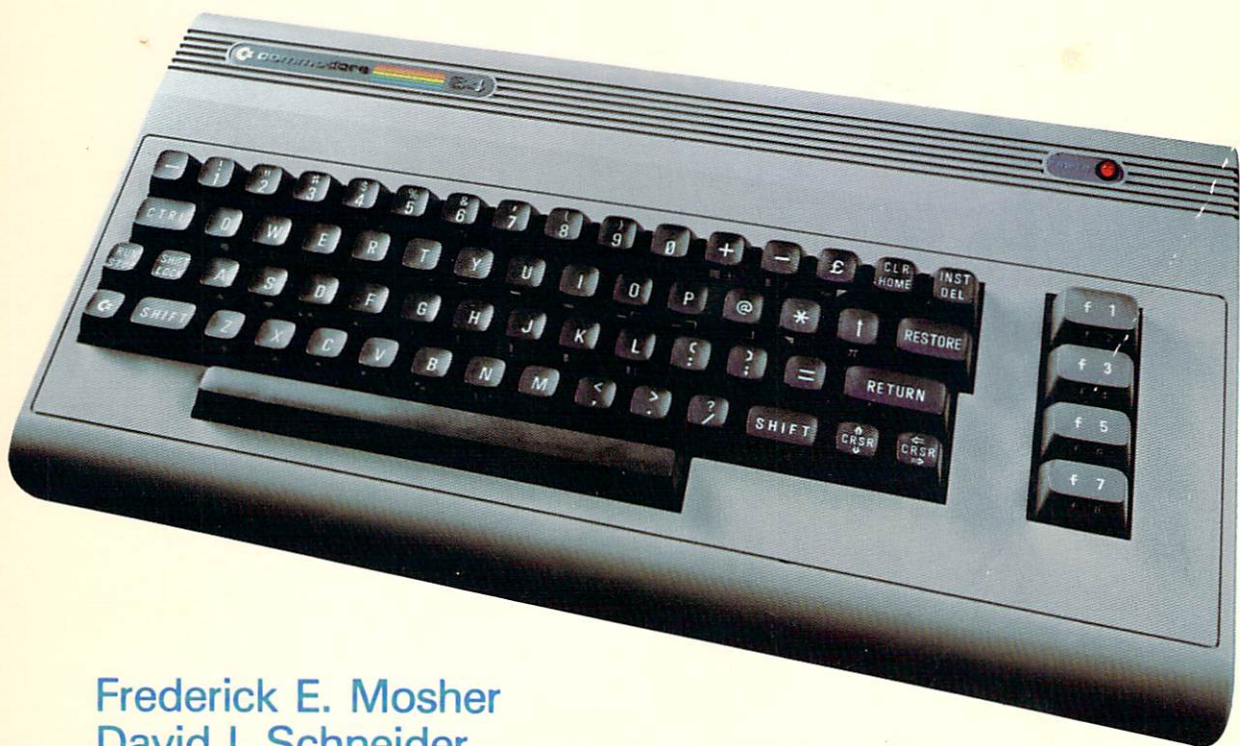


Handbook of

BASIC

for the

commodore 64



Frederick E. Mosher
David I. Schneider

Some Important Numbers

- 1 number of bytes used to store a character
- 2 number of significant characters in variable name
- 2 number of bytes used to store an integer value
- 5 number of bytes used to store a floating point value
- 8 number of binary digits in one byte
- 9 maximum number of digits displayed in a floating-point number
- 10 maximum number of open files or devices
- 10 maximum range of a subscripted variable without using DIM
- 21 rows of dots in a SPRITE
- 24 columns of dots in a SPRITE
- 25 number of rows of text on screen
- 38 maximum number of characters as INPUT prompt
- 40 number of columns in a physical line
- 80 number of character positions in a logical line
- 80 maximum number of characters in a BASIC program line
- 200 number of pts. in vert. line in hi-res. graphics mode
- 255 maximum length of string variable
- 255 largest value which can be POKEd into a memory location
- 320 number of pts. in hor. line in hi-res. graphics mode
- 1024 beginning location of screen memory map
- 1024 number of bytes in 1K
- 32767 largest positive integer constant
- 32768 negative integer constant of greatest magnitude
- 55296 beginning location of color memory map
- 63999 largest allowable line number in a program
- 65535 address of highest memory location
- 3.141592654 numeric constant given by the π key
- 1.70141884E + 38 positive machine infinity (largest number possible)
- 2.93873588e-39 smallest positive number

Device Numbers

- 0 keyboard
- 1 cassette player
- 2 RS-232 interface
- 3 screen
- 4 printer
- 8 disk drive

Some Handy PEEKs & POKEs

- | | |
|------------------|---|
| PEEK(198) | number of characters in keyboard buffer |
| PEEK(631) | first character in keyboard buffer |
| POKE 785,addlow | USR address, low byte |
| POKE 786,addhi | USR address, hi byte |
| POKE 53269,0 | turn off all SPRITEs |
| POKE 53280,color | screen border color |
| POKE 53281,color | screen background color |
| POKE 54296,0 | turn off all voices (volume = 0) |

Handbook of BASIC for the Commodore 64

Publishing Director: David Culverwell
Acquisitions Editor: Terrell Anderson
Production Editor: Karen Zack
Art Director/Cover Design: Don Sellers
Assistant Art Director: Bernard Vervin
Manufacturing Director: John A. Komsa
Cover Photo/Photography: George Dodson
Typesetter: Caledonia Composition
Waynesboro, PA
Printer: Fairfield Graphics
Fairfield, PA
Typefaces: Aster (text and display), Dot Matrix (programs)

Handbook of BASIC for the Commodore 64

*Frederick E. Mosher
David I. Schneider*

Robert J. Brady Co.
A Prentice-Hall Publishing and Communications Company
Bowie, MD 20715

Handbook of BASIC for the Commodore 64

Copyright © 1984 by Robert J. Brady Company.
All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage and retrieval system, without permission in writing from the publisher. For information, address Robert J. Brady Co., Bowie, Maryland 20715

Library of Congress Cataloging in Publication Data

Mosher, Frederick E., 1955-
Handbook of BASIC for the Commodore 64.

Includes index.

1. Commodore 64 (Computer)—Programming. 2. Basic (Computer program language) I. Schneider, David I.

II. Title.

QA76.8.C64M674 1984 001.64'24 83-26618

ISBN 0-89303-505-X

Prentice-Hall International, Inc., London
Prentice-Hall Canada, Inc., Scarborough, Ontario
Prentice-Hall of Australia, Pty., Ltd., Sydney
Prentice-Hall of India Private Limited, New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Prentice-Hall of Southeast Asia Pte. Ltd., Singapore
Whitehall Books, Limited, Petone, New Zealand
Editora Prentice-Hall Do Brasil LTDA., Rio de Janeiro

Printed in the United States of America

84 85 86 87 88 89 90 91 92 93 94 10 9 8 7 6 5 4 3 2 1

Contents

Preface

Preliminary Material

A.	Direct Mode versus Program Mode	1
B.	Multiple-statement Lines	1
C.	Numeric Constants and Variables	2
D.	String Constants and Variables	2
E.	Numeric Operators and Expressions	2
F.	String Operators and Expressions	3
G.	Relational Operators, Conditions, and Logical Operators	3
H.	Order of Operations	3
I.	Character Sets	5
J.	Standard versus Reverse Video Display Mode	6
K.	Quote Mode	6
L.	Insert Mode	7
M.	ASCII/Commodore Codes	7
N.	Physical Screen Line versus Logical Statement Line	7
O.	Peripheral Devices and Device Numbers	7
P.	Filenames	8
Q.	Conventions Used in This Handbook	8

Part I: BASIC Commands, Statements, and Functions

ABS	13	INPUT	100
ASC	15	INPUT#	108
ATN	20	INT	120
CHR\$	23	LEFT\$	123
CLOSE	26	LEN	126
CLR	29	LET	128
CMD	31	LIST	132
CONT	35	LOAD	135
COS	38	LOG	139
DATA	41	MID\$	143
DEF FN	45	NEW	147
DIM	50	ON . . . GOSUB and	
END	56	ON . . . GOTO	149
EXP	59	OPEN	154
FOR and NEXT	62	PEEK	165
FRE	72	POKE	167
GET	74	POS	170
GET#	78	PRINT	172
GOSUB and RETURN	83	PRINT#	181
GOTO	90	READ	186
IF	94		

REM	190	STOP	221
RESTORE	192	STR\$	224
RIGHT\$	195	SYS	226
RND	198	TAB	229
RUN	205	TAN	234
SAVE	207	TIME	236
SGN	209	TIMES\$	239
SIN	211	USR	242
SPC	213	VAL	244
SQR	216	VERIFY	246
STATUS	218	WAIT	248

Part II: SPRITES

SPRITE Design (High-Resolution)	253
SPRITE Design & Color (Multi-Color)	259
Expanding SPRITES	265
Enabling SPRITES	267
Moving SPRITES	269
SPRITE Display Priorities	273
SPRITE Collision Detection	275

Part III: MUSIC SYNTHESIS

MUSIC NOTE SELECTION	281
MUSIC VOLUME AND AMPLITUDE ENVELOPE	285
MUSIC WAVEFORMS AND ON/OFF CONTROL	289

Appendices

A. ASCII/Commodore Values and the Character Sets	295
B. Binary Representation of Numbers	305
C. Color Codes	307
D. Disk Operating System Commands	308
E. Error Messages	312
F. Data Files	316
G. Generator Program for SPRITES	319
H. Memory Allocation in the Commodore 64	322
I. Reserved Words	324
J. 0 to 255 in Decimal and Binary	325
K. Printer Control Characters	327
L. Logical Operators	331
M. Mathematical Concepts	336
N. SPRITE POKE Summary	340
O. Music Synthesis POKE Summary	341
P. RS-232 Interface Communications	343
Q. Standard Graphics	345
R. Three-Voice Chorus Program	349
S. Music Note Scaled Frequencies	354

Preface

Books on BASIC are primarily of two types: reference manuals that provide a formal description of each BASIC statement, and textbooks intended to be read from beginning to end. This book combines features of both. It has the organization and depth of a reference manual, yet provides the motivation and applications found in textbooks.

The BASIC language, as enhanced for the Commodore 64 computer, has a repertoire of more than 60 statements, functions, and commands. Most of these have numerous extensions and variations. We have tried to make each of them accessible by first discussing them in their most used forms, with concrete examples, before proceeding to their subtler and more sophisticated variations.

Concepts are best explained by illustrating them with carefully thought-out examples. Therefore, we have included over 350 examples. Most of the times that a program is presented, the result of executing the program is also shown.

Prerequisites The reader should have a minimal knowledge of BASIC at the level that can be acquired by reading the first few chapters of a beginning book on BASIC (preferably a book specifically written for the Commodore 64). For instance, we assume that the reader can understand short programs using the 9 common BASIC statements PRINT, LET, GOTO, FOR, NEXT, RUN, INPUT, READ, and DATA.

Demonstration programs Most of the programs in this book are intended to illustrate the uses of BASIC statements. On the other hand, there are some programs, referred to as “demonstration programs,” which the reader should not necessarily try to analyze, but should just type in and run. An example of a demonstration program is the SPRITE generating program in Appendix G.

Acknowledgments

Many people have assisted us in writing this book. First of all, the staff of the Robert J. Brady Co. demonstrated how important the publisher is at every stage of the production of a manuscript. Harry Gaines, David Culverwell, and Terry Anderson suggested that we undertake this project and provided encouragement and direction. Bernard Vervin worked closely with us to produce the extensive artwork, and Karen Zack provided patient editorial assistance. In addition, we are grateful to Neil Lofgren and Thomas Bechtold for their assistance in checking the programs.

Frederick E. Mosher
David I. Schneider
Silver Spring, Maryland

Limits of Liability and Disclaimer of Warranty

The authors and publisher of this book have used their best efforts in preparing this book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regards to these programs or the documentation contained in this book. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Trademarks of Material Mentioned in This Text

Commodore, Commodore 64, VIC, and Datasette are trademarks of Commodore Business Machines, Inc.

NOTE TO AUTHORS

Do you have a manuscript or a software program related to personal computers? Do you have an idea for developing such a project? If so, we would like to hear from you. The Brady Co. produces a complete range of books and applications software for the personal computer market. We invite you to write to David Culverwell, Publishing Director, Robert J. Brady Co., Bowie, MD 20715.

PRELIMINARY MATERIAL

A. Direct Mode Versus Program Mode

One way of adding the two numbers 2 and 3 is to type in the statement `PRINT 2 + 3` without any line number preceding it and then press the RETURN key. The results will appear as follows:

```
PRINT 2 + 3
5
```

```
READY,
```

Another way to perform the same operation is to write a one line BASIC program and then run it:

```
10 PRINT 2 + 3
RUN
5
```

```
READY,
```

We say that the statement was executed in *direct* mode in the first case and in *program* mode in the second case. Every BASIC statement, except `CONT`, `DEF FN GET`, `GET#`, `INPUT`, `INPUT#`, and `DATA`, can be executed in both modes. `CONT` must be executed in direct mode, while the others listed must be executed in program mode. In addition, if any of the commands `LIST`, `LOAD`, `NEW`, or `RUN` are used in program mode, the program will `END` at the point where the command appears, and then the command will execute in direct mode.

B. Multiple-Statement Lines

More than one statement can be written on the same line in either direct mode or program mode. If this is done, the statements must be separated by a colon. Two examples follow:

```
FOR I = 1 TO 3: PRINT I:: NEXT I: PRINT
1 2 3
```

```
READY,
```

```
10 A$ = "HASTE MA": B$ = "KES WASTE"
20 PRINT A$; B$
RUN
HASTE MAKES WASTE
```

```
READY,
```

PRELIMINARY MATERIAL

C. Numeric Constants and Variables

Numeric constants of two *precisions* are recognized by the Commodore 64: integer constants and floating-point constants. Integer constants can represent whole numbers from -32768 to 32767. Floating-point constants can represent numbers with at most 10 significant digits and whose absolute value falls in the range from $2.93873588 \times 10^{-39}$ to $1.70141183 \times 10^{+38}$. (Note: Ten digits are stored for floating-point constants but only 9 are displayed.) Floating-point constants are often expressed in “scientific” or “floating-point” notation; that is, as a number times a power of 10. In the computer’s notation, the letter E followed by the integer n denotes 10 raised to the n th power.

The name assigned to a constant determines the precision, integer or floating-point, in which the constant is used and stored. A variable name which ends with % indicates that the value is to be used and stored as an integer. A variable name that does not end with % (or \$, see next section) indicates that the value is to be used and stored as a floating-point constant. (See LET for a complete discussion of choosing variable names.) For example, A and A% are two distinct variables. The variable A will have its value used and stored in floating-point form, while A% will have its value used and stored in integer form. If a floating-point constant is assigned to an integer variable, the constant’s value will be converted to an integer value, if possible, according to the rules of the INT function.

One floating-point constant is predefined. The character π assumes the value 3.141592654 when used in numeric computations. This value is an approximation of the mathematical quantity pi, the area of the circle having radius one.

D. String Constants and Variables

Words, phrases, and sentences are examples of string constants or, more simply, *strings*. More generally, any sequence of characters to be treated as a single entity may be regarded as a string. Most of the time, the sequence of characters constituting a string is surrounded by quotation marks to clearly specify the beginning and end of the string.

A string variable is a name ending with a dollar sign, to which string constants can be assigned. See the discussion of LET for further details.

E. Numeric Operators and Expressions

The Commodore 64 recognizes 5 numeric operators: +, -, *, /, and \uparrow , corresponding to addition, subtraction, multiplication, division, and exponentiation. A *numeric expression* is obtained by using any of these operators to join two or more numeric constants, variables, or function values. (A single numeric constant or variable also will be called a

PRELIMINARY MATERIAL

numeric expression.) In addition, parentheses may be used in numeric expressions to control the order in which operations are performed. Some examples of numeric expressions are 123, 2 + 3, Z*4/W, ABS(X)-12*D, and 3*(A + 6).

F. String Operators and Expressions

The Commodore 64 recognizes the single string operator “+” which corresponds to concatenation; that is, the joining together of two strings, to form a new, single string. A *string expression* is obtained by using the plus sign to join two or more string constants, variables, or function values. A single string constant or variable also will be called a string expression. Some examples of string expressions are “MOON”, “SUPER”+“MAN”, A\$+B\$, and CHR\$(34)+“RIVER”+CHR\$(34).

G. Relational Operators, Conditions, and Logical Operators

Six operators are provided by the Commodore 64 for comparing the relationship between two numeric expressions or between two string expressions. These are =, <>, >, <, >=, and <=, corresponding to “is equal to,” “is not equal to,” “is greater than,” “is less than,” “is greater than or equal to,” and “is less than or equal to.” A *simple condition* is formed by using one of these relational operators to join two numeric or two string expressions. Some examples of simple conditions are A=8, C<D+5, and A\$>=B\$.

In the case of simple conditions involving strings, the symbol < means precedes alphabetically and the symbol > means follows alphabetically. The truth value is determined by a character-by-character comparison, starting with the leftmost character, in the same manner that the alphabetical order of words is decided. Thus the condition “APPLE”<“AT” is true since “APPLE” is less than (comes before) “AT” in the dictionary. Strings involving non-letter characters are compared using the order of characters given in the ASCII/Commodore tables of Appendix A.

The Commodore 64 provides 3 logical operators, NOT, AND, and OR, for joining simple conditions into general conditions or logical expressions. See Appendix L for a complete discussion of these operators.

H. Order of Operations

When an expression contains more than one operator, and parentheses are not present to indicate which operation is to be performed first, the computer uses a set of rules, like those of algebra, to decide the order in which the operations will be carried out. These rules make

PRELIMINARY MATERIAL

use of a hierarchy or order of priority of the operators. The following table shows the hierarchy of operators used by the Commodore 64, where the highest priority operator is \uparrow , and the lowest priority operator is OR. Whenever two entries appear on the same line in this table, they have equal priority.

<u>Operator Hierarchy</u>	
	\uparrow
	- (negation)
	* /
	+ -
=	$<>$ $<$ $>$ $<=$ $>=$
	NOT
	AND
	OR

When evaluating an expression, sub-expressions contained in parentheses are evaluated first. When an expression contains more than one operation, and parentheses are not present, operations are performed in the order of their priority. Thus any exponentiation is done first, then any negation, and so forth. If an expression contains several operators of equal priority, then these operations will be performed one at a time, starting with the leftmost operation and proceeding to the rightmost. For example, the expression $2*(3+4)/4+2\uparrow 3*2$ will be evaluated in the following way:

<u>evaluate</u>	<u>leaving</u>
$(3+4)$	$2*7/4+2\uparrow 3*2$
$2\uparrow 3$	$2*7/4+8*2$
$2*7$	$14/4+8*2$
$14/4$	$3.5+8*2$
$8*2$	$3.5+16$
$3.5+16$	19.5

The expression $2*3<6$ AND $4>=7$ OR NOT $3<>5$ will be evaluated as follows:

<u>evaluate</u>	<u>leaving</u>
$2*3$	$6<6$ AND $4>=7$ OR NOT $3<>5$
$6<6$	"false" AND $4>=7$ OR NOT $3<>5$
$4>=7$	"false" AND "false" OR NOT $3<>5$
$3<>5$	"false" AND "false" OR NOT "true"
NOT "true"	"false" AND "false" OR "false"
"false" AND "false"	"false" OR "false"
"false" OR "false"	"false"

PRELIMINARY MATERIAL

Note: Within the computer, "false" is the integer 0, and "true" is the integer -1. Hence PRINTing a logical expression produces 0 or -1. Consider the following statement:

```
PRINT 2*3<6 AND 4>=7 OR NOT 3<>5
```

0

READY.

I. Character Sets

The Commodore 64 provides two sets of 256 characters which can be displayed on the screen or printed on the printer. The following demonstration program displays the 256 characters of Set 1, the Uppercase/Graphics set. After running the program, press SHIFT-**C** (i.e., press the **C** key, located in the lower left-hand corner of the keyboard, while holding down the SHIFT key) to display Set 2, the Lowercase/Uppercase set.

```
10 PRINT CHR$(147); :REM CLEAR THE SCREEN
20 PRINT CHR$(142); :REM CHOOSE UPPERCASE/GRAPHICS
                        DISPLAY MODE
30 SKIP=0
40 FOR K=0 TO 255
50 POKE 1024+2*K+40*SKIP, K
60 POKE 55296+2*K+40*SKIP, 1
70 IF (K+1)/20 = INT((K+1)/20) THEN SKIP=SKIP+1
80 NEXT K
90 GOTO 90
```

Table 3 of Appendix A, entitled "Set 1 \leftrightarrow Set 2 Correspondence," shows all of the 256 characters in each set and specifies which characters appear in place of each other when SHIFT-**C** is pressed.

When the computer is first turned on, the screen is in the Uppercase/Graphics mode. Pressing letter keys displays uppercase letters, and pressing SHIFTed letters displays the graphics characters shown on the right-front part of the keys. Pressing the SHIFT and **C** keys together switches the screen display to Lowercase/Uppercase mode. Each Uppercase/Graphics character currently displayed switches to its corresponding Lowercase/Uppercase character, as shown in Table 3 of Appendix A. Now, pressing letter keys displays lowercase letters, while pressing SHIFTed letter keys displays uppercase letters. Each time that the SHIFT and **C** keys are pressed together, the entire display is switched from the current character set to the other. In both character

PRELIMINARY MATERIAL

sets, pressing the **C** key and a letter key displays the graphics character shown on the left-front part of the key.

Each set of 256 characters can be divided into two equal parts: 128 standard characters and 128 *reverse video* characters. Reverse video characters appear within a cursor-sized rectangle like a photographic negative of the standard character. All standard characters can be entered from the keyboard using a single key or a combination of the SHIFT or **C** keys with another key. Reverse video characters are displayed by first pressing CTRL-9 to enter reverse video mode, and then typing the key or keys for the standard form of the character.

J. Standard Versus Reverse Video Display Mode

Each character that is displayed on the screen is constructed of an 8-dot-wide by 8-dot-high rectangle of small dots. In standard display mode, the dots portraying a character are turned on in what is referred to as the foreground color, while those dots not needed to portray a character are left "turned off" in the background color of the screen. (For a discussion of colors, see Appendix C.) If the key combination CTRL-9 (RVS ON) is pressed, characters are displayed in *reverse video* mode; that is, those dots needed to portray a character are left "turned off" in the background color, while those dots not needed to portray a character are turned on in the foreground color. Reverse video mode will remain in effect until the RETURN key or the key combination CTRL-0 (RVS OFF) is pressed.

K. Quote Mode

Certain keys and key combinations produce special effects. For example, CTRL-3 causes subsequent characters to be displayed in red, SHIFT-CLR/HOME clears the screen and moves the cursor to the upper left-hand corner, and <= CRSR => moves the cursor one column to the right. Quote mode allows these special effects to be included within character strings. The special effects included in a string will be executed when the string is PRINTed.

Quote mode is active whenever an odd number of quotation marks have been typed on a logical line, and becomes inactive whenever an even number of quotation marks have been typed on the line or the RETURN key is typed. (Note: DELETing a quotation mark does not cancel quote mode. A second quotation mark must be typed. If desired, both can then be DELETed.) When quote mode is active, special effects do not occur but instead display as reverse video characters. The tables in Appendix A include a list of special effects, along with the reverse video character in quote mode for each effect. Note: the special effects produced by the keys INS/DEL, RETURN, and SHIFT-RETURN are not affected by quote mode.

PRELIMINARY MATERIAL

L. Insert Mode

As in quote mode, special effect keys display as reverse video characters rather than taking effect, if INSert mode is active. INSert mode is active for each space created in a line by typing the key combination SHIFT-INS/DEL. INSert mode becomes inactive for a given space when a character is typed there, or for all spaces on a line when the RETURN key is pressed. Note: the special effects produced by the keys SHIFT-INS/DEL, RETURN, and SHIFT-RETURN are not affected by INSert mode.

M. ASCII/Commodore Codes

All standard characters and special effects (e.g., cursor controls, character color, and clearing the screen) have been assigned numbers ranging from 0 to 255, called ASCII/Commodore values. (We use the term "ASCII/Commodore" values to distinguish these values from the "ASCII" values used by some other computers.) Appendix A contains tables for both of the Commodore 64 character modes. See the discussions of CHR\$, ASC, and PRINT for further details.

N. Physical Screen Line Versus Logical Statement Line

The Commodore 64 displays 40 characters across a single line of the screen. We refer to such a line on the screen as a *physical* line. When typing in a BASIC statement, up to 80 characters may be used. After 40 characters have been typed on a physical line, additional characters typed appear on the next physical line. These two physical lines will be treated by BASIC as a single line, which we refer to as a *logical* line. A logical statement line ends when the RETURN key is pressed, not when the end of the first physical line is reached. (If 80 characters are typed, the cursor moves to a new physical *and* a new logical line. To enter an 80-character logical line, the cursor must be moved back onto either of the two physical lines which make up the logical line, and then the RETURN key must be pressed.)

O. Peripheral Devices and Device Numbers

A *peripheral device*, or simply *device*, is an object that is connected to the central processing unit of the computer. Each device has an identifying number associated with it. The following table lists the most common devices and their device numbers.

<u>Device</u>	<u>Device Number</u>
Disk drive	8
Cassette player	1

PRELIMINARY MATERIAL

Printer	4
Screen	3
Keyboard	0
RS-232 interface	2

A switch on the printer allows it to be chosen either as device number 4 or 5. We assume in this book that the switch is set so that the printer is device number 4. The RS-232 interface is used to communicate with certain types of modems or printers.

P. Filenames

Two types of files can be created in BASIC, program files and data files. Program files are created when we record a program on cassette or disk with the command SAVE. Data files are created to store items of information so that they can be accessed by BASIC programs. (See Appendix F for a discussion of data files.) Every file can have a *filename* consisting of up to 16 characters. Any character may be used in a filename except the following five: quotation mark ("), asterisk (*), comma (,), colon (:), or question mark (?). In practice, only letters, digits, and the period character tend to be used to form filenames.

Q. Conventions Used in this Book

1. The word "LET" in LET statements can be omitted. For instance, the statement LET N = 25 can be abbreviated to N = 25. We consistently use the abbreviated form.
2. We refer to the person operating the computer as the "user."
3. INPUT statements request information to be typed by the user. We have underlined this information when it appears during the execution of a program.
4. When we specify two keys separated by a hyphen, we imply that the user will hold down the first key while pressing the second. For instance, the combination SHIFT-CLR/HOME can be used to clear the screen.
5. When special effects are to be included in a string, generally as part of a PRINT statement, we do not show the reverse video character that would appear on the screen, but instead enclose in braces, {}, the key combination that is used to obtain the special effect. For example, a program line to clear the screen will be given in the form

```
10 PRINT "{SHIFT-CLR/HOME}"
```

PRELIMINARY MATERIAL

When entering this statement, after typing the first quotation mark, press the key combination SHIFT-CLR/HOME, and then, finally, the second quotation mark. The following will be displayed on the screen:

```
10 PRINT "C"
```

6. We use the word "cassette" to refer to a cassette tape used with the Commodore Datasette recorder, and "disk" to refer to a 5 1/4 inch diskette used with the VIC-1541 disk drive.

7. Unless otherwise noted, all examples have been written and run with the computer in the Uppercase/Graphics character mode.

8. We often refer to "entering" a command or program statement. The term "entering" means to type out all the characters which form the command or statement, and, in addition, to press the RETURN key.

Part I: BASIC Commands, Statements, and Functions

The function ABS strips the minus signs from negative numbers and leaves other numbers unaltered; that is, it returns the *absolute value* of the given number. In particular, if X is any number, then the value of

ABS(X)

is -X when X is negative, and X when X is nonnegative.

Examples

```
1. PRINT ABS(-5); ABS(5); ABS(0)
   5  5  0
```

READY.

```
2. 10 A = -8
   20 B = ABS(A)
   30 PRINT A; B; ABS(B); ABS(A*5);
   40 PRINT ABS((-2)^5); ABS(1.5)
   RUN
   -8  8  8  40  32  1.5
```

READY.

Comments

1. Mathematicians write $|X|$ instead of ABS(X).
2. The graph of $y = \text{ABS}(X)$ is shown in Figure 1.
3. The X in ABS(X) may be a numeric constant, variable, or expression, and may be of either integer or floating-point precision.
4. The value of ABS(X) will have the same precision as X, with one exception. Although -32768 can be stored as an integer, ABS(-32768), or 32768, must be stored in floating-point precision.

```
10 PRINT ABS(-21055); ABS(-8.308E-3)
20 X%=-32768
30 Y%=ABS(X%)
RUN
21055  8.308E-03
```

```
?ILLEGAL QUANTITY  ERROR IN 30
READY.
```

ABS

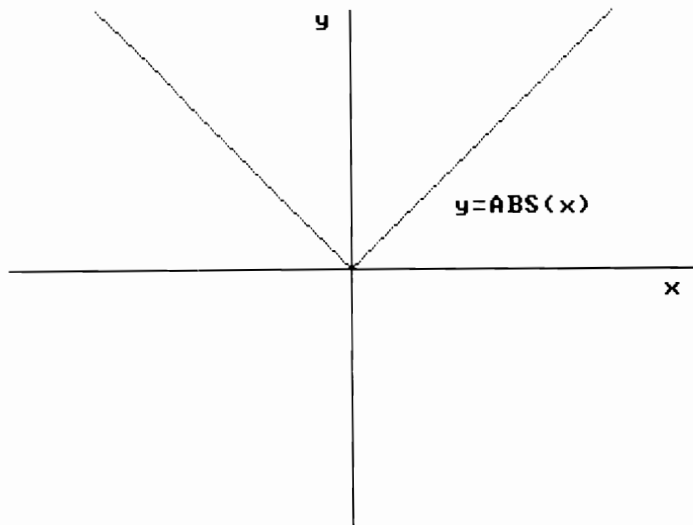


FIGURE 1

Applications

1. The distance between two numbers A and B is $\text{ABS}(A-B)$. For instance, if one person's age is A and another person's age is B , then the difference in their ages is $\text{ABS}(A-B)$.
2. ABS is used to simplify statements. For instance, we usually write $\text{ABS}(X) < 3$ instead of $-3 < X$ AND $X < 3$.
3. In certain numerical calculations for which rounding errors can occur, we hesitate to test whether $A = B$. Instead, we test whether $\text{ABS}(A-B) < T$, where T is an error tolerance we choose. For example, an error of .00001 might be acceptable, and we would test $\text{ABS}(A-B) < .00001$.

The two Commodore 64 character modes are referred to as the Uppercase/Graphics mode and the Lowercase/Uppercase mode. The word "character" refers to both ordinary characters (i.e., symbols) and control characters that can produce special effects. (Two examples of effects are "home cursor" and "change foreground color to yellow".)

Refer to Table 1 or 2 of Appendix A. Each table lists the numbers from 0 to 255, and next to each of these numbers a symbol or effect. (The blank spaces in the "Effect" column refer to the "Null" effect.) Some symbols occur more than once in the table. If so, only one of these occurrences is followed by an entry in its "Key(s)" column.

Each symbol and most of the effects listed in the table for the current character mode can be placed in the string A\$ by first typing

```
A$ = "
```

then pressing the corresponding entry in the "Key(s)" column, and finally typing a quotation mark. Some examples follow.

```
A$ = "█": PRINT A$ (user pressed ⌘-B to produce symbol)
█
```

```
READY.
```

```
A$ = "▣": PRINT A$;"X" (user pressed the CRSR key after
X the first quotation mark to obtain the effect "cursor right")
```

```
READY.
```

If A\$ is a string containing as its first character one of the characters listed in the table of Appendix A corresponding to the current character mode, then the value of the function

```
ASC(A$)
```

is the ASCII/Commodore value associated with the occurrence of the symbol that is followed by an entry in its "Key(s)" column. For instance, in Uppercase/Graphics mode, ASC("\") is 206.

Examples

```
1. PRINT ASC("A"); ASC("%"); ASC("CBM")
   65 37 67
```

```
READY.
```

ASC

```
2. 10 INPUT A$, B$, C$
    20 PRINT ASC(A$); ASC(B$); ASC(C$)
    RUN
    ? A, CBM, " "
      65 67 32

    READY,
```

3. The following program is shown and runs in the Lowercase/Upper-case character set. It asks the user to type a name and then displays the name in uppercase letters.

```
10 print chr$(14) :rem select lowercase
   /uppercase character set
20 print "{SHIFT-CLR/HOME}"
30 input "Name"; n$
40 for i = 1 to len(n$)
50 a = asc(mid$(n$,i,1))
60 if a>64 and a<91 then a = a + 32
70 cn$ = cn$ + chr$(a)
80 next i
90 print cn$
run
Name? George H. "Babe" Ruth
GEORGE H. "BABE" RUTH

ready,
```

Comments

1. The function CHR\$ is the inverse of ASC. For instance, CHR\$(ASC("A")) is A, and ASC(CHR\$(202)) is 202.

2. The function ASC actually is defined for the quotation mark. The difficulty arises in specifying a string whose first character is a quotation mark. (The statement LET A\$ = "" clearly won't work.) One possibility is LET A\$ = CHR\$(34). Another possibility is the program line INPUT S\$: A\$ = MID\$(S\$,2,1) with the user responding by typing any string whose second character is a quotation mark. (In general, the value of MID\$(S\$,n,1) is the string consisting of the nth character of S\$.)

3. There are three control characters that cannot be placed in strings as described in the second paragraph: RETURN, DELETE, and SHIFT-RETURN. However, these characters can be entered into strings with

ASC

the CHR\$ function. The ASCII/Commodore values, which are the values returned by ASC, for these 3 characters are 13, 20, and 141, respectively.

```
A$ = "A" + CHR$(13) + "X": PRINT A$  
A  
X  
  
READY.
```

4. The function ASC is not defined for the empty (or null) string "" which contains no characters. If A\$ = "", then using ASC(A\$) results in the message ?ILLEGAL QUANTITY ERROR. When there is a chance that A\$ is equal to "", as when using GET to read characters from the keyboard, it is a good idea to use the expression ASC(A\$ + CHR\$(0)). If A\$ is not the empty string, this expression is equivalent to ASC(A\$). On the other hand, if A\$ is the empty string, then this expression gives the value 0 rather than causing an error.

5. Characters that have different ASCII/Commodore values can appear the same when printed. For example, in Uppercase/Graphics mode, the characters with ASCII/Commodore values 126, 222, and 255, all appear as π. However, within the computer's memory, these characters are distinct, having different ASCII/Commodore values. The function ASC returns the ASCII/Commodore value stored in memory for a character, and so can return any of the 256 values from 0 to 255.

```
10 A$=CHR$(120): B$=CHR$(216)  
20 PRINT A$;B$;ASC(A$);ASC(B$)  
RUN  
❖❖ 120 216  
  
READY.
```

6. In each character mode there is one exception to the definition of ASC given earlier. If data is read from the screen using the INPUT statement, then in the Uppercase/Graphics mode, ASC("π") is 255 and in the Lowercase/Uppercase mode, asc("❖") is 255. However, if GET is used to read characters directly as they are typed at the keyboard, then ASC will be consistent with the definition given, and will return the value 222 for these characters.

Applications

1. ASC can be used to make a program user-friendly by allowing the user to enter a number without restrictions on using commas. To achieve

ASC

this, the number is input as a piece of string data, and then this string is converted to its equivalent numeric value. Any commas found in the string are simply skipped over. The following program converts A\$ to a whole number V:

```
10 V=0
20 PRINT "TYPE A POSITIVE WHOLE NUMBER:";
30 GET A$: IF A$="" THEN 30
40 PRINT A$;
50 IF (A$>="0") AND (A$<="9") THEN V=10*
V+(ASC(A$)-ASC("0"))
60 IF ASC(A$)<>13 THEN 30
70 PRINT: PRINT V
RUN
TYPE A POSITIVE WHOLE NUMBER:4,341,728
4341728

READY.
```

This program is a partial extension (for positive whole numbers only) of the BASIC function VAL. It can be extended to handle an even greater variety of input by the user.

2. The function ASC can be used to associate a whole number value with a string by adding up the ASCII/Commodore value of each character in the string. For example, the string "HONEY" would be associated with $ASC("H") + ASC("O") + ASC("N") + ASC("E") + ASC("Y")$, which gives $72 + 79 + 78 + 69 + 89 = 387$. This ability to associate numbers with letters allows simple programs that can encode and decode a character string.

A specific application of encoding a string as a number is the *Hash Sort*. In this sorting scheme, a string is stored in an array using a subscript determined by encoding a portion of the string, called the "key", as a number, like 387 for "HONEY", and then finding the remainder when this number is divided by the size of the entire array. (For this process to work well, the size of the array is chosen to be a prime number.) It can occur that two different strings produce the same subscript when encoded in this manner. If one string has been stored at location N in an array, and a second string is given that should also be stored at location N, we say that a *collision* has occurred. Several rules exist for deciding where to store a string that collides, the simplest of which is to try and store the string at location N + 1, and if this is used then N + 2, and so on.

ASC

The following is a simple hash sort program. The program requests a string, and then either stores it in an unused location in the array, or returns the location where the string has been stored previously.

```
10 DIM A$(100) :REM SIZE OF ARRAY IS 101
20 INPUT "YOUR STRING";B$
30 GOSUB 100 :REM ENCODE B$ TO GET C
40 GOSUB 200 :REM STORE OR FIND STRING
50 GOTO 20
100 C=0
110 FOR I=1 TO LEN(B$)
120 C=C+ASC(MID$(B$,I,1))
:REM ADD UP A/C VALUES
130 NEXT I
140 C=C-101*INT(C/101)
:REM FIND REMAINDER
150 RETURN
200 D=C :REM RECORD STARTING LOCATION
210 IF A$(C)="" THEN A$(C)=B$ :RETURN
:REM STORE STRING AT LOCATION C
220 IF A$(C)=B$ THEN PRINT "PREVIOUSLY
STORED AT";C :RETURN
230 C=C+1 : IF C=101 THEN C=0
240 IF C=D THEN PRINT "NO ROOM TO STORE
STRING" :END
250 GOTO 210
```

ATN

ATN is the trigonometric function arctangent, the inverse of the tangent function. For any number X, the value of the function

ATN(X)

is an angle whose tangent is X. The angle is given in radians and lies in the range from $-\pi/2$ to $\pi/2$ (-1.57079683 to 1.57079683). See Appendix M for a discussion of radian measure and the definition of the tangent function.

Examples

```
1. PRINT ATN(3); ATN(-4.8933); ATN(2E+8)
    1.24904577 -1.36921097  1.57079632
```

READY.

```
2. 10 LET A%=2: B=.5
    20 PRINT ATN(A%); ATN(B); ATN(A%+B)
    RUN
    1.10714872  .463647609  1.19028995
```

READY.

Comments

1. Figure 1 shows the graph of $y = \text{ATN}(X)$.
2. The X in ATN(X) may be any numeric constant, variable, or expression.

```
A=2.3: PRINT ATN(5*A13)
    1.5543599
```

READY.

3. The function ATN is the inverse of the function TAN in the sense that if Y equals ATN(X), then X will equal TAN(Y). Also, if A is an angle between $-\pi/2$ and $\pi/2$, and X equals TAN(A), then A will equal ATN(X).

```
10 Y=ATN(3.6): X=TAN(Y)
    20 Z=TAN(.567): A=ATN(Z)
    30 PRINT Y; X; Z; A
```

```

RUN
  1.29984948  3.6  .63674414  .567
READY.
  
```

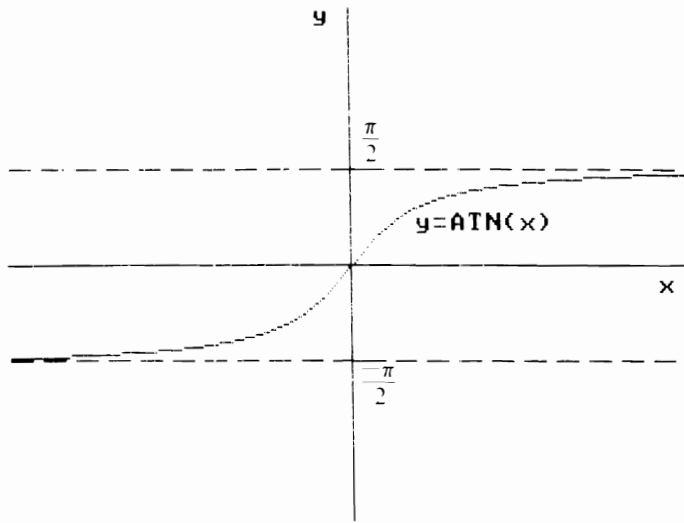


FIGURE 1

Applications

1. Surveyors use ATN to determine angles of elevation. For instance, if you are standing at a distance D feet from a building and the building is H feet taller than you, then you must raise your head at an angle of $\text{ATN}(H/D)$ radians to look at the top of the building. See Figure 2.

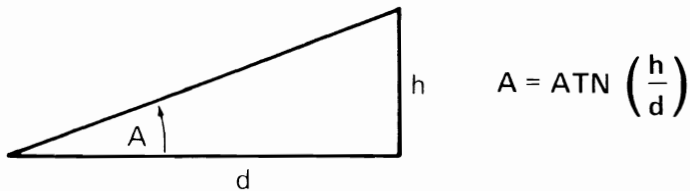


FIGURE 2

2. The other inverse trigonometric functions can be evaluated using ATN. The formulas are:

ATN

$$\arcsin(X) = \text{ATN}(X/\text{SQR}(1-X*X))$$

$$\arccos(X) = \pi/2 - \text{ATN}(X/\text{SQR}(1-X*X))$$

$$\text{arcsec}(X) = \text{ATN}(\text{SQR}(X*X-1)) + (X < 0) * \pi$$

$$\text{arccsc}(X) = \text{ATN}(1/\text{SQR}(X*X-1)) + (X < 0) * \pi$$

$$\text{arccot}(X) = \pi/2 - \text{ATN}(X)$$

Note: SQR is the square root function. The expression (X<0) takes on the value -1 if X is negative, and 0 otherwise. This is because true statements are given the value -1 and false statements are given the value 0. The product (X<0)* π must be added to the principal value of the arctangent, which ATN returns, to get the correct sign.

The Commodore 64 has two character modes, the Uppercase/Graphics mode and the Lowercase/Uppercase mode. Tables 1 and 2 of Appendix A refer to these character modes. Both tables have the numbers from 0 to 255 in their left-hand columns. Next to each of these numbers is a character or effect. If N is a whole number from 0 to 255, then the value of the function

`CHR$(N)`

is the string consisting of the symbol or special effect associated with N in the table representing the current character mode. Each number N is called an ASCII/Commodore value.

Examples

1. The following line was entered in the Uppercase/Graphics mode:

```
PRINT CHR$(49),CHR$(65),CHR$(98),CHR$(186)
1          A          |          ▾
READY,
```

Notice that the character 1 is displayed without a leading space since it is a string here and not a number.

2. The following line was entered in Lowercase/Uppercase mode.

```
print chr$(49),chr$(65),chr$(98),chr$(186)
1          a          B          ✓
ready,
```

3. `CHR$(34)` is the string consisting of the quotation mark.

```
10 A$ = CHR$(34) + "HELLO" + CHR$(34)
20 PRINT A$
RUN
"HELLO"

READY,
```

4. `CHR$(13)` is the string consisting of the RETURN character.

CHR\$

```
PRINT "MERRY" + CHR$(13) + "XMAS"  
MERRY  
XMAS  
  
READY.
```

5. CHR\$(18) is the string consisting of the special effect “reverse on”, and CHR\$(146) is the string consisting of the special effect “reverse off”.

```
10 R$=CHR$(18): N$=CHR$(146)  
20 PRINT "H"+R$+"OW"+N$+"DY"+R$+"!"+"N$  
RUN  
HU!DYH  
  
READY.
```

Comments

1. The function ASC is the inverse of CHR\$. For instance, ASC(CHR\$(202)) is 202, and CHR\$(ASC("A")) is A.

2. The N in CHR\$(N) may be a numeric constant, variable, or expression. If the value of N is not a whole number, CHR\$ uses the truncated value of N, and if this value is not between 0 and 255 inclusive, the message ?ILLEGAL QUANTITY ERROR results.

```
10 A=38.2: PRINT CHR$(A); CHR$(A*2)  
20 PRINT CHR$(10*A)  
RUN  
&L  
  
?ILLEGAL QUANTITY ERROR IN 20  
READY.
```

3. Within the computer’s memory, each of the 256 characters of a given character set is distinct. However, two characters which are different in memory can appear the same when printed. For example, in the Uppercase/Graphics character set, CHR\$(126), CHR\$(222), and CHR\$(255) all appear as π .

4. When two ASCII/Commodore values cause the same character to be printed, that character can be entered from the keyboard in only one way. For example, in Table 1 the ASCII/Commodore values 109 and 205 both correspond to the symbol \. However, only ASCII/Commodore value 205 has an entry in its column labeled “Key(s)”. The func-

CHR\$

tion CHR\$ is required in order to place \ into a string in such a way that the number 109 will be associated with it in memory.

```
10 PRINT "\"; ASC("\")
20 PRINT CHR$(109); ASC(CHR$(109))
RUN
\ 205
\ 109

READY.
```

Applications

1. The CHR\$ function is especially useful in placing the characters INSert, DELeTe, RETURN, and SHIFT-RETURN into strings. In addition, there are four special effects for which no key combinations exist, and thus which must be placed in strings using the CHR\$ function. These four effects are CHR\$(8) to enable the SHIFT-**C** key combination for switching character sets, CHR\$(9) to disable the SHIFT-**C** key combination for switching character sets, CHR\$(14) to select the Lowercase/Uppercase character set, and CHR\$(142) to select the Uppercase/Graphics character set.
2. The CHR\$ function is used extensively to place quotation marks in strings, as in Example 3.
3. The CHR\$ function is used with the statement PRINT# to send control codes to the printer. Appendix K lists the printer control codes.
4. The CHR\$ function may be used to generate special effects characters such as SHIFT-CLR/HOME.

```
CL$=CHR$(147) :REM CLEAR SCREEN CHARACTER
```

This provides a clear way to write special effects characters within a program without producing reverse-video graphics characters which can prove illegible, or at best have no meaning, at a later time when the program is being revised.

CLOSE

Data files on cassettes and disks are created and accessed by OPEN statements. In addition, OPEN statements can be used to access the screen, printer, and modem. When a file or a device is OPENed, it is assigned a reference number and referred to by this number when being written to or read from. (See the discussion of OPEN for details.) Further, each reference number has a corresponding reserved portion of memory, called its buffer, that holds information until it is processed. If a file or device was OPENed with number N, then the statement

CLOSE N

sends all of the information currently in N's buffer to the appropriate place and frees the buffer and the number N from any association with that file or device.

Examples

1. In the following program, line 10 OPENS a cassette data file, with reference number 3, for writing. After the file is written in line 20, it is CLOSED in line 30. Line 60 reOPENS the file, with reference number 5, for reading. It is then CLOSED again in line 90. All files that are OPENed within a program should also be CLOSED within the program.

```
10 OPEN 3,1,1,"STATES"
20 PRINT#3, "ALASKA"
30 CLOSE 3
40 PRINT "REWIND AND HIT RETURN";
50 GET A$: IF A$<>CHR$(13) THEN 50
60 OPEN 5,1,0,"STATES"
70 INPUT#5, A$
80 PRINT A$
90 CLOSE 5
RUN
PRESS RECORD & PLAY ON TAPE
OK
REWIND AND HIT RETURN
PRESS PLAY ON TAPE
OK
ALASKA

READY.
```

CLOSE

2. The following program creates, writes to, and reads a sequential disk file.

```
10 OPEN 2,8,2,"SPORTS,S,W"  
20 PRINT#2, "SOCCER"  
30 PRINT#2, "BASEBALL"  
40 CLOSE 2  
50 OPEN 2,8,2,"SPORTS,S,R"  
60 INPUT#2, A$  
70 PRINT A$  
80 INPUT#2, A$  
90 PRINT A$  
100 CLOSE 2  
RUN  
SOCCER  
BASEBALL
```

READY.

Comments

1. Without the appropriate CLOSE statement, cassette files will be incomplete, new disk files will be empty, and existing disk files being updated will not reflect all of the changes made. Subsequent input from a disk file that was not CLOSED will cause an error with an accompanying blinking red light on the disk drive. For example, if line 30 of Example 1 were deleted, the following would result:

```
RUN  
PRESS RECORD & PLAY ON TAPE  
OK  
REWIND AND HIT RETURN  
PRESS PLAY ON TAPE  
OK
```

(for a new tape, the cassette runs and runs
for an old tape, the cassette may stop, with READY.
printing on the screen.)

Nothing was PRINTed because nothing was written on the tape. The computer saved the information PRINTed by line 20 in the buffer and waited for the buffer to fill up before it took the time to actually write the information on the tape. Without a CLOSE statement, the computer failed to write the partially filled buffer to the tape.

CLOSE

2. If a file number N is OPEN, a CLOSE N statement must be executed before N may be used in another OPEN statement. For example, if line 40 were deleted in Example 2, the following would result:

```
RUN  
  
?FILE OPEN  ERROR IN 50  
READY,
```

3. If the appropriate CLOSE statement is omitted in a program, the file will remain OPEN in direct mode after the program has finished running. (Files also can be OPENed in direct mode independent of any program.) Whenever a file is OPEN in direct mode, care must be taken that the file not be “forgotten” before it is CLOSEd. When BASIC “forgets” a file, it frees the buffer and number N from any association with the file or device without first performing the other actions of the CLOSE statement which are needed to correctly complete files being written to cassette or disk. If a file is “forgotten”, it cannot be CLOSEd, and Comment 1 applies.

BASIC will “forget” all OPEN files if any of the following actions occur in direct mode:

- (a) One of the commands CLR, LOAD, NEW, or RUN is executed.
- (b) A program line is added or edited.
- (c) The RETURN key is hit with the cursor on a (logical) line beginning with a number.

(Actions (b) and (c) are actually equivalent.)

```
OPEN 1,4 : REM DEVICE 4 IS THE PRINTER  
  
READY,  
10 REM A PROGRAM LINE IS ENTERED  
PRINT#1, "FILE 1 HAS BEEN FORGOTTEN"  
  
?FILE NOT OPEN  ERROR  
READY,
```

Whenever you run a program, as soon as you enter the command RUN, all variables are cleared from memory. This same result can be accomplished in the middle of a program by giving the command

CLR

The CLR command is used to free memory space.

Examples

```
1. 10 A = 12: B(5) = 12.89
    20 CLR
    30 PRINT A; B(5)
    RUN
    0 0
```

READY,

```
2. 10 DIM A(7000)
    20 FOR I = 1 TO 7000
    30 A(I) = I*I
    40 NEXT I
    50 CLR
    60 DIM B(7000)
    70 FOR J = 1 TO 7000
    80 B(J) = J*J*J
    90 NEXT J
```

The Commodore 64 has 64K (i.e., 65536 bytes) of random access memory. Of these, approximately 38900 bytes are available for BASIC programs and variables. Lines 10 through 40 will use approximately 35100 bytes of memory, leaving about 3800 bytes. If we hadn't CLeaRed the variables in line 50, we would have received the error message ?OUT OF MEMORY ERROR IN 60.

Comments

1. The CLR command not only removes all variables from memory, but removes the following as well:

- (a) all information set with DEF FN statements
- (b) all file buffers and the meanings of all file reference numbers
- (c) the return addresses for all GOSUB statements which have been executed but not completed by a RETURN statement

CLR

- (d) all information about FOR . . . NEXT loops that are currently executing

```
10 FOR I=1 TO 20
20 PRINT I;
30 IF I=10 THEN CLR
40 NEXT I
RUN
 1  2  3  4  5  6  7  8  9  10

?NEXT WITHOUT FOR  ERROR IN 40
READY.
```

2. The CLR command causes all DIMensioned arrays to become un-dimensioned. For instance, consider the following program, similar to Example 2, but incorporating line 60 into line 10.

```
10 DIM A(3000),B(3000)
20 FOR I = 1 TO 3000
30 A(I) = I*I
40 NEXT I
50 CLR
70 FOR J = 1 TO 3000
80 B(J) = J*J*J
90 NEXT J
RUN

?BAD SUBSCRIPT  ERROR IN 80
READY.
```

The error message resulted since the DIMensioning of the array B(J) was CLearEd, and now the array is only meaningful for J = 0 to 10.

3. The RUN statement is equivalent to the combination of a CLR and a GOTO statement.
4. When used in direct mode, the LOAD statement automatically executes a CLR before loading the program into memory.

Normally, the LIST command displays the current program on the screen. However, this is not always practical since the screen can display only 23 program lines (plus the READY. prompt) at one time. A printout of the complete LISTing of a program can be very useful while developing a program, and can serve as a permanent record when the program is complete.

The printer is just one of several peripheral devices that can be connected to the computer. Some others are a cassette player, a disk drive, and a modem. The discussion of the OPEN statement explains how data files and devices are accessed and assigned a reference number.

The CMD command provides the means of sending a program LISTing to any device or file that has been OPENed. After a device or file has been OPENed with reference number N, the command

CMD N

directs the computer to send all of its output to the device or file associated with N. All information typed by the user will still appear on the screen, but the information provided by the computer that would normally be displayed on the screen will, instead, be "displayed" on the indicated device, or printed into the indicated file. Output that is being *redirected* from the screen to the device or file includes the prompt (READY.), all LISTings, the user's response to an INPUT statement, and the information produced when PRINT statements execute. Normal output to the screen is restored by using PRINT#N to send a blank line to the device or file.

Examples

1. The following commands and statements LIST the current program on the printer and then restore normal output to the screen.

```
OPEN 1,4 :REM THE PRINTER IS DEVICE 4
```

```
READY.
```

```
CMD 1      (the printer types READY.)
```

```
LIST      (the program is LISTed on the printer)
```

```
PRINT#1   (normal output is restored to the screen)
```

```
READY.
```

```
CLOSE 1
```

```
READY.
```

CMD

2. The following commands and statements place a LISTing of the current program in the disk file PROGLIST.

```
OPEN 3,8,3,"PROGLIST,S,W"
```

```
READY.  
CMD 3: LIST  
PRINT#3
```

```
READY.  
CLOSE 3
```

```
READY.
```

Comments

1. The number N in CMD N is normally a whole number constant between 0 and 255 corresponding to a device or file that has been OPENed with reference number N. In general, however, N may also be a numeric variable. If the value of N is not between 0 and 255 inclusive, the message ?ILLEGAL QUANTITY ERROR results. Furthermore, if the value of N has not been associated to a file or device by an OPEN statement, the message ?FILE NOT OPEN ERROR results.

