2.
• 201D BD 00 FF	LDA \$FF00,X	Get the register contents.
• 2020 F0 15	BEQ \$2037	If zero, a zero must be returned.
• 2022 30 04	BMI \$2028	When negative, no more multiplying required.
• 2024 88	DEY	Decrement multiplication index.
• 2025 0A	ASL	Multiply by two.
• 2026 D0 FA	BNE \$2022	This is always taken.
• 2028 85 62	STA \$62	Store the mantissa in floating point accumulator.
• 202A A9 00	LDA #\$00	Get a zero.
• 202C A2 03	LDX #\$03	Load .X with size of rest of accumulator.
• 202E 95 63	STA \$63,X	Fill rest of accumulator with zero.
• 2030 CA	DEX	Decrement position in accumulator.
• 2031 10 FB	BPL \$202E	If not done, go back.
• 2033 18	CLC	Prepare to add.
• 2034 98	TYA	Get multiplication index.
• 2035 69 80	ADC #\$80	Add 128 to get exponent for accumulator.
• 2037 85 61	STA \$61	Store in exponent.
• 2039 60	RTS	Return.
• 203A A9 80	LDA #\$80	Get value with sign bit set.
• 203C 85 66	STA \$66	Put in sign register to indicate error.
• 203E 60	RTS	Return.

## The SYS Command

The SYS command transfers control from a BASIC program to a machine-language subroutine. When the subroutine is completed with an RTS instruction, control is returned to the BASIC program. When it is necessary to pass parameters to and from the machine-language subroutine, the values are normally stored in memory with POKE commands and retrieved using PEEK.

*Note:* You can set the values of the accumulator, X register, Y register, and status register on entry to your subroutine automatically by POKEing their desired values into 2034–2037 (\$07F2–\$07F5). The values of the registers on exit from your subroutine are returned in the same locations.

**Example:** In this example, a machine code routine to fill an area of the screen with a given character is called from a BASIC program using the SYS command. The BASIC program is as follows:

```
10 PRINTCHR$(147)"INPUT FILL CHARACTER ";:GETKEYA$:
N=ASC(A$)
```

```

20 IFN<32THEN10
30 IFN<64THENM=N:GOTO100
40 IFN<96THENM=N-64:GOTO100
50 IFN<128THENM=N-32:GOTO100
60 IFN<160THEN10
70 IFN<192THENM=N-64:GOTO100
80 IFN<224THENM=N-128:GOTO100
90 IFN<255THENM=N-128:ELSEM=126
100 PRINTA$
110 PRINT"INPUT UPPER LEFT CORNER (X,Y)":;INPUTX1,Y1
120 PRINT"INPUT LOWER RIGHT CORNER (X,Y)":;INPUTX2,Y2
130 POKE216,X1:POKE217,Y1:POKE218,X2:POKE219,Y2:
    POKE220,M
140 SYS8192:IFPEEK(221)>127THENPRINT
    "ERROR IN COORDINATES"

```

### Line-by-Line Explanation

- 10 Ask user for the character to fill with, and get one key.
- 20-90 Translate CHR\$ code into screen code for use by the machine code routine.
- 100 Echo the character back to the user.
- 110 Ask for, and get, the upper left coordinate of area to fill.
- 120 Ask for, and get, the lower right coordinate of area to fill.
- 130 Set up the parameters in \$D8-\$DC for the machine code routine.
- 140 Call the machine-language routine. Check for error return (\$DD contains \$FF), and put out error message if necessary.

The machine code subroutine to process the SYS command can be entered with the machine-language monitor. The subroutine fills the specified area of the screen with the specified character.

• 2000 A9 03	LDA #\$03	Byte with low two bits set.
• 2002 85 DF	STA \$DF	Initialize high byte of screen pointer.
• 2004 A5 D9	LDA \$D9	Get upper left corner y-coordinate.
• 2006 C9 19	CMP #\$19	Compare to maximum y value (plus one).
• 2008 B0 52	BCS \$205C	If greater than, or equal to, return with error.
• 200A 0A	ASL	Multiply y-coordinate by two.
• 200B 0A	ASL	Multiply by two again.
• 200C 65 D9	ADC \$D9	Add y-coordinate to get y-coordinate times five.

. 200E	0A	ASL	Multiply by two.
. 200F	0A	ASL	Multiply by two.
. 2010	26 DF	ROL \$DF	Double precision is now required.
. 2012	0A	ASL	Multiply by two low byte.
. 2013	26 DF	ROL \$DF	Multiply by two high byte.
. 2015	85 DE	STA \$DE	Store low byte of screen pointer.
. 2017	A5 D8	LDA \$D8	Get upper left corner x-coordinate.
. 2019	C9 28	CMP #\$28	Compare to maximum x value (plus one).
. 201B	B0 3F	BCS \$205C	If greater than, or equal to, return with error.
. 201D	65 DE	ADC \$DE	Add to low byte of screen pointer.
. 201F	85 DE	STA \$DE	Store low byte of screen pointer.
. 2021	90 02	BCC \$2025	If no carry, go on.
. 2023	E6 DF	INC \$DF	Increment high byte of screen pointer.
. 2025	A5 DA	LDA \$DA	Get lower right corner x-coordinate.
. 2027	C9 28	CMP #\$28	Compare to maximum x value (plus one).
. 2029	B0 31	BCS \$205C	If greater than, or equal to, return with error.
. 202B	38	SEC	Prepare to subtract.
. 202C	E5 D8	SBC \$D8	Subtract upper left x-coordinate.
. 202E	30 2C	BMI \$205C	If result is negative, return with error.
. 2030	85 DD	STA \$DD	Number of columns is in \$DD.
. 2032	A5 DB	LDA \$DB	Get lower right corner y-coordinate.
. 2034	C9 19	CMP #\$19	Compare to maximum y value (plus one).
. 2036	B0 24	BCS \$205C	If greater than, or equal to, return with error.
. 2038	38	SEC	Prepare to subtract.
. 2039	E5 D9	SBC \$D9	Subtract upper left y-coordinate.
. 203B	30 1F	BMI \$205C	If result is negative, return with error.
. 203D	AA	TAX	Number of lines is in .X.
. 203E	A5 DC	LDA \$DC	Get fill character.
. 2040	A4 DD	LDY \$DD	Set .Y to number of columns.
. 2042	91 DE	STA (\$DE),Y	Store fill character on screen.
. 2044	88	DEY	Decrement column count.
. 2045	10 FB	BPL \$2042	If not done, go back.
. 2047	CA	DEX	Decrement line count.
. 2048	30 0D	BMI \$2057	If finished, go to success return.
. 204A	18	CLC	Prepare to add.
. 204B	A5 DE	LDA \$DE	Get low byte of screen pointer.
. 204D	69 28	ADC #\$28	Add one line.
. 204F	85 DE	STA \$DE	Store in low byte of screen pointer.
. 2051	90 EB	BCC \$203E	If carry clear, go back to continue.
. 2053	E6 DF	INC \$DF	Increment high byte of screen pointer.
. 2055	B0 E7	BCS \$203E	This is always taken.
. 2057	A9 00	LDA #\$00	Get ready for success return.
. 2059	85 DD	STA \$DD	Store zero in return status location.
. 205B	60	RTS	Return to BASIC program.
. 205C	A9 FF	LDA #\$FF	Get ready for error return.
. 205E	85 DD	STA \$DD	Store \$FF in return status location.
. 2060	60	RTS	Return to BASIC program.

## Adding a BASIC Wedge

BASIC retrieves each character from RAM by calling a routine referred to as CHRGET, which is located at \$0473. Because this is in RAM (initialized on power up), it can be altered to recognize and process symbols as desired by the programmer. It is possible to add BASIC commands in this way.

**Example:** This example shows one way to add a direct-mode-only command to BASIC. The command is the English pound symbol (£), and it increments the border color. The first step is to enter the machine-language wedge through the machine-language monitor.

. 2000 48	PHA	Save the character on the stack.
. 2001 A5 9A	LDA \$9A	Look at the message flag.
. 2003 10 0B	BPL \$2010	If positive, then this is not direct mode so leave.
. 2005 68	PLA	Get the character.
. 2006 C9 5C	CMP #\$5C	Compare with the English pound symbol.
. 2008 D0 07	BNE \$2011	Not one, so leave.
. 200A EE 19 FF	INC \$FF19	Increment the border color.
. 200D 4C 73 04	JMP \$0473	Go to get the next character.
. 2010 68	PLA	Get the character.
. 2011 8D 3E FF	STA \$FF3E	Back to ROM.
. 2014 4C 84 04	JMP \$0484	Go back to the operating system routine.

After the wedge code is in place, change the statement at \$0481 from STA \$FF3E to JMP \$2000. Then exit to BASIC. When in direct mode, you can input the English pound symbol to increment the border color. Everything else works normally.

## Relocating BASIC RAM

BASIC keeps track of the RAM it is permitted to use in a series of pointers, \$2B-\$38. Some care must be taken when you change these pointers. The main reason for moving BASIC is to reserve some of the RAM usually used by BASIC for a machine-language routine.

Moving the top of BASIC down is the easiest way to restrict the RAM available to BASIC. This can be done with the following sequence of commands:

**POKE 55,low byte of new top of RAM plus one**  
**POKE 56,high byte of new top of RAM plus one**  
**CLR**

To execute machine code that is located underneath the BASIC or operating system ROM, the routine must begin somewhere NOT underneath the ROMs and bank in RAM (see the section on banking) before transferring control. The transfer code could be located in the cassette buffer (\$0333-\$03F2) or some other safe location. The machine code routine must bank ROM in before returning to BASIC. The pointer to the top of memory at 1331-1332 (\$0533-\$0534) must also be changed in some applications.

Moving the bottom of BASIC up can be accomplished with the following:

**POKE 43,low byte of new bottom of RAM**  
**POKE 44,high byte of new bottom of RAM**  
**POKE new bottom of RAM minus one,O**  
**NEW**

*Note:* The pointer to the bottom of memory at 1329-1330 (\$0531-\$0532) does not appear to be used by BASIC.

After executing these commands, machine code could be placed in the vacated RAM at \$1000. The GRAPHIC command moves the bottom of BASIC up \$3000 from its location before the command is given, and GRAPHICCLR moves it down \$3000 back to its original location. But the graphic screen itself is always in the same position at \$1800-\$3FFF. Care must be taken to avoid conflicts.

---

# **6** Using Peripheral Devices

---

The Plus/4 is compatible with the 1541 disk drive and Commodore's printers available for the Commodore 64 and VIC 20, such as the MPS-801 and 1526. A new Direct Memory Access (DMA) fast disk drive has been promised, but as of this writing, it is not available. The Plus/4 is NOT compatible with the DATASSETTE cassette player used with the earlier computers (the C2N/1530) but rather can use a new Commodore DATASSETTE, the 1531. The Plus/4 has a new RS232 output that is not completely compatible with the 64. Therefore, you cannot use the VIC modem or automodems sold for the 64 and VIC 20. The Plus/4 is compatible with the model 1660 MODEM/300. The Plus/4 also uses a new joystick, the T-1341.

## **The Disk Drive**

The disk drive is a hardware device that is controlled by software known as the Disk Operating System (DOS). Commodore disk drives generally have the DOS in the disk drive rather than in the computer. Hence, the drives are referred to as "intelligent." The 1541 has its own built-in 6502 microprocessor, a DOS ROM, and some RAM. Programs running in the computer send commands to the DOS through the serial bus. The BASIC section of this chapter covers the commands recognized by the DOS. This information will be useful for machine-language programmers as well.

Most often the data and programs stored on a diskette are organized into files. The only exception is when direct-access programming is used to put information on the diskette. Each file is identified with a file name that is assigned to it when it is created. The file name of each file on the same diskette must be unique. Occasionally (where noted below), it is useful to refer to a file (or group of files) by using a "wild card" in the file name position of a command. The Plus/4 supports two such wild cards. One is the asterisk, which when used in a file name means that any characters appearing at and after the position of the asterisk match. For example, the use of "F\*" as a file name matches all file names on the diskette that

begin with the letter F. The second wild card is the question mark, which is used to mean any character in that specific position matches. For example, the use of "F??PRG" as a file name matches all file names on the diskette starting with F, followed by two characters, and ending in PRG. The key difference between the two wild cards is that the asterisk matches all combinations of any number of characters and must be at the end of the string, whereas the question mark matches any single character and can be in any position in the string.

The information in this section has been verified with a 1541 disk drive. The BASIC and machine-language access to the DMA disk should be the same. However, the commands that the drive itself recognizes may be different. Please refer to the manual for specific information on the DMA disk.

## Using a Disk Drive with BASIC

The disk drive can be used to save BASIC programs or to create and manage data files through BASIC. There are different types of files that can be used for information storage, or a program can access specific areas on the disk directly. Each type of file and all the direct-access commands are covered in this section.

A number of maintenance functions are available for use with diskettes. These are detailed in Chapter 1 and reviewed in this chapter (see Diskette Maintenance). In particular, a new diskette must be formatted before its first use.

If the red light on the disk drive blinks at any time, an error has occurred. To read the error, type

**PRINT DS\$**

For an explanation of DOS errors, see Appendix A. A program can check for DOS errors by looking at the reserved variable DS. If DS is not zero, then an error has occurred and DS gives the error number. The error message can then be printed by printing DS\$. Error checking should be done frequently in BASIC programs that use the disk. Most errors in disk I/O are reported in DS and DSS. A notable exception is the end-of-file that is returned in the status variable, ST with bit 6 set (a decimal value of 64).

## Saving, Loading, and Verifying Programs

Since one of the main uses for a disk drive is to save and retrieve BASIC programs, BASIC has built-in commands to make it easy. You can save programs with the DSAVE (or the SAVE) command.

**DSAVE *filename,Ddrive,Uunit***

The *filename* is a string expression of up to 16 characters by which the program will be known on the disk. Each file on a disk must have a unique name. The *drive* is the optional drive number (which is not needed for the 1541). The *unit* is optional; it is the device number for the disk and is assumed to be 8 if not specified.

#### **SAVE filename,device**

The *filename* is the same as above. The *device* is the device number for the disk. Neither of these parameters may be omitted in using the SAVE command to save a program on the disk.

To retrieve the program, the DLOAD (or LOAD) command can be used.

#### **DLOAD filename,Ddrive,Uunit**

The *filename* is the name used when saving the program. The *drive* and *unit* are the same as DSAVE and are optional. Wild cards may be used in the file name of a LOAD command. The first file in the directory of the diskette with a matching name is loaded.

#### **LOAD filename,device,absolute/relative flag**

Again, the *filename* is the same name used to save the program, and the *device* is the disk device number. The *absolute/relative flag* is optional. If it is omitted or is 0, the file is loaded as a BASIC program starting at the beginning of BASIC RAM. If it is 1, the file is loaded at the location from which it was saved.

To compare a program in memory with one stored on disk without altering memory, use the VERIFY command. It reports if the file is not identical to the program in memory.

#### **VERIFY filename,device,absolute/relative flag**

All of the parameters are the same as for LOAD.

## **Using Data Files**

Several types of data files can be created, but they all share the BASIC commands required to access them. The first step in creating any data file is to open it for use.

#### **OPEN filenumber,device,secondary,filename,type,mode**

The *filenumber* is the logical number to be associated with this file. It is for reference purposes only and can be 0 - 127 (128-255 will send a linefeed character following every carriage return). The *device* is the device number of the disk drive. The *secondary* address is the channel number to be used. For a data file it should be 2-14, and when more than one data file is in use, each must use a different channel. Channel numbers 0 and 1 are used by the DOS for saving and loading and are not normally used in programs. Channel number 15 is the command channel, discussed in the diskette maintenance and direct-access programming sections. The last parameter consists of the file name (a string expression of up to 16 characters), the type of file, and the mode of access separated by commas. If omitted, the mode is assumed to be read, and the type of file is the same as when the file was created. The type of file can be

S for sequential

P for program

L for relative

U for user

The mode can be

W for write

R for read

After the file is open, output may be directed to it by referring to its logical file number in a PRINT# or CMD command. Input may be received from it by referring to its logical file number in an INPUT# or GET# command.

When the input to or output from a file is complete, the file must be closed with

**CLOSE filenumber**

where *filenumber* is the logical file number of the file to close.

## **Sequential Files**

The most straightforward type of data file is the sequential file. It is opened with a file type parameter of S. After a sequential file is opened, each byte that is sent to it is stored sequentially on the disk. The format of the bytes written to a sequential file must be designed with the method of retrieval in mind. If they are to be read one at a time by the GET# command, any format is all right. However, if they are to be read by the INPUT# command, care must be taken to store comma characters between values and carriage return characters between lines of input.

Example:

```
10 OPEN1,8,2,"DATAFILE,S,W"
20 IFDS<>OTHENPRINTDS$:CLOSE1:END
30 FORI=1TO10
40 PRINT"WHAT ARE" I "TH VALUES OF X AND Y";:INPUTX,Y
50 PRINT#1,X,":,Y
60 IFDS=OTHENNEXT:ELSEPRINTDS$:CLOSE1:END
70 CLOSE1
80 OPEN1,8,2,"DATAFILE"
90 I=1
100 INPUT#1,X,Y
110 PRINT I "TH VALUES ARE",X,Y
120 IFST=OTHENI=I+1:GOTO100
130 CLOSE1
```

### Line-by-Line Explanation

- 10 Open a sequential file called DATAFILE on device 8 using channel 2 for writing.
- 20 Check for an error. If one is found, print message and stop.
- 30 I runs from 1 to 10.
- 40 Accept input from the keyboard for the values of X and Y.
- 50 Output the values of X and Y to the data file separated by a comma. A carriage return is automatically sent because no semicolon is found at the end of the statement.
- 60 Check for an error. If there is no error, get next I. If there is an error, print message and stop.
- 70 Close the data file just created.
- 80 Now, open DATAFILE for reading. Since the type and mode are omitted, they are assumed to be sequential and read.
- 90 I counts the records read.
- 100 Read a set of X and Y.
- 110 Print the values read.
- 120 Check for end-of-file. If not found, add one to I and read next record.
- 130 Close the file.

## Program Files

The program file type is used for storage of programs. It can also be used for data storage and is opened with a file type parameter of P. Its format is identical to the sequential file format. When a program file is created by a save command, the first 2 bytes of the file contain the address from which the program was saved. The following example illustrates this. Type in the program and save it with the following command:

```
DSAVE"PROGRAM"
```

Then run the program. It reads in the load address that was saved in the file and prints it out.

Example: 10 OPEN1,8,2,"PROGRAM"  
 20 IFDS<>OTHENPRINTDS\$:CLOSE1:END  
 30 GET#1,AL\$:GET#1,AH\$  
 40 A=ASC(AH\$)\*256+ASC(AL\$)  
 50 PRINT"THE LOAD ADDRESS IS ";A  
 60 CLOSE1

### Line-by-Line Explanation

- 10 Open the program file with read access (the default).
- 20 Check for errors. If there is an error, quit.
- 30 Get the first 2 bytes in the file.
- 40 Calculate the load address.
- 50 Output the load address.
- 60 Close the file.

## Relative Files

The types of files discussed previously store information sequentially; that is, to read a single item somewhere in the middle of the file, you must first read all the items before it. To reread the item, you must start at the beginning again. When data will be accessed in a "random" fashion (i.e., you are equally likely to want to access any piece of data), a relative file can be used.

A relative file is opened with a file-type parameter of L. A relative file is a series of "records." Each record contains the data for one individual in the database and is of a predetermined length. When you create a relative file, the record length must be specified. The syntax is

**OPEN filenumber,device,channel,"filename,L,"+CHR\$(length)**

where *filenumber* is the logical file number (0-127), *device* is the disk drive's device number, *channel* is the channel to use (2-14), and *filename* is the name by which the file is known. The record *length* is a number (1-255) or numeric expression. The file type and record length may be omitted when you are opening an existing relative file.

To communicate with a relative file, you must open the command channel of the disk drive. The following command does that:

**OPEN filenumber,device,15**

where *filenumber* is the logical file number (0-127, but not the same as the relative file itself), and *device* is the drive's device number.

To read or write a record in the relative file, the file pointer must be positioned to the desired record with the following command:

**PRINT#filenumber,"P"CHR\$(chnl)CHR\$(rclo)CHR\$(rchi)  
CHR\$(pos)**

where *filenumber* is the logical file number assigned to the command channel, *chnl* is the channel number assigned to the relative file in its OPEN command, *rclo* is the low byte of the desired record number, *rchi* is the high byte of the desired record number, and *pos* is the position in the record. Errors should be caught after this statement by examining DS and DS\$. If the record requested does not exist, DS returns a 50. It should be noted, however, that whenever a new sector on the disk is added to the file, every record that it would contain exists, even if that record has not been written to. The first byte of a record that has not been written to is always a CHR\$(255).

This example program sets up a relative file containing names and phone numbers. The record size is 21 (10 for the name, then a separating comma, and 10 for the phone number). Each time the program is run, the user may add a name and number or retrieve a name and number.

**Example:**

```

10 OPEN1,8,2,"PHONES,L,"+CHR$(21)
20 IFDS<>0THENPRINTDSS:CLOSE1:END
30 OPEN15,8,15
40 IFDS<>0THENPRINTDSS:GOTO170
50 PRINT"DO YOU WANT TO ADD A NAME";:INPUTA$ 
60 IFLEFTS$(A$,1)="Y"THEN180
70 PRINT"WHOSE NUMBER DO YOU NEED";:INPUTN$ 
80 N$=LEFTS$(N$,10)
90 R=1
100 PRINT#15,"P"CHR$(2)CHR$(RAND255)CHR$(INT(R/256))CHR$(1)
110 IFDS<>0THENPRINTDSS:GOTO170

```

```

120 INPUT#1,NA$,NU$
130 IFNA$=N$THEN160
140 IFASC(NA$)=255THENPRINTNS;" NOT FOUND":GOTO170
150 R=R+1:GOTO100
160 PRINTNS;'"S NUMBER IS ";NU$
170 CLOSE1:CLOSE15:END
180 PRINT"WHAT IS THE NAME"::INPUTNA$
190 NA$=LEFT$(NA$,10)
200 PRINT"WHAT IS ";NA$;'"S NUMBER"::INPUTNU$
210 NU$=LEFT$(NU$,10)
220 R=1
230 PRINT#15,"P"CHR$(2)CHR$(RAND255)CHR$(INT(R/256))CHR$(1)
240 D=DS:IFD=0THEN260
250 IFD=50THENGOTO280:ELSEPRINTDSS:GOTO310
260 INPUT#1,N$
270 IFASC(N$)<>255THENR=R+1:GOTO230
280 PRINT#15,"P"CHR$(2)CHR$(RAND255)CHR$(INT(R/256))CHR$(1)
290 D=DS:IFD<>0ANDD<>50THENPRINTDSS:GOTO310
300 PRINT#1,NA$+", "+NU$
310 CLOSE1:CLOSE15:END

```

### Line-by-Line Explanation

- 10 Open the relative file called PHONES with record size 21.
- 20 If an error occurred, then quit.
- 30 Open the command channel of the disk drive.
- 40 If an error occurred, then quit.
- 50 Find out if this is add or retrieve.
- 60 If it is add, go down to line 180.
- 70 Find out the name to search for.
- 80 Trim off unused characters.
- 90 Start with record number 1.
- 100 Position file pointer to beginning of record.
- 110 If an error occurred, then quit.
- 120 Get the name and number on this record.
- 130 If it is the correct name, go down to line 160.
- 140 If the record has never been used, the search has failed.
- 150 Go on to the next record.
- 160 Print out the name and number.
- 170 Close the files and quit.

- 180 Find out the name to add.
- 190 Trim off unusable characters.
- 200 Find out the number to add.
- 210 Trim off unusable characters.
- 220 Start with record number 1.
- 230 Position file pointer to beginning of record.
- 240 Save the error number in D. If no error, continue.
- 250 If a record not found error occurs, then here is the position to put a new record. Otherwise, inform user, and quit.
- 260 Read the name on this record.
- 270 If it is a valid name, go on to next record. Otherwise, here is the position to put a new record.
- 280 Position file pointer to beginning of record.
- 290 Read the error. If no error or a record not found error, ignore. Otherwise, inform user and quit.
- 300 Output the new name and number.
- 310 Close the files and quit.

## User Files

The final file type that can be created with the OPEN command is a user file, which is designated with a file type parameter of U. The user file type lets you enter into the directory of the diskette a file with a user-defined structure. If it is opened and written to in the normal manner, a user file has the structure of a sequential file. Designing a custom file structure requires the use of the direct-access commands detailed in the next section.

**Diskette Maintenance** A number of useful commands can be used to handle diskettes. Most of these are available in two formats on the Plus/4. The Plus/4 supports BASIC 3.5 disk handling commands that may also be sent to the drive through the command channel. The BASIC 3.5 commands detailed in Chapter 1 are reviewed. The equivalent command channel commands (which are especially useful in machine language programming) are fully described here.

To use the command channel, it must first be opened with

**OPEN filenumber,device,15**

where *filenumber* is the logical file number to be associated with the command channel and *device* is the device number of the disk. As usual, when communication through the command channel is complete it must be closed as follows:

**CLOSE** *filenumber*

where *filenumber* is the same logical file number used to open the channel.

### **Examining the Directory of a Diskette**

The directory of a diskette lists all the files stored on the diskette and the number of sectors each file occupies. The directory cannot list individual sectors allocated with direct-access commands, but the blocks free total does reflect those allocations. The BASIC 3.5 command is

**DIRECTORY** *Ddrive,Uunit,filename*

where *drive* is the optional drive number (which is not needed for the 1541), *unit* is the optional device number of the disk drive, and *filename* is an optional file name to search for. The *filename* is a literal string (in quotes) or a string expression (in parentheses). Wild cards may be used to get a directory of all those files with similar names.

Another way of examining the directory is to type

**LOAD**"\$"*,device*

where *device* is the device number of the disk drive, and then

**LIST**

*Note:* The LOAD command erases the program in BASIC memory.

### **Formatting a Diskette**

Before a diskette can be used for storing information, it must be formatted. Formatting creates data areas on the diskette that the disk drive can recognize. Formatting a diskette previously used to store information erases all the information on the diskette. To format a new diskette, the HEADER command can be used. Its syntax is

**HEADER** *diskname,Iidentification,Ddrive,ON Uunit*

where the *diskname* is a string of up to 16 characters that is displayed whenever a directory of the disk is requested. The *diskname* is a literal string (in quotes) or a string expression (in parentheses). The *identification* is two characters that the DOS uses to identify the disk. It is a good idea to give a different identification to each disk. For a brand new disk, an identification must be given, but an old disk can be cleared of information more quickly by omitting the identification parameter. The *drive number* is required (for the 1541, use 0). The *unit* number, which is optional, is the device number for the disk drive. It is set by the factory to 8. The device number can be changed in the hardware according to instructions in the disk drive manual. The device number can also be changed temporarily by the disk address change utility program on the diskette that comes with the drive. Whenever the device number is optional in a command, it is assumed to be 8 if omitted.

To format a diskette by direct command to the DOS, use

```
PRINT# filenumber,"N:diskname,identification"
```

where *filenumber* is the logical file number of the previously opened command channel, *diskname* is the name (up to 16 characters) to be displayed with the directory of the diskette and *identification* is the two characters used by DOS to identify the diskette. As with the header command, any information previously on the diskette is lost. You can reformat a reused disk more quickly by omitting the identification.

## Initializing a Diskette

The DOS keeps its own local copy of the diskette's ID and BAM (block availability map). It uses the BAM information to decide what areas of the diskette are available for writing. If DOS discovers that a diskette it is attempting to write on has a different ID from that it has stored, an error (error number 29) occurs. Unfortunately, the DOS identifies diskettes on the basis of their ID number only. If two diskettes have the same ID number, and they have been switched since the DOS last updated its information, the DOS uses an incorrect BAM. This is a disastrous occurrence. You can easily avoid this by ensuring that the DOS has the correct BAM in memory. This is done with the initialize command that is sent through the command channel:

```
PRINT# filenumber,"I"
```

where *filenumber* is the logical file number of the previously opened command channel. It is a good idea to perform an initialization whenever you switch diskettes, regardless of their respective ID numbers.

## Validating a Diskette

Whenever a file is added to a diskette, the DOS updates the directory and the BAM. Occasionally the two do not match, when, for example, a file was opened but never closed. The COLLECT or validate command must be used to remedy this situation. Never SCRATCH an unclosed file. (Unclosed files are denoted with an asterisk in the directory listing.) Do not COLLECT a diskette that has had sectors written to it with the direct-access commands described in the next section. The COLLECT command syntax is

**COLLECT Ddrive, ON Uunit**

where *drive* is the optional drive number (which is not needed for the 1541) and *unit* is the disk drive's device number. If omitted, the unit is assumed to be 8.

To collect a diskette by direct command to the DOS, use

**PRINT# filenumber,"V"**

where *filenumber* is the logical file number of the previously opened command channel.

## Deleting Files on a Diskette

To delete a properly closed file from the diskette, the SCRATCH command is used. Its syntax is

**SCRATCH filename,Ddrive,Uunit**

where *filename* is the name of the file to be deleted, *drive* is the optional drive number (which is not needed on the 1541), and *unit* is the disk drive's device number, which is assumed to be 8 if not specified. Do not SCRATCH an unclosed file. Instead, use the COLLECT command described previously. Wild cards may be used in the file name for a SCRATCH to delete every file with a matching file name, but caution is advised because one can easily delete large numbers of files by using them.

To delete a file by direct command to the DOS, use

**PRINT# filenumber,"S:filename"**

where *filenumber* is the logical file number of the previously opened command channel and *filename* is the name of the file to be deleted.

*Note:* If you inadvertently SCRATCH a valuable file, you may be able to recover it. For more information, see the direct-access programming section.

## Renaming Files on a Diskette

Occasionally it is necessary to change the name of an existing (and properly closed) file on a diskette. The BASIC RENAME command is

`RENAMEDdrive,"old name"TO"new name",Uunit`

where *drive* is the optional drive number (which is not needed on the 1541), *old name* is the original name of the file, *new name* is the final name of the file, and *unit* is the optional device number of the disk drive.

To rename a file by direct command to the DOS, use

`PRINT# filenumber,"R:new name=old name"`

where *filenumber* is the logical file number of the previously opened command channel, and *new name* and *old name* are the final and original file names.

## Copying Files on a Diskette

To create an exact duplicate of a file on the diskette, the COPY command is used. The new copy must be on the same diskette and have a different name from the original. Relative files cannot be copied with this command.

`COPYDdrive,"orig name"TOdrive,"copyname",ON Uunit`

where *drive* is the optional drive number (which is not needed on the 1541), *orig name* is the name of the original file, *copy name* is the name of the copy, and *unit* is the optional device number of the disk drive.

To copy a file by direct command to the DOS, use

`PRINT# filenumber,"C:copy name=orig name"`

where *filenumber* is the logical file number of the previously opened command channel, and *copy name* and *orig name* are the names of the copy and the original file.

**Direct-Access Programming** The DOS accepts nine commands in addition to those already described. These commands allow the programmer to read and write specific sectors on the diskette and to access the memory in the disk drive. Remember that sectors you write directly onto the diskette do not normally appear in the directory and are deallocated if the disk is validated (COLLECTed).

Direct-access programming requires opening two channels to the disk drive. The first is the command channel, which is opened with

**OPEN filenumber,device,15**

where *filenumber* is the logical file number to associate with the command channel and *device* is the disk drive's device number. The second is a direct-access channel, which is opened with

**OPEN filenumber,device,channel,"#buffer"**

where *filenumber* is the logical file number to associate with the direct-access channel, *device* is the disk drive's device number, and *channel* is a data channel to use for direct access (2–14). The 1541 has several internal buffers. Four of these are used for direct-access programming. You need not specify which one to use. Alternatively, you may do so by using *buffer* equal to 0 through 3. The 1541 memory corresponding to each buffer is as follows:

<i>Buffer</i>	<i>1541 Memory Used</i>
0	\$0300–\$03FF
1	\$0400–\$04FF
2	\$0500–\$05FF
3	\$0600–\$06FF

Each channel opened must be closed, following its use, with

**CLOSE filenumber**

The diskette is organized into tracks and sectors. There are 35 tracks, numbered 1 to 35, on a 1541 format diskette. The number of sectors within a track varies according to the following:

<i>Tracks</i>	<i>Number of Sectors</i>
1–17	21
18–24	19
25–30	18
31–35	17

The sectors are numbered starting with zero. Hence, for example, the sectors on track 1 are numbered 0–20.

The status (allocated or free) of each sector is recorded in the BAM (Block Availability Map) for the diskette. On a 1541 format disk the BAM is located on track 18 in sector 0. The directory for the diskette starts on track 18 in sector 1.

## Block Read (U1)

The contents of the designated sector are read into a buffer in the disk drive by this command. They can then be retrieved from the buffer with a GET#. The syntax is

**PRINT# filenumber,"U1:"channel;drive;track;sector**

where *filenumber* is the logical file number of the command channel, *channel* is the channel number of the direct-access channel, and *drive* is the drive number. The *track* can be any track on the diskette (1 to 35) and the *sector* any sector of that track.

The command B-R is also a block read command, but it occasionally malfunctions, so use U1.

**Example:**

```
10 OPEN15,8,15,"I"
20 IFDS<>OTHENPRINTDS$:GOTO120
30 OPEN1,8,2,"#"
40 IFDS<>OTHENPRINTDS$:GOTO120
50 PRINT"WHICH TRACK, SECTOR";:INPUTT,S
60 PRINT#15,"U1:"2;0;T;S
70 IFDS<>OTHENPRINTDS$:GOTO120
80 FORI=0TO255
90 GET#1,A$
100 PRINTASC(A$),;
110 NEXT
120 CLOSE1:CLOSE15
```

### Line-by-Line Explanation

- 10 Open the command channel (and initialize the disk).
- 20 Check for an error. If one is found, exit.
- 30 Open the direct-access channel.
- 40 Check for an error. If one is found, exit.
- 50 Ask user for desired track and sector.
- 60 Read desired sector into the buffer.

- 70 Check for an error. If one is found, exit.
- 80 I will count the bytes.
- 90 Get a byte from the buffer.
- 100 Print the value of the byte.
- 110 Go on to the next byte.
- 120 Close files.

## Block Write (U2)

The data currently in the buffer of the 1541 can be written to any sector of the diskette using this command. To fill the buffer with data, the PRINT# command is used to the direct-access channel. The syntax of block write is

`PRINT# filenumber,"U2:"channel;drive;track;sector`

where *filenumber* is the logical file number of the command channel, *channel* is the channel number of the direct-access channel, and *drive* is the drive number. The *track* can be any track on the diskette (1 to 35) and the *sector* any sector of that track.

The command B-W is also a block write command, but it occasionally malfunctions, so use U2.

**Example:** This example should be run only on a formatted disk containing no information you want to preserve. It writes data to the specified sector, and may overwrite information which is already there.

```

10 OPEN15,8,15,"I"
20 IFDS<>OTHENPRINTDS$:GOTO160
30 OPEN1,8,2,"#"
40 IFDS<>OTHENPRINTDS$:GOTO160
50 PRINT"WHAT TO WRITE":INPUTW$
60 PRINT#1,W$
70 PRINT"WHICH TRACK, SECTOR":INPUTT,S
80 PRINT#15,"U2:"2;0;T;S
90 IFDS<>OTHENPRINTDS$:GOTO160
100 PRINT#15,"U1:"2;0;T;S
120 I=0
130 GET#1,A$
140 PRINTA$;
150 IFA$<>CHR$(13)THEN I=I+1:IFI<256THEN130
160 CLOSE1:CLOSE15

```

## Line-by-Line Explanation

- 10 Open the command channel and initialize the disk.
- 20 Check for error. If one is found, exit.
- 30 Open the direct access channel.
- 40 Check for error. If one is found, exit.
- 50 Ask for a string to write.
- 60 Put the input string followed by a carriage return in the buffer of the 1541.
- 70 Ask which track and sector to use.
- 80 Write the buffer to the disk.
- 90 Check for error. If one is found, exit.
- 100 Read the sector back into the buffer.
- 110 Check for error. If one is found, exit.
- 120 I will count the bytes.
- 130 Get 1 byte from the buffer.
- 140 Output the byte to the screen.
- 150 If the byte is a carriage return, the end of the string has been reached.  
Otherwise, go on to the next byte.
- 160 Close the files.

## Block Allocate (B-A)

The BAM is a record of which sectors on the disk are in use. When a sector is allocated, it is safe from being written on by the DOS in the course of normal writing (such as a SAVE). It is not safe from being written on by the direct-access commands, nor is it safe from being deallocated by a validate (COLLECT) operation. The block allocate command updates the BAM to show the designated sector as used. The BAM is actually written out to the disk when a direct-access channel is closed, so it is a good idea to open and close a direct-access channel when allocating a block, even if it is not needed for any other purpose. The syntax for a block allocate is

**PRINT# filenumber,"B-A:";drive;track;sector**

where *filenumber* is the logical file number of the command channel and *drive* is the drive number. The *track* can be any track on the diskette (1 to 35) and the *sector* any sector of that track.

**Example:** This example should be run only on a formatted disk containing no information you want to preserve. It allocates the desired block. Unless the block you specify was already allocated, a DIRECTORY performed prior to running the program shows one more free block than one performed after running the program.

```

10 OPEN15,8,15,"I"
20 IFDS<>OTHENPRINTDS$:GOTO80
30 OPEN1,8,2,"#"
40 IFDS<>OTHENPRINTDS$:GOTO80
50 PRINT"WHICH TRACK, SECTOR";:INPUTT,S
60 PRINT#15,"B-A:":O;T;S
70 IFDS<>OTHENPRINTDS$
80 CLOSE1:CLOSE15

```

### Line-by-Line Explanation

- 10 Open the command channel and initialize the disk.
- 20 Check for error. If one is found, exit.
- 30 Open the direct access channel.
- 40 Check for error. If one is found, exit.
- 50 Ask which track and sector to use.
- 60 Allocate the specified sector.
- 70 Check for error. If one is found, exit.
- 80 Close the files.

### Block Free (B-F)

The block free command updates the BAM to show the designated sector as not used. The BAM is actually written out to the disk when a direct-access channel is closed, so it is a good idea to open and close a direct-access channel when freeing a block, even if it is not needed for any other purpose. The syntax for a block free is

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

where *filenumber* is the logical file number of the command channel and *drive* is the drive number. The *track* can be any track on the diskette (1 to 35) and the *sector* any sector of that track.

**Example:** This example should be run only on a formatted disk containing no information you want to preserve. It frees the desired block. Unless the block you specify was

not allocated, a DIRECTORY performed prior to running the program shows one less free block than one performed after running the program.

```
10 OPEN15,8,15,"I"
20 IFDS<>OTHENPRINTDS$:GOTO80
30 OPEN1,8,2,"#"
40 IFDS<>OTHENPRINTDS$:GOTO80
50 PRINT"WHICH TRACK, SECTOR";:INPUTT,S
60 PRINT#15,"B-F:":O;T;S
70 IFDS<>OTHENPRINTDS$
80 CLOSE1:CLOSE15
```

#### Line-by-Line Explanation

- 10 Open the command channel and initialize the disk.
- 20 Check for error. If one is found, exit.
- 30 Open the direct access channel.
- 40 Check for error. If one is found, exit.
- 50 Ask which track and sector to use.
- 60 Free the specified sector.
- 70 Check for error. If one is found, exit.
- 80 Close the files.

#### Buffer Pointer (B-P)

The buffer pointer command designates the position in the 1541 buffer to access. This command can be used before reading from the buffer to start GETting bytes from a position other than the beginning of the buffer. B-P can be used before writing to the buffer to start writing at a position other than the beginning. The syntax for the buffer pointer command is

*PRINT# filenumber,"B-P:";channel;position*

where *filenumber* is the logical file number of the command channel and *channel* is the channel number of the direct-access channel. The *position* can be any byte in the buffer (0 to 255).

**Example:** This example reads the first sector of the directory, moves the buffer pointer to the first byte of the first file name in the directory, and prints out the first file name.

```

10 OPEN15,8,15,"I"
20 IFDS<>OTHENPRINTDS$:GOTO130
30 OPEN1,8,2,"#"
40 IFDS<>OTHENPRINTDS$:GOTO130
50 PRINT#15,"U1:"2;0;18;1
60 IFDS<>OTHENPRINTDS$:GOTO130
70 PRINT#15,"B-P:";2;5
80 IFDS<>OTHENPRINTDS$:GOTO130
90 FORI=OTO15
100 GET#1,A$
110 PRINTA$;
120 NEXT
130 CLOSE1:CLOSE15

```

### Line-by-Line Explanation

- 10 Open the command channel and initialize disk.
- 20 Check for error. If an error is present, exit.
- 30 Open a direct access channel.
- 40 Check for error. If an error is present, exit.
- 50 Read the first directory sector into the buffer.
- 60 Check for error. If an error is present, exit.
- 70 Place the buffer pointer at position 5 in the buffer, where the first file name begins.
- 80 Check for error. If an error is present, exit.
- 90 I counts the 16 bytes reserved for the first file name.
- 100 Get a byte of the file name.
- 110 Print out the byte on the screen.
- 120 Go on to the next byte.
- 130 Close the files.

### Block Execute (B-E)

This command reads the designated sector from the diskette into the buffer. It then transfers program control of the 1541 processor to byte 0 of the buffer. This execution continues until an RTS instruction is encountered. It is important to understand that a subroutine executed in this way is addressing only memory in the 1541; it is independent of the Plus/4. The syntax is

**PRINT# filenumber,"B-E:"channel;drive;track;sector**

where *filenumber* is the logical file number of the command channel, *channel* is the channel number of the direct-access channel, and *drive* is the drive number. The *track* can be any track on the diskette (1 to 35) and the *sector* any sector of that track.

**Example:** This example should be run only on a formatted disk containing no information you want to preserve. The program writes a short machine-language subroutine onto sector 0 of track 1, then block executes that sector. The subroutine examines the 1541's memory to determine which of its internal buffers (0 through 7) is being used, stores this information in the buffer for the BASIC program's retrieval, and returns.

```

10 OPEN15,8,15,"I"
20 IFDS<>0THENPRINTDSS:GOTO160
30 OPEN1,8,2,"#"
40 IFDS<>0THENPRINTDSS:GOTO160
50 PRINT#15,"B-P:";2;0
60 FORI=0TO6:READN
70 PRINT#1,CHR$(N);
80 NEXT
90 PRINT#15,"U2:"2;0;1;0
100 IFDS<>0THENPRINTDSS:GOTO160
110 PRINT#15,"B-E:";2;0;1;0
120 IFDS<>0THENPRINTDSS:GOTO160
130 PRINT#15,"B-P:";2;7
140 GET#1,A$
150 PRINT"USING BUFFER: ";ASC(A$)
160 CLOSE1:CLOSE15
170 DATA 160,7,165,249,145,48,96

```

### Line-by-Line Explanation

- 10 Open the command channel and initialize the disk.
- 20 Check for error. If an error is present, exit.
- 30 Open a direct access channel.
- 40 Check for error. If an error is present, exit.
- 50 Set buffer pointer to byte 0.
- 60 I counts the bytes in the machine-language routine.
- 70 Put a byte in the buffer.
- 80 Go on to the next byte.
- 90 Write buffer out to track 1, sector 0.

- 100 Check for error. If an error is present, exit.
- 110 Execute the machine code subroutine on track 1, sector 0.
- 120 Check for error. If an error is present, exit.
- 130 Position the buffer pointer to point at the data byte returned.
- 140 Retrieve the data byte (which buffer is in use).
- 150 Output the information.
- 160 Close the files.
- 170 The data are this machine code routine:

LDY #\$07	Set .Y to point at location in buffer to store data.
LDA \$F9	Get the buffer in use from disk drive's zero page.
STA (\$30),Y	Store in location 7 of the buffer.
RTS	Return from routine.

## Memory Read (M-R)

The memory resident in the 1541 (both RAM and ROM) can be read into a buffer with this command, 256 bytes at a time. The information can then be retrieved from the buffer with GET# from the command channel. The syntax is

**PRINT# filenumber,"M-R"CHR\$(ladd)CHR\$(hadd)CHR\$(nbyt)**

where *filenumber* is the logical file number of the command channel, *ladd* is the low byte of the address to begin reading, *hadd* is the high byte of the address to begin reading, and *nbyt* is the optional number of bytes to read (up to 255). If the final parameter indicating the number of bytes is not included, 1 byte is read.

**Example:**

```

10 OPEN15,8,15
20 IFDS<>OTHENPRINTDS$:GOTO80
30 PRINT#15,"M-R"CHR$(183)CHR$(229)CHR$(17)
40 FORI=1TO17
50 GET#15,A$
60 PRINTCHR$(ASC(A$)AND127);
70 NEXT
80 CLOSE15

```

### Line-by-Line Explanation

- 10 Open the command channel.
- 20 Check for error. If an error is present, exit.

- 30 Execute a memory read to put the contents of locations \$35B7-\$35C7 of the 1541's ROM into the buffer.
- 40 I counts the bytes read.
- 50 Get a byte from the buffer through the command channel.
- 60 Print out the character after stripping the high bit.
- 70 Go on to the next byte.
- 80 Close the command channel.

## Memory Write (M-W)

This command allows information to be written into the RAM of the 1541. Up to 34 bytes at a time can be sent to the command channel for transmission. The syntax is

`PRINT# filenumber,"M-W"CHR$(ladd)CHR$(hadd)CHR$(nbyt)data`

where *filenumber* is the logical file number of the command channel, *ladd* is the low byte of the address to begin writing, *hadd* is the high byte of the address to begin writing, and *nbyt* is the number of the data bytes included (up to 34).

**Example:** In this example, whatever the user types in is stored in the 1541's RAM with a memory write and retrieved with a memory read.

```

10 OPEN15,8,15
20 IFDS<>OTHENPRINTDS$:GOTO110
30 PRINT"WHAT TO WRITE";:INPUTW$
40 W$=LEFT$(W$,34):N=LEN(W$)
50 PRINT#15,"M-W"CHR$(0)CHR$(5)CHR$(N)W$
60 PRINT#15,"M-R"CHR$(0)CHR$(5)CHR$(N)
70 FORI=1TON
80 GET#15,A$
90 PRINTA$;
100 NEXT
110 CLOSE15

```

### Line-by-Line Explanation

- 10 Open the command channel.
- 20 Check for error. If an error is present, exit.
- 30 Ask user what to write.

- 40 Take only the leftmost 34 bytes. N is the length.  
 50 Write this information into the 1541's RAM beginning at \$0500.  
 60 Execute a memory read to put the contents of locations \$0500-\$0510 of the 1541's RAM into the buffer.  
 70 I counts the bytes read.  
 80 Get a byte from the buffer through the command channel.  
 90 Print out the character.  
 100 Go on to the next byte.  
 110 Close the command channel.

### Memory Execute (M-E)

This command transfers program control of the 1541 processor to the designated address in the 1541's memory. This execution continues until an RTS instruction is encountered. It is important to understand that a subroutine executed in this way is addressing only memory in the 1541; it is independent of the Plus/4. The syntax is

**PRINT# *filenumber*,"M-E:"CHR\$(*ladd*)CHR\$(*hadd*)**

where *filenumber* is the logical file number of the command channel, *ladd* is the low byte of the address to begin executing, and *hadd* is the high byte of the address to begin executing.

**Example:** This example writes a short machine-language subroutine into a buffer in the 1541. This subroutine counts the number of active buffers and stores it in the 1541's RAM for retrieval. The number of active buffers found can be changed by opening some direct-access channels just before running the program.

```

10 OPEN15,8,15
20 IFDS<>OTHENPRINTDSS:GOTO90
30 B$=""::FORI=0TO16:READN:B$=B$+CHR$(N):NEXT
40 PRINT#15,"M-W"CHR$(0)CHR$(5)CHR$(17)B$
50 PRINT#15,"M-E"CHR$(0)CHR$(5)
60 PRINT#15,"M-R"CHR$(27)CHR$(0)
70 GET#15,A$
80 PRINT"NUMBER OF BUFFERS = ";ASC(A$)
90 CLOSE15
100 DATA 160,0,162,6,181,167,201,255,240,1,200,202,16,246,132,
     27,96
  
```

## Line-by-Line Explanation

- 10 Open the command channel.
- 20 Check for error. If an error is present, exit.
- 30 Construct a string containing the machine-language subroutine.
- 40 Write the subroutine into the 1541's memory at \$0500.
- 50 Execute the machine-language subroutine.
- 60 Retrieve the saved count from the 1541's memory.
- 70 Get the count.
- 80 Print it out.
- 90 Close the command channel.
- 100 The data are this machine code routine:

\$0500 LDY #\$00	Count of active buffers found.
\$0502 LDX #\$06	Pointer to which buffer checking.
\$0504 LDA \$A7,X	Get channel number of this buffer.
\$0506 CMP #\$FF	Compare to inactive buffer value.
\$0508 BEQ \$050B	If equal, skip next instruction.
\$050A INY	Increment the count.
\$050B DEX	Decrement the pointer.
\$050C BPL \$0504	If not done, go back to get next the channel number.
\$050E STY \$1B	Store count for BASIC program's retrieval.
\$0510 RTS	Return.

## UnSCRATCHing a Disk File

When the disk drive is instructed to SCRATCH a file, it does not actually remove the file's contents from the diskette. Instead, it changes a byte in the diskette's directory to indicate that the file was SCRATCHed and deallocates all of the file's sectors in the BAM.

If you discover that you have SCRATCHed an important file for which you have no backup, you can sometimes recover the file. It is important to note that this procedure is for EMERGENCIES only. This is NOT a recommended procedure. It is far, far better to keep plenty of backup copies of all of your work.

If you must try it, please note the following:

1. Your chances of success are MUCH better if NOTHING has been written to the diskette since the SCRATCH occurred. Do not try this procedure if you performed disk write operations after the SCRATCH.

2. The diskette is validated as part of this procedure. Any sectors used for direct access will be deallocated as in any validate operation.
3. If possible, make copies of unaffected files you need from the diskette before running this program.

```

10 DIMA$(255)
20 FORI=1TO16:SS=SS+CHR$(160):NEXT
30 SCNCLR:PRINT
40 PRINT"INSERT DISK CONTAINING FILE TO UNSCRATCH":PRINT
50 PRINT"PRESS ANY KEY WHEN DONE":PRINT
60 GETKEYK$
70 PRINT"INITIALIZING DISK":PRINT
80 OPEN15,8,15,"I"
90 GOSUB460
100 INPUT"FILE TO UNSCRATCH";F$
110 F$=LEFT$(F$+SS,16)
120 PRINT:PRINT"LOOKING FOR ";F$
130 OPEN1,8,2,"#"
140 GOSUB460
150 T=18:S=1
160 PRINT#15,"U1:";2;0;T;S
170 GOSUB460
180 FORI=0TO255
190 GET#1,A$(I)
200 NEXT
210 F=0
220 B=32*F+4:FF$=""
230 FORI=1TO16
240 FF$=FF$+A$(B+I):NEXT
250 IFF$=FF$THEN300
260 F=F+1:IFF<8THEN220
270 T=ASC(A$(0)):IFT=0THEN290
280 S=ASC(A$(1)):GOTO160
290 PRINT:PRINTF$;" NOT FOUND":GOTO450
300 PRINT:PRINT"FOUND ";F$
310 B=B-2
320 IFASC(A$(B))<>0THENPRINT:PRINTF$;" IS NOT A SCRATCHED FILE":GOTO450
330 PRINT:PRINT"1 = SEQ, 2 = PRG, 3 = USR, 4 = REL"
340 INPUT"What type of file";N%
350 IFN%<1ORN%>4THEN330
360 PRINT: INPUT"OKAY TO UNSCRATCH (Y/N) ";K$
370 IFK$<>"Y"THEN450
380 PRINT#15,"B-P:";2;B
390 PRINT#1,CHR$(128+N%);
400 PRINT#15,"U2:";2;0;T;S
410 GOSUB460
420 PRINT:PRINT"VALIDATING DISK"
430 PRINT#15,"V"
440 GOSUB460
450 CLOSE1:CLOSE15:END
460 IFDS=0THENRETURN
470 PRINTD$:$CLOSE1:CLOSE15:END

```

## Line-by-Line Explanation

- 10 Dimension array to hold directory sector data.
- 20 Create a string of 16 shifted spaces.
- 30 Clear the screen and skip a line.
- 40-50 Print messages to user.
- 60 Wait for user to hit a key.
- 70 Print message to user.
- 80 Open the disk command channel, and initialize the disk.
- 90 Check for disk errors.
- 100 Get file name to unscratch.
- 110 Pad with shifted spaces.
- 120 Print message to user.
- 130 Open direct-access channel to disk drive.
- 140 Check for disk errors.
- 150 T is the current track; S is the current sector.
- 160 Read a directory sector into the disk buffer.
- 170 Check for disk errors.
- 180 Count the bytes from the disk buffer.
- 190 Get a byte.
- 200 Go on to the next byte.
- 210 F is the current file entry.
- 220 B is the base address for the current file name entry; start FF\$ as a null string.
- 230 I counts the bytes in the file name.
- 240 Concatenate each file name byte to FF\$.
- 250 If FF\$ is the file sought, go to line 300.
- 260 Add 1 to file count. If below 8 (not done with this sector), go on to the next file entry.
- 270 Get the next directory track. If zero, then there are no more sectors, so the file name was not found.
- 280 Get the next directory sector and return to line 160.

- 290      Inform user that the file was not found and quit.
- 300      Inform user that the file was found.
- 310      Now, B points to the file type byte for the current file entry.
- 320      If the file is not scratched, inform the user and quit.
- 330–340    Determine the desired file type.
- 350      If not a valid input, repeat the question.
- 360      Final chance to bail out without affecting the disk.
- 370      If it is not a definite yes, quit.
- 380      Move the disk buffer pointer to the file type byte for the current file entry.
- 390      Put the new file type byte into the disk buffer.
- 400      Write the disk buffer out to the disk.
- 410      Check for disk errors.
- 420      Print message to user.
- 430      Perform a validate function on the disk to update the BAM.
- 440      Check for disk errors.
- 450      Close the files and stop processing.
- 460      If no disk error, return.
- 470      Print out the error message, close the files, and quit.

## Using a Disk Drive with Machine Language

All of the direct-access programming techniques described in the preceding BASIC section are available in machine language. The function of the BASIC commands described to open and use the command and direct-access channels can be accomplished in machine code with the appropriate operating system subroutines.

In general terms, you can choose from three levels of communication with the disk drive. At the highest level are the save and load routines that perform all the necessary functions to save or load a section of RAM from disk. The middle level allows the maintenance of multiple open files, similar to OPEN commands in BASIC. At the most fundamental level, it is possible to listen and talk to the disk drive directly through the serial bus subroutines. Each of these levels is explored and example programs are given in this section. The complete descriptions of the operating system subroutines used can be found in Chapter 5.

**Saving, Loading, and Verifying Files** The key subroutines for saving and loading sections of RAM are SAVESP (\$FFD8) and LOADSP (\$FFD5). Using these routines is ideal for saving sections of memory into a disk file for later retrieval. The routines automatically perform the appropriate operations on the serial bus. The messages printed to the screen during these operations are controlled by the message flag set with SETMSG (\$FF90).

## Save

When a section of memory is saved to disk, a file name must be provided. The name is 1 to 16 characters in length and must be stored in CHR\$ codes in ascending order somewhere in memory. The following outlines the operations required:

1. Store the low and high bytes of the address at which the save is to begin into 2 consecutive bytes on zero page.
2. Load .X with the device number of the disk drive and call SETLFS (\$FFBA).
3. Load .A with the length of the file name, .X with the low byte of the address at which the name is stored, and .Y with the high byte of the address at which the name is stored, and call SETNAM (\$FFBD).
4. Load .A with the address of the zero page location in which the begin save address is stored (see step 1 above). Load .X with the low byte of the address at which to stop saving plus 1. Load .Y with the high byte of the address at which to stop saving plus 1. Call SAVESP (\$FFD8).
5. Check for errors by checking the carry status and the status variable and by reading the disk error channel. The status variable can be loaded into .A with READSS (\$FFB7).

## Load

When a file is to be loaded into memory, the file name by which it is known on the diskette must be stored in CHR\$ codes in ascending order somewhere in memory. The following outlines the operations required:

1. Load .X with the device number of the disk drive. Load .Y with \$00 for a relocated load, or a nonzero value for a nonrelocated load. Call SETLFS (\$FFBA).
2. Load .A with the length of the file name, .X with the low byte of the address at which the name is stored, and .Y with the high byte of the address at which the name is stored, and call SETNAM (\$FFBD).

3. Load .A with \$00 to indicate a load. If it is to be a relocated load, load .X with the low address to begin loading, and .Y with the high address to begin loading. Call LOADSP (\$FFD5).
4. Check for errors by checking the carry status and the status variable and by reading the disk error channel. The status variable can be loaded into .A with READSS (\$FFB7). A normal load sets bit 6 of the status variable to 1, meaning that the end-of-file was reached.

## Verify

When a file is to be verified against memory, the file name by which it is known on the diskette must be stored in CHR\$ codes in ascending order somewhere in memory. The following outlines the operations required:

1. Load .X with the device number of the disk drive. Load .Y with \$00 for a relocated verify, or a nonzero value for a nonrelocated verify. Call SETLFS (\$FFBA).
2. Load .A with the length of the file name, .X with the low byte of the address at which the name is stored, and .Y with the high byte of the address at which the name is stored, and call SETNAM (\$FFBD).
3. Load .A with \$01 (or other nonzero value) to indicate a verify. If it is to be a relocated verify, load .X with the low address to begin verifying and .Y with the high address to begin verifying. Call LOADSP (\$FFD5).
4. Check for errors by checking the carry status and the status variable and by reading the disk error channel. The status variable can be loaded into .A with READSS (\$FFB7). A normal verify sets bit 6 of the status variable to 1, meaning that the end-of-file was reached. A verify error sets bit 4 of the status variable to 1 as well.

**Example:** This example program saves itself on disk, verifies the resulting file against itself, and performs a relocated load of the file. No call to SETMSG is made. If the program is executed from the machine-language monitor, monitor messages are printed to the screen. The program is in two parts, the main one at \$2000 and a routine to read the disk drive's error channel at \$2080. Some CHR\$ data are required at memory locations \$20F8 and \$2100. It is shown after the program.

. 2000	A9 00	LDA #\$00	Low byte of the begin save address.
. 2002	85 D8	STA \$D8	Store on zero page
. 2004	A9 20	LDA #\$20	High byte of the begin save address.
. 2006	85 D9	STA \$D9	Store in next byte on zero page.
. 2008	A2 08	LDX #\$08	Device number of disk drive.
. 200A	20 BA FF	JSR \$FFBA	SETLFS.
. 200D	A9 06	LDA #\$06	Length of file name.

. 200F	A2 00	LDX #\$00	Low byte of file name address.
. 2011	A0 21	LDY #\$21	High byte of file name address.
. 2013	20 BD FF	JSR \$FFBD	SETNAM.
. 2016	A9 D8	LDA #\$D8	Address of zero page location at \$2003.
. 2018	A2 06	LDX #\$06	Low byte of end save address.
. 201A	A0 21	LDY #\$21	High byte of end save address.
. 201C	20 D8 FF	JSR \$FFD8	SAVESP.
. 201F	B0 5F	BCS \$2080	If error, read disk error channel.
. 2021	20 B7 FF	JSR \$FFB7	Get status byte.
. 2024	D0 5A	BNE \$2080	If error, read disk error channel.
. 2026	A9 80	LDA #\$80	To signal verify.
. 2028	20 D5 FF	JSR \$FFD5	LOADSP. (All the other information was already set).
. 202B	B0 53	BCS \$2080	If error, read disk error channel.
. 202D	20 B7 FF	JSR \$FFB7	Get status byte.
. 2030	C9 40	CMP #\$40	Normal end-of-file.
. 2032	D0 3C	BNE \$2070	If not, then go to error handler.
. 2034	A2 08	LDX #\$08	Device number of disk drive.
. 2036	A0 00	LDY #\$00	To indicate a relocated load.
. 2038	20 BA FF	JSR \$FFBA	SETLFS.
. 203B	A9 00	LDA #\$00	To signal load.
. 203D	A2 00	LDX #\$00	Low byte of address for load start.
. 203F	A0 30	LDY #\$30	High byte of address for load start.
. 2041	20 D5 FF	JSR \$FFD5	LOADSP.
. 2044	B0 3A	BCS \$2080	If error, read disk error channel.
. 2046	20 B7 FF	JSR \$FFB7	Get status byte.
. 2049	C9 40	CMP #\$40	Normal end-of-file.
. 204B	D0 33	BNE \$2080	If not, read disk error channel.
. 204D	00	BRK	Stop processing.