2. A variation of the CMD statement allows a string value to be sent to the file or device without the use of a separate PRINT statement. If A\$ is a string constant or variable, then the statement CMD N, A\$ redirects output and sends the value of A\$ to the appropriate file or device. That is, the command is equivalent to CMD N: PRINT#N,A\$. Note that the prompt "READY." appears after a statement or multiple-statement is executed in direct mode. To avoid having "READY." appear after A\$ is printed, make the CMD command the first part of a multiple-statement, as in the following example.

```
OPEN 1,4
```

```
READY.  
CMD 1,"ACCOUNTING PROGRAM": LIST  
PRINT#1
```

```
READY.  
CLOSE 1
```

```
READY.
```

CMD

Here, the CMD statement was used to title a LISTing to the printer. The prompt does not print until the end of the listing.

3. After a CMD N statement is executed, the file or device associated with the number N is said to be *listening*. While a file or device is listening, it will grab up all output from the computer normally meant for the screen. CLOSEing a file or device for which a CMD statement is active causes all of the normal functions of CLOSE to be performed, but does not “unlisten” the file or device. Thus before CLOSEing a file or device which is listening, the file or device should first be unlistened by sending a blank line to it with a PRINT#N statement. If a file or device is accidentally CLOSED before it is unlistened, simply reOPEN it, PRINT a blank line to it, and then CLOSE it again.

```
OPEN 1,4      (Device 4 is the printer.)

READY,
CMD 1
CLOSE 1
LIST          (The READY. prompt and the LISTing
OPEN 1,4     still appear on the printer.)
PRINT#1      (The printer is now unlistened.)

READY,
CLOSE 1

READY,
```

4. The occurrence of an error message, such as ?SYNTAX ERROR or ?ILLEGAL QUANTITY ERROR, while a CMD statement is active, will immediately unlisten the file or device. The error message and all subsequent output will again be displayed on the screen. Note, however, that the file or device is not closed by the error. Cassette and disk files must still be CLOSED if they are to be complete and readable.

```
OPEN 1,4

READY,
CMD 1        (Printer is now listening.)
LIST        (LISTing on printer.)
PRNT 23*76  (A syntax error is caused.)

?SYNTAX ERROR
READY,
PRINT 23*76
1748       (The printer is no longer listening.)

READY,
```

CMD

```
PRINT#1, "ARE YOU THERE?"      (But the printer is
                                still OPEN as #1.)
READY,
```

5. If a CMD statement has been executed to redirect output to a file or device, and that file or device is accidentally “forgotten” before being unlistened with a PRINT# statement, then the file or device will be incompletely unlistened; output will again appear on the screen, but the cursor will not always advance to a new line when the RETURN key is typed. The easiest way to return to a normal screen display is to cause a syntax error by typing some meaningless letters and hitting RETURN. See Comment 4 of CLOSE for a discussion of how a file or device is “forgotten” and the consequences for cassette and disk files.

6. The CMD statement can be used to redirect output to the screen itself. This might seem pointless, but there are some differences in how information is displayed when the screen is accessed through a CMD statement. If CMD is used in program mode to “redirect” output to the screen, the question mark and space that the INPUT statement normally generates will not appear. However, any prompt supplied by the programmer in the INPUT statement will still appear. When the user presses RETURN to enter the value typed, the cursor will not move to a new line, but will simply move one space to the right. Also, if an INPUT statement requesting numeric data is instead given string data, the ?REDO FROM START message will not be given, but instead, the ?FILE DATA ERROR message will occur and program execution will stop. Use of the CMD statement in direct mode to “redirect” output to the screen has no advantages and many disadvantages, and therefore is not recommended.

```
10 OPEN 1,3 :REM DEVICE 3 IS THE SCREEN
20 CMD 1 :REM OUTPUT IS "REDIRECTED" TO THE SCREEN
30 INPUT "ENTER A NUMBER: ";A
40 IF A=INT(A) THEN PRINT "WHOLE NUMBERS
   ARE NOT VERY EXCITING."
50 PRINT#1: CLOSE 1
RUN
ENTER A NUMBER: 543 WHOLE NUMBERS ARE NO
T VERY EXCITING.

READY,
```

There are four ways that the execution of a program can be stopped prematurely. The operator can press the RUN/STOP key, one of the statements STOP or END can be encountered, or an error can occur. After each of these occurrences, the computer will be in direct mode. In each of the first three cases (that is, where an error has not occurred), the command

CONT

causes the execution of the program to continue. In the case of an error, entering the command CONT results in the message ?CAN'T CONTINUE ERROR.

Examples

```
1. 10 PRINT "SNOW"  
   20 PRINT "RAIN"  
   30 STOP  
   40 PRINT "SLEET"  
   50 PRINT "HAIL"  
   RUN  
   SNOW  
   RAIN
```

```
BREAK IN 30  
READY,  
CONT  
SLEET  
HAIL
```

```
READY,
```

```
2. 10 PRONT 2*10*55  
   20 PRINT 8*13*58  
   RUN
```

```
?SYNTAX ERROR IN 10  
READY,  
CONT
```

```
?CAN'T CONTINUE ERROR  
READY,
```

CONT

Comments

1. CONT causes execution to continue with the first statement after the point at which the RUN/STOP, STOP, or END occurred.

```
10 PRINT "WIND"  
20 STOP: PRINT "SUN"  
30 PRINT "FOG"  
RUN  
WIND  
  
BREAK IN 20  
READY,  
CONT  
SUN  
FOG  
  
READY,
```

Note that execution continued with the next statement after the STOP statement, not the next line.

2. After the execution of the program has stopped and the computer is in direct mode, you can display and change values of variables, make calculations and do most anything you like. However, if you alter a line of the program, add a line to the program, at any time hit the RETURN key on a line which begins with a number, or cause an error, then you cannot use CONT to resume execution of the program. If you try, the message ?CAN'T CONTINUE ERROR will be displayed. However, you still can continue by using a GOTO or GOSUB statement.

```
10 A = 30  
20 PRINT 20  
30 PRINT A  
40 STOP  
50 PRINT 50  
60 END  
70 PRINT 60  
RUN  
20  
30  
  
BREAK IN 40  
READY,  
A = 40
```

CONT

```
READY,  
CONT  
50
```

```
READY,  
30 PRINT A+100  
CONT
```

```
?CAN'T CONTINUE ERROR  
READY,
```

```
NEW
```

```
READY,  
10 A = 7654321  
20 B% = A  
30 PRINT B%  
RUN
```

```
?ILLEGAL QUANTITY ERROR IN 20  
READY,  
B% = 7654
```

```
READY,  
CONT
```

```
?CAN'T CONTINUE ERROR  
READY,
```

Applications

1. CONT is used in conjunction with STOP to debug programs. After the execution has been halted, the programmer knows the line number at which the break occurred and, while in direct mode, can have the value of any or all variables displayed in order to determine if the program is operating as it should. When satisfied, the programmer can use CONT to resume execution.

2. The CONT command can be used in conjunction with the END or STOP command to prevent information from scrolling off the screen. See Example 1 in the discussions of END or STOP for a specific illustration.

COS

COS is the trigonometric function cosine. For an acute angle in a right triangle, the cosine of the angle is the ratio:

$$\frac{\text{length of the side adjacent to the angle}}{\text{length of hypotenuse}}$$

The definition of the cosine function for arbitrary angles and a discussion of radian measure can be found in Appendix M. For any number X, the value of the function

`COS(X)`

is the cosine of the angle of X radians.

Examples

```
1. PRINT COS(1); COS(-5.678); COS(2E+8)
    .540302306 .822396688 -.740951125
```

```
READY,
```

```
2. 10 A% = 2: B = 7*A%: C = .643501109
   20 PRINT COS(A%); COS(B); COS(C)
   RUN
   -.416146836 .136737221 .8
```

```
READY,
```

Comments

1. Although X can be any number, COS(X) will always be between -1 and 1. Figure 1 contains the graph of $y = \text{COS}(X)$.

2. The X in COS(X) may be any numeric constant, variable, or expression.

```
A%=2: PRINT COS(3.2*A%+7)
    .672193084
```

```
READY,
```

3. The inverse of the cosine function is the function arccosine. This

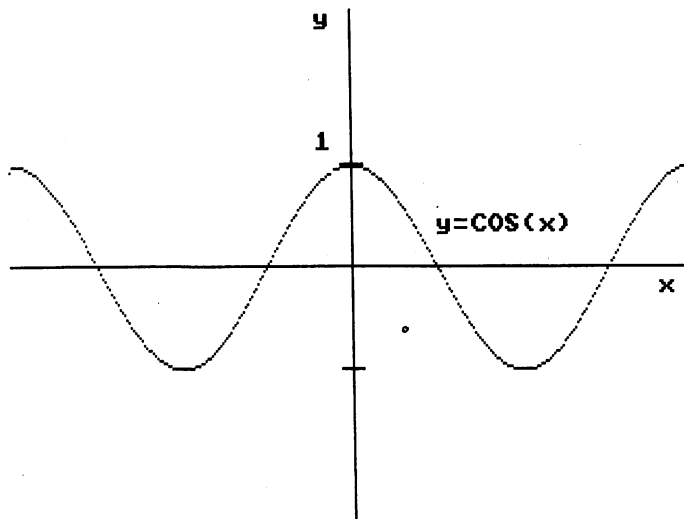


FIGURE 1

function is not available directly as a BASIC function. However, it can be defined in terms of ATN and SQR, which are BASIC functions.

$$\arccos(X) = \pi/2 - \text{ATN}(X/\text{SQR}(1-X*X))$$

Arccos(X) is the angle between 0 and π with cosine X.

```

10 DEF FNARCCS(X) =  $\pi/2$  - ATN(X/SQR(1-X*X))
20 A = COS(1): B = FNARCCS(A)
30 C = FNARCCS(.5): D = COS(C)
40 PRINT A; B; C; D
RUN
.540302306 1 1.04719755 .5

```

READY.

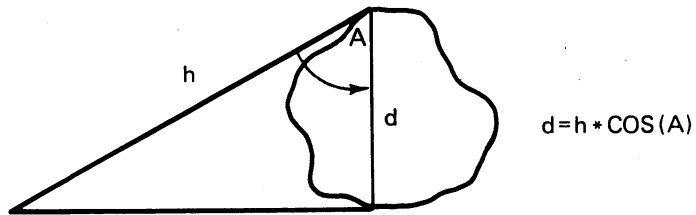
In general, for any number X between 0 and π , FNARCCS(COS(X)) is X, and for any number X between -1 and 1, COS(FNARCCS(X)) is X.

Note: Because we cannot use BASIC keywords within the names of variables and functions that we create, we could not use COS in naming the arccosine function.

Applications

1. Surveyors use COS to determine distances. For instance, the distance across the lake shown in Figure 2 is $H * \text{COS}(A)$.

COS



2. Certain periodic phenomena occurring in nature can be modeled with the cosine function. For instance, the tap water temperature (in degrees Fahrenheit) in Dallas, Texas, T days after the beginning of a year, is given approximately by the formula

$$59 + 14 * \text{COS}((T-208) * \pi / 183)$$

where T is between 0 and 365.

DATA statements are only used in conjunction with READ statements. READ statements call for the next item of a list of constants to be assigned to a variable, and DATA statements store the list. Consider the list

```
constant 1  
constant 2  
constant 3
```

where each entry is a numeric or string constant. (A numeric constant is a number and a string constant is a sequence of characters.) This list can be accessed during the execution of a program by including

```
DATA constant 1, constant 2, constant 3
```

as a line of the program. The line can appear anywhere in the program. It needn't precede the READ statement.

Examples

```
1. 10 DATA 7.8, GABRIEL, " AUGUST 23,1980"  
   20 READ A, B$, C$\br/>   30 PRINT A; B$; C$\br/>   RUN  
   7.8 GABRIEL AUGUST 23,1980  
  
READY.
```

```
2. 10 READ T$, Z  
   20 READ D, M  
   30 DATA ORANGEVALE, 95660, 9, 6, 1972  
   40 READ Y  
   50 PRINT T$; Z; D; M; Y  
   RUN  
   ORANGEVALE 95660 9 6 1972  
  
READY.
```

Comments

1. A BASIC program can contain many DATA statements. READ statements will begin by accessing the constants from the DATA statement having the lowest line number and then, after reading all of its items, will proceed to access the constants from the DATA statement with the next highest line number.

DATA

```
10 DATA 4, 34
20 READ A, B, C, D
30 PRINT A; B; C; D
40 DATA 17, 28
RUN
 4 34 17 28
```

READY.

2. Quotation marks surrounding a string constant are optional unless the string constant contains commas, colons, significant leading or trailing blanks, SHIFTed letters, graphics or cursor controls. (The data in line 10 of Example 1 had both a comma and a significant leading blank.) Surrounding quotation marks are not read by READ statements. Each string constant in the following DATA statements requires surrounding quotation marks.

```
10 DATA "12,345", "THE DEAL: ALL ACES!"
20 DATA " CENTER ", "♠♣♥♦"
```

If these DATA statements were written as

```
10 DATA 12,345, THE DEAL: ALL ACES!
20 DATA CENTER , ♠♣♥♦
```

the following would result: The string 12,345 would be read as the two strings 12 and 345. THE DEAL would be read as a string constant, but the colon would then indicate the end of the DATA statement, causing ALL ACES! to produce a ?SYNTAX ERROR IN 10 message. CENTER would be read as a six-character string without any leading or trailing blanks, and ♠♣♥♦ would be treated as “nothing”; if line 20 were listed, nothing would appear after the comma.

3. Quotation marks may be used as data within a string constant, provided that the string constant does not begin with a quote. Thus we are free to place quotes within a string constant being read from a DATA statement, as long as the constant is not to include characters such as commas, colons, SHIFTed letters, etc., which would require surrounding quotes.

```
10 DATA THE "BIG" CAT, QUOTES""
20 DATA "TRY THAT!" HE SAID
30 READ A$,B$
40 PRINT A$,B$
50 READ C$
```

DATA

```
RUN
THE "BIG" CAT      QUOTES""""
?SYNTAX ERROR IN 20
READY,
```

4. The maximum number of characters allowed in any BASIC program line is 80. You can include as many constants as you like in a single DATA statement, provided that the total number of characters in the statement doesn't exceed 80.

5. Numeric expressions (e.g., $3+4$ or $ABS(-2)$) and numeric variables cannot be used as numeric constants in DATA statements. Attempts to do so produce the error statement ?SYNTAX ERROR. Similarly, expressions involving strings (e.g., $LEFT$("FLORIDA",2)+"A"$) and string variables should not be used as string constants in DATA statements.

```
10 U$ = "UNITED STATES"
20 DATA U$
30 READ A$
40 PRINT A$
RUN
U$

READY,
```

6. In the following program an asterisk is the last data item and signals the end of the data. Such an item is referred to as a *trailer value*, *signal*, or *flag*.

```
10 READ A$
20 IF A$ = "*" THEN END
30 PRINT A$, :GOTO 10
40 DATA E, PLURIBUS, UNUM, *
RUN
E      PLURIBUS  UNUM
READY,
```

7. DATA statements are often used to provide values for special parameters within a program. In different runs of the program, some of these parameters may not be needed. It is possible to delete values for these parameters when typing up a new DATA statement and yet not alter the pairings of the variables in the READ statement with the

DATA

values in the DATA statement. To delete an item in a DATA statement without also deleting the corresponding variable in the READ statement, delete the value but leave a comma following the space previously occupied by the value. The corresponding variable will be assigned the value zero if it is numeric, and the empty string if it is a string variable.

```
10 DATA 3,14,,92,MONTH,,YEAR
20 READ A, B, C, D, E$, F$, G$
30 PRINT A, B, C, D
40 PRINT E$, F$, G$
RUN
  3          14          0          92
MONTH                      YEAR

READY.
```

8. See the discussions of READ and RESTORE for further details about DATA statements.

Numeric formulas occur often in programming applications. An example might be $4.9 * T * T$, which is the distance, in meters, that an object will have fallen T seconds after it was dropped. Another example might be $5050 * \text{EXP}(N * \text{LOG}(1.01)) - 5000$, which is the balance after N months in an annuity earning 12% annual interest compounded monthly, with monthly deposits of 50 dollars. Both of the formulas above involve a variable whose value must be given before the formula can be evaluated. In the first formula the variable is T ; in the second, it is N . In mathematics, formulas are often given names and are then referred to by these names followed in parentheses by the variable needed to evaluate the formula. For the above formulas we might have names like $\text{FALL}(T)$ and $\text{BALANCE}(N)$.

Names for formulas are referred to as functions. BASIC has several built-in functions, such as EXP , LOG , SIN , INT , and SQR . In addition, we may define our own functions; that is, we can give a complicated formula a simple name and use this name in place of the formula throughout our program. If *name* is a valid variable name, X a numeric variable, and *rule* a formula whose value depends on the value of X , then the statement

```
DEF FNname(X)=rule
```

defines the function $\text{FNname}(X)$. The value of $\text{FNname}(A)$ will be the value of the formula *rule* when X is given the value A . Functions for the two formulas above would be defined as follows:

```
DEF FNFALL(T)=4.9*T*T
DEF FNBALANCE(N)=5050*EXP(N*LOG(1.01))-5000
```

Examples

1.

```
10 DEF FNDOUBLE(X)=2*X
20 PRINT FNDOUBLE(8)
RUN
16

READY.
```
2.

```
10 DEF FNCUBE(Z)=Z*Z*Z
20 PRINT FNCUBE(3)
30 A=2: PRINT FNCUBE(A)
```

DEF FN

```
RUN
 27
  8
```

READY.

```
3. 10 DEF FNTWO(M)=SGN(M)*INT(ABS(M)*100+.5)/100
    20 REM FNTWO ROUNDS M TO TWO DECIMAL PLACES.
    30 PRINT FNTWO(12.3456)
    40 A=73.3478: B=237.832
    50 PRINT FNTWO(A);FNTWO(A)+FNTWO(B)
RUN
 12.35
 73.35  311.18
```

READY.

Comments

1. Function names are formed using the letters FN followed by a floating-point variable name. That is, function names cannot end with the suffix % which would be used for integer variables. If it is desired to have a function return an integer value, the function INT should be used as part of the formula for the function.

```
10 DEF FNA%(X)=X/3
RUN
```

```
?SYNTAX  ERROR IN 10
READY.
```

```
10 DEF FNA(X)=INT(X/3)
20 INPUT K
30 PRINT FNA(K)
RUN
? 8
  2
```

READY.

2. The X in DEF FNA(X) is called the formal argument of the function. Only a single formal argument may be used in a DEF FN statement, and this formal argument must be a floating-point numeric variable. That is, a formal argument may not have the % suffix. When a function is used within a program, the formal argument is replaced by a numeric constant or variable. If a numeric variable is used, it may be an integer variable with the % suffix.

DEF FN

```
10 DEF FNA(L%)=L% * 6  
RUN
```

```
?SYNTAX ERROR IN 10  
READY.
```

```
10 DEF FNA(L)=L*6  
20 INPUT E%  
30 PRINT FNA(E%)  
RUN  
? 12  
72
```

```
READY.
```

3. The name of the formal argument in a DEF FN statement is just a dummy variable. Changing this name does not change the function. For instance, the statement `DEF FNCUBE(NUMBER) = NUMBER*NUMBER*NUMBER` has the same effect as `DEF FNCUBE(X) = X*X*X`. Also, the name of a formal argument may be the same as the name of another constant that occurs elsewhere in the program.

```
10 Y = 5: M = 4  
20 DEF FNA(Y) = Y*Y  
30 PRINT FNA(3); FNA(Y); FNA(M)  
RUN  
9 25 16
```

```
READY.
```

When execution of the program reached line 20, Y was treated as a dummy variable with no meaning whatsoever except as a placeholder in defining the function A. However, in line 30, Y was used as it was designated in line 10.

4. When defining a function, variables with names other than the argument can appear in the rule.

```
10 Y = 5  
20 DEF FNA(X) = X+Y  
30 PRINT FNA(2)  
RUN  
7
```

```
READY.
```

DEF FN

5. If an error is typed into a DEF FN statement, the error is not detected until the function is called. The error is then identified as having occurred in the calling line of the program, not in the line of the DEF FN statement. Consider the following program in which a right parenthesis was omitted in line 10.

```
10 DEF FNA(X) = SIN(X
20 PRINT FNA(1)
RUN
```

```
?SYNTAX ERROR IN 20
READY.
```

6. The definition of a function may make use of BASIC's built-in functions or user-defined functions which previously have been established by a DEF FN statement.

```
10 DEF FN(A) = SIN(A)*COS(A)
20 DEF FNXX(A) = FN(A)*FN(1/A)
30 INPUT A
40 PRINT FNXX(A)
RUN
? 2
-.159206835
```

```
READY.
```

It is not possible, however, to use the function being defined as part of the definition of the function. Attempting to do so will lead to an ?OUT OF MEMORY ERROR message.

```
10 DEF FNY(A)=2*A*FNY(A-1)
20 PRINT FNY(3)
RUN
```

```
?OUT OF MEMORY ERROR IN 20
READY.
```

7. The DEF FN statement cannot be used in direct mode. Attempting to do so results in the ?ILLEGAL DIRECT ERROR message.

8. If the DEF FN statement defining a function is omitted, and the function name still is used, the message ?UNDEF'D FUNCTION ERROR results.

9. Functions defined by the user in a program will retain their defi-

DEF FN

inition after program execution ends. These definitions are lost when a program line is added or edited, or when one of the commands CLR, LOAD, NEW, or RUN is executed.

Applications

1. Whenever a function occurs more than once in a program, it may be efficient to define it with a DEF FN statement early (i.e., before it is first used) in the program. The following program is used to find the zeros of a function using the bisection method. The function is specified in line 10. Line 20 specifies a second function which returns, for a given value of X, the sign of the function of line 10. The INPUT statement in line 30 asks for two numbers, A and B, such that the function has values of opposite signs at A and B.

```
10 DEF FNG(X) = X*X*X+X-50
20 DEF FNSG(X) = SGN(FNG(X))
30 INPUT A,B
40 IF FNSG(A) = FNSG(B) GOTO 30
50 S = FNSG(B)
60 C = .5*(A+B)
70 IF ABS(FNG(C))<.0000001 THEN PRINT C: END
80 IF S*FNG(C)<0 THEN A = C: GOTO 50
90 B = C: GOTO 50
RUN
? 2, 5
  3.59356955

READY.
```

DIM

A variable is a name to which the computer can assign a single value. An array variable is a name to which the computer can assign an entire collection of values. The values are thought of as being organized in an array. Examples of arrays are:

<u>Array B</u>		<u>Array P</u>			
563.00	0	0	0	0	0
452.63	0	1	2	3	4
341.16	0	2	4	6	8
228.57	0	3	6	9	12
114.85					
0.00					

Array B is a one-dimensional array. The values are successive balances on a loan of \$563. (The loan, at 12% interest compounded monthly, is paid off with 5 monthly payments of \$116.) If $B(I)$ is the balance after I months, then $B(0) = 563$, $B(1) = 452.63$, $B(2) = 341.16$, $B(3) = 228.57$, $B(4) = 114.85$, and $B(5) = 0$. The array variable B is also referred to as a subscripted variable with subscripts ranging from 0 to 5. The statement

`DIM B(5)`

establishes the number of elements allowed for this array variable and sets aside space in memory to store the values.

A single column of values is said to form a one-dimensional array. If the values are numbered from 0 to N , and a name is chosen for the array variable to hold these values, then a statement of the form

`DIM arrayname(N)`

is used to establish the number of values allowed for the array variable. Space is set aside for $N + 1$ values referred to as $arrayname(0)$, $arrayname(1)$, . . . , $arrayname(N)$.

The rectangular array P above is a two-dimensional array. The values form part of a multiplication table. Think of the rows as being labeled 0, 1, 2, 3 and the columns as being labeled 0, 1, 2, 3, 4. Let $P(R,C)$ be the entry in the R th row and C th column, which in this example is the product of the numbers R and C . For instance, $P(0,0) = 0$, $P(2,3) = 6$, and $P(3,1) = 3$. The array variable P is sometimes referred to as a double-subscripted variable with the first subscript ranging from 0 to 3 and the second subscript ranging from 0 to 4. The statement

DIM

DIM P(3,4)

specifies the size of this array variable and sets aside space in memory to store its values.

Any rectangular array of values is said to form a two-dimensional array. If the rows are numbered from 0 to M, the columns numbered from 0 to N, and a name chosen for the array variable to hold these values, then the statement

DIM *arrayname*(M,N)

is used to specify the size of the array variable. Space is set aside for $(M + 1) * (N + 1)$ values. The value in the Rth row and Cth column will be referred to as *arrayname*(R,C). (Don't forget that there is a 0th row and a 0th column.)

Three-dimensional or higher arrays cannot be easily written down; however, they can be specified. A three-dimensional array variable (or triple-subscripted variable) is specified by a statement of the form

DIM *arrayname*(M,N,P)

where the first subscript ranges from 0 to M, the second from 0 to N, and the third from 0 to P. (The value for which the first subscript is R, the second is C, and the third is L is referred to as *arrayname*(R,C,L).)

Examples

1. Consider the one-dimensional array B presented at the beginning of this discussion. The following program assigns the given data to the array variable and provides access to this data.

```
10 DIM B(5)
20 FOR I = 0 TO 5
30 READ B(I)
40 NEXT I
50 DATA 563, 452.63, 341.16
60 DATA 228.57, 114.85, 0
70 INPUT "NUMBER OF MONTHS"; M
80 PRINT "THE BALANCE AFTER"; M;
90 PRINT "MONTHS IS";B(M)
RUN
NUMBER OF MONTHS? 3
THE BALANCE AFTER 3 MONTHS IS 228.57

READY.
```

DIM

2. The chart below assigns personnel in a small business to certain tasks for each day of the week.

	<u>Sun</u>	<u>Mon</u>	<u>Tue</u>	<u>Wed</u>	<u>Thu</u>	<u>Fri</u>	<u>Sat</u>
Open-up	BOB	AL	TOM	SUE	CARL	JAN	KEN
Clean-up	TOM	CARL	SUE	BOB	AL	KEN	JAN
Lock-up	AL	BOB	CARL	JAN	KEN	SUE	TOM

Think of the three tasks as numbered 1, 2, and 3, and think of the days of the week as numbered 1 to 7. The following program assigns this information to an array variable and provides access to the information.

```
10 DIM WHO$(3,7)
20 FOR I = 1 TO 3
30 FOR J = 1 TO 7
40 READ WHO$(I,J)
50 NEXT J
60 NEXT I
70 DATA BOB, AL, TOM, SUE, CARL, JAN
80 DATA KEN, TOM, CARL, SUE, BOB, AL
90 DATA KEN, JAN, AL, BOB, CARL, JAN
100 DATA KEN, SUE, TOM
110 INPUT "NUMBER OF TASK, DAY"; A, B
120 PRINT WHO$(A,B);" IS ASSIGNED TASK ";
130 PRINT A;" ON DAY ";B
RUN
NUMBER OF TASK, DAY? 1,3
TOM IS ASSIGNED TASK 1 ON DAY 3

READY,
```

Note that we did not make use of the zeroth row or column in the array WHO\$. Since we did not need to worry about having sufficient memory, we chose to start counting from 1 rather than 0 to make the workings of the program easier to follow.

Comments

1. An array variable must be one of two types: numeric or string. There is no such thing as a mixed array variable. That is, the values must be either all numeric or all string constants.

2. The rules for naming array variables are the same as the rules for naming ordinary variables. In particular, names ending in \$ refer to

DIM

string array variables, and names ending in % refer to integer array variables. Integer array variables use much less space than floating-point array variables. (See Example 2 in the discussion of FRE.)

3. A single DIM statement can specify the sizes of several array variables. For instance, the statement

```
DIM A(25), B$(3,7)
```

allocates space for a numeric array variable named A with 26 values referred to as A(0) to A(25), and a two-dimensional string array variable named B\$ with 32 values referred to as B\$(0,0) to B\$(3,7).

4. Using a subscript outside of the specified range results in the message ?BAD SUBSCRIPT ERROR.

```
10 DIM PRICE%(15)
20 FOR I = 0 TO 15
30 PRICE%(I) = 5*I + 20
40 NEXT I
50 PRICE%(25) = 123
RUN

?BAD SUBSCRIPT ERROR IN 50
READY.
```

5. All array variables will be cleared from memory if one of the commands CLR, RUN, NEW, or LOAD is executed, or if a new program line is entered, an old program line edited (and entered), or the RETURN key typed with the cursor on any line which begins with a number.

6. Once an array variable has been DIMensioned, we cannot change the range of the subscripts. Within a program, our only recourse is to clear all arrays, variables, etc. with the CLR statement and then state the new range with another DIM statement. Employing a second DIM statement without first CLearRing results in the message ?REDIM'D ARRAY ERROR.

7. A one-dimensional array variable can be used without being preceded by a DIM statement to specify its size, provided that the subscript appearing ranges from 0 to 10. If so, the array automatically will be assumed to have a maximum subscript of 10. Thus, in Example 1, we could have omitted line 10. However, it is a good idea to DIMen-

DIM

sion all array variables. Doing so saves space in memory and makes the program easier for others to follow.

```
10 A(8) = 123.45
20 DIM A(7)
30 FOR I = 0 TO 7
40 A(I) = I + 10
50 NEXT I
RUN

?REDIM'D ARRAY  ERROR IN 20
READY.
```

Line 10 assigned a value to the array variable A without first using a DIM statement. The computer automatically DIMensioned A with a range from 0 to 10, as if the statement DIM A(10) had been supplied. Therefore, line 20 is attempting to reDIMension A, which is not allowed.

8. An array variable of two or more dimensions may also be used before it is DIMensioned, in which case each subscript will be assumed to range from 0 to 10. For example, if the statement BOX(1,1,1)=4 is executed before a DIM statement for BOX occurs, then BASIC automatically executes the statement DIM BOX(10,10,10).

9. It is easy to DIMension an array so large that it uses all available memory. For example, the statement DIM A(20,20,20) results in the message ?OUT OF MEMORY ERROR. The array A requires over 40000 bytes of memory, which exceeds the 38900 or so bytes available for both variables and programs! The following table gives the information needed to calculate in advance just how much memory an array will require.

<u>DIM Statement</u>	<u>Bytes of Memory Required</u>
DIM A(J)	$7 + 5*(J + 1)$
DIM A%(J)	$7 + 2*(J + 1)$
DIM A\$(J)	$7 + 3*(J + 1) + \text{number of characters which will be stored}$
DIM A(J,K)	$9 + 5*(J + 1)*(K + 1)$
DIM A%(J,K)	$9 + 2*(J + 1)*(K + 1)$
DIM A\$(J,K)	$9 + 3*(J + 1)*(K + 1) + \text{number of characters which will be stored}$

DIM

DIM A(J,K,L)	$11 + 5*(J+1)*(K+1)*(L+1)$
DIM A%(J,K,L)	$11 + 2*(J+1)*(K+1)*(L+1)$
DIM A\$(J,K,L)	$11 + 3*(J+1)*(K+1)*(L+1) +$ number of characters which will be stored

For example, the statement DIM TAX(20,30) is of the form DIM A(J,K), and so requires $9 + 5*(20+1)*(30+1)$, or 3264 bytes. The rules for arrays with 4 or more dimensions follow the same pattern as established in the above rules for one, two, and three-dimensional arrays.

10. The parameters M, N, and P in a DIM statement may be numeric constants, variables, or expressions. In addition, they may be of either integer or floating-point precision. If they are not whole numbers, their truncated values are used to establish the size of the array. For example,

```
10 INPUT S,T: DIM B(8*S+T)
20 INPUT B$: DIM A$(LEN(B$))
30 INPUT B(20), A$(12)
40 PRINT B(20); A$(12)
50 INPUT B(21)
RUN
? 2,4,8
? INTERPLANETARY
? 21314,ERHS
  21314 ERHS
? 7601

?BAD SUBSCRIPT  ERROR IN 50
READY.
```

END

Traditionally, the last executable line in a BASIC program consists of the statement END to indicate that execution is complete. Programs for the Commodore 64 do not have to end with END. However, the END statement can be used for other purposes.

Including the statement

```
END
```

as a line of a program causes the program to stop execution at that line. The prompt "READY." is displayed, and the computer returns to direct mode. By entering the command CONT, execution can be resumed with the first statement following the END statement.

Examples

1. The following program will display the balance after each month (for 360 months) for a \$100,000 mortgage at 12% interest compounded monthly. Upon running the program you will see the balances for the first 12 months, followed by READY. If you then type CONT and press the RETURN key, the next 12 months will be displayed, and so on.

```
10 FOR I = 1 TO 359 STEP 12
20 FOR J = 0 TO 11
30 B = 102861.23-2861.23*1.01↑(I+J)
40 PRINT I+J,INT(100*B)/100
50 NEXT J
60 END
70 NEXT I
```

Without the END statement, the first 338 months would have flickered by too fast to be read and only the last 22 months would have been clearly displayed.

Comments

1. After the execution has been ENDED, there are four options for continuing: CONT, GOTO, GOSUB, and RUN. Example 1 shows how the CONT command is used to cause the program to continue execution with the statement after END. The statements

GOTO *m* and GOSUB *m*

cause execution to resume at line *m*. The RUN command reruns the program from the beginning.

END

2. After the program ENDS and the computer is in direct mode, you can display and change values of variables, make calculations, or do most anything. However, if you alter or add a line to the program, type the RETURN key on a line beginning with a number, or cause an error message to appear, you cannot use CONT to resume execution of the program. You can continue, however, by using a GOTO or GOSUB statement.

```
10 A = 30
20 PRINT 20
30 PRINT A
40 END
50 PRINT 50
RUN
  20
  30
```

```
READY,
A = 40
```

```
READY,
GOTO 30
  40
```

```
READY,
CONT
  50
```

```
READY,
RUN
  20
  30
```

```
READY,
30 PRINT A+100
CONT
```

```
?CAN'T CONTINUE  ERROR
READY,
```

3. The STOP statement is similar to the END statement. Both cause the program to stop execution and can be followed by CONT, GOTO, GOSUB, or RUN to resume execution. The only difference between STOP and END is that STOP causes a BREAK IN *n* message to be displayed.

END

Applications

1. The END statement can be used to prevent scrolling until the user has had an opportunity to read the screen. When the user is ready he just enters the CONT command. Example 1 provides an illustration of one way to use the END statement to chop up a long list of data into manageable pieces.
2. Programmers normally insert an END statement before a subroutine to guarantee that the subroutine will only be executed as the result of a GOSUB statement.
3. The END statement can be used to have two programs in memory at the same time. See Application 1 in the discussion of RUN for details.

An exponential function is a function of the form

$$B^X$$

The number B is called the base of the function. The most important exponential function is the one having as base a special number known as “ e ”. The value of “ e ”, to 9 significant digits, is 2.71828183. For any number X ,

$$\text{EXP}(X)$$

has the value e^X .

Examples

```
1. PRINT EXP(1); EXP(-2,345); EXP(3E-2)
    2,71828183 ,0958472021  1,03045453
```

```
READY,
```

```
2. 10 A = 5: B = 3*A
    20 PRINT EXP(A); EXP(B); EXP(A-2)
    RUN
    148,413159  3269017,37  20,0855369
```

```
READY,
```

Comments

1. Although X can be any number, the value of $\text{EXP}(X)$ will always be a positive number. Figure 1 contains the graph of $y = \text{EXP}(X)$.
2. The X in $\text{EXP}(X)$ may be any numeric constant, variable, or expression, and may be of either integer or floating-point precision.
3. EXP is the inverse of the natural logarithmic function, LOG . For any X , $\text{EXP}(X)$ is the number whose LOG is X . That is, for any number X , the value of $\text{LOG}(\text{EXP}(X))$ is X , and for any positive number X , the value of $\text{EXP}(\text{LOG}(X))$ is X .

```
10 A = EXP(3,2): B = LOG(A)
20 C = LOG(1234): D = EXP(C)
30 PRINT A; B; C; D
```

EXP

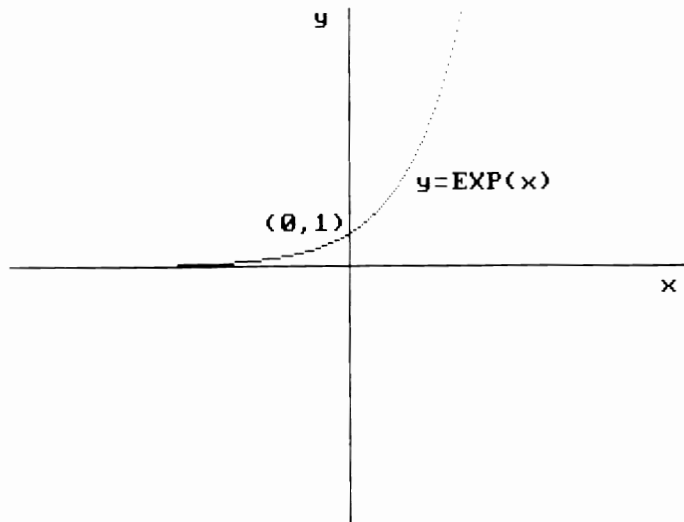


FIGURE 1

```
RUN
 24.5325302  3.2  7.11801621  1234

READY.
```

4. Any other exponential function can be expressed in terms of EXP and LOG. For any number X, the value of B^X is $\text{EXP}(X \cdot \text{LOG}(B))$.

```
10 INPUT "BASE";B
20 INPUT "EXPONENT";X
30 PRINT:PRINT B;"RAISED TO THE";X
40 PRINT "IS";EXP(X*LOG(B))
RUN
BASE? 7
EXPONENT? 3

 7 RAISED TO THE 3
IS 343

READY.
RUN
BASE? 3.4
EXPONENT? 5.6

 3.4 RAISED TO THE 5.6
IS 946.851642

READY.
```

EXP

5. The number $1.70141183E + 38$ is often referred to as *positive machine infinity*. The EXP function grows so fast that it reaches positive machine infinity when $X = 88.0296920$. For $X = 88.0296920$ or higher, asking for EXP(X) results in the ?OVERFLOW ERROR message being displayed.

6. The value of the function EXP(X) approaches 0 as X becomes a negative number of larger and larger magnitude. The limits of the computer's floating-point representation of numbers cause the value of EXP(X) to be 0 for all values of X which are less than or equal to -88.0296920 .

Applications

1. If \$1000 is invested at an interest rate of 12% compounded continuously, then the balance after T years is given by the formula

$$1000 * \text{EXP}(.12 * T)$$

In general, if P dollars is invested at interest rate R (compounded continuously), then the balance after T years is given by the formula

$$P * \text{EXP}(R * T)$$

2. The decay of radioactive elements is described by the function EXP. For instance, if you start with 2 grams of Strontium-90, the amount present after T years is given by the formula

$$2 * \text{EXP}(-.0244 * T)$$

3. The normal curve of probability has the equation

$$Y = 1 / (S * \text{SQR}(2 * \pi)) * \text{EXP}(-.5 * ((X - M) / S) \uparrow 2)$$

where S is the standard deviation and M is the mean.

FOR and NEXT

Many programming applications require that a set of statements be executed several times. Rather than having a user write the same set of statements over and over again, BASIC provides two statements, FOR and NEXT, between which we can place the set of statements that we wish to have repeated. If I is a numeric variable and N is a positive whole number, then a sequence of program lines of the form

```
100 FOR I=1 TO N
    .
    . (lines to be repeated)
    .
200 NEXT I
```

will cause the lines between 100 and 200 to be executed N times. (The numbers 100 and 200 are used here only for illustration.) The variable I is referred to as the *index* of the FOR . . . NEXT loop.

```
10 FOR COUNT=1 TO 9
20 PRINT "WE'RE #1 ";
30 NEXT COUNT
RUN
WE'RE #1 WE'RE #1 WE'RE #1 WE'RE #1
WE'RE #1 WE'RE #1 WE'RE #1 WE'RE #1
WE'RE #1
READY.
```

In some applications, we are not so concerned with how many times a loop is repeated, but rather with repeating a set of statements in which a variable assumes each value in a certain sequence of values. For example, we may wish to repeat a set of statements and let the variable I assume the values 1970, 1971, 1972, . . . , 1984 successively. If I is any numeric variable, and A and B are any two whole numbers, with A less than or equal to B, then a sequence of program lines of the form

```
100 FOR I=A TO B
    .
    .
    .
200 NEXT I
```

will cause all lines between 100 and 200 to be executed B-A + 1 times.

FOR and NEXT

More importantly, the lines between 100 and 200 are executed once with $I=A$, once with $I=A+1$, once with $I=A+2$, etc., and once with $I=B$. Execution then continues with the first line after line 200.

```
10 FOR YEAR=1970 TO 1984
20 PRINT YEAR,
30 NEXT YEAR
RUN
 1970      1971      1972      1973
 1974      1975      1976      1977
 1978      1979      1980      1981
 1982      1983      1984
READY,
```

So far, we have only used whole numbers as the starting and stopping values for loops. It also is possible to use floating-point numbers or numeric variables for these purposes. Further, the index in all cases above was incremented by 1 each time NEXT was encountered. In general, it is possible to add any fixed amount to the index when NEXT is encountered. If A , B , and C are any numeric values or variables, then a sequence of program lines of the form

```
100 FOR I=A TO B STEP C
    .
    .
    .
200 NEXT I
```

will execute the enclosed lines according to the following rules:

- (a) The index I is assigned the value A and the enclosed set of lines is executed once.
- (b) The value of I is incremented by C ; that is, I is given the value $I+C$.
- (c) If the value of I is greater than B , then execution continues with the first line after line 200. Otherwise, the enclosed set of lines is executed again with the new value of I . Rules (b) and (c) are repeated until the value of I is greater than B .

As a further embellishment, a FOR . . . NEXT loop can have its index steadily decreased. If C has a negative value, then the phrase “greater than” in rule (c) is replaced by “less than”.

Examples

```
1. 10 PRINT "OUTFIELD OF '27 YANKEES: " ;
```

FOR and NEXT

```
20 FOR J=1 TO 3
30 READ A$
40 PRINT A$+" ";
50 NEXT J
60 DATA MEUSEL, COMBS, RUTH
RUN
OUTFIELD OF '27 YANKEES: MEUSEL COMBS RU
TH
READY.
```

```
2. 10 INPUT A
20 FOR T=A TO 2
30 PRINT T; T*T,
40 NEXT T
RUN
? -1
-1 1 0 0 1 1 2 4
READY.
RUN
? .5
.5 .25 1.5 2.25
READY.
```

(The last time that line 40 was executed, the value 2.5 was assigned to T. Since $2.5 > 2$, the loop was then terminated.)

```
3. 10 FOR X=0 TO 1.6 STEP .5
20 PRINT X; SQR(X),
30 NEXT X
RUN
0 0 .5 .707106781 1 1
1.5 1.22474487
READY.

4. 10 FOR F=74 TO 10 STEP -8
20 PRINT F;
30 NEXT F
RUN
74 66 58 50 42 34 26 18 10
READY.
```

Comments

1. The index in a FOR ... NEXT statement must be a floating-point variable; that is, the variable cannot end with the % character to make

FOR and NEXT

it an integer variable. Attempting to use an integer variable for the index results in the message ?SYNTAX ERROR.

```
10 FOR A%=1 TO 5
20 PRINT A%
30 NEXT A%
RUN
```

```
?SYNTAX ERROR IN 10
READY.
```

2. If the NEXT statement is accidentally omitted, the computer will not notify you of your error. Instead, the FOR statement which is missing the NEXT will execute only once, with the index assigned the starting value of the FOR statement.

```
10 FOR J=100 TO 1000 STEP 10
20 PRINT 5*J
30 FOR I=1 TO 3
40 PRINT I;
50 NEXT I
RUN
500
1 2 3
READY.
```

3. Within a single FOR I . . . NEXT I loop, the statement NEXT I may appear more than once. For instance, the following program avoids displaying the unlucky number 13.

```
10 INPUT A
20 FOR I=A to A+3
30 IF I=13 THEN NEXT I
40 PRINT I;
50 NEXT I
RUN
? 11
11 12 14
READY.
```

The above program is equivalent to

```
10 INPUT A
20 FOR I=A TO A+3
30 IF I=13 THEN GOTO 50
```

FOR and NEXT

```
40 PRINT I ;  
50 NEXT I
```

4. Every FOR . . . NEXT loop is equivalent to a loop using IF and GOTO statements. For example, the two programs below are equivalent.

```
10 FOR I=2 TO 6           10 I=2  
20 PRINT I ;             20 PRINT I ;  
30 NEXT I                 30 I=I+1  
                           40 IF I<=6 GOTO 20
```

However, FOR . . . NEXT loops are preferable, since they execute faster and are easier for a programmer to decipher than IF and GOTO statements.

5. The name of the index variable usually can be omitted in the NEXT statement. For instance, in Example 1, line 50 of the program could have been 50 NEXT.

The index is only essential if branching might take place inside two FOR . . . NEXT loops. However, it is very poor programming to write a program which uses GOTO statements to branch in and out of FOR . . . NEXT loops, and thus a well-structured program will not require indices with NEXT statements. As an example of a poorly written program which does require that indices be given with the NEXT statement, consider the following program:

```
10 FOR I=1 TO 2  
20 PRINT I ;  
30 GOTO 60  
40 NEXT I  
50 END  
60 FOR J=10 TO 12  
70 PRINT J ;  
75 IF J=11 GOTO 40  
80 NEXT J  
90 GOTO 40  
RUN  
1 10 11 2 10 11  
READY.
```

If line 40 is changed to 40 NEXT, the outcome will be

FOR and NEXT

```
RUN
 1 10 11 12 2 10 11 12
READY.
```

When NEXT (with no index) is encountered, it is associated with the FOR statement that has been most recently executed and not yet completed. Thus, when line 75 sends the program to line 40, the most recent FOR statement which is incomplete is line 60. When line 90 is reached, the J FOR . . . NEXT loop has been completed. Thus, when line 90 sends the program to line 40, the most recent FOR statement that is incomplete is line 10. The lines are executed in the following order:

```
10, 20, 30, 60, 70, 75, 80, 70, 75, 40, 70, 75, 80, 90, 40,
20, 30, 60, 70, 75, 80, 70, 75, 40, 70, 75, 80, 90, 40, 50
```

(What a mess!)

6. FOR . . . NEXT loops execute fastest when the index variable is omitted from the NEXT statement. Consider these programs:

```
10 FOR I=1 TO 30000          10 FOR I=1 TO 30000
20 NEXT I                    20 NEXT
```

The program on the left takes 46 seconds to run, whereas the program on the right takes 37 seconds. (The same loop written using IF and GOTO statements requires over 5 minutes to run!)

7. It is not uncommon for programs to contain time-consuming FOR . . . NEXT loops. To keep the user from thinking that the program has crashed, it is a good idea for the programmer to insert an assuring PRINT statement before entering the loop. Some possibilities are "COMPUTING. PLEASE WAIT" or "SEARCHING FOR ACCOUNT. SHORT DELAY."

8. A FOR . . . NEXT loop always executes at least once. Consider a FOR . . . NEXT loop beginning with a statement of the form FOR I=A TO B STEP C. Even if A is greater than B, with C positive, or if B is greater than A, with C negative, the lines inside the loop will be executed once, and the program will branch to the line after the line containing the NEXT statement. If C is zero, an infinite loop will be produced, unless A equals B.

```
10 A=20: B=10: C=2
20 FOR I=A TO B STEP C
30 PRINT I
```

FOR and NEXT

```
40 NEXT I
RUN
20
```

READY.

9. When STEP C is included in a FOR statement, and the value of C has a fractional part, the computer may not be able to store the value precisely. In such cases, a rounding error will result that may cause the FOR ... NEXT loop to perform differently than intended. However, precautions can be taken to guarantee the desired result.

```
10 FOR I=1 to 2 STEP .1
20 PRINT I;
30 NEXT
RUN
1 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1
.8 1.9
READY.
```

We expected the last number PRINTed to be 2. This can be achieved by changing line 10 to read 10 FOR I=1 to 2.05 STEP .1. In general, when working with values of C which include a fraction, it is best to give an ending value which is $C/2$ more than the actual value at which you wish the loop to end.

10. The value of the index may be altered within a FOR ... NEXT loop:

```
10 FOR I=1 TO 20
20 PRINT I;
30 IF I=3 THEN I=17
40 NEXT I
RUN
1 2 3 18 19 20
READY.
```

If the starting and stopping values and the increment are also given by variables, the values of these variables may be changed inside the loop without affecting the execution of the loop. This is because the starting and stopping values and the increment are stored in special memory location when the FOR statement is first encountered.

```
10 L=5: S=2
20 FOR I=1 TO L STEP S
30 PRINT I;
```

FOR and NEXT

```
40 L=3: S=1
50 NEXT I
RUN
  1  3  5
READY.
```

11. A FOR ... NEXT loop can contain another FOR ... NEXT loop inside it. However, the second loop must be completely inside the first loop. Schematically:

```

      first loop {
        FOR I=A TO B
        :
        FOR J=D TO E } second loop
        :
        NEXT J
        :
        NEXT I

```

Such loops are referred to as *nested* loops.

The following program is used to alphabetize the members of a baseball team.

```
10 FOR I = 1 TO 9: READ A$(I): NEXT
20 FOR I = 1 TO 8
30 FOR J = 1 TO 8
40 IF A$(J)<=A$(J+1) THEN 80
50 T#=A$(J)
60 A$(J)=A$(J+1)
70 A$(J+1)=T#
80 NEXT J
90 NEXT I
100 FOR I = 1 TO 9:PRINT A$(I)+" ";:NEXT
110 DATA COMBS, KOENIG, RUTH, GEHRIG
120 DATA MEUSEL, LAZZERI, DUGAN
130 DATA COLLINS, HOYT
RUN
COLLINS COMBS DUGAN GEHRIG HOYT KOENIG L
AZZERI MEUSEL RUTH
READY.
```

The program contains 4 FOR ... NEXT loops, with the third loop nested inside the second.

If the statements NEXT J and NEXT I in lines 80 and 90 of the program above were interchanged, the sequence of statements 20

FOR and NEXT

through 90 would not constitute valid nested loops. Consider the following program:

```
10 FOR I=1 TO 5
20 PRINT I;
30 FOR J=6 TO 10
40 PRINT J;
50 NEXT I
60 NEXT J
RUN
 1 6 2 6 3 6 4 6 5 6
?NEXT WITHOUT FOR ERROR IN 60
READY.
```

When line 50 is encountered, the FOR . . . NEXT loop started in 30 is terminated, even though only the value 6 has been used for J. After the FOR . . . NEXT loop indexed by I is completed, execution proceeds to line 60. But no FOR loop for J is active, and thus the error message.

12. Consider the first program in Comment 11. Since the lines consisting of NEXT J and NEXT I are adjacent to each other, they can be combined into a single line consisting of the statement

```
NEXT J,I
```

(Here, the order of the letters J and I cannot be reversed.) If there were a third FOR . . . NEXT loop with index variable K nested inside the second loop, and the statement NEXT K directly preceded the other two NEXT statements, the three statements could be combined into NEXT K,J,I, and so on.

13. A FOR . . . NEXT loop is *active* if the loop has not yet experienced a natural termination. A FOR . . . NEXT loop terminates naturally when the index, having been incremented by NEXT, exceeds the ending value in the FOR statement. Whenever a FOR . . . NEXT loop is active, the computer keeps track of the loop, using a special portion of memory often referred to as BASIC's *stack*. This stack is limited in size, and must also be used to keep track of the return addresses for GOSUB statements. So that stack space is not wasted, it is important that each FOR . . . NEXT loop be "deactivated" when it is no longer needed.

⌘ A FOR . . . NEXT loop deactivates automatically when it terminates normally, or when a new loop is begun which uses the same variable for the index as was used by the old loop. In those cases where a FOR . . . NEXT loop is exited early, it is a good idea not to rely on a later

FOR and NEXT

loop to cause deactivation. The two programs below include the possibility of an early exit. The first program does not deactivate the loop for an early exit. On the other hand, the second program uses a simple technique to make an early exit and deactivate the loop.

```
10 FOR I=1 TO 100
20 IF I*I>1000 THEN 50
30 PRINT I;
40 NEXT I
50 ....
60 ....
```

```
10 FOR I=1 TO 100
20 IF I*I>1000 THEN I=100: GOTO 40
30 PRINT I;
40 NEXT I
50 ....
60 ....
```

In the first program, an early exit to the FOR . . . NEXT loop is caused by line 20. Even though lines outside the loop are then executed and the loop is no longer used, BASIC will consider the loop active. In the second program, the loop is exited through a normal termination. This normal termination is forced early in line 20 by setting the index equal to the ending value of the loop and branching to the NEXT statement. BASIC then increments the index, and goes to the FOR statement. Since the index now exceeds the ending value for the loop, the loop is deactivated and program execution continues with the first statement after NEXT. With a little thought, it is always possible to use the technique shown in this second program to deactivate loops when early exits are made. This should always be done.

Applications

1. FOR . . . NEXT statements are among the most often used BASIC statements. The discussions in this handbook of the other BASIC statements reveal that many of them achieve their full power when used in conjunction with FOR . . . NEXT statements.
2. FOR . . . NEXT statements can be used to sort data from lists, prepare tables, graph functions, solve optimization problems, and produce delays.

FRE

Each memory location contains a block of 8 binary bits (that is, zeros and ones), referred to as a byte. The Commodore 64 contains 64K bytes of RAM (Random Access Memory). One K of memory contains 1024 bytes. When the computer is first turned on, information similar to the following is displayed on the screen:

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

The second line tells us that there are 38911 memory locations available for use in storing and executing a BASIC program. We also can determine the amount of available bytes of memory by using the function FRE. At any time, the value of

FRE (0)

is the number of bytes of memory available for use at that time.

The value of FRE(0) will occasionally be a negative number. This occurs when the number of available bytes is greater than 32767. To get the correct number of available bytes, just add 65535 to this negative number. The following formula automatically makes this adjustment and returns the correct number of available bytes:

$$\text{FRE}(0) - (\text{FRE}(0) < 0) * 65535$$

Examples

1. The following statements were typed immediately after turning on the computer. (The same results can be obtained at any other time by first entering the commands NEW and CLR.)

```
PRINT FRE(0) - (FRE(0) < 0) * 65535
38908
```

```
READY,
10 A = 123.45
PRINT FRE(0) - (FRE(0) < 0) * 65535
38893
```

```
READY,
RUN
```

```
READY,
```

FRE

```
PRINT FRE(0)-(FRE(0)<0)*65535
38886
```

READY.