The contents of the X and Y register are the low and high bytes of the address of the last byte loaded plus 1, if a normal exit is achieved.

. 2070	C9 50	CMP #\$50	Look for verify error.
. 2072	D0 0C	BNE \$2080	If not, read disk error channel.
. 2074	A2 05	LDX #\$05	Length of error message.
. 2076	BD F8 20	LDA \$20F8,X	Get a byte of message.
. 2079	20 D2 FF	JSR \$FFD2	Write it to the screen.
. 207C	CA	DEX	Point at next byte.
. 207D	10 F7	BPL \$2076	If more, go back.
. 207F	00	BRK	Stop processing.
. 2080	A9 0F	LDA #\$0F	Logical file number to use.
. 2082	A2 08	LDX #\$08	Device number of disk drive.
. 2084	A0 0F	LDY #\$0F	Command channel.
. 2086	20 BA FF	JSR \$FFBA	SETLFS.
. 2089	A9 00	LDA #\$00	No file name.
. 208B	20 BD FF	JSR \$FFBD	SETNAM.
. 208E	20 C0 FF	JSR \$FFC0	OPEN.
. 2091	B0 16	BCS \$20A9	If an error is present, quit.
. 2093	A2 0F	LDX #\$0F	Logical file number of command channel.
. 2095	20 C6 FF	JSR \$FFC6	CHKIN.
. 2098	B0 0F	BCS \$20A9	If an error is present, quit.

. 209A A9 0D	LDA #\$0D	Carriage return character.
. 209C 20 D2 FF	JSR \$FFD2	Send to screen.
. 209F 20 CF FF	JSR \$FFCF	Get a character from command channel.
. 20A2 20 D2 FF	JSR \$FFD2	Send to screen.
. 20A5 C9 0D	CMP #\$0D	Compare to carriage return.
. 20A7 D0 F6	BNE \$209F	If not, get another byte.
. 20A9 A9 0F	LDA #\$0F	Logical file number of command channel.
. 20AB 20 C3 FF	JSR \$FFC3	CLOSE.
. 20AE 20 CC FF	JSR \$FFCC	CLRCH.
. 20B1 00	BRK	Stop processing.

The following data need to be placed in memory:

```
>20F8 52 4F 52 52 45 20 00 00 : RORRE ..
>2100 53 41 56 50 52 47 00 00 : SAVPRG..
```

**Using Data Files** This section outlines the machine-language equivalents for the BASIC statements OPEN, CLOSE, GET#, and PRINT#. They can be used for all the purposes outlined in the preceding BASIC sections, including data-file handling and direct-access programming. The key subroutines are OPEN (\$FFC0), CLOSE (\$FFC3), CHIN (\$FFC6), CHOUT (\$FFC9), CLRCH (\$FFCC), BASIN (\$FFCF), and BSOUT (\$FFD2). Any messages printed to the screen during these operations are controlled by the message flag set with SETMSG (\$FF90).

## Open

When a file is to be opened, and a file name must be provided, the name (1 to 16 characters) and the type of file and mode (see the BASIC section on OPENing data files) must be stored in CHR\$ codes in ascending order somewhere in memory. Exactly the same situation exists as in BASIC. That is, if the type and mode are omitted, the mode is assumed to be read, and the type is the existing type of the file. A direct-access channel can be opened by using a pound sign for the file name. The following outlines the operations required:

1. Load .A with the logical file number to use for the file, .X with the device number of the disk drive, and .Y with the channel number. Call SETLFS (\$FFBA).
2. Load .A with the length of the file name (0 if no name), .X with the low byte of the address where the file name is stored, and .Y with the high byte of the address where the file name is stored. Call SETNAM (\$FFBD).
3. Call OPEN (\$FFC0).
4. Check for errors by examining the carry bit and the disk error channel. The status byte is not particularly useful for discovering OPEN errors.

## Close

Files can be closed individually with the CLOSE (\$FFC3) routine:

1. Load .A with the logical file number of the file to be closed. Call CLOSE (\$FFC3).
2. Check for errors by examining the carry bit, status byte, and disk error channel. Generally, checking the carry bit and disk error channel is sufficient.

All the files in use can be closed and the input and output channels reset to their default devices by calling CLALL (\$FFE7).

## Print

To send information into a file, it must have been opened with a mode of write. The following outlines the operations required:

1. Load .X with the logical file number of the output file. Designate it as the output channel by calling CHOUT (\$FFC9).
2. Check for errors.
3. Send the data to the channel with BSOUT (\$FFD2).
4. When all data has been sent, return the input and output channels to default (keyboard and screen) by calling CLRCH (\$FFCC) and close the file. Or call CLALL (\$FFE7) to close all files and reset the input and output channels.

## Get

To receive information from a file, it must have been opened with a mode of read (no mode defaults to read). The following outlines the operations required:

1. Load .X with the logical file number of the input file. Designate it as the input channel by calling CHKIN (\$FFC6).
2. Check for errors.
3. Receive the data from the channel with BASIN (\$FFCF).
4. When all data has been received, return the input and output channels to default (keyboard and screen) by calling CLRCH (\$FFCC) and close the file. Or call CLALL (\$FFE7) to close all files and reset the input and output channels.

**Example:** This example opens a new file called DATFIL and writes data accepted from the keyboard to it. It then closes the file, reopens it for reading, and retrieves the data,

displaying it on the screen. Because no call is made to SETMSG, machine-language monitor error messages are displayed if the program is executed from the monitor.

- 2000 A9 02 LDA #\$02 Logical file number for data file.
- 2002 A2 08 LDX #\$08 Device number of disk drive.
- 2004 A0 02 LDY #\$02 Channel number to use.
- 2006 20 BA FF JSR \$FFBA SETLFS.
- 2009 A9 0A LDA #\$0A Length of file name.
- 200B A2 00 LDX #\$00 Low byte of address of file name.
- 200D A0 21 LDY #\$21 High byte of address of file name.
- 200F 20 BD FF JSR \$FFBD SETNAM.
- 2012 20 C0 FF JSR \$FFC0 OPEN.
- 2015 B0 5F BCS \$2076 Check for I/O errors.
- 2017 20 A0 20 JSR \$20A0 Check the disk error channel.
- 201A A9 0D LDA #\$0D Carriage return character.
- 201C 20 D2 FF JSR \$FFD2 Send to the screen.
- 201F A2 02 LDX #\$02 Logical file number for data file.
- 2021 20 C9 FF JSR \$FFC9 Set up as the output channel.
- 2024 B0 50 BCS \$2076 Check for I/O errors.
- 2026 20 CF FF JSR \$FFCF Receive input from keyboard.
- 2029 20 D2 FF JSR \$FFD2 Send to file.
- 202C C9 0D CMP #\$0D Check if received carriage return.
- 202E D0 F6 BNE \$2026 If not, continue.
- 2030 20 CC FF JSR \$FFCC Return I/O channels to defaults.
- 2033 A9 02 LDA #\$02 Logical file number for data file.
- 2035 20 C3 FF JSR \$FFC3 Close the file.
- 2038 B0 3C BCS \$2076 Check for I/O errors.
- 203A 20 A0 20 JSR \$20A0 Check disk error channel.
- 203D A9 02 LDA #\$02 Logical file number to be used for data file.
- 203F A2 08 LDX #\$08 Device number of disk drive.
- 2041 A0 02 LDY #\$02 Channel number to use.
- 2043 20 BA FF JSR \$FFBA SETLFS.
- 2046 A9 06 LDA #\$06 Length of file name (no longer use ,S,W part).
- 2048 A2 00 LDX #\$00 Low byte of address of file name.
- 204A A0 21 LDY #\$21 High byte of address of file name.
- 204C 20 BD FF JSR \$FFBD SETNAM.
- 204F 20 C0 FF JSR \$FFC0 OPEN.
- 2052 B0 22 BCS \$2076 Check for I/O errors.
- 2054 20 A0 20 JSR \$20A0 Check disk error channel.
- 2057 A2 02 LDX #\$02 Logical file number of data file.
- 2059 20 C6 FF JSR \$FFC6 Set up as the input channel.
- 205C B0 18 BCS \$2076 Check for I/O errors.
- 205E 20 CF FF JSR \$FFCF Get a character from the file.
- 2061 20 D2 FF JSR \$FFD2 Put it on the screen.
- 2064 20 B7 FF JSR \$FFB7 Read the status byte.
- 2067 F0 F5 BEQ \$205E If zero, not at end-of-file so continue.
- 2069 20 CC FF JSR \$FFCC Return I/O channels to defaults.
- 206C A9 02 LDA #\$02 Logical file number of data file.
- 206E 20 C3 FF JSR \$FFC3 Close the file.

- . 2071 B0 03 BCS \$2076 Check for I/O errors.
- . 2073 20 A0 20 JSR \$20A0 Check the disk error channel.
- . 2076 20 E7 FF JSR \$FFE7 Make sure all files are closed.
- . 2079 00 BRK Stop processing.

The following subroutine reads the disk error channel. If two zero characters are received, then there is no error, and it exits. If not, the message is displayed, the files are closed, and the stack pointer is restored to its proper value. Processing is then stopped.

- . 20A0 A9 01 LDA #\$01 Logical file number to use for command channel.
- . 20A2 A2 08 LDX #\$08 Device number of disk drive.
- . 20A4 A0 0F LDY #\$0F Command channel.
- . 20A6 20 BA FF JSR SFFBA SETLFS.
- . 20A9 A9 00 LDA #\$00 No file name required.
- . 20AB 20 BD FF JSR \$FFBD SETNAM.
- . 20AE 20 C0 FF JSR \$FFC0 OPEN.
- . 20B1 B0 35 BCS \$20E8 Check for I/O errors.
- . 20B3 A2 01 LDX #\$01 Logical file number of channel.
- . 20B5 20 C6 FF JSR \$FFC6 Set up as input channel.
- . 20B8 B0 2E BCS \$20E8 Check for I/O errors.
- . 20BA 20 CF FF JSR \$FFCF Get a character.
- . 20BD C9 30 CMP #\$30 Compare to a zero character.
- . 20BF D0 12 BNE \$20D3 If not, there is an error.
- . 20C1 20 CF FF JSR \$FFCF Get a character.
- . 20C4 C9 30 CMP #\$30 Compare to a zero character.
- . 20C6 D0 0E BNE \$20D6 If not, there is an error.
- . 20C8 A9 01 LDA #\$01 Logical file number of channel.
- . 20CA 20 C3 FF JSR \$FFC3 Close the file.
- . 20CD B0 19 BCS \$20E8 Check for I/O errors.
- . 20CF 20 CC FF JSR \$FFCC Return I/O channels to defaults.
- . 20D2 60 RTS Return from subroutine with no error.
- . 20D3 20 CF FF JSR \$FFCF Get a character.
- . 20D6 20 CF FF JSR \$FFCF Get a character.
- . 20D9 A9 0D LDA #\$0D Carriage return character.
- . 20DB 20 D2 FF JSR \$FFD2 Send to screen.
- . 20DE 20 CF FF JSR \$FFCF Get a character from the error channel.
- . 20E1 20 D2 FF JSR \$FFD2 Send to screen.
- . 20E4 C9 0D CMP #\$0D Compare to a carriage return.
- . 20E6 D0 F6 BNE \$20DE If not, continue.
- . 20E8 20 E7 FF JSR \$FFE7 Close down all files and restore I/O channels.
- . 20EB 68 PLA Pull the return address off the stack.
- . 20EC 68 PLA
- . 20ED 00 BRK Stop processing.

The file name must be in memory at \$2100.

```
>2100 44 41 54 46 49 4C 2C 53 : DATFIL,S
>2108 2C 57 00 00 00 00 00 00 : ,W.....
```

**Programming the Serial Bus**   Occasionally it is desirable to program directly to the serial bus. The family of operating subroutines used for this purpose consists of LISTN (\$FFB1), SECND (\$FF93), CIOUT (\$FFA8), UNLSN (\$FFAE), TALK (\$FFB4), TKSA (\$FF96), ACPTR (\$FFA5), and UNTLK (\$FFAB). Programming at this level eliminates the file structure used with OPEN and CLOSE. Instead of thinking in terms of logical file numbers, think in terms of the channels of the disk drive. A channel may be opened and told to send data to the computer (to talk) or to receive data from the computer (to listen). The error checking must be done using the status variable for device not present, end-of-file, and such, and by reading the error channel. When these operations are done depends to a large extent on what assumptions the particular application allows. When in doubt, check for an error.

## **Opening a File**

Here is an outline of a procedure:

1. Set the status variable (\$90) to zero.
2. Load .A with the device number of the disk drive (and store in the current device number at \$AE if this has not already been done). Call LISTN (\$FFB1). Check for a device not present error in the status variable.
3. Load .A with the disk drive channel number you want to use ORed with \$F0 (to indicate open up this channel). Call SECND (\$FF93).
4. Send the file name (if any) through .A to the drive using CIOUT (\$FFA8).
5. Call UNLSN (\$FFAE).

The channel specified is now opened to the file specified.

## **Getting Data from an Open Channel**

1. Load .A with the device number of the disk drive and call TALK (\$FFB4).
2. Load .A with the channel number ORed with \$60 (to tell the drive this is a secondary address). Call TKSA (\$FF96).
3. Call ACPTR (\$FFA5) to receive bytes in .A. When bit 6 of the status variable is set, the end-of-file has been reached.
4. Call UNTLK (\$FFAB).

## **Sending Data to an Open Channel**

1. Load .A with the device number of the disk drive and call LISTN (\$FFB1).
2. Load .A with the channel number ORed with \$60 (to tell the drive this is a secondary address). Call SECND (\$FF93).

3. Call CIOUT (\$FFA8) to send bytes from .A.
4. Call UNLSN (\$FFAE).

## Closing a File

1. Load .A with the device number of the disk drive. Call LISTN (\$FFB1).
2. Load .A with the disk drive channel number you want to close ORed with \$E0 (to indicate close this channel). Call SECND (\$FF93).
3. Call UNLSN (\$FFAE).

**Example:** This example reads the first sector of the directory of a diskette and places it in memory. Errors are detected and acted on, but no user interface is in place to deliver error messages to the user.

• 2000 A9 00	LDA #\$00	
• 2002 85 90	STA \$90	Clear the status variable.
• 2004 A9 08	LDA #\$08	Disk drive device number.
• 2006 85 AE	STA \$AE	Store in the current device number.
• 2008 20 B1 FF	JSR \$FFB1	Tell drive to listen.
• 200B 20 B7 FF	JSR \$FFB7	Read status variable.
• 200E D0 7B	BNE \$208B	If not zero, then error.
• 2010 A9 F2	LDA #\$F2	Open command for channel 2.
• 2012 20 93 FF	JSR \$FF93	Send as secondary address to drive.
• 2015 A9 23	LDA #\$23	This is the code for a pound sign (#).
• 2017 20 A8 FF	JSR \$FFA8	Open a direct-access channel.
• 201A 20 AE FF	JSR \$FFAE	Tell drive to stop listening.
• 201D A9 08	LDA #\$08	Disk drive device number.
• 201F 20 B1 FF	JSR \$FFB1	Tell drive to listen.
• 2022 A9 FF	LDA #\$FF	Open command for channel 15.
• 2024 20 93 FF	JSR \$FF93	Send as secondary address to drive.
• 2027 A2 00	LDX #\$00	.X counts bytes to send.
• 2029 BD 00 21	LDA \$2100,X	Get a byte of the Ul command.
• 202C 20 A8 FF	JSR \$FFA8	Send to the drive.
• 202F E8	INX	Increment count.
• 2030 E0 0B	CPX #\$0B	Compare to length of Ul command.
• 2032 90 F5	BCC \$2029	Not done, continue sending.
• 2034 20 AE FF	JSR \$FFAE	Tell drive to stop listening.
• 2037 A9 08	LDA #\$08	Disk drive device number.
• 2039 20 B4 FF	JSR \$FFB4	Tell drive to talk.
• 203C A9 6F	LDA #\$6F	Access command channel (for errors).
• 203E 20 96 FF	JSR \$FF96	Send as secondary address to drive.
• 2041 A2 00	LDX #\$00	.X counts the bytes received.
• 2043 20 A5 FF	JSR \$FFA5	Receive a byte from the drive.
• 2046 9D 00 23	STA \$2300,X	Store in error buffer.
• 2049 E8	INX	Increment byte count.
• 204A 20 B7 FF	JSR \$FFB7	Read status variable.
• 204D F0 F4	BEQ \$2043	If zero, more to receive so go back.
• 204F 20 AB FF	JSR \$FFAB	Tell drive to stop talking.
• 2052 AD 00 23	LDA \$2300	Get first byte of error buffer.
• 2055 0D 01 23	ORA \$2301	OR with the second byte.

- . 2058 C9 30      CMP #\$30      If no error, the result will be \$30.
- . 205A D0 18      BNE \$2074      If not, error has been detected so quit.
- . 205C A9 08      LDA #\$08      Disk drive device number.
- . 205E 20 B4 FF    JSR \$FFB4      Tell drive to talk.
- . 2061 A9 62      LDA #\$62      Access data channel (to get results of U1).
- . 2063 20 96 FF    JSR \$FF96      Send as secondary address to drive.
- . 2066 A2 00      LDX #\$00      .X counts the bytes received.
- . 2068 20 A5 FF    JSR \$FFA5      Receive a byte from the drive.
- . 206B 9D 00 22    STA \$2200,X      Store in directory buffer.
- . 206E E8          INX      Increment byte count.
- . 206F D0 F7      BNE \$2068      Get one whole sector.
- . 2071 20 AB FF    JSR \$FFAB      Tell drive to stop talking.
- . 2074 A9 08      LDA #\$08      Disk drive device number.
- . 2076 20 B1 FF    JSR \$FFB1      Tell drive to listen.
- . 2079 A9 E2      LDA #\$E2      Close command for channel 2.
- . 207B 20 93 FF    JSR \$FF93      Send as secondary address to drive.
- . 207E 20 AE FF    JSR \$FFAE      Tell drive to stop listening.
- . 2081 A9 08      LDA #\$08      Disk drive device number.
- . 2083 20 B1 FF    JSR \$FFB1      Tell drive to listen.
- . 2086 A9 EF      LDA #\$EF      Close command for channel 15.
- . 2088 20 93 FF    JSR \$FF93      Send as secondary address to drive.
- . 208B 20 AE FF    JSR \$FFAE      Tell drive to stop listening.
- . 208E 00          BRK      Stop processing.

The U1 command is stored as data.

```
>2100 55 31 3A 32 2C 30 2C 31 : U1:2,0,1
>2108 38 2C 31 00 00 00 00 00 : 8,1.....
```

## The Datasette Tape Recorder

The Datasette is a specialized tape recorder for use with Commodore computers only. The Datasette designed for use with the Plus/4 is called the 1531. A Datasette can be used to store and retrieve programs and data. Working with tape is slow, and the capacity of even the longest recommended tape (30 minutes) is limited. However, cassette tapes are not as easily destroyed as diskettes (in the mail or in the hands of young children). The Datasette is also much less expensive than a disk drive, and cassette tapes are cheaper than diskettes. For the lower-budget computer user, or a beginner just starting to program, a Datasette may be a good investment.

Files on tape can, but need not, be identified with names. It is generally a good idea to give every file on a tape a name unless only one file is to be stored on the tape. When you access a tape file, the computer begins searching the tape from wherever it is sitting. In other words, the computer does not rewind the tape, so you must be careful to position the tape before you ask the computer to access a file.

Usually, it is best to simply give every file a name and always access a file using its name. Then, always rewind the tape completely before you ask for a file. It may sometimes be convenient to use the tape counter on the Datasette to position the tape. Remember to reset the counter to 000 after you rewind the tape. Remember, too, that the counter is only approximate. Fast forward to a position before the exact position where the file is expected to start and then access the file by name.

Whenever the Datasette is being accessed, the screen is blanked. This is done to speed up tape access by eliminating the time taken by the computer to manage the screen display.

## Using a Datasette with BASIC

The Datasette can be used to save BASIC programs or to create and retrieve data files through BASIC. Errors, end-of-files, and end-of-tape marks can be detected by examining the status variable, ST. The status variable does NOT detect hardware problems such as failure to press the record button when you attempt to create a file, reaching the physical end of a tape prior to completing the file, and so on. For this reason, it is a good idea to check that programs are saved properly by using the VERIFY command and to include verifying code in programs that create data files.

### ST for the Datasette

<i>Bit</i>	<i>Meaning</i>
0	Unused
1	Unused
2	Short block
3	Long block
4	Read or verify error
5	Checksum error
6	End-of-file mark encountered
7	End-of-tape mark encountered

An additional location of interest when working with the Datasette is 64784 (\$FD10). Bit 2 of this location can be used to detect whether or not a button is pushed down on the Datasette. It does not distinguish between the buttons. The statement

**WAIT 64784,4**

waits until no button is down. The statement

**WAIT 64784,4,4**

waits until a button is down.

**Saving, Loading, and Verifying Programs** Because one of the main uses for a Datasette is to save and retrieve BASIC programs, BASIC has built-in commands to make it easy. Saving programs can be done with the **SAVE** command:

**SAVE *filename,device,EOT/file type flag***

The *filename* is the optional name by which the file is known on the tape. It can be up to 17 characters in length. The *device* is the optional device number for the Datasette (1). If the file name is omitted, the program is saved with no name. The *EOT/file type flag* can be omitted (and default to 0) or have values 0–3. If it is 1, the file cannot be relocated on subsequent LOADs. If it is 2, an end-of-tape (rather than an end-of-file) marker is written on the tape after the file. A value of 3 enables both features, and 0 disables both.

To retrieve the program, the **LOAD** command can be used:

**LOAD *filename,device,absolute/relative flag***

The *filename* is the same one used to save the program, and the *device* is the optional device number for the Datasette (1). The *absolute/relative flag* is also optional. If it is omitted or 0, the file is loaded as a BASIC program starting at the beginning of BASIC RAM. If it is 1, the file will be loaded at the location from which it was saved. This flag is overridden if the file was SAVED as a nonrelocatable file as described previously. If the file name is included, the computer begins reading the tape and continues until it finds a file with the specified name (or an end-of-tape mark, which results in a **BREAK** error). If the file name is omitted, the first file found is loaded. Every time the computer finds a file, it prints to the screen:

**FOUND *filename***

and pauses. To hurry it along, you can press the **C** key. To interrupt a tape operation at any time, you can press the **STOP** key.

To compare a program in memory with one stored on tape without altering memory, the **VERIFY** command can be used. It reports if the file is not identical to the program in memory.

**VERIFY *filename,device,absolute/relative flag***

All of the parameters are the same as for **LOAD**.

The VERIFY command is also useful when you want to position the tape beyond the last file recorded on it. After you rewind the tape, request a VERIFY of the last file on the tape. Of course, a VERIFY ERROR is reported (unless the last file is identical to the program in memory). But the tape is positioned just after the file, ready for a new file to be recorded.

**Using Data Files** A data file on tape consists of a long series of bytes stored on the tape. It can be identified with a name and is concluded with an end-of-file (or end-of-tape) mark. The purpose for which the file is being accessed must be given in the secondary address sent with the OPEN statement:

**OPEN filenumber,1,secondary,filename**

The *filenumber* is the logical file number to be associated with this file. It is for reference purposes only and can be 0-127 (128-255 will send a linefeed character following every carriage return). The *secondary* address specifies the use of the file. If it is 0 (or omitted), the file is read. If it is 1, the file is written and ended with an end-of-file. If it is 2, the file is written and ended with an end-of-tape. The last parameter is the file name (a string expression up to 17 characters).

After the file is open, output may be directed to it by referring to its logical file number in a PRINT# or CMD command. Input may be received from it by referring to its logical file number in a INPUT# or GET# command.

When the input to or output from a file is complete, the file must be closed with

**CLOSE filenumber**

where *filenumber* is the logical file number of the file to close.

After a file is opened for writing, each byte that is sent to it is stored sequentially on the tape. The format of the bytes written to a tape file must be designed with the method of retrieval in mind. If they are to be read one at a time using the GET# command, any format is all right. However, if they are to be read using the INPUT# command, care must be taken to store comma characters between values and carriage return characters between lines of input.

Example:

```
10 GOSUB130
20 OPEN1,1,1,"DATFIL"
30 PRINTCHR$(13)"WHAT TO STORE";:INPUTA$
40 PRINT#1,A$
50 CLOSE1
60 GOSUB130
70 PRINT"REWIND THE TAPE":WAIT64784,4,4
80 PRINT"WHEN DONE, ";:GOSUB130
90 OPEN1,1,0,"DATFIL"
```

```

100 INPUT#1,B$
110 IFA$<>B$THENPRINTCHR$(13)"VERIFY ERROR"
120 CLOSE1:PRINTCHR$(13):GOSUB130:END
130 IF(PEEK(64784)AND4)=0THENPRINT" HIT STOP ON
    DATASSETTE":WAIT64784,4
140 RETURN

```

### Line-by-Line Explanation

- 10 Make sure no buttons are pressed on Datasette.
- 20 Open a data file for writing.
- 30 Get some input to record.
- 40 Output the string to the Datasette.
- 50 Close the file.
- 60 Make sure no buttons are pressed on Datasette.
- 70 Tell the user to rewind the tape, and wait for the button to be down.
- 80 Make sure no buttons are pressed on Datasette.
- 90 Open the file for reading.
- 100 Get the string from the Datasette.
- 110 Compare it with the string typed in by the user, and report error.
- 120 Close the file and make sure no buttons are pressed.
- 130 If a button is down, print a message, and wait until none are down.
- 140 Return to main program.

### Using a Datasette from Machine Language

In general terms, you can choose one of two levels of communication when you use the Datasette. At the higher level are the save and load routines that perform all the necessary functions to save or load a section of RAM from disk. The lower level allows the maintenance of an open file on the Datasette in addition to other open files, similar to using OPEN commands in BASIC. These levels are explored and example programs are given in this section. The complete descriptions of the operating system subroutines used can be found in Chapter 5.

***Saving, Loading, and Verifying Files*** The key subroutines for saving and loading sections of RAM are SAVESP (\$FFD8) and LOADSP (\$FFD5). These

routines are ideal for saving sections of memory into a tape file for later retrieval. They automatically perform the appropriate operations on the Datassette. The messages printed to the screen during these operations are controlled by the message flag set with SETMSG (\$FF90).

## Save

When a section of memory is saved to tape, a file name may be provided. The name is 1 to 17 characters long and must be stored in CHR\$ codes in ascending order somewhere in memory. The following outlines the operations required:

1. Store the low and high bytes of the address at which the save is to begin into two consecutive bytes on zero page.
2. Load .X with \$01 (the device number of the Datassette). Load .Y with the desired secondary address (\$00-\$03). If it is \$01, the file cannot be relocated on subsequent LOADs. If it is \$02, an end-of-tape (rather than an end-of-file) marker is written on the tape after the file. If it is \$03, both of the features are enabled. If it is \$00, neither of them are in force. Call SETLFS (\$FFBA).
3. Load .A with the length of the file name (\$00 if none), .X with the low byte of the address at which the name is stored, and .Y with the high byte of the address at which the name is stored, and call SETNAM (\$FFBD).
4. Load .A with the address of the zero page location in which the begin save address is stored (see step 1 above). Load .X with the low byte of the address at which to stop saving plus 1. Load .Y with the high byte of the address at which to stop saving plus 1. Call SAVESP (\$FFD8).
5. Check for errors by checking the carry status and the status variable. The status variable can be loaded into .A with READSS (\$FFB7).

**Example:** This program saves itself on tape. It is in three parts: the save routine, the error message routine, and data for the error message and file name.

. 2000 A9 00	LDA #\$00	Low byte of address to save from.
. 2002 85 D8	STA \$D8	Store on zero page.
. 2004 A9 20	LDA #\$20	High byte of address to save from.
. 2006 85 D9	STA \$D9	Store on zero page.
. 2008 A2 01	LDX #\$01	Device number for Datassette.
. 200A A0 00	LDY #\$00	Secondary address for standard file.
. 200C 20 BA FF	JSR \$FFBA	SETLFS.
. 200F A9 06	LDA #\$06	Length of file name.
. 2011 A2 00	LDX #\$00	Low byte of address of the file name.
. 2013 A0 21	LDY #\$21	High byte of address of the file name.
. 2015 20 BD FF	JSR \$FFBD	SETNAM.

. 2018 A9 D8	LDA #\$D8	Location of save from address.
. 201A A2 06	LDX #\$06	Low byte of the address to save to plus 1.
. 201C A0 21	LDY #\$21	High byte of the address to save to plus 1.
. 201E 20 D8 FF	JSR \$FFD8	SAVESP.
. 2021 B0 5D	BCS \$2080	Carry set indicates error.
. 2023 20 B7 FF	JSR \$FFB7	Get the status byte.
. 2026 D0 58	BNE \$2080	If there is an error, put out error message.
. 2028 00	BRK	Stop processing.

The following routine prints the word ERROR to the screen:

. 2080 A2 06	LDX #\$06	Length of the word.
. 2082 BD F8 20	LDA \$20F8,X	Get a character of the message.
. 2085 20 D2 FF	JSR \$FFD2	Send to the screen.
. 2088 CA	DEX	Decrement the character counter.
. 2089 10 F7	BPL \$2082	If not done, continue.
. 208B 00	BRK	Stop processing.

These data are for the message and file name:

```
>20F8 52 4F 52 52 45 0D 20 20 : RORRE.  
>2100 53 41 56 50 52 47 00 00 : SAVPRG..
```

## Load

When a file is to be loaded into memory, the file name by which it is known on the tape may be stored in CHR\$ codes in ascending order somewhere in memory. If no file name is specified, the next file found on the tape will be loaded. The following outlines the operations required:

1. Load .X with \$01 (the device number of the Datasette). Load .Y with \$00 for a relocated load, or a nonzero value for a nonrelocated load. This flag is overridden if the file was SAVED as a nonrelocatable file, as described previously. Call SETLFS (\$FFBA).
2. Load .A with the length of the file name (\$00 if none), .X with the low byte of the address at which the name is stored, and .Y with the high byte of the address at which the name is stored, and call SETNAM (\$FFBD).
3. Load .A with \$00 to indicate a load. If it is to be a relocated load, load .X with the low address to begin loading and .Y with the high address to begin loading. Call LOADSP (\$FFD5).