After CLearing memory we used FRE(0) to determine the maximum number of bytes which can be used by our BASIC program and variables. Then we stored a program into memory using up 15 bytes of memory. Next, we ran the program and used an additional 7 bytes to store the variable A in memory.

2. The following program illustrates the efficiency of using integer versus floating-point numeric constants.

```
10 PRINT FRE(0)-(FRE(0)<0)*65535
20 FOR I = 1 TO 10: A(I) = I: NEXT I
30 PRINT FRE(0)-(FRE(0)<0)*65535
40 FOR J = 1 TO 10: B%(J) = J: NEXT J
50 PRINT FRE(0)-(FRE(0)<0)*65535
RUN
38777
38708
38672
```

READY.

The numbers 1 to 10 required 69 bytes when stored as floating-point numbers, but only 36 bytes when stored as integers.

Comments

1. The value of FRE(0) is the same as the value of FRE(N), where N is any constant. For instance, line 10 of Example 2 could have been written 10 PRINT FRE(34.5)-(FRE(50)<0)*65535.

Applications

1. The programmer uses FRE(0) while writing a program to ensure that there is sufficient space in memory to continue the program. If not, he can take certain steps, such as storing the values of array variables in a sequential file instead of in memory, to make more space in memory available.

GET

The following program takes about 12 seconds to run. Suppose that while it is still running, we type the letters A, B, and C.

```
10 FOR I = 1 TO 10000
20 NEXT I
RUN
```

```
READY,
ABC█ ← cursor
```

The keyboard contains a buffer which can save up to 10 characters when the computer is busy elsewhere. The three letters were stored in the keyboard's buffer and were displayed on the screen after the program finished running. Now add two additional lines and proceed exactly as before.

```
10 FOR I = 1 TO 10000
20 NEXT I
30 GET A$
40 PRINT A$
RUN
A
```

```
READY,
BC█ ← cursor
```

In line 30, the GET statement read the letter A from the keyboard's buffer and assigned it to the string variable A\$. Line 40 PRINTed the value of A\$ onto the screen before the program ended. Then, as in the previous program, all remaining letters in the buffer were displayed.

In general, if A\$ is any string variable, then the statement

GET A\$

takes one of two actions, depending on the state of the keyboard's buffer.

1. If the buffer is empty, the string variable is assigned the empty string. The situation is the same as if the statement `A$ = ""` had been executed.
2. If the buffer is not empty, then the string variable is assigned the first character in the buffer, and space becomes available in the buffer for a new character.

Examples

1. The following program poses a multiple choice question.

```

10 PRINT "WHAT IS THE LONGEST RIVER IN THE WORLD?"
20 PRINT
30 PRINT "  A,  MISSISSIPPI"
40 PRINT "  B,  YANGTZE"
50 PRINT "  C,  NILE"
60 PRINT "  D,  AMAZON"
70 PRINT
80 PRINT "(TYPE A, B, C, OR D)"
90 GET A$
100 IF A$ = "" GOTO 90
110 IF A$ = "C" THEN PRINT "CORRECT, THE NILE": END
120 PRINT "TRY ANOTHER ANSWER": GOTO 90

```

The program displays the question and waits for the user to press A, B, C, or D. As soon as a key is pressed, the computer responds. The user does not have to press the RETURN key.

Comments

1. Characters read by GET statements will not be displayed on the screen or take effect as they are typed unless a PRINT statement is added for this purpose.
2. Unlike the INPUT statement, the GET statement has no provision to produce a question mark or a prompt to tell the user that a response is being requested. The cursor doesn't even appear. A prompt can be produced by a separate PRINT statement.
3. Every character which can be typed is read by the GET statement. This includes RETURN, DELEte, INSert, cursor controls, color controls, etc. Thus GET can be used to form a string consisting of *every* character typed by a user, up to and including the RETURN. The following is such a "line input" routine.

```

100 N$="" :REM N$ WILL HOLD A LINE OF INPUT
110 GET A$: IF A$="" THEN 110
120 PRINT A$;
130 N$=N$+A$
140 IF A$<>CHR$(13) THEN 110

```

4. Numeric variables can also be used with a GET statement. The

GET

statement GET A is used to assign to A a single digit from the keyboard buffer. If the buffer is empty, A is assigned the value 0. If the first character in the buffer is a digit, that digit will be assigned to the numeric variable. The following summarizes what happens when the first character in the buffer is not a digit:

- (a) If the first character is +, -, ., or E, then this character is discarded from the buffer, and the new first character is examined.
- (b) If the first character is a comma or a colon, then this character is discarded from the buffer, the message ?EXTRA IGNORED is displayed, and the new first character is examined.
- (c) In all other cases, the message ?SYNTAX ERROR is displayed, and the computer returns to direct mode.

Since syntax errors are so readily caused when using GET with a numeric variable, it is generally better to read all data as string data and have subroutines within the program decipher the input.

```
10 GET A: IF A=0 THEN GOTO 10
```

```
20 PRINT A
```

```
RUN
```

(The 5 key is typed)

```
5
```

```
READY.
```

```
RUN
```

(The keys 4 and 7 are typed)

```
4
```

```
READY.
```

```
7_ (Press RETURN before entering RUN)
```

```
RUN
```

(The keys - and 6 are typed)

```
6
```

```
READY.
```

```
RUN
```

(The keys , and 3 are typed)

```
?EXTRA IGNORED
```

```
3
```

```
READY.
```

5. More than one variable may be included in a single GET state-

GET

ment. If so, the keyboard buffer will be read to assign each variable a single character or digit.

```
10 GET A$, B$, X, Y
```

6. The GET statement cannot be used in direct mode. An ?ILLEGAL DIRECT ERROR message will result if this is attempted.

Applications

1. In programs involving music synthesis or animation with SPRITES, we do not wish the program to stop and wait for a response to the INPUT statement. With the GET statement we can check for input as part of a large, ever-executing control loop, and if no data is being entered, simply continue the loop.

2. The GET statement is the most general way to read in response by the user of your programs. When data is read in character by character, a program can be written to fully test the incoming data and thereby catch and deal with all erroneous data without causing program terminating errors or obscure error statements such as ?REDO FROM START.

GET#

A data file is a sequence of pieces of information residing on a cassette or disk. A data file is accessed with an OPEN statement which assigns a reference number, N, to the file. (See the discussion of OPEN for details.) Appendix F and the discussion of PRINT# explain how information is entered into files and how files look on cassettes and disks.

If a cassette or disk file has been OPENed for reading as number N, and A\$ is any string variable, then the statement

```
GET#N, A$
```

assigns to A\$ the next available (not yet read) character in the file. Successive uses of GET# will read successive characters from the file.

A cassette player and a disk drive are only two of several peripheral devices that can be attached to the computer. Other devices include a modem, screen, and keyboard. Any of these devices can be OPENed and read with GET# in much the same way as data files. Note that with the keyboard or modem, it is possible that no character is available to assign to A\$ when GET#N, A\$ is executed. In such cases, the effect will be the same as the assignment LET A\$ = "".

Examples

In the following examples, the symbol <R> stands for the RETURN character.

1. Suppose that the cassette file named JUNK begins as follows:

```
A,6<R>"NEW"<R> . . .
```

```
10 OPEN 2,1,0,"JUNK" :REM CASSETTE IS DEVICE 1
20 FOR I=1 TO 10
30 GET#2, A$
40 PRINT A$; ASC(A$);
50 NEXT I
60 CLOSE 2
RUN
```

```
PRESS PLAY ON TAPE
OK
A 65 , 44 6 54
13 " 34 N 78 E 69 W 87 " 34
13
READY.
```


GET#

Notice that the control character RETURN, with ASC value of 13, was read into A\$, and, when PRINTed, caused a new line to be started.

2. The following program finds the first "word" in a file. A "word" is defined here as beginning with a letter and ending when the next character is something other than a letter. Suppose that the sequential disk file "RIVERS" begins as follows:

```
8/13/83<R>NILE4160<R> ...

10 OPEN 2,8,2,"RIVERS,S,R"
20 B$=""
30 GET#2, A$
40 IF A$<"A" OR A$>"Z" THEN 30
50 B$=B$+A$
60 GET#2, A$
70 IF A$>="A" AND A$<="Z" THEN 50
80 PRINT B$
90 CLOSE 2
RUN
NILE

READY,
```

Note: This program treats SHIFTed letters as graphic characters, and would therefore need modification to find words written in the Low-ercase/Uppercase mode using SHIFTed letters.

Comments

1. Every character that can be entered into a file or device is read by the GET# statement. This includes RETURN, DELeTe, INSErt, cursor controls, color controls, etc., as illustrated in Example 2 for the RETURN character. Thus GET# can be used to form a string consisting of *every* character in a file or from a device, up to a delimiting character of our choice (perhaps RETURN). The following routine reads and displays an entire line of input (up to a RETURN character) from the file or device which is OPEN with reference number 1.

```
100 N$="" :REM N$ WILL HOLD A LINE OF INPUT
110 GET#1, A$: IF A$="" THEN 110
120 PRINT A$;
130 IF A$<>CHR$(13) THEN N$=N$+A$: GOTO 110
```

Note: A\$ is checked to see if it is RETURN before adding A\$ onto N\$. Thus N\$ will not include RETURN. N\$ can be made to include RE-

GET#

TURN as its final character by adding A\$ onto N\$ before checking for RETURN.

2. Numeric variables can also be used with a GET# statement. The statement GET#N,A is used to assign to A a single digit from the device or file referenced by N. If no character is currently available from the file or device, A is assigned the value 0. If characters are available, and the first character is a digit, that digit will be assigned to the numeric variable. The following rules summarize the results when the first available character is not a digit.

- (a) If the first character is +, -, E, or a period, then this character is discarded, and the new first character is examined.
- (b) If the first character is a comma or colon, then this character is discarded, the message ?EXTRA IGNORED is displayed, and the new first character is examined.
- (c) In all other cases, the message ?SYNTAX ERROR is displayed and the computer returns to direct mode.

Since syntax errors are so readily caused when using GET# with a numeric variable, it is generally better to read all data as string data and have subroutines within the program decipher the input.

```
10 OPEN 3,8,3,"VALUES,S,R"  
20 GET#3, A: IF A=0 THEN GET#3, A  
30 PRINT A
```

Assume VALUES begins 58.37<R> . . .

```
RUN  
5  
  
READY,
```

Assume VALUES begins -638<R> . . .

```
RUN  
6  
  
READY,
```

Assume VALUES begins ,348<R> . . .

```
RUN  
?EXTRA IGNORED  
3  
  
READY,
```

GET#

3. More than one variable may be included in a single GET# statement. The file or device associated with N will be read and a single character or digit will be assigned to each variable.

```
50 GET#5, A$, B$, X, Y
```

4. The character # must immediately follow the T in GET#. If a space is inserted, the ?SYNTAX ERROR message results. However, spaces may be placed after #. For example, GET #2, A\$ is not permitted, but GET# 2, A\$ is permitted.

5. When the statement GET#N, A\$ encounters the character having ASCII/Commodore value 0, A\$ is assigned the empty string, "". This fact is particularly important to keep in mind when working with the function ASC, since computing ASC of the empty string results in the message ?ILLEGAL QUANTITY ERROR and the termination of the program. When reading data with GET#, it is best to use ASC in the form ASC(A\$ + CHR\$(0)). If A\$ is not the empty string, this is the same as ASC(A\$), and if A\$ is the empty string, the value returned will be 0.

6. Information cannot be read from the printer. If a statement of the form OPEN N,4 is used to access the printer, then statements of the form GET#N, A\$ will cause the computer to go into an infinite waiting loop. To break this loop, hold down the RUN/STOP key and then hit the RESTORE key until the READY. prompt appears. This causes the computer to execute a *warm start*, which resets parameters such as screen background and border colors to their power-up values. However, your program will still remain unchanged in memory.

7. The GET# statement may be used to read characters from the screen, just as if the screen were a sequential file. The screen has 1000 positions that can be read in this way. After an OPEN statement of the form OPEN N,3 has been executed, the statement GET#N, A\$ will read the character at the current cursor position on the screen, and then the cursor will be advanced to the next position. To start reading with the first character on the screen, use PRINT "{CLR/HOME}"; to home the cursor. The last character on each logical line is read by GET# as a RETURN, no matter what this character really is. Finally, if the cursor has reached the lower right-hand corner of the screen and GET#N, A\$ is executed, the screen will scroll, and the first character (a space) on the new last line will be read.

8. When the end of a sequential disk file is reached and no further characters can be read, each use of GET# will return the character whose ASCII/Commodore value is 199.

GET#

9. The GET# statement cannot be used in direct mode. An ?ILLEGAL DIRECT ERROR message will result if this is attempted.

10. The INPUT# statement can also be used to read the information stored in a file. This statement has the advantage of being able to read an entire numeric value or a string of up to 80 characters all at one time, but has the disadvantage of requiring that RETURN characters be included to separate each data item when writing the file.

11. The GET statement performs just like the GET# statement, except that the former takes input only from the keyboard.

Applications

1. The GET# statement is the most general way of reading data from a file. When a file is read character by character, a program can be written to fully test the incoming data, and thereby catch and deal with all erroneous data without causing program terminating errors.

GOSUB and RETURN

GOSUB statements act like GOTO statements, but with an important embellishment. When the statement

```
GOSUB n
```

is encountered, the program branches to line *n*; that is, line *n* will be the next line executed. However, the computer remembers where it branched from, and when it encounters the statement

```
RETURN
```

it branches back to the statement immediately following the GOSUB statement. If several GOSUB statements are executed before a RETURN is encountered, the program branches to the statement after the most recently encountered GOSUB.

Think of the location of each GOSUB statement as being written on a sheet of paper. When the first GOSUB is encountered, its sheet of paper is placed on a table. When the second GOSUB is encountered, its sheet of paper is placed on top of the first one. As each GOSUB is encountered, its sheet is placed on top of the pile. When a RETURN statement is encountered, the piece of paper that is currently on the top of the pile is used to determine where to branch back to, and then this piece of paper is discarded. (This is called a "LIFO", Last In First Out, or "pushdown" stack.)

Examples

```
1. 10 PRINT "ONE ";
    20 GOSUB 100
    30 PRINT "THREE"
    90 END
    100 PRINT "TWO ";
    110 RETURN
    RUN
    ONE TWO THREE

    READY.
```

We can think of lines 100 and 110 as consisting of a simple subroutine. Line 90 ensures that the subroutine cannot be entered accidentally. (If line 90 is deleted, the word TWO will be printed a second time, and a ?RETURN WITHOUT GOSUB ERROR message will result.)

GOSUB and RETURN

2. The sequence in which the lines below are executed is 10, 20, 100, 110, 200, 210, 120, 130, 30, 90:

```
10 PRINT "ONE ";
20 GOSUB 100
30 PRINT "FIVE"
90 END
100 PRINT "TWO ";
110 GOSUB 200
120 PRINT "FOUR ";
130 RETURN
190 END
200 PRINT "THREE ";
210 RETURN
RUN
ONE TWO THREE FOUR FIVE

READY.
```

3. In the following program, the RETURN statement branches back to the statement after the GOSUB statement, not the line after the line containing the GOSUB statement.

```
10 PRINT "ONE ";; GOSUB 100: PRINT "THREE ";
20 PRINT "FOUR"
90 END
100 PRINT "TWO ";;RETURN
RUN
ONE TWO THREE FOUR

READY.
```

Comments

1. A subroutine is a relatively small program that resides inside a larger program. The subroutine performs some specific task that may have to be repeated many times while running the main program. Subroutines are generally designed to be accessed by a GOSUB statement which branches to the first line of the routine, and contain one or more RETURN statements to cause program execution to return to the main program at the statement after the GOSUB statement. This explains the choice of names for the GOSUB and RETURN statements.

2. Consider the pile of sheets of paper discussed above. In the event that a RETURN statement is encountered and the pile is empty, the

GOSUB and RETURN

?RETURN WITHOUT GOSUB ERROR message results. This can happen if a RETURN statement is encountered before any GOSUB has been executed, or if more RETURN statements have been encountered than GOSUB statements.

```
10 INPUT A: IF A<0 THEN GOSUB 30
20 PRINT "THE SQUARE ROOT IS";SQR(A)
30 PRINT"THE VALUE MUST NOT BE NEGATIVE"
40 INPUT A
50 RETURN
RUN
? 16
THE SQUARE ROOT IS 4
THE VALUE MUST NOT BE NEGATIVE
? 25

?RETURN WITHOUT GOSUB ERROR IN 50
READY.
```

The problem in this example is that the subroutine beginning at line 30 was entered even though the first value given for A was positive. After the user responded to the second INPUT prompt, the program reached the RETURN statement without coming through the GOSUB in line 10. The program can be made to run correctly by adding an END statement after line 20.

3. If more GOSUB statements than RETURN statements are encountered during the execution of a program, no error message will result. Those GOSUB statements for which no matching RETURN statement is found will appear to act like GOTO statements. However, if a program ends with some GOSUB statements unmatched, and the RETURN statement is entered in direct mode, the program will resume execution with the statement after the most recently executed and unmatched GOSUB statement. Note, however, that the computer will forget any unmatched GOSUB statements if one of the commands CLR, LOAD, NEW, or RUN is executed, or if a program line is added or edited.

```
10 GOSUB 30
20 PRINT "TWO" :END
30 PRINT "THREE"
40 GOSUB 20
50 PRINT "FIVE"
RUN
THREE
TWO
```

GOSUB and RETURN

```
READY,  
RETURN  
FIVE
```

```
READY,  
RETURN  
TWO
```

```
READY,
```

GOSUB statements should not be used in place of GOTO statements, as was the case in this example. The “stack” where BASIC keeps track of all the return locations for unmatched GOSUBs can only hold 23 locations. Too many GOSUBs without matching RETURNS will lead to an ?OUT OF MEMORY ERROR message.

```
10 I=0  
20 PRINT I; :I=I+1 :REM PRINT NUMBER OF  
GOSUBS EXECUTED  
30 GOSUB 20  
RUN  
 0  1  2  3  4  5  6  7  8  9 10 11 1  
2 13 14 15 16 17 18 19 20 21 2  
2 23
```

```
?OUT OF MEMORY  ERROR IN 20  
READY,
```

4. The n in GOSUB n must be a numeric constant. It may not be a variable or expression. Thus computed GOSUBs, such as INPUT A% : GOSUB A%*10, are not permitted. If for some reason n is not a whole number, GOSUB uses the truncated value of n . If the (truncated) value of n is not the number of a line in the program, an ?UNDEF'D STATEMENT ERROR message results.

```
10 INPUT A%: GOSUB A%*10  
RUN  
? 1
```

```
?SYNTAX  ERROR IN 10  
READY,  
10 GOSUB 20  
30 PRINT "THREE"  
RUN
```


GOSUB and RETURN

```
?UNDEF'D STATEMENT  ERROR IN 10  
READY.
```

5. GOSUB, like RUN and GOTO, can be used in direct mode to start the execution of a program at any line. Unlike RUN, however, GOSUB will not clear memory of all variables, arrays, etc. And, unlike GOTO, if RETURN is encountered, the computer returns to direct mode without giving an error message. Thus, GOSUB can be used conveniently from direct mode to debug subroutines.

Applications

1. Consider a program that processes orders for a company. When an order is received, the customer's name should be checked against the list of current customers. This task can be accomplished by a subroutine with a GOSUB statement.

2. Consider a tutorial program in which a question is asked to determine a student's understanding. If the student gives an incorrect response, the program should branch to a subroutine that explains the misunderstood concept. Since this subroutine might be branched to from many places in the lesson, the branching should be done via GOSUB statements.

3. It is good programming practice to break down a large program into smaller routines. In BASIC programming, each of these routines becomes a subroutine which is accessed by a GOSUB statement. With appropriate REM statements labeling each GOSUB statement, the main body of a program can be very easy to write and, more importantly, easy to verify that it will work. In turn, each subroutine should be simple enough that it too is easy to understand and verify. This technique of programming is termed *Top-Down* or *Structured* programming. The following program gives an example of this approach. Even though the main body of the program would not be too long if written without GOSUB statements, it still should be easy to see how breaking down a programming problem in this manner can greatly simplify the thought required to understand and correctly write any one part of the program.

```
10 REM     SIMPLE COMMAND INTERPRETING  
11 REM           PROGRAM  
12 REM  
20 GOSUB 100  
:REM READ A COMMAND FOR C$  
30 IF C$="ADD" THEN GOSUB 200
```

GOSUB and RETURN

```
:REM ADD A SET OF NUMBERS
40 IF C$="MUL" THEN GOSUB 300
:REM MULTIPLY A SET OF NUMBERS
50 IF C$="END" THEN END
60 GOTO 20
97 REM
98 REM   READ A COMMAND, PLACE IN C$
99 REM
100 PRINT "{SHIFT-CLR/HOME}"
110 PRINT "WHAT OPERATION DO YOU WISH TO
   PERFORM, "
120 PRINT "{CTRL-9}ADD{CTRL-0} OR {CTRL-
9}MULTIPLY{CTRL-0} A SET OF NUMBERS"
130 INPUT "OR {CTRL-9}END{CTRL-0}";C$
140 C$=LEFT$(C$,3)
150 IF NOT(C$="ADD" OR C$="MUL" OR C$="E
ND") THEN 100
160 RETURN
197 REM
198 REM   ADD A SET OF NUMBERS
199 REM
200 INPUT "HOW MANY NUMBERS ARE TO BE AD
DED";N
210 S=0
220 FOR I=1 TO N
230 INPUT A
240 S=S+A
250 NEXT I
260 PRINT "THE SUM IS";S
270 GOSUB 400:REM PAUSE UNTIL KEY STRUCK
280 RETURN
297 REM
298 REM   MULTIPLY A SET OF NUMBERS
299 REM
300 PRINT "HOW MANY NUMBERS ARE TO BE MU
LTIPLIED";N
310 S=1
320 FOR I=1 TO N
330 INPUT A
340 S=S*A
350 NEXT I
360 PRINT "THE PRODUCT IS";S
370 GOSUB 400:REM PAUSE UNTIL KEY STRUCK
380 RETURN
```

GOSUB and RETURN

```
397 REM
398 REM   WAIT UNTIL USER PRESSES A KEY
399 REM
400 PRINT
410 PRINT "PRESS ANY KEY TO CONTINUE."
420 GET A$
430 IF A$="" THEN 420
440 RETURN
```

GOTO

The statement

GOTO n

where n is the number of a line of the current program causes line n to be the next line executed. The statement is used both within a program and in direct mode.

Examples

```
1. 10 PRINT "ONE ";
    20 GOTO 40
    30 PRINT "THREE ";
    40 PRINT "FOUR ";
    50 PRINT "FIVE"
    RUN
    ONE FOUR FIVE
```

```
READY,
GOTO 30
THREE FOUR FIVE
```

```
READY,
```

```
2. 10 INPUT A
    20 IF A=0 THEN END
    30 PRINT A; "SQUARED IS"; A*A
    40 GOTO 10
    RUN
    ? 8
      8 SQUARED IS 64
    ? -1
     -1 SQUARED IS 1
    ?0
```

```
READY,
```

When “infinite” loops are created by a GOTO statement, as in this example, it is best to include an IF . . . THEN statement, as in line 10, which checks to see if a specially chosen control value is entered, and, if so, terminates the loop.

Comments

1. GOTO is frequently used with the IF statement in

IF condition GOTO n

Whenever the condition following the word IF is true, line *n* of the current program will be the next line to be executed.

```
10 INPUT "DID YOU CHOP DOWN THE CHERRY TREE"; A$
20 IF A$ = "NO" GOTO 10
RUN
DID YOU CHOP DOWN THE CHERRY TREE? NO
DID YOU CHOP DOWN THE CHERRY TREE? YES

READY,
```

2. In direct mode, the statement GOTO *n* is similar to the statement RUN *n*. Both cause the program to be executed beginning with line *n*. However, the RUN statement automatically clears all variables from memory, removes all information that has been set with DEF FN statements, causes all DIMensioned arrays to become undimensioned, deletes all file buffers and removes the meaning of all file numbers. The GOTO statement does not.

```
10 A = 3
20 PRINT A + B
RUN
3
```

```
READY,
B = 4: RUN
3
```

```
READY,
B = 4: GOTO 10
7
```

```
READY,
```

3. The statement GOTO should be used with moderation. Many programmers try to avoid its use entirely. It should not be used for long jumps, since this makes the program difficult to follow, modify, or debug. Be careful as well not to create "infinite loops" as in the following program.

GOTO

```
10 PRINT "INFINITE LOOP ";
20 GOTO 10
RUN
INFINITE LOOP INFINITE LOOP INFINITE L
OOP INFINITE LOOP INFINITE LOOP INFIN
ITE LOOP INFINITE LOOP
BREAK IN 10
READY.
```

This program will PRINT the words "INFINITE LOOP" indefinitely. To get out of the program press RUN/STOP.

4. The n in GOTO n must be a numeric constant. It may not be a variable or expression. Thus computed GOTOs, such as INPUT A% : GOTO A%*10, are not permitted. If for some reason n is a floating-point constant, GOTO uses the truncated value of n . If the (truncated) value of n is not the number of a line in the program, an ?UNDEF'D STATEMENT ERROR message results.

```
10 INPUT A%: GOTO A%*10
RUN
? 1
```

```
?SYNTAX ERROR IN 10
READY.
```

```
10 GOTO 20
30 PRINT "THREE"
RUN
```

```
?UNDEF'D STATEMENT ERROR IN 10
READY.
```

Applications

1. The GOTO statement allows us to repeat a process as long as we like. The following program turns the computer into a stopwatch.

```
10 PRINT "{SHIFT-CLR/HOME}";
20 PRINT "PRESS ANY KEY TO START"
30 GET A$: IF A$="" GOTO 30
40 TIME$="000000"
50 PRINT: PRINT: PRINT TIME$;
60 PRINT TIME-60*INT(TIME/60)
70 PRINT "{CLR/HOME}";
```

GOTO

```
80 PRINT "PRESS ANY KEY TO STOP "  
90 GET A$: IF A#="" GOTO 50  
100 PRINT:PRINT:PRINT
```

2. When used in command mode, the *GOTO* statement is a powerful debugging tool. We can set the values of the variables to whatever we like and then start the execution of the program at any point.
3. Suppose that a program produces graphics and we do not want to have the cursor and the prompt (*READY.*) appear at the end. This can be accomplished by making the last line of the program something like *999 GOTO 999*.

IF

IF statements provide the capability to make decisions. A statement of the form

IF condition THEN action

causes the program to take the specified *action* if the stated *condition* is true. If the *condition* is false, the *action* is not taken and the next line is executed.

Some common types of conditions involve the following relationships between numbers or strings.

<u>Relationship</u>	<u>Numbers</u>	<u>Strings</u>
=	is equal to	is identical to
<	is less than	precedes alphabetically
>	is greater than	follows alphabetically
<>	is not equal to	is not identical to
<=	is less than or equal to	precedes alphabetically or is identical to
>=	is greater than or equal to	follows alphabetically or is identical to

The most common types of actions are GOTO, GOSUB, PRINT, and LET statements. However, any BASIC statement can specify the *action* to be taken.

Examples

```
1. 10 PRINT "WHEN DID DARWIN PUBLISH"
    20 INPUT "HIS 'ORIGIN OF SPECIES'";Y
    30 IF Y = 1859 THEN GOTO 70
    40 IF Y < 1859 THEN PRINT "NOT THAT LONG AGO."
    50 IF Y > 1859 THEN PRINT "EARLIER THAN THAT."
    60 INPUT "TRY AGAIN: YEAR"; Y: GOTO 30
    70 PRINT "THAT'S CORRECT."
    RUN
    WHEN DID DARWIN PUBLISH
    HIS 'ORIGIN OF SPECIES'? 1850
    NOT THAT LONG AGO.
    TRY AGAIN: YEAR? 1859
    THAT'S CORRECT.

    READY.
```


2. The following program alphabetizes two words.

```

10 PRINT "TYPE TWO WORDS SEPARATED BY A COMMA."
20 INPUT "WORDS"; A$, B$
30 IF A$ <= B$ THEN GOTO 70
40 T$=A$ :REM EXCHANGE A$ WITH B$
50 A$=B$
60 B$=T$
70 PRINT A$, B$
RUN
TYPE TWO WORDS SEPARATED BY A COMMA,
WORDS? BYTE, BIT
BIT      BYTE

READY.
```

3.

```

10 PRINT "WHAT HORSE WON THE"
20 INPUT "TRIPLE CROWN IN 1973"; H$
30 IF H$ <> "SECRETARIAT" THEN GOTO 50
40 PRINT "CORRECT": END
50 PRINT "NO, SECRETARIAT"
RUN
WHAT HORSE WON THE
TRIPLE CROWN IN 1973? CITATION
NO, SECRETARIAT

READY.
```

Comments

1. Statements of the form *IF condition THEN GOTO n* are so common that BASIC has provided two ways to abbreviate them:

IF condition THEN n

IF condition GOTO n

For example, the following program contains all three forms.

```

10 PRINT 10;: A=2: B=3
20 PRINT 20;: IF A>1 THEN GOTO 40
30 PRINT 30;: IF B-A=1 GOTO 50
40 PRINT 40;: IF B=6/2 THEN 30
50 PRINT 50
```

IF

```
RUN
 10 20 40 30 50

READY.
```

2. The *action* part of an IF . . . THEN statement can consist of multiple BASIC statements separated by colons. The sequence

IF *condition* THEN *statement 1*: *statement 2*

should be thought of as (but not written as)

IF *condition* THEN (*statement 1*: *statement 2*)

That is, in the event that the *condition* is false neither *statement 1* nor *statement 2* will be executed.

3. The *action* part of an IF . . . THEN statement can consist of other IF statements. The form would be

IF *cond 1* THEN IF *cond 2* THEN *action*

which is interpreted as (but not written as)

IF *cond 1* THEN (IF *cond 2* THEN *action*)

This double IF . . . THEN statement can often be written as a single IF . . . THEN statement as follows:

IF *cond 1* AND *cond 2* THEN *action*

This single IF . . . THEN statement using the logical operator AND is generally easier to interpret, though a bit slower executing when *cond 1* is false.

```
10 INPUT "PITCHER, CATCHER"; P$, C$
20 IF P$="HOYT" AND C$="COLLINS" THEN PRINT "A WINNING COMBINATION": END
30 IF P$="HOYT" AND C$<>"COLLINS" THEN PRINT "WE WANT COLLINS": END
40 PRINT "HO HUM"
RUN
PITCHER, CATCHER? HOYT, ROBERTS
WE WANT COLLINS

READY.
```

IF

```
RUN
PITCHER, CATCHER? HOYT, COLLINS
A WINNING COMBINATION
```

READY.

There are occasions when it is not possible to reduce a double IF statement to a single statement as in the above example. Consider the problem of checking if $SQR(X)$ is less than 5. Since $SQR(X)$ is not defined for negative values of X , we should first check if X is greater than or equal to zero. Both checks can be accomplished by the double IF statement `IF X >= 0 THEN IF SQR(X) < 5 THEN action`. However, the corresponding single IF statement `IF X >= 0 AND SQR(X) < 5 THEN action` does not prevent $SQR(X)$ from being evaluated when X is negative, and so can lead to a program terminating error.

4. Conditions consisting of expressions involving numeric constants and variables often can be written without using any relationships (such as = and <>). When used in an IF statement, such conditions are considered to be false if the number obtained from evaluating the expression is zero and are considered to be true otherwise. So, for instance, the statement `IF A-5 THEN GOTO 99` has the exact same meaning as the statement `IF A <> 5 THEN GOTO 99`. (They both branch to line 99 only when the variable A has a value different than 5.)

```
10 INPUT "DENOMINATOR"; D
20 IF D THEN 40
30 GOTO 10
40 PRINT  $\pi/D$ 
RUN
DENOMINATOR? 0
DENOMINATOR? 2.7183
1.15571962
```

READY.

5. Complex conditions can be constructed from simple conditions by using logical operators such as AND, OR, and NOT. (Appendix L contains a detailed discussion of logical operators.) Just like arithmetic operators (+, -, *, \uparrow), logical operators are performed in a specific order. NOT is performed first, then AND, and finally OR. Also, arithmetic operators are evaluated before relationships, and both take precedence over logical operators. Some possibilities and their interpretations are:

```
NOT cond 1 AND cond 2 OR cond 3
((NOT cond 1) AND cond 2) OR cond 3
```

IF

cond 1 OR NOT cond 2 AND cond 3
cond 1 OR ((NOT cond 2) AND cond 3)

$C < A + B$ AND $A - 5$
 $(C < (A + B))$ AND $(A - 5)$

The following program will hire anyone who understands computers or who has both a doctorate and 5 years of work experience.

```
10 PRINT "DO YOU HAVE A DOCTORATE";
20 INPUT " (Y OR N)"; D$
30 PRINT "HOW MANY YEARS OF WORK"
40 INPUT "      EXPERIENCE DO YOU HAVE"; E
50 PRINT "DO YOU KNOW COMPUTERS";
60 INPUT " (Y OR N)"; C$
70 IF D$="Y" AND NOT E<5 OR C$="Y" THEN 90
80 PRINT "WE CAN'T USE YOU,":END
90 PRINT "YOU'RE HIRED."
```

6. The word THEN is not a BASIC statement and can only be used in conjunction with IF as illustrated in this discussion.

7. Due to rounding errors, the *condition* $A=B$ where A and B are floating-point numbers sometimes should be replaced by a *condition* such as $ABS(A-B)<.005$ to check the "absolute" error between the two numbers A and B , or $ABS((A-B)/A)<.0002$ to check the "relative" error of B with respect to A .

The following program finds the cube root of any number without using exponents or logarithms.

```
10 INPUT "NUMBER TO HAVE CUBE ROOT EXTRA
CTED"; A
20 B=A/3
30 C=(B+A/(B*B))/2
40 IF ABS((C-B)/C)<.000000001 THEN 60
50 B=C:GOTO 30
60 PRINT "THE CUBE ROOT OF"; A; "IS"; C;
RUN
NUMBER TO HAVE CUBE ROOT EXTRACTED? 125
THE CUBE ROOT OF 125 IS 5
READY.
```

IF

```
RUN  
NUMBER TO HAVE CUBE ROOT EXTRACTED? 1234  
56789  
THE CUBE ROOT OF 123456789 IS 497.933859
```

```
READY.
```

8. The ON statement allows the branching caused by several IF . . . GOTO statements to be combined together, and can be easier to interpret. See ON for further details.

INPUT

The INPUT statement is used to request information from the user of the program. A statement of the form

```
INPUT var
```

where *var* is a variable, causes the computer to display a question mark and a space. The computer then pauses until the user types in a value and presses the RETURN key, at which time the value is assigned to the variable.

Usually we want to tell the user the type of information requested by displaying a prompting message. A statement of the form

```
INPUT "prompt"; var
```

will display the message inside the quotation marks. The message will be followed by a question mark and a space.

Examples

```
1. 10 INPUT C$
    20 PRINT "***"+C$
    RUN
    ? USA
    ***USA

    READY.
```

After entering RUN, the user will see a question mark and a blinking cursor on the next line. Since "READY." hasn't appeared, the user knows that the computer has paused.

```
2. 10 INPUT N
    20 PRINT N*5
    RUN
    ? 23
    115

    READY.
```

```
3. 10 INPUT "WHAT IS YOUR BIRTHDATE";A$
    20 PRINT A$
    RUN
    WHAT IS YOUR BIRTHDATE? 2/10/55
    2/10/55

    READY.
```

INPUT

Comments

1. If the variable specified in the INPUT statement is a numeric variable, then the response to the request must be a numeric constant. Otherwise, a ?REDO FROM START message results and the request is repeated.

```
10 INPUT "AGE"; A
20 PRINT 7 + A/2
RUN
AGE? FORTY
?REDO FROM START
AGE?
```

Since A is a numeric variable, the computer could not assign the string constant FORTY to it.

2. If the variable specified in the INPUT statement is a numeric variable of integer precision (i.e., it ends with %), and the numeric constant entered has floating-point precision, then the floating-point constant will be converted to an integer constant according to the rules of the INT function. Integer constants entered for floating-point variables are converted to floating-point precision.

```
10 INPUT "AGE"; A%
20 PRINT A%
RUN
AGE? 40.7
40

READY.
```

Here the floating-point constant 40.7 was truncated when converted to be stored as the value of the integer variable A%.

3. When numeric constants are requested in an INPUT statement, the values entered must be of appropriate magnitudes for the corresponding variables. If integer variables are assigned numbers outside of the range from -32768 to 32767, the ?ILLEGAL QUANTITY ERROR message results, and if floating-point variables are assigned numbers with magnitude greater than machine infinity (1.701411183E+38), the ?OVERFLOW ERROR message results.

```
10 INPUT "AGE"; A%
20 PRINT A%
RUN
AGE? 40000
```

INPUT

```
?ILLEGAL QUANTITY ERROR IN 10  
READY.
```

4. Expressions are not allowed in response to an INPUT statement. For instance, in Example 1, the responses "US"+"A" and LEFT\$("USAmerica",3) would be rejected, and in Example 2, the responses 1/3 and EXP(5) would be rejected.

5. If the variable specified in the INPUT statement is a string variable, then the response should be a string constant. The string constant usually need not be enclosed in quotation marks. If it is, the quotation marks will be stripped from the constant. However, the string constant must be enclosed in quotation marks if it contains leading blanks, trailing blanks, commas, shifted letters, graphics, or control characters such as "cursor left", "red characters", and "clear screen". Also, if the leading character is a quotation mark, the string must not contain interior quotation marks.

```
10 INPUT A$: PRINT A$  
RUN  
? "TOY"  
TOY  
  
READY.  
RUN  
? "(1,2)" (Quotation marks are necessary  
(1,2) because of the comma.)  
  
READY.  
RUN  
? (1,2)  
?EXTRA IGNORED  
(1  
  
READY.  
RUN  
? "A "TWO-HEADED" DIME"  
?REDD FROM START  
? A "TWO-HEADED" DIME  
A "TWO-HEADED" DIME  
  
READY.
```

6. If the variable is a string variable and we respond with a number, the number will be treated as a string. Later it can be converted back to a number with the VAL function.

INPUT

```
10 INPUT A$: PRINT A$
RUN
? 12000
12000

READY.
```

We can tell from the way that 12000 was displayed that it is being stored as a string constant. Had it been a numeric constant, it would have been displayed with a leading space. If we want to do computations with this number, we can `LET S = VAL(A$)` and compute with S.

7. If we respond to the request for information by pressing the RETURN key, any previous value that the variable may have had will be unaltered. If a variable has not yet been assigned a value in a program, then the null string or 0 will be assigned to the variable, as is appropriate. To avoid having old values of a variable retained after an INPUT statement, assign the variables the empty string or zero, as is appropriate, before executing the INPUT statement. On the other hand, if a program requires that certain parameters be set, the variables storing these parameters can be assigned the most commonly used values as *default* values, and the user can be instructed to simply press RETURN if the default values are to be used.

```
10 A=4
20 INPUT A
30 INPUT B
40 PRINT A; B;
RUN
?      (RETURN is hit with no values entered.)
?      (RETURN is hit with no values entered.)
 4  0
READY.

10 A=15: B=10: C=55
20 PRINT "ENTER VALUE FOR PARAMETER OR ";
30 PRINT "PRESS RETURN TO USE DEFAULT VALUES."
40 INPUT "PAGE OFFSET";A
50 INPUT "TOP MARGIN";B
60 INPUT "LINES PER PAGE";C
.
.
.
```

8. Normally, during the execution of a program, if the key RUN/STOP

INPUT

is pressed, the **BREAK IN n** message is displayed and the computer returns to direct mode. If, however, the program is waiting for a response to an **INPUT** statement, pressing the **RUN/STOP** key has no effect. Yet it is often convenient to break a program when it requests input, perhaps to make some calculations. With just a little practice, the following maneuver will allow you to break and continue a program at an **INPUT** statement. The trick is to hit the **RETURN** and **RUN/STOP** keys at almost exactly the same time. The **RETURN** key must be struck first, but only a fraction of a second before the **RUN/STOP** key. Use both hands and strike (do not press and hold) the **RETURN** and **RUN/STOP** keys in rapid succession. If you succeed, you will see a **BREAK IN n** message, where n is the line number of the **INPUT** statement. You may now proceed as you wish in direct mode. If you would like to restart the program at the **INPUT** statement, do not use **CONT**, (we have already hit **RETURN** for the input, even though the line number in the break message was that of the **INPUT** statement) but rather, use **GOTO n** , where n is the line number of the **INPUT** statement.

```
10 INPUT "INCOME"; I#
20 PRINT I#
RUN
INCOME? (user performed the maneuver described above)

BREAK IN 10
READY,
PRINT 12000+1234.56
  13234.56

READY,
GOTO 10
INCOME? 13234.56
13234.56

READY,
```

After the first request, the user decided to use the computer as an adding machine to total various sources of income.

9. One **INPUT** statement can be used to request values for several variables. A statement of the form

```
INPUT var1, var2, var3
```

requests a response of three constants. These constants must be sepa-

INPUT

rated by commas and each should be of the same type as the corresponding variable.

10. If an INPUT statement contains more than one variable, and not all variables are assigned values, then the prompt ?? will be displayed to indicate that more values are needed. Typing RETURN after ?? will leave the next unassigned variable (not all the remaining variables) unchanged. The ?? prompt will continue to appear on successive lines until all variables in the INPUT statement have been given a new value, or individually left unchanged.

In any situation where more values are entered than are requested, the ?EXTRA IGNORED message will be displayed. Execution will continue with the program using as many of the values entered as are needed; the rest being discarded.

```
10 INPUT A,B,C
20 PRINT A;B;C
RUN
?      (user types RETURN without entering any values)
  0  0  0

READY,
RUN
? 5
??      (user enters no value)
?? 6
  5  0  6

READY,
RUN
? 7,4
?? 3
  7  4  3

READY,
RUN
? 9,6,4,2,1
?EXTRA IGNORED
  9  6  4

READY,
```

11. The prompt supplied with an INPUT statement must not exceed 40 characters, including the question mark and trailing space. This

INPUT

includes the case where a PRINT statement is used to supply a prompt, and the RETURN normally executed by a PRINT statement is suppressed. Using a prompt of more than 40 characters results in the prompt itself being interpreted as the first part of the response of the user. This causes confusion (to say the least) when INPUTing strings, and makes INPUTing a number impossible.

```
10 INPUT "THIS STATEMENT IS MUCH TOO LONG TO BE USED FOR INPUT";A$
20 PRINT A$
RUN
THIS STATEMENT IS MUCH TOO LONG TO BE USED FOR INPUT? FOR EXAMPLE
THIS STATEMENT IS MUCH TOO LONG TO BE USED FOR INPUT? FOR EXAMPLE
```

READY,

```
10 INPUT "ANOTHER EXCESSIVELY LONG PROMPT USED WITH NUMBERS "; A
20 PRINT A
RUN
ANOTHER EXCESSIVELY LONG PROMPT USED WITH NUMBERS ? 234
?REDO FROM START
ANOTHER EXCESSIVELY LONG PROMPT USED WITH NUMBERS ? 234
?REDO FROM START
ANOTHER EXCESSIVELY LONG PROMPT USED WITH NUMBERS ?
```

Press RUN/STOP-RESTORE to get out of this mess. Long prompts should be handled using several PRINT statements, and then an INPUT statement containing a short, final part of the prompt.

12. The INPUT statement cannot be used in direct mode. An ?ILLEGAL DIRECT ERROR message will result if this is attempted.

Applications

1. The INPUT statement is one of the most frequently used statements in BASIC. Computers are *data* processors, and the INPUT statement is a primary means of entering data into the computer.
2. Programmers make use of the INPUT statement when debugging

INPUT

a subroutine of a program. An INPUT statement is inserted temporarily in the subroutine to allow the programmer to assign values to the variables and observe the effects.

3. The INPUT statement can be used to stop execution of a program until the user is ready to continue. Suppose that the screen has been almost completely filled with data that the user should read before the program proceeds. The statement `INPUT "READY TO CONTINUE";C$` allows the user to take his time and absorb the material before continuing.

INPUT#

A *peripheral device*, or simply *device*, is an object that is connected to the central processing unit of the computer. Each device has an identifying number associated with it. The following table lists some common devices and their device numbers.

<u>Device</u>	<u>Device Number</u>
Disk drive	8
Cassette player	1
Screen	3
Keyboard	0
RS-232 interface	2

The INPUT# statement reads information from these devices in much the same way that the INPUT statement reads information from the screen.

A sequential data file is a sequence of pieces of information residing on a cassette or disk. Appendix F and the discussion of PRINT# explain how information is entered into sequential files and how files look on cassette or disk. In addition to the items of information inserted by the user, the files also contain the RETURN character, denoted as <R>. This character is inserted by BASIC for each PRINT# statement that does not end with a comma or semicolon.

To simplify our initial discussion, we assume that the following three conditions hold for each data file:

1. Each item (i.e., string or numeric constant) in the file is followed by a RETURN character.
2. Quotation marks enclose every string constant that contains a comma or a colon.
3. The character CHR\$(0) is not contained in any data file.

Comments 10, 11, and 12 discuss the effects of relaxing these conditions.

Before information can be read from a file or device, an OPEN statement first must be executed to access the file or device and associate it with a reference number N. (See the discussion of OPEN for details.) Information then can be read from the file or device, in order, starting at the beginning of the file. The statement

```
INPUT#N, A$
```

will assign to the string variable A\$ the next item not yet read from the file or device in the same manner that INPUT would read an item entered from the screen. In particular, if the item begins with a leading

INPUT#

quotation mark, then that quotation mark and any trailing quotation mark will not be assigned to the string.

The statement

```
INPUT#N, A
```

will assign to the numeric variable A the value of the next item not yet read from the file or device. If this item is not a numeric constant, the message ?FILE DATA ERROR results.

Examples

When the contents of a file are specified in the following examples, the symbol <R> denotes the RETURN character, and the character * denotes a space.

Examples 1 and 2 below refer to the disk data file called FRIENDS which begins as follows:

```
TOM*B.<R>16<R>RICH*S.<R>44<R>NEIL*L.<R>15<R>
```

```
1. 10 OPEN 2,8,2,"FRIENDS,S,R"  
   20 INPUT#2,A#  
   30 PRINT A#  
   40 CLOSE 2  
   RUN  
   TOM B.
```

```
READY.
```

```
2. 10 OPEN 3,8,3,"FRIENDS,S,R"  
   20 INPUT#3,A#  
   30 INPUT#3,B  
   40 INPUT#3,C#  
   50 PRINT A#,B,C#  
   60 INPUT#3,D  
   70 PRINT D  
   80 CLOSE 3  
   RUN  
   TOM B.      16      RICH S.
```

```
   44
```

```
READY.
```

3. Let COUNT be a data file on disk which begins as follows:

```
20*APPLES<R>15*PEARS<R>9*PEACHES<R>
```

INPUT#

```
10 OPEN 4,8,4,"COUNT,S,R"  
20 INPUT#4,B  
30 PRINT B  
40 CLOSE 4  
RUN
```

```
?FILE DATA ERROR IN 20  
READY.
```

The string "20 APPLES" is the first item in the file, and this item is not a numeric constant. RETURN characters should have been placed between each number and its associated fruit. In its current form, the file could be read using statements of the form INPUT#N, B\$. The function VAL can then be used to obtain the count of the fruit, while the function RIGHT\$ might be used to obtain the name of the fruit. See the discussion of these functions for further details.

4. Let ALPHA be a data file on cassette which begins as follows:

```
ABCDEF<R>"GH,IJ:KL"<R>-23.4E3<R>
```

```
10 OPEN 1,1,0,"ALPHA"  
20 INPUT#1,A$  
30 INPUT#1,B$  
40 INPUT#1,C  
50 PRINT A$,B$,C  
60 CLOSE 1  
RUN  
PRESS PLAY ON TAPE  
OK  
  
SEARCHING FOR ALPHA  
FOUND ALPHA  
ABCDEF    GH,IJ:KL  -23400  
  
READY.
```

Comments

When the contents of a file are specified in the following comments, the symbol <R> denotes the RETURN character, and the character * denotes a space.

1. A single INPUT# statement can assign values to several variables at once. The variables must be separated by commas.

INPUT#

The following example refers to the file RIVERS:

```
NILE<R>4160<R>AMAZON<R>4080<R>
```

```
10 OPEN 2,8,2,"RIVERS,S,R"  
20 FOR I = 1 TO 2  
30 INPUT#2, N$, L  
40 PRINT N$, L  
50 NEXT I  
60 CLOSE 2  
RUN  
NILE      4160  
AMAZON    4080
```

```
READY.
```

2. If a data file contains an item that is more than 88 characters long, including any surrounding quotation marks, then attempting to use INPUT# to read this item will result in the message ?STRING TOO LONG ERROR. Consider the example below using the file LONG:

```
*****THIS*FILE*BEGINS*WITH*MORE*THAN*88*  
CHARACTERS*WITHOUT*THE*RETURN*CHARACTER*  
OCCURRING.<R>
```

```
10 OPEN 2,8,2,"LONG,S,R"  
20 INPUT#2, A$: PRINT A$  
30 CLOSE 2  
RUN
```

```
?STRING TOO LONG ERROR IN 20  
READY.
```

3. Just as with the INPUT statement, a numeric item should not contain commas. If a comma is encountered in a numeric item, the INPUT# statement will discard the comma and all characters after it in the item. Consider the example below using the file BADNUMBER:

```
23,456.78<R>9,012<R>
```

```
10 OPEN 7,8,7,"BADNUMBER,S,R"  
20 INPUT#7,A  
30 INPUT#7,B  
40 PRINT A,B  
50 CLOSE 7
```

INPUT#

```
RUN
 23          9

READY.
```

4. When numeric data is being read by INPUT#, spaces before and between numeric characters are ignored. If an item consists entirely of spaces, a value of zero will be assigned to the variable. Consider the example below using the file CONSTANTS:

```
*****12345<R>-*3*2*4**23<R>***<R>

10 OPEN 2,8,2,"CONSTANTS,S,R"
20 INPUT#2, A,B,C
30 PRINT A,B,C
40 CLOSE 2
RUN
 12345      -324.23    0

READY.
```

5. When string data is being read by INPUT#, any leading spaces in an item are discarded. If the first non-blank character is a quotation mark, that quotation mark will be treated as a leading quotation mark rather than an embedded quotation mark (see Comment 6) and therefore will not be assigned to the string. Consider the following example using the file LEADING:

```
****LEADING*SPACES<R>****"QUOTE*TREATED*AS*LE
ADING"<R>

10 OPEN 4,8,4,"LEADING,S,R"
20 INPUT#4, C$,D$
30 PRINT C$
40 PRINT D$
50 CLOSE 4
RUN
LEADING SPACES
QUOTE TREATED AS LEADING

READY.
```

6. String constants to be read by INPUT# may contain embedded quotation marks, that is, quotation marks other than at the beginning and end of the string, as long as the string does not begin with a

INPUT#

quotation mark. Using INPUT# to read a string which begins with a quotation mark and contains embedded quotation marks will result in the message ?FILE DATA ERROR.

As mentioned in Comment 5, a quotation mark is considered to begin a string if it is the first *non-blank* character in the string. When a string does not begin with a quotation mark, all quotation marks which occur in the string will be assigned to the string variable by INPUT#. Consider the following example which uses the file QUOTES:

```
QUOTES*" *NOT*FIRST*ARE*PART*OF*" *DATA*" <R>
10 OPEN 100,8,14,"QUOTES,S,R"
20 INPUT#100, K$: PRINT K$
30 CLOSE 100
RUN
QUOTES " NOT FIRST ARE PART OF " DATA "
READY.
```

7. If a file contains consecutive RETURN characters, then "empty" items exist in the file. When INPUT# is assigning a value to a string variable and an empty item is found, the string variable is assigned the null string "". On the other hand, if INPUT# is assigning a value to a numeric variable, then it will skip over empty items until a non-empty item occurs. This item will then be used to assign a value to the numeric variable. For example, consider the file EMPTY:

```
THREE<R><R><R><R>3<R>
```

This file contains three empty items. The following program makes use of EMPTY.

```
10 OPEN 3,8,3,"EMPTY,S,R"
20 INPUT#3,A$,B$
30 INPUT#3,C
40 PRINT A$
50 PRINT B$
60 PRINT C
70 CLOSE 3
RUN
THREE
3
READY.
```

INPUT#

8. The INPUT# statement always considers the RETURN character as separating two items. Thus, if a string constant containing a RETURN character is placed in a file, the constant will be treated as two separate items by INPUT#. Note, however, that if a string constant contains a SHIFT-RETURN character, CHR\$(141), the constant will still be treated as a single item. Consider the following example which creates and then reads the following file. SHIFT-RETURN is denoted by <SR>.

```
"RETURN<R>INSIDE*QUOTES"<R>SHIFT<SR>RETURN<R>
```

```
10 OPEN 5,8,5,"RETURN,S,W"
20 A#=CHR$(34)+"RETURN"+CHR$(13)+
  "INSIDE QUOTES"+CHR$(34)
30 B#="SHIFT"+CHR$(141)+"RETURN"
40 PRINT#5,A#
50 PRINT#5,B#
60 CLOSE 5
70 OPEN 5,8,5,"RETURN,S,R"
80 INPUT#5, D#,E#,F#
90 PRINT D#
100 PRINT E#
110 PRINT
120 PRINT F#
130 CLOSE 5
RUN
RETURN
INSIDE QUOTES

SHIFT
RETURN

READY.
```

Note that even enclosing A\$ in quotation marks does not override the way INPUT# treats the first RETURN character. D\$ was only assigned characters up to the first <R>. "I" is then the first character assigned to E\$. Since this first character is not a quotation mark, all characters up to the second <R>, including the second quotation mark, are assigned to E\$. On the other hand, the SHIFT-RETURN character has no effect on INPUT#, and F\$ is assigned the same 12 characters as were originally assigned to B\$. When F\$ is PRINTed, the SHIFT-RETURN character causes the word RETURN to appear on a new line.

INPUT#

9. Cassette files contain a special character, CHR\$(0) as their last character. If this end-of-file character has been reached and the statement INPUT# is executed on a cassette file, the message ?STRING TOO LONG ERROR results. For disk files, there is no end-of-file character. The length of the file is recorded elsewhere on the disk so that it is not possible to read past the last item in a file. However, no error will result if the last item of a disk file has been read, and an attempt is made to read additional items from the file. Instead, the variables which were to be given new values are left with their old values, and execution continues. This could easily cause a program to go into an infinite loop. To prevent this, a check should be made to see if the end of the file has been reached by using the statement IF (STATUS AND 64) = 64 THEN *end-of-file action*.

10. If a string item contains a colon, and quotation marks do not surround the string, then the INPUT# statement will discard all characters from (and including) the first colon to the end of the item (up to the <R>). Consider the example below using the file NOQUOTE:

```
OGDEN:WEEMS<R>CLINGAN:SAMUELS<R>ROOSEVELT<R>
```

```
10 OPEN 2,B,2,"NOQUOTE,S,R"
20 INPUT#2,A$,B$,C$
30 PRINT A$,B$,C$
40 CLOSE 2
RUN
OGDEN      CLINGAN    ROOSEVELT

READY.
```

11. The INPUT# statement treats items containing commas in the same manner as does the INPUT statement. If a comma is not part of a string begun by a quote, then the comma separates the item into sub-items. If more sub-items are present than needed to assign values to the variables in the INPUT# statement, then these extra sub-items are ignored; however, no ?EXTRA IGNORED message appears. Consider the following program which uses the file COMMAS.

```
DAROLD,"SARAH,RICH",NANCY<R>474,277<R> 864,HOWARD
<R>JAN<R>
```

```
10 OPEN 2,B,2,"COMMAS,S,R"
20 INPUT#2,A$
```

INPUT#

```
30 INPUT#2,B
40 INPUT#2,C
50 INPUT#2,D$
60 PRINT A$,B,C,D$
70 CLOSE 2
RUN
DAROLD      474      864      JAN

READY,
```

The INPUT# statement in line 20 reads the first item, that is, everything up to the first <R> character. This item consists of three string sub-items: DAROLD / "SARAH,RICH" / NANCY. Since only one variable is to be assigned a value by this INPUT# statement, DAROLD is used and the remaining sub-items discarded. Similarly, the INPUT# statement in line 30 assigns the variable B the first sub-item of the next item, and discards 277 as extra data. Now consider the following program which uses the same file COMMAS.

```
10 OPEN 2,8,2,"COMMAS,S,R"
20 INPUT#2,A$,B$,C$
30 INPUT#2,D,E
40 INPUT#2,F,G$,H$
50 PRINT A$,B$,C$
60 PRINT D,E
70 PRINT F,G$,H$
80 CLOSE 2
RUN
DAROLD      SARAH,RICH      NANCY
  474      277
  864      HOWARD      JAN

READY,
```

The INPUT# statement in line 20 again reads the first item from the file. This time three variables are given to which values are to be assigned, so each of the three sub-items are required. Similarly, line 30 uses both sub-items of the next item to assign D and E their values. Note, also, what happens in line 70 when INPUT# is asked to assign three values, and the next item only contains two sub-items. INPUT# simply read in the next item to obtain a value for H\$.

Though these examples show that commas can be used within a file to separate items into sub-items for later reading with INPUT#, it is generally better to separate each piece of data by the RETURN character. This avoids the possibility of data being accidentally discarded.

INPUT#

Data can always be recovered if separated by RETURN, whereas a danger exists of losing data when commas act as separators.

12. The null character, CHR\$(0), is used in special ways by the computer when working with cassette and disk files. If this character is inserted into a file with the PRINT# statement, then difficulties arise when the INPUT# statement is used to recover data. Whenever encountered in an item, the null character causes all data after it and preceding the next RETURN character to be ignored. In addition, if the first character read from an item is the null character, the computer will go into an infinite wait loop. (To break this loop, hold down the RUN/STOP key and then hit the RESTORE key until the READY. prompt appears. This causes the computer to execute a *warm start*, which resets parameters such as screen background and border colors to their power-up values. However, your program will still remain unchanged in memory.)

13. Although normally used to read information from a cassette or disk, the INPUT# statement also may be used to read information that has been displayed on the screen. If a statement of the form OPEN N,3 has been executed, then the statement INPUT#N,var treats all of the characters on the screen from the cursor position to the end of the logical line as a single item, with the last character on the logical line always being interpreted as the RETURN character. The maximum length of a string read from the screen is therefore 79. The variable var is assigned a value from this item in accordance with the comments above.

```
10 PRINT "{SHIFT-CLR/HOME}"; :REM CLEAR SCREEN
20 PRINT "SOMETHING ON THE FIRST LINE"
30 PRINT "28"
40 PRINT "{CLR/HOME}";
50 OPEN 1,3
60 INPUT#1, A$,B
70 PRINT A$
80 PRINT LEN(A$)
90 PRINT B
100 CLOSE 1
RUN
(Screen is cleared.)
SOMETHING ON THE FIRST LINE
28
SOMETHING ON THE FIRST LINE
39
28

READY.
```

INPUT#

Note that the entire first line was read for A\$, as if spaces were typed after the word LINE and RETURN were typed at the last position on the line. Thus the length of A\$ is 39.

```
10 PRINT "{SHIFT-CLR/HOME}"; :REM CLEAR SCREEN
20 FOR I=1 TO 40: PRINT "A"; :NEXT I
30 PRINT "{CLR/HOME}"; :REM HOME CURSOR
40 OPEN 5,3
50 INPUT#5, A$
60 PRINT A$;LEN(A$)
70 CLOSE 5
```

RUN

(Screen is cleared.)

AA

AA

79

READY.

In this second example, a line of 40 A's causes the cursor to move, and therefore the logical line to extend to the second physical line. With the cursor again positioned at the beginning of the first line of the screen by the PRINT "{CLR/HOME}"; statement, the INPUT# statement reads the entire two physical lines which make up the logical line. Since the 80th position is treated as a RETURN and not included in the string, the length of A\$ is 79.

14. Although input from the keyboard is typically read with the INPUT or GET statements, it also may be read with the INPUT# statement. If a statement of the form OPEN N,0 has been executed, then the statement INPUT#N, *variable list* will assign values to the variables in *variable list* in exactly the same manner as the INPUT statement would. In particular, commas can be used in this case to separate multiple inputs in a single line of type.

There are several differences in the results from using INPUT# instead of INPUT to read from the keyboard. First, INPUT# does not provide a prompt. Second, when the RETURN key is pressed to end the input, the cursor does not move to the next line of the screen, but advances one space. Therefore, the next characters displayed on the screen will be on the same line as the value just entered.

```
10 OPEN 1,0
20 PRINT "WHAT'S YOUR NAME? ";
```


INPUT#

```
30 INPUT#1, N$
40 PRINT "GREETINGS!"
50 CLOSE 1
RUN
WHAT'S YOUR NAME? TOM GREETINGS!
```

READY.

```
10 OPEN 5,0
20 PRINT "@";
30 INPUT#5, A$,B,C$
40 PRINT A$;B;C$
50 CLOSE 5
RUN
@"TR:BSA",299,FUMC TR:BSA 299 FUMC
```

READY.

Note: The effect of a colon is still as described in Comment 10. If the quotation marks did not enclose the first item, A\$ would be assigned the value TR, and everything after the colon and preceding the pressing of RETURN would be ignored. Two further items would still need to be entered for B and C\$.

INT

If X is any number, then

$\text{INT}(X)$

is the greatest whole number less than or equal to X. On a number line, like the one shown in Figure 1, $\text{INT}(X)$ is X itself when X is a whole number, or the first whole number to the left of X when X is not a whole number.

Examples

```
1. PRINT INT(2.6);INT(-1.3);INT(1234E-2)
   2 -2  12
```

READY.

```
PRINT INT(3);INT(1/4);INT(876543.218)
   3  0 876543
```

READY.

```
2. 10 A = 45.67; B = -3.1
   20 PRINT INT(A);INT(B);INT(A+B);INT(B*B)
   RUN
   45 -4  42 -19
```

READY.

Comments

1. The INT function is often termed a “step” function. The reason for this is made clear by Figure 2, which shows the graph of $y = \text{INT}(X)$.

2. The function INT preserves the precision of the number on which it operates. Because of its name, one might think that INT returns only

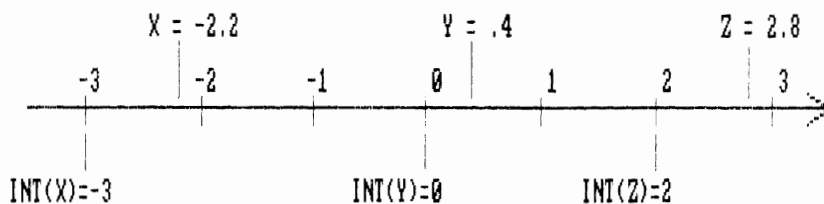


FIGURE 1

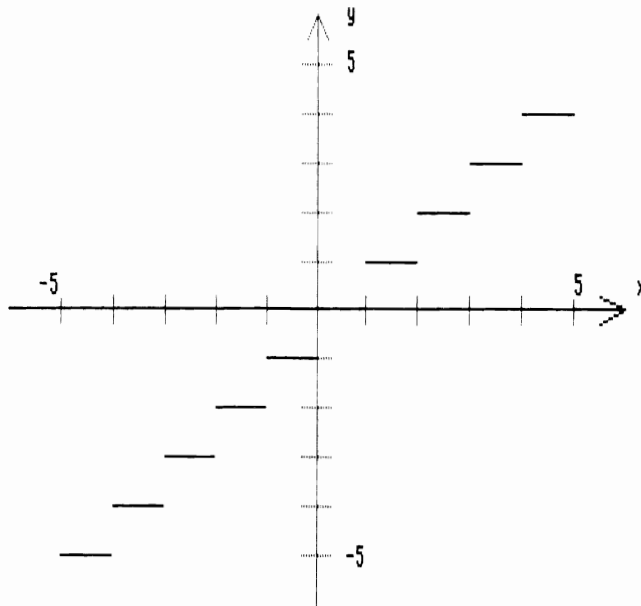


FIGURE 2

integer numeric constants. The last item in Example 1 shows that INT operates on floating-point values and returns a floating-point whole number. (If it were an integer, an error would have resulted since integers are less than or equal to 32767.)

3. The X in INT(X) may be any numeric constant, variable, or expression and may have either integer or floating-point precision.

```
A=299: PRINT INT(20*SQR(A))
345

READY.
```

4. When the INPUT statement requests a value for an integer variable and a floating-point constant is entered, the INT function is automatically used to convert the floating-point constant to a whole number. If this whole number doesn't lie in the range -32768 to 32767, an error message results.

```
10 INPUT A%
20 PRINT A%
```

INT

```
RUN
? 12.87
  12
```

```
READY.
RUN
? -432.22
-433
```

```
READY.
RUN
? 32800.76
```

```
?ILLEGAL QUANTITY ERROR IN 10
READY.
```

Applications

1. The INT function can be used to round numbers to a fixed number of decimal places. The following program rounds numbers to two decimal places.

```
10 INPUT A
20 B = INT(100*A)
30 IF 100*A-B < .5 GOTO 50
40 B = B+1
50 PRINT B/100
RUN
? 123.4567
  123.46

READY.
```

2. Certain formulas involve the INT function. For instance, the cost in cents of mailing a letter of weight X ounces is

$$20 + 17*\text{INT}(X)$$

3. INT is often used to interpret user input. For example, when asked his age, the user might enter 23.6. In this case, the value of INT(23.6) will most likely be more useful than either 23.6 or its rounded value.

The ability to manipulate strings of characters is an important power of computers. The LEFT\$ function allows us to extract a portion of a string. If A\$ is a string and N is a whole number between 0 and 255, inclusive, then

```
LEFT$(A$,N)
```

is the string consisting of the leftmost N characters of A\$.

Examples

- ```
1. PRINT LEFT$("COMMODORE",2)
 CO

 READY,
 PRINT LEFT$("MATTHEW WEBB",7)
 MATTHEW

 READY,
```
- ```
2. 10 T$="1234 RICHMOND STREET"
   20 PRINT LEFT$(T$,6),LEFT$(T$,1)
   RUN
   1234 R      1

   READY,
```
- ```
3. 10 PRINT "IS LA PAZ THE HIGHEST CITY"
 20 INPUT "IN THE WORLD"; A$
 30 B$ = LEFT$(A$,1)
 40 IF B$<>"Y" THEN PRINT"CORRECT":GOTO 60
 50 PRINT "INCORRECT"
 60 PRINT "LHASA, TIBET IS THE"
 70 PRINT "HIGHEST CITY IN THE WORLD."
 RUN
 IS LA PAZ THE HIGHEST CITY
 IN THE WORLD? YUP
 INCORRECT
 LHASA, TIBET IS THE
 HIGHEST CITY IN THE WORLD.

 READY,
```

# LEFT\$

By using LEFT\$, the program was able to tolerate many different responses to the question such as YES, YUP, and YEAH.

4. The following program isolates a person's first name from his full name. In line 50, the value of A will be the location of the first space in the name.

```
10 INPUT "FULL NAME";N$
20 FOR I=1 TO LEN(N$)
30 IF MID$(N$,I,1)=" " THEN 50
40 NEXT I
50 A=I
60 F$ = LEFT$(N$,A-1)
70 PRINT "YOUR FIRST NAME IS "; F$; ", "
RUN
FULL NAME? WILLIAM ARCHIBALD SPOONER
YOUR FIRST NAME IS WILLIAM.

READY.
```

## Comments

1. There are 256 different characters that can be used in a string. These characters and their ASCII/Commodore values are listed in Appendix A. All of these characters are counted when they appear in a string; even characters such as RETURN, DELETE, and INSERT. All spaces are counted, even leading and trailing spaces.

```
10 T$ = "VERY"+CHR$(13)+"TRULY YOURS"
20 PRINT LEFT$(T$,9)
RUN
VERY
TRUL

READY.
```

(Note: CHR\$(13) is the character RETURN.)

2. Although the LEFT\$ function is defined only for strings, it can be used indirectly to extract a specified number of digits from a number. For instance, if A is a positive whole number, then STR\$(A) is the string of characters consisting of a leading space followed by the digits corresponding to the value of A. Thus, taking into account the leading space, the N most significant digits of A are VAL(LEFT\$(STR\$(A),N+1)).

# LEFT\$

```
PRINT VAL(LEFT$(STR$(1875),3))
18

READY.
```

3. The LEFT\$ function creates a new string, but does not destroy the original string.

4. The N in LEFT\$(A\$,N) may be a numeric constant, variable, or expression and may be of either integer or floating-point precision. LEFT\$ uses the truncated value of N, and if this value is not between 0 and 255 inclusive, the ?ILLEGAL QUANTITY ERROR message results. If the whole part of N is zero, LEFT\$ will return the empty string, "". If N is greater than or equal to the number of characters in A\$, then LEFT\$ will return the complete value of A\$.

```
10 T$="NEIL ARMSTRONG"
20 PRINT LEFT$(T$,6,8)
30 PRINT LEFT$(T$,0)
40 PRINT LEFT$(T$,20)
50 PRINT LEFT$(T$,256)
RUN
NEIL A

NEIL ARMSTRONG

?ILLEGAL QUANTITY ERROR IN 50
READY.
```

## Applications

1. The LEFT\$ function can help make programs user-friendly, as in Example 3, and to manipulate strings, as in Example 4.

2. The LEFT\$ function can be used to *unpack* data from a string as shown in Example 4, though the MID\$ function is generally more useful in this regard. In fact, the LEFT\$ function can be defined in terms of MID\$:

LEFT\$(A\$,N) is equivalent to MID\$(A\$,1,N)

# LEN

If A\$ is a string, then the value of

LEN(A\$)

is the number of characters in the string.

## Examples

1. 

```
PRINT LEN("1 BYTE");LEN("MISSISSIPPI")
 6 11

READY.
```
2. 

```
10 A$ = "U. S. GRANT": B = LEN(A$)
20 E$ = "1882": F = -27
30 PRINT B; LEN(E$); LEN(STR$(F))
RUN
 11 4 3

READY.
```
3. 

```
10 INPUT "PHONE NUMBER"; P$
20 IF LEN(P$)=8 THEN PRINT "INCLUDE AREA
 CODE": GOTO 10
30 IF LEN(P$)<>12 GOTO 10
RUN
PHONE NUMBER? 123-4567
INCLUDE AREA CODE
PHONE NUMBER? 555-123-4567

READY.
```

## Comments

1. The Commodore 64 supplies 2 sets of 256 different characters which can be used in a string. These characters along with their ASCII/Commodore values are listed in Appendix A. All of these characters are counted when they appear in a string, even the characters such as RETURN, DELETE, and INSERT. All spaces are counted, even leading and trailing spaces.

```
A$="OH"+CHR$(13)+"MY": PRINT A$; LEN(A$)
OH
MY 5
```



# LEN

```
READY,
PRINT LEN(" **** ")
8
```

```
READY,
```

CHR\$(13) is the RETURN character. The asterisks are preceded and followed by two spaces.

2. The LEN function is defined only for strings. However, it can be used indirectly to determine how many digits are in a number. For instance, if A is a positive whole number (not in scientific form), then STR\$(A) is the string consisting of the digits in A plus the leading space placed with all positive numbers. Thus, the number of digits in A is LEN(STR\$(A))-1.

```
PRINT LEN(1822)

?TYPE MISMATCH ERROR
READY,
PRINT LEN(STR$(1822))-1
4

READY,
```

## Applications

1. LEN is used extensively when manipulating strings.
2. LEN is often used to obtain the terminating value for a FOR ... NEXT loop. The following program takes any string and displays its characters in reverse order.

```
10 INPUT A$
20 FOR L=1 TO LEN(A$)
30 PRINT MID$(A$,LEN(A$)+1-L,1);
40 NEXT L
RUN
? SMART
TRAMS
READY,
```

# LET

A variable is a name to which the computer can assign a value. Variables are of two types: numeric and string. Furthermore, numeric variables can be of two different precisions: integer and floating-point. A variable is assumed to be a numeric variable unless the last character of its name is a dollar sign. A numeric variable is assumed to have floating-point precision unless the last character of its name is a percent sign.

If *var* is a variable, and *val* is a value of the same type, then the statement

```
LET var = val
```

assigns the value to the variable.

## Examples

```
1. 10 LET A = 12345678
 20 LET B$ = "BOY"
 30 LET A% = 56
 40 LET SSN$(4) = "217-34-8087"
 50 PRINT A; B$; A%; SSN$(4)
 RUN
 12345678 BOY 56 217-34-8087

 READY.
```

## Comments

1. Variable names are formed using letters or digits, and, when appropriate, either the percent or dollar sign as the final character. (Note: SHIFTed letters may not be used.) Variable names must begin with a letter, and may contain as many letters or digits as desired (up to the limits of the 80-space logical line), allowing variable names to be chosen that convey their use in a program. However, if the variable name contains more than two letters or digits, the computer, in its internal bookkeeping, will use an abbreviated name for the variable, which consists of just the beginning letter and the next letter or digit of the original variable name. If the last character of the original variable name is a \$ or a %, then the abbreviated name also has this character as its last (second or third) character.

| <u>Original Name</u> | <u>Abbreviated Name</u> |
|----------------------|-------------------------|
| SUBJECT\$            | SU\$                    |
| PRICE                | PR                      |

# LET

```
QUANTITY% QU%
F16 F1
A$ A$
N N
```

```
10 ABC = 1
20 ABC% = 2
30 ABC$ = "HELLO"
40 PRINT AB, AB%, AB$
RUN
 1 2 HELLO

READY.
```

The following program illustrates the fact that two numeric variables of the same precision or two string variables having the same first two characters in their name will be regarded by the computer as being the same variable. Line 10 changes the value of the variable PR from 2 to 3, and then from 3 to 7.

```
10 LET PRICE=2: PROJECT=3: PRACTICE=7
20 PRINT PRICE; PROJECT; PRACTICE
RUN
 7 7 7

READY.
```

2. Caution must be taken to avoid including reserved words within the names of variables. Reserved words appearing anywhere within a variable name will cause a ?SYNTAX ERROR message. Appendix I contains a list of reserved words.

```
LET COST = 12.98

?SYNTAX ERROR
READY.
```

```
LET GALLONS = 34

?SYNTAX ERROR
READY.
```

The variable names GALLONS and COST contain the reserved words ON and COS.

# LET

3. If the variable is a string variable and the value is a string constant, then the value must be enclosed in quotation marks.

```
LET NAME$ = MOZART: PRINT NAME$

?TYPE MISMATCH ERROR
READY.

LET NAME$ = "BEETHOVEN": PRINT NAME$
BEETHOVEN

READY.
```

4. If the variable in a LET statement is a numeric variable with integer precision (ends with %), and the value is a floating-point numeric constant or variable, then the floating-point value will be converted to the largest whole number that is less than or equal to it. If this whole number lies in the range -32768 to 32767, then it will be converted to integer precision and assigned to the variable. Otherwise the message ?ILLEGAL QUANTITY ERROR will be displayed.

```
LET A% = 5.8: B% = -3.6: PRINT A%; B%
5 -4

READY.
LET C% = 43252

?ILLEGAL QUANTITY ERROR
READY.
```

5. If a floating-point variable is assigned a value with more than 9 significant digits, then 10 digits will be stored in memory and, when displayed, this 10-digit value will be rounded to a 9-digit number. Calculations use the full 10 digits being stored in memory.

```
10 LET A=6543210.9876
20 PRINT A; A-6000000
RUN
6543210.99 543210.988

READY.
```

6. Attempts to assign numeric values to string variables, or vice versa, result in the ?TYPE MISMATCH ERROR message.

```
LET A = "ADAM": PRINT A
```

# LET

```
?TYPE MISMATCH ERROR
READY,
LET A$ = 1234
```

```
?TYPE MISMATCH ERROR
READY,
```

7. LET statements can be used to assign expressions to variables. These expressions can be combinations of constants and/or variables.

```
10 LET A = 2*3+4
20 LET B = 5
30 LET B = B+1
40 LET C = 2*3+B
50 PRINT A; B; C
RUN
10 6 12

READY,
```

8. When the computer is given a statement of the form `LET var = expression`, the expression is evaluated first, then the answer assigned to the variable. Thus LET statements like `LET A=A+2` make sense;  $A+2$  is calculated, then this new value replaces the old value of  $A$ . Note then that LET statements do *not* set up an equation for the computer to solve. The statement `LET X=2*X+1` does not mean “if  $X$  equals  $2*X+1$  then what is  $X$ ”, but instead, “replace the current value of  $X$  by  $2*(\text{the current value of } X)+1$ .”

9. LET statements can omit the word LET. For instance, line 10 of Example 1 could have been written `A = 12345678`. In this handbook, we usually omit the word LET.

# LIST

The LIST command is used to obtain a listing of the current program with the line numbers in proper order. Variations of the command produce listings of specific lines or a sequence of consecutive lines.

The command

LIST

displays the entire program with the lines of the program in proper order. The command

LIST *n*

lists just the line numbered *n*. If  $m < n$ , the command

LIST *m-n*

lists all lines with numbers between *m* and *n* inclusive. The command

LIST -*n*

lists all lines up to and including line number *n*. The command

LIST *m-*

lists line number *m* and all lines following it.

## Examples

```
1. 10 PRINT "TEN"
 20 PRINT "TWENTY"
 30 PRINT "THIRTY"
 25 PRINT "TWENTY FIVE"
 LIST

 10 PRINT "TEN"
 20 PRINT "TWENTY"
 25 PRINT "TWENTY FIVE"
 30 PRINT "THIRTY"
 READY,
 LIST 20-25

 20 PRINT "TWENTY"
 25 PRINT "TWENTY FIVE"
```

```
READY,
LIST 25-

25 PRINT "TWENTY FIVE"
30 PRINT "THIRTY"
READY,
LIST -20

10 PRINT "TEN"
20 PRINT "TWENTY"
READY,
```

## Comments

1. If more than 23 lines of a program are requested by a LIST command, they will not all fit on the screen at one time, and scrolling will take place. The scrolling can be slowed substantially by holding down the CTRL key, which allows additional time to view the lines before they scroll off the screen. Pressing the RUN/STOP key will completely terminate the listing of the program.
2. The line numbers appearing in a LIST command needn't correspond to actual existing lines in the program. The command still carries out the spirit of the request by listing all existing lines that meet the specified criteria.

```
10 PRINT "A"
20 PRINT "B"
30 PRINT "C"
40 PRINT "D"
LIST 14-33

20 PRINT "B"
30 PRINT "C"
READY,
```

3. The line numbers referred to in the LIST command must be whole number constants between 0 and 63999. Otherwise the message ?SYNTAX ERROR results.
4. The LIST command is used primarily in direct mode. When it is used in a program line, it causes the execution of the program to terminate as soon as the LISTing is complete.
5. The LIST command normally displays the current program on the screen. Using the CMD command, it is possible to have the program

# LIST

LISTing go to any device or file, such as the printer. The following statements and commands LIST the current program on the printer, and then restore normal output to the screen. See the discussion of CMD for further details.

```
OPEN 1,4

READY,
CMD 1: LIST
PRINT#1

READY,
CLOSE 1

READY,
```

## Applications

1. The LIST command is essential when writing and debugging programs since it puts lines in their proper order and allows the programmer to look at any part of the program.
2. When debugging a program, we often use the combination of STOP and CONT to interrupt the execution of the program and check that everything is working properly up to that point. However, if after STOPping the execution at line numbered  $n$ , we alter or add a program line, then we can't use CONT to continue. However, if we know the next line number after line  $n$ , then we can use a GOTO statement to continue execution. We can determine that next line number with the command LIST  $n$ .
3. A simple way to copy statements from one part of a program to another is to LIST on the screen the lines to be copied, and then simply change the line number of each line and press RETURN. The old lines will be unaffected. You can even make changes in the old lines, give them new line numbers and press RETURN to create new lines which are similar, though not identical, to old lines. (Note: the cursor must be on the line you are copying or editing when you press RETURN in order for the addition to occur.)



In order to work with a program that resides on cassette or disk, the program must first be copied or LOADED into memory. This is accomplished by using the command

```
LOAD "programe", D
```

where *programe* is the name given to the program when it was SAVED, and D is 1 (if the program resides on a cassette) or 8 (if the program resides on a disk).

## Examples

1. Suppose that the program named PROG consists of the two lines

```
10 PRINT "MY " ;
20 PRINT "COMPUTER"
```

and resides on cassette.

```
LOAD "PROG",1

PRESS PLAY ON TAPE
OK

SEARCHING FOR PROG
FOUND PROG
LOADING
READY,
RUN
MY COMPUTER

READY,
```

2. Suppose a program titled PRIMES resides on disk.

```
LOAD "PRIMES",8

SEARCHING FOR PRIMES
LOADING
READY,
LIST

10 PRINT "PRIMES 2 TO 101"
20 PRINT 2
```

# LOAD

```
30 PRINT 3
40 PRINT 5
50 PRINT 7
60 FOR I=9 TO 101 STEP 2
70 FOR J=3 TO SQR(I+1) STEP 2
80 IF I/J = INT(I/J) THEN 110
90 NEXT J
100 PRINT I
110 NEXT I
```

## Comments

1. When used in direct mode, the LOAD command first executes a CLR command, clearing all variables, undimensioning all arrays, removing functions defined by DEF FN, and causing BASIC to “forget” all OPEN files. (See CLOSE for an explanation of “forgetting” OPEN files.)
2. When LOADING programs from cassette, the number D can be omitted. When omitted, the number defaults to 1.

```
LOAD "CASPROG"

PRESS PLAY ON TAPE

.
.
.
```

3. The LOAD command may be given with no program name or number D. If so, the next program from cassette will be LOADED. Pressing the RUN/STOP key while the SHIFT key is depressed will automatically issue the LOAD command in this way, and will RUN the program as well.
4. The quotation marks are essential in the statement LOAD “*prog-name*”,D. The string constant created by these quotation marks and *progname* can be replaced by a string variable. This can allow one program to LOAD a second which is chosen based on a name INPUT by the user. See Comment 7 below for more details on using LOAD within a program.

```
100 PRINT "THE PROGRAMS AVAILABLE ARE"
120 PRINT
130 PRINT
140 INPUT "WHICH PROGRAM INTERESTS YOU";A$
150 LOAD A$,B
```

# LOAD

5. When *programe* is given, it may contain "wild card" characters. The character \* stands for all possible characters and any number of them. Thus LOAD "F\*",D will load the first program whose name begins with the letter F. The character ? stands for any single character. Thus LOAD "M??",D will load the first program whose name begins with M and consists of a total of 3 characters. The command LOAD "\*",8 will LOAD the first program in the disk directory. Finally, LOAD "\*" is the same as simple LOAD for a cassette.

6. The LOAD statement provides access to the directory of files stored on disk. The statement LOAD "\$",8 will load the disk directory into memory as if it were a program. The directory can then be LISTed, though of course not RUN.

7. The LOAD command can be executed from within a program to LOAD and automatically RUN a new program. This process is referred to as *chaining* programs. For one program to successfully LOAD and execute a second, BASIC must be told at the beginning of the first program how much space the largest of the two programs requires. To determine the space required by a program in this sense, LOAD the program in direct mode and enter PRINT PEEK(45) + 256\*PEEK(46). Add 100 to the value displayed for the program containing the LOAD statement (to adjust for the six additional statements to be added below), and then let MAXSIZE be the larger of this value and the value displayed for the program to be LOAded. The following lines should be added at the *beginning* of the program containing the LOAD statement.

```
POKE 45,MAXSIZE-256*INT(MAXSIZE/256)
POKE 46,INT(MAXSIZE/256)
POKE 47,MAXSIZE-256*INT(MAXSIZE/256)
POKE 48,INT(MAXSIZE/256)
POKE 49,MAXSIZE-256*INT(MAXSIZE/256)
POKE 50,INT(MAXSIZE/256)
```

Note: To save yourself typing, use direct mode to calculate the value of INT(MAXSIZE/256) and then MAXSIZE-256\*INT(MAXSIZE/256). Do not, however, save typing by assigning variables the results of these calculations or setting MAXSIZE up as a variable. The six POKE statements above must be executed before any variables or arrays are used or DIMensioned in the program.

If, as suggested by the example in Comment 4, several programs are candidates for chaining, or if a chained-to program itself chains to a third program, and so on, determine the value of MAXSIZE based on the value of PEEK(45) + 256\*PEEK(46) for all programs involved. The six POKE statements must appear at the beginning of the original

# LOAD

program. They need not appear in chained-to programs which themselves do additional chaining.

When the LOAD command is executed from within a program, all variables, arrays, and functions defined by DEF FN retain their meanings and values, and all files which have been OPENed remain OPEN.

As an example of chaining, suppose that the program TEST consists of the lines

```
10 PRINT "WHO WAS THE 8TH PRESIDENT"
20 INPUT "OF THE UNITED STATES";A$
30 IF A$="VAN BUREN" THEN 50
40 PRINT "SORRY ";N$;"", TRY AGAIN,":GOTO 10
50 PRINT "A GOLD STAR FOR YOU!"
```

and resides on disk. In addition, the following program resides in memory:

```
10 POKE 45,2 :POKE 46,9
20 POKE 47,2 :POKE 48,9
30 POKE 49,2 :POKE 50,9
40 INPUT "WHAT IS YOUR NAME";N$
50 PRINT "HOW ABOUT A HISTORY QUIZ?"
60 INPUT "ARE YOU GAME (Y OR N)";A$
70 IF A$ <> "N" THEN LOAD "TEST",8
80 PRINT "WELL MAYBE NEXT TIME,"
```

The program in memory is now run:

```
RUN
WHAT IS YOUR NAME? MARY
HOW ABOUT A HISTORY QUIZ?
ARE YOU GAME (Y OR N)? Y
WHO WAS THE 8TH PRESIDENT
OF THE UNITED STATES? HARRISON
SORRY MARY, TRY AGAIN,
WHO WAS THE 8TH PRESIDENT
OF THE UNITED STATES? VAN BUREN
A GOLD STAR FOR YOU!

READY.
```

The suspicious programmer can use the LIST command to verify that the program TEST now resides in memory.

For any positive number X, the value of

LOG(X)

is the natural logarithm (or log to the base e) of X.

## Mathematical Preliminaries

The most familiar logarithmic function is  $\log_{10}(x)$ , known as the common logarithmic function, or the log to the base 10. For any positive number x,  $\log_{10}(x)$  is the exponent to which 10 must be raised in order to get x. For instance,  $\log_{10}(100) = 2$ ,  $\log_{10}(1,000,000) = 6$ , and  $\log_{10}(.001) = -3$ . Values of this function usually are obtained from a table or a calculator.

Logarithms can be defined to bases other than 10. If b is any positive number, then  $\log_b(x)$  is the exponent to which b must be raised in order to get x. For instance,  $\log_2(8) = 3$ ,  $\log_{64}(8) = .5$  and  $\log_5(2) = -1$ .

The most important logarithm, called the natural logarithm, is the one having as its base the number known as "e." The value of "e" to 9 significant digits is 2.71828183. Whenever we write  $\log(x)$  without referring to a base, it is implied that the base is "e". Hence,  $\log(x)$  is the exponent to which "e" must be raised in order to get x. Values of the natural logarithmic function are obtained by using the LOG function.

## Examples

```
1. PRINT LOG(3);LOG(.5);LOG(3.2E+12)
 1.09861229 -.69314718 28.7941719
```

READY.

```
2. PRINT LOG(5+2↑4);LOG(2.71828183);LOG(1)
 3.04452244 1 0
```

READY.

```
3. 10 A% = 1234: B = 4.3
 20 PRINT LOG(A%);LOG(2*A%+B);LOG(B)
 RUN
 7.11801621 7.81290417 1.45861502
```

READY.

# LOG

## Comments

1. LOG(X) is defined only for positive values of X. Figure 1 contains the graph of  $y = \text{LOG}(X)$ .
2. The X in LOG(X) may be either a numeric constant, variable or expression, and may have either integer or floating-point precision. The value of LOG(X) will always be a floating-point numeric constant. If the value of X is negative or zero, then the message ?ILLEGAL QUANTITY ERROR results.
3. LOG is the inverse of the function EXP. That is, for any positive number X, EXP(LOG(X)) has the value X, and for any number X, LOG(EXP(X)) has the value X.

```
10 A = LOG(1234): B = EXP(A)
20 C = EXP(3.2): D = LOG(C)
30 PRINT A; B; C; D
RUN
 7.11801621 1234 24.5325302 3.2
```

READY.

4. The logarithm to the base B can be obtained from LOG by using the formula

$$\log_B(X) = \text{LOG}(X)/\text{LOG}(B)$$

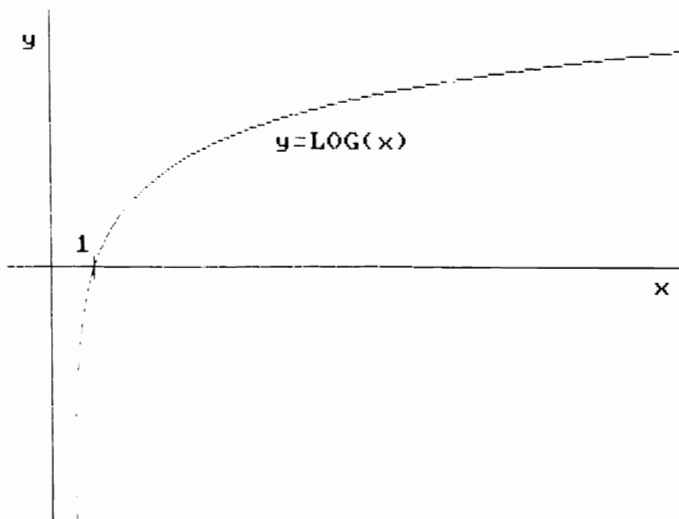


FIGURE 1

In particular,

$$\begin{aligned}\log_{10}(X) &= \text{LOG}(X)/\text{LOG}(10) \\ &= \text{LOG}(X)/2.30258509 \\ &= .434294482 * \text{LOG}(X)\end{aligned}$$

The following programs compute logarithms to bases other than “e.”

```
10 INPUT "POSITIVE NUMBER"; X
20 PRINT "THE COMMON LOG OF"; X;
30 PRINT "IS"; LOG(X)/LOG(10)
RUN
POSITIVE NUMBER? 2
THE COMMON LOG OF 2 IS .301029996
```

```
READY,
NEW
```

```
READY,
```

```
10 INPUT "BASE OF LOGARITHM"; B
20 INPUT "ARGUMENT"; X
30 PRINT "THE LOGARITHM TO THE BASE";
40 PRINT B; "OF"; X; "IS"; LOG(X)/LOG(B)
RUN
BASE OF LOGARITHM? 2
ARGUMENT? 64
THE LOGARITHM TO THE BASE 2 OF 64 IS 6
```

```
READY,
```

5. The most frequently used abbreviations for the natural (base “e”) and common (base 10) logarithmic functions are  $\ln$  and  $\log$ , respectively. For instance, these abbreviations usually appear on calculators. Our BASIC function LOG is actually the function  $\ln$ .

## Applications

1. Money invested at the interest rate of 12% compounded continuously will double after

$$\text{LOG}(2)/.12$$

years. In general, the number of years required for money to increase N-fold when invested at the interest rate R compounded continuously is

# LOG

LOG(N)/R

2. Scientists use the LOG function for many calculations. It is involved in measuring the intensities of earthquakes, determining the ages of ancient artifacts, and predicting the temperatures of cooling objects.

3. Logarithms were invented to reduce multiplication of numbers to addition:

$\text{LOG}(X*Y)$  equals  $\text{LOG}(X) + \text{LOG}(Y)$

With a modern computer, this is no longer necessary unless the result of the multiplication is larger than machine infinity, or smaller than “machine zero,” in which case logarithms may be useful. (By “machine zero” we mean the number of smallest magnitude that the computer can represent, other than zero itself.)



The ability to manipulate strings of characters is an important feature of computers. The MID\$ function allows us to extract a portion of a string. If A\$ is a string constant or variable and B and L are positive whole numbers, then

```
MID$(A$,B,L)
```

will be the string consisting of L successive characters of A\$ beginning with the Bth character. The number L can be omitted. If so,

```
MID$(A$,B)
```

will be the string consisting of all of the characters of A\$ from the character in location B on.

## Examples

```
1. PRINT MID$("CHESTER ALAN ARTHUR",9,4)
 ALAN
```

```
READY,
PRINT MID$("1830",2,1)
8
```

```
READY,
```

```
2. 10 A$ = "TOMORROW"
 20 C$ = MID$(A$,4): B = 4: L = 3
 30 PRINT C$, MID$(A$,B,L), A$
 RUN
 ORROW ORR TOMORROW
```

```
READY,
```

Notice that even though the MID\$ function extracted a portion of the string A\$ to form a new string, the value of A\$ was not changed.

3. The INPUT statement will not accept a fraction as a response unless the fraction is treated as a string. The following program converts the string to a number. In line 50, the value of S will be the location of the division sign.

```
10 INPUT "YOUR FRACTION"; F$
20 FOR I=1 TO LEN(F$)
```

## MID\$

```
30 IF MID$(F$,I,1)="/" THEN 50
40 NEXT I
50 S=I
60 F = VAL(MID$(F$,1,S-1))/VAL(MID$(F$,S+1))
70 PRINT F
RUN
YOUR FRACTION? 17/25
.68
```

READY.

```
4. 10 REM PROGRAM TO FIND THE WORDS
 20 REM IN A SENTENCE
 30 INPUT "YOUR SENTENCE";A$
 40 I=1: L=LEN(A$)
 50 GOSUB 200: REM FIND BEGINNING OF WORD
 60 IF I>L THEN END
 70 B=I :REM RECORD BEGINNING OF WORD
 80 GOSUB 300: REM FIND END OF WORD
 90 N=I-B: REM RECORD LENGTH OF WORD
 100 PRINT MID$(A$,B,N)
 110 IF I<=L THEN 50: REM FIND NEXT WORD
 120 END
 200 B$=MID$(A$,I,1)
 210 REM WORD BEGINS WITH LETTER OR DIGIT
 220 IF (B$>="0" AND B$<="9") THEN RETURN
 230 IF (B$>="A" AND B$<="Z") THEN RETURN
 240 I=I+1
 250 IF I>L THEN RETURN
 260 GOTO 200
 300 B$=MID$(A$,I,1)
 310 REM WORD CONTAINS LETTERS, DIGITS
 AND APOSTROPHES
 320 IF (B$>="0" AND B$<="9") THEN 360
 330 IF (B$>="A" AND B$<="Z") THEN 360
 340 IF (B$="'") THEN 360
 350 RETURN
 360 I=I+1
 370 IF I>L THEN RETURN
 380 GOTO 300
RUN
YOUR SENTENCE? THE 2ND WORD ISN'T THIRD!
THE
2ND
WORD
```

```

ISN'T
THIRD

READY,

```

## Comments

1. MID\$(A\$,B,L) will be the same string as MID\$(A\$,B) if there are fewer than L characters to the right of the Bth character of A\$.

```

PRINT MID$("FRIDAY",3,6),MID$("FRIDAY",3)
IDAY IDAY

READY,

```

2. The string A\$ in MID\$(A\$,B,L) may be a string constant, string variable, or an expression involving strings. The numbers B and L may be numeric constants, variables, or expressions and may have either integer or floating-point precision. Whenever the values of B or L are not whole numbers, MID\$ uses their truncated values. Since 255 is the maximum number of characters in a string, neither B nor L can exceed 255. Further, the value of B must be greater than or equal to 1, and the value of L greater than or equal to 0. Values of B or L outside these limits result in the ?ILLEGAL QUANTITY ERROR message.

```

PRINT MID$("ELEPHANT",2,3,4.8)
LEPH

READY,
PRINT MID$("ZEBRA",0,1)

?ILLEGAL QUANTITY ERROR
READY,

```

3. The MID\$ function creates a new string, but does not destroy the original string. (See Example 2.)

4. If B is greater than the number of characters in A\$ or if L is zero, then MID\$(A\$,B,L) will be the null string, "".

```

PRINT MID$("TIGER",6,2)

READY,

```

## MID\$

5. The Commodore 64 supplies two sets of 256 different characters which can be used in a string. These characters, along with their ASCII/Commodore values, are listed in Appendix A. All of these characters are counted when they appear in a string, even characters such as RETURN, DELETE, and INSERT. All spaces are counted, including leading and trailing spaces.

```
10 A$="CLEAN AS"+CHR$(13)+"A WHISTLE"
20 PRINT "FIT"+MID$(A$,6,6)+"FIDDLE"
RUN
FIT AS
A FIDDLE

READY.
```

## Applications

1. MID\$ is used to make programs user-friendly and to manipulate strings. It is essential for word processing.
2. MID\$ is used to access individual characters and *unpack* "words" from strings, as in Example 4.

Suppose that you have been working with a program and decide to abandon it and start all over again. The command

**NEW**

will delete the current program from memory and clear all values assigned to variables.

## Examples

```
1. 10 PRINT "TEN"
 20 A = 20: PRINT 20
 RUN
 TEN
 20
```

```
READY.
NEW
```

```
READY.
RUN
```

```
READY.
LIST
```

```
READY.
30 PRINT "TWENTY"
40 PRINT A
RUN
TWENTY
0
```

```
READY.
```

The **NEW** command reset the values of all numeric variables to 0 and caused the original program consisting of lines 10 and 20 to be deleted from memory.

## Comments

1. The **NEW** command does all that the **CLR** command does, and in addition removes all knowledge of the current program from memory.

# NEW

Of particular importance, NEW deletes all file buffers and the meanings of all file reference numbers from memory. Thus all open files should be CLOSED before issuing the NEW command. If files are not CLOSED and the NEW command is given, those on disk will be unreadable, and those on cassette will be incomplete.

2. The NEW command can be used within a program; however, the program will terminate as soon as it is executed.

# ON . . . GOSUB and ON . . . GOTO

The ON . . . GOSUB and ON . . . GOTO statements are really composites of the statements IF, GOSUB, and GOTO. (If you are unfamiliar with any of these three statements, refer to them before proceeding.)

ON . . . GOSUB and ON . . . GOTO statements involve a numeric variable (call it *I*), which normally assumes values like 1, 2, or 3, and a sequence of line numbers. If *m*, *n*, and *r* are line numbers, a statement of the form

```
ON I GOTO m, n, r
```

causes the program to GOTO *m* if the value of *I* is 1, GOTO *n* if the value of *I* is 2, and GOTO *r* if the value of *I* is 3. That is, the statement is equivalent to the following sequence of three statements:

```
IF I=1 GOTO m
IF I=2 GOTO n
IF I=3 GOTO r
```

We are not limited to branching to just three lines, but can allow branching to as many lines as we like. A statement of the form

```
ON I GOSUB m, n, r
```

has an analogous meaning to the statement discussed above.

## Examples

```
1. 10 INPUT I
 20 ON I GOTO 40, 50
 30 PRINT "THIRTY": END
 40 PRINT "FORTY ";
 50 PRINT "FIFTY": GOTO 10
 RUN
 ? 1
 FORTY FIFTY
 ? 2
 FIFTY
 ? 3
 THIRTY

 READY,
```

Notice that no branching resulted when *I* was assigned the value of 3.

## ON . . . GOSUB and ON . . . GOTO

```
2. 10 INPUT J
 20 ON J GOSUB 100, 200, 300, 400
 30 PRINT 30
 40 GOTO 10
 90 END
 100 PRINT 100;: RETURN
 200 PRINT 200;: RETURN
 300 PRINT 300;: RETURN
 400 END
 RUN
 ? 3
 300 30
 ? 2
 200 30
 ? 4
```

READY.

3. The following program uses the function SGN. SGN(A) is 1, 0, or -1 depending upon whether A is positive, zero, or negative.

```
10 INPUT "TOTAL TAXES"; T
20 INPUT "AMOUNT OF PREPAID TAXES"; A
30 ON 2+SGN(A-T) GOTO 40, 50, 60
40 PRINT "BALANCE DUE: $"; T-A: END
50 PRINT "ALL TAXES ARE PREPAID.":END
60 PRINT "AMOUNT OVERPAID: $"; A-T: END
RUN
TOTAL TAXES? 9876.54
AMOUNT OF PREPAID TAXES? 12345.67
AMOUNT OVERPAID: $ 2469.13
```

READY.

In this case, A-T was a positive number, SGN(A-T) was 1, and 2 + SGN(A-T) was 3.

4. The following incomplete program is an outline of a program to access a list of names and addresses. Each subroutine consists of a short program that performs the designated task.

```
10 PRINT " MENU"
20 PRINT "1. ADD ENTRY"
30 PRINT "2. DELETE ENTRY"
40 PRINT "3. CHANGE ENTRY"
```



## ON . . . GOSUB and ON . . . GOTO

```
50 PRINT "4. DISPLAY ENTRY"
60 PRINT "5. QUIT": PRINT
70 INPUT "SELECTION"; SELECT
80 ON SELECT GOSUB 100,200,300,400,500
90 PRINT "{SHIFT-CLR/HOME}": GOTO 10
99 END
```

```
100 Add entry subroutine code.
 :
199 RETURN
200 Delete entry subroutine code.
 :
299 RETURN
300 Change entry subroutine code.
 :
399 RETURN
400 Display entry subroutine code.
 :
499 RETURN
500 END
RUN
```

```
 MENU
1. ADD ENTRY
2. DELETE ENTRY
3. CHANGE ENTRY
4. DISPLAY ENTRY
5. QUIT
```

SELECTION?

5. Insurance premium rates are determined by age groups. Suppose that a policy has the following rate schedule:

| <u>Age</u> | <u>Rate</u> |
|------------|-------------|
| 20 - 39    | 76.54       |
| 40 - 59    | 98.76       |
| 60 - 79    | 123.45      |

```
10 INPUT "AGE LAST BIRTHDAY"; A
20 ON A/20 GOTO 50, 60, 70
30 PRINT "WE DON'T INSURE PERSONS"
40 PRINT "UNDER 20 OR OVER 80.": END
50 PRINT "YOUR ANNUAL RATE IS $76.54":END
60 PRINT "YOUR ANNUAL RATE IS $98.76":END
70 PRINT "YOUR ANNUAL RATE IS $123.45"
```

## ON . . . GOSUB and ON . . . GOTO

```
RUN
AGE LAST BIRTHDAY? 45
YOUR ANNUAL RATE IS $98.76

READY.
```

### Comments

1. The argument  $I$  in `ON I GOTO  $m, n, r$`  and `ON I GOSUB  $m, n, r$`  may be a numeric constant, variable or expression, and may be of either integer or floating-point precision. The `ON . . . GOTO` and `ON . . . GOSUB` statements use the truncated value of  $I$  to determine which of the line numbers  $m, n, r$ , etc., to execute next. The (truncated) value of  $I$  must be between 0 and 255 inclusive, or the ?ILLEGAL QUANTITY ERROR message results. The following are examples of the use of expressions in `ON . . . GOTO` and `ON . . . GOSUB` statements:

```
DN 10*I+3 GOTO 100, 200
DN LOG(I)/LOG(10) GOSUB 100, 200, 300
DN J-I GOTO 100, 200
```

(Note:  $\text{LOG}(I)/\text{LOG}(10)$  gives the logarithm to the base 10 of the number  $I$ .)

Consider the following runs of Example 1.

```
RUN
? 1.7
FORTY FIFTY
? 2.25
FIFTY
? 256

?ILLEGAL QUANTITY ERROR IN 20
READY.
```

2. If the (truncated) value of  $I$  is zero or greater than the number of line numbers listed following `ON I GOTO` or `ON I GOSUB`, then no branching occurs; execution continues with the next statement following the `ON . . . GOTO` or `ON . . . GOSUB` statement.

```
10 K = 5
20 ON K GOTO 10, 20, 30, 40 : PRINT "A"
30 PRINT "B"
40 END
```

## ON . . . GOSUB and ON . . . GOTO

```
RUN
A
B

READY,
```

Note that with  $K$  greater than 4, execution continued with the statement, not the line, following the ON . . . GOTO statement.

3. The numbers  $m$ ,  $n$ ,  $r$ , etc., in an ON . . . GOTO or ON . . . GOSUB statement must be numeric constants. Computed GOTOs or GOSUBs using variables or expressions for  $m$ ,  $n$ ,  $r$ , etc., are not allowed. If the values of  $n$ ,  $m$ , or  $r$  are not whole numbers, their truncated values will be used. If the (truncated) value of one of these numbers is not the number of a line in the program, an ?UNDEF'D STATEMENT ERROR message results.

```
10 INPUT A,B
20 INPUT I
30 ON I GOTO A, B, 10
RUN
? 10,50
? 1

?UNDEF'D STATEMENT ERROR IN 30
READY,
```

```
10 INPUT "PRICE";A
20 ON A/200 GOTO 4, 50
30 PRINT "CREDIT ALLOWED": END
40 PRINT "SPECIAL CREDIT REQUIRED": END
50 PRINT "NO CREDIT ON THIS SALE": END
RUN
PRICE? 235

?UNDEF'D STATEMENT ERROR IN 20
READY,
```

# OPEN

A *peripheral device*, or simply *device*, is an object that is connected to the central processing unit of the computer. Each device has an identifying number associated with it. The following table lists the most common devices and their device numbers.

| <u>Device</u>    | <u>Device Number</u> |
|------------------|----------------------|
| Cassette player  | 1                    |
| Disk drive       | 8                    |
| Printer          | 4                    |
| Screen           | 3                    |
| Keyboard         | 0                    |
| RS-232 interface | 2                    |

The OPEN statement is used to gain access to each of these devices.

The cassette player and the disk drive store programs and data files. BASIC programs are stored by the SAVE command and retrieved by the LOAD command. Information is written into data files with PRINT# statements and read from data files with INPUT# and GET# statements. However, before information can be written into or read from a specific data file, the file must be assigned a temporary reference number N. The statements PRINT#N, INPUT#N, and GET#N will then refer their operations to this file. (See Appendix F for a general discussion of data files.)

OPEN statements that apply to data files on the cassette player and the disk drive have the general form

```
OPEN N, D, . . .
```

where N (1 to 127) is the reference number of the file and D (1 or 8) is the device number. A statement of the form

```
OPEN N,1,1,"filename"
```

creates a cassette data file, with reference number N and specified name, that can be *written* to. A statement of the form

```
OPEN N,1,0,"filename"
```

accesses the specified cassette data file so that it can be *read* from and assigns it the reference number N. In the following program, the cassette player is accessed first in line 10 for writing, and then again in line 60 for reading. The STOP statement in line 50 causes program execution to be halted so that the cassette may be rewound to the

# OPEN

beginning of the file NEIGHBORS. The CONT statement then causes program execution to continue at line 60.

```
10 OPEN 1,1,1,"NEIGHBORS"
20 PRINT#1, "SUSAN"
30 PRINT#1, "BILL"
40 CLOSE 1
50 STOP
60 OPEN 1,1,0,"NEIGHBORS"
70 INPUT#1, A$, B$
80 PRINT: PRINT A$, B$
90 CLOSE 1
RUN
PRESS RECORD & PLAY ON TAPE
OK
```

```
BREAK IN 50
READY.
CONT
PRESS PLAY ON TAPE
OK
```

```
SEARCHING FOR NEIGHBORS
FOUND NEIGHBORS
SUSAN BILL
```

```
READY.
```

A statement of the form

```
OPEN N,8,C,"filename,S,W"
```

creates a disk (Sequential) data file, with reference number N and specified name, that can be *Written* to. The number C (2 to 14) is referred to as the channel number and usually has the same value as N. A statement of the form

```
OPEN N,8,C,"filename,S,R"
```

accesses the specified disk (Sequential) data file so that it can be *Read* from and assigns it the reference number N.

In the following program, Line 10 creates the disk file NEIGHBORS, OPENing it for writing. After information is written to the file, it is CLOSED and then reOPENed for reading by line 50.

# OPEN

```
10 OPEN 7,8,7,"NEIGHBORS,S,W"
20 PRINT#7,"MARY"
30 PRINT#7,"JAMES"
40 CLOSE 7
50 OPEN 3,8,3,"NEIGHBORS,S,R"
60 INPUT#3, A$, B$
70 PRINT A$, B$
80 CLOSE 3
RUN
MARY JAMES
```

READY.

Finally, access to the disk drive's own internal control system, DOS, is given by the statement

```
OPEN N,8,15
```

which allows statements of the form `PRINT#N,"command"` to issue commands to the disk drive. For instance, these commands can `SCRATCH` a file from the disk, or prepare a `NEW` disk to store files and programs. (See Appendix D for a discussion of DOS commands.) Further, when a flashing red light on the disk drive indicates that an error has occurred, statements of the form `INPUT#N,A$,B$,C$,D$` can be used to read an error message from DOS describing the problem. The following statements "erase" the file `NEIGHBORS` from the disk.

```
OPEN 15,8,15

READY.
PRINT#15,"SCRATCH0:NEIGHBORS"

READY.
```

Characters can be sent to the printer by `PRINT#` statements in much the same way that characters are sent to the screen by `PRINT` statements. The statement

```
OPEN N,4
```

provides access to the printer and assigns it the reference number `N` (1 to 127). This statement allows statements of the form `PRINT#N,data` to type out data on the printer. The following program prints a brief message.