4. Check for errors by checking the carry status and the status variable. The status variable can be loaded into .A with READSS (\$FFB7).

**Example:** This example does a relocated load of the file created in the previous example.

. 2000 A2 01	LDX #\$01	Device number of the Datasette.
. 2002 A0 00	LDY #\$00	Signal a relocated load.
. 2004 20 BA FF	JSR \$FFBA	SETLFS.
. 2007 A9 06	LDA #\$06	Length of the file name.
. 2009 A2 00	LDX #\$00	Low byte of address of the file name.
. 200B A0 21	LDY #\$21	High byte of address of the file name.
. 200D 20 BD FF	JSR \$FFBD	SETNAM.
. 2010 A9 00	LDA #\$00	Signal load.
. 2012 A2 00	LDX #\$00	Low byte of the load address.
. 2014 A0 30	LDY #\$30	High byte of the load address.
. 2016 20 D5 FF	JSR \$FFD5	LOADSP.
. 2019 B0 65	BCS \$2080	Carry set indicates error.
. 201B 20 B7 FF	JSR \$FFB7	Read the status byte.
. 201E D0 60	BNE \$2080	If an error is detected, send out message.
. 2020 00	BRK	Stop processing.

This routine sends the word ERROR to the screen:

. 2080 A2 06	LDX #\$06	Length of the word.
. 2082 BD F8 20	LDA \$20F8,X	Get a character of the message.
. 2085 20 D2 FF	JSR \$FFD2	Send to the screen.
. 2088 CA	DEX	Decrement the count.
. 2089 10 F7	BPL \$2082	If not done, continue.
. 208B 00	BRK	Stop processing.

```
>20F8 52 4F 52 52 45 0D 20 20 : RORRE.  
>2100 53 41 56 50 52 47 00 00 : SAVPRG..
```

## Verify

When a file is to be verified against memory, the file name by which it is known on the tape may be stored in CHR\$ codes in ascending order somewhere in memory. If it is omitted, the first file found on the tape is used. The following outlines the operations required:

1. Load .X with \$01 (the device number of the Datasette). Load .Y with \$00 for a relocated verify, or a nonzero value for a nonrelocated verify. This flag is

overridden if the file was SAVED as a nonrelocatable file, as described previously. Call SETLFS (\$FFBA).

2. Load .A with the length of the file name (\$00 if none), .X with the low byte of the address at which the name is stored, and .Y with the high byte of the address at which the name is stored, and call SETNAM (\$FFBD).
3. Load .A with \$80 to indicate a verify. If it is to be a relocated verify, load .X with the low address to begin verifying and .Y with the high address to begin verifying. Call LOADSP (\$FFD5).
4. Check for errors by checking the carry status and the status variable. The status variable can be loaded into .A with READSS (\$FFB7). A verify error sets bit 4 of the status variable to 1.

**Using Data Files** This section outlines the machine-language equivalents for the BASIC statements OPEN, CLOSE, GET#, and PRINT#. They can be used for data file handling. The key subroutines are OPEN (\$FFC0), CLOSE (\$FFC3), CHKIN (\$FFC6), CHOUT (\$FFC9), CLRCH (\$FFCC), BASIN (\$FFCF), and BSOUT (\$FFD2). Any messages printed to the screen during these operations are controlled by the message flag set with SETMSG (\$FF90).

## Open

When a file is to be opened, and a file name is to be provided, the name (1 to 17 characters) must be stored in CHR\$ codes in ascending order somewhere in memory. The following outlines the operations required:

1. Load .A with the logical file number to use for the file, .X with \$01 (the device number of the Datasette), and .Y with the secondary address (0=read, 1=write with end-of-file, 2=write with end-of-tape). Call SETLFS (\$FFBA).
2. Load .A with the length of the file name (0 if no name), .X with the low byte of the address where the file name is stored, and .Y with the high byte of the address where the file name is stored. Call SETNAM (\$FFBD).
3. Call OPEN (FFC0).
4. Check for errors by examining the carry bit.

## Close

Files can be closed individually with the CLOSE (\$FFC3) routine:

1. Load .A with the logical file number of the file to be closed. Call CLOSE (\$FFC3).

## 2. Check for errors by examining the carry bit and status byte.

All the files in use can be closed and the input and output channels reset to their default devices by calling CLALL (\$FFE7).

## Print

To send information into a file, it must have been opened with a secondary address of 1 or 2 for write. The following outlines the operations required:

1. Load .X with the logical file number of the output file. Designate it as the output channel by calling CHOUT (\$FFC9).
2. Check for errors.
3. Send data to the channel with BSOUT (\$FFD2).
4. When all data has been sent, return the input and output channels to default (keyboard and screen) by calling CLRCH (\$FFCC) and close the file. Or, call CLALL (\$FFE7) to close all the files and reset the input and output channels.

## Get

To receive information from a file, the file must have been opened with a secondary address of 0 for read. The following outlines the operations required:

1. Load .X with the logical file number of the input file. Designate it as the input channel by calling CHKIN (\$FFC6).
2. Check for errors.
3. Receive data from the channel with BASIN (\$FFCF).
4. When complete, return the input and output channels to default (keyboard and screen) by calling CLRCH (\$FFCC) and close the file. Or call CLALL (\$FFE7) to close all files and reset the input and output channels.

**Example:** This program creates a data file on tape and records what the user types in (until a carriage return is received).

. 2000 A9 02	LDA #\$02	Logical file number to use for Datasette file.
. 2002 A2 01	LDX #\$01	Device number of Datasette.
. 2004 A0 01	LDY #\$01	Secondary address signifies write (end-of-file).
. 2006 20 BA FF	JSR \$FFBA	SETLFS.
. 2009 A9 06	LDA #\$06	Length of file name.
. 200B A2 00	LDX #\$00	Low byte of address of file name.
. 200D A0 21	LDY #\$21	High byte of address of file name.

. 200F	20 BD FF JSR \$FFBD	SETNAM.
. 2012	20 C0 FF JSR \$FFC0	OPEN.
. 2015	B0 69 BCS \$2080	If error, go to error message routine.
. 2017	A9 0D LDA #\$0D	Carriage return character.
. 2019	20 D2 FF JSR \$FFD2	Send to screen.
. 201C	A2 02 LDX #\$02	Logical file number of data file.
. 201E	20 C9 FF JSR \$FFC9	Set to output channel.
. 2021	B0 5D BCS \$2080	If error, go to error message routine.
. 2023	20 CF FF JSR \$FFCF	Get a character from keyboard.
. 2026	48 PHA	Save on stack.
. 2027	20 D2 FF JSR \$FFD2	Send to Datassette.
. 202A	68 PLA	Retrieve character.
. 202B	C9 0D CMP #\$0D	Compare to carriage return.
. 202D	D0 F4 BNE \$2023	If not, go on.
. 202F	20 CC FF JSR \$FFCC	Reset I/O channels.
. 2032	A9 02 LDA #\$02	Logical file number of data file.
. 2034	20 C3 FF JSR \$FFC3	CLOSE.
. 2037	B0 47 BCS \$2080	If error, go to error message routine.
. 2039	00 BRK	Stop processing.

This routine sends the message ERROR to the screen:

. 2080	20 E7 FF JSR \$FFE7	Close all the files and reset I/O channels.
. 2083	A2 05 LDX #\$05	Length of message.
. 2085	BD F8 20 LDA \$20F8,X	Get a character of message.
. 2088	20 D2 FF JSR \$FFD2	Send to the screen.
. 208B	CA DEX	Decrement the pointer.
. 208C	10 F7 BPL \$2085	If not done, go on.
. 208E	00 BRK	Stop processing.

These data are needed:

```
>20F8 52 4F 52 52 45 0D 00 00 : RORRE...
>2100 44 41 54 46 49 4C 00 00 : DATFIL..
```

Example: This program reads the data file created by the previous example, and sends the contents to the screen.

. 2000	A9 02 LDA #\$02	Logical file number for data file.
. 2002	A2 01 LDX #\$01	Device number of Datassette.
. 2004	A0 00 LDY #\$00	Secondary address signifies read.
. 2006	20 BA FF JSR \$FFBA	SETLFS.
. 2009	A9 06 LDA #\$06	Length of file name.
. 200B	A2 00 LDX #\$00	Low byte of address of file name.
. 200D	A0 21 LDY #\$21	High byte of address of file name.
. 200F	20 BD FF JSR \$FFBD	SETNAM.
. 2012	20 C0 FF JSR \$FFC0	OPEN.

. 2015 B0 69	BCS \$2080	If error, go to error message routine.
. 2017 A9 0D	LDA #\$0D	Carriage return character.
. 2019 20 D2 FF	JSR \$FFD2	Send to screen.
. 201C A2 02	LDX #\$02	Logical file number of file.
. 201E 20 C6 FF	JSR \$FFC6	Set to input channel.
. 2021 B0 5D	BCS \$2080	If error, go to error message routine.
. 2023 20 CF FF	JSR \$FFCF	Get a character from the file.
. 2026 20 D2 FF	JSR \$FFD2	Send to the screen.
. 2029 20 B7 FF	JSR \$FFB7	Check the status byte.
. 202C F0 F5	BEQ \$2023	If zero, continue.
. 202E C9 40	CMP #\$40	Compare to end-of-file.
. 2030 D0 4E	BNE \$2080	If not, go to error message routine.
. 2032 20 CC FF	JSR \$FFCC	Reset I/O channels.
. 2035 A9 02	LDA #\$02	Logical file number of file.
. 2037 20 C3 FF	JSR \$FFC3	CLOSE.
. 203A B0 44	BCS \$2080	If error, go to error message routine.
. 203C 00	BRK	Stop processing.

This routine sends the message ERROR to the screen:

. 2080 20 E7 FF	JSR \$FFE7	Close all the files and reset I/O channels.
. 2083 A2 05	LDX #\$05	Length of message.
. 2085 BD F8 20	LDA \$20F8,X	Get a character of message.
. 2088 20 D2 FF	JSR \$FFD2	Send to the screen.
. 208B CA	DEX	Decrement the pointer.
. 208C 10 F7	BPL \$2085	If not done, go on.
. 208E 00	BRK	Stop processing.

These data are needed:

```
>20F8 52 4F 52 52 45 0D 00 00 : RORRE...
>2100 44 41 54 46 49 4C 00 00 : DATFIL..
```

## The Printer

Numerous printers are available for use with the Plus/4. Commodore manufactures a number of them, but other vendors' printers can also be used. All of Commodore's printers are intelligent and are connected directly to the serial port. They can be daisy-chained with other serial devices (e.g., a disk drive). Most other vendors offer an interface cable that connects their printer to the serial port and emulates a Commodore printer. Commercial software (e.g., word processors) generally assumes that the printer is configured in this way.

Some vendors may interface through an RS232 connection. Programming for such printers requires using the RS232 port described in another section of this

chapter. Commercial software may or may not be able to access an RS232 printer. When selecting a printer, make sure that the software you intend to use will access it properly.

Each type of printer accepts a different set of commands. Some may allow the printing of graphics characters, upper and lower case, custom designed characters, and so on. It is not possible to completely describe all of the commands accepted by every type of printer here. Refer to the manual that comes with your printer for the commands it accepts.

This section describes the method of accessing any printer through the serial port and, for exemplary purposes, describes the commands accepted by Commodore's MPS-801 printer.

## Using a Printer with BASIC

The first step in accessing the printer is to issue an OPEN statement:

**OPEN filenumber,device,secondary address**

The *filenumber* is the logical file number to be associated with the printer. It can be 0-255, but 128-255 causes a linefeed character to be sent after the carriage return at the end of a line. The MPS-801 does a carriage return and linefeed upon receiving a carriage return character or a linefeed character. As a result, a logical file number from 128 to 255 causes a blank line after each printed line. Other printers may react differently to the two types of logical file numbers. The *device* is the device number; it is generally 4 for a printer. The MPS-801 can be switched between 4 and 5. The *secondary address* is interpreted by the printer.

Printers generally accept commands at two levels. The first is within the secondary address sent to the printer in the open statement. The second is special nonprinting control characters that the printer interprets. The MPS-801 accepts only two values for the secondary address:

0 = default setting is upper case/graphics mode.

7 = default setting is upper/lower case mode.

Other printers may accept different secondary addresses with different results.

Once the printer file is open, data may be directed to the printer in two ways. The first is the CMD command. This is particularly useful in direct mode to get a printout of the program currently in memory. The CMD command changes the default output device to the file named:

**CMD filenumber**

where *filenumber* is the logical file number to which output should be directed. Once this command is executed, all output that would normally go to the screen goes to the named file instead. So, to list a program, type

```
OPEN 4,4:CMD4:LIST  
PRINT#4:CLOSE4
```

The PRINT#4 is needed to restore the default output device to the screen and to tell the printer to stop listening on the serial bus.

When output to the printer is finished, the file should always be closed using

```
CLOSE filenumber
```

where *filenumber* is the logical file number used for the printer. To ensure that the printer's internal buffer has been emptied, and a stop listening command sent, the last data byte sent to the printer should be a carriage return. Usually, in a program, this happens to be the case. When it is not, the CLOSE command should be preceded by

```
PRINT#filenumber
```

The second method of directing data to the printer is with the PRINT# command. Every character sent to the printer is interpreted as a command or data. Generally, CHR\$ codes of 32 through 127, and 160 through 255 are printable characters whereas other values may be control characters. The interpretation of CHR\$ codes depends on the particular printer; check your manual.

For the MPS-801, the following control codes are available:

- 8 = enter dot mode
- 10 = linefeed (and carriage return)
- 13 = carriage return (and linefeed)
- 14 = enter double width character mode
- 15 = enter standard width character mode
- 16 = set the print head position
- 17 = switch to upper/lower case
- 18 = turn on reverse field
- 26 = repeat dot pattern
- 27 = specify dot address for print head

34 = toggle quote mode

145 = switch to upper case/graphics

146 = turn off reverse field

The MPS-801 is always operating in standard width character mode, double width character mode, or dot mode. the default is standard width character mode. In double width character mode, data bytes sent to the printer are interpreted as characters to be printed at twice their normal width. In dot mode, data bytes are interpreted as vertical dot patterns to be printed one column at a time. These modes are switched with the CHR\$ codes 8, 14, and 15. Each remains in effect until another mode switch character is received (even after the file is closed and program ends) or the printer is turned off.

***The Character Modes*** In both standard and double width character mode, two character sets are available. These are upper case/graphics and upper/lower case. Choose which of these sets will be the default set by choosing a secondary address of 0 (or omitting the secondary address) or 7 when opening the printer file. Regardless of which choice is made, the CHR\$ codes 17 (cursor down) and 145 (cursor up) can be used to switch between the sets on a given line. When a carriage return is received, the character set is reset to the default (as chosen with the secondary address).

In addition, the reversed image of each character is available. The CHR\$ codes 18 (reverse on) and 146 (reverse off) can be used to switch between reversed and normal characters on a given line. When a carriage return is received, reverse is automatically turned off.

In either character mode, the spacing between lines is greater than the height of a character (to enhance readability). This results in six lines per inch.

The print head can be positioned to any character position required by using the CHR\$(16) command. The two characters following the CHR\$(16) are interpreted as CHR\$ codes for two-decimal digits specifying the desired character position of the print head. For example,

```
PRINT#4,CHR$(16)"10THIS IS A TEST."
```

prints the sentence starting in position 10 (counting from 0). This position is the same whether in standard or double width mode.

Quote mode for the printer is similar to quote mode for the screen. That is, after an odd number of CHR\$(34) codes on a given line, control codes are printed as reversed characters rather than interpreted as commands until another CHR\$(34) or a carriage return is received.

**Dot Mode** When the printer is placed in dot mode with CHR\$(8), the data characters it receives are interpreted as dot patterns. Each column of seven dots is addressed individually. The low bit is the top dot in the column, and bit 6 is the bottom bit in the column. When the bit is a 1, the dot is inked, and when the bit is a 0, the dot is left blank. The high bit of a data byte must always be set to 1. The printer uses this bit to determine that it has received a data byte rather than a control byte while in dot mode.

In dot mode, the spacing between lines is exactly the height of a column. This results in nine lines per inch.

The print head can be positioned to any column position required by using the CHR\$(27) and CHR\$(16) commands. First the CHR\$(27) is sent to indicate a positioning based on dot columns (rather than character positions). Then the CHR\$(16) is sent to indicate that the head is to be positioned. The first character following the CHR\$(16) is interpreted as the high bit of a 9-bit column position. The next character is interpreted as the low byte of the column position. For example,

```
PRINT#4,CHR$(8)CHR$(27)CHR$(16)CHR$(1)CHR$(4)CHR$(255)
```

prints a vertical bar (specified by CHR\$(255)) in column 260 (counting from 0).

In dot mode, it is possible to repeat a given column of dots a number of times. This is done using the CHR\$(26) command. The byte following this command is interpreted as the number of repetitions of the column, and the next byte as the data to repeat. For example,

```
PRINT#4,CHR$(8)CHR$(255)CHR$(26)CHR$(10)CHR$(193)  
CHR$(255)
```

prints a rectangle consisting of a vertical bar (CHR\$(255)), followed by 10 copies of a column with dots at the top and bottom, and finished with another vertical bar.

**Example:** This example program is meant to be run from a computer attached to an MPS-801 printer set to device number 4. It prints out the lyrics of the Star Spangled Banner, with little (custom-made) flags beside them. Inside the quotation marks, type everything in upper case while holding down the SHIFT key and everything in lower case without using SHIFT.

```
10 OPEN4,4,7  
20 PRINT#4,CHR$(14)"The Star Spangled Banner"  
30 FOR I=1 TO 11:READ A:A$=A$+CHR$(A):NEXT  
40 FOR I=1 TO 8:READ L$(I):NEXT  
50 PRINT#4,CHR$(8)
```

```

60 FORI=1TO8:P=I*10
70 PRINT#4,CHR$(27)CHR$(16)CHR$(0)CHR$(P)A$CHR$(15)" "
L$(I)CHR$(8)
80 PRINT#4,CHR$(27)CHR$(16)CHR$(0)CHR$(P)CHR$(255)
90 NEXT
100 CLOSE4
110 DATA255,213,218,213,218,213,213,213,213,213,213
120 DATA"Oh! say can you see by the dawn's early light,"
130 DATA"What so proudly we hailed at the twilight's last gleaming!"
140 DATA"Whose broad stripes and bright stars thro' the perilous
    fight,"
150 DATA"o'er the ramparts we watched were so gallantly
    streaming;"
160 DATA"And the rockets' red glare, the bombs bursting in air,"
170 DATA"Gave proof thro' the night that our flag was still there."
180 DATA"Oh! say does that star spangled banner yet wave"
190 DATA"O'er the land of the free, and the home of the brave."

```

### Line-by-Line Explanation

- 10 Open a file to the printer with upper/lower case as default.
- 20 Put the printer into double width character mode, and print out the title.
- 30 Read the data for the flag symbol and store in A\$.
- 40 Read the data for the lyrics and store in L\$.
- 50 Put the printer into dot mode.
- 60 I counts the lines. P is the position to start this line. Since I goes from 1 to 8, P will go from 10 to 80, so the high bit of the position is always 0.
- 70 Position the print head in column P, print out a flag, put the printer into standard width character mode, print a line of lyrics, and put the printer back into dot mode.
- 80 Position the print head in column P again, and print the flag pole.
- 90 Go on to the next line.
- 100 Close the printer file.
- 110 Data for flag symbol.
- 120-190 Data for lyrics.

## Using a Printer from Machine Language

All of the printer commands and the functions they perform in BASIC are the same for machine language. Again, the manual relating to the specific printer in use should be referred to for the interpretation of secondary addresses and control codes. The function of the BASIC commands described previously to open and send data to printer files can be accomplished in machine code by using the appropriate operating system subroutines.

In general terms, you can choose one of two levels of communication when you use the printer. The higher level allows the maintenance of multiple open files similar to using OPEN commands in BASIC. At the lower level, it is possible to send data to the printer directly through the serial bus subroutines. Both of these levels are explored and example programs are given in this section. The complete descriptions of the operating system subroutines used can be found in Chapter 5.

**Using the File Structure** This section outlines the machine-language equivalents for the BASIC statements OPEN, CLOSE, and PRINT#. They can be used for all the purposes outlined in the preceding BASIC sections. The key subroutines are OPEN (\$FFC0), CLOSE (\$FFC3), CHOUT (\$FFC9), CLRCH (\$FFCC), and BSOUT (\$FFD2). Any messages printed to the screen during these operations are controlled by the message flag set with SETMSG (\$FF90).

### Open

The OPEN subroutine performs the same function for the printer in machine code as the OPEN command does in BASIC. A secondary address may be sent to control printer functions. The following outlines the operations required:

1. Load .A with the logical file number to use for the file, .X with the device number of the printer, and .Y with the secondary address. Call SETLFS (\$FFBA).
2. Load .A with \$00 (the length of the file name). Call SETNAM (\$FFBD).
3. Call OPEN (\$FFC0).
4. Check for errors by examining the carry bit and status byte.

### Close

Files can be closed individually by using the CLOSE (\$FFC3) routine:

1. Load .A with the logical file number of the file to be closed. Call CLOSE (\$FFC3).

2. Check for errors by examining the carry bit and status byte.

All the files in use can be closed and the input and output channels reset to their default devices by calling CLALL (\$FFE7).

## Print

The following outlines the operations required to send a byte (data or control) to an open printer file:

1. Load .X with the logical file number of the printer file. Designate it as the output channel by calling CHOUT (\$FFC9).
2. Check for errors.
3. Send data to the channel with BSOUT (\$FFD2).
4. When all data has been sent, return the input and output channels to default (keyboard and screen) by calling CLRCH (\$FFCC) and close the file. Or call CLALL (\$FFE7) to close all the files and reset the input and output channels.

**Example:** This program opens a printer file and sends whatever the user types in on the keyboard to it, until a carriage return is received. It uses device number 4 and a secondary address of 7 (which means upper/lower case on the MPS-801). If these are not appropriate, change them.

. 2000 A9 02	LDA #\$02	Logical file number of printer file.
. 2002 A2 04	LDX #\$04	Device number of printer.
. 2004 A0 07	LDY #\$07	Secondary address to send.
. 2006 20 BA FF	JSR \$FFBA	SETLFS.
. 2009 A9 00	LDA #\$00	No filename.
. 200B 20 BD FF	JSR \$FFBD	SETNAM.
. 200E 20 C0 FF	JSR \$FFC0	OPEN.
. 2011 B0 6D	BCS \$2080	If error, go to error message routine.
. 2013 A9 0D	LDA #\$0D	Carriage return character.
. 2015 20 D2 FF	JSR \$FFD2	Send to screen.
. 2018 A2 02	LDX #\$02	Logical file number of printer file.
. 201A 20 C9 FF	JSR \$FFC9	Set to output device.
. 201D B0 61	BCS \$2080	If error, go to error message routine.
. 201F 20 CF FF	JSR \$FFCF	Get keypress from keyboard.
. 2022 20 D2 FF	JSR \$FFD2	Send to printer.
. 2025 C9 0D	CMP #\$0D	Compare to a carriage return character.
. 2027 D0 F6	BNE \$201F	If not, continue.
. 2029 20 CC FF	JSR \$FFCC	Reset I/O channels.
. 202C A9 02	LDA #\$02	Logical file number of printer file.
. 202E 20 C3 FF	JSR \$FFC3	CLOSE.
. 2031 B0 4D	BCS \$2080	If error, go to error message routine.
. 2033 20 B7 FF	JSR \$FFB7	Check status byte.
. 2036 D0 48	BNE \$2080	If not zero, then error.
. 2038 00	BRK	Stop processing.

This is a routine to close all the files and print ERROR on the screen:

- . 2080 20 E7 FF JSR \$FFE7 Close all the files and reset I/O channels.
- . 2083 A2 05 LDX #\$05 Counter for message.
- . 2085 BD F8 20 LDA \$20F8,X Get a byte of message.
- . 2088 20 D2 FF JSR \$FFD2 Send to the screen.
- . 208B CA DEX Decrement the counter.
- . 208C 10 F7 BPL \$2085 If not done, go on.
- . 208E 00 BRK Stop processing.

These are the data for the error message:

```
>20F8 52 4F 52 52 45 OD 00 00 : [RORRE...]
```

**Programming the Serial Bus** Occasionally it is desirable to program directly to the serial bus. The family of operating subroutines used for this purpose consists of LISTN (\$FFB1), SECND (\$FF93), CIOUT (\$FFA8), and UNLSN (\$FFAE). Programming at this level eliminates the file structure used with OPEN and CLOSE. Error checking must be done using the status variable for device not present, and such. When this is done depends to a large extent on what assumptions the particular application allows. When in doubt, check for an error.

## To Send Data to a Printer File

Here is an outline of a procedure:

1. Set the status variable (\$90) to zero.
2. Load .A with the device number of the printer (and store in the current device number at \$AE if this has not been done previously). Call LISTN (\$FFB1). Check for a device not present error in the status variable.
3. Load .A with the secondary address you want to send ORed with \$60. Call SECND (\$FF93).
4. Send the data through .A to the printer using CIOUT (\$FFA8).
5. Call UNLSN (\$FFAE).

**Example:** This program commands the printer to listen and sends to it whatever the user types on the keyboard until a carriage return is received. It uses device number 4 and a secondary address of 7 (which means upper/lower case on the MPS-801). If these are not appropriate, change them.

. 2000 A9 0D	LDA #\$0D	Carriage return character.
. 2002 20 D2 FF	JSR \$FFD2	Send to the screen.
. 2005 A9 00	LDA #\$00	Load .A with zero.
. 2007 85 90	STA \$90	Store in the status register.
. 2009 A9 04	LDA #\$04	Device number of printer.
. 200B 85 AE	STA \$AE	Store in the current device number.
. 200D 20 B1 FF	JSR \$FFB1	LISTN.
. 2010 20 B7 FF	JSR \$FFB7	Read the status register.
. 2013 D0 6B	BNE \$2080	If not zero, error.
. 2015 A9 67	LDA #\$67	Secondary address (7 ORed with \$60). SECND.
. 2017 20 93 FF	JSR \$FF93	Read a keypress from the keyboard.
. 201A 20 CF FF	JSR \$FFCF	Send to the serial bus.
. 201D 20 A8 FF	JSR \$FFA8	Compare with a carriage return.
. 2020 C9 0D	CMP #\$0D	If not equal, continue.
. 2022 D0 F6	BNE \$201A	UNLSN.
. 2024 20 AE FF	JSR \$FFAE	Read the status register.
. 2027 20 B7 FF	JSR \$FFB7	If not zero, error.
. 202A D0 54	BNE \$2080	Stop processing.
. 202C 00	BRK	

This routine sends an unlisten and prints out the error message:

. 2080 20 AE FF	JSR \$FFAE	UNLSN.
. 2083 A2 05	LDX #\$05	Length of message.
. 2085 BD F8 20	LDA \$20F8,X	Get a character of message.
. 2088 20 D2 FF	JSR \$FFD2	Send to the screen.
. 208B CA	DEX	Decrement the pointer.
. 208C 10 F7	BPL \$2085	If not done, continue.
. 208E 00	BRK	Stop processing.

These data are the error message:

>20F8 52 4F 52 52 45 OD 00 00 : RORRE...

## The Modem and Other RS232 Devices

The RS232 port on the Plus/4 is not identical with that of the Commodore 64 and VIC-20. As a result, RS232 interfaces that work with those computers may or may not work with the Plus/4. In particular, the VIC modem and automodems sold for use with the VIC and 64 do not work with the Plus/4. Commodore's new 1660 MODEM/300 is compatible with the Plus/4 as well as the Commodore 64 and VIC-20.

The commands described in this section have been verified with a 1660. The use of other RS232 devices should be similar. Please refer to the documentation

packaged with the device or interface for differences. The use of the RS232 port in machine language is virtually identical to its use in BASIC. Machine-language programmers will find both the BASIC and machine-language descriptions helpful.

## Using an RS232 Device with BASIC

The RS232 port is opened as an input/output file. When it is opened, the communications protocol to be used is established through the control and command registers. The values of these registers are passed to the RS232 port through the file name parameter of the OPEN command. That is, the first two characters of the parameter are interpreted as the CHR\$ codes for the control and command register values. Any subsequent characters are ignored. The secondary address parameter of the OPEN command is ignored. The OPEN command syntax is

`OPEN filenumber,2,0,CHR$(control register)CHR$(command register)`

where *filenumber* is the logical file number to associate with the RS232 port. The logical file number can be 1-255 (128-255 causes a linefeed to follow each carriage return). The control and command register values are passed as a two-character string.

The control register values are as follows:

<i>Bit(s)</i>	<i>Meaning</i>
0-3	baud rate generator
	0000 = 16 times external clock
0001	= 50 baud
0010	= 75
0011	= 109.92
0100	= 134.58
0101	= 150
0110	= 300
0111	= 600 *
1000	= 1200 *
1001	= 1800 *
1010	= 2400 *
1011	= 3600 *
1100	= 4800 *
1101	= 7200 *
1110	= 9600 *
1111	= 19200 *

4	receiver clock source 0 = external receiver clock 1 = baud rate generator
5-6	word length 00 = 8 bits 01 = 7 bits 10 = 6 bits 11 = 5 bits
7	stop bits 0 = 1 stop bit 1 = 2 stop bits

---

\*The 1660 cannot operate above 300 baud.