# OPEN

```
10 OPEN 2,4
20 PRINT#2,"THIS APPEARS ON THE PRINTER"
30 CLOSE 2
RUN
```

(The message is typed on the printer.)

```
READY.
```

In the next program, access to the printer is given by the statement `OPEN 1,4`. Then the `CMD` statement is used to redirect all output to the printer, including the requested `LISTING` of the current program. See the discussion of `CMD` for further details.

```
OPEN 1,4

READY.
CMD 1
LIST
PRINT#1

READY.
```

## Comments

1. Only 10 reference numbers may be associated with files or devices at any one time. Attempting to associate an 11th reference number results in the message `?TOO MANY FILES ERROR`.
2. The second value given in the `OPEN` statement is the number of the device to be accessed. Device numbers greater than 3 refer to devices on the *serial bus*, such as the printer and disk drive. The printer can be set up as device 4 or 5 by altering the device selection switch on the rear of the printer. (We have made the convention that the switch is set to 4.) The disk drive is normally device 8, though additional drives can be hooked to the computer as devices 9 thru 11.
3. A given reference number may be used to identify only one file or device at a time. If an `OPEN` statement is executed using a reference number that is already associated with a file or device, the `?FILE OPEN ERROR` message results. The reference number `N` remains associated with the file or device specified in the `OPEN` statement until a `CLOSE N` statement is executed, or until `BASIC` "forgets" the association as a result of one of the commands `CLR`, `LOAD`, `NEW`, or `RUN` being executed, or a program line being added or edited. See the dis-

# OPEN

discussion of CLOSE for further details of why BASIC “forgets” file references.

```
10 OPEN 1,4
20 OPEN 2,4
30 OPEN 1,3
RUN
?FILE OPEN ERROR IN 30
READY,
```

Note that a single device may have more than one reference number associated with it, but not vice versa.

4. If one of the statements `CMD`, `GET#`, `INPUT#`, or `PRINT#` is executed using a reference number which has not been associated with a file or device by an `OPEN` statement, or a reference number whose association to a file or device has been “forgotten” (see Comment 3 and the `CLOSE` statement), then the `?FILE NOT OPEN ERROR` message results.

```
OPEN 1,4

READY,
NEW

READY,
PRINT#1, "TITLE"

?FILE NOT OPEN ERROR
READY,
```

5. If an `OPEN` statement is executed to access a named file from cassette, and the named file does not exist on the cassette, the cassette will continue to be read in search of the file until the tape runs out or until an end-of-tape character is read. In the latter case, the message `?DEVICE NOT PRESENT ERROR` results. If an `OPEN` statement is executed to access a disk file, and the file does not exist on the disk, the `?FILE NOT FOUND ERROR` message results.

6. The names of cassette or disk files may be up to 16 characters long. If a longer name is given in an `OPEN` statement, only the first 16 characters will be used. Any character may be used in a file name except the following five: quotation mark ("), asterisk (\*), comma (,), colon (:), or question mark (?). The use of these five special characters can make it impossible to access or `SCRATCH` the file at a later time.



# OPEN

In practice, only letters, digits and the period character are used to form file names.

7. When OPENing cassette files, the file name may be omitted. A statement of the form OPEN N,1,0 will access the next file on the cassette, making it available for reading, while a statement of the form OPEN N,1,1 will create an unnamed data file on cassette, making it available for writing. Further, the statement OPEN N,1,0 can be simplified to OPEN N,1 or just OPEN N .

8. A statement of the form OPEN N,1,2 accesses the cassette for writing, as OPEN N,1,1 does, but in addition causes an end-of-tape mark to be written when the file is CLOSED. The end-of-tape mark may be written after a file at any point on the tape. It can be used after the last file on the tape to stop a LOAD or VERIFY statement from reading all the way to the physical end of the tape in search of a file.

9. A statement of the form OPEN N,4 both accesses the printer and selects the Uppercase/Graphics character set. If the Lowercase/Uppercase set is desired, the printer must be initialized into the Lowercase/Uppercase print mode by using the modified statement OPEN N,4,7 . It is also possible to alternate between character sets by sending certain control characters to the printer. See PRINT# and Appendix K for further details.

```
10 OPEN 1,4
20 PRINT#1, "CAPS & GRAPHICS ⬠⬡ | ⌞ "
30 CLOSE 1
40 OPEN 1,4,7
50 PRINT#1, "SMALL LETTERS & -⬠⬡ | ⌞ "
60 CLOSE 1
```

The execution of this program produces the following output on the printer.

```
CAPS & GRAPHICS ⬠⬡ | ⌞
small letters & CAPS
```

10. If desired, the printer may be accessed by statements of the form OPEN N,4,M, "*title or comment*", where M is 0 or 7. If so, *title or comment* will be printed in the Uppercase/Graphics character set (even if M is 7) as soon as a CMD statement or a PRINT# statement without terminating punctuation is executed. All additional material will be printed in the Uppercase/Graphics character set if M is 0 and the Lowercase/Uppercase character set if M is 7.

# OPEN

11. Information cannot be read from the printer. If a statement of the form `OPEN N,4` is used to access the printer, then `INPUT#N` and `GET#N` statements will cause the computer to go into an infinite waiting loop. To break this loop, hold down the `RUN/STOP` key and then hit the `RESTORE` key until the `READY.` prompt appears. This causes the computer to execute a warm start, which resets parameters such as screen background and border colors to their power-up values. However, the current program will still remain unchanged in memory.

12. If several disk files are to be accessed at one time, the number `C` should be unique in each `OPEN N,8,C,"filename,S,R"` statement. No error messages will appear if this rule is not followed, but the results of read and write operations to files using the same `C` value will not be as expected. Since the reference number `N` also must be unique for each file, one way to assure that `C` is also chosen uniquely is to use the same value for both `N` and `C` when writing an `OPEN` statement.

13. If a disk file has already been created, and it is desired to replace that file with a new version, the file name used in the `OPEN` statement must be preceded by the three characters `@0:` ("at", zero, and colon). For example, to replace the contents of the disk file `DATA` with new values, access the file with a statement of the form `OPEN N,8,C,"@0:DATA,S,W"`. Without the `@0:` prefix, attempting to rewrite an old file will not cause a `BASIC` error, but a `FILE ALREADY EXISTS` error will occur within the disk drive's `DOS`, and the file will not be replaced.

14. The same disk file can be `OPENed` for reading more than once by using different values of `N` and `C`. For example, a program could contain both of the statements `OPEN 2,8,2,"DATA,S,R"` and `OPEN 3,8,3,"DATA,S,R"`. Two reference numbers might be used in reading a file, using one number with `GET#` statements to "look ahead" for some special character before using `INPUT#` statements to read in an entire data item.

Suppose the file `DATA` consists of the following:

```
123<R>79PINS<R>34,5E+4<R>
```

Note that `<R>` stands for the character `RETURN`.

The following program uses `GET#` to check if the next item in the file contains any non-numeric characters. If not, it uses `INPUT#` to read in the numeric value.

```
10 OPEN 2,8,2,"DATA,S,R"
20 OPEN 3,8,3,"DATA,S,R"
30 FOR I = 1 TO 3
```

# OPEN

```
40 GET#2, A$
50 IF A$="+" OR A$="-" OR A$="," GOTO 40
60 IF A$>="0" AND A$<="9" GOTO 40
70 IF A$=" " OR A$="E" GOTO 40
80 IF A$=CHR$(13) THEN INPUT#3,B:PRINT B:GOTO 130
90 INPUT#3, B$
100 PRINT B$;" IS NOT NUMERIC"
110 GET#2, A$
120 IF A$<>CHR$(13) GOTO 110
130 NEXT I
140 CLOSE 2
150 CLOSE 3
RUN
 123
79PIN IS NOT NUMERIC
 345000

READY,
```

15. An existing disk file may be both OPENed for reading and OPENed for rewriting at the same time. This is handy for adding information to a file. Rewriting an old file does not actually “destroy” the old contents, but builds a new file using the old name. Until the newly written file is CLOSEd, the old contents can be accessed through an OPEN statement. For example, the following program inserts the contents of the file NEWNAMES at the beginning of the file OLDNAMES. The STATUS function is used in lines 60 and 120 to determine when all characters have been read from the files referenced by the preceding GET statements in lines 50 and 110.

```
10 OPEN 2,8,2,"@:OLDNAMES,S,W"
20 OPEN 3,8,3,"OLDNAMES,S,R"
30 OPEN 4,8,4,"NEWNAMES,S,R"
40 HALT=0
50 GET#4, A$
60 IF STATUS<>0 THEN HALT=1
70 PRINT#2, A$;
80 IF HALT<>1 THEN 50
90 PRINT "NEWNAMES COPIED"
100 HALT=0
110 GET#3, A$
120 IF STATUS<>0 THEN HALT=1
130 PRINT#2, A$;
140 IF HALT<>1 THEN 110
150 PRINT "OLDNAMES RECOPIED"
160 CLOSE 2
```

# OPEN

```
170 CLOSE 3
180 CLOSE 4
```

## 16. The statement

OPEN N,3

allows data to be displayed on the screen with statements of the form PRINT#N,data, and read from the screen with statements of the form INPUT#N,var or GET#N,var. Of course, since the screen is the default output device, the statement PRINT displays data on the screen directly, without first requiring an OPEN statement.

```
10 OPEN 12,3
20 PRINT#12, "{SHIFT-CLR/HOME}";
30 PRINT#12, "PRINT AND READ THE SCREEN,"
40 PRINT "{CLR/HOME}";
50 INPUT#12, A$
60 PRINT A$
70 CLOSE 12
RUN
(The screen is cleared.)
PRINT AND READ THE SCREEN.
PRINT AND READ THE SCREEN.
```

READY.

Messages can be displayed on the screen using the normal PRINT statement, as in lines 40 and 60, or through the OPEN N,3 and PRINT#N statements as in lines 10, 20, and 30. When the screen is read, as in line 50, reading begins with the current cursor position. Thus, line 40 was used to home the cursor so that the first line on the screen could be read.

## 17. The statement

OPEN N,0

allows data to be read from the keyboard using statements of the form INPUT#N,var or GET#N,var. Note: Since the keyboard is the default input device, the statements INPUT and GET receive data from the keyboard directly, without first requiring an OPEN statement.

```
10 INPUT A
20 PRINT A
30 OPEN 1,0
```

```

40 INPUT#1, B
50 PRINT B
60 CLOSE 1
RUN
? 34
 34
87.6 87.6

READY.

```

Line 10 provides normal access to the keyboard with the INPUT statement, while lines 30 and 40 access the keyboard through the OPEN N,0 and INPUT#N statements. Access to the keyboard through the INPUT# statement versus the INPUT statement has two significant differences. First, note that INPUT# provides no prompt. This can be remedied by using a PRINT statement preceding the INPUT# statement. Second, note that after the user gave the response 87.6 and hit RETURN, the cursor did not move to a new line. Instead, the cursor advanced one space when the RETURN key was pressed, and then waited for the next piece of output (the value PRINTed by line 50). The fact that the cursor does not proceed to the next line with INPUT# allows flexibility in program design.

18. When OPEN N,0 is used to access the keyboard, the statements GET# and INPUT# may be used to read the contents of the keyboard buffer. However, attempting to use the statement PRINT# to place values in the buffer results in the ?NOT OUTPUT FILE ERROR message.

19. Access to the built-in RS-232 interface is given by the statement

```
OPEN N,2,C,"specifications"
```

where C (0 to 255) is referred to as the communications channel. RS-232 channels are used to provide connections to any modem, printer, or other device through an RS-232 cable. Modems allow computers to communicate with other computers by telephone. (See Appendix P for a discussion of *specifications*.)

20. The reference number N assigned to a file or device by the OPEN statement may actually be *any* whole number from 0 to 255. However, when a PRINT#N statement is used with N greater than 127, BASIC places a CHR\$(10) character after each RETURN character, CHR\$(13), that is printed to the file or device. This is needed with monitors or printers which interpret CHR\$(13) as the symbol for "go to the first

# OPEN

position on the current line", and use CHR\$(10) as the symbol for "move down to next line".

```
10 OPEN 200,8,2,"COURSES,S,W"
20 PRINT#200,"MATH": PRINT#200,"MUSIC"
30 CLOSE 200
40 OPEN 3,8,3,"COURSES,S,R"
50 FOR I=1 TO 13
60 GET#3,A$: PRINT A$; ASC(A$);
70 NEXT
RUN
M 77 A 65 T 84 H 72
 13 10 M 77 U 85 S 83 I 73 C 67
 13 10
READY.
```

## Applications

1. One of the primary uses of computers is the processing of data. This data usually is stored on cassette or disk. The OPEN statement provides access to this data.
2. Printouts of the listing and runs of a program are useful both for debugging and for showing the program and its output to others. The OPEN statement, together with the CMD statement, makes it easy to get these printouts.
3. Many information and computer networks exist which the COM-MODORE 64 can tie into via the telephone. The OPEN statement gives the means of accessing a modem via the RS-232 interface, for telephone communications.

The Commodore 64 contains 65536 Random Access Memory (64K RAM) locations in which data can be stored. This data consists of such items as the current BASIC program, variables, and screen display. (See Appendix H for a breakdown of the uses of RAM.) Each of the 64K RAM locations has an *address* associated with it and contains a number from 0 to 255. If N is a whole number from 0 to 65535, then the value of

PEEK(N)

will be the number contained in the memory location with address N.

## Examples

1. The BASIC error messages can be found beginning with the memory location having address 41374. The following program PEEKs into memory to retrieve the first three messages.

```
10 FOR N = 41374 TO 41409
20 A = PEEK(N)
30 PRINT A;
40 NEXT N
50 PRINT
60 FOR N = 41374 TO 41409
70 PRINT CHR$(PEEK(N));
80 NEXT N
90 PRINT
RUN
 84 79 79 32 77 65 78 89 32 70
 73 76 69 211 70 73 76 69 32 79
 80 69 206 70 73 76 69 32 78 7
9 84 32 79 80 69 206
TOO MANY FILES FILE OPEN/FILE NOT OPEN
READY,
```

2. The keyboard buffer consists of the 10 memory locations with addresses 631 to 640. The following lines could be placed within a program to print out the contents of the keyboard buffer. Unlike GET or INPUT, these lines of code do not delete characters from the buffer as it is read.

```
1000 FOR I=631 TO 640
1010 PRINT CHR$(PEEK(I));
1020 NEXT I
```

# PEEK

## Comments

1. For most memory locations, PEEKing into the location does not change the value stored there. In a very few cases, however, reading the contents of the memory location with PEEK will set the number in the location to 0. The locations 53278 and 53279, which are used to detect collisions involving SPRITEs, have this property.

2. Do not be afraid to PEEK around in memory. Try the following program which PEEKs through the BASIC interpreter and produces a touch of computer art.

```
10 FOR I=40960 TO 49151
20 T = PEEK(I)
30 IF T<>13 THEN PRINT CHR$(T);
40 NEXT I
```

3. The N in PEEK N may be a numeric constant, variable, or expression and may have either integer or floating-point precision. If N is not a whole number, PEEK uses the truncated value of N, and if this value is not between 0 and 65535, the message ?ILLEGAL QUANTITY ERROR results.

4. The statement POKE is complimentary to PEEK. It is used to place numbers into memory locations.

## Applications

1. PEEK is used extensively when working with machine language programs, SPRITE graphics, and music synthesis. See the sections on these latter two topics for a sampling of the many uses of PEEK.

2. The PEEK statement is used within the POKE statement to read the current value stored in a memory location so that the new value POKEd can be based in part on the old value. For example, location 53269 controls which of the 8 possible SPRITEs is enabled. The statement POKE 53269,16 will turn on SPRITE 4, but turn off any other SPRITE which was on. On the other hand, the statement POKE 53269, PEEK(53269) OR 16 will turn on SPRITE 4 without turning off any other SPRITEs which may be on.



The Commodore 64 contains 65536 Random Access Memory (64K RAM) locations in which data can be stored. This data consists of such items as the current BASIC program, variables, and the screen display. (See Appendix H for a breakdown of the uses of RAM.) Each of the 64K RAM locations has an *address* associated with it and contains a number from 0 to 255. If N is a whole number from 0 through 65535, and M is a whole number from 0 to 255, then the statement

```
POKE N,M
```

changes the value contained in the memory location with address N to the number M.

## Examples

1. The colors to be used for the border and background on the screen are controlled by the values in memory locations 53280 and 53281. Appendix C gives the 16 different colors available on the screen, along with the value POKEd to obtain each color. The following program steps through all the possible combinations of background and border colors. Press any key except Q to stop the progression, then any key except Q to start it again. Press Q to quit the program. The border and background will return to their original colors.

```
10 A=PEEK(53280): B=PEEK(53281)
20 FOR I=1 TO 16
30 FOR J=1 TO 16
40 POKE 53280,I: POKE 53281,J
50 PRINT "{SHIFT-CLR/HOME}";
60 PRINT "BORDER IS COLOR";I
70 PRINT "BACKGROUND IS COLOR";J
80 FOR K=1 TO 200
90 GET A$: IF A$<>" " THEN 140
100 NEXT K
110 NEXT J
120 NEXT I
130 GOTO 170
140 IF A$="Q" THEN 170
150 GET A$: IF A$="" THEN 150
160 IF A$<>"Q" THEN 110
170 POKE 53280,A: POKE 53281,B
180 PRINT "{SHIFT-CLR/HOME}"
```

2. RAM locations 54272 through 54296 control the music synthesis

# POKE

chip. The following program uses some of these locations to create a warning beeper.

```
10 POKE 54296, 15
20 POKE 54278, 240
30 POKE 54273, 80
40 POKE 54276, 33
50 FOR I=1 TO 80: NEXT I
60 POKE 54276, 0
70 FOR I=1 TO 20: NEXT I
80 GET A$: IF A$="" THEN 40
```

Type any key to stop the beeper.

## Comments

1. The memory locations with addresses 55296 through 56319, which are used to hold the color for each character displayed on the screen, will each only hold a number from 0 to 15. POKEing the number  $N$  into one of these locations results in the number  $N \text{ MOD } 16$  being placed in the location. (Note:  $N \text{ MOD } 16$  is the remainder when  $N$  is divided by 16.)

2. Whereas PEEKing around in memory does not alter the way the computer is running, POKEing around in memory may produce some strange results. For instance, memory location 214 records the number of the line on which the cursor is being displayed. Try POKEing a number greater than 24 into location 214 and then hit RETURN repeatedly. Type RUN/STOP-RESTORE to get back to normal.

Though POKEing around can produce strange results, do not be afraid to experiment with POKES. They are one of the most powerful ways of controlling the computer, and the worst that can come from POKEing around is having to turn the computer off and on again to reset things to normal. There is no way you can harm the computer by using POKE.

3. The  $N$  and  $M$  in POKE  $N,M$  may be numeric constants, variables, or expressions and may have either integer or floating-point precision. Whenever the values of  $N$  and  $M$  are not whole numbers, POKE uses their truncated values. The (truncated) value of  $N$  must lie between 0 and 65535 inclusive, and the (truncated) value of  $M$  between 0 and 255 inclusive, or the message ?ILLEGAL QUANTITY ERROR results.

4. The statement POKE is complementary to PEEK, which is used to read the contents of memory locations.

## Applications

1. POKE is the key to SPRITE graphics and music synthesis. See the sections on these latter two topics for a sampling of the many uses of POKE.
2. POKE is used to place machine language programs into unused parts of memory. These programs can then be used as subroutines by BASIC. See SYS and USR for further details.

# POS

The Commodore 64 can display 40 characters on a line of the screen. A single line of a BASIC program can contain up to 80 characters. If a program line contains more than 40 characters, the pair of physical lines of the screen holding the program line are referred to as a single logical line. Logical lines of up to 80 characters also arise when PRINT statements cause strings of more than 40 characters to be displayed. The column positions of the characters in each logical line are numbered from 0 to 39 (or from 0 to 79), beginning at the left side of the screen (and possibly continuing on to a second physical line for positions 40 to 79). At any time, the value of

POS(0)

will be the number of the column position of the cursor.

## Examples

```
1. 10 PRINT "E";: A = POS(0)
 20 PRINT " PLURIBUS";: B = POS(0)
 30 PRINT " UNUM";: C = POS(0)
 40 PRINT A; B; C
 RUN
 E PLURIBUS UNUM 1 10 15
```

READY.

```
2. PRINT "0123456789012345678901234567890123456789012
 345678901234567";POS(0)
 0123456789012345678901234567890123456789
 01234567 48
```

READY.

3. In the following program, POS is used to assure that each line of the display will be, at most, 30 characters long.

```
10 FOR I = 1 TO 35
 20 READ A$
 30 IF LEN(A$)>30-POS(0) THEN PRINT
 40 PRINT A$+" ";
 50 NEXT
 60 DATA ABS,ASC,ATN,CHR,CLOSE,CLR,CMD,CO
 NT,COS,DATA,DEF FN,DIM,END,EXP,FOR,FRE
 70 DATA GET,GOSUB,GOTO,IF,INPUT,INT,LEFT
```

```
,LEN,LET,LIST,LOAD,LOG,MID,NEW,NEXT,NOT
80 DATA ON,OPEN,PEEK
RUN
ABS ASC ATN CHR CLOSE CLR CMD
CONT COS DATA DEF FN DIM END
EXP FOR FRE GET GOSUB GOTO IF
INPUT INT LEFT LEN LET LIST
LOAD LOG MID NEW NEXT NOT ON
OPEN PEEK
READY.
```

## Comments

1. The argument of the POS function is a dummy argument. Any number, not just 0, can be used with the same result. For instance, we can write POS(1) or POS(-6).
2. The value of POS is taken from memory location 211, where BASIC stores the current column position of the cursor.

## Applications

1. The function POS can be used when PRINTing user-supplied data, to determine the amount of space remaining on the current line, from which a decision can be made whether or not to PRINT the next item on the current line:

```
IF LEN(A$)<= 39-POS(0) THEN PRINT A$;
```

# PRINT

Without the ability to display the results of calculations and data manipulations, a computer is almost useless. Thus, the single most common BASIC statement is probably the PRINT statement, which tells the computer to display information on the screen.

Information to be displayed falls into two categories: string data and numeric data. A string may contain any of the 256 characters which can be entered at the keyboard or produced by the CHR\$ function. Appendix A lists these characters along with their ASCII/Commodore values. The characters with ASCII/Commodore values 0 through 31 and 128 through 159 often are called *control* characters. In standard printing mode, control characters do not display, but instead have an effect on the cursor position or the manner in which later characters are displayed.

If A\$ is a string, then the statement

```
PRINT A$
```

causes the characters in the string to be displayed or to produce their special effect. For instance, if A\$ consists of the three characters T, CuRSoR down, and U, then the result of the statement PRINT A\$ will be

```
T
U
```

```
READY.
```

If N is a number, then the statement

```
PRINT N
```

causes the number to be displayed on the screen. The PRINT statement automatically inserts a space following the displayed number. Also, a space is inserted preceding positive numbers and zero. Negative numbers are preceded by "-". For instance

```
PRINT 123: PRINT -3.2
123
-3.2
```

```
READY.
```

# PRINT

A line on the screen consists of 40 columns numbered 0 to 39, and is divided into 4 *print zones*, each of width 10. The first zone consists of columns 0 to 9, the second zone of columns 10 to 19, and so on.

Normally, PRINT statements execute a carriage return and line feed after displaying characters on the screen. This causes the next PRINT statement or prompt from an INPUT statement to display its characters on the next line. However, if a PRINT statement is followed by a comma, the carriage return and line feed are suppressed, and the next PRINT statement or prompt from an INPUT statement displays its characters beginning in the first position of the next unused print zone. Similarly, if a PRINT statement is followed by a semicolon, the carriage return and line feed are suppressed, and the next PRINT statement or prompt from an INPUT statement displays its characters immediately following the first set of characters. For instance

```
PRINT "APPLE",: PRINT "ORANGE"
APPLE ORANGE
```

```
READY,
PRINT 365;: PRINT "DAYS IN A YEAR"
 365 DAYS IN A YEAR
```

```
READY,
10 PRINT "WHEN IS HALL";
20 INPUT "OWEEN";A$
RUN
WHEN IS HALLOWEEN? OCTOBER 31
```

```
READY,
```

Note that the space before and after 365 should be thought of as part of the number, and thus DAYS follows immediately after the number 365.

Commas and semicolons can be used to condense several PRINT statements into one multiple PRINT statement. A multiple statement consists of several values separated by commas or semicolons. For instance, the line

```
PRINT A$,: PRINT B;: PRINT C;: PRINT D$
```

can be written as

```
PRINT A$, B; C; D$
```

# PRINT

## Examples

```
1. 10 PRINT "TEN",
 20 PRINT "DOLL";
 30 PRINT "ARS"
 RUN
 TEN DOLLARS
```

READY.

```
2. 10 PRINT "ZONE 1", "ZONE 2"
 20 PRINT 1, 2; 3
 RUN
 ZONE 1 ZONE 2
 1 2 3
```

READY.

The digit 1 is displayed in column 1 (the second position on the line), after the leading space in column 0. Similarly, the digit 2 is displayed in column 11 following the leading space which is displayed in column 10, the beginning of zone 2. The digit 3 is displayed in column 14, separated from the digit 2 by the trailing space after the 2 and the leading space before the 3. Recall that a positive number is always preceded and followed by a space.

```
3. PRINT "A";"B","C","D";"E";"F";"GEE","H"
 AB C DEFGEE H
```

READY.

```
4. PRINT "TUNE", TUNE
 TUNE 0
```

READY.

The first item is a string constant and the second is a numeric variable, which has value 0 until assigned another value.

## Comments

1. The word PRINT can be abbreviated as a question mark. This can be very convenient in direct mode to save typing. If used in a program, time may be saved in typing, but no memory is saved. When LISTed, a program entered using ? for PRINT will show PRINT.



# PRINT

```
? 23+34
57
```

```
READY,
10 ? 12*13
LIST
10 PRINT 12*13
```

```
READY,
```

2. The A\$ in the statement PRINT A\$ may be a string constant, string variable, or expression involving strings. Expressions and variables will be evaluated and then PRINTed.

```
10 LET A$="TENNIS": B$="BALL"
20 PRINT "SPORTS", A$, "BASKET"+B$;
RUN
SPORTS TENNIS BASKETBALL
```

```
READY,
```

3. The N in a PRINT N statement may be a numeric constant, variable, or expression and may have either integer or floating-point precision. Expressions and variables will be evaluated and then PRINTed.

```
10 LET A%=3: B=2.5
20 PRINT 34; A%; A%+B; -6*B+5
RUN
34 3 5.5 -10
```

```
READY,
```

4. PRINT statements often display numbers in a different form than they are originally typed. The displayed form is the one most appropriate to the precision of the number. Some examples are:

| <u>Original form</u> | <u>Displayed form</u> |
|----------------------|-----------------------|
| 3.20                 | 3.2                   |
| 2E+4                 | 20000                 |
| 1234567654321        | 1.234567654E+13       |
| 624E+11              | 6.24E+13              |

5. Semicolons separating two items in a multiple PRINT statement often may be omitted. Semicolons may be omitted after any string variable, string constant, BASIC or user-defined function, integer numeric variable (ending with %), and numeric constants not followed

# PRINT

by another numeric constant. In addition, semicolons may be omitted before any string constant.

```
A$="CAT":PRINT "A" 23 A$ "B" "C"
A 23 CATBC
```

```
READY,
PRINT "E" "F" 2.34CHR$(65)
EF 2.34 A
```

```
READY,
```

Semicolons cannot be omitted after floating-point numeric variables, except when the next item is a string constant.

```
10 A=5: B=8
20 PRINT A B "PROBLEMS"
30 PRINT A;B "OK"
RUN
 0 PROBLEMS
 5 8 OK
```

```
READY,
```

In line 20, A B is treated as the single variable AB. Since this variable has not been assigned a value, its value is taken to be 0.

Note: Because semicolons cannot be omitted in every case, we have chosen to include them at all times in this handbook.

6. A PRINT statement with no list of items will simply produce a carriage return and line feed. Thus, such PRINT statements can be used to generate blank lines. Note that PRINT by itself need not always produce a blank line. If the previously executed PRINT statement ended with a comma or semicolon, then PRINT by itself will simply supply the carriage return and line feed which had been suppressed. The statement PRINT "", which prints the empty string, has the same effect as PRINT by itself.

```
10 PRINT "ART"
20 PRINT
30 PRINT "SCULPTURE"
RUN
ART
```

```
SCULPTURE
```

# PRINT

```
READY,

10 FOR I=1 TO 10
20 PRINT I;
30 NEXT I
40 PRINT
50 PRINT "I CAN COUNT,"
RUN
 1 2 3 4 5 6 7 8 9 10
I CAN COUNT,

READY,
```

In the second program, line 40 produced the carriage return and line feed which line 20 had suppressed. Without line 40, the message in line 50 would have begun on the same line on which the numbers were displayed.

7. If an item is too long to fit entirely on the remainder of the current line, as much of the item as possible will be printed on the current line, with the remainder of the item *wrapping around* to the beginning of the next line.

```
PRINT 1, 2, 3, "THIS ITEM MUST WRAP"
 1 2 3 THIS ITEM
MUST WRAP

READY,
```

8. When a comma separates two items in a multiple PRINT statement, there will always be at least one space separating the displayed items. Thus, if the first item ends in the last column of a zone, the next print zone will be skipped, and the second item printed in the following zone.

```
PRINT "TEN SPACES", "SO ZONE SKIPPED"
TEN SPACES SO ZONE SKIPPED

READY,

PRINT 12345678, "THIRD ZONE"
 12345678 THIRD ZONE

READY,
```

In the second example, the digit 8 is not in the last column of the first zone, but the trailing space is. Since this space counts as part of the

# PRINT

numeric item, the space required between items forces the string constant to be displayed in the third zone.

9. Displaying quotation marks is often accomplished using the CHR\$ function. For example, the statement PRINT CHR\$(34); CHR\$(34) displays two successive quotation marks.

10. A number of keys and key combinations produce special effects rather than displaying a symbol. These include the cursor control keys, and the combination of a number key with the ConTRoL or Commodore keys. The special effects produced can include moving the cursor in any of 4 directions, changing the color used to display additional characters, and setting reverse-video mode. These *control* characters can be placed into strings using *quote* mode or INsert mode (see Comments 11 and 12). By using strings which contain cursor control and color control characters, the PRINT statement can produce a wide variety of graphic displays.

11. Quote Mode: While the user is entering code, quote mode is active whenever an odd number of quotation marks have been typed on the current (logical) line. This is the normal situation whenever a string constant is being placed in a program. Quote mode is cancelled when an even number of quotation marks have been typed or whenever the RETURN key is pressed. In quote mode, control characters do not produce their special effects (except as noted in Comment 13 below), but instead display as reverse-video characters. For instance, typing the statement A\$ = "{CTRL-3}AB" will produce A\$ = "▒AB" on the screen. Appendix A includes a list of the control characters (those having ASCII/Commodore values 0 through 31 and 128 through 159) and the symbols which appear for them in quote mode. Special effect characters also can be placed in strings by using the CHR\$ function.

When PRINT statements are executed, quote mode is active whenever an odd number of quotation marks have been PRINTed on the current logical line of the screen. If so, any control characters in a string will produce reverse-video symbols when PRINTed. Otherwise, they produce their special effects. For instance, if quote mode is not active and A\$ has the value assigned to it above, then the statement PRINT A\$ will print the letters AB using red characters. Thus, strings can be assigned values which, when PRINTed, move the cursor, change the color of characters, cause characters to print in reverse-video, or produce any of the other special effects shown in Appendix A.

By using the cursor-down character in line 40, the following program displays your favorite word diagonally. LEN(A\$) is the number of characters in A\$, and MID\$(A\$,I,1) is the Ith character in A\$.

```
10 PRINT "{SHIFT-CLR/HOME}"
20 INPUT "YOUR FAVORITE WORD";A$
```

# PRINT

```
30 FOR I = 1 TO LEN(A$)
40 PRINT MID$(A$,I,1); "{CURSOR DOWN}";
50 NEXT I
RUN
YOUR FAVORITE WORD? FOOD
F
 D
 D
 D
```

READY.

Consider the following program:

```
10 PRINT "ONCE UPON A TIME"
20 PRINT "THERE WAS A FROG"
RUN
ONCE UPON A TIME
THERE WAS A FROG
```

READY.

The quote mode characters **UPON** and **TIME** represent the special effects "reverse-video ON" and "reverse-video OFF", which are obtained by typing CTRL-9 and CTRL-0. Reverse-video is also turned off when RETURN is typed by the user or produced by the PRINT statement. Notice that line 20 did not appear in reverse-video even though CTRL-0 was not PRINTED.

In the following statement, quote mode is established for the display by first printing a quotation mark.

```
PRINT CHR$(34);" \VLVn-! /Xo|!++"
" \VLVn-! /Xo|!++"
```



READY.

12. **INSert Mode:** **INSert** mode is often used when editing a statement. It is in active for each space created by typing **SHIFT-INS/DEL**, until a character is typed in the space, or **RETURN** is pressed. When **INSert** mode is active, special effect characters do not produce an effect (except as noted in Comment 13 below), but display as reverse-video characters. Thus it is easy to place additional special effect characters into a string while editing.

13. There are six special effects that cannot be entered into a string using quote or **INSert** mode. The simplest way to place these charac-

# PRINT

ters into a string is to use the CHR\$ function. The six characters and their corresponding values of CHR\$ are RETURN, CHR\$(13); SHIFT-RETURN, CHR\$(141); switch to Lowercase/Uppercase character mode, CHR\$(14); switch to Uppercase/Graphics character mode, CHR\$(142); disable character set switching keys, CHR\$(8); and enable character set switching keys, CHR\$(9). In addition, DELete cannot be placed in a string in quote mode (use CHR\$(20) or INSert mode), and INSert cannot be placed in a string in INSert mode (use CHR\$(148) or quote mode).

14. There are two ways that the symbols corresponding to special effects can be displayed by the PRINT statement, rather than producing their effects. The first is to note that all special effect symbols are reverse-video characters. Thus, for instance, the symbol , which corresponds to the effect "cursor down", can be displayed by the statement PRINT " Q". The first character inside the quotes is CTRL-9 or "reverse-video on". A second way to display the symbols for special effects, or, from another point of view, to prevent most characters PRINTed from having a special effect, is to PRINT a quote as the first character of the line. Note, however, that there are 7 characters mentioned in Comment 13 that will not be affected. In particular, DELete is not affected, and can be used to erase the quote once it has been PRINTed. The following example illustrates this.

```
PRINT CHR$(34);CHR$(20);"-♥| ↯Xo◊ |◆"
-♥| ↯Xo◊ |◆
READY.
```

Even though the quote was erased, quote mode remained in effect.

15. The PRINT statement normally displays characters on the screen. However, the CMD statement allows PRINT statements to send their output to the cassette, disk, printer or other devices. See the discussion of CMD for details.

16. BASIC provides two special functions, TAB and SPC, which are used in conjunction with the PRINT statement to format data on the screen. See the sections on these functions for further details.

## Applications

1. The PRINT statement is indispensable for providing output from a program.
2. PRINT is a useful debugging tool. After a part of a program has been executed, intermediate results can be displayed.

A *peripheral device*, or simply *device*, is an object that is connected to the central processing unit of the computer. Each device has an identifying number associated with it. The following table lists some common devices and their device numbers.

| <u>Device</u>    | <u>Device Number</u> |
|------------------|----------------------|
| Cassette player  | 1                    |
| Disk drive       | 8                    |
| Printer          | 4                    |
| Screen           | 3                    |
| RS-232 interface | 2                    |

The PRINT# statement sends (or writes) information to the above devices in much the same way that the PRINT statement displays information on the screen. The following discussion assumes familiarity with the PRINT statement.

Before information can be sent to a file or device, an OPEN statement first must be executed to access the file or device and associate a reference number N with it. (See Appendix F and the discussion of OPEN for further details.) Items of information then can be sent, in order, to the file or device. If A\$ is a string, then the statement

```
PRINT#N, A$
```

sends the string A\$ to the file or device with reference number N. If A is a number, then the statement

```
PRINT#N, A
```

sends the number A, with a trailing space (and a leading space if the number is positive or zero) to the file or device with reference number N.

The PRINT# statement writes information on the screen in exactly the same way as the PRINT statement. However, PRINT# differs from PRINT in three ways when writing information onto the printer or into a data file on cassette or disk. First, control characters affect the printer in a completely different manner than they affect output to the screen (see Appendix K), and do not have any special effects when written to cassette or disk files. When written to cassette or disk files, control characters are simply recorded like any other characters. (See Appendix F for a discussion of how data is recorded in cassette and disk files.) Second, no print zones exist on the printer or in a data file. When commas are used to separate items in PRINT# statements to the printer or to cassette or disk files, 10 spaces are written between

# PRINT#

each pair of items. Third, the TAB function, which often is used in conjunction with PRINT, does not work as expected with PRINT#. See the discussion of TAB for further details.

If a PRINT# statement does not end with a semicolon or a comma, the newly entered pieces of information are followed automatically with a RETURN character. In the examples in this discussion, the RETURN character is denoted by <R>.

## Examples

```
1. 10 OPEN 1,4: REM DEVICE 4 IS THE PRINTER
 20 PRINT#1, "SAN FRANCISCO", 1906
 30 PRINT#1, "SAN FRANCISCO"; 1906
 40 CLOSE 1
 RUN

 READY.
```

The following appeared on the printer:

```
SAN FRANCISCO 1906
SAN FRANCISCO 1906
```

In the first line, 11 spaces are printed between SAN FRANCISCO and 1906; 10 for the comma and 1 for the leading space with the number.

```
2. 10 OPEN 1,1,1,"DUES"
 20 PRINT#1, "NEIL"; 20
 30 PRINT#1, "JEFF"; 60
 40 CLOSE 1
 RUN
 PRESS PLAY & RECORD ON TAPE
 OK

 READY.
```

The following characters are now stored in the cassette file DUES:

```
NEIL 20 <R>JEFF 60 <R><EOF>
```

The symbol <EOF> represents the End-Of-File character for cassette files, CHR\$(0). CHR\$(0) is written to the file when the CLOSE statement is executed.



## PRINT#

```
3. 10 OPEN 3,8,3,"SYMBOLS,S,W"
20 PRINT#3, CHR$(34);CHR$(44);
30 PRINT#3, CHR$(58);CHR$(34)
40 CLOSE 3
RUN
```

READY.

The following characters are now stored in the disk file SYMBOLS:

```
" , : "<R>
```

Any character can be placed in a file using the CHR\$ function. In particular, CHR\$ is used to place quotes around items in a file and RETURNS between items. Note also that disk files do not contain an <EOF> symbol. Instead, the length of the file is recorded elsewhere on the disk.

```
4. 10 OPEN 1,4
20 PRINT#1, CHR$(14);
30 PRINT#1, "THIS ISN'T LOWERCASE"
40 CLOSE 1
RUN
```

READY.

The following appeared on the printer:

```
THIS ISN'T LOWERCASE
```

Note that when working with the screen, CHR\$(14) switches the display to the Lowercase/Uppercase character set. However, the printer has a different interpretation of CHR\$(14): switch to printing double-width characters. See Appendix K for a discussion of the control codes used with the printer.

```
5. 10 OPEN 4,8,4,"CODES,S,W"
20 PRINT#4, "LALBLC"
30 CLOSE 4
RUN
```

READY.

The following characters are now stored in the disk file CODES:

```
LALBLC<R>
```

# PRINT#

Note that the characters `␣`, `␣`, and `␣`, which are control codes when used with the screen, are stored the same as any other character in the disk file.

## Comments

1. The character `#` must immediately follow the `T` in `PRINT#`. If a space is inserted, the message `?SYNTAX ERROR` results. However, spaces may be placed after `#`. For example, `PRINT #2` is not permitted, whereas `PRINT# 2` is permitted.
2. If the number `N` in a `PRINT#` statement has not been associated with a file or device by an `OPEN` statement, then the message `?FILE NOT OPEN ERROR` will result when the `PRINT#` statement is executed.
3. The `OPEN` statement does not check to see if the printer or disk drive is connected to the system or is turned on. This checking occurs when a statement like `PRINT#` tries to access these devices. If the printer or disk drive is disconnected or not turned on, the message `?DEVICE NOT PRESENT ERROR` will result when a `PRINT#` statement attempts to write data to the device.
4. Attempting to use `PRINT#` to write data to the keyboard buffer, device 0, or to a cassette file that has been `OPENed` for *reading*, results in the `?FILE NOT OUTPUT FILE ERROR` message. Attempting to use `PRINT#` to write data to a disk file that has been `OPENed` for *reading* does not produce an error, but also has no effect on the file. The situation is the same as if the `PRINT#` statement had not occurred.
5. When a `PRINT#` statement is executed to place data on a cassette or disk, you will often not hear the cassette player or disk drive operating. The computer stores the pieces of information in a *buffer*, a reserved portion of memory, and then physically records them onto the cassette or disk when the buffer is filled. `CLOSE` statements cause all information remaining in the buffer to be recorded.
6. The null character, `CHR$(0)`, should be avoided when writing data to a cassette or disk file, particularly if the `INPUT#` statement will be used to read the data from the file. The null character has a special meaning when files are read with `INPUT#` which can cause subsequent data to be ignored, or can even cause the computer to go into an infinite waiting loop. See the discussion of `INPUT#` for further details.
7. If the `INPUT#` statement is to be used to read data placed in a file by `PRINT#`, then the individual items in the file must be separated

## PRINT#

from each other by the RETURN character. This is most easily done by setting a string variable equal to CHR\$(13), and then PRINTing this variable between successive items.

```
10 OPEN 2,8,2,"WEIGHTS,S,W"
20 R$ = CHR$(13)
30 PRINT#2, "MERRIAM";R$;56;R$;
40 PRINT#2, "KIM";R$;74;R$;
50 CLOSE 2
RUN
```

READY.

The following characters are now stored in the disk file WEIGHTS:

```
MERRIAM<R> 56 <R>KIM<R> 74 <R>
```

## Applications

1. Cassette and disk files provide compact storage for data. This data then can be accessed by other programs. The PRINT# statement is used to construct these files.

# READ

READ statements are only used in conjunction with DATA statements. DATA statements store a list of constants, and READ statements assign successive values from the list to variables.

Suppose that a program has just one DATA statement consisting of, say, several string constants, and the first READ statement to be encountered is the statement

```
READ A$
```

This statement results in the variable A\$ being assigned the first constant in the DATA statement. The statement READ A\$ is like a LET statement. It's as if we said "LET A\$ = the first constant in the list specified by the DATA statement." If the next READ statement encountered is

```
READ B$
```

then the variable B\$ will be assigned the second constant in the DATA statement as its value. Successive READ statements assign successive constants to variables.

```
10 DATA RED, WHITE, BLUE, "JULY 4, 1776"
20 READ A$
30 READ B$
40 READ C$
50 READ D$
60 PRINT A$, B$, C$, D$
RUN
RED WHITE BLUE JULY 4,177
6
```

```
READY.
```

A single READ statement can assign values to several variables at once. For instance, lines 20 - 50 could have been replaced by the single line

```
20 READ A$, B$, C$, D$
```

In general, a program might have several DATA statements appearing at various places in the program. READ statements treat the constants as if they are combined into one long DATA statement, with the constants from the lowest numbered DATA statement coming first, the constants from the next higher numbered DATA statement coming next, and so on.

# READ

Up to now, we have focused on DATA statements consisting only of string constants. However, DATA statements also can consist of numeric constants or a mixture of numeric and string constants. The important point to keep in mind is that each variable appearing in a READ statement should be of the same type as the constant assigned to it. Otherwise, the result will be different than intended (see Example 5) or the message ?SYNTAX ERROR will result (see Example 4).

## Examples

```
1. 10 DATA JAN, FEB, MAR
 20 DATA 23, 17, 6
 30 READ A$, B$, C$
 40 PRINT A$, B$, C$
 50 READ A, B, C
 60 PRINT A, B, C
 RUN
 JAN FEB MAR
 23 17 6

 READY.
```

```
2. 10 READ A, B$
 20 DATA 123.45
 30 READ C$, D
 40 DATA SIX, SEVEN, 8, 9
 50 PRINT A, B$, C$, D
 RUN
 123.45 SIX SEVEN 8

 READY.
```

Consider line 10. After assigning the value 123.45 to variable A, all the values in line 20 were used. Hence, the next DATA statement (line 40) was entered to find the next value. Also notice that the value 9 in line 40 was never assigned to a variable. There can be an excess of constants, but not an excess of variables.

```
3. 10 GOTO 30
 20 DATA TWENTY
 30 READ A$
 40 DATA FORTY
 50 PRINT A$
```

# READ

```
RUN
TWENTY

READY,
```

The program seems to skip line 20 and take no notice of it. However, the READ statement does not care that line 20 was skipped. It searches through the entire program and finds the lowest numbered DATA statement, no matter where the statement occurs.

```
4. 10 DATA TEN
 20 READ A
 30 PRINT A
 RUN

 ?SYNTAX ERROR IN 10
 READY,
```

```
5. 10 DATA 10
 20 READ A$
 30 PRINT A$
 RUN
 10

 READY,
```

Did you expect this program to produce an error statement? The READ statement treated 10 as a string constant and assigned it to the string variable A\$. Had line 30 been PRINT A\$ + 5, the result would not be 15 but rather the message ?TYPE MISMATCH ERROR.

## Comments

1. See the discussion of the DATA statement for some useful comments about placing string and numeric constants in DATA statements.
2. If the DATA statements contain fewer constants than are called for by the READ statements, the ?OUT OF DATA ERROR message results.

```
10 FOR I = 1 TO 4
20 READ A
30 PRINT A;
40 NEXT I
50 DATA 6, 7, 8
```

# READ

```
RUN
 6 7 8
?OUT OF DATA ERROR IN 20
READY,
```

Note that the line number given with the error message is that of the READ statement, not the DATA statement.

3. The RESTORE statement can be used in a program to cause subsequent READ statements to reuse the DATA statements of the program, beginning with the lowest numbered DATA statement. See the discussion of RESTORE for further details.

```
10 READ A,B,C
20 DATA 1,2,3
30 RESTORE
40 READ D,E
50 PRINT A;B;C;D;E
RUN
 1 2 3 1 2

READY,
```

4. There are two precisions of numeric constants and variables: integer and floating-point. Numeric variables occurring in READ statements usually can be assigned numeric constants having different precisions. However, if the variable specified in the READ statement is of type integer (ends with %), and the corresponding constant in the DATA statement is of type floating-point, then the floating-point constant will be converted to the largest integer constant that is less than or equal to it (if possible). Integer constants assigned to floating-point variables are converted with no change in value.

```
10 DATA 2.6
20 READ A%
30 PRINT A%
RUN
 2

READY,
```

The floating-point constant 2.6 was truncated to 2 when converted to an integer constant.

# REM

It is a good idea to make notes about a program to remind you of the purposes of various parts of the program. A statement of the form

`REM remark`

allows the programmer to enter a remark as a line of a program. The remark is displayed whenever the program is LISTed, but is overlooked when the program is executed.

## Examples

```
1. 10 REM COMPOUND INTEREST PROGRAM
 20 INPUT "AMOUNT DEPOSITED"; A
 30 INPUT "INTEREST RATE PER PERIOD"; I
 40 INPUT "NUMBER OF PERIODS"; N
 50 B = A*(1+I)^N: REM INTEREST FORMULA
 60 PRINT "THE BALANCE IS"; B
```

When this program is LISTed, it will appear exactly as above. When the program is run, the computer will not act on the REM statements, but will proceed to the next line of the program.

```
2. 10 REM *****
 20 REM *
 30 REM * THIS PROGRAM WRITTEN BY *
 40 REM *
 50 REM * JOHN SMITH *
 60 REM *
 70 REM *****
 80
 .
 .
```

## Comments

1. Whenever a REM statement is used as one of several statements on a single line, the REM statement must be last. Otherwise, the statements following it will be overlooked by the computer. For instance,

```
10 PRINT "ONE": REM COUNTING PROGRAM: PRINT "TWO"
RUN
ONE

READY.
```



# REM

Since the REM statement preceded the statement PRINT "TWO", this statement was not executed.

2. A GOTO or GOSUB statement may branch to a REM statement. If so, the computer will just execute the next line after the REM statement. However, this is not regarded as good programming practice. If the REM statement is later removed, so that the program will take up less space and execute faster, the GOTO or GOSUB will have no destination.

3. REM statements slow execution of the program slightly. Hence, it is not a good idea to include them in a loop that will be executed many times. Some programmers maintain two copies of a program: one with REM statements and one without. That way they have a well-documented program and also a version that runs quickly.

## Applications

1. Most long programs consist of several parts. Some programmers like to have space separating these parts on the printout of the program. This can be accomplished by preceding each part with a few lines consisting of just line numbers and the statement REM.

2. A subroutine is a part of a program that is called upon several times during execution of the program to perform some specific task. It is good programming practice to precede each subroutine with a REM statement identifying the task performed by the subroutine. This is helpful if you go back to the program after a considerable time period or if another person tries to understand the program.

3. When debugging a program, you might want to remove a line from the program and put it back later. Instead of deleting it and later retyping it, just INSERT the word REM after the line number. The line will be treated as a REM statement and will not be executed. Later, all you have to do is DELETE the word REM.

# RESTORE

The RESTORE statement is only used in conjunction with DATA and READ statements. DATA statements store a list of constants, and READ statements assign successive values from the list to variables. (See the discussions of the DATA and READ statements.) Suppose that the statement

```
RESTORE
```

is encountered during the execution of a program. Nothing happens until the next READ statement is encountered. That READ statement then acts as if it were the first READ statement in the program. It assigns values beginning with the first constant of the lowest numbered DATA statement.

As a model of how the RESTORE statement operates, imagine that at any time there is an arrow pointing at the next DATA constant to be accessed by a READ statement. Each time a READ statement assigns a value to a variable, the arrow moves to the next DATA constant. When the RESTORE statement is encountered, the arrow is set back to the first constant of the lowest numbered DATA statement.

## Examples

```
1. 10 DATA ONE, TWO, THREE
 20 READ A$, B$
 30 RESTORE
 40 READ C$
 50 PRINT A$, B$, C$
 RUN
 ONE TWO ONE
 READY.
```

## Applications

1. The RESTORE statement can be used to access a directory. The following program looks up phone numbers in a simplified telephone directory.

```
10 INPUT "NAME";N$
20 IF N$="*" THEN END
30 READ A$, B$
```

# RESTORE

```
40 IF A$ <> N$ GOTO 30
50 PRINT B$
60 RESTORE
70 GOTO 10
80 DATA AL,123-4567, TOM,987-6543
90 DATA JANE,202-765-4321, BILL,666-6666
RUN
NAME? TOM
987-6543
NAME? JANE
202-765-4321
NAME? *

READY,
```

If a name not in the directory is given in response to the question, an ?OUT OF DATA ERROR IN 30 message results. A friendlier way to handle a name not in the directory is to report such with a PRINT statement, rather than having the program die an unnatural death due to an error. To achieve a friendly response when a name is not found, add a *trailer value* to the directory, for instance

```
100 DATA ZZZZ,000-0000
```

Then add

```
35 IF A$="ZZZZ" THEN PRINT "NAME NOT FOUND"
:GOTO 60
```

2. The RESTORE statement can be used to display words vertically on the screen. Consider the following program:

```
10 DATA ONE, TWO, THREE, FOUR, FIVE, SIX
20 FOR I = 1 TO 5
30 FOR J = 1 TO 6
40 READ C$
50 D$=MID$(C$,I,1)
60 PRINT TAB(5*J) D$;
70 NEXT J
80 PRINT
90 RESTORE
100 NEXT I
```

# RESTORE

RUN

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| O | T | T | F | F | S |
| N | W | H | O | I | I |
| E | O | R | U | V | X |
|   |   | E | R | E |   |
|   |   | E |   |   |   |

READY.

# RIGHT\$

The ability to manipulate strings of characters is an important feature of computers. The RIGHT\$ function allows us to extract a portion of a string. If A\$ is a string variable or constant and N is a whole number between 0 and 255 inclusive, then

```
RIGHT$(A$,N)
```

is the string consisting of the rightmost N characters of A\$.

## Examples

```
1. PRINT RIGHT$("COMMODORE",2)
 RE
```

```
 READY,
 PRINT RIGHT$("MATTHEW WEBB",4)
 WEBB
```

```
 READY,
```

```
2. 10 T$="1234 RICHMOND STREET"
 20 PRINT RIGHT$(T$,6),RIGHT$(T$,1)
 RUN
 STREET T
```

```
 READY,
```

3. The following program isolates a person's last name from his full name.

```
10 INPUT "FULL NAME";N$
20 FOR I=1 TO LEN(N$)
30 A$=RIGHT$(N$,I)
40 IF ASC(A$)=32 THEN 70
50 NEXT I
60 A$=" "+N$:REM LAST NAME IS ENTIRE NAME
70 PRINT "YOUR LAST NAME IS"; A$; ", "
RUN
FULL NAME? JOHN PHILLIP SOUSA
YOUR LAST NAME IS SOUSA.
```

```
READY,
```

# RIGHT\$

## Comments

1. There are 256 different characters that can be used in a string. These characters and their ASCII/Commodore values are listed in Appendix A. All of these characters are counted when they appear in a string, even characters such as RETURN, DELeTe, and INSeRt. All spaces are counted, even leading and trailing spaces.

```
10 T$ = "VERY TRULY"+CHR$(13)+"YOURS"
20 PRINT RIGHT$(T$,10)
RUN
RULY
YOURS

READY,
```

(Note: CHR\$(13) is the character RETURN.)

2. Although the RIGHT\$ function is defined only for strings, it can be used indirectly to extract a specified number of digits from a number in standard form. For instance, if A is a whole number, then the N least significant digits of A are VAL(RIGHT\$(STR\$(A),N)).

```
PRINT VAL(RIGHT$(STR$(1875),2))
75

READY,
```

Note: When a number is PRINTed, a space is automatically inserted after the number. In all other situations, as in converting a number to a string using STR\$, no space follows the number.