The command register values are as follows:

<i>Bit(s)</i>	<i>Meaning</i>															
0	data terminal ready 0 = disable receiver and all interrupts (DTR high) 1 = enable receiver and all interrupts (DTR low)															
1	receiver interrupt enable 0 = IRQ interrupt enabled from bit 3 of status register 1 = IRQ interrupt disabled															
2-3	transmitter controls:															
	<table> <thead> <tr> <th>INTERRUPT</th> <th>RTS LEVEL</th> <th>TRANSMITTER STATUS</th> </tr> </thead> <tbody> <tr> <td>00 = disabled</td> <td>high</td> <td>off</td> </tr> <tr> <td>01 = enabled</td> <td>low</td> <td>on</td> </tr> <tr> <td>10 = disabled</td> <td>low</td> <td>on</td> </tr> <tr> <td>11 = disabled</td> <td>low</td> <td>transmit BRK</td> </tr> </tbody> </table>	INTERRUPT	RTS LEVEL	TRANSMITTER STATUS	00 = disabled	high	off	01 = enabled	low	on	10 = disabled	low	on	11 = disabled	low	transmit BRK
INTERRUPT	RTS LEVEL	TRANSMITTER STATUS														
00 = disabled	high	off														
01 = enabled	low	on														
10 = disabled	low	on														
11 = disabled	low	transmit BRK														
4	mode for receiver 0 = normal 1 = echo (bits 2 and 3 must be 00)															
5-7	parity --0 = disabled 001 = odd parity receive and transmit															

---

011 = even parity receive and transmit  
101 = mark parity bit transmit/no parity check  
111 = space parity bit transmit/no parity check

---

Errors associated with the RS232 port can be read from the status variable (ST). The meanings of its bits are as follows:

<i>Bit</i>	<i>Meaning</i>
0	parity error
1	framing error
2	receiver buffer overrun
3	receiver buffer empty
4	clear to send missing*
5	—
6	data set ready missing (data set refers to the modem)
7	break detected*

\*From disassembling the operating system, it appears that these bits will never be set.

Bits 2 and 3 reflect the status of the RS232 buffer. When bit 3 is set, no character was received. When bit 2 is set, the buffer is full. Bit 6 may be set if communication through the port is begun before it fully initializes.

The RS232 logic has a built-in capability to send and receive x-on (transmitting on) and x-off (transmitting off) characters. It is enabled by storing the nonzero x-on character in location 252 (\$FC) and the nonzero x-off character in location 253 (\$FD). When the RS232 buffer is almost full, the x-off character is sent to the host computer. When it is almost empty again, the x-on character is sent. Similarly, when the x-off character is received, the operating system holds off sending data until an x-on character is received. Check the specifications of the host computer you are communicating with to determine the proper values for x-on and x-off.

As with all logical files, the RS232 port should be closed using

#### CLOSE *file number*

When the file is closed, the buffer area and its associated registers are cleared. To check that everything has been received before this is done, you can check the value found at 2003 (\$07D3). This location contains the count for the RS232 buffer.

**Handling the Phone with the 1660 Automodem** The 1660 can be used to dial and answer the phone. Dialing can be done with touch tone frequencies or standard pulse dialing.

Location 64784 (\$FD10) controls the phone. First, make sure the phone is hung up:

**POKE 64784,PEEK(64784) OR 64**

Then, wait briefly and pick up the phone with

**POKE 64784,PEEK(64784) AND 191**

Touch tone dialing is done by sending two tones to the phone simultaneously. Each tone corresponds to a row or column on the touch tone pad.

Column 1: SOUND 1,931,5

2: SOUND 1,940,5

3: SOUND 1,948,5

Row 1: SOUND 2,864,5

2: SOUND 2,879,5

3: SOUND 2,893,5

4: SOUND 2,905,5

Pulse dialing is done by repeatedly picking the phone up and putting it down (as described above; see the example program).

To answer the phone, first hang it up (as above); then wait for a ring with

*linenumber* IF PEEK(64784)AND128 THEN *linenumber*

When the phone is ringing, bit 7 is cleared and the logical statement is false, so the program continues. Then execute the statement that picks up the phone.

**Example:** This example uses the touch tone frequencies to dial the phone, and then starts communicating, until a **C** Q is typed. For communication with many systems, the data received and sent has to be translated to and from Commodore CHR\$ codes. This program does not perform such a function. Also, it may sometimes be necessary to check the status variable after a GET# or PRINT#.

```
10 VOL8
20 FOR I=0 TO 2:READNC(I):NEXT
30 FOR I=0 TO 3:READNR(I):NEXT
```

```

40 DATA931,940,948
50 DATA864,879,893,905
60 PRINT"MAKE SURE A-O SWITCH IS ON O"
70 INPUT"NUMBER TO DIAL";N$
80 IFLEN(N$)<>7THEN70
90 POKE64784,(PEEK(64784)OR64)
100 FORI=1TO500:NEXT
110 POKE64784,(PEEK(64784)AND191)
120 FORI=1TO500:NEXT
130 FORI=1TO7:N=ASC(MID$(N$,I,1))-48
140 IFN<0ORN>9THEN200
150 IFN=0THENNN=11
160 R=INT((N-1)/3):C=N-1-R*3
170 SOUND1,NC(C),5:SOUND2,NR(R),5
180 FORT=1TO50:NEXTT,I
190 PRINT"DIAL COMPLETE":GOTO220
200 PRINT"INVALID NUMBER"
210 VOL0:POKE64784,(PEEK(64784)OR64):END
220 OPEN1,2,0,CHR$(22)+CHR$(5)
230 GET#1,A$:IFA$<>""THENPRINTA$;
240 GETA$:IFA$=""THEN230
250 IFA$=CHR$(171)THEN280
260 PRINT#1,A$;
270 GOTO230
280 CLOSE1:GOTO210

```

### Line-by-Line Explanation

- 10 Turn on volume for tone generation.
- 20 Read the frequencies for columns.
- 30 Read the frequencies for rows.
- 40-50 Data for frequencies.
- 60 Remind the user he or she is originating the communication.
- 70 Receive the number to dial.
- 80 Check for correct length of number.
- 90 Hang up the phone.
- 100 Wait.
- 110 Pick up the phone.
- 120 Wait.
- 130 I points to the digits. Get the value of a digit.
- 140 If not a valid digit, leave.
- 150 A zero is the eleventh button.

- 160 Find the row and column.
- 170 Sound the appropriate tones.
- 180 Wait, then go on to next digit.
- 190 Dialing is completed.
- 200 Print the error message.
- 210 Turn off the volume, and hang up the phone.
- 220 Open the RS232 port.
- 230 Get a byte from the RS232 buffer; if not a null, print it on the screen.
- 240 Get a byte from the keyboard; if a null, go back to 230.
- 250 Check for **C** key pressed with Q key to quit.
- 260 Send the character to RS232.
- 270 Go back to 230.
- 280 Close the RS232 port and go to 210.

**Example:** This example uses pulses to dial the phone and then starts communicating until a **C** or **Q** is typed.

```

10 PRINT"MAKE SURE A-O SWITCH IS ON 0"
20 INPUT"NUMBER TO DIAL";N$
30 IFLEN(N$)<>7THEN20
40 POKE64784,(PEEK(64784)OR64)
50 FORI=1TO500:NEXT
60 POKE64784,(PEEK(64784)AND191)
70 FORI=1TO500:NEXT
80 FORI=1TO7:N=ASC(MIDS(N$,I,1))-48
90 IFN<0ORN>9THEN160
100 IFN=0THENNN=10
110 FORJ=1TON
120 POKE64784,(PEEK(64784)OR64):FORT=1TO40:NEXT
130 POKE64784,(PEEK(64784)AND191):FORT=1TO25:NEXTT,J
140 FORT=1TO350:NEXTTT,I
150 PRINT"DIAL COMPLETE":GOTO180
160 PRINT"INVALID NUMBER"
170 POKE64784,(PEEK(64784)OR64):END
180 OPEN1,2,0,CHR$(22)+CHR$(5)
190 GET#1,A$:IFA$<>""THENPRINTA$;
200 GETA$:IFA$=""THEN190
210 IFA$=CHR$(171)THEN240
220 PRINT#1,A$;
230 GOTO190
240 CLOSE1:GOTO170

```

### Line-by-Line Explanation

- 10 Remind the user he or she is originating the communication.
- 20 Receive the number to dial.
- 30 Check for the correct length of the number.
- 40 Hang up the phone.
- 50 Wait.
- 60 Pick up the phone.
- 70 Wait.
- 80 I points to the digits. Get the value of a digit.
- 90 If not a valid digit, leave.
- 100 A zero is really 10 pulses.
- 110 J counts the pulses.
- 120 Hang up and wait.
- 130 Pick up and wait. Go on to the next pulse.
- 140 Wait, then go on to the next digit.
- 150 Dialing is completed.
- 160 Print error message.
- 170 Hang up the phone.
- 180 Open the RS232 port.
- 190 Get a byte from the RS232 buffer; if not a null, print it on the screen.
- 200 Get a byte from the keyboard; if a null, go back to 190.
- 210 Check for **C** key pressed with Q key to quit.
- 220 Send the character to RS232.
- 230 Go back to 190.
- 240 Close the RS232 port and go to 170.

**Example:** This example waits to answer the phone, and then starts communicating until a **C** **Q** is typed.

```
10 PRINTCHR$(147)“MAKE SURE A-O SWITCH IS ON A”:PRINT:  
PRINT
```

```
20 POKE64784,(PEEK(64784)OR64)
30 FORI=1TO500:NEXT
40 PRINTCHR$(145)"WAITING FOR CALL"
50 IFPEEK(64784)AND128THEN50
60 PRINT"CALL RECEIVED"
70 POKE64784,(PEEK(64784)AND191)
80 PRINT"CALL ANSWERED"
90 OPEN1,2,0,CHR$(22)+CHR$(5)
100 GET#1,A$:IFA$<>""THENPRINTA$;
110 GETA$:IFA$=""THEN100
120 IFA$=CHR$(171)THEN150
130 PRINT#1,A$;:PRINTA$;
140 GOTO100
150 CLOSE1
160 POKE64784,(PEEK(64784)OR64)
```

### Line-by-Line Explanation

- 10 Remind the user he or she is answering.
- 20 Hang up the phone.
- 30 Wait.
- 40 Print the wait message.
- 50 Wait for a ring.
- 60 Print the received message.
- 70 Pick up the phone.
- 80 Print the answered message.
- 90 Open the RS232 port.
- 100 Get a byte from the RS232 buffer; if not a null, print it on the screen.
- 110 Get a byte from the keyboard; if a null, go back to 100.
- 120 Check for **C** key pressed with Q key to quit.
- 130 Send the character to RS232.
- 140 Go back to 100.
- 150 Close the RS232 port.
- 160 Hang up the phone.

## Using an RS232 Device from Machine Language

The RS232 port is handled the same way in machine language as in BASIC. The status variable is read with READSS.

### Open

When the file is to be opened, the control and command registers (two characters) must be stored in ascending order somewhere in memory. The following outlines the operations required:

1. Load .A with the logical file number to use for the file, .X with \$02 (the device number of the RS232 port), and .Y with \$FF (to signify no secondary address). Call SETLFS (\$FFBA).
2. Load .A with the length of the register information (usually 2), .X with the low byte of the address where the registers are stored, and .Y with the high byte of the address where the registers are stored. Call SETNAM (\$FFBD).
3. Call OPEN (\$FFC0).
4. Check for errors by examining the carry bit.

### Close

Files can be closed individually with the CLOSE (\$FFC3) routine:

1. Load .A with the logical file number of the file to be closed. Call CLOSE (\$FFC3).
2. Check for errors by examining the carry bit.

All the files in use can be closed and the input and output channels reset to their default devices by calling CLALL (\$FFE7).

### Print

To send information to the RS232 port, the port must have been opened. The following outlines the operations required:

1. Load .X with the logical file number of the output file. Designate it as the output channel by calling CHOUT (\$FFC9).
2. Check the status byte for errors.

3. Send data to the channel with BSOUT (\$FFD2).
4. Check the status byte for errors.
5. When all data has been sent, return the input and output channels to default (keyboard and screen) by calling CLRCH (\$FFCC) and close the file, or call CLALL (\$FFE7) to close all the files and reset the input and output channels.

## Get

To receive information from the RS232 port, the port must have been opened. The following outlines the operations required:

1. Load .X with the logical file number of the input file. Designate it as the input channel by calling CHKIN (\$FFC6).
2. Check the status byte for errors.
3. Receive data from the channel with BASIN (\$FFCF).
4. Check the status byte for errors.
5. When all data received, return the input and output channels to default (keyboard and screen) by calling CLRCH (\$FFCC) and close the file, or call CLALL (\$FFE7) to close all the files and reset the input and output channels.

## The Joystick Ports

The Commodore Plus/4 is equipped with two joystick ports located on the rear of the computer. The Commodore T-1341 joystick is compatible with the Plus/4. Older Commodore joysticks are not compatible. The ports are numbered 1 and 2.

Fundamentally, a Commodore joystick consists of five switches, each of which is assigned to one of the following: up, down, right, left, and fire. In machine language, the programmer must examine a bit reflecting the status of each switch to read joystick input. In BASIC, however, the JOY function can be used to determine the joystick status.

## Using Joysticks with BASIC

The JOY function has the following syntax:

**JOY (port)**

where *port* is the joystick port to use (either 1 or 2). The function returns a

numeric value depending on the state of the joystick. The following table lists the possible values returned:

<i>Value</i>	<i>Joystick State</i>
0	neutral
1	up
2	up and right
3	right
4	down and right
5	down
6	down and left
7	left
8	up and left

In addition, when the fire button is pressed, 128 is added to the values listed in the table.

**Example:** This program plots and unplots a circle on the high-resolution graphic screen. Whenever the joystick in port 1 is pushed away from neutral, the circle is moved in a corresponding direction. Whenever the fire button of the joystick in port 1 is pressed, a circle is plotted and remains in place. To exit, press the space bar.

```

10 FOR I=1 TO 8:READDX(I),DY(I):NEXT
20 GRAPHIC1,1
30 CIRCLE1,160,100,10,10
40 SSHAPECS$,150,90,170,110
50 GSHAPECS$,150,90,4
60 X=150:Y=90
70 GSHAPECS$,X,Y,4
80 GSHAPECS$,X,Y,4
90 GETA$:IFA$=""THEN150
100 J=JOY(1):IFJ=0THEN70
110 IF (JAND128)=0THEN130
120 GSHAPECS$,X,Y,4
130 J=JAND15:X=X+DX(J):Y=Y+DY(J)
140 GOTO70
150 CHAR1,1,23,"HIT KEY"
160 GETKEYA$:GRAPHIC0:END
170 DATA0,-1,1,-1,1,0
180 DATA1,1,0,1,-1,1,-1,0
190 DATA-1,-1

```

### Line-by-Line Explanation

- 10        Read in the amount to change *x*- and *y*-coordinates based on JOY value.

- 20     Enter high-resolution graphic mode.
- 30     Draw a circle.
- 40     Save the circle in C\$.
- 50     Erase the circle.
- 60     Set the initial coordinate values.
- 70     Draw the circle.
- 80     Erase the circle.
- 90     Look for the space bar.
- 100    Get the current value of JOY. If neutral, go back to line 70.
- 110    Check for fire button. If not, go on to line 130.
- 120    Draw the circle.
- 130    Use the value of JOY (fire button status removed) to change the coordinates.
- 140    Go back to line 70.
- 150    Write HIT KEY on the screen.
- 160    Wait for a key, then return to the text mode and stop.
- 170-190 Data for change in  $x$ - and  $y$ -coordinates based on JOY value.

## Using Joysticks from Machine Language

In machine language, the joysticks must be read through the keyboard latch at \$FF08 on the graphics chip. Since the normal system interrupt service routine uses this register for keyboard scanning, interrupts would normally be turned off during this process. For joystick port 1, store \$FA to \$FF08 and then read \$FF08. For joystick port 2, use \$FD. Usually, this should be done at least twice and the results compared to avoid spurious readings. Additional debouncing, such as reading the ports somewhat infrequently, may be required to provide smooth joystick action.

The results of reading the keyboard latch are slightly different for the two ports. In particular, port 1 uses bit 6 for the fire button switch, whereas port 2 uses bit 7. The relationships between bit and switch are as follows:

<i>Bit</i>	<i>Switch</i>
0	Up
1	Down

2	Left
3	Right
4	Unused
5	Unused
6	Fire for port 1 (unused for 2)
7	Fire for port 2 (unused for 1)

When a switch is engaged, the bit is cleared to 0; otherwise it is set to 1. All of the unused bits are always set to 1. As a result, an \$FF indicates a neutral position for the joystick.

**Example:** This is a subroutine for reading joystick port 1. On return, carry clear means FIRE; carry set is no FIRE. The X register is set to -1 (\$FF), 0, or 1 to indicate left, middle, and right positions, respectively, in the horizontal direction. The Y register is set to -1 (\$FF), 0, or 1 to indicate down, middle, and up positions, respectively, in the vertical direction.

. 2000 A2 FA	LDX #\$FA	Latch value for port 1.
. 2002 78	SEI	Disable interrupts.
. 2003 8E 08 FF	STX \$FF08	Store latch value in keyboard latch register.
. 2006 AD 08 FF	LDA \$FF08	Read keyboard latch register.
. 2009 8E 08 FF	STX \$FF08	Store latch value in keyboard latch register.
. 200C CD 08 FF	CMP \$FF08	Compare to previous reading.
. 200F D0 F2	BNE \$2003	If not equal, do it again.
. 2011 58	CLI	Enable interrupts.
. 2012 A2 00	LDX #\$00	Initially set changes to zero.
. 2014 A0 00	LDY #\$00	
. 2016 4A	LSR	Look at bit 0.
. 2017 B0 01	BCS \$201A	If set, switch is off.
. 2019 C8	INY	Up switch is on, increment y change.
. 201A 4A	LSR	Look at bit 1.
. 201B B0 01	BCS \$201E	If set, switch is off.
. 201D 88	DEY	Down switch is on, decrement y change.
. 201E 4A	LSR	Look at bit 2.
. 201F B0 01	BCS \$2022	If set, switch is off.
. 2021 CA	DEX	Left switch is on, decrement x change.
. 2022 4A	LSR	Look at bit 3.
. 2023 B0 01	BCS \$2026	If set, switch is off.
. 2025 E8	INX	Right switch is on, increment x change.
. 2026 4A	LSR	Skip bit 4.
. 2027 4A	LSR	Skip bit 5.
. 2028 4A	LSR	Carry reflects status of fire switch.
. 2029 60	RTS	Return.

For port 2, the \$FA in the instruction at \$2000 must be changed to \$FD and an additional LSR instruction placed at \$2029, with the RTS at \$202A.



---

# Appendix A

## Error Messages

### **BASIC 3.5 Error Messages**

This appendix explains the BASIC 3.5 error messages in numerical order. When you are using BASIC and get an error, you can display the number of the error by PRINTing ER. ER is the system reserved variable that contains the number of the current error. PRINT ERR\$(ER) to display the brief error message for the current error. The only purpose for the error number is to identify the error with the ER and ERR\$ functions, which are especially useful in error-trapping routines. See the TRAP command for additional information.

#### **1 TOO MANY FILES**

BASIC allows only 10 open files at a time. This message appears after you attempt to OPEN an 11th file. To remedy, CLOSE an opened file and then reissue the OPEN command that was rejected for being the 11th file.

#### **2 FILE OPEN**

This message actually tells you that you have illegally used a duplicate logical file number in an OPEN command. BASIC does not permit duplicate logical file numbers, even if you are opening different types of devices. As long as a file remains open, you cannot reuse its logical file number in another OPEN command.

#### **3 FILE NOT OPEN**

This message tells you that you have issued a file command without first issuing an OPEN command. BASIC does not permit commands sent to a file or device until after that file or device has been accessed by an OPEN command.

**4 FILE NOT FOUND**

This message is displayed in response to an unsuccessful LOAD or DLOAD command. If you are searching a tape for a file, this message indicates that an end-of-tape marker has been read. If you are loading from disk, the message indicates that the file is not saved on that disk. Issue a DIRECTORY command to display the disk contents. It is possible that you misspelled the file name when you issued the DLOAD command.

**5 DEVICE NOT PRESENT**

This message is displayed in response to a command to a peripheral device when the computer cannot access that device because it is not turned on or is not properly connected.

**6 NOT INPUT FILE**

This message indicates that you have tried to INPUT or GET data from a file you created as an output-only file.

**7 NOT OUTPUT FILE**

This message indicates that you have tried to output data to a file you created as an input-only file.

**8 MISSING FILE NAME**

This message is displayed when BASIC requires a file name in a LOAD, OPEN, or SAVE command you have issued to the disk drive.

**9 ILLEGAL DEVICE NUMBER**

This message indicates that you sent a command to a peripheral that cannot execute the command. You probably used the wrong device number in the command (e.g., SAVE "LETTERFILE",4, which tells BASIC to save a file to the printer).

**10 NEXT WITHOUT FOR**

BASIC requires a FOR command for every NEXT, but not a NEXT for every FOR (although no NEXT would mean the FOR loop would execute only once). This message indicates that you omitted or misplaced the NEXT command. This is most likely to occur when multiple FOR ... NEXT loops are nested incorrectly.

## 11 SYNTAX ERROR

This message indicates that you have not followed the BASIC syntax rules. Check the command to see if you misspelled a BASIC keyword, typed a non-BASIC word without enclosing it in quotes or placing it in a REM statement, added something to a command that is not allowed, omitted something required in a command, or used an odd number of parentheses.

## 12 RETURN WITHOUT GOSUB

RETURN commands must always accompany GOSUB commands. This message indicates that BASIC found a RETURN command when a GOSUB was not currently executing.

## 13 OUT OF DATA

This message indicates that the current READ command cannot find a DATA value because no DATA command exists or all the DATA values have been read previously. Although it is legal to have excess DATA values in a program, all READ variables must be matched to DATA values. You can use the RESTORE command to reREAD DATA.

## 14 ILLEGAL QUANTITY

This message indicates that the current command or function parameter contains a number that is outside the legal range for the circumstances. For example, the first parameter in a COLOR command, color source, can be 0-4, so COLOR 6,1,2 results in an ILLEGAL QUANTITY error.

## 15 OVERFLOW

This message indicates that the result of the current calculation is a number with absolute value greater than 1.701411833E+38, which is the largest legal number in BASIC 3.5.

## 16 OUT OF MEMORY

This message indicates that the current program is too large for the available RAM, or that the program contains too many DO, FOR, or GOSUB commands, or that the program contains DO commands with no LOOP or EXIT command, or GOSUB commands with no RETURN command. This error can also be caused by repeated assignments to the same string variable; a call to the FRE function may correct this problem.

**17 UNDEF'D STATEMENT**

This message indicates that the current command refers to a line number that does not exist in the program. For example, a GOTO or THEN command that sends the program to a line that does not exist in the program results in an UNDEF'D STATEMENT error.

**18 BAD SUBSCRIPT**

This message indicates that the current command refers to an array subscript that is outside the array dimensions established in the array's DIM command or the default array dimensions.

**19 REDIM'D ARRAY**

This message indicates that you have tried to reDIM an array that was previously DIMed, that you have inadvertently reused an array variable name in another DIM command, or that you have tried to DIM an array after you used the array in the program. Arrays can be DI Mensioned only once, and you cannot dimension an array after you use it in the program even if you have not previously DIMed the array. If you access an array before you DIM it, BASIC automatically DIMs the array with a default dimension of 10. You cannot DIM an array after it has been DIMed by default.

**20 DIVISION BY ZERO**

This message indicates that the current command tried to perform a division with a divisor of zero, which is illegal. This usually happens when you are dividing with variables and the divisor variable is assigned a value of zero. Use a TRAP routine to check for this error to avoid aborting the program.

**21 ILLEGAL DIRECT**

This message indicates that you issued a command in immediate mode (also called direct mode) that can be issued only in program mode.

**22 TYPE MISMATCH**

This message indicates that the current command assigns a value to the wrong type of variable. For example, assigning a text string value to an integer variable results in a TYPE MISMATCH error.

**23 STRING TOO LONG**

This message indicates that a text string is longer than 255 characters, which is the maximum size accepted by BASIC 3.5.

**24 FILE DATA**

This message indicates that BASIC has read bad data from a tape file.

**25 FORMULA TOO COMPLEX**

This message indicates that the current calculation is too long or contains too many expressions in parentheses.

**26 CAN'T CONTINUE**

This message indicates that the CONT command you just executed cannot resume program execution. You may have changed the program after you stopped it. See CONT.

**27 UNDEF'D FUNCTION**

This message indicates that the current command refers to a user-defined function that has not been defined in the program. User-defined functions must be defined in the program before they can be executed.

**28 VERIFY**

This message appears after you execute a VERIFY command to compare the current program with one on tape or disk, or when you use VERIFY to search a tape. The VERIFY message indicates that the program currently in memory is being compared with the program on tape or disk. If it is followed by the word ERROR, the program in memory does not match the program on tape or disk.

**29 LOAD**

This message indicates that the LOAD command just issued was not successful. Repeat the load.

**30 BREAK**

When you interrupt a program execution by pressing the STOP key or including a STOP command, the BREAK message is displayed. BREAK indicates the line number at which the program was stopped.

**31 CAN'T RESUME**

This message indicates that the program found a RESUME command when a TRAP command was not currently executing. RESUME works only with the TRAP command.

**32 LOOP NOT FOUND**

This message indicates that BASIC cannot locate a LOOP command, which must accompany a DO command. Either you left out the LOOP command or improper nesting of multiple loops isolated the DO and LOOP commands.

**33 LOOP WITHOUT DO**

This message indicates that BASIC cannot locate a DO command, which must accompany a LOOP command. Either you left out the DO command or improper nesting of multiple loops isolated the DO and LOOP commands.

**34 DIRECT MODE ONLY**

This message indicates that you issued a command in a program that can be issued only in immediate mode (also called direct mode).

**35 NO GRAPHICS AREA**

This message indicates that BASIC has found a drawing command, such as BOX, when none of the graphic drawing modes was in effect. A GRAPHIC command turning on modes 1 through 4 must precede drawing commands.

**36 BAD DISK**

This message indicates that the disk you are trying to HEADER is defective or that you attempted to execute a partial header on a disk that has not previously been fully headered. Repeat the command to be sure.

**DOS Error Messages**

DOS error messages are returned with an error number, a message, and the track and sector of the error. Messages returned with error numbers less than 20 can be ignored. Everything is okay if the error number is 00. And, 01 indicates a SCRATCHed file report. Some errors in the transmission of data from a diskette

(23, 24, and 27) can be caused by improper grounding (e.g., you are not using a three-prong outlet, your outlet is not properly grounded, etc.). Check that your equipment is grounded properly if these errors occur. In general, if a diskette causes errors that cannot be explained, you should make copies of all the information on the diskette. Then, attempt to do a complete HEADER on the diskette. If it is performed successfully, you can reuse the diskette. Otherwise, toss it.

---

**20 READ ERROR (header block not found)**

The header of the requested data block could not be located by the DOS. Either an illegal sector number was specified, or the header block is corrupt.

**21 READ ERROR (sync mark not found)**

The sync mark of the requested track could not be located by the DOS. This message can occur because of misalignment of the heads. Misalignment can be a hardware problem or an improperly inserted diskette. This message can also occur if no diskette is present or if the diskette is not formatted.

**22 READ ERROR (data block not found)**

The data read from the requested sector was corrupt. This can indicate an illegal sector number request, or a corrupt sector.

**23 READ ERROR (data block checksum error)**

The checksum read from the requested sector did not match the checksum of its data. This can indicate a corrupt sector.

**24 READ ERROR (data block decoding error)**

At least 1 byte of data read from the requested sector was corrupt. This can indicate a corrupt sector.

**25 WRITE ERROR (verification of data error)**

When validated, the data byte just written to the diskette did not match the data in memory. This can be caused by a physically damaged diskette. An unclosed file will result. Perform a validate (COLLECT) operation to delete the unclosed file.

**26 WRITE PROTECT ON**

A write protect tab was detected on a diskette on which a write operation was attempted. Remove the tab or use another formatted diskette.

**27 READ ERROR (header block checksum error)**

The checksum from the header block of the requested sector was in error. This can indicate a corrupt sector.

**28 WRITE ERROR (data block too long)**

The sync mark of the next header block could not be located. This message can occur as a result of a hardware problem or an incorrectly formatted diskette.

**29 DISK ID MISMATCH**

The ID read from the header block did not match the ID in memory. The ID of a diskette is permanently recorded in every header block. This error can result from not initializing the diskette or from a corrupt header block.

**30 SYNTAX ERROR (general)**

DOS could not understand a command sent to it through the command channel. This error can be caused by an incorrect number of file names, incorrect punctuation, or other errors in the DOS command.

**31 SYNTAX ERROR (command not valid)**

The command sent to the DOS through the command channel was not recognized by DOS. This can be caused by not starting the command in the first position of the string sent.

**32 SYNTAX ERROR (command too long)**

A command longer than 58 characters was sent to DOS through the command channel.

**33 SYNTAX ERROR (file name not valid)**

A file name sent to DOS was incorrect. This can mean a wildcard was used illegally.

**34 SYNTAX ERROR (file name missing)**

DOS did not receive a necessary file name. The file name could be present but not recognizable (e.g., necessary separating punctuation is missing).

**39 SYNTAX ERROR (command not valid)**

A command sent to the DOS through the command channel was not recognized by DOS.

**50 RECORD NOT PRESENT**

This error is the result of attempting to access a record beyond the end of a relative file. During addition of a new record, this message is expected and can be ignored. Reposition the pointer before attempting a GET or INPUT.

**51 OVERFLOW IN RECORD**

The information sent to a relative file record in a PRINT# command exceeded the record length in the file. The information is truncated. It should be noted that a carriage return character sent as a record terminator counts as a byte in the record.

**52 FILE TOO LARGE**

Creating the record number in the current Position command in a relative file will cause a disk overflow.

**60 WRITE FILE OPEN**

Read access was requested for a write file that was not closed.

**61 FILE NOT OPEN**

An attempt was made to access a file that was never opened.

**62 FILE NOT FOUND**

A file was requested that does not exist on the diskette.

**63 FILE EXISTS**

The file name designated for a file being created already appears on the diskette.

**64 FILE TYPE MISMATCH**

The designated file type of a requested file was not the same as the file's type on the diskette.

**65 NO BLOCK**

The block designated in a block allocate (B-A) command was already allocated. The track and sector returned indicate the next available block with a higher number. If no block with a higher number is available, zeroes are returned.

**66 ILLEGAL TRACK AND SECTOR**

The designated sector does not exist on the diskette. This can be caused by an attempt to direct access an illegal sector or by a corrupt track / sector link to the next block in a file.

**67 ILLEGAL SYSTEM T OR S**

The DOS attempted to access an illegal sector.

**70 NO CHANNEL**

The designated channel was already in use, or all channels were in use. Five sequential files, or six direct-access channels can be open at any one time.