3. The RIGHT\$ function creates a new string, but does not destroy the original string.

4. The N in RIGHT\$(A\$,N) may be a numeric constant, variable, or expression of either integer or floating-point precision. If N is not a whole number, RIGHT\$ uses the truncated value of N. If this value is not between 0 and 255 inclusive, the ?ILLEGAL QUANTITY ERROR message results. If N is zero, RIGHT\$ returns the empty string, "". If N is greater than or equal to the number of characters in A\$, then RIGHT\$ returns the complete value of A\$.

```
10 T$="NEIL ARMSTRONG"
20 PRINT RIGHT$(T$,6.8)
30 PRINT RIGHT$(T$,0)
```

# RIGHT\$

```
40 PRINT RIGHT$(T$,20)
50 PRINT RIGHT$(T$,256)
RUN
STRONG
```

```
NEIL ARMSTRONG
```

```
?ILLEGAL QUANTITY ERROR IN 50
READY,
```

## Applications

1. The RIGHT\$ function is used to manipulate strings, as in Example
3. It has wide uses in word processing.

# RND

The true capabilities of the computer are revealed as we explore the various applications of the RND function. This function is pivotal in simulation, whether it be the simulation of a game of chance or a complex business operation.

Consider the spinner pictured in Figure 1. Think of the circumference of the circle as having length one and every point on the circumference as being labeled with its clockwise distance from the top of the circle. Suppose that the spinner is well balanced and that the numbers can be read to 9 significant digits. Each time the pointer is flicked, it will give a number from 0 to .99999999. The spinner selects a number at *random* from 0 to .99999999. The function

RND(0)

acts as just such a spinner, returning as its value a random number between 0 and .99999999 inclusive.

Traditionally, random number tables were used to obtain a sequence of random numbers. The Commodore 64 has the capability of calling up many such fixed tables. Each table is identified by a negative number. If  $X$  is a negative number, then the function RND( $X$ ) returns the first number in the list identified by  $X$ . We say that  $X$  *seeds* the random number generator. Each use of RND( $X$ ) for a given negative value produces the same random number. After seeding the random number generator, the function RND( $Y$ ), where  $Y$  is any *positive* number, returns the next number from the list. (The exact value of  $Y$  is of no consequence.)

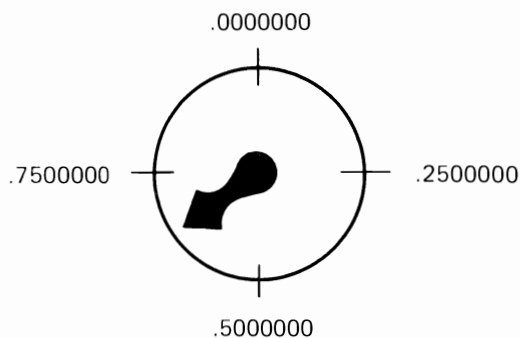


FIGURE 1



## Examples

```
1. PRINT RND(0);RND(0);RND(0)
 .113234409 .0351594687 .693597088

READY.
```

(Note: Results will differ.)

```
2. PRINT RND(-.87);RND(-.87);RND(-.87)
 .919215731 .919215731 .919215731

READY.
```

```
3. 10 PRINT RND(-.87);RND(1);RND(1)
 20 PRINT RND(-.87);RND(2);RND(3)
 RUN
 .919215731 .561426142 .356926548
 .919215731 .561426142 .356926548

READY.
```

The same starting random number was guaranteed for both program lines by using RND of the same negative number. The same list of random numbers was produced by each program line, since RND of a positive number was used to calculate the second and third random numbers based on the previous value of RND. These numbers are called *pseudorandom* because they appear to be random, and satisfy many tests of random numbers, but are in fact generated by a simple algorithm which gives the same result each time.

```
4. 10 A = RND(0): B = 5*RND(0)
 20 PRINT A;B;B+2
 RUN
 .349172318 3.76198346 5.76198346

READY.
```

(Note: Results will differ.)

5. The following program “tosses a coin” to decide how to answer a question.

```
10 INPUT "WHAT IS YOUR QUESTION";Q$
20 A = RND(0)
```

# RND

```
30 IF A<.5 THEN PRINT "YES": END
40 PRINT "NO":END
RUN
WHAT IS YOUR QUESTION? SHOULD I PLAY BRIDGE TONIGHT?
YES

READY.
```

Note: Line 30 could just as well have used  $A \leq .5$  for the test. Including the equals sign means we have added one more value, .5, to those that produce a YES answer. But  $RND(0)$  will come out to be exactly .5 less than once in a billion tries, on the average, which isn't worth fussing over.

## Comments

1. The X in  $RND(X)$  can be any numeric constant, variable, or expression.

```
A=.29: PRINT RND(-3*A)
 .923121981

READY.
```

2. Being able to generate the same list of random numbers each time a program is run can be very helpful when debugging. As shown in Example 3, this can be achieved by using  $RND$  for the first time with a fixed negative argument and then with a positive argument in subsequent uses. When debugging is complete, a changing list of random numbers can be obtained by simply replacing the negative argument of  $RND$  with zero.

3. Though  $RND$  produces values from 0 to .999999999, it is possible to use  $RND$  in conjunction with other functions to select a number at random from any collection of numbers. See Applications 1 and 2 below.

## Applications

1. It is possible to simulate the outcome of rolling a pair of dice. Each die will show a whole number from 1 to 6. If A is a number between 0 and .999999999, then  $6*A$  will be between 0 and 5.999999999. If we throw away the decimal portion of the number by using the function  $INT$  we will have a whole number from 0 to 5. Hence, the result of

$\text{INT}(6*\text{RND}(0))+1$

will be a whole number from 1 to 6, with each number just as likely to occur as any other number. This is the same outcome that results from rolling a well-balanced die.

The following program prints the outcomes from rolling a pair of dice 7 times.

```

10 FOR I = 1 TO 7
20 D1 = INT(6*RND(0))+1
30 D2 = INT(6*RND(0))+1
40 PRINT D1; D2; " ";
50 NEXT I
RUN
 6 4 6 4 4 5 6 6
 3 2 5 5 1 2
READY,

```

(Note: Results will differ.)

By letting the computer “throw” the dice many times while it keeps track of the results of each roll, we can let the computer estimate the odds of getting a given roll. If an experiment (rolling a pair of dice) has been conducted  $N$  times and a specific outcome (perhaps getting a total of 7 on the two dice) has occurred  $C$  times, then the approximate probability of the outcome occurring is  $C/N$  or 1 in  $N/C$  rolls. The larger  $N$  is, the closer  $C/N$  will be to the exact probability of the specific outcome occurring. The following program finds the approximate probability of getting each of the 11 possible totals from the roll of a pair of dice.

```

10 DIM C(12)
20 FOR I=1 TO 5000
30 X=INT(6*RND(0))+1
40 Y=INT(6*RND(0))+1
50 C(X+Y)=C(X+Y)+1 :REM COUNT UP THE
 OCCURRENCE OF THE TOTAL "X+Y"
60 NEXT I
70 PRINT
80 PRINT "TOTAL ON DICE";TAB(20);"ODDS"
90 FOR J=2 TO 12
100 PRINT
110 PRINT TAB(5);J;TAB(20);"1 IN";5000/C(J)
120 NEXT J
RUN

```

# RND

| TOTAL ON DICE | ODDS            |
|---------------|-----------------|
| 2             | 1 IN 36,4963542 |
| 3             | 1 IN 17,9211553 |
| 4             | 1 IN 11,6279192 |
| 5             | 1 IN 8,74125974 |
| 6             | 1 IN 7,51879731 |
| 7             | 1 IN 6,03136337 |
| 8             | 1 IN 7,12250704 |
| 9             | 1 IN 8,99280625 |
| 10            | 1 IN 11,9504117 |
| 11            | 1 IN 18,3150284 |
| 12            | 1 IN 35,7142993 |

READY,

(Note: Results will differ. This program takes approximately one and a half minutes to run.)

2. As a generalization of Application 1, the result of

$$M * \text{RND}(0) + N$$

is a number between  $N$  and  $N + M$ , including  $N$ , and, if  $M$  and  $N$  are whole numbers, the result of

$$\text{INT}(M * \text{RND}(0)) + N$$

is a whole number between  $N$  and  $N + M - 1$  inclusive. Here,  $M$  is positive, but  $N$  can be positive, negative, or zero.

The outcome of the spin of a roulette wheel can be simulated as the result of

$$\text{INT}(38 * \text{RND}(0)) - 1$$

where “-1” is interpreted as “00”.

# RND

3. Airlines simulate their operations in order to determine the best allocation of personnel and equipment. There are many unknowns, such as volume of business, failure of equipment, and weather conditions. Statisticians analyze these uncertainties to determine patterns and make predictions. Let's examine one pattern.

Suppose that the air conditioning system for each plane in a certain fleet of airplanes breaks down after an average of 100 hours of use. The time between successive breakdowns is unpredictable. It might be as short as 1 hour or as long as 500 hours. When simulating the total operation of the airline, statisticians use the RND function to come up with a plausible sequence of times between breakdowns. For reasons discussed in the next section, the best function to use is

$$-100*\text{LOG}(1-\text{RND}(0))$$

Consider the following program.

```
10 FOR I = 1 TO 10
20 PRINT -100*LOG(1-RND(0)),
30 NEXT I
RUN
 241.254128 80.7895434
 460.066902 73.1524248
 94.5083855 132.371183
192.137205 351.136778
 47.9144626 36.5253157

READY,
```

This program simulates breakdowns of the air conditioning system of a particular airplane to occur after 241.25 hours, then after another 80.79 hours, etc. This result, when used in conjunction with simulations of all of the other demands on the maintenance crew of the airline, enables the airline to determine how many people to allocate to maintenance. Such calculations are called *Monte Carlo* calculations because of their use of random numbers.

4. FOR STATISTICIANS: In Application 3, the times between successive breakdowns of the air conditioning system of an airplane was an exponential random variable with mean 100. The general procedure for obtaining a sample from an arbitrary continuous random variable is as follows:

Let  $F(x)$  be the cumulative density function for the random variable. Let  $G(y)$  be the inverse of  $F(x)$ . That is,  $G(y)$  is obtained by solving  $y=F(x)$  for  $x$  in terms of  $y$ . Then the number  $G(\text{RND}(0))$  will be a random number from the given distribution.

## RND

For instance, an exponential random variable with expected value  $M$  has the cumulative density function

$$F(x) = 1 - \text{EXP}(-x/M)$$

Solving  $y = F(x)$  for  $x$ , we obtain

$$x = -M * \text{LOG}(1-y)$$

or  $G(y) = -M * \text{LOG}(1-y)$

The result of running the program

```
10 FOR I = 1 TO 10
20 PRINT -M*LOG(1-RND(O))
30 NEXT I
```

will be a random sample of 10 observations from the random variable. (Note: The exponential distribution is valid for numerous observations, including lifetimes of electrical components, durations of phone calls, interarrival times, and time intervals between successive emissions of particles by radioactive material.)

After a BASIC program has been written, the computer will be in direct mode, and the program will be in memory. The command

```
RUN
```

will then execute the program.

A variation of the RUN command causes the execution of the program to begin at a designated line instead of at the lowest numbered line. The command

```
RUN n
```

causes the program to begin execution at line *n*.

## Examples

```
1. 10 A$ = "COMPUTER"
 20 PRINT A$
 RUN
 COMPUTER
```

```
READY.
```

```
2. 10 PRINT "TEN"
 20 PRINT "TWENTY"
 30 PRINT "THIRTY"
 RUN 20
 TWENTY
 THIRTY
```

```
READY.
```

## Comments

1. The RUN command is equivalent to the combination of a CLR and a GOTO command. Thus, before a program RUNs, all variables are cleared from memory, all DIMensioned arrays become undimensioned, and all information set with DEF FN statements, all file buffers and the meaning of all file reference numbers are removed from memory. However, RUN does not reset printer specifications, such as print mode and double width characters.

```
10 PRINT A$
A$ = "COMPUTER"
```

```
READY.
```

# RUN

```
RUN
```

```
READY.
```

Nothing was displayed, since the RUN command cleared all variables from memory before executing the program. The string variable A\$ took its unspecified form, the null string.

2. Since RUN removes all file buffers and the meaning of all file reference numbers, care must be taken that all open files are CLOSED before a RUN command is given. If files are not CLOSED and the RUN command is given, those on disk will be unreadable, and those on cassette will be incomplete.

3. The *n* in RUN *n* must be a numeric constant. It may not be a variable or expression. If for some reason *n* is a floating-point constant, RUN uses the truncated value of *n*. If the (truncated) value is not the number of a line in the program, the message ?UNDEF'D STATEMENT ERROR results.

4. Although the RUN command usually is used in command mode, it also can be used within a program. When executed, it will reRUN the program from the beginning, or the line number specified. As always, before the program is RUN, memory will be CLeaRed.

The following lines might be placed at the end of a game program.

```
970 PRINT "WOULD YOU LIKE TO PLAY AGAIN"
980 INPUT "(Y OR N)";A$
990 IF A$ = "Y" THEN RUN
1000 END
```

## Applications

1. The RUN command can be used to work with two different programs in memory at the same time. Suppose the first program has line numbers 10-90. Type the second program using line numbers from 110 on, and put the statement END in line 100. Then use the command RUN to execute the first program, and use the command RUN 110 to execute the second program.

2. The command RUN *n* is frequently used to test a part of a larger program without executing the entire program.



After writing a BASIC program, you can store a copy as a file on a cassette or disk with the **SAVE** command. You can then recall the program at a later time to execute or edit it. The command

```
SAVE "progrname"
```

records the current program onto cassette, while the command

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

writes the current program onto disk. In both cases, *progrname* is the program name under which the program is to be stored.

## Examples

1. The command

```
SAVE "ACCOUNTING"
```

copies the program currently in memory onto a cassette and gives the program the name **ACCOUNTING**.

2. The command

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

copies the current program onto a disk under the name **SCHEDULE**.

## Comments

1. After a program is **SAVEd**, the program remains in memory. Hence, you can continue to edit the program. If you change the program and then **SAVE** it on the same disk using the same name as before, the previous version will be erased. In order to retain two versions of the same program on disk, you must give the new version a name different from that of the previous version. With a cassette, you may use the same name for later versions of a program without causing earlier versions to be erased. If so, however, when later **LOADing** the program, it can be hard to **LOAD** the intended version.

2. The **SAVE** command may be used within a program. When executed, the program currently in memory, which is of course the one

# SAVE

currently RUNning, will be SAVEd onto a cassette or disk. Normal execution of the program will then continue with the next statement after the SAVE statement.

3. The quotation marks surrounding "*prognam*e" are required. The string constant created by these quotation marks and *prognam*e may be replaced by a string variable. This might be done within a program to allow the user to rename the program.

```
120 PRINT "NEW NAME UNDER WHICH TO SAVE"
130 INPUT "THIS PROGRAM";A$
140 A$="@0:"+A$
150 SAVE A$,8
```

4. The three characters @0: in SAVE "@0:*prognam*e",8 instruct the disk operating system to replace any old version of *prognam*e with the new version currently in memory. These three characters may be omitted the *first* time the program is SAVEd. However, if a program called *prognam*e already exists on disk and the SAVE command is given without the characters @0: included, the program currently in memory will not be SAVEd. Instead, a disk operating system error will occur with an accompanying blinking red light, but BASIC will not notify you of the problem.

5. It is a good idea to SAVE your program frequently during the development and editing of the program. That way, a recent copy is always available in case some mishap causes you to lose the program in memory. Some programmers make it a practice to execute a SAVE command after each addition or edit of 20 lines in a program.

The function SGN tells whether a given number is positive, zero, or negative. Specifically, the value of the function

$$\text{SGN}(X) \text{ is } \begin{cases} 1 & \text{if } X > 0 \\ 0 & \text{if } X = 0 \\ -1 & \text{if } X < 0 \end{cases}$$

## Examples

```
1. PRINT SGN(3,40); SGN(0); SGN(-35)
 1 0 -1
```

READY,

```
2. 10 A = 45E+12: B = -1*-1*-1
 20 PRINT SGN(A); SGN(B); SGN(1+B)
 RUN
 1 -1 0
```

READY,

## Applications

1. The SGN function is useful whenever we want to choose a course of action that depends on whether a certain number is below, equal to, or above another number. The following program tests a person's knowledge of the earliest all-electronic digital computer.

```
10 PRINT "IN WHAT YEAR WAS THE"
20 INPUT "ENIAC COMPUTER COMPLETED";A
30 ON 2+SGN(A-1946) GOTO 40,50,60
40 PRINT "NOT THAT LONG AGO!": GOTO 10
50 PRINT "CORRECT":END
60 PRINT "EARLIER THAN THAT!": GOTO 10
```

2. The SGN function can be used in conjunction with the INT function to round numbers. The following program rounds numbers to 3 decimal places.

```
10 INPUT A
20 B = ABS(A) * 1000 + .5
30 PRINT SGN(A) * INT(B) / 1000
```

# SGN

To round to  $r$  decimal places, replace 1000 with "1" followed by  $r$  zeros.

3. The statement

```
FOR I=A TO B STEP C*SGN(B-A)
```

where  $C$  is a positive number, can be used to guarantee that the loop is executed for the values between  $A$  and  $B$  in steps of  $C$ , regardless of whether  $A$  is less than  $B$  or vice versa. This can be a handy tool when the values for  $A$  and  $B$  will be requested from the user, and we don't wish to burden ourselves or the user with entering the values in a specific order.

```
10 INPUT A,B
20 FOR I=A TO B STEP .2*SGN(B-A)
30 PRINT I;
40 NEXT I
50 PRINT
```

RUN

? 2,3

2 2.2 2.4 2.6 2.8

READY.

RUN

? 3,2

3 2.8 2.6 2.4 2.2

READY.

Note: The final value in each set of answers did not appear due to rounding errors. See Comment 9 in FOR and NEXT. The following revision of line 20 can be used to solve this problem:

```
20 FOR I=A TO B+.5*(.2*SGN(B-A)) STEP .2*SGN(B-A)
```

SIN is the trigonometric function sine. For an acute angle in a right triangle, the sine of the angle is the ratio:

$$\frac{\text{length of the side opposite the angle}}{\text{length of hypotenuse}}$$

See Appendix M for the definition of the sine function for arbitrary angles and a discussion of radian measure. For any number X, the value of the function

SIN(X)

is the sine of the angle of X radians.

## Examples

```
1. PRINT SIN(1); SIN(-5.678); SIN(2E+8)
 ,841470985 ,568914483 -.671558955
```

READY.

```
2. 10 A% = 2: B = 7*A%: C = ,643501109
 20 PRINT SIN(A%); SIN(B); SIN(C)
 RUN
 ,909297427 ,990607356 ,6
```

READY.

## Comments

1. Although X can be any number, SIN(X) will always be between -1 and 1. Figure 1 contains the graph of  $y = \text{SIN}(X)$ .

2. The inverse of the sine function is the function arcsine. This function is not available directly as a BASIC function. However, it can be defined in terms of ATN and SQR, which are BASIC functions.

$$\arcsin(X) = \text{ATN}(X/\text{SQR}(1-X*X))$$

$\arcsin(X)$  is the angle between  $-\pi/2$  and  $\pi/2$  with sine X.

```
10 DEF FNARCSN(X) = ATN(X/SQR(1-X*X))
20 A = SIN(1): B = FNARCSN(A)
30 C = FNARCSN(.5): D = SIN(C)
```

# SIN

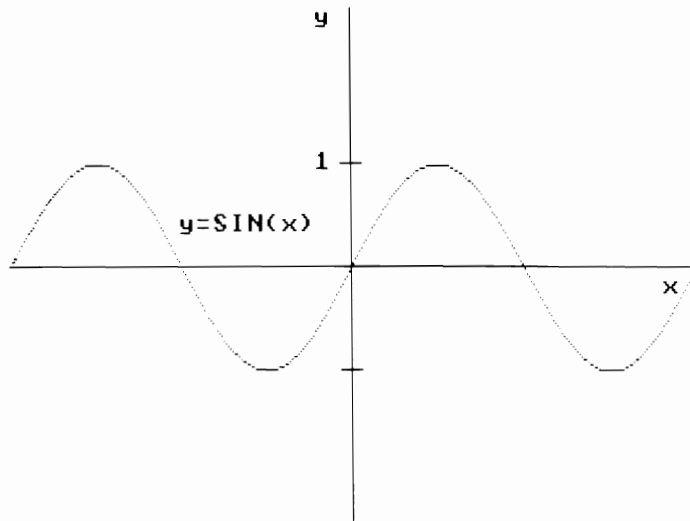


FIGURE 1

```
40 PRINT A; B; C; D
RUN
.841470985 1 .523598776 .5

READY.
```

In general, for any number  $X$  between  $-\pi/2$  and  $\pi/2$ ,  $\text{FNARCSN}(\text{SIN}(X))$  is  $X$ , and for any number  $X$  between  $-1$  and  $1$ ,  $\text{SIN}(\text{FNARCSN}(X))$  is  $X$ . Note: Because we cannot use BASIC keywords within the names of variables and functions that we create, we could not use SIN in naming the arcsine function.

## Applications

1. The sine function appears in many formulas of physics, mathematics, and engineering. For instance, if a projectile is fired at an initial velocity of  $V$  feet per second at an angle  $A$  radians with the ground, then (in the absence of air resistance) the projectile will be in flight for  $V \cdot \text{SIN}(A)/16$  seconds.

The SPC function is used in conjunction with the PRINT statement to insert spaces among the items being displayed. Since the PRINT statement has so many subtleties, considerable care is required to describe accurately the effects of the SPC function. We recommend that the discussion of PRINT be read before proceeding.

Suppose that we are displaying items on the screen. If N is a whole number from 0 to 255 and

SPC(N)

is placed in a PRINT statement, then N spaces will be displayed at that point. If the current line does not have N columns remaining, then those spaces which will not fit on the current line will be displayed at the beginning of the next line, and so on to successive lines as needed. If the SPC function is the last item in a PRINT statement, it will suppress the terminating carriage return and line feed that PRINT would normally perform.

## Examples

```
1. PRINT SPC(3); "LEFT"; SPC(5); "RIGHT"
 LEFT RIGHT
```

READY.

```
2. PRINT SPC(38); "DOWN" ,SPC(7); "UP"
 WN UP DO
```

READY.

The word "DOWN" is split between the first and second line, since it cannot fit on the first line after SPC(38). The comma after the word "DOWN" results in the next item, the seven spaces of SPC(7), starting in print zone 2. The word "UP" starts in column 17, immediately after these seven spaces.

3. In the following example, spaces are denoted by asterisks.

```
PRINT "LINE1";SPC(40);"LINE2"
LINE1*****
*****LINE2
```

READY.

# SPC

```
4. 10 PRINT "MERRY";SPC(1)
 20 PRINT "CHRISTMAS"
 RUN
 MERRY CHRISTMAS

 READY.
```

Notice that the first PRINT statement had the same result as if it had ended with a semicolon.

5. The following program uses SPC to right-justify a list of names:

```
10 FOR I = 1 TO 4
20 READ A$
30 PRINT SPC(10-LEN(A$));A$
40 NEXT I
50 DATA WASHINGTON, ADAMS
60 DATA JEFFERSON, MADISON
RUN
WASHINGTON
 ADAMS
 JEFFERSON
 MADISON

READY.
```

6. The following program centers a string in the middle of a line of width 40:

```
10 INPUT A$
20 PRINT SPC(20-LEN(A$)/2);A$
```

## Comments

1. SPC can be used with PRINT# in exactly the same way that it is used with PRINT. SPC *must* be used in conjunction with one of these two statements.

2. The N in SPC(N) may be a numeric constant, variable, or expression of either integer or floating-point precision. SPC uses the truncated value of N, and if this value is not between 0 and 255 inclusive, the ?ILLEGAL QUANTITY ERROR message results.

3. If the line on which items are currently being PRINTed already has characters displayed from an earlier PRINT, LIST, etc., SPC will



# SPC

not alter these characters as it spaces over the specified amount. Consider the following program.

```
10 PRINT "{SHIFT-CLR/HOME}";
20 PRINT TAB(18);"MIDDLE"
30 PRINT "{CLR/HOME}";
40 PRINT "BEGINNING";SPC(28);"END"
RUN
BEGINNING MIDDLE END
```

READY.

## Applications

1. The SPC function is used to display information in an orderly fashion. For instance, in the above examples, we used SPC to right-justify a list of names and to center a word on a line.

# SQR

SQR is the square root function. For any nonnegative number X, the value of

SQR(X)

is the nonnegative number whose square is X. The graph of SQR is shown in Figure 1.

## Examples

```
1. PRINT SQR(9); SQR(3.14); SQR(2E+30)
 3 1.77200452 1.41421356E+15
```

READY.

```
2. 10 A = .5: B = 8*A: C% = 256
 20 PRINT SQR(A);SQR(A+1);SQR(B);SQR(C%)
 RUN
 .707106781 1.22474487 2 16
```

READY.

## Comments

1. The X in SQR(X) may be a numeric constant, variable, or expres-

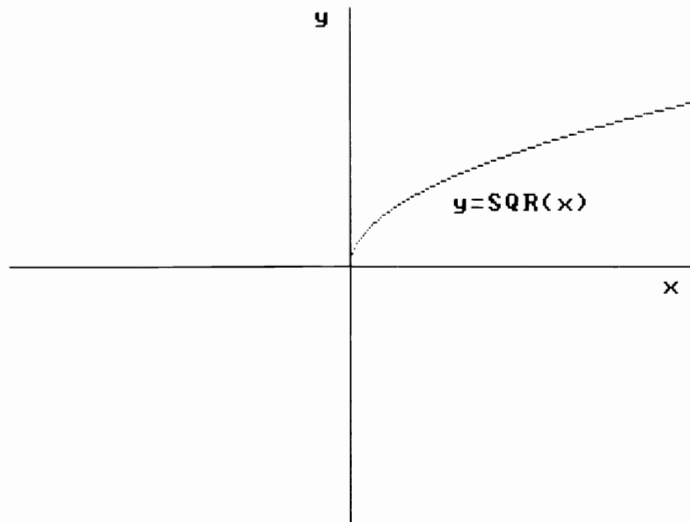


FIGURE 1

# SQR

sion of either integer or floating-point precision. If the value of X is less than zero, the ?ILLEGAL QUANTITY ERROR message will result.

2. SQR(X) is the same as  $X \uparrow .5$ . However, SQR(X) is calculated faster by the arithmetic unit.

## Applications

1. The SQR function is involved in many mathematical and statistical formulas. For instance, the length of the hypotenuse of a right triangle is the square root of the sum of the squares of the other two sides. In statistics, the standard deviation of a collection of data is the square root of the sum of the squares of the distances of each of the pieces of data from the mean, divided by the number of pieces of data.

# STATUS

When a program is reading data from a file, it is important for the program to be able to tell when the last item of data has been read. One way to deal with this need is to write a special piece of data, called the *trailer value*, as the last item in the file. The program can then check for the trailer value as each item of data is read. This approach has the disadvantages of requiring that a trailer value be determined that would never occur in normal data, and that this trailer value be remembered when writing later programs.

A second and less troublesome way of letting the program know that all of the data has been read from a file is provided by the STATUS function. If a data item has just been read from a cassette or disk file, then the value of the expression

STATUS AND 64

will be 64 if the data item was the last in the file, and 0 otherwise. In general, the value of the function

STATUS

is determined by whether or not one or more of certain conditions exist after an I/O (Input/Output) operation.

0. If the printer or disk drive does not acknowledge when told to PRINT a character, STATUS is given the value 1.
1. If the disk drive does not return a character when told to read, STATUS is given the value 2.
2. If, when told to read a file, the cassette returns fewer characters than are required to fill the input buffer, STATUS is given the value 4.
3. If, when told to read a file, the cassette player returns more characters than are required to fill the input buffer, STATUS is given the value 8.
4. If a bad piece of tape causes an unrecoverable read error, or a mismatch is found by the VERIFY statement, STATUS is given the value 16.
5. If an attempt to read data results in a value that the computer knows is incorrect, STATUS is given the value 32.
6. If the data item just read is the last in the file, STATUS is given the value 64.
7. If an end-of-tape marker is read, or the printer or disk drive is not connected or turned on, STATUS is given the value -128.

If more than one of the conditions above occur at the same time, the

# STATUS

value of STATUS will be the sum of the values assigned to the conditions which have occurred. To check for any one condition, use the statement

```
IF (STATUS AND V) = V THEN . . .
```

where V is the value which corresponds to the condition being checked.

## Examples

1. The following program reads character by character from the cassette file MEMBERS until the end of the file is reached.

```
10 OPEN 1,1,0,"MEMBERS"
20 GET#1, A$: PRINT A$
30 IF (STATUS AND 64) = 64 THEN END
40 GOTO 20
```

2. Suppose that a program in memory is checked against a program on disk with the VERIFY command, and the programs do not match.

```
VERIFY "BANNER",B

SEARCHING FOR BANNER
VERIFYING
?VERIFY ERROR
READY,
PRINT STATUS
80

READY,
```

The value 80 corresponds to the two condition values 16 and 64, corresponding to a mismatch in VERIFYing and reaching the end of the file.

## Comments

1. If no problems occur during an I/O operation, the value of STATUS will be 0.

2. The value of STATUS is the value of the byte at memory location 144. If the eight bits at location 144 are numbered from right to left as 0 to 7, then bit number n will be 1 if condition n above is true, and 0 otherwise.

# STATUS

3. The abbreviation `ST` may be used in place of `STATUS`. Although the word `STATUS` may not be used as any part of a variable name, the letters `ST` may be used as part of a variable name as long as they are not the first letters of the name. Thus `FAST` is a valid variable name.

## Applications

1. The `STATUS` function is used extensively when working with data files in order to determine when all data has been read from a file.

While running a program, you can press RUN/STOP to stop the execution of the program. You cannot predict in advance exactly where the program will stop. However, including the statement

**STOP**

as a line of the program guarantees that the program will stop execution at that line. The computer will display **BREAK IN  $n$**  where  $n$  is the number of the line containing the **STOP** statement. The computer will then be in direct mode. The command **CONT** entered while in direct mode causes the program to continue execution with the first statement following the **STOP** statement in line  $n$ .

## Examples

```
1. 10 FOR I = 1 TO 109 STEP 12
 20 PRINT "{SHIFT-CLR/HOME}"
 30 FOR J = 0 TO 11
 40 PRINT I+J,1000*(1.01)^(I+J)
 50 NEXT J
 60 STOP
 70 NEXT I
```

This program will print the balance after each month (for 120 months) when \$1000 is deposited at 12% interest compounded monthly. Upon running the program, you will see the balances for the first 12 months followed by the message **BREAK IN 60**. If you then type **CONT** and press the **RETURN** key, the next 12 months will be displayed. Without the **STOP** statement, the first 108 months would have flickered by too fast to be read, and only the last 12 months would be clearly displayed.

```
2. 10 PRINT "DAVID"
 20 STOP: PRINT "ALLAN"
 30 PRINT "DORRIS"
 RUN
 DAVID

 BREAK IN 20
 READY.
 CONT
 ALLAN
 DORRIS

 READY.
```

# STOP

Notice that the program CONTinued with the first statement after STOP, not the first line after STOP.

## Comments

1. After STOPping a program, there are further options for proceeding other than by using CONT. The command RUN will rerun the program from the beginning. The statements GOTO *m* and GOSUB *m* will continue execution of the program beginning with line number *m*.
2. After a program has been STOPped and the computer is in direct mode, you can display and change values of variables and make calculations. However, if you alter or add a line to the program, type the RETURN key on a line beginning with a number, or cause an error message to appear, then you cannot use CONT to resume running the program. You can, however, continue execution by using a GOTO or GOSUB statement.

```
10 A = 30
20 PRINT 20
30 PRINT A
40 STOP
50 PRINT 50
RUN
 20
 30
```

```
BREAK IN 40
READY.
A = 40
```

```
READY.
GOTO 30
 40
```

```
BREAK IN 40
READY.
CONT
 50
```

```
READY.
RUN
 20
 30
```



# STOP

```
BREAK IN 40
READY.
30 PRINT A+100
CONT
```

```
?CAN'T CONTINUE ERROR
READY.
```

3. The END statement is similar to the STOP statement. Both cause the program to stop execution and can be followed by CONT, GOTO, GOSUB, or RUN to resume execution. The only difference between STOP and END is that STOP causes a BREAK IN *n* message to be displayed.

## Applications

1. STOP is used when debugging a program. The programmer inserts the STOP statement at a crucial point in the program and then, while in direct mode, checks the values of certain variables to make sure that everything is in order. After he is convinced that the program is functioning properly, he deletes the STOP statement.

2. Sometimes programmers insert a STOP statement before a subroutine to guarantee that the subroutine will only be executed as the result of a GOSUB statement.

3. The STOP statement can be used to prevent scrolling until the user has had an opportunity to read the screen. When the user is ready, he just enters the CONT command. Example 1 provides an illustration of one way to use the STOP statement to chop a long list of data into manageable pieces.

# STR\$

Each number has a standard representation that is appropriate to its precision and magnitude. Some examples follow.

| <u>Number</u> | <u>Standard Representation</u> |
|---------------|--------------------------------|
| 2.50          | 2.5                            |
| 3E+2          | 300                            |
| .00000003     | 3E-08                          |
| 12345678      | 12345678                       |
| -56           | -56                            |

If N is a number, then

STR\$(N)

is the string consisting of the characters in the standard representation of N. If N is a negative number, the first character of STR\$(N) is a minus sign. Otherwise, the first character is a space. Also, STR\$(N) does not place a trailing space after the digits in N as is the case with PRINT N.

## Examples

```
1. PRINT STR$(-123);STR$(2.50);STR$(1E-8)
 -123 2.5 1E-08
```

READY.

```
2. 10 A = 23.45: B% = 17: C = 10987654321
 20 PRINT STR$(A); STR$(B%); STR$(C)
 RUN
 23.45 17 1.09876543E+10
```

READY.

In line 10, the floating-point variable C can retain only 9 digits of precision and so cannot be displayed as the given whole number.

3. Often, house addresses are used as strings but sorted by street number. The STR\$ function allows us to have it both ways.

```
10 INPUT "HOUSE NUMBER,STREET NAME";N,S$
20 PRINT "THE ADDRESS IS";STR$(N)+" "+S$
```

```

RUN
HOUSE NUMBER,STREET NAME? 1600,PENN AVE.
THE ADDRESS IS 1600 PENN AVE.

READY.

```

## Comments

1. The N in STR\$(N) may be any numeric constant, variable, or numeric expression.

```

PRINT STR$(2+3)
5

READY.

```

2. The VAL function undoes the STR\$ function in the sense that, for any number N, VAL(STR\$(N)) is equal to N.

```

10 N = -5*3
20 N$ = STR$(N)
30 PRINT N$;VAL(N$);VAL(N$)+5
RUN
-15-15 -10

READY.

```

Notice that N\$ has no trailing space, but VAL(N\$), which is the numeric constant -15, does.

3. Most numbers take up much more space in memory when stored as strings rather than as numeric constants.

## Applications

1. In order to make programs user-friendly, we often work with numbers as strings and use the functions STR\$ and VAL to go back and forth.

2. The STR\$ function is essential in programs doing symbolic manipulation of numbers. One example is a program that will express the sum of two fractions as a fraction.

# SYS

Problems exist that we would like the computer to solve, yet writing a BASIC program to do the job can prove extremely awkward, or, when written, can prove to be very slow to execute. This is especially true for programs that manipulate individual bits or bytes of memory, or programs that execute a set of instructions a large number of times. In these cases it is often useful to solve part of the problem using a machine language subroutine.

If a machine language subroutine has been placed in memory beginning at location A, the statement

**SYS A**

transfers control from BASIC to the machine language subroutine. Execution of any statements following SYS will occur when control is passed back to BASIC by the execution of an RTS (ReTurn from Subroutine) statement within the machine language subroutine.

## Examples

```
1. 10 GOSUB 100 :REM POKE MACHINE CODE TO
 PERFORM LEFT CIRCULAR BIT SHIFT
 20 INPUT "VALUE"; A
 30 POKE 49152,A
 40 SYS 49154
 50 PRINT PEEK(49152)
 60 END
 100 FOR I=49154 TO 49169
 110 READ D%: POKE I,D%
 120 NEXT
 130 DATA 14,0,192,46,1,192,78,0,192
 140 DATA 110,1,192,46,0,192,96
 150 RETURN
 RUN
 VALUE? 5
 10

 READY.
 RUN
 VALUE? 128
 1

 READY.
```

Lines 100 thru 150 make use of the POKE statement to place a ma-

# SYS

chine language subroutine into memory beginning at location 41954. Lines 20 and 30 enter a piece of data into memory, and then line 40 transfers control to the machine language subroutine which takes the data at location 49152 and rotates these 8 bits one place to the left, with the leftmost bit becoming the rightmost. The result, which was left at location 49152, is PRINTed in line 50.

2. The following program uses the predefined machine language routine PLOT to position the cursor at any point on the screen.

```
10 GOSUB 100 :REM POKE MACHINE CODE TO
 POSITION CURSOR AT ROW R, COLUMN C
20 PRINT "{SHIFT-CLR/HOME}"; :REM CLEAR SCREEN
30 PRINT "ROW AND COLUMN AT WHICH TO "
40 PRINT "LOCATE CURSOR AND PRINT "
50 INPUT "MESSAGE";R,C
60 POKE 49152,R: POKE 49153,C: SYS 49154
70 PRINT "X MARKS THE SPOT"
80 END
100 FOR I=49154 TO 49164
110 READ D%: POKE I,D%
120 NEXT
130 DATA 174,0,192,172,1,192,24
140 DATA 32,240,255,96
150 RETURN
RUN
ROW AND COLUMN AT WHICH TO
LOCATE CURSOR AND PRINT
MESSAGE? 5,15
```

X MARKS THE SPOT

READY.

The value of R may range from 0 to 23, and C from 0 to 39.

3. The following statement causes the computer to completely restart itself, as if the power had been turned off and then on again.

SYS 64738

## Comments

1. A machine language subroutine called by SYS may have any desired amount of numeric data passed to it. This is accomplished prior

# SYS

to the SYS call by using the POKE statement to place the data into those memory locations the machine language subroutine will be accessing. Similarly, any number of results may be returned to BASIC by having them placed in specific locations by the subroutine and then PEEKed by BASIC.

2. Unlike variables, arrays, and user-defined functions created by BASIC, machine language subroutines used by SYS are not erased by commands such as RUN, NEW, LOAD, and CLR. Once a machine language subroutine is POKEd into memory, the only way to change it is to rePOKE the appropriate memory locations, or, if the subroutine has been POKEd between locations 2048 and 40959, add sufficient lines to the BASIC program that it overwrites the subroutine.

3. The best locations in which to place machine language subroutines are those having addresses 49152 to 53247. These 4K RAM locations are not disturbed by BASIC, nor do they have any other system use.

4. Machine language subroutines called by SYS must end with the RTS (ReTurn from Subroutine) statement so that control of execution is passed back to BASIC.

5. Both the USR function and SYS statement allow BASIC programs to pass control to machine language subroutines. Since the SYS statement uses the beginning location of the machine language subroutine as its argument rather than requiring that the starting address be POKEd into a special location, the SYS statement is more versatile.

The appearance of output on the screen is often enhanced by placing the data in columns. The arrangement of output into columns is easily obtained by using the TAB function in conjunction with the PRINT statement.

A single line on the screen consists of 40 columns numbered 0 to 39, starting at the left edge of the screen. If N is a positive whole number between 0 and 39 inclusive, and A\$ is a string constant or variable, the statement

```
PRINT TAB(N); A$
```

will display the value of A\$ beginning at column N. A similar result holds for numeric variables or constants; however, we must take into account the leading and trailing spaces usually displayed with numbers. If the TAB function is the last item in a PRINT statement, it will suppress the terminating carriage return and line feed that PRINT would normally perform.

## Examples

- ```
1. 10 PRINT "01234567890123456789"  
20 PRINT "WAYNE";TAB(10);"PENNSYLVANIA"  
RUN  
01234567890123456789  
WAYNE      PENNSYLVANIA  
  
READY.
```
- ```
2. 10 FOR I = 1 TO 5
20 READ A$, B$, C$
30 PRINT A$; TAB(7); B$; TAB(28); C$
40 NEXT I
50 DATA YEAR, BEST PICTURE, DIRECTOR
60 DATA 1960, THE APARTMENT, WILDER
70 DATA 1961, WEST SIDE STORY, WISE
80 DATA 1962, LAWRENCE OF ARABIA, LEAN
90 DATA 1963, TOM JONES, RICHARDSON
RUN
YEAR BEST PICTURE DIRECTOR
1960 THE APARTMENT WILDER
1961 WEST SIDE STORY WISE
```

# TAB

```
1962 LAWRENCE OF ARABIA LEAN
1963 TOM JONES RICHARDSON
```

```
READY.
```

```
3. 10 PRINT "DO";TAB(5)
 20 PRINT "RE";TAB(10)
 30 PRINT "MI"
 RUN
 DO RE MI
```

```
READY.
```

Notice that the `PRINT` statements in lines 10 and 20 produced the same result as if they had ended with a semicolon.

## Comments

1. The `TAB` function respects all of the subtleties of the `PRINT` statement. For instance, if the `TAB` function is followed by a comma rather than a semicolon, then the next item will be displayed not at the `TAB` location, but at the beginning of the next print zone after the `TAB` location.

```
PRINT "RED";TAB(12);"GREEN"
RED GREEN
```

```
READY.
```

`GREEN` was placed at the beginning of print zone 3, not in column 12.

2. The `TAB` function will not cause the cursor to backspace. If the number `N` specifies a column that is to the left of the current cursor position, the next item will be placed beginning at the current cursor position, as if no `TAB` had occurred.

```
PRINT "PSYCHOLOGY";TAB(5);"PHILOSOPHY"
PSYCHOLOGYPHILOSOPHY
```

```
READY.
```

3. The `N` in `TAB(N)` may be a numeric constant, variable, or expression of either integer or floating-point precision. `TAB` uses the truncated value of `N`, and if this value is not between 0 and 255 inclusive, the `?ILLEGAL QUANTITY ERROR` message results.



# TAB

```
10 PRINT "012345678901234567890123456789"
20 PRINT TAB(5,8);"CENTENNIAL"
30 A = 3
40 PRINT TAB(2+2*A);"CHESAPEAKE"
50 PRINT TAB(256);"MICHENER"
RUN
012345678901234567890123456789
 CENTENNIAL
 CHESAPEAKE

?ILLEGAL QUANTITY ERROR IN 50
READY.
```

4. On the screen, values of N from 0 through 39 give column numbers for the line on which the cursor currently lies. Values of N from 40 to 79 correspond to columns 0 through 39 of the line after the line currently containing the cursor. Similarly, values of N from 80 to 119 refer to positions on the second line below the line currently containing the cursor, and so on. Once the cursor moves to a new line, future values of N in TAB(N) will refer to columns numbered from the beginning of this new line, starting with 0.

```
10 PRINT "01234567890123456789"
20 PRINT TAB(42);"A";TAB(6);"B"
RUN
01234567890123456789

 A B

READY.
```

At the start of line 20, the cursor was positioned at the beginning of the line below the list of numbers. Thus column 42 was column 2 of the second line below the list of numbers. The cursor moved to this new line and printed A. But then TAB(6) referred to column 6 of the line on which A was printed.

5. If the line to contain the displayed items already has some characters displayed in it, TAB will not alter these characters as it skips to the specified column. The following program uses the cursor up character to illustrate this feature.

```
10 PRINT "FIRST"
20 PRINT "{CURSOR UP}";TAB(7);"SECOND"
```

# TAB

```
RUN
FIRST SECOND
```

```
READY.
```

6. The TAB function only performs as expected when used to format output on the screen. When used with the printer or data files, TAB(N) has the same effect as SPC(N).

```
10 OPEN 1,4
20 PRINT#1, "CAT";TAB(5);"DOG"
RUN
```

```
READY.
```

The following appeared on the printer.

```
CAT DOG
```

Notice that there are 5 spaces between the words. If this line were PRINTed on the screen instead, there would be 2 spaces between the words.

7. TAB must be used in conjunction with one of the statements PRINT or PRINT#. As noted in Comment 6, unless these statements print to the screen, the results will be the same as using SPC.

## Applications

1. The TAB function is used to organize data into columns as in Example 2.

2. TAB can be used to draw rudimentary graphs of functions. The following program draws a simple graph of the equation  $Y = X^2 + 1$ . See Figure 1. Tilt your head to the right when viewing the graph. The lowest point on the graph has coordinates (0,1).

```
10 GOSUB 100 :REM DRAW AXES
20 FOR X = -6 TO 6
30 PRINT TAB(X^2 + 1);"*"
40 NEXT
50 END
100 PRINT "{SHIFT-CLR/HOME}"; :REM CLEAR SCREEN
110 FOR X=-6 TO 6
```



# TAN

TAN is the trigonometric function tangent. For an acute angle in a right triangle, the tangent of the angle is the ratio:

$$\frac{\text{length of the side opposite the angle}}{\text{length of the side adjacent to the angle}}$$

The definition of the tangent function for arbitrary angles and a discussion of radian measure are presented in Appendix M. For any number X, except as noted in Comment 1 below, the value of the function

TAN(X)

is the tangent of the angle of X radians.

## Examples

```
1. PRINT TAN(1); TAN(-5.678); TAN(2E+8)
 1.55740772 .691776234 .906347169
```

READY,

```
2. 10 A% = 2: B = 7*A%: C = .732815102
 20 PRINT TAN(A%); TAN(B); TAN(C)
 RUN
 -2.18503987 7.24460677 .9
```

READY,

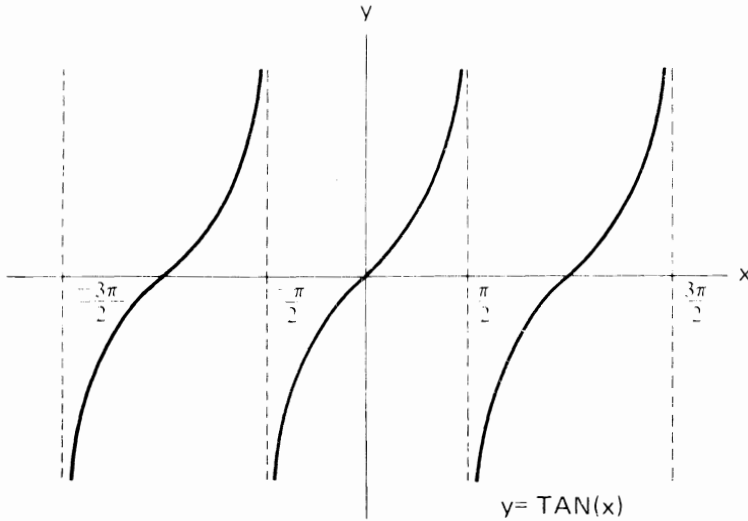
## Comments

1. The X in TAN(X) may be a numeric constant, variable, or expression. The tangent function is defined for all values of X except for  $X = \pi/2, -\pi/2, 3*\pi/2, -3*\pi/2$ , etc. Figure 1 shows the graph of  $y = \text{TAN}(X)$ .

2. The inverse of the tangent function is the function arctangent. This function is available directly in BASIC as the function ATN. ATN(X) is the angle between  $-\pi/2$  and  $\pi/2$  with tangent X.

```
10 A = TAN(1): B = ATN(A)
20 C = ATN(.5): D = TAN(C)
30 PRINT A; B; C; D
```

# TAN



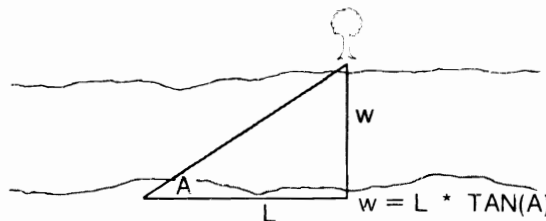
**FIGURE 1**

```
RUN
1.55740772 1 .463647609 .5
READY.
```

In general, for any number  $X$  between  $-\pi/2$  and  $\pi/2$ ,  $ATN(TAN(X))$  is  $X$ , and for any number  $X$ ,  $TAN(ATN(X))$  is  $X$ .

## Applications

1. Surveyors use the tangent function to measure the distance across a river. In Figure 2, the angle  $A$  is determined by using a transit and sighting a tree on the opposite side of the river. The width of the river is computed to be  $L * TAN(A)$ .



**FIGURE 2**

# TIME

The computer contains an internal timer or "jiffy clock" which records one tick every 1/60 of a second. The timer is set to zero when the computer is first turned on, and may be reset at any point by using the TIME\$ function.

At any time, the value of the function

TIME

is the number of ticks recorded on the internal timer. The abbreviation TI may be used for TIME.

## Examples

1. Suppose that the computer was turned on about 2 minutes ago and that the time was not reset.

```
PRINT TIME
7341
```

```
READY.
```

The computer has been operating for 7341 ticks of the timer. Dividing by 60 gives the number of seconds, which would be 7341/60 or approximately 122.

2. 

```
10 FOR I=1 TO 6
20 PRINT TI;
30 NEXT
RUN
7823 7826 7829 7831 7834 7837
READY.
```

Notice that each execution of the FOR . . . NEXT loop took 3 ticks on the timer.

3. In the following speed quiz, the response time is reported.

```
10 T = TI
20 PRINT
30 PRINT "WHAT IS THE SQUARE ROOT OF 5"
40 INPUT "TIMES THE SQUARE ROOT OF 20";A
50 PRINT
60 IF A <> 10 THEN PRINT "WRONG":GOTO 20
70 PRINT "CORRECT. YOU ANSWERED THE "
```

```

80 PRINT "QUESTION IN";(TI-T)/60;"SECOND
S."
RUN
WHAT IS THE SQUARE ROOT OF 5
TIMES THE SQUARE ROOT OF 20? 8

WRONG

WHAT IS THE SQUARE ROOT OF 5
TIMES THE SQUARE ROOT OF 20? 10

CORRECT, YOU ANSWERED THE
QUESTION IN 19.75 SECONDS.

READY.
```

## Comments

1. The value of TIME is a floating-point whole number between 0 and 5183999.
2. If the internal timer is reset using the TIME\$ function, then the value of TIME is reset to 60 times the number of seconds represented by the new value of TIME\$.

```

10 TIME$ = "000012"
20 PRINT TIME
30 TIME$ = "000300"
40 PRINT TIME
RUN
 720
10800

READY.
```

The second time assigned to TIME\$ is 3 minutes or 180 seconds past midnight. 60 times 180 gives 10800.

3. Although the function TIME\$ is related to TIME, it is important to remember that the value of TIME\$ is a string, and the value of TIME is a number.
4. Contrary to the normal rules for naming variables, the letters TI or TIME may be used as part of a variable name as long as they are not the first letters of the name. Thus LATIN and DAYTIME are valid variable names.

## TIME

5. The timer's operation is suspended during tape I/O. Thus, after loading or saving a program using the cassette, the value given by TIME will not include the ticks required for the tape I/O.

### **Applications**

1. TIME can be used, as in Example 3, to time user responses.
2. TIME can be used to determine the running time for a section of a program.



The computer has an internal timer or "jiffy clock" which records one tick every 1/60 of a second. TIME\$ can be used as a statement to set the clock or as a function to read the clock.

The value of the string function

TIME\$

is a string of 6 digits giving the hours, minutes, and seconds corresponding to the number of ticks that have been recorded by the timer. The form of this string is "hhmmss" where the digits hh range from 00 to 23 and give the hours, mm range from 00 to 59 and give the minutes, and ss range from 00 to 59 and give the seconds.

If T\$ is a string of six digits stating the time in the form "hhmmss", then the statement

TIME\$ = T\$

sets the clock to the stated time. The abbreviation TI\$ may be used for TIME\$.

## Examples

```
1. PRINT TIME$
 001212

 READY.
```

(Note: Results will differ, depending on how long the computer has been turned on.)

```
2. PRINT TI$: TI$="000000": PRINT TI$
 010203
 000000

 READY.
```

3. The following program turns the computer into a stopwatch. The statement GET A\$ assigns A\$ a value from a pressed key. Hence, A\$ has the value "" until a key is pressed.

```
10 PRINT "{SHIFT-CLR/HOME}"
20 PRINT "PRESS ANY KEY TO START WATCH"
30 GET A$: IF A$="" GOTO 30
```

# TIMES

```
40 PRINT "{SHIFT-CLR/HOME}"
50 PRINT "PRESS ANY KEY TO STOP"
60 TIME$="000000"
70 PRINT "{CLR/HOME}ELAPSED TIME IS: ";
80 PRINT TIME$
90 GET A$:IF A$="" GOTO 70
RUN
PRESS ANY KEY TO START WATCH
(key pressed, screen cleared)
ELAPSED TIME IS: 000210
PRESS ANY KEY TO STOP
```

The time stated in the last sentence will change until a key is pressed.

## Comments

1. The string T\$ must be a 6-character string of digits. The first and second digits give the hour and should be between 00 and 23. Midnight is 0 o'clock and 11 pm is 23 o'clock. The third and fourth digits give the minutes and should be between 00 and 59. The fifth and sixth digits give the seconds and also should be between 00 and 59. Some possibilities and their corresponding times are:

| <u>T\$</u> | <u>Corresponding Time</u>           |
|------------|-------------------------------------|
| 032151     | 21 minutes and 51 seconds past 3 am |
| 140500     | 2:05 pm                             |
| 000000     | midnight                            |
| 235959     | 1 second before midnight            |
| 120100     | 1 minute past noon                  |

2. When the computer is first turned on, the value of TIME\$ is automatically set to 000000. At any later time, the value of TIME\$ will be the time elapsed since the computer was turned on, unless the clock is reset by the user, or the statement SYS 64738 is executed (which causes the computer to do a "cold start", as if the computer were turned off and on again).

3. The clock is temporarily turned off, but not reset, during reading or writing of the tape cassette. Thus after tape I/O the clock will not show the correct elapsed time since power up or reset.

4. Contrary to the normal rules for naming variables, the characters TI\$ or TIME\$ may be used as the final part of a string variable name

## **TIME\$**

as long as they are preceded by other letters. Thus LATI\$ and DAY-TIME\$ are valid variable names.

### **Applications**

1. TIME\$ can be used to determine the running time of a program by setting the time to zero in the first line of the program and having the time displayed as the last line of the program.
2. The computer can be turned into an alarm clock via TIME\$.

# USR

Problems exist that we would like the computer to solve, yet writing a BASIC program to do the job can prove extremely awkward, or, when written, can prove to be very slow to execute. This is especially true for programs that manipulate individual bits or bytes of memory, or programs that execute a set of instructions a large number of times. In these cases it is often useful to solve part of the problem using a machine language subroutine.