**71 DIRECTORY ERROR**

The BAM in memory is corrupt. Either allocation has been performed incorrectly or the BAM in memory has been overwritten. The diskette can be reinitialized to correct this problem, but active files may be affected by this action.

**72 DISK FULL**

All of the data blocks, or all of the directory area on the diskette, has been used.

**73 DOS MISMATCH**

This message can occur when an attempt to write on a disk formatted by another version of DOS is made. More often it occurs when the drive is first turned on; the message can be ignored.

**74 DRIVE NOT READY**

There is no diskette in the designated drive, or the diskette is not properly inserted.

---

**Machine Language Monitor and ROM Subroutine Error Messages**

- 0 The routine was terminated by the STOP key.
- 1 The logical file table is full (too many files are open).
- 2 The specified logical file is already open.
- 3 The specified logical file is not open.
- 4 The specified file name is not found on the specified device.
- 5 The specified device is not present.
- 6 The specified device is not an input device.
- 7 The specified device is not an output device.
- 8 The file name was missing.
- 9 The specified device is illegal for this purpose.

---

# Appendix B

## BASIC Tokens

In BASIC programs, the keywords (e.g., PRINT, GET, etc.) are not stored as character strings. Each keyword has a unique “token” assigned to it that is stored instead of the word itself. BASIC recognizes the tokens because their high bit is always set. When you enter a BASIC line, BASIC translates the keywords into tokens for storage. When you LIST a program, BASIC translates the tokens back into keywords.

<i>Dec</i>	<i>Hex</i>	<i>Keyword</i>	<i>Dec</i>	<i>Hex</i>	<i>Keyword</i>
128	\$80	END	150	\$96	DEF
129	\$81	FOR	151	\$97	POKE
130	\$82	NEXT	152	\$98	PRINT#
131	\$83	DATA	153	\$99	PRINT
132	\$84	INPUT#	154	\$9A	CONT
133	\$85	INPUT	155	\$9B	LIST
134	\$86	DIM	156	\$9C	CLR
135	\$87	READ	157	\$9D	CMD
136	\$88	LET	158	\$9E	SYS
137	\$89	GOTO	159	\$9F	OPEN
138	\$8A	RUN	160	\$A0	CLOSE
139	\$8B	IF	161	\$A1	GET
140	\$8C	RESTORE	162	\$A2	NEW
141	\$8D	GOSUB	163	\$A3	TAB(
142	\$8E	RETURN	164	\$A4	TO
143	\$8F	REM	165	\$A5	FN
144	\$90	STOP	166	\$A6	SPC(
145	\$91	ON	167	\$A7	THEN
146	\$92	WAIT	168	\$A8	NOT
147	\$93	LOAD	169	\$A9	STEP
148	\$94	SAVE	170	\$AA	+
149	\$95	VERIFY	171	\$AB	-

<i>Dec</i>	<i>Hex</i>	<i>Keyword</i>	<i>Dec</i>	<i>Hex</i>	<i>Keyword</i>
172	\$AC	*	214	\$D6	RESUME
173	\$AD	/	215	\$D7	TRAP
174	\$AE	↑	216	\$D8	TRON
175	\$AF	AND	217	\$D9	TROFF
176	\$B0	OR	218	\$DA	SOUND
177	\$B1	>	219	\$DB	VOL
178	\$B2	=	220	\$DC	AUTO
179	\$B3	<	221	\$DD	PUDEF
180	\$B4	SGN	222	\$DE	GRAPHIC
181	\$B5	INT	223	\$DF	PAINT
182	\$B6	ABS	224	\$E0	CHAR
183	\$B7	USR	225	\$E1	BOX
184	\$B8	FRE	226	\$E2	CIRCLE
185	\$B9	POS	227	\$E3	GSHAPE
186	\$BA	SQR	228	\$E4	SSHAPE
187	\$BB	RND	229	\$E5	DRAW
188	\$BC	LOG	230	\$E6	LOCATE
189	\$BD	EXP	231	\$E7	COLOR
190	\$BE	COS	232	\$E8	SCNCLR
191	\$BF	SIN	233	\$E9	SCALE
192	\$C0	TAN	234	\$EA	HELP
193	\$C1	ATN	235	\$EB	DO
194	\$C2	PEEK	236	\$EC	LOOP
195	\$C3	LEN	237	\$ED	EXIT
196	\$C4	STR\$	238	\$EE	DIRECTORY
197	\$C5	VAL	239	\$EF	DSAVE
198	\$C6	ASC	240	\$F0	DLOAD
199	\$C7	CHR\$	241	\$F1	HEADER
200	\$C8	LEFT\$	242	\$F2	SCRATCH
201	\$C9	RIGHT\$	243	\$F3	COLLECT
202	\$CA	MID\$	244	\$F4	COPY
203	\$CB	GO	245	\$F5	RENAME
204	\$CC	RGR	246	\$F6	BACKUP
205	\$CD	RCLR	247	\$F7	DELETE
206	\$CE	RLUM	248	\$F8	RENUMBER
207	\$CF	JOY	249	\$F9	KEY
208	\$D0	RDOT	250	\$FA	MONITOR
209	\$D1	DEC	251	\$FB	USING
210	\$D2	HEX\$	252	\$FC	UNTIL
211	\$D3	ERR\$	253	\$FD	WHILE
212	\$D4	INSTR	254	\$FE	unknown
213	\$D5	ELSE	255	\$FF	π

---

# Appendix C

## CHR\$ Codes

CHR\$ (character) codes are recognized by BASIC as letters, numbers, other characters, and commands. The command codes (0–31 and 128–159) tell BASIC to change its output in some way (e.g., change the color of the characters PRINTed on the screen, switch to upper/lower case mode, etc.). The remaining codes represent the characters (letters, numbers, symbols, and graphics) the Plus/4 recognizes.

CHR\$ Number		Display (Quote Mode)			Function	Keys to Press
Dec	Hex	Upper Case/ Graphic	Upper/lower Case			
0	00					
1	01					
2	02					
3	03					
4	04					
5	05	E	e	White		Control/Wht
6	06					
7	07					
8	08	H	h	Disable SHIFT	Control/H	
9	09	I	i	Enable SHIFT	Control/I	
10	0A					
11	0B					
12	0C					
13	0D	M	m	RETURN	Return	
14	0E	N	n	Turn on Upper/ lower case mode	Control N	
15	0F					
16	10					
17	11	Q	q	Cursor down	Cursor down	

<i>CHR\$ Number</i>		<i>Display (Quote Mode)</i>			<i>Function</i>	<i>Keys to Press</i>
<i>Dec</i>	<i>Hex</i>	<i>Upper Case/ Graphic</i>	<i>Upper/lower Case</i>			
18	12	<b>R</b>	<b>r</b>	Reverse on	Control Rvs On	
19	13	<b>S</b>	<b>s</b>	Home	Clear/Home	
20	14	<b>T</b>	<b>t</b>	Delete	Inst/Del	
21	15					
22	16					
23	17					
24	18					
25	19					
26	1A					
27	1B	<b>L</b>	<b>l</b>	Escape	Esc	
28	1C	<b>E</b>	<b>e</b>	Red	Control/Red	
29	1D	<b>J</b>	<b>j</b>	Cursor right	Cursor right	
30	1E	<b>U</b>	<b>u</b>	Green	Control/Grn	
31	1F	<b>H</b>	<b>h</b>	Blue	Control/Blu	
32	20					
33	21					
34	22					
35	23					
36	24					
37	25					
38	26					
39	27					
40	28					
41	29					
42	2A					
43	2B					
44	2C					
45	2D					
46	2E					
47	2F					
48	30	<b>0</b>	<b>0</b>			
49	31	<b>1</b>	<b>1</b>			
50	32	<b>2</b>	<b>2</b>			
51	33	<b>3</b>	<b>3</b>			
52	34	<b>4</b>	<b>4</b>			
53	35	<b>5</b>	<b>5</b>			
54	36	<b>6</b>	<b>6</b>			
55	37	<b>7</b>	<b>7</b>			
56	38	<b>8</b>	<b>8</b>			

*CHR\$ Number**Display (Quote Mode)**Upper Case/ Upper/lower*

<i>Dec</i>	<i>Hex</i>	<i>Graphic</i>	<i>Case</i>	<i>Function</i>	<i>Keys to Press</i>
57	39	9	9		
58	3A	:	:		
59	3B	;	;		
60	3C	<	<		
61	3D	=	=		
62	3E	>	>		
63	3F	?	?		
64	40	C	c		
65	41	A	a		
66	42	B	b		
67	43	C	c		
68	44	D	d		
69	45	E	e		
70	46	F	f		
71	47	G	g		
72	48	H	h		
73	49	I	i		
74	4A	J	j		
75	4B	K	k		
76	4C	L	l		
77	4D	M	m		
78	4E	N	n		
79	4F	O	o		
80	50	P	p		
81	51	Q	q		
82	52	R	r		
83	53	S	s		
84	54	T	t		
85	55	U	u		
86	56	W	w		
87	57	X	x		
88	58	Y	y		
89	59	Z	z		
90	5A	[	[		
91	5B	£	£		
92	5C	]	]		
93	5D	↑	↑		
94	5E	↓	↓		
95	5F	←	←		

<i>CHR\$ Number</i>		<i>Display (Quote Mode)</i>			<i>Function</i>	<i>Keys to Press</i>
<i>Dec</i>	<i>Hex</i>	<i>Upper Case/ Graphic</i>	<i>Upper/lower Case</i>			
96	60					
97	61	█	▀			
98	62	█	▀			
99	63	█	▀			
100	64	█	▀			
101	65	█	▀			
102	66	█	▀			
103	67	█	▀			
104	68	█	▀			
105	69	█	▀			
106	6A	█	▀			
107	6B	█	▀			
108	6C	█	▀			
109	6D	█	▀			
110	6E	█	▀			
111	6F	█	▀			
112	70	█	▀			
113	71	█	▀			
114	72	█	▀			
115	73	█	▀			
116	74	█	▀			
117	75	█	▀			
118	76	█	▀			
119	77	█	▀			
120	78	█	▀			
121	79	█	▀			
122	7A	█	▀			
123	7B	█	▀			
124	7C	█	▀			
125	7D	█	▀			
126	7E	█	▀			
127	7F	█	▀			
128	80					
129	81	█	▀		Orange	
130	82	█	▀		Flash on	Control/Flash On
131	83					
132	84	█	▀		Flash off	Control/Flash Off
133	85				F1*	
134	86				F3*	

<i>CHR\$ Number</i>		<i>Display (Quote Mode)</i>		<i>Function</i>	<i>Keys to Press</i>
<i>Dec</i>	<i>Hex</i>	<i>Upper Case/ Graphic</i>	<i>Upper/lower Case</i>		
135	87			F5*	
136	88			F7*	
137	89			F2*	
138	8A			F4*	
139	8B			F6*	
140	8C			HELP(F8)*	
141	8D	▀	▀	Disabled RETURN	Shift Return
142	8E	▀	▀	Switch to Upper case/ Graphic mode	
143	8F				
144	90	█	█	Black	Control/ Blk
145	91	█	█	Cursor up	Cursor up
146	92	█	█	Reverse off	Control/ Rvs Off
147	93	█	█	Clear/ Home	Shift/ Clear/ Home
148	94	█	█	Insert	Shift/ Inst/ Del
149	95	█	█	Brown	█ Brn
150	96	█	█	Yellow-Green	█ Yl Grn
151	97	█	█	Pink	█ Pink
152	98	█	█	Blue-Green	█ Bl Grn
153	99	█	█	Light Blue	█ L Blu
154	9A	█	█	Dark Blue	█ D Blu
155	9B	█	█	Light Green	█ L Grn
156	9C	█	█	Purple	Control/ Pur
157	9D	█	█	Cursor left	Cursor left
158	9E	█	█	Yellow	Control/ Yel
159	9F	█	█	Cyan	Control/ Cyn
160	A0				
161	A1				
162	A2				
163	A3				
164	A4				
165	A5				
166	A6				
167	A7				
168	A8				
169	A9				
170	AA				
171	AB				
172	AC				

<i>CHR\$ Number</i>		<i>Display (Quote Mode)</i>			<i>Function</i>	<i>Keys to Press</i>
<i>Dec</i>	<i>Hex</i>	<i>Upper Case/ Graphic</i>	<i>Upper/lower Case</i>			
173	AD	□	□			
174	AE	□	□			
175	AF	□	□			
176	B0	□	□			
177	B1	□	□			
178	B2	□	□			
179	B3	□	□			
180	B4	□	□			
181	B5	□	□			
182	B6	□	□			
183	B7	□	□			
184	B8	□	□			
185	B9	□	□			
186	BA	□	□			
187	BB	□	□			
188	BC	□	□			
189	BD	□	□			
190	BE	□	□			
191	BF	□	□			
192	C0	□	□			
193	C1	□	□			
194	C2	□	□			
195	C3	□	□			
196	C4	□	□			
197	C5	□	□			
198	C6	□	□			
199	C7	□	□			
200	C8	□	□			
201	C9	□	□			
202	CA	□	□			
203	CB	□	□			
204	CC	□	□			
205	CD	□	□			
206	CE	□	□			
207	CF	□	□			
208	D0	□	□			
209	D1	□	□			
210	D2	□	□			
211	D3	□	□			

<i>CHR\$ Number</i>	<i>Display (Quote Mode)</i>	<i>Upper Case/ Graphic</i>	<i>Upper/lower Case</i>	<i>Function</i>	<i>Keys to Press</i>
Dec	Hex				
212	D4				
213	D5				
214	D6				
215	D7				
216	D8				
217	D9				
218	DA				
219	DB				
220	DC				
221	DD				
222	DE				
223	DF				
224	E0				
225	E1				
226	E2				
227	E3				
228	E4				
229	E5				
230	E6				
231	E7				
232	E8				
233	E9				
234	EA				
235	EB				
236	EC				
237	ED				
238	EE				
239	EF				
240	F0				
241	F1				
242	F2				
243	F3				
244	F4				
245	F5				
246	F6				
247	F7				
248	F8				
249	F9				
250	FA				

CHR\$ Number Dec	Hex	Display (Quote Mode)			Function	Keys to Press
		Upper Case/ Graphic	Upper/lower Case			
251	FB					
252	FC					
253	FD					
254	FE					
255	FF					

\*To use CHR\$ codes for the function keys, you must first use the KEY command to define a key with the appropriate CHR\$ code number.

---

# Appendix D

## ASCII Codes

The American Standard Code for Information Interchange is a frequently used standard for data communications. This table is presented to allow Plus/4 programmers to translate Commodore character (CHR\$) codes to ASCII.

<i>Dec</i>	<i>Hex</i>	<i>Meaning</i>	<i>Dec</i>	<i>Hex</i>	<i>Meaning</i>
0	\$00	NUL — Null	64	\$40	@
1	\$01	SOH — Start of heading	65	\$41	A
2	\$02	STX — Start of text	66	\$42	B
3	\$03	ETX — End of text	67	\$43	C
4	\$04	EOT — End of transmission	68	\$44	D
5	\$05	ENQ — Enquiry	69	\$45	E
6	\$06	ACK — Acknowledge	70	\$46	F
7	\$07	BEL — Bell	71	\$47	G
8	\$08	BS — Backspace	72	\$48	H
9	\$09	HT — Horizontal tab	73	\$49	I
10	\$0A	LF — Line feed	74	\$4A	J
11	\$0B	VT — Vertical tab	75	\$4B	K
12	\$0C	FF — Form feed	76	\$4C	L
13	\$0D	CR — Carriage return	77	\$4D	M
14	\$0E	SO — Shift out	78	\$4E	N
15	\$0F	SI — Shift in	79	\$4F	O
16	\$10	DLE — Data link escape	80	\$50	P
17	\$11	DC1 — Device control 1	81	\$51	Q
18	\$12	DC2 — Device control 2	82	\$52	R
19	\$13	DC3 — Device control 3	83	\$53	S
20	\$14	DC4 — Device control 4	84	\$54	T
21	\$15	NAK — Negative acknowledge	85	\$55	U
22	\$16	SYN — Synchronous idle	86	\$56	V
23	\$17	ETB — End of transmission block	87	\$57	W

<i>Dec</i>	<i>Hex</i>	<i>Meaning</i>	<i>Dec</i>	<i>Hex</i>	<i>Meaning</i>
24	\$18	CAN — Cancel	88	\$58	X
25	\$19	EM — End of medium	89	\$59	Y
26	\$1A	SUB — Substitute	90	\$5A	Z
27	\$1B	ESC — Escape	91	\$5B	[
28	\$1C	FS — File separator	92	\$5C	\
29	\$1D	GS — Group separator	93	\$5D	]
30	\$1E	RS — Record separator	94	\$5E	↑
31	\$1F	US — Unit separator	95	\$5F	←
32	\$20	SP — Space	96	\$60	blank
33	\$21	!	97	\$61	a
34	\$22	"	98	\$62	b
35	\$23	#	99	\$63	c
36	\$24	\$	100	\$64	d
37	\$25	%	101	\$65	e
38	\$26	&	102	\$66	f
39	\$27	'	103	\$67	g
40	\$28	(	104	\$68	h
41	\$29	)	105	\$69	i
42	\$2A	*	106	\$6A	j
43	\$2B	+	107	\$6B	k
44	\$2C	,	108	\$6C	l
45	\$2D	-	109	\$6D	m
46	\$2E	.	110	\$6E	n
47	\$2F	/	111	\$6F	o
48	\$30	0	112	\$70	p
49	\$31	1	113	\$71	q
50	\$32	2	114	\$72	r
51	\$33	3	115	\$73	s
52	\$34	4	116	\$74	t
53	\$35	5	117	\$75	u
54	\$36	6	118	\$76	v
55	\$37	7	119	\$77	w
56	\$38	8	120	\$78	x
57	\$39	9	121	\$79	y
58	\$3A	:	122	\$7A	z
59	\$3B	;	123	\$7B	{
60	\$3C	<	124	\$7C	
61	\$3D	=	125	\$7D	}
62	\$3E	>	126	\$7E	~
63	\$3F	?	127	\$7F	DEL — Delete

---

# Appendix E

## Screen Display Codes

The current contents of the screen display are stored in an area of RAM called screen memory. The values stored in screen memory to represent characters are different from the CHR\$ codes.

Number		Upper Case/ Graphic Mode		Upper/lower Case Mode		Number		Upper Case/ Graphic Mode		Upper/lower Case Mode	
Dec	Hex					Dec	Hex				
0	00	█	█	█	█	22	16	█	█	█	█
1	01	█	█	█	█	23	17	█	█	█	█
2	02	█	█	█	█	24	18	█	█	█	█
3	03	█	█	█	█	25	19	█	█	█	█
4	04	█	█	█	█	26	1A	█	█	█	█
5	05	█	█	█	█	27	1B	█	█	█	█
6	06	█	█	█	█	28	1C	█	█	█	█
7	07	█	█	█	█	29	1D	█	█	█	█
8	08	█	█	█	█	30	1E	█	█	█	█
9	09	█	█	█	█	31	1F	█	█	█	█
10	0A	█	█	█	█	32	20	█	█	█	█
11	0B	█	█	█	█	33	21	█	█	█	█
12	0C	█	█	█	█	34	22	█	█	█	█
13	0D	█	█	█	█	35	23	█	█	█	█
14	0E	█	█	█	█	36	24	█	█	█	█
15	0F	█	█	█	█	37	25	█	█	█	█
16	10	█	█	█	█	38	26	█	█	█	█
17	11	█	█	█	█	39	27	█	█	█	█
18	12	█	█	█	█	40	28	█	█	█	█
19	13	█	█	█	█	41	29	█	█	█	█
20	14	█	█	█	█	42	2A	█	█	█	█
21	15	█	█	█	█	43	2B	█	█	█	█

Upper Case/ Graphic Mode			Upper Case/ Graphic Mode		
Number	Dec	Hex	Number	Dec	Hex
44	2C	█	83	53	█
45	2D	—	84	54	█
46	2E	·	85	55	█
47	2F	▀	86	56	█
48	30	█	87	57	█
49	31	█	88	58	█
50	32	█	89	59	█
51	33	█	90	5A	█
52	34	█	91	5B	█
53	35	█	92	5C	█
54	36	█	93	5D	█
55	37	█	94	5E	█
56	38	█	95	5F	█
57	39	█	96	60	█
58	3A	:	97	61	█
59	3B	:	98	62	█
60	3C	█	99	63	█
61	3D	█	100	64	█
62	3E	█	101	65	█
63	3F	█	102	66	█
64	40	█	103	67	█
65	41	█	104	68	█
66	42	█	105	69	█
67	43	█	106	6A	█
68	44	█	107	6B	█
69	45	█	108	6C	█
70	46	█	109	6D	█
71	47	█	110	6E	█
72	48	█	111	6F	█
73	49	█	112	70	█
74	4A	█	113	71	█
75	4B	█	114	72	█
76	4C	█	115	73	█
77	4D	█	116	74	█
78	4E	█	117	75	█
79	4F	█	118	76	█
80	50	█	119	77	█
81	51	█	120	78	█
82	52	█	121	79	█

Number		<i>Upper Case/ Graphic Mode</i>		Number		<i>Upper Case/ Graphic Mode</i>	
Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex
122	7A			125	7D		
123	7B			126	7E		
124	7C			127	7F		

Codes 128–255 display the reversed images of codes 0–127. To find the number to POKE to display a reversed symbol, use the chart to find the number for the nonreversed symbol, and then add 128 to that number.

---

# Appendix F

## Note Tables

The decimal values used in the SOUND command to get five octaves of musical notes are displayed in this table.

Note	Octave 1 Frequency	Octave 2 Frequency	Octave 3 Frequency	Octave 4 Frequency	Octave 5 Frequency
A	7	516	770	897	960
A#	64	544	784	904	964
B	118	571	798	911	967
C	169	596*	810	917	971
C#	217	620	822	923	974
D	262	643	834	929	976
D#	305	664	844	934	979
E	345	685	854	939	982
F	383	704	864	944	984
F#	419	722	873	948	986
G	453	739	881	953	988
G#	485	755	889	957	990

\*The setting for middle C.

*Note:* Use the following formula to calculate a note value for some other output frequency (FO):

$$\text{Frequency} = 1024 - \text{INT}(111860.781/\text{FO})$$

The hexadecimal values for sound and music programming in machine language are displayed in the table on page 422.

<i>Note</i>	<i>Octave 1 Frequency</i>	<i>Octave 2 Frequency</i>	<i>Octave 3 Frequency</i>	<i>Octave 4 Frequency</i>	<i>Octave 5 Frequency</i>
A	7	204	302	381	3C0
A#	40	220	310	388	3C4
B	76	23B	31E	38F	3C7
C	A9	254	32A	395	3CB
C#	D9	26C	336	39B	3CE
D	106	283	342	3A1	3D0
D#	131	298	346	3A6	3D3
E	159	2AD	356	3AB	3D6
F	17F	2C0	360	3B0	3D8
F#	1A3	2D2	369	3B4	3DA
G	1C5	2E3	371	3B9	3DC
G#	1E5	2F3	379	3BD	3DE

---

# Appendix G

## Plus/4 Memory Maps

Three different maps are presented. The first describes each of the registers of the graphics chip. The second gives an overview of the Plus/4's memory usage. The third gives as much detail as possible about each location in the Plus/4.

### Graphics Chip Register Map

The graphics chip is located at \$FF00–\$FF1F (65280- 65311). The usage of each register is described as completely as possible.

Hexadecimal	Decimal	Bits	Function
\$FF00	65280	0–7	Low byte of reload value for timer 1
\$FF01	65281	0–7	High byte of reload value for timer 1
\$FF02	65282	0–7	Low byte of start value for timer 2
\$FF03	65283	0–7	High byte of start value for timer 2
\$FF04	65284	0–7	Low byte of start value for timer 3
\$FF05	65285	0–7	High byte of start value for timer 3
\$FF06	65286	0–2	Vertical screen scroll position 3      0 = 24 rows, 1 = 25 rows 4      0 = blank screen, 1 = display screen 5      Bit map mode: 0 = off, 1 = on 6      Extended color mode: 0 = off, 1 = on 7      TEST (should always be cleared to 0)
\$FF07	65287	0–2	Horizontal screen scroll position 3      0 = 38 columns, 1 = 40 columns 4      Multicolor mode: 0 = off, 1 = on 5      Flashing: 0 = yes, 1 = no 6      TV standard: 0 = PAL, 1 = NTSC 7      Reverse characters through hardware: 0 = yes, 1 = no

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Bits</i>	<i>Function</i>
\$FF08	65288	0-7	Latch register for keyboard
\$FF09	65289	0	Not connected
		1	Raster interrupt flag
		2	Light pen interrupt flag
		3	Timer 1 interrupt flag
		4	Timer 2 interrupt flag
		5	Not connected
		6	Timer 3 interrupt flag
		7	Interrupt occurred flag
\$FF0A	65290	0	High bit for raster interrupt value
		1	Raster interrupt enable
		2	Light pen interrupt enable
		3	Timer 1 interrupt enable
		4	Timer 2 interrupt enable
		5	Not connected
		6	Timer 3 interrupt enable
		7	Not connected
\$FF0B	65291	0-7	Low byte for raster interrupt value
\$FF0C	65292	0-1	High bits for hardware cursor position
		2-7	Not connected
\$FF0D	65293	0-7	Low byte for hardware cursor position
\$FF0E	65294	0-7	Low byte of frequency for voice 1
\$FF0F	65295	0-7	Low byte of frequency for voice 2
\$FF10	65296	0-1	High bits of frequency for voice 2
		2-7	Not connected
\$FF11	65297	0-3	Volume: 0 = off, F = highest
		4	Select voice 1
		5	Select tone generator for voice 2
		6	Select noise generator for voice 2
		7	Sound reload switch: 0 = on , 1 = off
\$FF12	65298	0-1	High bits of frequency for voice 1
		2	0 = get data from RAM, 1 = get data from ROM
		3-5	Base address for bit map
		6-7	Not connected
\$FF13	65299	0	Status of clock
		1	Single clock set
		2-7	Base address for character data
\$FF14	65300	0-2	Not connected
		3-7	Base address for color and screen memory

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Bits</i>	<i>Function</i>
\$FF15	65301	0–3	Background color
		4–6	Background luminance
		7	Not connected
\$FF16	65302	0–3	Color for color 1
		4–6	Luminance for color 1
		7	Not connected
\$FF17	65303	0–3	Color for color 2
		4–6	Luminance for color 2
		7	Not connected
\$FF18	65304	0–3	Color for color 3
		4–6	Luminance for color 3
		7	Not connected
\$FF19	65305	0–3	Border color
		4–6	Border luminance
		7	Not connected
\$FF1A	65306	0–1	High bits for bit map reload
		2–7	Not connected
\$FF1B	65307	0–7	Low byte for bit map reload
\$FF1C	65308	0	High bit of current vertical raster position
		1–7	Not connected
		0–7	Low byte of current vertical raster position
\$FF1E	65310	0–7	Current horizontal raster position
\$FF1F	65311	0–2	Vertical sub address
		3–6	Blink address
		7	Not connected

## Memory Usage on the Plus/4

### MEMORY RANGE

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$0000–\$0001	0–1	Processor on-chip data registers
\$0002–\$00CF	2–207	System zero page storage
\$00D0–\$00D7	208–215	Zero page area reserved for speech software
\$00D8–\$00E8	216–232	Free zero page area for applications
\$00E9–\$00FF	233–255	System zero page storage
\$0100–\$0122	256–290	System storage
\$0123–\$01FF	291–511	Processor stack

## MEMORY RANGE

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$0200-\$0258	512-600	Input buffer for BASIC and the monitor
\$0259-\$025C	601-604	BASIC storage
\$025D-\$02AC	605-684	Storage for DOS information
\$02AD-\$02CB	685-715	BASIC graphics storage
\$02CC-\$02E3	716-739	BASIC work area
\$02E4-\$02F1	740-753	BASIC graphics storage
\$02F2-\$02F5	754-757	BASIC pointers
\$02F6-\$02FD	758-765	Free
\$02FE-\$0331	766-817	System vectors
\$0332-\$03F2	818-1010	Cassette buffer
\$03F3-\$03F6	1011-1014	Storage for cassette information
\$03F7-\$0436	1015-1078	RS232 buffer
\$0437-\$0472	1079-1138	Storage for cassette information
\$0473-\$04E6	1139-1254	BASIC RAM subroutines
\$04E7-\$0508	1255-1288	BASIC storage
\$0509-\$054A	1289-1354	System storage
\$054B-\$055C	1355-1372	Monitor storage
\$055D-\$05E6	1373-1510	Function key area
\$05E7-\$05EB	1511-1515	Storage for DMA information
\$05EC-\$05EF	1516-1519	Cartridge address table
\$05F0-\$05F4	1520-1524	Long jump routine and storage
\$05F5-\$06EB	1525-1771	RAM for speech and cartridges
\$06EC-\$07AF	1772-1967	BASIC stack
\$07B0-\$07F1	1968-2033	System I/O storage
\$07F2-\$07FF	2034-2047	System storage
\$0800-\$0BE7	2048-3047	Color memory
\$0BE8-\$0BFF	3048-3071	Free
\$0C00-\$0FE7	3072-4071	Screen memory
\$0FE8-\$0FFF	4072-4095	Free
\$1000-\$3FFF	4096-16383	RAM used by BASIC (in text mode)
\$1000-\$17FF	4096-6143	Free (in BASIC graphics mode)
\$1800-\$1BE7	6144-7143	Luminance memory (in BASIC graphics mode)
\$1BE8-\$1BFF	7144-7167	Free (in BASIC graphics mode)
\$1C00-\$1FE7	7168-8167	Color memory (in BASIC graphics mode)
\$1FE8-\$1FFF	8168-8191	Free (in BASIC graphics mode)
\$2000-\$3F3F	8192-16191	Graphics screen (in BASIC graphics mode)
\$3F40-\$3FFF	16192-16383	Free (in BASIC graphics mode)
\$4000-\$7FFF	16384-32767	RAM used by BASIC