If a machine language subroutine has been placed in memory beginning at location A, the statements

```
POKE 785,A-256*INT(A/256)
POKE 786,INT(A/256)
```

record the location of this machine language subroutine for later calls of the USR function. If N is a numeric constant, variable, or expression, the value of the function

```
USR(N)
```

is determined by the machine language subroutine whose beginning location has been recorded most recently at locations 785 and 786. The single numeric argument, N, is allowed for passing data to the machine language subroutine.

## Examples

```
1. 10 GOSUB 100 :REM POKE MACHINE CODE TO
 ADD 4 TO BASE 2 EXPONENT OF NUMBER
20 POKE 785,49152-256*INT(49152/256)
30 POKE 786 INT(49152/256)
40 A=8: B=12.7
50 PRINT USR(-6);USR(A);USR(B/5)
60 END
100 FOR I=49152 TO 49163
110 READ D%: POKE I,D%
120 NEXT
130 DATA 173,97,0,24,109,11
140 DATA 192,141,97,0,96,4
150 RETURN
RUN
-96 128 40.64

READY.
```

# USR

Lines 100 thru 150 make use of the `POKE` statement to place a machine language subroutine into memory beginning at location 49152. Lines 20 and 30 record this starting location, and then in line 50, `USR` is used just like any other built-in or user-defined function.

## Comments

1. A machine language subroutine called by `USR` may have only one piece of numeric data passed to it directly. However, it is always possible, before a use of the `USR` function, to `POKE` additional data into memory which later can be accessed by the machine language subroutine.
2. Unlike user-defined functions established by the `DEF FN` statement, `USR` functions are not erased by commands such as `RUN`, `NEW`, `LOAD`, and `CLR`. To associate a new machine language subroutine with `USR`, the starting address for the new subroutine must be `POKE`d into locations 785 and 786.
3. When `BASIC` executes `USR(N)`, the value of `N` is first computed, if necessary, by `BASIC`. The value of `N` is then placed in the floating-point accumulator which begins at location 97, and control passes to the machine language subroutine, which can use the value in this accumulator as data. The value returned by the `USR` function is the number which resides in this same floating-point accumulator when control is passed back to `BASIC` by the `RTS` machine code statement.
4. The best locations in which to place machine language subroutines are those having addresses from 49152 to 53247. These 4K RAM locations are not disturbed by `BASIC`, nor do they have any other system use.
5. Both the `USR` function and `SYS` statement allow `BASIC` programs to pass control to machine language subroutines. The `SYS` statement uses the beginning location of the machine language subroutine as its argument, and is therefore more versatile.

# VAL

If the leading characters of the string A\$ correspond to a number, then

```
VAL(A$)
```

will be the number represented by these characters. Otherwise, VAL(A\$) will be zero.

## Examples

```
1. PRINT VAL("123"); VAL("8.5 PERCENT")
 123 8.5
```

```
READY.
```

```
2. PRINT VAL("$3.45"); VAL(",23"); VAL("9,876")
 0 .23 9
```

```
READY.
```

```
3. 10 A$ = "-67.00"; B$ = "TWO"; C$ = "2E+3"
 20 PRINT VAL(A$); VAL(B$); VAL(C$)
 RUN
 -67 0 2000
```

```
READY.
```

4. The following program provides a way to interact with the computer while dealing with numeric input.

```
10 INPUT "NUMBER (ENTER E TO EXIT)"; N$
20 IF N$="E" THEN END
30 N = VAL(N$)
40 PRINT "THE SQUARE OF "; N$; " IS"; N*N
50 GOTO 10
RUN
NUMBER (ENTER E TO EXIT)? 15
THE SQUARE OF 15 IS 225
NUMBER (ENTER E TO EXIT)? E
```

```
READY.
```

Had we used a numeric variable in line 10, we would not have had such a convenient way to exit the program.

## Comments

1. The VAL function undoes the STR\$ function in the sense that for any number N, VAL(STR\$(N)) is equal to N.
2. The VAL function ignores all spaces.

```
PRINT VAL(" 23");VAL(" 12 98")
23 1298
```

```
READY,
```

## Applications

1. The VAL function can make programs user-friendly. Numeric data is input as a string and then converted back to a number after being altered as necessary. (See Example 3 in the discussion of MID\$.)

# VERIFY

After a program has been **SAVED** on a cassette or disk, it is a good practice to confirm that the **SAVE** was properly carried out. At other times, we may wonder if the program currently in memory is the same as another program previously **SAVED**. Both of these checks are easily done with the command **VERIFY**.

If *progrname* is the name of a program that resides on a cassette or disk, then the statement

```
VERIFY "progrname",D
```

compares the program currently in memory with the program *progrname* residing on either a cassette (if D is 1) or a disk (if D is 8). If the two programs match, the computer displays the message **OK** and proceeds to the next statement. However, if the programs do not match, the message **?VERIFY ERROR** is produced.

## Examples

1. `LOAD "PROG1",1`

```
PRESS PLAY ON TAPE
OK
```

```
SEARCHING FOR PROG1
FOUND PROG1
LOADING
READY,
VERIFY "PROG2",1
```

```
SEARCHING FOR PROG2
FOUND PROG2
VERIFYING
?VERIFY ERROR
READY,
```

Here **PROG1** was **LOADed** from a cassette, and then checked against **PROG2** which came later on the tape. The **?VERIFY ERROR** indicates that the two programs differ.

2. `SAVE "SPRITEMAKER",8`

```
SAVING SPRITEMAKER
READY,
VERIFY "SPRITEMAKER",8
```



# VERIFY

```
SEARCHING FOR SPRITEMAKER
VERIFYING
OK
```

```
READY.
```

After SAVEing the program SPRITEMAKER on disk, we VERIFYied that the SAVE command executed correctly.

## Comments

1. If the number D in VERIFY "*progname*",D is omitted, the program is assumed to be on cassette. Using values of 0, 2, or 3 for D results in the ?ILLEGAL DEVICE NUMBER ERROR. Values of D greater than 3, except those which correspond to a disk drive (usually just 8), will lock up the computer and should be avoided.
2. When working with programs on cassette, both *progname* and D may be omitted. The statement VERIFY alone will compare the next program on tape with the program currently in memory.
3. On the disk, the VERIFY statement can be used within a program.

```
900 SAVE "INDEXER",8
910 VERIFY "INDEXER",8
```

4. The name *progname* may either be given in quotation marks or as the value of a string variable. The following lines might be used within a program that is undergoing continuous modification, where each version must be SAVEd and VERIFYied.

```
900 INPUT "VERSION NUMBER";N$
910 A$="PROJECTIONS"+",V"+N$
920 SAVE A$,8
930 VERIFY A$,8
```

5. VERIFY can be used repeatedly to find and read past each program residing on a cassette. Then the tape will be positioned after the last program and ready to SAVE a new program without overwriting an old one.

# WAIT

The Commodore 64 is capable of communicating with other computers and devices by using the built-in RS-232 interface. In programs which make use of this interface, it may be necessary to have the computer suspend program execution until data has been received from the external device or computer. External events, like the receiving of data through the RS-232 interface, as well as data being entered at the keyboard, cause the values stored in certain memory locations to change. The WAIT statement is used to cause the execution of a BASIC program to be suspended until the occurrence of some external event alters the value in a certain memory location in a specific way.

The values stored in most memory locations consist of a byte (corresponding to an integer from 0 to 255). A byte is made up of eight bits, zeros or ones, which give the binary representation of the value stored in the memory location. (See Appendix B for a discussion of binary representations.) The rightmost bit in a byte is referred to as bit 0, the next bit as bit 1, . . . , and the leftmost bit as bit 7. For instance, the byte corresponding to the number 34 has ones as its first and fifth bits and zeros elsewhere.

If L is a whole number from 0 to 65535, and N is a whole number from 0 to 255, then the statement

WAIT L,N

causes the execution of the program to be suspended until the byte at location L has a 1 in one or more of the same positions that the binary representation of N has a 1. This is the same as saying that the program WAITs until the value of the expression PEEK(L) AND N is not zero. (See Appendix L for a discussion of the logical operator AND.)

For example, bit 2 of memory location 653 is changed to 1 whenever the ConTRoL key is pressed. Since the binary representation of 4 has a 1 in bit 2 and 0s everywhere else, the statement WAIT 653,4 causes program execution to halt until the ConTRoL key is pressed.

If L is a whole number from 0 to 65535, and N is a whole number from 0 to 255, then the statement

WAIT L,N,255

causes the execution of the program to be suspended until the byte at location L has a 0 in one or more of the same positions that the binary representation of N has a 1. For example, the statement WAIT 653,4,255 will suspend program execution until the ConTRoL key is not being pressed.

Now suppose that M is any whole number from 0 to 255. The statement

# WAIT

WAIT L,N,M

causes the execution of the program to be suspended until the byte at location L has a 0 at one or more of the positions for which the binary representations of both N and M have 1s, or has a 1 at one or more of the positions for which the binary representation of N has a 1 and M has a 0. This is the same as saying that the program WAITs until the value of the expression (PEEK(L) AND N) xor M) is not zero. (Note: “xor” is not a BASIC operator, but is discussed in Appendix L for completeness.)

## Examples

1. Memory location 198 records the number of characters that are currently held in the keyboard buffer for processing. Since the binary representation of 8 has a 1 in bit 3, bit 3 of location 198 will not become 1 until 8 characters are pending. The program line

```
100 WAIT 198,8
```

suspends program execution until 8 keys are typed.

2. Bit 5 in memory location 1 records whether the cassette motor is on or off. If bit 5 is 0, the motor is on. To check if bit 5 is 0, we must put a 1 in bit 5 of the binary representations of both N and M, giving them both the value 32. Thus the program line

```
100 WAIT 1,32,32
```

suspends program execution until the cassette motor is turned on by pressing play, rewind, or fast forward on the cassette.

## Comments

1. WAIT statements can easily cause infinite loops. The key RUN/STOP alone will not break program execution if a WAIT is active. However, by holding down the RUN/STOP key and then pressing the RESTORE key (perhaps several times), the computer can be caused to execute a *warm start*. The current program will remain in memory after the warm start, but various parameters, such as screen background and border color which you may have altered with the appropriate POKE, will be reset to their power-up values.

2. The L, M, and N in WAIT L,M,N may be any numeric constants, variables, or expressions of either integer or floating-point precisions.

# WAIT

WAIT uses the truncated values of L, M, and N. If the resulting value of L is not in the range 0 to 65535 or the value of M or N is not in the range 0 to 255, the message ?ILLEGAL QUANTITY ERROR results.

## Applications

1. The WAIT statement is used in communications software to monitor the status of communications registers and buffers and to pause until certain patterns appear. The GET statement is also used in communications, but only to wait until something besides the empty string, usually character data, appears in a memory location.

## Part II: SPRITES



# SPRITE Design (High-Resolution)

All objects displayed on the screen are made up of small dots. In high-resolution mode, the screen can display 320 dots from left to right and 200 dots from top to bottom. Each standard character, like the letter A, is formed within an 8 by 8 rectangle of dots by turning some dots on in the foreground color and leaving the remainder off in the background color. For example, Figure 1 shows a blowup of the upper left-hand corner of the screen where the letters A and B have been typed in columns 0 and 1 of row 1. A grid is superimposed around the dots to clearly show the 8 by 8 rectangles of dots used to form each letter. The data telling which dots are to be turned on and which not is permanently recorded in Character ROM (Read Only Memory).

In one respect, SPRITES can be thought of as giant characters. Each SPRITE is formed within a 24 wide by 21 high rectangle of dots (see Figure 2 below). The programmer decides which of these 504 dots are to be turned on in the SPRITE color, and which are to remain off in the background color. This information is coded, as discussed below, into 63 values, each an integer from 0 to 255, called the SPRITE data. The SPRITE data is stored in RAM (Random Access Memory), where it is used by the Commodore's Video Interface Chip to display the SPRITE at any point of our choosing on (or off) the screen. (See Moving SPRITES.) In this respect, SPRITES are more versatile than characters which only can be placed in one of the predefined 25 x 40 character positions on the screen.

Up to 8 SPRITES, numbered 0 to 7, can be displayed simultaneously by the Video Interface Chip, and additional SPRITE data can be stored in RAM, ready to give a new form to any of these 8 SPRITES. Complex video arcade games can thus be designed, with SPRITES exploding or changing shape in a form of animation.

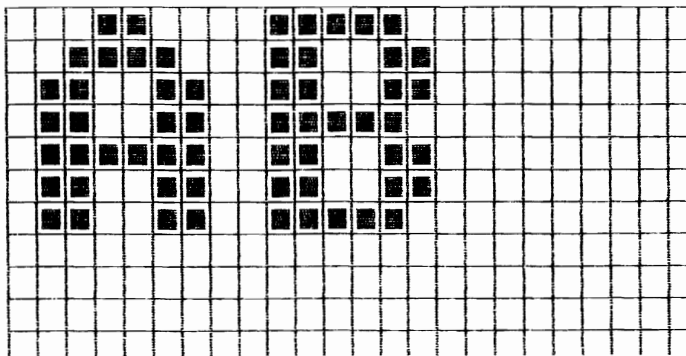


FIGURE 1

# SPRITE Design (High-Resolution)

Let  $d_0, d_1, d_2, \dots, d_{63}$ , be the whole numbers between 0 and 255 which code the design of a **SPRITE**. Let  $B$  be a whole number between 0 and 255 which indicates where the **SPRITE** data is to be stored in memory. And let  $S$  be the number, 0 to 7, of the **SPRITE** which this data is to describe. Then the statements

```
FOR I = 0 TO 62
 READ D
 POKE B*64+I, D
NEXT I
POKE 2040+S, B
DATA d0, d1, d2, . . . , d63
```

place the **SPRITE** data in memory and record the location of this data in the appropriate **SPRITE pointer** (memory location  $2040+S$ .)

In high-resolution mode, a single color is chosen for each **SPRITE**. The **SPRITE** color can be any of the 16 colors described in Appendix C. All dots turned on in a **SPRITE** appear in the chosen **SPRITE** color. All dots not turned on are *transparent*, allowing the background color, background characters, and other **SPRITES** to show through.

Let  $C$  be a whole number from 0 to 15 corresponding to a color chosen from Appendix C. If  $S$  is the number, 0 to 7, of the **SPRITE** which is to be given this color, then the statement

```
POKE 53287+S, C
```

assigns the color  $C$  to **SPRITE**  $S$ .

## Examples

1. The simplest **SPRITE** to design is one in which every dot is turned on. In this case, the **SPRITE** data consists of the number 255 repeated 63 times. (Example 2 below will make it clear where the number 255 comes from.) The following program creates **SPRITE** 0 with all its dots turned on in green and displays the **SPRITE** in the center of the screen. Press any key to return to direct mode and cause the **SPRITE** to disappear.

```
10 PRINT "{SHIFT-CLR/HOME}"
20 FOR I = 0 TO 62 :REM CREATE RECTANGULAR SPRITE
30 POKE 200*64+I, 255
40 NEXT I
50 POKE 2040, 200
60 POKE 53287, 5 :REM GREEN SPRITE
```



# SPRITE Design (High-Resolution)

```
200 POKE 53269, 1 :REM ENABLE SPRITE
210 POKE 53271, 1 :REM EXPAND SPRITE VERTICALLY
220 POKE 53277, 1 :REM EXPAND SPRITE HORIZONTALLY
300 POKE 53248, 170 :REM POSITION SPRITE
310 POKE 53249, 140
320 POKE 53264, 0
330 GET A$: IF A$="" THEN 300
400 POKE 53269, 0 :REM DISABLE SPRITE
```

Lines 20 thru 50 place the SPRITE data in memory and set the SPRITE pointer for SPRITE 0 to this data. Line 60 chooses green for the SPRITE color. Lines 210 and 220 double the height and width of the SPRITE. (See Expanding SPRITES). Lines 300 thru 320 position the SPRITE. (See Moving SPRITES. You may wish to replace lines 300 thru 330 with Example 1 or 2 from that section.) Lines 200 and 400 enable and disable the SPRITE. (See Enabling SPRITES).

2. Figure 2 shows a 24 wide by 21 high grid on which a SPRITE has been sketched by darkening those dots which should be turned on.

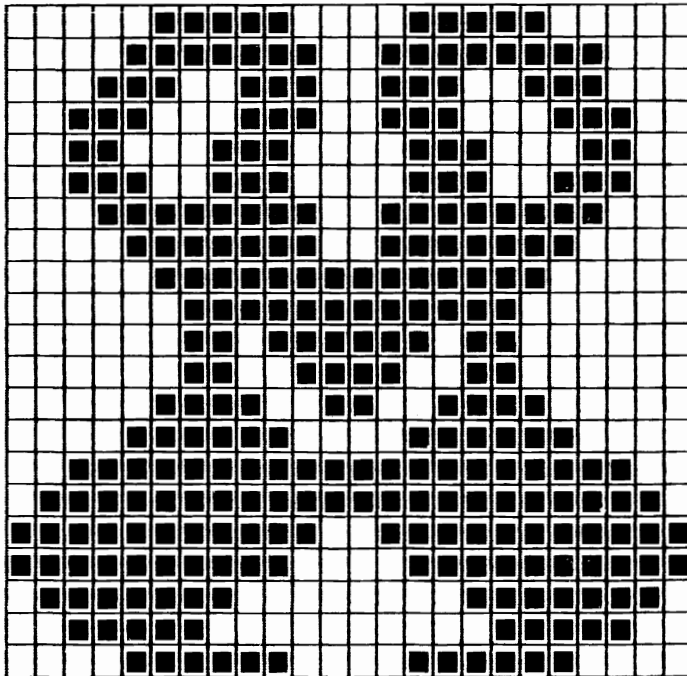


FIGURE 2

## SPRITE Design (High-Resolution)

This is the first step in designing a SPRITE. Now this sketch must be translated into SPRITE data. This is done by dividing each row of dots in Figure 2 into 3 groups of 8 dots. Each group of 8 dots is now thought of as the 8 bits of a binary number. Each dot that is shaded becomes a 1, and each dot not shaded a 0. Thus the first row of dots in Figure 2 corresponds to the three binary numbers 00000111, 11000011, and 11100000. Appendix J gives the decimal (base 10) value for each of the 256 possible 8-bit binary numbers. These 3 particular binary numbers translate to 7, 195, and 224. Proceeding in this manner, we get 3 numbers for each of the 21 rows in our SPRITE sketch. These 63 numbers, taken in order from left to right, row by row, give us our SPRITE data. (See Appendix G for a program that allows you to design your SPRITE on the screen and calculates the SPRITE data.)

A good block of memory in which to store this data is block 200. If this data is to be used to describe SPRITE 0, then the following lines would be used within the program.

```
10 FOR I = 0 TO 62
20 READ D
30 POKE 200*64+I, D
40 NEXT I
50 POKE 2040+0,200
60 DATA 7,195,224,15,231,240,28,231,56
70 DATA 56,231,28,49,195,140,57,195,156
80 DATA 31,231,248,15,231,240,7,255,224
90 DATA 3,255,192,3,126,192,3,60,192
100 DATA 7,153,224,31,195,248,63,255,252
110 DATA 127,255,254,255,231,255
120 DATA 255,195,255,127,0,254,62,0,124
130 DATA 15,195,240
```

If this SPRITE is to be colored light blue, color 14, the following statement must be added to the program:

```
140 POKE 53287+0, 14
```

The expression 53287+0 is illustrative and could have been written simply as 53287.

To actually see this SPRITE, we must also position and enable it. For example, add lines 200 thru 400 of Example 1 to this program, perhaps replacing lines 300-330 with the lines from Example 1 or 2 in Moving SPRITES, and then enter RUN.

# SPRITE Design (High-Resolution)

## Comments

1. Since SPRITE data always requires 63 memory locations, and 64 memory locations is a convenient number for the computer, the memory location given by the value of B is the beginning of a block of 64 memory locations. Block 0 starts at memory location 0, block 1 at memory location 64, block 200 at memory location 12800, and so forth. Some values of B give blocks of memory for which the computer has other uses (see Appendix H). These should be avoided. The best choices of B are those from 200 to 255. The blocks of memory corresponding to these values of B lie in the portion of memory reserved for BASIC programs and variables. A large BASIC program possibly could extend up into this memory, but a value of 200 for B leaves room for a BASIC program requiring over 8000 bytes without any overlap of memory.
2. In order that your SPRITE sketch have the same proportions that the SPRITE will have on the screen, the boundary of your 24 by 21 grid should be approximately a square.
3. One block of SPRITE data may be used for as many SPRITES as desired. To use the data in block B for several SPRITES, just execute the statement `POKE 2040+S,B` for each of the desired SPRITES. For instance, to create SPRITES 2 and 3 using the data in block 200, execute the statements `POKE 2042,200` and `POKE 2043,200`.
4. The color of SPRITE S may be changed at any time by POKEing a new value from Appendix C into location `53287+S`. For example, to switch the color of SPRITE 5 to orange, execute the statement `POKE 53292, 8`.
5. The shape of a SPRITE may be altered, producing animation effects. One method of altering a SPRITE is to POKE new data into some or all of the 63 locations where the SPRITE data is stored. A second method, and generally more effective, is to set up two or more blocks of data corresponding to each of the shapes which a SPRITE is to take, like different frames in a motion picture. By switching the SPRITE data pointer from one block to another, the shape of a SPRITE can be changed instantly, producing animation. The following example sets up SPRITE data in blocks 200 and 201, and uses these blocks to animate SPRITE 0.

```
10 PRINT "{SHIFT-CLR/HOME}"
20 B1 = 200*64
```

## SPRITE Design (High-Resolution)

```
30 B2 = 201*64
40 FOR I= 0 TO 20: POKE B1+I,255: NEXT I
50 FOR I=21 TO 41: POKE B1+I,0: NEXT I
60 FOR I=42 TO 62: POKE B1+I,255: NEXT I
70 FOR I= 0 TO 20: POKE B2+I,0:NEXT I
80 FOR I=21 TO 41: POKE B2+I,255: NEXT I
90 FOR I=42 TO 62: POKE B2+I,0: NEXT I
100 POKE 53287, 14 :REM LIGHT BLUE SPRITE
200 POKE 53269, 1 :REM ENABLE SPRITE
210 POKE 53271, 1 :REM EXPAND SPRITE VERTICALLY
220 POKE 53277, 1 :REM EXPAND SPRITE HORIZONTALLY
300 POKE 53248, 100 :REM POSITION SPRITE
310 POKE 53249, 100
320 POKE 53264, 0
400 POKE 2040, 200
410 FOR I=1 TO 100: NEXT I
420 POKE 2040, 201
430 FOR I=1 TO 100: NEXT I
440 GOTO 400
```

Lines 40 thru 90 place the SPRITE data in blocks 200 and 201, line 100 chooses light blue for the color of SPRITE 0, line 200 enables SPRITE 0 (see Enabling SPRITEs), and lines 300 thru 320 position SPRITE 0 on the screen. (See Moving SPRITEs. You may wish to replace lines 300 thru 320 with some of the examples given there.) Lines 400 thru 440 alternate the data used to form SPRITE 0 between blocks 200 and 201, with a small delay between switches.

# SPRITE Design and Color (Multi-Color)

All objects displayed on the screen are made up of small dots. In multi-color mode, the screen can display 160 dots from left to right and 200 dots from top to bottom. Dots in multi-color mode are twice as wide as those in high-resolution mode. But dots in multi-color mode can be one of 4 colors: transparent (background), SPRITE, multi-color 1, or multi-color 2, while dots in high-resolution mode can be just one of 2 colors: transparent or SPRITE. Each multi-color SPRITE is formed within a 12 wide by 21 high rectangle of dots. For each of these 252 dots, the programmer chooses one of the 4 colors mentioned above. This information is coded, as discussed below, into 63 bytes (numbers from 0 to 255) called the SPRITE data. The SPRITE data is stored in RAM (Random Access Memory), where it is used by the Commodore's Video Interface Chip to display the SPRITE at any point of our choosing on the screen.

Let  $d_0, d_1, d_2, \dots, d_{63}$ , be the whole numbers between 0 and 255 which code the shape of the multi-color SPRITE. Let  $B$  be a whole number between 0 and 255 which indicates where the SPRITE data is to be stored in memory. And let  $S$  be the numbers, 0 to 7, of the SPRITE which this data is to describe. Then the statements

```
FOR I = 0 TO 62
READ D
POKE B*64 + I, D
NEXT I
POKE 2040 + S, B
DATA d0, d1, d2, . . . , d63
```

place the SPRITE data in memory and record the location of this data in the appropriate SPRITE *pointer*. (Note that these statements are identical to those used in the case of a high-resolution SPRITE.)

In multi-color mode, two colors, called multi-color 1 and multi-color 2, are chosen from the 16 colors listed in Appendix C. These same two colors will be used by all SPRITES which are displayed in multi-color mode. In addition, a third color, called the SPRITE color, is chosen separately for each SPRITE from among the 16 available colors.

Let  $M_1, M_2$ , and  $SC$  be whole numbers from 0 to 15 corresponding to colors chosen from Appendix C. If  $S$  is one of the numbers 0 to 7, then the statements

```
POKE 53285, M1
POKE 53286, M2
```

# SPRITE Design and Color (Multi-Color)

assign M1 as multi-color 1 and M2 as multi-color 2 for all SPRITES, and the statement

```
POKE 53287+S, SC
```

assigns SC as the SPRITE color for SPRITE number S.

The final step in designing a multi-color SPRITE is to inform the computer that the SPRITE data is to be interpreted as multi-color data, rather than high-resolution data. The statement

```
POKE 53276, PEEK(53276) OR 21S
```

causes the SPRITE data for SPRITE number S to be interpreted as multi-color data.

## Examples

1. Figure 1 shows a 12 wide by 21 high grid on which a SPRITE has been sketched using 3 colors plus unshaded (transparent) dots. This is the first step in designing a multi-color SPRITE.

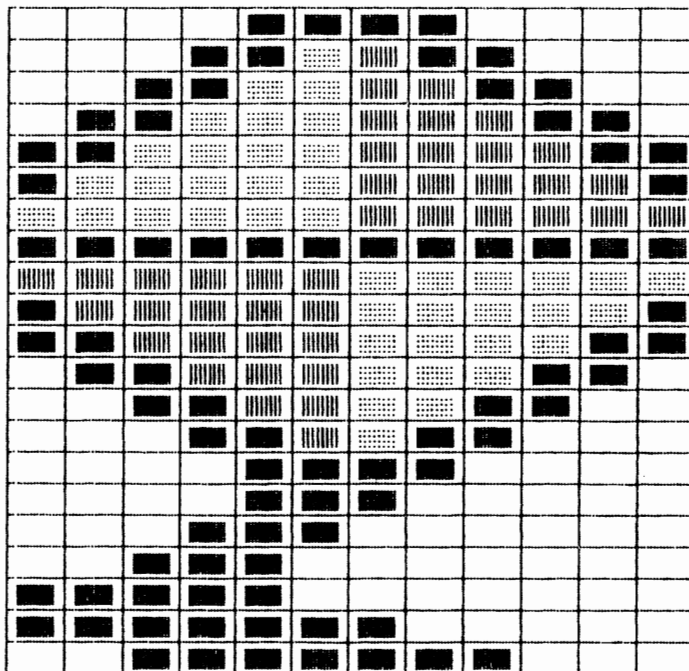





FIGURE 1

# SPRITE Design and Color (Multi-Color)

Once a sketch has been drawn, the SPRITE data can be generated through the following steps.

First we must decide which shading will have multi-color 1, which will have multi-color 2, and which will have the SPRITE color. Assume we make the following choices:

-  is multi-color 1,
-  is multi-color 2, and
-  is SPRITE color.

Our choice is conveyed to the computer by a 2-bit code:

- 00 is transparent,
- 01 is multi-color 1,
- 10 is SPRITE color, and
- 11 is multi-color 2.

Figure 2 shows a second 12 by 21 grid in which the shadings of Figure 1 have been replaced by 2-bit codes according to the choices made above:

 = 00,  = 01,  = 10, and  = 11

The final step in generating the SPRITE data is to create 8-bit binary numbers from the 2-bit codes in Figure 2 and convert these 8-bit binary numbers into normal base 10 numbers. Three 8-bit binary numbers can be formed from each row of the SPRITE sketch by grouping four 2-bit codes together. For example, the first row of Figure 2 gives the three 8-bit binary numbers 00000000, 10101010, and 00000000. Appendix J gives the decimal (base 10) value for each of the 256 possible 8-bit binary numbers. These 3 particular binary numbers translate to 0, 170, and 0. Proceeding in this manner, we get 3 numbers for each of the 21 rows in our SPRITE sketch. These 63 numbers, taken in order from left to right, row by row, give us our SPRITE data.

A good block of memory in which to store this data is block 200. If this data is to be used to describe SPRITE 0, then the following lines would be used within our program.

```
10 PRINT "{SHIFT-CLR/HOME}"
20 FOR I = 0 TO 62
30 READ D
40 POKE 200*64+I, D
50 NEXT I
60 POKE 2040+0,200
70 DATA 0,170,0,2,182,128,10,245,160
```

# SPRITE Design and Color (Multi-Color)

|    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|
| 00 | 00 | 00 | 00 | 10 | 10 | 10 | 10 | 00 | 00 | 00 | 00 |
| 00 | 00 | 00 | 10 | 10 | 11 | 01 | 10 | 10 | 00 | 00 | 00 |
| 00 | 00 | 10 | 10 | 11 | 11 | 01 | 01 | 10 | 10 | 00 | 00 |
| 00 | 10 | 10 | 11 | 11 | 11 | 01 | 01 | 01 | 10 | 10 | 00 |
| 10 | 10 | 11 | 11 | 11 | 11 | 01 | 01 | 01 | 01 | 10 | 10 |
| 10 | 11 | 11 | 11 | 11 | 11 | 01 | 01 | 01 | 01 | 01 | 10 |
| 11 | 11 | 11 | 11 | 11 | 11 | 01 | 01 | 01 | 01 | 01 | 01 |
| 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| 01 | 01 | 01 | 01 | 01 | 01 | 11 | 11 | 11 | 11 | 11 | 11 |
| 10 | 01 | 01 | 01 | 01 | 01 | 11 | 11 | 11 | 11 | 11 | 10 |
| 10 | 10 | 01 | 01 | 01 | 01 | 11 | 11 | 11 | 11 | 10 | 10 |
| 00 | 10 | 10 | 01 | 01 | 01 | 11 | 11 | 11 | 10 | 10 | 00 |
| 00 | 00 | 10 | 10 | 01 | 01 | 11 | 11 | 10 | 10 | 00 | 00 |
| 00 | 00 | 00 | 10 | 10 | 01 | 11 | 10 | 10 | 00 | 00 | 00 |
| 00 | 00 | 00 | 00 | 10 | 10 | 10 | 10 | 00 | 00 | 00 | 00 |
| 00 | 00 | 00 | 00 | 10 | 10 | 10 | 00 | 00 | 00 | 00 | 00 |
| 00 | 00 | 00 | 10 | 10 | 10 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00 | 00 | 10 | 10 | 10 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 10 | 10 | 10 | 10 | 10 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 10 | 10 | 10 | 10 | 10 | 10 | 10 | 00 | 00 | 00 | 00 | 00 |
| 00 | 00 | 10 | 10 | 10 | 10 | 10 | 10 | 00 | 00 | 00 | 00 |

**FIGURE 2**

```

80 DATA 43,245,104,175,245,90,191,245,86
90 DATA 255,245,85,170,170,170,85,95,255
100 DATA 149,95,254,165,95,250,41,95,232
110 DATA 10,95,160,2,158,128,0,170,0
120 DATA 0,168,0,2,160,0,10,128,0
130 DATA 170,128,0,170,168,0,10,170,128

```

We have not yet actually chosen multi-color 1, multi-color 2, or the SPRITE color. If it is decided to use red for multi-color 1, yellow for multi-color 2, and orange for the SPRITE color of SPRITE number 0, then the following lines would be added to our program.

```

140 POKE 53285, 2 : REM RED IS COLOR 2
150 POKE 53286, 7 : REM YELLOW IS COLOR 7
160 POKE 53287+0, 8 : REM ORANGE IS COLOR 8

```

The expression 53287+0 is illustrative and could have been written simply as 53287 for SPRITE 0.

To see this SPRITE we must still position it and enable it. For ex-



# SPRITE Design and Color (Multi-Color)

ample, add the following lines and run the program. Press any key to return to direct mode and cause the **SPRITE** to disappear.

```
200 POKE 53269, 1 :REM ENABLE SPRITE
210 POKE 53271, 1 :REM EXPAND SPRITE VERTICALLY
220 POKE 53277, 1 :REM EXPAND SPRITE HORIZONTALLY
300 POKE 53248, 100 :REM POSITION SPRITE
310 POKE 53249, 100
320 POKE 53264, 0
330 GET A$: IF A$="" THEN 300
400 POKE 53269, 0 :REM DISABLE SPRITE
```

See **Moving SPRITES** and **Enabling SPRITES** for more details on these statements. You may wish to replace lines 300 thru 330 with Example 1 or 2 given in **Moving SPRITES**.

## Comments

1. Since **SPRITE** data always requires 63 memory locations, and 64 memory locations is a convenient number for the computer, the memory location given by the value of **B** is the beginning of a block of 64 memory locations. Block 0 starts at memory location 0, block 1 at memory location 64, block 200 at memory location 12800, and so forth. Some values of **B** give blocks of memory for which the computer has other uses (see Appendix H). These should be avoided. The best choices of **B** are those from 200 to 255. The blocks of memory corresponding to these values of **B** lie in the portion of memory reserved for **BASIC** programs and variables. A large **BASIC** program possibly could extend up into this memory, but a value of 200 for **B** leaves room for a **BASIC** program requiring over 8000 bytes without any overlap of memory.
2. In order that your **SPRITE** sketch have the same proportions that the **SPRITE** will have on the screen, the boundary of your 12 by 21 grid should be approximately a square.
3. Multi-colors 1 and 2 can be changed at any time. Memory location 53285 stores the value of multi-color 1, and 53286 the value of multi-color 2. When either of these locations is **POKE**d with a color value from 0 to 15, the new color will immediately take effect in all **SPRITES**.
4. The **SPRITE** color associated with **SPRITE** number **S** may be changed at any time by **POKE**ing a new value from Appendix C into location 53287 + **S**.

## SPRITE Design and Color (Multi-Color)

5. One block of SPRITE data may be used for as many SPRITEs as desired. In fact, the same block of data can be used as multi-color data for one SPRITE and as high-resolution data for another.
6. The shape of a SPRITE may be altered, producing animation effects. One method of altering a SPRITE is to POKE new data into some or all of the 63 locations where the SPRITE data is stored. A second method, and generally more effective, is to set up two or more blocks of data corresponding to each of the shapes which a SPRITE is to take, like different frames in a motion picture. By switching the SPRITE data pointer at location  $2040 + S$  from one block to another, the shape of a SPRITE can be instantly changed, producing animation. See Comment 5 of SPRITE Design (High-Resolution) for a program illustrating this idea.
7. Multi-color mode for each SPRITE may be turned on or off at any time. The statement `POKE 53276, PEEK(53276) AND (255 - 2 * S)` cancels multi-color mode for SPRITE S, and displays SPRITE S in high-resolution mode instead.

## Enabling SPRITEs

Even though all steps in designing a **SPRITE** have been executed and the **SPRITE** has been positioned to be visible on the screen, the **SPRITE** will not appear on the screen until it is turned on or *enabled*. The statement

```
POKE 53269, PEEK(53269) OR 2↑S
```

enables **SPRITE** number **S**. **SPRITEs** may be enabled or disabled as desired. The statement

```
POKE 53269, PEEK(53269) AND (255-2↑S)
```

disables **SPRITE** number **S**.

### Examples

1. The following program line enables **SPRITE** 0.

```
200 POKE 53269, PEEK(53269) OR 1
```

2. The following program line disables **SPRITE** 4.

```
200 POKE 53269, PEEK(53269) AND 239
```

### Comments

1. The single statement `POKE 53269,255` enables all 8 **SPRITEs**, while the statement `POKE 53269,0` disables all 8 **SPRITEs**.

2. A single statement may be used to enable and disable any number of **SPRITEs**. If **S1**, **S2**, . . . , are the **SPRITEs** to be enabled, and if **N** equals  $2↑S1 + 2↑S2 + \dots$ , then the statement `POKE 53269,N` will enable **SPRITEs** **S1**, **S2**, . . . , and disable all other **SPRITEs**. For instance, since  $2↑0 + 2↑2 + 2↑5 = 37$ , the statement `POKE 53269,37` will enable **SPRITEs** 0, 2, and 5 and disable **SPRITEs** 1, 3, 4, 6, and 7.

3. Enabled **SPRITEs** cannot be **CLear**ed from the screen by pressing **SHIFT-CLR/HOME**. **SPRITEs** must be either disabled or positioned off the screen if they are not to be displayed.

4. Even when a **SPRITE** is enabled, it may not appear on the screen. This can be caused by not assigning the **SPRITE** a color (it then ap-

## Enabling SPRITEs

pears in the background color, sometimes causing a ghost image), or by not positioning the **SPRITE** so that at least part of it falls in the viewable part of the expanded screen (see **Moving SPRITEs**).

5. When the computer is powered up, memory location 53269 assumes the value 0. Thus the *default* condition calls for all **SPRITEs** to be disabled. This default condition can be regained at any time by pressing **RUN/STOP-RESTORE**.

## Expanding SPRITEs

Whether designed and displayed in high-resolution or multi-color mode, a **SPRITE** normally occupies a rectangular region of the screen the size of 3 rows of 3 regular characters each. It is not possible to give more data to increase the size of a **SPRITE**, but it is possible to cause each dot making up the **SPRITE** to be doubled in size in either the horizontal or vertical directions or both, thereby doubling the size of the entire **SPRITE** in either or both directions.

Each **SPRITE** can be independently expanded and unexpanded. The statement

```
POKE 53277,PEEK(53277) OR 2↑S
```

doubles the width of **SPRITE** number **S**, while the statement

```
POKE 53271,PEEK(53271) OR 2↑S
```

doubles the height. To return to the normal width of **SPRITE** number **S**, use the statement

```
POKE 53277,PEEK(53277) AND (255-2↑S)
```

while to return to normal height, use the statement

```
POKE 53271,PEEK(53271) AND (255-2↑S)
```

### Examples

1. The statement

```
POKE 53277,PEEK(53277) OR 1
```

doubles the width of **SPRITE** 0.

2. The statements

```
POKE 53277,PEEK(53277) OR 8
POKE 53271,PEEK(53271) OR 8
```

double the width and height of **SPRITE** 3.

# Expanding SPRITEs

## 3. The statements

```
POKE 53277,PEEK(53277) AND 223
POKE 53271,PEEK(53271) AND 223
```

return SPRITE 5 to normal width and height.

## Comments

1. The single statement `POKE 53271,255` expands all 8 SPRITEs vertically, while the statement `POKE 53271,0` returns all 8 SPRITEs to normal height. Similarly, the single statement `POKE 53277,255` expands all 8 SPRITEs horizontally, while the statement `POKE 53277,0` returns all SPRITEs to normal width.

2. A single statement may be used to expand any number of SPRITEs vertically. If  $S_1, S_2, \dots$ , are the SPRITEs to be expanded vertically, with all other SPRITEs to be of normal height, and if  $N$  equals  $2 \uparrow S_1 + 2 \uparrow S_2 + \dots$ , then the statement `POKE 53271,N` will expand SPRITEs  $S_1, S_2, \dots$ , vertically and leave all other SPRITEs with normal height. For instance, since  $2 \uparrow 1 + 2 \uparrow 3 + 2 \uparrow 4 = 26$ , the statement `POKE 53271,26` expands SPRITEs 1, 3, and 4 vertically and leaves SPRITEs 0, 2, 5, 6, and 7 with normal height.

3. A single statement may be used to expand any number of SPRITEs horizontally. If  $S_1, S_2, \dots$ , are the SPRITEs to be expanded horizontally, with all other SPRITEs to be of normal width, and if  $N$  equals  $2 \uparrow S_1 + 2 \uparrow S_2 + \dots$ , then the statement `POKE 53277,N` will expand SPRITEs  $S_1, S_2, \dots$ , horizontally and leave all other SPRITEs with normal width. For instance, since  $2 \uparrow 0 + 2 \uparrow 3 + 2 \uparrow 5 = 41$ , the statement `POKE 53277,41` expands SPRITEs 0, 3, and 5 horizontally and leaves SPRITEs 1, 2, 4, 6, and 7 with normal width.

4. When the computer is powered up, memory locations 53271 and 53277 assume the value 0. Thus the *default* condition calls for all SPRITEs to be of normal height and width. This default state can be regained at any other time by pressing `RUN/STOP-RESTORE`.

## Moving SPRITEs

All objects displayed on the screen are made up of small dots. The screen has 320 dot positions from left to right and 200 dot positions from top to bottom on the screen. A SPRITE is positioned on the screen by giving the dot where the upper left-hand corner of the SPRITE is to be placed. This location can be any dot on the screen, as well as “imaginary” dots off the edge of the screen. Having dot positions off the screen allows a SPRITE to be placed so that only a portion appears on the screen. Positions for the upper left-hand corner of a SPRITE may be changed whenever desired, and, if changed by only a few dots at a time, will produce nearly smooth movement of the SPRITE about the screen.

To visualize how a SPRITE is positioned on the screen, think of the screen as a window which permits viewing of a portion of a larger, “expanded” screen. (See Figure 1.) This expanded screen has 512 dots from left to right and 256 dots from top to bottom. The horizontal position of a dot on the expanded screen is referred to as the dot’s X position. X positions are numbered from 0 on the left of the screen to 511 on the right. The vertical position of a dot on the expanded screen is referred to as the dot’s Y position. Y positions are numbered from 0 at the top of the screen to 255 at the bottom. Our window to this expanded screen sees dots whose X position is between 24 and 343 and whose Y position is between 50 and 249.

A SPRITE may be placed with its upper left-hand corner at any dot position in the expanded screen, with all, some, or none of the SPRITE showing in the normal screen’s window. In general, if S, X, and Y are numeric constants or variables, with S being a whole number from 0 to 7, X a whole number from 0 to 511, and Y a whole number from 0 to 255, then the statements

```
POKE 53248 + 2*S, X-256*INT(X/256)
POKE 53249 + 2*S, Y
POKE 53264, PEEK(53264) AND (255-2↑S) OR (2↑S*INT(X/256))
```

place SPRITE number S with its upper left-hand corner X dots to the right and Y dots down from the upper left-hand corner of the expanded screen.

### Examples

In the following program segments, which illustrate moving SPRITES, the design and enabling of the SPRITES is not included.

# Moving SPRITEs

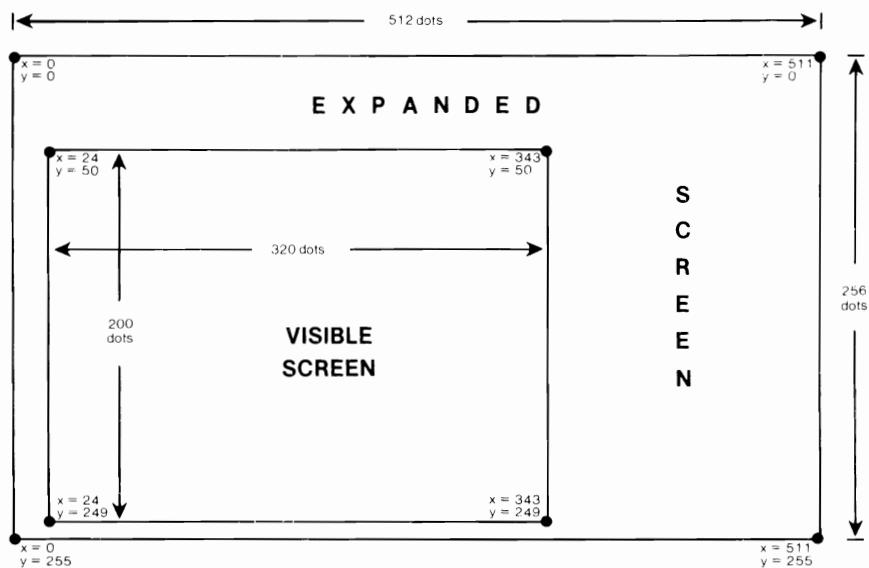


FIGURE 1

1. The following program lines will move SPRITE 0 diagonally across the screen.

```
300 FOR I=0 TO 255
310 X=2*I: Y=I
320 POKE 53248,X-256*INT(X/256)
330 POKE 53249,Y
340 POKE 53264,PEEK(53264) AND 254 OR INT(X/256)
350 NEXT I
```

2. The following program lines allow the user of the program to choose where to position SPRITE 0 on the screen.

```
300 INPUT "X POSITION ON EXPANDED SCREEN":X
310 IF X<0 OR X>511 THEN 380
320 INPUT "Y POSITION ON EXPANDED SCREEN":Y
330 IF Y<0 OR Y>255 THEN 380
340 POKE 53248,X-256*INT(X/256)
350 POKE 53249,Y
360 POKE 53264,PEEK(53264) AND 254 OR INT(X/256)
370 PRINT "{SHIFT-CLR/HOME}":GOTO 300
380 PRINT "VALUE OUT OF RANGE":END
```



# Moving SPRITEs

3. The following program lines move SPRITEs 2 and 4 in opposite directions across the top and bottom of the visible screen.

```
300 FOR I=0 TO 343
310 S=2: X=I: Y=55: GOSUB 1000
320 S=4: X=343-I: Y=220: GOSUB 1000
330 NEXT I
1000 POKE 53248+2*S, X-256*INT(X/256)
1010 POKE 53249+2*S, Y
1020 POKE 53264, PEEK(53264)AND(255-2↑S)
OR(2↑S*INT(X/256))
1030 RETURN
```

4. The following program lines smoothly scroll SPRITE 5 into view at the top of the screen.

```
300 X=100
310 POKE 53258, X
320 POKE 53264, PEEK(53264) AND 223
OR 32*INT(X/256)
330 FOR Y=0 TO 60
340 POKE 53259, Y
350 NEXT Y
```

## Comments

1. Whether or not a SPRITE is displayed in normal or expanded mode, the position POKEd into memory is always used for the upper left-hand corner of the SPRITE. Thus, for example, if a SPRITE is visible on the screen in normal mode and is then expanded in the vertical direction, the SPRITE will expand downward.

2. When a SPRITE is moved toward the right edge of the expanded screen, any part of the SPRITE which moves off the right edge reappears at the left edge. This “wraparound” effect is not normally seen since the edges of the expanded screen are not visible, but can cause unexpected collisions between SPRITEs if not taken into account.

One application which makes use of wraparound comes when working with a SPRITE which is expanded in the X, i.e. horizontal, direction. A horizontally expanded SPRITE can be up to 48 dots in width. If such a SPRITE is placed at the left edge of the expanded screen, the right half of the SPRITE will be located in the “window”, and therefore be visible. Thus, at first, it might seem impossible to move a

## Moving SPRITEs

horizontally expanded **SPRITE** smoothly in and out of view on the left side of the screen. However, the following program segment illustrates how a **SPRITE**, number 2 in this example, can be scrolled smoothly onto the left side of the visible screen by starting with the **SPRITE** near the right edge of the expanded screen.

```
300 X=488: Y=100
310 POKE 53252, X-256*INT(X/256)
320 POKE 53253, Y
330 POKE 53264, PEEK(53264) AND 251 OR
4*INT(X/256)
340 IF X=511 THEN X=0: GOTO 310
350 IF X=30 THEN END
360 X=X+1: GOTO 310
```

3. Wraparound also occurs when a **SPRITE** is moved toward the bottom edge of the expanded screen, with any part of the **SPRITE** which moves off the bottom edge reappearing at the top of the expanded screen. However, unlike right to left wraparound, this feature is not needed to smoothly scroll a **SPRITE** in and out of view at the top of the screen, since, when positioned at the top of the expanded screen, even a vertically expanded **SPRITE** will not appear in the window which is the normal screen. See Example 4.

4. The values **X** and **Y** for the horizontal and vertical positions of a **SPRITE** may be given as numeric constants, variables, or expressions. These values may be of either integer or floating-point precision, but the **POKE** statement will truncate all values to whole numbers. If **X** or **Y** is negative or **Y** is greater than 255, the message ?ILLEGAL QUANTITY ERROR results. If **X** is greater than 511, no error message will occur, but the positioning of several **SPRITEs** may not be as expected.

5. Two or more **SPRITEs** may be located in exactly the same position at the same time. Following the rules of **SPRITE** priorities, when two **SPRITEs** occupy the same location, the lower numbered **SPRITE** will always be fully visible, while the higher numbered **SPRITE** will only be seen through holes in the lower numbered **SPRITE**, and, if the higher numbered **SPRITE** is expanded while the lower numbered **SPRITE** is not, then the bottom and/or right half of the higher numbered **SPRITE** also will be visible.

# SPRITE Display Priorities

To achieve realism, when two objects cross paths on the screen, one object should appear to pass in front of the other. This is done by giving objects on the screen a *display priority*. When two objects occupy a portion of the screen at the same time, the object with the higher display priority is shown.

The display priority between SPRITES is preset. When two SPRITES cross on the screen, the lower-numbered SPRITE has the higher priority. Thus, SPRITE 0 crosses in front of all other SPRITES, and SPRITE 7 passes behind all other SPRITES.

Whether a SPRITE passes in front of or behind regular characters on the screen is independently selected for each SPRITE. The statement

```
POKE 53275, PEEK(53275) OR 2↑S
```

gives SPRITE number S lower priority than regular characters, while the statement

```
POKE 53275, PEEK(53275) AND (255-2↑S)
```

gives SPRITE number S higher priority than regular characters.

## Examples

1. The program line

```
100 POKE 53275, PEEK(53275) OR 1
```

will cause SPRITE 0 to pass behind all regular characters on the screen.

2. The program line

```
100 POKE 53275, PEEK(53275) AND 251
```

will cause SPRITE 2 to pass in front of all regular characters on the screen. SPRITE 2 will also pass behind SPRITE 0 and 1 and in front of SPRITES 3 thru 7, if any of these SPRITES also are displayed.

## Comments

1. The statement `POKE 53275,0` gives all SPRITES higher priority

## SPRITE Display Priorities

than regular characters, while the statement `POKE 53275,255` gives all SPRITEs lower priority than regular characters.

2. The priority of all 8 SPRITEs with respect to regular characters can be set in one statement. If SPRITEs  $S_1, S_2, \dots$ , are the SPRITEs that are to have lower priority than regular characters, and if  $N$  equals  $2 \uparrow S_1 + 2 \uparrow S_2 + \dots$ , then the statement `POKE 53275,N` will give SPRITEs  $S_1, S_2$ , etc., lower priority than regular characters, and all other SPRITEs higher priority than regular characters. For instance, since  $2 \uparrow 1 + 2 \uparrow 2 + 2 \uparrow 4 = 22$ , the statement `POKE 53275,22` gives SPRITEs 1, 2, and 4 lower priority than regular characters, and SPRITEs 0, 3, 5, 6, and 7 higher priority.

3. When the computer is powered up, memory location 53275 assumes the value 0. Thus the *default* condition is for all SPRITEs to have higher priority than regular characters. This default condition can be regained at any time by pressing `RUN/STOP-RESTORE`.

# SPRITE Collision Detection

Programs written using SPRITES often have SPRITES bouncing off, exploding, or vanishing when coming into contact with other SPRITES or fixed background characters. Programming these effects requires knowing when a SPRITE comes into contact with another SPRITE or a background character. The Commodore's Video Interface Chip, which controls the display of SPRITES and regular characters, makes this easy by automatically recording in memory both when a SPRITE collides with another SPRITE and when a SPRITE collides with a background character. By PEEKing the values in these memory locations, it is easy to detect collisions involving SPRITES. This automatic detection of collisions involving SPRITES, which is built into the Video Interface Chip, makes the Commodore a good video-games machine, as opposed to the more expensive, business-oriented computers which do not have this feature.

If SPRITE number  $S$  has collided with any other SPRITE, then the value of the expression

`PEEK(53278) AND 2↑S`

will be  $2↑S$ . Otherwise, no collision has occurred involving SPRITE  $S$  with other SPRITES, and the value of the expression will be 0. Thus a collision involving SPRITE  $S$  with other SPRITES can be checked by a statement of the form `IF (PEEK(53278) AND 2↑S) = 2↑S THEN action`. Note: both sets of parentheses are required in this statement.

Similarly, the value of the expression

`PEEK(53279) AND 2↑S`

will be  $2↑S$  if SPRITE  $S$  has collided with a regular character on the screen, and 0 otherwise. Thus the statement `IF (PEEK(53279) AND 2↑S) = 2↑S THEN action` can be used to execute *action* when SPRITE  $S$  collides with a background character.

## Examples

1. The statement

```
100 IF (PEEK(53278) AND 1) = 1 THEN GOSUB 1000
```

will cause a branch to the subroutine at line 1000 if SPRITE 0 has collided with another SPRITE.

# SPRITE Collision Detection

## 2. The statement

```
200 IF (PEEK(53279) AND 32) = 32 THEN GOSUB 2000
```

will cause a branch to the subroutine at line 2000 if SPRITE 5 has collided with any background characters.

## Comments

1. When a collision occurs between a SPRITE and another SPRITE or a background character, the incident is recorded by the VIC chip at locations 53278 or 53279, and remains recorded there even after the collision has ended. Thus, a collision does not have to be noticed by your program the instant it occurs, but can be checked for at any time. However, when PEEK reads the value at locations 53278 or 53279, the value is automatically reset, and all information about past collisions is lost. Since information regarding all 8 SPRITES is stored in these locations, and a program is likely to deal with only one SPRITE at a time, it is best to place the value PEEKed from location 53278 or 53279 into a variable. This variable can then be used in as many tests as desired to see what collisions, if any, have occurred. For example:

```
300 COL=PEEK(53278)
310 IF (COL AND 1) = 1 THEN GOSUB 1000
320 IF (COL AND 2) = 2 THEN GOSUB 2000
330 IF COL = 0 THEN GOSUB 5000: GOTO 300
```

Lines 310 and 320 cause a branch to an appropriate subroutine if SPRITES 0 or 1 have been involved in a collision with another SPRITE. If no collisions are recorded for any SPRITES ( $COL = 0$ ), then a subroutine beginning at line 5000 is called to move the SPRITES, after which checks are again made for any collisions.