## MEMORY RANGE

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$8000-\$FCFF	32768-64767	RAM used by BASIC (under ROM)
\$8000-\$CDFF	32768-52735	BASIC and monitor ROM
\$CE00-\$CFFF	52735-53247	Operating system ROM
\$D000-\$D7FF	53248-55295	Character ROM
\$D800-\$FBFF	55296-64511	Operating system ROM
\$FC00-\$FCFF	64512-64767	Banking routines ROM (in all ROM maps)
\$FD00-\$FD0F	64768-64783	ACIA chip (used for RS232, in all maps)
\$FD10-\$FD1F	64784-64799	Parallel port (6529, in all maps)
\$FD20-\$FDCC	64800-64975	Unknown (in all maps)
\$FDD0-\$FDDF	64976-64991	Cartridge banking port (in all maps)
\$FDE0-\$FEFF	64992-65279	Direct Memory Access disk (in all maps)
\$FF00-\$FF3F	65280-65343	Graphics chip (in all maps)
\$FF40-\$FFFF	65344-65535	RAM (under ROM)
\$FF40-\$FFFF	65344-65535	Operating system ROM

## Detailed Memory Map of the Plus/4

## MEMORY LOCATION(S)

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$0000	0	Processor on-chip data direction register
\$0001	1	Processor on-chip data register
\$0002	2	BASIC token to search for in BASIC stack
\$0003-\$0006	3-6	Storage for RENUMBER
\$0007	7	Start character to search for in BASIC text
\$0008	8	End character to search for in BASIC text
\$0009	9	Save last TAB column
\$000A	10	Flag for load (\$00) or verify (\$01)
\$000B	11	Buffer pointer for input/number of subscripts
\$000C	12	Flag for default array dimension
\$000D	13	Flag for data type (\$00=numeric, \$FF=string)
\$000E	14	Flag for data type (\$00=floating, \$80=integer)
\$000F	15	Flag for garbage collect/DATA scan/LIST quote

## MEMORY LOCATION(S)

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$0010	16	Flag for subscript/FNx function
\$0011	17	Flag for input (\$00=INPUT, \$40=GET, \$98=READ)
\$0012	18	Sign of TAN/comparison flag
\$0013	19	Flag for I/O prompt
\$0014-\$0015	20-21	Integer value
\$0016	22	Pointer to temporary string stack
\$0017-\$0018	23-24	Address of last temporary string
\$0019-\$0021	25-33	Stack for temporary strings
\$0022-\$0025	34-37	Utility pointers
\$0026-\$002A	38-42	Floating point result of multiply
\$002B-\$002C	43-44	Pointer to start of BASIC text
\$002D-\$002E	45-46	Pointer to start of variables in BASIC RAM
\$002F-\$0030	47-48	Pointer to start of arrays in BASIC RAM
\$0031-\$0032	49-50	Pointer to end of arrays in BASIC RAM (+1)
\$0033-\$0034	51-52	Pointer to bottom of strings in BASIC RAM
\$0035-\$0036	53-54	Pointer to current string
\$0037-\$0038	55-56	Pointer to top of BASIC RAM (+1)
\$0039-\$003A	57-58	Current BASIC line number
\$003B-\$003C	59-60	Pointer to current BASIC text
\$003D-\$003E	61-62	Pointer to BASIC stack for CONT
\$003F-\$0040	63-64	Current line number in DATA
\$0041-\$0042	65-66	Current address of DATA item
\$0043-\$0044	67-68	Vector to INPUT
\$0045-\$0046	69-70	Name of current variable
\$0047-\$0048	71-72	Address of current variable
\$0049-\$004A	73-74	Pointer to FOR/NEXT index
\$004B-\$0060	75-96	Temporary storage area
\$0061	97	Floating point accumulator 1, exponent
\$0062-\$0065	98-101	Floating point accumulator 1, mantissa
\$0066	102	Floating point accumulator 1, sign
\$0067	103	Series evaluation constant pointer
\$0068	104	Floating point accumulator 1, overflow
\$0069	105	Floating point accumulator 2, exponent
\$006A-\$006D	106-109	Floating point accumulator 2, mantissa
\$006E	110	Floating point accumulator 2, sign

## MEMORY LOCATION(S)

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$006F	111	Sign comparison of FPA 1 and FPA 2/pointer
\$0070	112	Floating point accumulator 2, overflow
\$0071-\$0072	113-114	Cassette buffer/series pointer
\$0073-\$0074	115-116	Increment for AUTO (\$00=none)
\$0075	117	Flag for graphics area (\$00=no, \$FF=yes)
\$0076-\$0077	118-119	Key work area
\$0078	120	Temporary storage for indirect load
\$0079-\$007B	121-123	Disk error message descriptor
\$007C-\$007D	124-125	BASIC stack pointer
\$007E-\$007F	126-127	Temporary storage for sounds
\$0080	128	Temporary parameter storage
\$0081	129	Flag for RUNning (\$00=no, \$80=yes)
\$0082	130	Flag used for DOS commands
\$0083	131	Graphic mode (\$00=text, \$20=high-res, \$60=split high-res, \$A0=multicolor, \$E0=split multicolor)
\$0084	132	Current color source for drawing
\$0085	133	Current color/luminance for COLOR 2
\$0086	134	Current color/luminance for COLOR 1
\$0087	135	Maximum number of screen columns
\$0088	136	Maximum number of screen rows
\$0089	137	Flag for PAINT (left)
\$008A	138	Flag for PAINT (right)
\$008B	139	Flag for PAINT border (\$00=same color, \$80=nonbackground color)
\$008C-\$008D	140-141	Pointer to bit map color information
\$008E	142	Temporary storage
\$008F	143	Temporary storage
\$0090	144	I/O status word (ST)
\$0091	145	STOP key flag
\$0092	146	Temporary storage
\$0093	147	Flag for load (\$00) or verify (\$01)
\$0094	148	Flag for buffered serial output (\$00=no, \$80=yes)
\$0095	149	Buffered serial output byte
\$0096	150	Save .X for BASIN
\$0097	151	Count of open logical files
\$0098	152	Current input device

## MEMORY LOCATION(S)

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$0099	153	Current output device
\$009A	154	Flag for message output (\$00=program, \$80=BASIC direct mode, \$C0=monitor)
\$009B-\$009C	155-156	Current SAVE address
\$009D-\$009E	157-158	End SAVE address (+1)/LOAD address
\$009F-\$00A0	159-160	Temporary storage
\$00A1-\$00A2	161-162	Vector for monitor
\$00A3-\$00A5	163-165	System clock
\$00A6	166	Part of serial bus EOI count
\$00A7	167	Buffered tape I/O byte
\$00A8	168	Buffered serial input byte
\$00A9-\$00AA	169-170	Color vector for scrolling/temporary storage
\$00AB	171	Current file name length
\$00AC	172	Number of current logical file
\$00AD	173	Secondary address of current logical file
\$00AE	174	Device number of current logical file
\$00AF-\$00B0	175-176	Pointer to name of current logical file
\$00B1	177	Count of tape errors
\$00B2-\$00B3	178-179	Begin SAVE address
\$00B4-\$00B5	180-181	Relocated LOAD address
\$00B6-\$00B7	182-183	Pointer for cassette buffer
\$00B8-\$00B9	184-185	Address for VECTOR
\$00BA-\$00BB	186-187	Temporary storage for cassette I/O
\$00BC-\$00BD	188-189	Pointer for tape messages
\$00BE-\$00BF	190-191	Pointer to fetch byte for ROM banking
\$00C0-\$00C1	192-193	Text vector for scrolling/temporary storage
\$00C2	194	Flag for reversed text (\$00=off, \$12=on)
\$00C3	195	Last cursor column for input
\$00C4	196	Temporary cursor line pointer
\$00C5	197	Temporary cursor column pointer
\$00C6	198	Flag for shift/control key input
\$00C7	199	Flag for input from keyboard (\$00) or screen (\$03)
\$00C8-\$00C9	200-201	Address of current screen line
\$00CA	202	Current cursor column
\$00CB	203	Flag for quote mode (\$00=no, \$01=yes)
\$00CC	204	Temporary storage for editor

## MEMORY LOCATION(S)

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$00CD	205	Current cursor line
\$00CE	206	Last character input
\$00CF	207	Counter for insert mode characters
\$00D0-\$00D7	208-215	Zero page area reserved for speech software
\$00D8-\$00E8	216-232	Free zero page area for applications
\$00E9	233	Segment size for CIRCLE
\$00EA-\$00EB	234-235	Address of current color line
\$00EC-\$00ED	236-237	Indirect for key scan table
\$00EE	238	Temporary storage for key scan
\$00EF	239	Keyboard queue index
\$00F0	240	Flag for screen I/O pause (CTRL S/T)
\$00F1-\$00F4	241-244	Zero page area for monitor
\$00F5	245	Cassette I/O checksum
\$00F6	246	Zero page location for monitor
\$00F7	247	Pass number for cassette I/O
\$00F8	248	Type of cassette block
\$00F9	249	Flag for DMA disk (\$00=not present, \$80=present)
\$00FA	250	Temporary storage for .X during stop key check
\$00FB	251	Number of ROM bank currently enabled
\$00FC	252	X-on character for RS232 I/O
\$00FD	253	X-off character for RS232 I/O
\$00FE	254	Temporary storage for cursor line in editor
\$00FF	255	Used as base address for indexing forward
\$0100-\$010F	256-271	Floating point operation buffer
\$0110-\$0112	272-274	Temporary storage for .A, .X, .Y during I/O
\$0113 \$0122	275 290	RAM color/luminance table for color keys
\$0123-\$01FF	291-511	Processor stack
\$0200-\$0258	512-600	Input buffer
\$0259-\$025A	601-602	Last BASIC line number
\$025B-\$025C	603-604	Last BASIC text pointer
\$025D	605	Loop counter for DOS
\$025E-\$026D	606-621	Buffer for disk file name
\$026E	622	Length of disk file name 1
\$026F	623	Drive for disk file 1
\$0270-\$0271	624-625	Address of disk file name 1

## MEMORY LOCATION(S)

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$0272	626	Length of disk file name 2
\$0273	627	Drive for disk file 2
\$0274-\$0275	628-629	Address of disk file name 2
\$0276	630	Logical address for DOS
\$0277	631	Physical address for DOS
\$0278	632	Secondary address for DOS
\$0279-\$027A	633-634	Disk identifier
\$027B	635	Flag for disk id specified
\$027C	636	Buffer for disk output string
\$027D-\$02AC	637-684	Disk output string area
\$02AD-\$02AE	685-686	Graphics current <i>x</i> -coordinate
\$02AF-\$02B0	687-688	Graphics current <i>y</i> -coordinate
\$02B1-\$02B2	689-690	Final <i>x</i> -coordinate
\$02B3-\$02B4	691-692	Final <i>y</i> -coordinate
\$02B5-\$02C4	693-708	Graphics calculation area
\$02C5	709	Current angle's sign
\$02C6-\$02C7	710-711	Current angle's sine
\$02C8-\$02C9	712-713	Current angle's cosine
\$02CA-\$02CB	714-715	Temporary storage for distance routines
\$02CC-\$02E3	716-739	PRINT USING/CIRCLE/SHAPE work area
\$02E4	740	CHAR command character ROM address
\$02E5	741	Temporary storage for GSHAPE
\$02E6	742	Flag for SCALE (\$00=off, \$01=on)
\$02E7	743	Flag for double width
\$02E8	744	Flag for BOX fill
\$02E9	745	Temporary storage of bit mask
\$02EA	746	Length of string
\$02EB	747	Flag for TRON (\$00=off, \$FF=on)
\$02EC-\$02EE	748-750	Temporary storage for DIRECTORY
\$02EF	751	Temporary storage for graphics
\$02F0	752	Count of graphics parameters
\$02F1	753	Flag for relative or absolute
\$02F2-\$02F3	754-755	Vector for floating point to integer conversion
\$02F4-\$02F5	756-757	Vector for integer to floating point conversion
\$02F6-\$02FD	758-765	Free
\$02FE-\$02FF	766-767	Vector for cartridge software startup

## MEMORY LOCATION(S)

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$0300-\$0301	768-769	Vector for error routine
\$0302-\$0303	770-771	Vector for BASIC warm start
\$0304-\$0305	772-773	Vector for tokenization routine
\$0306-\$0307	774-775	Vector for token PRINT
\$0308-\$0309	776-777	Vector to execute BASIC code
\$030A-\$030B	778-779	Vector for arithmetic symbol evaluation
\$030C-\$030D	780-781	Vector to escape token crunch routine
\$030E-\$030F	782-783	Vector to escape token PRINT routine
\$0310-\$0311	784-785	Vector to escape execute routine
\$0312-\$0313	786-787	Vector for vertical blank IRQ routine
\$0314-\$0315	788-789	Vector for IRQ routine
\$0316-\$0317	790-791	Vector for BRK instruction processing
\$0318-\$0319	792-793	Vector for OPEN
\$031A-\$031B	794-795	Vector for CLOSE
\$031C-\$031D	796-797	Vector for CHKIN
\$031E-\$031F	798-799	Vector for CHOUT
\$0320-\$0321	800-801	Vector for CLRCH
\$0322-\$0323	802-803	Vector for BASIN
\$0324-\$0325	804-805	Vector for BSOUT
\$0326-\$0327	806-807	Vector for STOP
\$0328-\$0329	808-809	Vector for GETIN
\$032A-\$032B	810-811	Vector for CLALL
\$032C-\$032D	812-813	Free vector
\$032E-\$032F	814-815	Vector for LOADSP
\$0330-\$0331	816-817	Vector for SAVESP
\$0332-\$03F2	818-1010	Buffer for cassette I/O
\$03F3-\$03F4	1011-1012	Number of characters to write to tape
\$03F5-\$03F6	1013-1014	Number of characters to read from tape
\$03F7-\$0436	1015-1078	RS232 buffer
\$0437-\$0454	1079-1108	Cassette I/O error stack (low bytes)
\$0455-\$0472	1109-1138	Cassette I/O error stack (high bytes)
\$0473-\$0478	1139-1144	Entry to get next character from BASIC text RAM
\$0479-\$0484	1145-1156	Entry to get same character from BASIC text RAM
\$0485-\$0493	1157-1171	Entry to check for numeric input
\$0494-\$04A1	1172-1185	Self-modifying routine to fetch RAM data via (.A), Y
\$04A2-\$04A4	1186-1188	Zeroes

## MEMORY LOCATION(S)

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$04A5-\$04AF	1189-1199	Fetch RAM via BASIC text pointer (\$3B),Y
\$04B0-\$04BA	1200-1210	Fetch RAM from (\$22),Y
\$04BB-\$04C5	1211-1221	Fetch RAM from (\$24),Y
\$04C6-\$04D0	1222-1232	Fetch RAM from (\$6F),Y
\$04D1-\$04DB	1233-1243	Fetch RAM from (\$5F),Y
\$04DC-\$04E6	1244-1254	Fetch RAM from (\$64),Y
\$04E7	1255	PRINT USING symbol for blank
\$04E8	1256	PRINT USING symbol for comma
\$04E9	1257	PRINT USING symbol for decimal point
\$04EA	1258	PRINT USING symbol for dollar sign
\$04EB-\$04EE	1259-1262	Temporary storage for string functions
\$04EF	1263	Number of latest error
\$04F0-\$04F1	1264-1265	Line number of latest error
\$04F2-\$04F3	1266-1267	TRAP line
\$04F4	1268	Temporary storage for TRAP
\$04F5-\$04F6	1269-1270	BASIC text pointer at latest error
\$04F7	1271	BASIC stack pointer at latest error
\$04F8-\$04F9	1272-1273	DO storage of BASIC text pointer
\$04FA-\$04FB	1274-1275	DO storage of line number
\$04FC	1276	Low byte of sound 1 duration
\$04FD	1277	Low byte of sound 2 duration
\$04FE	1278	High byte of sound 1 duration
\$04FF	1279	High byte of sound 2 duration
\$0500-\$0502	1280-1282	USR function jump instruction
\$0503-\$0507	1283-1287	Random number registers
\$0508	1288	Checked for cold/warm start (\$A5=warm)
\$0509-\$0512	1289-1298	Table of OPEN logical file numbers
\$0513-\$051C	1299-1308	Table of OPEN device numbers
\$051D-\$0526	1309-1318	Table of OPEN secondary addresses
\$0527-\$0530	1319-1328	Keyboard queue
\$0531-\$0532	1329-1330	Bottom of memory
\$0533-\$0534	1331-1332	Top of memory
\$0535	1333	Timeout flag (DMA)
\$0536	1334	EOF flag (DMA)
\$0537	1335	Number of bytes in buffer
\$0538	1336	Valid bytes in buffer
\$0539	1337	Pointer to buffer
\$053A	1338	Type of current tape file

## MEMORY LOCATION(S)

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$053B	1339	Current character color/luminance
\$053C	1340	Current character flash flag (\$00=no, \$80=yes)
\$053D	1341	Free
\$053E	1342	Screen memory start (high byte)
\$053F	1343	Length of keyboard queue
\$0540	1344	Flag for key repeat (\$80=all keys repeat, \$40=no keys repeat, \$00=space,INST/DEL, and cursor keys repeat)
\$0541-\$0542	1345-1346	Counters for key repeats
\$0543	1347	Flag for shift key
\$0544	1348	Pattern of last shift
\$0545-\$0546	1349-1350	Vector for keyboard table
\$0547	1351	Commodore SHIFT enable (\$00=enabled, \$80=disabled)
\$0548	1352	Flag for scrolling (not used)
\$0549-\$054A	1353-1354	Temporary storage during screen output
\$054B-\$0551	1355-1361	Storage for monitor
\$0552-\$0558	1362-1368	Register storage for monitor (PC,SR,A,X,Y,SP)
\$0559-\$055C	1369-1372	Storage for monitor
\$055D	1373	Number of characters left to get from function key definition
\$055E	1374	Pointer to current character in function key definition
\$055F-\$0566	1375-1382	Table of lengths of function key definitions
\$0567-\$05E6	1383-1510	Function key definitions
\$05E7	1511	Temporary storage for data write (DMA)
\$05E8	1512	Read or write (DMA)
\$05E9	1513	Device number (DMA)
\$05EA	1514	Presence flag (DMA)
\$05EB	1515	Temporary storage for open type (DMA)
\$05EC-\$05EF	1516-1519	Table of physical addresses for cartridge ROMs
\$05F0-\$05F1	1520-1521	Long jump for banking routines
\$05F2	1522	.A for long jump
\$05F3	1523	.X for long jump
\$05F4	1524	.Y for long jump

## MEMORY LOCATION(S)

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$05F5-\$065D	1525–1629	Banking RAM area
\$065E-\$06EB	1630–1771	RAM reserved for speech software
\$06EC-\$07AF	1772–1967	BASIC stack
\$07B0	1968	Tape byte to write
\$07B1	1969	Temporary parity calculation byte
\$07B2-\$07B3	1970–1971	Temporary storage for tape header write
\$07B4	1972	Free
\$07B5	1973	Temporary index for reading bytes
\$07B6	1974	Pointer to error stack
\$07B7	1975	Count of errors on initial pass
\$07B8-\$07BD	1976–1981	Constants for timing
\$07BE	1982	Pointer to stack for STOP recover
\$07BF	1983	Pointer to stack for drop key recover
\$07C0-\$07C3	1984–1987	Read block parameters
\$07C4	1988	Temporary status for read block
\$07C5	1989	Count of leader shorts to find
\$07C6	1990	Count of read errors fatal
\$07C7	1991	Temporary storage for VERIFY
\$07C8-\$07CC	1992–1996	Temporary storage for tape I/O
\$07CD	1997	RS232 data character to send buffer
\$07CE	1998	RS232 flag for data character to send (\$00=no, \$80=yes)
\$07CF	1999	RS232 control character to send buffer
\$07D0	2000	RS232 flag for control character to send (\$00=no, \$80=yes)
\$07D1	2001	RS232 pointer to current start of input queue
\$07D2	2002	RS232 pointer to current end of input queue
\$07D3	2003	RS232 current count of input queue
\$07D4	2004	RS232 status
\$07D5	2005	RS232 temporary storage of character input
\$07D6	2006	RS232 flag for local hold off
\$07D7	2007	RS232 flag for remote hold off
\$07D8	2008	RS232 flag for presence of ACIA
\$07D9-\$07E4	2009–2020	Indirect indexed RAM fetch routine via contents of \$07DF
\$07E5	2021	Bottom screen line of current window

## MEMORY LOCATION(S)

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$07E6	2022	Top screen line of current window
\$07E7	2023	Left screen column of current window
\$07E8	2024	Right screen column of current window
\$07E9	2025	Flag to disable scrolling (\$00=scroll, \$80=don't)
\$07EA	2026	Auto insert flag (\$00=off, \$FF=on)
\$07EB	2027	Latest character printed
\$07EC	2028	Storage for screen line management
\$07ED	2029	Color under cursor
\$07EE-\$07F1	2030–2033	Line link table for screen
\$07F2	2034	.A to send to SYS
\$07F3	2035	.X to send to SYS
\$07F4	2036	.Y to send to SYS
\$07F5	2037	Status register to send to SYS
\$07F6	2038	Index for key scan
\$07F7	2039	Flag for control S/T (\$00=enabled, else disabled)
\$07F8	2040	Bank for monitor fetches (\$00=ROM, \$80=RAM)
\$07F9	2041	Bank for color key color/luminance value table (\$00=RAM (\$0113–\$0122), \$80=ROM)
\$07FA	2042	Bit map mask for split screen
\$07FB	2043	Screen memory mask for split screen
\$07FC	2044	Cassette motor lock signal
\$07FD	2045	Time-of-day for PAL (not used)
\$07FE–\$07FF	2046–2047	Free
\$0800–\$0BE7	2048–3047	Color memory
\$0BE8–\$0BFF	3048–3071	Free
\$0C00–\$0FE7	3072–4071	Screen memory
\$0FE8–\$0FFF	4072–4095	Free
\$1000–\$3FFF	4096–16383	RAM used by BASIC (in text mode)
\$1000–\$17FF	4096–6143	Free (in BASIC graphics mode)
\$1800–\$1BE7	6144–7143	Luminance memory (in BASIC graphics mode)
\$1BE8–\$1BFF	7144–7167	Free (in BASIC graphics mode)
\$1C00–\$1FE7	7168–8167	Color memory (in BASIC graphics mode)
\$1FE8–\$1FFF	8168–8191	Free (in BASIC graphics mode)
\$2000–\$3F3F	8192–16191	Graphics screen (in BASIC graphics mode)

## MEMORY LOCATION(S)

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$3F40-\$3FFF	16192-16383	Free (in BASIC graphics mode)
\$4000-\$7FFF	16384-32767	RAM used by BASIC
\$8000-\$FCFF	32768-64767	RAM used by BASIC (under ROM)
\$8000-\$CDFF	32768-52735	BASIC and monitor ROM
\$CE00-\$CFFF	52736-53247	Operating system ROM
\$D000-\$D7FF	53248-55295	Character ROM
\$D800-\$FBFF	55296-64511	Operating system ROM
\$FC00-\$FCFF	64512-64767	Banking routines ROM (in all ROM maps)
\$FCF1-\$FCF3	64753-64755	Jump to interrupt routine for cartridge
\$FCF4-\$FCF6	64756-64758	Jump to cartridge reenable routine
\$FCF7-\$FCF9	64759-64761	Jump to long fetch routine
\$FCFA-\$FCFC	64762-64764	Jump to long jump routine
\$FCFD-\$FCFF	64765-64767	Jump to long interrupt routine
\$FD00-\$FD0F	64768-64783	ACIA chip (used for RS232, in all maps)
\$FD10-\$FD1F	64784-64799	Parallel port (6529, in all maps)
\$FD20-\$FDCC	64800-64975	Unknown (in all maps)
\$FDD0-\$FDDF	64976-64991	Cartridge banking port (in all maps)
\$FDE0-\$FEFF	64992-65279	Direct Memory Access disk (in all maps)
\$FF00-\$FF3F	65280-65343	Graphics chip (in all maps)
\$FF40-\$FFFF	65344-65535	RAM (under ROM)
\$FF40 \$FFFF	65344-65535	Operating system ROM
\$FF49-\$FF4B	65353-65355	Jump to function key definition routine
\$FF4C-\$FF4E	65356-65358	Jump to PRINT routine
\$FF4F-\$FF51	65359-65361	Jump to print message routine
\$FF52-\$FF55	65362-65364	Jump to monitor
\$FF80	65408	Version number of operating system ROM (most significant bit 0=NTSC, 1=PAL)
\$FF81-\$FF83	65409-65411	Jump to CINT
\$FF84-\$FF86	65412-65414	Jump to IOINIT
\$FF87-\$FF89	65415-65417	Jump to RAMTAS
\$FF8A-\$FF8C	65418-65420	Jump to RESTOR
\$FF8D-\$FF8F	65421-65423	Jump to VECTOR
\$FF90 \$FF92	65424-65426	Jump to SETMSG
\$FF93-\$FF95	65427-65429	Jump to SECND
\$FF96-\$FF98	65430-65432	Jump to TKSA
\$FF99-\$FF9B	65433-65435	Jump to MEMTOP
\$FF9C-\$FF9E	65436-65438	Jump to MEMBOT
\$FF9F-\$FFA1	65439-65441	Jump to SCNKEY
\$FFA2-\$FFA4	65442-65444	Jump to SETTMO

## MEMORY LOCATION(S)

<i>Hexadecimal</i>	<i>Decimal</i>	<i>Usage</i>
\$FFA5-\$FFA7	65445-65447	Jump to ACPTR
\$FFA8-\$FFAA	65448-65450	Jump to CIOUT
\$FFAB-\$FFAD	65451-65453	Jump to UNTLK
\$FFAE-\$FFB0	65454-65456	Jump to UNLSN
\$FFB1-\$FFB3	65457-65459	Jump to LISTN
\$FFB4-\$FFB6	65460-65462	Jump to TALK
\$FFB7-\$FFB9	65463-65465	Jump to READSS
\$FFBA-\$FFBC	65466-65468	Jump to SETLFS
\$FFBD-\$FFBF	65469-65471	Jump to SETNAM
\$FFC0-\$FFC2	65472-65474	Jump to OPEN
\$FFC3-\$FFC5	65475-65477	Jump to CLOSE
\$FFC6-\$FFC8	65478-65480	Jump to CHKIN
\$FFC9-\$FFCB	65481-65483	Jump to CHOUT
\$FFCC-\$FFCE	65484-65486	Jump to CLRCH
\$FFCF-\$FFD1	65487-65489	Jump to BASIN
\$FFD2-\$FFD4	65490-65492	Jump to BSOUT
\$FFD5-\$FFD7	65493-65495	Jump to LOADSP
\$FFD8-\$FFDA	65496-65498	Jump to SAVESP
\$FFDB-\$FFDD	65499-65501	Jump to SETTIM
\$FFDE-\$FFE0	65502-65504	Jump to RDTIM
\$FFE1-\$FFE3	65505-65507	Jump to STOP
\$FFE4-\$FFE6	65508-65510	Jump to GETIN
\$FFE7-\$FFE9	65511-65513	Jump to CLALL
\$FFEA-\$FFEC	65514-65516	Jump to UDTIM
\$FFED-\$FFEF	65517-65519	Jump to SCRORG
\$FFF0-\$FFF2	65520-65522	Jump to PLOT
\$FFF3-\$FFF5	65523-65525	Jump to IOBASE
\$FFF6-\$FFF8	65526-65528	Bank in ROM
\$FFF9	65529	Jump opcode
\$FFFA-\$FFFB	65530-65531	Vector (used for reset, not NMIs)
\$FFFC-\$FFFD	65532-65533	RESET vector
\$FFFE-\$FFFF	65534-65535	IRQ vector

**User Groups**

More information on using your Plus/4 is available in a number of magazines. Commodore's magazines list User Groups for owners of Commodore computers. Many of them welcome Plus/4 users. Mr. Calvin Demmon contacted us about his group that specializes in the Plus/4. You can contact them at this address

The Plus/4 Users' Group  
Box 1001  
Monterey, CA 93940

---

# Index

- A (assemble) command, 228–229  
Abbreviations of BASIC commands, 10–12  
ABS, 13–14  
Absolute indexed mode, 295–296  
Absolute mode, 294  
Accessing file manager records, 114, 118, 119, 148  
Accumulator, 240, 244–245, 248–249  
ACPTR, 308  
ADC instruction, 242–244  
Addition, double precision (machine language), 243, 246  
Addressing modes, 293–297  
AND, 7  
AND instruction, 244–245  
Animation, 212–219  
Arcs, 21–23  
Arctangent, 14–15  
Arithmetic operators  
    BASIC, 5  
    spreadsheet, 120–121  
Arrays, 12–13, 31–32, 180–182  
ASC, 14, 21, 113  
ASCII, 113  
ASCII codes, 416–417  
ASL instruction, 245–246  
Assemble (A) command, 228–229  
ATN, 14–15  
AUTO command, 15  
AUTO mode (spreadsheet), 122  
Automatic insert mode, 159–160  
Automatic line numbering, 15
- BACKUP, 16–17, 27–28  
BAM, 339  
Banking, 303–304  
BASIC error messages, 395–405  
BASIC tokens, 406–407  
BASIN, 312  
BCC instruction, 246–247  
BCS instruction, 247  
BEQ instruction, 247–248  
BIT instruction, 248–249
- Bit-mapping, 45–47, 222–224, 225–226  
BLKMAP command (spreadsheet), 122–123  
Block allocate (B-A), 339–340  
Block execute (B-E), 342–344  
Block free (B-F), 340–341  
Block read (U1), 337–338  
Block write (U2), 338–339  
Blocks of text  
    BASIC (using Escape functions), 159–161  
    creating (word processor), 102–103  
    deleting (word processor), 103–104  
    inserting (word processor), 106  
    moving, 122–123, 133–135  
BMI instruction, 249  
BNE instruction, 250  
BOX, 17–18, 198  
BPL instruction, 250–251  
Branching programs, 45, 49–51, 63–64  
BRK instruction, 251–252  
BSOUT, 312  
Buffer Pointer (B-P), 341–342  
Built-in software  
    bar graphs, 100  
    command mode, 98–99  
    file manager commands, 140–149  
    formatting disks, 99–100  
    formatting printed documents, 112–120  
    point graphs, 101  
    screen colors, 99, 125  
    spreadsheet commands, 120–140  
    switching between programs, 98, 112, 139, 148  
    word processor commands, 101–120  
BVC instruction, 252  
BVS instruction, 252–253
- C (compare) command, 229–230  
CA (built-in software), 102, 123, 141  
Canceling  
    half-screen mode (built-in programs), 103, 104, 129  
    other modes, 158  
Carry bit, 242–244, 246–247

- Cassette recorder, 360–371  
 CLOSE, 23–24, 368–369  
 loading programs, 59–60, 362  
 machine language, 364–371  
 OPEN, 64–66, 368  
 positioning a tape (VERIFY), 96, 362–363, 367–368  
 saving programs, 83–84, 362
- Catalog (disk), 32–33, 102, 123, 141, 332
- CB command (built-in software), 102–103
- CCO command (spreadsheet), 124
- CDEL command (spreadsheet), 124
- Cell (spreadsheet), 120
- CENTER instruction (word processor), 114
- Chaining programs, 34, 60
- Changing 6502 registers (; command), 234–235
- Changing memory contents (> command), 234–235
- CHAR, 18–20, 190
- Character animation, 212–215
- Character codes, 14, 20–21
- Character modes, 374–375
- Character sets, 200–209  
 expanding, 220
- Character strings, 3–4, 10, 53–54, 56–57, 61–62, 81, 91, 95
- Characters, 220–222, 224–225
- CHKIN, 311
- CHOUT, 311
- CHR\$ function, 14, 20–21, 55
- CHR\$ codes, 14, 20–21, 408–415, 416, 418
- CINS command (spreadsheet), 124
- CINT, 306
- CIOUT, 308
- CIRCLE, 21–23
- CLALL, 315
- CLC instruction, 253
- CLD instruction, 254
- Clearing  
 files (COLLECT), 25, 334  
 graphic mode screens, 45–47, 85, 156, 188  
 memory (built-in programs), 103, 137–138  
 memory (NEW), 62  
 RESET (spreadsheet), 137–138
- SCNCLR command, 85, 156  
 variables, 24
- CLI instruction, 254–255
- Clock (system), 93, 314
- CLOSE, 23–24, 326
- CLOSE (machine language), 311, 355, 359–360, 368–369, 377, 389
- CLR, 24
- CLRCH, 312
- CLV instruction, 255
- CM command (built-in programs), 103, 125
- CMD, 24–25
- CMP instruction, 255–257
- COLLECT, 25, 334
- Color  
 character color, 25–27  
 COLOR command (BASIC), 24–27, 186–189, 193–194, 196
- COLOR command (spreadsheet), 99, 125
- color sources, 17, 25–27, 66–67
- functions, 75, 81–82
- graphic modes, 17, 26, 66–67
- luminance, 25–27, 81–82, 186–188
- PAINT, 66–67
- programming (BASIC), 186–188
- of screen, 25–27, 125
- Color keys, 155, 189
- Columns (spreadsheet)  
 deleting (CDEL), 124  
 inserting blank columns (CINS), 125
- Command line length (BASIC), 10, 152
- Command mode (built-in programs), 98–99
- Commas, 69–76
- Comments in programs (REM), 77
- Compare (C) command, 229–230
- Comparison operators, 6, 50, 131
- Conditional commands (BASIC), 34–36, 40–42, 49–51, 63–64
- Conditional commands (spreadsheet), 121, 131–132, 135–136, 140
- CONT, 27
- Converting numbers, 30, 49
- Coordinates, 192–193, 195–196, 197–199
- COPY (BASIC), 27–28
- COPY (spreadsheet), 126
- Copying  
 a disk, 16–17, 27–28, 335–337  
 cell entries, 124, 126, 136–137
- COS, 29
- CP command (word processor), 103
- CPX instruction, 257–258
- CPY instruction, 258–259
- Crunching programs, 10–13
- CT command (word processor), 103
- Cursor control  
 escape functions, 158  
 half-screen mode (built-in programs), 105–106  
 pixel cursor, 36–38, 60–61, 197–199  
 POS function, 68  
 spreadsheet, 120, 129, 130  
 Word processor, 101–102
- Cursor position, 68, 315
- Custom character sets, 200–209
- D (disassemble) command, 230–231
- DATA command, 29–30, 76, 79
- Data files, 325–332, 354–360, 363–364, 368–371

- Data types, 3–4  
 Database program. *See* File manager.  
 Datasette, 360–371  
**DB** command (word processor), 103–104  
 Debugging programs, 39, 80, 94–95, 185  
**DEC**, 30  
**DEC** instruction, 259–260  
 Decimal mode, 280–281, 292–293  
   additional example, 254  
**DEF FN**, 30–31  
 Default, 2  
 Defining  
   function keys, 55–56  
   functions, 30–31  
 Deleting  
   blocks of text (word processor), 103–104  
   cell entries, 124, 137  
   ~~DELETE~~, 31  
   files (BASIC), 85–86, 334  
   files (built-in programs), 104, 126  
   lines (BASIC), 160–161  
   lines (word processor), 104  
 Device numbers, 65  
**DEX** instruction, 261  
**DEY** instruction, 259–260  
**DF** command (built-in programs), 104, 126  
**DIM**, 31–32, 180–181  
 Dimensioning arrays, 31–32, 180–181  
 Direct-access programming, 335–337  
**DIRECTORY** command, 32–33, 123, 141, 332  
**Disassemble (D)** command, 230–231  
 Disk drives, 323–360  
   CLOSE, 23–24, 311, 355, 359–360  
   deleting files, 85–86, 334, 347–350  
   directories, 32–33, 123, 141, 332  
   drive status, 38  
   error numbers, 38, 400–405  
   formatting, 48–49, 99–100, 332–333  
   initializing, 130–131, 333  
   loading (BASIC), 34, 324–325  
   loading (built-in software), 107–108  
   machine language, 350–360  
   OPEN, 64–66, 325–326  
   RAM, accessing, 342–347  
   saving (BASIC), 38–39, 83–84, 324–325  
   saving (built-in software), 110–111  
   unSCRATCHing files, 347–350  
   verifying, 96, 324–325  
 Disk maintenance, 331–332  
 Disk sorts (file manager), 141–142  
 Displaying text  
   `CHAR`, 18–20, 190  
   `PRINT`, 19, 68–70, 190  
   `PRINT USING`, 71–73, 74, 190  
   `PUDEF`, 71, 74–76  
 Displaying records (file manager), 146, 147  
 Division by two (example), 276  
 DL command (word processor), 104  
**DLOAD**, 34, 59, 325  
**DO . . . WHILE/UNTIL . . . LOOP**, 34–36  
 Document printing (word processor), 108–109, 113–120  
 Documenting programs (REM), 77  
 Dollar format (spreadsheet), 140  
**DOS**, 323–324  
 DOS error messages, 400–405  
 Dot mode, 375–376  
**DRAW**, 36–38  
**DS (BASIC)**, 38  
**DS (file manager)**, 141–142  
**DS\$**, 38  
**DSAVE**, 38–39, 324–325  
 Duplicating a disk, 16–17, 27–28  
 Duration of a sound, 87–88  
  
 EL, 39  
**ELSE**, 39, 50–52  
**END**, 39  
**EOF?** instruction (file manager), 114  
**EOR** instruction, 261–263  
**EP** command (word processor), 103, 104  
**ER**, 39  
**ERR\$**, 39  
 Errors  
   BASIC error messages, 395–399  
   debugging, 39, 94–95  
   disk drive errors, 38, 324, 400–405  
   DOS error message, 400–405  
   trapping routines, 39, 80, 94, 185  
 Escape key functions  
   editing the screen, 159–161  
   features, 156–163  
   reducing screen-display size, 161  
   screen windows, 162  
   use in programs, 162–163  
**Exclusive OR (XOR)**, 9–10, 97  
**Execute (G)** command, 231–232  
 Execution times (clock cycles), 289–291  
**EXIT**, 34–36, 40  
 Exiting to BASIC, 238  
 Exponentiation, 4, 5–6, 173  
 Expression, 2  
 Extended color mode, 207–209, 221, 224–225  
  
 F (fill) command, 231  
 Fields (file manager), 114, 141  
 File manager  
   accessing from other programs, 98, 112, 139  
   commands, 140–149  
   displaying records, 146, 147

- fields, 114, 141  
new files, 143–144  
records, 119, 143–144, 146  
storing records, 148–149  
subfiles, 141–143, 145–147
- Files  
built-in programs, 102, 107–108, 110–111, 141  
closing, 23–24  
copying, 27–28, 335  
deleting (built-in programs), 126  
directories, 32–33, 102, 123, 141, 332  
disk, 323–360  
duplicating, 15–16, 27–28, 335  
file manager files, 140–149  
linking (word processor), 101, 108–109, 115  
logical file numbers, 23–24  
opening, 64–66, 310–311, 325–326, 358, 368, 377, 389  
renaming, 335  
SCRATCHing, 84–85, 334, 347–350  
sorting (file manager), 141–142  
spreadsheet files, 132–133  
unSCRATCHing, 347–350  
word processor files, 107–108
- Fill (F) command, 231
- FIT command (spreadsheet), 122, 127
- FLD instruction, 114–115
- Floating point accumulator, 316–318
- Floating point format (spreadsheet), 127–128
- Floating point numbers, 3
- FOR . . . TO . . . NEXT, 40–42  
examples, 194, 197
- FORMAT command (spreadsheet), 99–100, 128
- Format defaults (word processor), 113
- Formatting disks  
BASIC, 48–49, 332–333  
Built-in software, 99–100, 128
- Formatting output, BASIC  
PRINT, 68–71  
PRINT USING, 71–73  
PUDEF, 74  
SPaCing, 88–89  
TAB, 92
- Formatting output, built-in software, 112–120
- Formulas (spreadsheet), 120–121, 131–132, 135–136
- FRE (BASIC), 42
- FReeze command (spreadsheet), 129, 140
- FU, 105, 129
- Full-screen mode (built-in programs), 105–106
- Function, 2
- Function keys, defining, 13, 55–56
- Functions, user-defined, 30–31
- G (go) command, 231–232
- GET, 42
- GET#, 42–43  
Get (machine language), 355–358, 369–371, 390
- GETIN, 314
- GETKEY, 42, 56
- GETKEY#, 42
- Go (G) command, 231–232
- GOSUB, 44–45, 63, 80
- GOTO (BASIC), 45, 49–50, 64
- GOTO (spreadsheet), 129
- GR command (built-in programs), 98
- Graphic modes  
clearing the screen, 45–47, 85, 156, 188  
color, 25–27, 66–67, 193–194, 196  
coordinates, 192–193, 195, 197–199  
drawing commands, 17–18, 21–23, 36–38  
exiting, 13, 45–47  
functions, 46–47, 75–76  
GRAPHIC command, 31, 45 47, 156, 188, 192, 195  
high-resolution mode, 45–47, 192–194, 201–204, 220–221, 222–226  
machine language programming, 219–226  
multicolor mode, 18–20, 22, 45–47, 195–197, 204–209, 221–226  
saving and recalling areas, 47–48, 89–90, 215–219  
scaling, 84–85  
split-screen modes, 45–47, 192, 195  
text/graphic mode, 45–47, 188–192  
text on graphic screens (CHAR), 18–20, 190
- Graphics chip register map, 423–425
- Graphs  
accessing built-in graph generator, 98  
bar graphs, 100  
point graphs, 101  
transferring to word processor, 100  
using other symbols in a graph, 101
- GSHAPE, 47–48, 89–90, 215–219
- H (hunt) command, 232
- HA command (built-in programs), 105–106, 129–130
- Half-screen mode (built-in programs), 105–106, 129–130, 134
- HEADER  
BASIC, 48–49, 332–333  
formatting in the built-in programs, 128  
partial header, 49
- HELP, 49
- Hexadecimal values, 30, 49
- HEX\$ function, 49
- Hierarchy of operators, 6
- HIGHRC (spreadsheet), 142–143, 145
- High-resolution graphic mode, 45–47, 192–194  
bit maps, 222, 225–226  
characters, 201–204, 220–221, 224  
colors, 193–194

- HOME command (spreadsheet), 130  
 Hunt (H) command, 232
- IB command (word processor), 106  
 ID command (built-in programs), 106, 130–131  
 IFTRUE (spreadsheet), 121, 131–132, 140  
 IF . . . GOTO . . . ELSE, 49–50  
 IF . . . THEN . . . ELSE, 50–51  
 IL command (word processor), 106–107  
 Immediate mode, 294  
 INC instruction, 263  
 Indexed indirect mode, 297  
 Indirect indexed mode, 296–297  
 Indirect mode, 295  
 Initializing a disk, 106, 130–131, 333–334  
 INPUT, 52  
 INPUT#, 52–53  
 Input/output operations, STatus, 90  
 Insert mode  
     automatic, 159–160  
     manual, 154, 155, 158  
 Inserting  
     blocks of text (word processor), 106  
     columns (spreadsheet), 125  
     lines (BASIC), 160  
     lines (word processor), 106–107  
     rows (spreadsheet), 138  
 INSTR, 53–54, 168  
 Instruction set, 241–292  
 INT, 54  
 INteger format (spreadsheet), 132  
 Integer variables, 3–4  
 Interrupts, 251–252, 245–255, 281–282, 297–303  
 Interval timer, 93–94  
 INX instruction, 264  
 INY instruction, 265  
 IOBASE, 316  
 IONINIT, 306
- JMP instruction, 265–266  
 JSR instruction, 266–267  
 JOY, 54–55  
 Joystick readings, 55  
 Joysticks, 54–55, 390–393  
 Justification (word processor), 113, 115, 116, 132, 138
- KEY command, 55–56  
 Keyword, 1
- L (load) command, 232–233  
 LDA instruction, 267–268  
 LDX instruction, 268–269  
 LDY instruction, 269–270  
 LEFT\$, 56–57, 167, 168  
 LEFTJ (word processor), 132  
 LEN, 57, 167–168  
 LET, 58  
 LF command (built-in programs), 107, 132–133  
 Line length (BASIC), 10, 152  
 Lines per page (PAGELEN, word processor), 117  
 LINKFILE (word processor), 101, 108–109, 115  
 Linking word processor files, 101, 108–109, 115  
 LIST, 58  
 LISTing to the printer, 24–25  
 LISTN, 309  
 LOAD, 59–60, 324–325, 362  
 Load (L) command, 232–233  
 Load (machine language), 232–233, 351–352, 362, 366–367  
 Loading programs  
     cassette tapes, 59, 362  
     disks, 59–60, 324–325, 351–352  
     built-in software, 98, 107–108, 132–133  
 LOADSP, 313  
 LOCATE, 60–61  
 LOG, 61  
 Logarithms, 61, 173  
 Logical operators, 6–10  
 Loops  
     DO . . . WHILE/UNTIL . . . EXIT . . . LOOP, 34–36  
     FOR . . . TO . . . NEXT command sequence, 40–42  
 LSR instruction, 270–271  
 Luminance, 25–27, 81–82, 186–188
- M (memory) command, 233–234  
 Machine language  
     close, 311, 355, 359, 368, 377–378, 389  
     data files, 354, 368  
     datasette, 364–371  
     disk drive, 350–360  
     examining a program (disassemble), 230–231  
     executing programs, 91–92  
     function key definitions, 171–172  
     get, 355–358, 369–371, 390  
     graphics, 219–226  
     joysticks, 392–393  
     loading, 232–233, 351–352, 362, 366–367  
     monitor, 62, 405  
     music, 178–180  
     open, 310–311, 354, 358, 368, 377, 389

- print, 355, 369, 378–379, 389–390  
printer, 377–380  
programming, 227–322  
RS232, 389–390  
saving, 236, 351, 365–366  
subroutines, 95, 266–267  
unNEWing programs, 183–184  
verifying, 237–238, 352–354, 367–368  
Manual mode (spreadsheet), 133  
MAP (spreadsheet), 100, 105, 130, 133–135  
Margins (word processor), 113, 116, 119  
Mathematics  
  calculations, 172–174  
  functions, 13, 14, 15, 29, 30, 40, 49, 54, 61, 92, 95  
  operators (BASIC), 5  
  spreadsheet, 120–121  
Matrix. *See* Arrays.  
MEMBOT, 307  
Memory available, 42  
Memory execute (M-E), 346–347  
Memory (M) command, 233–234  
Memory maps, 224–226, 423–439  
Memory read (M-R), 344–345  
Memory usage, 425–427  
  BASIC graphics, 46, 188  
  reducing, 12–13  
Memory write (M-W), 345–346  
MEMTOP, 307  
Merging word processor files, 107–108  
MF command (word processor), 107–108  
Microprocessor (6502), 227, 239–297  
MID\$, 61–62, 167, 168  
Modems, 380–388  
MONITOR, 62, 183–184, 228  
Moving BASIC RAM, 321–322  
Multicolor mode, 45–47, 195–197  
  bit maps, 223–224, 226  
  CHAR, 18–20  
  characters, 204–209, 221–222, 225  
  colors, 22, 196  
  example, 272  
Multiplication by two (example), 275  
Music, 174–180, 421–422
- Nesting loops, 41, 51  
NEW command, 62, 183–184  
NEWTF (file manager), 143–144  
NEXTPAGE (word processor), 116  
NOJUSTIFY (word processor), 116  
NOP instruction, 271  
NOT, 8–9  
Notes (musical), 87–88, 174–180, 421–422  
NOTIFTRUE (spreadsheet), 121, 135–136, 140
- NO#PAGE (word processor), 117  
NOWRAP (word processor), 116  
NR command (file manager), 144  
Numbers, 3
- OFF (spreadsheet), 136  
ON . . . GOSUB, 63  
ON . . . GOTO, 64  
Opcodes, 229, 288–289, 291–292  
OPEN, 64–66, 325–326  
Open (machine language), 310–311, 354, 358, 368, 377, 389  
Operating system, 303–316  
Operator, 2  
Operators  
  arithmetic, 5  
  comparison, 6  
  logical, 6–10  
  mathematical (BASIC), 5  
  mathematical (spreadsheet), 120–121  
  relational, 6  
OR, 8, 9–10  
ORA instruction, 271–272  
OTHER (word processor), 117  
Output. *See also* Formatting.  
  linking word processor files, 101, 108–109, 115  
  printer commands (BASIC), 23–24, 24–25, 64–65, 68–71  
  printer commands (word processor), 112–120  
  printer types (word processor), 117  
  redirecting (CMD), 24–25  
  suspending a printout (word processor), 117, 118  
Overflow, two's complement, 239, 252
- P\* command (word processor), 108  
PAGELEN (word processor), 117, 118  
PAGEPAUSE (word processor), 117  
#PAGE (word processor), 117  
PAINT, 22, 66–67  
PAPERSIZE (word processor), 117–118  
Parameter, 2  
PAUSE, 118  
PEEK, 67, 151–152  
PHA instruction, 272–273  
PHP instruction, 273–274  
PI command (file manager), 145–146  
Pixel cursor, 18, 38, 60–61, 197–199  
PLA instruction, 274  
PLOT, 315  
PLP instruction, 274–275  
Pointers (word processor), 103, 104, 106, 111

POKE, 68, 151–152  
 POS, 68  
 PR command (word processor), 108–109  
 Precision, numeric, 3–4, 172  
 PRINT, 19, 68–70, 190  
 Print (machine language), 355, 369, 377–380, 389–390  
 PRINT USING, 71–73, 74, 190  
 PRINT#, 70–71, 334  
 PRINT# USING, 71–73  
 Printer command characters, 373–374  
     word processor, 113  
 Printers, 371–380  
 Printing programs, 24–25  
 Printing word processor files, 108, 112–120  
 Program counter, 239  
 Program files, 328  
 Programs  
     lines, 152–153  
     loading BASIC, 34, 59–60, 325  
     loading machine language, 232–233, 351–352, 362, 366–367  
     machine language, 227–322  
     saving BASIC, 38–39, 83–84, 325  
     saving machine language, 236, 351, 365–366  
     verifying, 237–238, 325, 352–354, 367–368  
 PUDEF, 71, 74  
 Punctuation, 69–70

Quote mode, 21, 153–154, 155, 158, 408–415  
     printer, 374

R (register) command, 235  
 RAM relocating, 321–322  
 RAMTAS, 306  
 Random number generation, 82–83, 174  
     example, 181  
 Raster interrupts, 299–300  
 Raster lines (examples), 256–257, 299–300  
 RC instruction (word processor), 118  
 #RC instruction (word processor), 118  
 RCLR, 75, 76  
 RCO (spreadsheet), 136–137  
 RDEL (spreadsheet), 137  
 RDOT, 75–76  
 RDTIM, 314  
 RE command (word processor), 109–110  
 READ, 29, 76–77, 79  
 READSS, 309–310  
 Records (file manager), 141, 144, 146, 148–149  
 Redefining function keys, 55–56, 169–170  
 Reducing screen display, 161–162  
 Register (R) command, 235

Registers, 239–240  
     changing (:) command, 235–236  
     R (register) command, 236  
 Relational operators. *See* Comparison operators.  
 Relative files, 328–331  
 Relative mode, 294–295  
 Relocating BASIC RAM, 321–322  
 REM, 77  
 RENAME, 77–78, 335  
 RENUMBER, 78–79  
 RESET, 137–138  
 RESET (machine language), 316  
 RESETLIST (file manager), 146–147  
 RESTOR, 306  
 RESTORE, 29–30, 76–77, 79  
 RESUME, 80  
 RETURN, 80  
 Reverse mode, 20, 99, 158  
 RGR, 80–81  
 Right justification (word processor), 115, 138  
 RIGHTS\$, 81, 167, 168  
 RIGHTJ (spreadsheet), 138  
 RINS (spreadsheet), 138  
 RLUM, 81–82  
 RMARG (word processor), 119  
 RND, 82–83  
 ROL instruction, 275–276  
 ROM, 304–305  
 ROM subroutine error messages, 405  
 ROR instruction, 276  
 Rounding numbers, 54, 173  
 RS-232C, 380–390  
 RTI instruction, 277  
 RTS instruction, 278  
 RUN, 83

S (save) command, 236  
 SAVE, 83–84, 325, 362  
 Save (machine language), 236, 351, 365–367  
 SAVESP, 313–314  
 Saving programs  
     DSAVE command, 38–39, 324  
     SAVE command, 83–84, 325, 362  
     SF (built-in programs), 110–111, 139  
 Saving records (file manager), 148–149  
 SBC instruction, 278–280  
 SCALE, 84–85  
 Scientific notation, 4  
 SCNCLR, 85, 156  
 SCNKEY, 308  
 SCRATCH, 85–86, 334, 347–350  
 Screen  
     changing colors, 24–27, 66–67, 99, 125, 193–194  
     clearing, 45–47, 85, 154, 156, 188  
     display codes, 418–420

- display reduction, 161
- editing with Escape keys, 156–163
- graphic modes, 45–47, 186–222
- memory map, 151
- windows, 162
- Screen memory (examples), 261, 282
- Scrolling control, 158–159, 209–211
- SCRORG, 315
- Search and replace (word processor), 109–110
- Searching file manager files, 147–148
- Searching word processor documents, 109–110, 111–112
- SEC instruction, 280
- SECND, 307
- SED instruction, 280–281
- SEI instruction, 281–282
- Sequential files, 326–327
- Serial bus, 358, 379
- SETLFS, 310
- SETMSG, 307
- SET#PG (word processor), 119
- SETNAM, 310
- SETTIM, 314
- Setting pointers (word processor), 111
- SETTMO, 308
- SF (built-in programs), 110–111, 139
- SGN, 86–87
- SIN, 87
- Sorting file manager records, 141–143
- SOUND, 87–88, 96, 174–176, 421–422
- Sound (machine language), 178–180
- SPC, 88–89, 92
- Speeding program execution, 13, 172–173
- Split-screen graphic modes
  - CHAR messages, 20
  - GRAPHIC command, 45–47
  - high resolution, 45–47, 192
  - multicolor, 45–47, 195
- Spreadsheet
  - accessing, 98, 112, 148
  - commands, 120
  - cursor control, 120, 129, 130
  - formulas, 120–121
  - graphs, 100–101
  - transferring to word processor, 122–123, 129–130, 133–135
- SQR, 89
- SR command (file manager), 147–148
- SSHAPE, 47–48, 89–90, 215–219
- STA instruction, 282
- Stack, 240–241
  - example, 285–286
- Stack pointer, 240
- STatus, 90
- Status register, 239–240
- STOP, 27, 91
- STOP (machine language), 314
- STR\$, 91, 168–169
- String variables, 2–3, 5
- Strings, comparing, 10
- Strings, searching, 53
- STX instruction, 283
- STY instruction, 283–284
- Subfiles (file manager), 141–143, 145–147
- Subroutines
  - BASIC, 44–45, 62–63, 80
  - machine language, 95, 241, 266–267, 278, 318–320
- Substring functions, 167
- Subtraction, double precision (machine language), 247, 279
- SYS, 91–92, 318–320
- \$\$ format (spreadsheet), 140
  
- T (transfer) command, 237
- TAB, 89, 92
- Tabs (word processor), 101, 103
- TALK, 309
- TAN, 92–93
- TAX instruction, 284
- TAY instruction, 285
- TC command (built-in programs), 112
- Telecomputing, 380–390
- Text mode, 188–192
- Text strings, 3–4, 10, 53–54, 56–57, 61–62, 81, 91, 95, 163–169
- TF command (built-in programs), 112, 139, 148
- TF::RC instruction (word processor), 119
- THAW command (spreadsheet), 140
- TI, 93
- TI\$, 93–94
- Timer interrupts, 300–303
- TKSA, 307
- TRANSFER (spreadsheet), 140
- Transfer (T) command, 237
- Transferring data
  - file manager to word processor, 114–115, 118, 119
  - graph generator to word processor, 100
  - spreadsheet to word processor, 105, 122–123, 129–130, 133–135
- TRAP, 39, 80, 94, 185
  - example, 64
- Trigonometric calculations, 14, 28, 87, 92, 173
- TROFF, 94
- TRON, 95
- TSX instruction, 285–286
- TTL instruction (word processor), 119
- TW command (built-in programs), 139, 148
- TXA instruction, 286–287
- TXS instruction, 287
- TYA instruction, 287–288

- UD, 148–149  
 UDTIM, 315  
 UNLSN, 309  
 UnNEWing, 183  
 UnSCRATCHing, 347–350  
 UNTIL, 34–36, 95  
 UNTLK, 309  
 User-defined functions, 30–31  
 User files, 331–332  
 User groups, 439  
 USR, 316–317
- V (verify) command, 237–238  
 VAL, 95, 168–169  
 Validating a disk, 334  
 Variables, 2–3, 5, 24  
 VECTOR, 306  
 VERIFY, 96, 326, 362–363  
 Verify (V) command, 237–238  
 Verify (machine language), 352–354, 367–368  
 VOL, 87, 96–97, 174
- Word processor  
 accessing, 98, 139, 148  
 commands, 101–120  
 cursor control, 101–102  
 formatting documents, 112–120  
 graphs, 100–101  
 linking multiple files, 101  
 transferring from file manager, 114–115, 118, 119  
 transferring from spreadsheet, 105, 122–123,  
     133–135  
 Word wrap, 116, 120  
 WRAPON (word processor), 120
- X (exit) command, 238  
 X register, 240  
 X-off, 383, 431  
 X-on, 383, 431  
 XOR, 9–10
- Y register, 240

- WAIT, 97  
 example, 361–362  
 Wedge (BASIC), 321  
 WHILE, 34–36, 97
- Zero page indexed mode, 296  
 Zero page mode, 294

## **More Commodore Books from Scott, Foresman and Company**

### **Programming Commodore Graphics with Your 64 or 128**

This highly readable tutorial will help you expand your graphics programming skills on your C-64 *and* on the new C-128. It covers character graphics, sprites, and bit-mapped graphics, and offers timesaving programming tools. By Lane. **\$14.95**, 224 pages

### **The Commodore 64 Family Helper**

This book/software package gives you 5 extremely useful family programs: a complete word processing program, a database manager, memo calendar, checkbook manager, and backgammon game. By Daley & Daley. **\$19.95**, 192 pages

### **Commodore 64 Tutor for Home and School**

Learn Logo, PILOT, and BASIC quickly and easily on your Commodore 64. Includes many examples, illustrations, and 3 full-length sprite and music programs. By Knott & Prochnow. **\$21.95**, 210 pages

#### **To order,**

contact your local bookstore or computer store, or send the order card to

#### **Scott, Foresman and Company, PPG**

1900 East Lake Avenue

Glenview, IL 60025

#### **In Canada, contact**

Macmillan of Canada

164 Commander Blvd.

Agincourt, Ontario

M1S 3C7

## Order Form

Send me:

Programming Commodore Graphics, \$14.95, 18084

Commodore 64 Family Helper, \$19.95, 18059

Commodore 64 Tutor, \$15.95, 18074

Check here for a free catalog

Please check method of payment:

Check/Money Order     MasterCard     Visa

Amount Enclosed \$ \_\_\_\_\_

Credit Card No. \_\_\_\_\_

Expiration Date \_\_\_\_\_

Signature \_\_\_\_\_

Name (please print) \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip \_\_\_\_\_

Add applicable sales tax, plus 6% of Total for U.P.S. (Publisher pays regular book rate).

Full payment must accompany your order. Offer good in U.S. only.

A18249

17 4 680

A 17 87 4







**"Anyone who is writing software for this computer will find it necessary to have this book at hand."**—Dr. Richard F. Daley,  
Microcomputer writer and consultant

Whether you're writing programs for yourself or for commercial distribution, you'll find this book an indispensable guide to programming the Commodore Plus/4.

Written in a clear, easy-to-understand style, this complete handbook includes extensive reference material and many practical programming examples. The **Programmer's Reference Guide for the Commodore Plus/4**

- fully describes the use of each BASIC 3.5 command
- gives tips on using the built-in integrated software
- reviews major programming techniques
- offers a graphics tutorial in BASIC and machine language
- explains how to use the machine language monitor, 6502 assembly language, and the operating system
- offers information on using major Plus/4 peripherals and more.

You'll find over 200 short program examples, in-depth information not found in the Plus/4 manual, and helpful explanations of DOS and BASIC error messages. In addition, the appendices provide the Plus/4 memory and register maps and other important technical specifications. This quick, handy reference will be invaluable to all programmers, from beginners to professionals.

A resident of Abington, Pennsylvania, **Cyndie Merten** is a founding member of Dyadic Software Associates, a microcomputer consulting firm. She was previously a programmer for Commodore Business Machines.

**Sarah Meyer** is a free-lance technical writer who has published two other books about the Plus/4. A resident of Conway, Massachusetts, Ms. Meyer was formerly a Technical Writer and Editor at Commodore Business Machines.

**Scott, Foresman and Company**

ISBN 0-673-18249-