2. Let COL be assigned the value of PEEK(53278). The statement  $IF (COL AND 2 \uparrow S) = 2 \uparrow S THEN action$  causes *action* to be executed when SPRITE S collides with *any* other SPRITE. To test if two particular SPRITES S and T have collided with each other, a statement of the form  $IF (COL AND 2 \uparrow S) = 2 \uparrow S AND (COL AND 2 \uparrow T) = 2 \uparrow T THEN action$  can be used. However, if SPRITES S and T have each collided with another SPRITE but not with each other, the condition given will still test out as true, and *action* will be executed.

3. Collisions occur when the “visible” portion of a SPRITE overlaps the “visible” portion of another SPRITE or a background character,

## SPRITE Collision Detection

not when the rectangle of dots associated with the SPRITE data overlaps another SPRITE or background character. When using multi-color mode, multi-color 1 is treated as *invisible*, just like the background color, though of course multi-color 1 is visible to us. Thus, if you wish to design a SPRITE which will never “collide” with other SPRITES, use multi-color 1 exclusively in coloring the SPRITE.

4. If a SPRITE has been enabled, then it can be involved in collisions, whether it is in view or positioned out of view on the “expanded” screen. A SPRITE which is positioned at the far right side of the expanded screen and “wraps around” to the left side can be involved in collisions at both ends of the screen. The same is true for a SPRITE positioned at the bottom of the expanded screen and wrapping around to the top. See Moving SPRITES for a discussion of the expanded screen and SPRITES “wrapping around” the screen.





## Part III: Music Synthesis



# Music Note Selection

The music synthesizer in the Commodore 64 can produce notes over a complete 8-octave range, covering the range of a standard piano. Each note has associated with it a specific *frequency* or rate of vibrations in the air. Frequencies are normally measured in cycles per second or *Hertz*. The note A above middle C has a frequency of 440 Hertz. If the frequency is doubled, the new note is one octave above the old. Thus 880 Hertz is the frequency of the note A in the second octave above middle C (see Figure 1). Similarly, if the frequency is halved, the new note is one octave below the old. Thus 220 Hertz is the frequency of the note A below middle C.

The Commodore 64's music synthesizer uses a *scaled* frequency to decide which note is to be played. Each scaled frequency used by the Commodore 64 is approximately 16 times the normal Hertz frequency for the note. The following table gives the scaled frequency used by the synthesizer to produce each note from middle C to the C above middle C. A complete table for all 8 octaves is given in Appendix S.

| Note | Scaled Frequency |
|------|------------------|
| C    | 4291             |
| C#   | 4547             |
| D    | 4817             |
| D#   | 5103             |
| E    | 5407             |
| F    | 5728             |
| F#   | 6069             |
| G    | 6430             |
| G#   | 6812             |
| A    | 7217             |
| A#   | 7657             |
| B    | 8101             |
| C    | 8583             |

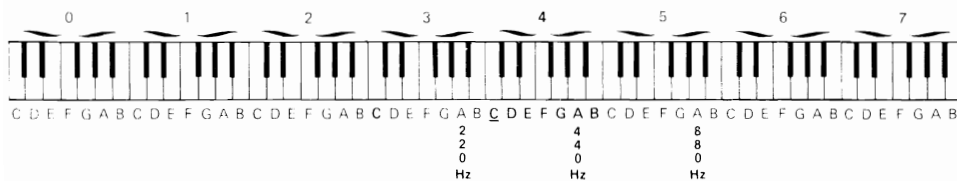


FIGURE 1

# Music Note Selection

The synthesizer can play 3 notes or *voices* at the same time. If F is a whole number between 0 and 65535, then the statements

```
POKE 54272, F-256*INT(F/256)
POKE 54273, INT(F/256)
```

prepare voice 1 to play the note corresponding to the scaled frequency F. To prepare voice 2 to play this same note, use the statements

```
POKE 54279, F-256*INT(F/256)
POKE 54280, INT(F/256)
```

and to prepare voice 3, use the statements

```
POKE 54286, F-256*INT(F/256)
POKE 54287, INT(F/256)
```

## Examples

1. The statements

```
100 POKE 54272, 4291-256*INT(4291/256)
110 POKE 54273, INT(4291/256)
```

prepare voice 1 to play middle C.

2. The statements

```
200 FOR I=0 TO 14 STEP 7
210 READ F
220 DATA 5728,7217,8583
230 POKE 54272+I, F-256*INT(F/256)
240 POKE 54273+I, INT(F/256)
250 NEXT I
```

assign voices 1, 2, and 3 the scaled frequencies for the notes F above middle C, A above middle C, and C above middle C, respectively.

3. The statements

```
300 POKE 53279, 6430-256*INT(6430/256)
310 POKE 53280, INT(6430/256)
320 FOR I=1 TO 3500: NEXT I
```

# Music Note Selection

```
330 POKE 53279, 12860-256*INT(12860/256)
340 POKE 53280, INT(12860/256)
```

assign voice 2 the scaled frequency of the note G above middle C, then, after approximately five seconds, assign voice 2 twice this value, which is the scaled frequency for the note G in the second octave above middle C.

## Comments

1. The frequency assigned to a voice may be changed at any time without altering any of the other parameters that control the voice. The following example uses this feature to play the C major scale using voice 1.

```
10 POKE 54296,15: REM SET VOLUME
20 POKE 54278,15*16: REM SET SUSTAIN LEVEL
30 POKE 54276,33: REM SELECT WAVEFORM
AND TURN ON VOICE
40 FOR I=1 TO 8: REM PLAY 8 NOTES
50 READ F
60 DATA 4291,4817,5407,5728,6430,7217
70 DATA 8101,8583
80 POKE 54272,F-256*INT(F/256)
90 POKE 54273,INT(F/256)
100 FOR J=1 TO 200: NEXT J
110 NEXT I
120 POKE 54276,0: REM TURN OFF VOICE
```

The statements in lines 10, 20, 30, and 120 will be explained later, in our discussion of music synthesis.

2. Having to specify notes by their scaled frequency can be inconvenient, especially if the user of a program is to select the note. It is much easier to enter music data if we can simply give the letter name of the note and its octave. The following program allows this form of data input. It makes use of the fact that the scaled frequency for any note can be obtained from the scaled frequency of the same named note in the highest octave by dividing by 2 an appropriate number of times. For example, the scaled frequency of the C in octave 3 is 2145. This same value can be obtained from the scaled frequency 34334 for C in the highest octave by dividing by 16 (the same as dividing by 2 four times) and using the INT function:  $2145 = \text{INT}(34334/16)$ . To end the program, enter Q for note name and any number for octave.

## Music Note Selection

```
5 DIM F(12)
10 FOR I=1 TO 12
20 READ F(I): REM HI OCTAVE FREQUENCIES
30 DATA 34334,36376,38539,40830,43258,45830
40 DATA 48556,51443,54502,57743,61176,64814
50 NEXT I
60 POKE 54296,15: REM SET VOLUME
70 POKE 54278,15*16: REM SET SUSTAIN LEVEL
80 PRINT: INPUT "NOTE NAME";A$
90 INPUT "OCTAVE (0-7; 4 IS MIDDLE)";N
100 IF A$="C" THEN I=1: GOTO 300
110 IF A$="C#" THEN I=2: GOTO 300
120 IF A$="D-" THEN I=2: GOTO 300
130 IF A$="D" THEN I=3: GOTO 300
140 IF A$="D#" THEN I=4: GOTO 300
150 IF A$="E-" THEN I=4: GOTO 300
160 IF A$="E" THEN I=5: GOTO 300
170 IF A$="F" THEN I=6: GOTO 300
180 IF A$="F#" THEN I=7: GOTO 300
190 IF A$="G-" THEN I=7: GOTO 300
200 IF A$="G" THEN I=8: GOTO 300
210 IF A$="G#" THEN I=9: GOTO 300
220 IF A$="A-" THEN I=9: GOTO 300
230 IF A$="A" THEN I=10: GOTO 300
240 IF A$="A#" THEN I=11: GOTO 300
250 IF A$="B-" THEN I=11: GOTO 300
260 IF A$="B" THEN I=12: GOTO 300
270 IF A$="Q" THEN END
280 PRINT "USE A,B,C,D,E,F, OR G, (#=SHARP, -=FLAT)"
290 GOTO 80
300 SF=INT(F(I)/(2↑(7-N)))
310 POKE 54272,SF-256*INT(SF/256)
320 POKE 54273,INT(SF/256)
330 POKE 54276,33: REM SELECT WAVEFORM
AND TURN ON VOICE 1
340 FOR J=1 TO 500: NEXT J
350 POKE 54276,32: REM RELEASE VOICE 1
360 GOTO 80
```

It is possible to eliminate the need to specify the octave with each note. To do this, a variable, `OCTAVE`, is added to the program. `OCTAVE` is assigned an initial value, perhaps 4, that remains in effect until the user types in a special symbol to move up or down an octave. See Appendix R for a program which uses this technique in giving data for a 3-voice chorus.

# Music Volume and Amplitude Envelope

The loudness or volume of sound produced by a voice is controlled in two manners. First, a single, maximum volume level is chosen for all three voices. Second, for each voice, an *amplitude envelope* is chosen. This amplitude envelope determines how rapidly a voice rises to its maximum volume, whether it is sustained at some intermediate volume level, and how quickly it drops back to zero volume when the voice is released. The amplitude envelope chosen, together with the waveform selected, affects whether the voice will sound like a guitar, a piano, an organ, or another instrument.

The overall volume control for all three voices can be set at any one of 16 values. No sound is produced by a setting of 0, while maximum possible volume results from a setting of 15. A setting of 7 will produce half the volume of a setting of 15, a setting of 4 one quarter the volume, and so forth. If  $V$  is an integer between 0 and 15, then the statement

```
POKE 54296, V
```

sets the volume level at  $V$ .

The amplitude envelope for a voice is determined by four values. These values control the *Attack*, *Decay*, *Sustain*, and *Release* segments of the envelope, as illustrated in Figure 1.

**Attack:** Controls how rapidly the volume of a voice rises from zero when the voice is turned on, to the overall maximum volume level set for all three voices described above.

**Decay:** Controls how rapidly the volume of a voice falls from the maximum level to the fraction of the maximum level specified by the Sustain parameter.

**Sustain:** Controls the volume level, relative to the maximum level reached in the Attack, at which a voice is kept until the voice is released.

**Release:** Controls how rapidly the volume of a voice drops from the Sustain level to zero, once the voice is released.

If  $A$ ,  $D$ ,  $S$ , and  $R$  are values for Attack, Decay, Sustain, and Release chosen from the tables below, then the statements

```
POKE 54277, A*16 + D
POKE 54278, S*16 + R
```

assign these values for voice 1. The corresponding statements for voice 2 are

```
POKE 54284, A*16 + D
POKE 54285, S*16 + R
```

# Music Volume and Amplitude Envelope

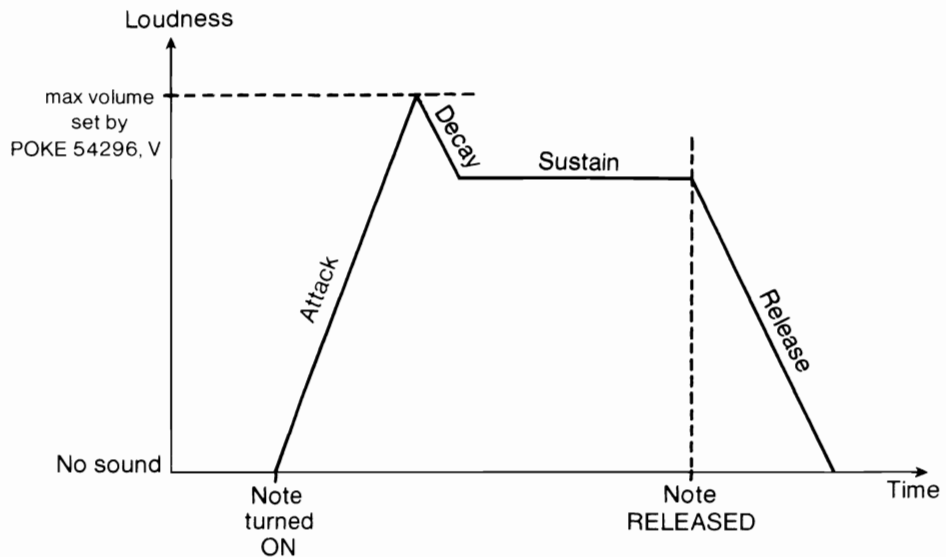


FIGURE 1

and for voice 3 are

POKE 54291, A\*16+D  
 POKE 54292, S\*16+R

| <u>value</u> | <u>ATTACK</u><br>time to<br>full<br>volume | <u>DECAY</u><br>time to<br>full<br>volume | <u>SUSTAIN</u><br>percent<br>of full<br>volume | <u>RELEASE</u><br>time to<br>zero<br>volume |
|--------------|--------------------------------------------|-------------------------------------------|------------------------------------------------|---------------------------------------------|
| 0            | 2 ms                                       | 6 ms                                      | 0                                              | 6 ms                                        |
| 1            | 8 ms                                       | 24 ms                                     | 7                                              | 24 ms                                       |
| 2            | 16 ms                                      | 48 ms                                     | 13                                             | 48 ms                                       |
| 3            | 24 ms                                      | 72 ms                                     | 20                                             | 72 ms                                       |
| 4            | 38 ms                                      | 114 ms                                    | 27                                             | 114 ms                                      |
| 5            | 56 ms                                      | 168 ms                                    | 33                                             | 168 ms                                      |
| 6            | 68 ms                                      | 204 ms                                    | 40                                             | 204 ms                                      |
| 7            | 80 ms                                      | 240 ms                                    | 47                                             | 240 ms                                      |
| 8            | 100 ms                                     | 300 ms                                    | 53                                             | 300 ms                                      |
| 9            | 250 ms                                     | 750 ms                                    | 60                                             | 750 ms                                      |



# Music Volume and Amplitude Envelope

|    |        |       |     |       |
|----|--------|-------|-----|-------|
| 10 | 500 ms | 1.5 s | 67  | 1.5 s |
| 11 | 800 ms | 2.4 s | 73  | 2.4 s |
| 12 | 1 s    | 3 s   | 80  | 3 s   |
| 13 | 3 s    | 9 s   | 87  | 9 s   |
| 14 | 5 s    | 15 s  | 93  | 15 s  |
| 15 | 8 s    | 24 s  | 100 | 24 s  |

## Examples

1. The statement

```
10 POKE 54296,15
```

sets the overall volume control at maximum.

2. The easiest instrument to imitate is an organ. The attack is immediate so  $A=0$ . Set  $D=0$ , even though the value of  $D$  will not matter since the sustain will be at 100%. The sustain is at full volume, so  $S=15$ . Finally the release is immediate, so  $R=0$ . Thus, the following statements could be used to set up voice 1 as an organ.

```
100 POKE 54277,0*16+0
110 POKE 54278,15*16+0
```

or

```
100 POKE 54277,0
110 POKE 54278,240
```

Below is a list of suggested values for  $A$ ,  $D$ ,  $S$ , and  $R$  to imitate various families of instruments.

|                                    | <u>A</u> | <u>D</u> | <u>S</u> | <u>R</u> |
|------------------------------------|----------|----------|----------|----------|
| strings (violin, cello, . . .)     | 10       | 8        | 10       | 9        |
| percussion (drums, cymbals, . . .) | 0        | 9        | 0        | 9        |
| piano, harpsichord                 | 0        | 9        | 0        | 0        |
| organ                              | 0        | 0        | 15       | 0        |
| brass (trumpet, tuba, . . .)       | 5        | 8        | 12       | 0        |

3. The following subroutine can be added to the program in Comment 2 of Music Note Selection. Then, if line 70 of that program is deleted, and the line 325 GOSUB 600 is added, the program not only will allow the user to select the note to be played, but also will allow the amplitude envelope to be specified.

# Music Volume and Amplitude Envelope

```
600 PRINT: INPUT "ATTACK RATE (0-15)";A
610 INPUT "DECAY RATE (0-15)";D
620 INPUT "SUSTAIN LEVEL (0-15)";S
630 INPUT "RELEASE RATE (0-15)";R
640 POKE 54277,A*16+D
650 POKE 54278,S*16+R
660 RETURN
```

## Comments

1. At least one of the values for Attack, Decay, or Sustain must be non-zero if the voice is to be heard.
2. The attack phase begins as soon as the voice is turned on by the appropriate POKE into locations 54276, 54283, and 54290 as discussed in Music Waveforms and On/Off Control. The release phase begins as soon as the voice is "released" (see the same discussion). Note that if a value of 0 is POKEd into any or all of the locations 54276, 54283, and 54290, the corresponding voices will be "turned off". When a voice is turned off, the release phase occurs at a frequency of 0, which produces no sound, and thus the volume of the voice falls immediately to zero.
3. The values for Attack, Decay, Sustain, and Release may be changed as desired, although normally they are set early in a program and not altered.

# Music Waveforms and On/Off Control

When two different types of instruments play the same note, we hear the same frequency in each case, but with different tonal qualities. These tonal qualities correspond in part to different *waveforms* produced by each instrument. The Commodore 64 produces sounds using one of four different waveforms, allowing it to sound like many different instruments. The *triangular* waveform is low in harmonics and has a mellow, flute-like quality. The *sawtooth* waveform is rich in harmonics and has a lively, brassy quality. The *pulse* waveform can be controlled to produce varying harmonic contents, with tones ranging from a bright, hollow square wave to a nasal, reedy pulse. The *noise* waveform produces random sound, which varies from a low rumbling to a sharp hissing as the frequency of the voice is changed from low to high values.

The same memory location which controls the waveform chosen for a voice also controls whether a voice is turned on or off. Thus, the waveform for a voice is selected as the voice is turned on. If N is a value chosen from the following table

| <u>Waveform</u> | <u>Value for N</u> |
|-----------------|--------------------|
| triangular      | 16                 |
| sawtooth        | 32                 |
| pulse           | 64                 |
| noise           | 128                |

then the statement

```
POKE 54276, N + 1
```

turns voice 1 on and selects the waveform corresponding to N. (Turning a voice on does not guarantee that a sound will be heard. See Comment 1 below.) The statement

```
POKE 54276, N
```

“releases” the voice, causing the volume of the voice to begin its drop to zero (see Music Volume and Amplitude Envelope).

The corresponding statements for voice 2 are

```
voice on: POKE 54283, N + 1
release voice: POKE 54283, N
```

# Music Waveforms and On/Off Control

and for voice 3 are

```
voice on: POKE 54290, N + 1
release voice: POKE 54290, N
```

## Examples

1. The statement

```
POKE 54276,17
```

turns voice 1 on with a triangular waveform.

2. The lines

```
100 POKE 54283,33
110 FOR J=1 TO 180: NEXT J
120 POKE 54283,32
```

turn voice 2 on with a sawtooth waveform for about 1/4 second, then release it.

## Comments

1. Several steps must be performed in order to produce a note from the music synthesizer, of which turning on the voice and selecting a waveform are only two. Sound will be produced when a voice is turned on (and waveform selected) only if the following steps have also been performed: selecting a frequency (the note to be played), setting the volume, and selecting an amplitude envelope. These topics are covered in the other sections on music synthesis.

2. If the pulse waveform is selected, then an additional value, called the *pulse width*, also must be selected. Pulse widths can vary from 0 to 100 percent, producing tones which vary from nasal and reedy (for low widths) to bright and hollow (for intermediate widths) and back to nasal and reedy (for high widths). If voice 1 is to be a pulse wave, and a width of N percent is desired, then the statements

# Music Waveforms and On/Off Control

```
POKE 54274, N*40.95-256*INT(N*40.95/256)
POKE 54275, INT(N*40.95/256)
```

must be executed before the statement POKE 54276,65 is used to select the pulse waveform and turn on the voice. The corresponding statements for voice 2 are

```
POKE 54281, N*40.95-256*INT(N*40.95/256)
POKE 54282, INT(N*40.95/256)
```

and for voice 3 are

```
POKE 54288, N*40.95-256*INT(N*40.95/256)
POKE 54289, INT(N*40.95/256)
```

The following subroutine can be added to the program in Comment 2 of Music Note Selection. Then, by changing line 330 of that program to 330 GOSUB 400 and line 350 to 350 POKE 54276,WF, the program will allow you not only to select the note to be played, but also to select the waveform to be used.

```
400 PRINT: PRINT "WAVEFORM ?"
410 PRINT "TRIANGULAR = 16"
420 PRINT "SAWTOOTH = 32"
430 PRINT "PULSE = 64"
440 PRINT "NOISE = 128"
450 INPUT "SELECTION";WF
460 IF WF<>64 THEN 500
470 INPUT "PULSE WIDTH PERCENT(0-100)";W
480 POKE 54274,W*40.95-256*INT(W*40.95/256)
490 POKE 54275,INT(W*40.95/256)
500 POKE 54276,WF+1
510 RETURN
```

3. As illustrated in Example 2, the duration of a note can be controlled by executing a delaying FOR . . . NEXT loop between the statements which turn the note on and release it.

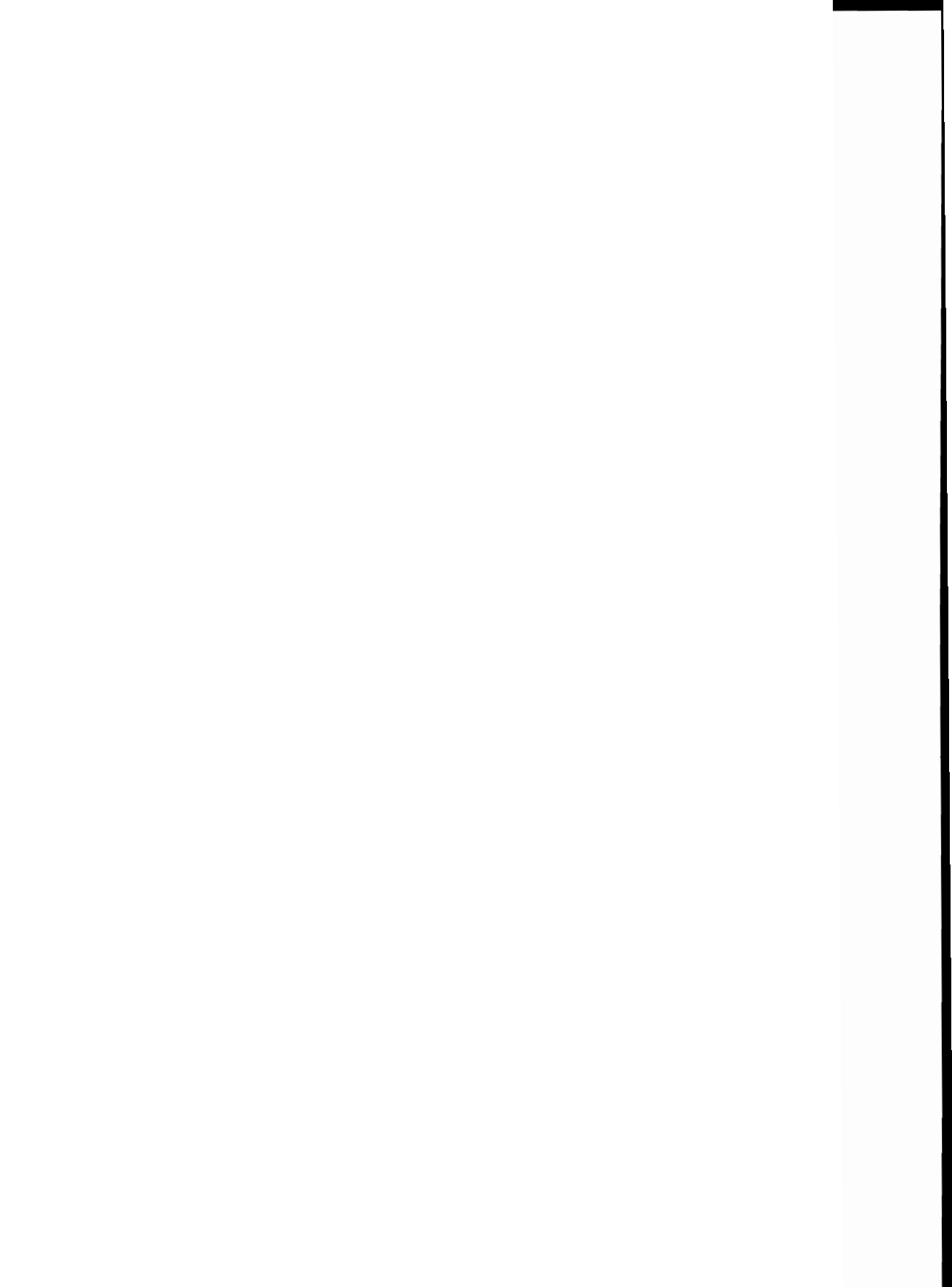
4. POKEing 0 into any or all of locations 54276, 54283, and 54290 cause the release phase of the corresponding voice to occur at a frequency of 0, which produces no sound. POKEing 0 into these locations "turns off" the corresponding voices, which is a good idea as the last action in a music program. When playing notes within a program, an

## Music Waveforms and On/Off Control

immediate drop of volume usually is obtained by releasing the voice after POKEing a release rate of 0 rather than by turning off the voice.

5. A voice need not be released or turned off between different notes. Instead, the frequency assigned to the voice can be changed after the completion of a delaying FOR. . .NEXT loop. This will produce a flowing, legato composition. See the program in Comment 1 of Music Note Selection.

# Appendices





**Table 1: ASCII/Commodore Values  
Uppercase/Graphics Mode**

| ASCII Value | Effect              | Key(s)           | Quote Mode Display |
|-------------|---------------------|------------------|--------------------|
| 0           |                     | CTRL-@           | ␣                  |
| 1           |                     | CTRL-A           | ␣                  |
| 2           |                     | CTRL-B           | ␣                  |
| 3           |                     | CTRL-C           | ␣                  |
| 4           |                     | CTRL-D           | ␣                  |
| 5           | white foreground    | CTRL-2 or CTRL-E | ␣                  |
| 6           |                     | CTRL-F           | ␣                  |
| 7           |                     | CTRL-G           | ␣                  |
| 8           | disable SHIFT-␣     | CTRL-H           | ␣                  |
| 9           | enable SHIFT-␣      | CTRL-I           | ␣                  |
| 10          |                     | CTRL-J           | ␣                  |
| 11          |                     | CTRL-K           | ␣                  |
| 12          |                     | CTRL-L           | ␣                  |
| 13          | start new line      | RETURN or CTRL-M | ␣                  |
| 14          | switch to lowercase | CTRL-N           | ␣                  |
| 15          |                     | CTRL-O           | ␣                  |
| 16          |                     | CTRL-P           | ␣                  |
| 17          | cursor down         | CRSR↓↑ or CTRL-Q | ␣                  |
| 18          | reverse-video on    | CTRL-9 or CTRL-R | ␣                  |
| 19          | home cursor         | HOME or CTRL-S   | ␣                  |
| 20          | delete character    | DEL or CTRL-T    | ␣                  |
| 21          |                     | CTRL-U           | ␣                  |
| 22          |                     | CTRL-V           | ␣                  |
| 23          |                     | CTRL-W           | ␣                  |
| 24          |                     | CTRL-X           | ␣                  |
| 25          |                     | CTRL-Y           | ␣                  |
| 26          |                     | CTRL-Z           | ␣                  |
| 27          |                     | CTRL-[           | ␣                  |
| 28          | red foreground      | CTRL-3 or CTRL-£ | ␣                  |
| 29          | cursor right        | CRSR⇌ or CTRL-]  | ␣                  |
| 30          | green foreground    | CTRL-6 or CTRL-↑ | ␣                  |
| 31          | blue foreground     | CTRL-7 or CTRL-= | ␣                  |

# Appendix A

Table 1: continued

| AV Sacter<br>Character<br>I e r | Key(s) | AV Sacter<br>Character<br>I e r | Key(s) | AV Sacter<br>Character<br>I e r | Key(s)  |     |   |
|---------------------------------|--------|---------------------------------|--------|---------------------------------|---------|-----|---|
| 32                              | ☐      | space                           | 64     | Ⓔ                               | @       | 96  | ☐ |
| 33                              | !      | SHIFT-1                         | 65     | Ⓐ                               | A       | 97  | ♣ |
| 34                              | "      | SHIFT-2                         | 66     | Ⓑ                               | B       | 98  |   |
| 35                              | #      | SHIFT-3                         | 67     | Ⓒ                               | C       | 99  | ☐ |
| 36                              | \$     | SHIFT-4                         | 68     | Ⓓ                               | D       | 100 | ☐ |
| 37                              | %      | SHIFT-5                         | 69     | Ⓔ                               | E       | 101 | ☐ |
| 38                              | &      | SHIFT-6                         | 70     | Ⓕ                               | F       | 102 | ☐ |
| 39                              | '      | SHIFT-7                         | 71     | Ⓖ                               | G       | 103 |   |
| 40                              | (      | SHIFT-8                         | 72     | Ⓗ                               | H       | 104 |   |
| 41                              | )      | SHIFT-9                         | 73     | Ⓘ                               | I       | 105 | ♠ |
| 42                              | *      | *                               | 74     | Ⓙ                               | J       | 106 | ♠ |
| 43                              | +      | +                               | 75     | Ⓚ                               | K       | 107 | ♠ |
| 44                              | ,      | ,                               | 76     | Ⓛ                               | L       | 108 | Ⓛ |
| 45                              | -      | -                               | 77     | Ⓜ                               | M       | 109 | Ⓜ |
| 46                              | .      | .                               | 78     | Ⓝ                               | N       | 110 | Ⓝ |
| 47                              | /      | /                               | 79     | Ⓞ                               | O       | 111 | ☐ |
| 48                              | 0      | 0                               | 80     | Ⓟ                               | P       | 112 | ☐ |
| 49                              | 1      | 1                               | 81     | Ⓠ                               | Q       | 113 | ♣ |
| 50                              | 2      | 2                               | 82     | Ⓡ                               | R       | 114 | ☐ |
| 51                              | 3      | 3                               | 83     | Ⓢ                               | S       | 115 | ♣ |
| 52                              | 4      | 4                               | 84     | Ⓣ                               | T       | 116 |   |
| 53                              | 5      | 5                               | 85     | Ⓤ                               | U       | 117 | ♠ |
| 54                              | 6      | 6                               | 86     | Ⓥ                               | V       | 118 | ✕ |
| 55                              | 7      | 7                               | 87     | Ⓦ                               | W       | 119 | Ⓞ |
| 56                              | 8      | 8                               | 88     | ✕                               | X       | 120 | ✕ |
| 57                              | 9      | 9                               | 89     | Ⓨ                               | Y       | 121 |   |
| 58                              | :      | :                               | 90     | Ⓩ                               | Z       | 122 | ♠ |
| 59                              |        | ;                               | 91     |                                 | SHIFT-: | 123 | + |
| 60                              | Ⓚ      | SHIFT-,                         | 92     | £                               | £       | 124 | ♠ |
| 61                              | =      | =                               | 93     |                                 | SHIFT-; | 125 |   |
| 62                              | Ⓛ      | SHIFT-.                         | 94     | ↑                               | ↑       | 126 | ♠ |
| 63                              | ?      | SHIFT-/                         | 95     | ←                               | ←       | 127 | ♠ |

# Appendix A

Table 1: continued

| ASCII Value | Effect               | Key(s)       | Quote Mode Display |
|-------------|----------------------|--------------|--------------------|
| 128         |                      |              | ▬                  |
| 129         | orange foreground    | ␣-1          | ▬                  |
| 130         |                      |              | ▬                  |
| 131         |                      |              | ▬                  |
| 132         |                      |              | ▬                  |
| 133         | as programmed        | F1           | ▬                  |
| 134         | as programmed        | F3           | ▬                  |
| 135         | as programmed        | F5           | ▬                  |
| 136         | as programmed        | F7           | ▬                  |
| 137         | as programmed        | SHIFT-F1     | ▬                  |
| 138         | as programmed        | SHIFT-F3     | ▬                  |
| 139         | as programmed        | SHIFT-F5     | ▬                  |
| 140         | as programmed        | SHIFT-F7     | ▬                  |
| 141         | start new line       | SHIFT-RETURN | ▬                  |
| 142         | switch to uppercase  |              | ▬                  |
| 143         |                      |              | ▬                  |
| 144         | black foreground     | CTRL-1       | ▬                  |
| 145         | cursor up            | SHIFT-CRSR↑  | ▬                  |
| 146         | reverse-video off    | CTRL-0       | ▬                  |
| 147         | clear screen         | SHIFT-HOME   | ▬                  |
| 148         | insert character     | SHIFT-DEL    | ▬                  |
| 149         | brown foreground     | ␣-2          | ▬                  |
| 150         | light red foreground | ␣-3          | ▬                  |
| 151         | gray 1 foreground    | ␣-4          | ▬                  |
| 152         | gray 2 foreground    | ␣-5          | ▬                  |
| 153         | lt. green foreground | ␣-6          | ▬                  |
| 154         | lt. blue foreground  | ␣-7          | ▬                  |
| 155         | gray 3 foreground    | ␣-8          | ▬                  |
| 156         | purple foreground    | CTRL-5       | ▬                  |
| 157         | cursor left          | SHIFT-CRSR←  | ▬                  |
| 158         | yellow foreground    | CTRL-8       | ▬                  |
| 159         | cyan foreground      | CTRL-4       | ▬                  |

# Appendix A

Table 1: continued

| AV Sact Clt Iuere | Character | Key(s)  | AV Sact Clt Iuere | Character | Key(s)  | AV Sact Clt Iuere | Character | Key(s) |
|-------------------|-----------|---------|-------------------|-----------|---------|-------------------|-----------|--------|
|                   |           | SHIFT-  |                   |           |         |                   |           |        |
| 160               | □         | space   | 192               | ▣         | SHIFT-* | 224               | □         |        |
| 161               | ■         | ⌘-K     | 193               | ♠         | SHIFT-A | 225               | ■         |        |
| 162               | ▣         | ⌘-I     | 194               | ▢         | SHIFT-B | 226               | ▣         |        |
| 163               | ▣         | ⌘-T     | 195               | ▣         | SHIFT-C | 227               | ▣         |        |
| 164               | □         | ⌘-@     | 196               | ▣         | SHIFT-D | 228               | □         |        |
| 165               | ▢         | ⌘-G     | 197               | ▣         | SHIFT-E | 229               | ▢         |        |
| 166               | ⊞         | ⌘-+     | 198               | ▣         | SHIFT-F | 230               | ⊞         |        |
| 167               | ▢         | ⌘-M     | 199               | ▢         | SHIFT-G | 231               | ▢         |        |
| 168               | ⊞         | ⌘-£     | 200               | ▢         | SHIFT-H | 232               | ⊞         |        |
| 169               | ▣         | SHIFT-£ | 201               | ♠         | SHIFT-I | 233               | ▣         |        |
| 170               | ▢         | ⌘-N     | 202               | ♠         | SHIFT-J | 234               | ▢         |        |
| 171               | ▣         | ⌘-Q     | 203               | ♠         | SHIFT-K | 235               | ▣         |        |
| 172               | ▣         | ⌘-D     | 204               | ▣         | SHIFT-L | 236               | ▣         |        |
| 173               | ▣         | ⌘-Z     | 205               | ▣         | SHIFT-M | 237               | ▣         |        |
| 174               | ▣         | ⌘-S     | 206               | ▣         | SHIFT-N | 238               | ▣         |        |
| 175               | ▣         | ⌘-P     | 207               | ▣         | SHIFT-O | 239               | ▣         |        |
| 176               | ▣         | ⌘-A     | 208               | ▣         | SHIFT-P | 240               | ▣         |        |
| 177               | ▣         | ⌘-E     | 209               | ♠         | SHIFT-Q | 241               | ▣         |        |
| 178               | ▣         | ⌘-R     | 210               | ▣         | SHIFT-R | 242               | ▣         |        |
| 179               | ▣         | ⌘-W     | 211               | ♠         | SHIFT-S | 243               | ▣         |        |
| 180               | ▣         | ⌘-H     | 212               | ▢         | SHIFT-T | 244               | ▣         |        |
| 181               | ▣         | ⌘-J     | 213               | ♠         | SHIFT-U | 245               | ▣         |        |
| 182               | ▣         | ⌘-L     | 214               | ⊞         | SHIFT-V | 246               | ▣         |        |
| 183               | ▣         | ⌘-Y     | 215               | ▣         | SHIFT-W | 247               | ▣         |        |
| 184               | ▣         | ⌘-U     | 216               | ⊞         | SHIFT-X | 248               | ▣         |        |
| 185               | ▣         | ⌘-O     | 217               | ▢         | SHIFT-Y | 249               | ▣         |        |
| 186               | ▣         | SHIFT-@ | 218               | ♠         | SHIFT-Z | 250               | ▣         |        |
| 187               | ▣         | ⌘-F     | 219               | ⊞         | SHIFT-+ | 251               | ▣         |        |
| 188               | ▣         | ⌘-C     | 220               | ⊞         | ⌘-—     | 252               | ▣         |        |
| 189               | ▣         | ⌘-X     | 221               | ▢         | SHIFT-— | 253               | ▣         |        |
| 190               | ▣         | ⌘-V     | 222               | ▣         | SHIFT-↑ | 254               | ▣         |        |
| 191               | ▣         | ⌘-B     | 223               | ▣         | ⌘-*     | 225               | ▣         |        |

**Table 2: ASCII/Commodore Values  
Lowercase/Uppercase Mode**

| ASCII Value | Effect              | Key(s)           | Quote Mode Display |
|-------------|---------------------|------------------|--------------------|
| 0           |                     | CTRL-@           | ␣                  |
| 1           |                     | CTRL-A           | ␣                  |
| 2           |                     | CTRL-B           | ␣                  |
| 3           |                     | CTRL-C           | ␣                  |
| 4           |                     | CTRL-D           | ␣                  |
| 5           | white foreground    | CTRL-2 or CTRL-E | ␣                  |
| 6           |                     | CTRL-F           | ␣                  |
| 7           |                     | CTRL-G           | ␣                  |
| 8           | disable SHIFT-␣     | CTRL-H           | ␣                  |
| 9           | enable SHIFT-␣      | CTRL-I           | ␣                  |
| 10          |                     | CTRL-J           | ␣                  |
| 11          |                     | CTRL-K           | ␣                  |
| 12          |                     | CTRL-L           | ␣                  |
| 13          | start new line      | RETURN or CTRL-M | ␣                  |
| 14          | switch to lowercase | CTRL-N           | ␣                  |
| 15          |                     | CTRL-O           | ␣                  |
| 16          |                     | CTRL-P           | ␣                  |
| 17          | cursor down         | CRSR↓↑ or CTRL-Q | ␣                  |
| 18          | reverse-video on    | CTRL-9 or CTRL-R | ␣                  |
| 19          | home cursor         | HOME or CTRL-S   | ␣                  |
| 20          | delete character    | DEL or CTRL-T    | ␣                  |
| 21          |                     | CTRL-U           | ␣                  |
| 22          |                     | CTRL-V           | ␣                  |
| 23          |                     | CTRL-W           | ␣                  |
| 24          |                     | CTRL-X           | ␣                  |
| 25          |                     | CTRL-Y           | ␣                  |
| 26          |                     | CTRL-Z           | ␣                  |
| 27          |                     | CTRL-[           | ␣                  |
| 28          | red foreground      | CTRL-3 or CTRL-£ | ␣                  |
| 29          | cursor right        | CRSR⇐ or CTRL-]  | ␣                  |
| 30          | green foreground    | CTRL-6 or CTRL-↑ | ␣                  |
| 31          | blue foreground     | CTRL-7 or CTRL-= | ␣                  |

# Appendix A

Table 2: continued

| AV<br>Sact<br>Cl<br>Iu<br>Ie | C<br>h<br>a<br>r<br>a<br>c<br>t<br>e<br>r | Key(s)  | AV<br>Sact<br>Cl<br>Iu<br>Ie | C<br>h<br>a<br>r<br>a<br>c<br>t<br>e<br>r | Key(s)  | AV<br>Sact<br>Cl<br>Iu<br>Ie | C<br>h<br>a<br>r<br>a<br>c<br>t<br>e<br>r | Key(s) |
|------------------------------|-------------------------------------------|---------|------------------------------|-------------------------------------------|---------|------------------------------|-------------------------------------------|--------|
| 32                           | ␣                                         | space   | 64                           | @                                         | @       | 96                           | ▬                                         |        |
| 33                           | !                                         | SHIFT-1 | 65                           | A                                         | A       | 97                           | Ⓐ                                         |        |
| 34                           | "                                         | SHIFT-2 | 66                           | B                                         | B       | 98                           | Ⓑ                                         |        |
| 35                           | #                                         | SHIFT-3 | 67                           | C                                         | C       | 99                           | Ⓒ                                         |        |
| 36                           | \$                                        | SHIFT-4 | 68                           | D                                         | D       | 100                          | Ⓓ                                         |        |
| 37                           | %                                         | SHIFT-5 | 69                           | E                                         | E       | 101                          | Ⓔ                                         |        |
| 38                           | &                                         | SHIFT-6 | 70                           | F                                         | F       | 102                          | Ⓕ                                         |        |
| 39                           | ^                                         | SHIFT-7 | 71                           | G                                         | G       | 103                          | Ⓖ                                         |        |
| 40                           | (                                         | SHIFT-8 | 72                           | H                                         | H       | 104                          | Ⓗ                                         |        |
| 41                           | )                                         | SHIFT-9 | 73                           | I                                         | I       | 105                          | Ⓐ                                         |        |
| 42                           | *                                         | *       | 74                           | J                                         | J       | 106                          | Ⓙ                                         |        |
| 43                           | +                                         | +       | 75                           | K                                         | K       | 107                          | Ⓚ                                         |        |
| 44                           | ,                                         | ,       | 76                           | L                                         | L       | 108                          | Ⓛ                                         |        |
| 45                           | -                                         | -       | 77                           | M                                         | M       | 109                          | Ⓜ                                         |        |
| 46                           | .                                         | .       | 78                           | N                                         | N       | 110                          | Ⓝ                                         |        |
| 47                           | /                                         | /       | 79                           | O                                         | O       | 111                          | Ⓞ                                         |        |
| 48                           | 0                                         | 0       | 80                           | P                                         | P       | 112                          | Ⓟ                                         |        |
| 49                           | 1                                         | 1       | 81                           | Q                                         | Q       | 113                          | Ⓠ                                         |        |
| 50                           | 2                                         | 2       | 82                           | R                                         | R       | 114                          | Ⓡ                                         |        |
| 51                           | 3                                         | 3       | 83                           | S                                         | S       | 115                          | Ⓢ                                         |        |
| 52                           | 4                                         | 4       | 84                           | T                                         | T       | 116                          | Ⓣ                                         |        |
| 53                           | 5                                         | 5       | 85                           | U                                         | U       | 117                          | Ⓤ                                         |        |
| 54                           | 6                                         | 6       | 86                           | V                                         | V       | 118                          | Ⓥ                                         |        |
| 55                           | 7                                         | 7       | 87                           | W                                         | W       | 119                          | Ⓦ                                         |        |
| 56                           | 8                                         | 8       | 88                           | X                                         | X       | 120                          | Ⓧ                                         |        |
| 57                           | 9                                         | 9       | 89                           | Y                                         | Y       | 121                          | Ⓨ                                         |        |
| 58                           | :                                         | :       | 90                           | Z                                         | Z       | 122                          | Ⓩ                                         |        |
| 59                           | ;                                         | ;       | 91                           | Ⓛ                                         | SHIFT-: | 123                          | +                                         |        |
| 60                           | Ⓛ                                         | SHIFT-, | 92                           | £                                         | £       | 124                          | ∴                                         |        |
| 61                           | =                                         | =       | 93                           | Ⓜ                                         | SHIFT-; | 125                          |                                           |        |
| 62                           | Ⓜ                                         | SHIFT-, | 94                           | ↑                                         | ↑       | 126                          | ⊠                                         |        |
| 63                           | ?                                         | SHIFT-/ | 95                           | ←                                         | ←       | 127                          | ⊞                                         |        |

# Appendix A

Table 2: continued

| ASCII Value | Effect               | Key(s)       | Quote Mode Display |
|-------------|----------------------|--------------|--------------------|
| 128         |                      |              | ▬                  |
| 129         | orange foreground    | ⌘-1          | ⌘                  |
| 130         |                      |              | ⌘                  |
| 131         |                      |              | ⌘                  |
| 132         |                      |              | ⌘                  |
| 133         | as programmed        | F1           | ⌘                  |
| 134         | as programmed        | F3           | ⌘                  |
| 135         | as programmed        | F5           | ⌘                  |
| 136         | as programmed        | F7           | ⌘                  |
| 137         | as programmed        | SHIFT-F1     | ⌘                  |
| 138         | as programmed        | SHIFT-F3     | ⌘                  |
| 139         | as programmed        | SHIFT-F5     | ⌘                  |
| 140         | as programmed        | SHIFT-F7     | ⌘                  |
| 141         | start new line       | SHIFT-RETURN | ⌘                  |
| 142         | switch to uppercase  |              | ⌘                  |
| 143         |                      |              | ⌘                  |
| 144         | black foreground     | CTRL-1       | ⌘                  |
| 145         | cursor up            | SHIFT-CRSR↑  | ⌘                  |
| 146         | reverse-video off    | CTRL-0       | ⌘                  |
| 147         | clear screen         | SHIFT-HOME   | ⌘                  |
| 148         | insert character     | SHIFT-DEL    | ⌘                  |
| 149         | brown foreground     | ⌘-2          | ⌘                  |
| 150         | light red foreground | ⌘-3          | ⌘                  |
| 151         | gray 1 foreground    | ⌘-4          | ⌘                  |
| 152         | gray 2 foreground    | ⌘-5          | ⌘                  |
| 153         | lt. green foreground | ⌘-6          | ⌘                  |
| 154         | lt. blue foreground  | ⌘-7          | ⌘                  |
| 155         | gray 3 foreground    | ⌘-8          | ⌘                  |
| 156         | purple foreground    | CTRL-5       | ⌘                  |
| 157         | cursor left          | SHIFT-CRSR←  | ⌘                  |
| 158         | yellow foreground    | CTRL-8       | ⌘                  |
| 159         | cyan foreground      | CTRL-4       | ⌘                  |

# Appendix A

Table 2: continued

| AV<br>S a c t<br>C l t<br>I u e<br>I e r | C<br>h<br>a<br>r<br>a<br>c<br>t<br>e<br>r | Key(s)  | AV<br>S a c t<br>C l t<br>I u e<br>I e r | C<br>h<br>a<br>r<br>a<br>c<br>t<br>e<br>r | Key(s)   | AV<br>S a c t<br>C l t<br>I u e<br>I e r | C<br>h<br>a<br>r<br>a<br>c<br>t<br>e<br>r | Key(s) |
|------------------------------------------|-------------------------------------------|---------|------------------------------------------|-------------------------------------------|----------|------------------------------------------|-------------------------------------------|--------|
|                                          |                                           | SHIFT-  |                                          |                                           |          |                                          |                                           |        |
| 160                                      | ␣                                         | space   | 192                                      | ⇧                                         | SHIFT-*  | 224                                      | ␣                                         |        |
| 161                                      | ⌘                                         | ⌘-K     | 193                                      | ⇧                                         | SHIFT-A  | 225                                      | ␣                                         |        |
| 162                                      | ⌘                                         | ⌘-I     | 194                                      | ⇧                                         | SHIFT-B  | 226                                      | ␣                                         |        |
| 163                                      | ⌘                                         | ⌘-T     | 195                                      | ⇧                                         | SHIFT-C  | 227                                      | ␣                                         |        |
| 164                                      | ⌘                                         | ⌘-@     | 196                                      | ⇧                                         | SHIFT-D  | 228                                      | ␣                                         |        |
| 165                                      | ⌘                                         | ⌘-G     | 197                                      | ⇧                                         | SHIFT-E  | 229                                      | ␣                                         |        |
| 166                                      | ⌘                                         | ⌘-+     | 198                                      | ⇧                                         | SHIFT-F  | 230                                      | ⌘                                         |        |
| 167                                      | ⌘                                         | ⌘-M     | 199                                      | ⇧                                         | SHIFT-G  | 231                                      | ␣                                         |        |
| 168                                      | ⌘                                         | ⌘-£     | 200                                      | ⇧                                         | SHIFT-H  | 232                                      | ⌘                                         |        |
| 169                                      | ⇧                                         | SHIFT-£ | 201                                      | ⇧                                         | SHIFT-I  | 233                                      | ⇧                                         |        |
| 170                                      | ⌘                                         | ⌘-N     | 202                                      | ⇧                                         | SHIFT-J  | 234                                      | ␣                                         |        |
| 171                                      | ⌘                                         | ⌘-Q     | 203                                      | ⇧                                         | SHIFT-K  | 235                                      | ␣                                         |        |
| 172                                      | ⌘                                         | ⌘-D     | 204                                      | ⇧                                         | SHIFT-L  | 236                                      | ␣                                         |        |
| 173                                      | ⌘                                         | ⌘-Z     | 205                                      | ⇧                                         | SHIFT-M  | 237                                      | ␣                                         |        |
| 174                                      | ⌘                                         | ⌘-S     | 206                                      | ⇧                                         | SHIFT-N  | 238                                      | ␣                                         |        |
| 175                                      | ⌘                                         | ⌘-P     | 207                                      | ⇧                                         | SHIFT-O  | 239                                      | ␣                                         |        |
| 176                                      | ⌘                                         | ⌘-A     | 208                                      | ⇧                                         | SHIFT-P  | 240                                      | ␣                                         |        |
| 177                                      | ⌘                                         | ⌘-E     | 209                                      | ⇧                                         | SHIFT-Q  | 241                                      | ␣                                         |        |
| 178                                      | ⌘                                         | ⌘-R     | 210                                      | ⇧                                         | SHIFT-R  | 242                                      | ␣                                         |        |
| 179                                      | ⌘                                         | ⌘-W     | 211                                      | ⇧                                         | SHIFT-S  | 243                                      | ␣                                         |        |
| 180                                      | ⌘                                         | ⌘-H     | 212                                      | ⇧                                         | SHIFT-T  | 244                                      | ␣                                         |        |
| 181                                      | ⌘                                         | ⌘-J     | 213                                      | ⇧                                         | SHIFT-U  | 245                                      | ␣                                         |        |
| 182                                      | ⌘                                         | ⌘-L     | 214                                      | ⇧                                         | SHIFT-V  | 246                                      | ␣                                         |        |
| 183                                      | ⌘                                         | ⌘-Y     | 215                                      | ⇧                                         | SHIFT-W  | 247                                      | ␣                                         |        |
| 184                                      | ⌘                                         | ⌘-U     | 216                                      | ⇧                                         | SHIFT-X  | 248                                      | ␣                                         |        |
| 185                                      | ⌘                                         | ⌘-O     | 217                                      | ⇧                                         | SHIFT-Y  | 249                                      | ␣                                         |        |
| 186                                      | ⇧                                         | SHIFT-@ | 218                                      | ⇧                                         | SHIFT-Z  | 250                                      | ⇧                                         |        |
| 187                                      | ⌘                                         | ⌘-F     | 219                                      | +                                         | SHIFT-+  | 251                                      | ␣                                         |        |
| 188                                      | ⌘                                         | ⌘-C     | 220                                      | ⌘                                         | ⌘- —     | 252                                      | ␣                                         |        |
| 189                                      | ⌘                                         | ⌘-X     | 221                                      | ⇧                                         | SHIFT- — | 253                                      | ␣                                         |        |
| 190                                      | ⌘                                         | ⌘-V     | 222                                      | ⇧                                         | SHIFT-↑  | 254                                      | ␣                                         |        |
| 191                                      | ⌘                                         | ⌘-B     | 223                                      | ⇧                                         | ⌘-*      | 225                                      | ⌘                                         |        |







## Binary Representation of Numbers

Normally, we write integers in their decimal, that is, base 10, representations. For instance, if  $r, s, t, u, v$  are digits from 0 to 9, then

$rstuv$

represents the number

$$r*10000 + s*1000 + t*100 + u*10 + v$$

or  $r*10^4 + s*10^3 + t*10^2 + u*10 + v$

In the binary representation of numbers, the number 2 plays the role of 10. Also, the digits 0 to 9 are reduced to just the digits 0 and 1. In binary notation

$rstuv$

represents the number

$$r*2^4 + s*2^3 + t*2^2 + u*2 + v$$

or  $r*16 + s*8 + t*4 + u*2 + v$

Some binary numbers and their decimal equivalents are:

|          |                                                            |
|----------|------------------------------------------------------------|
| 101      | $5 = (1*4 + 0*2 + 1)$                                      |
| 10010    | $18 = (1*16 + 0*8 + 0*4 + 1*2 + 0)$                        |
| 111100   | $60 = (1*32 + 1*16 + 1*8 + 1*4 + 0*2 + 0)$                 |
| 11010010 | $210 = (1*128 + 1*64 + 0*32 + 1*16 + 0*8 + 0*4 + 1*2 + 0)$ |
| 11111111 | $255 = (1*128 + 1*64 + 1*32 + 1*16 + 1*8 + 1*4 + 1*2 + 1)$ |

Each digit in a binary number is called a *binary digit* or *bit*. Most memory locations in RAM store an 8-bit binary number or 1 *byte*. One byte can represent numbers ranging from 00000000 to 11111111 binary; that is, from 0 to 255 in decimal. (Color RAM, extending from locations 555296 to 56319, can only store 4-bit binary numbers or nybbles. One nybble can represent numbers ranging from 0000 to 1111 binary, or 0 to 15 decimal.)

In working with the PEEK and POKE statements, especially with SPRITE graphics and music synthesis, conversion between the deci-

## Appendix B

mal form and the 8-bit binary form of a number, or vice versa, is often required. The following program converts a decimal number between 0 and 255 to its 8-bit binary form. This program was used as a segment of a larger program to generate the table of binary numbers in Appendix J.

```
10 PRINT "{SHIFT-CLR/HOME}" :REM CLEAR SCREEN
20 INPUT "DECIMAL NUMBER"; A
30 PRINT "THE BINARY FORM IS ";
40 FOR I=7 TO 0 STEP -1
50 IF A AND 2↑I THEN PRINT "1";:GOTO 70
60 PRINT "0";
70 NEXT I
```

Occasionally the individual bits within a byte need to be referenced. For this purpose, the 8 bit positions in a byte are numbered from *right* to *left*. The rightmost bit in a byte is called bit zero. Proceeding to the left, the next bit is called bit one, and so on over to the leftmost bit, which is called bit seven. For example, the binary form of the decimal number 201 is 11001001. As illustrated in Figure 1, bit zero of this byte is a 1, bit one is 0, bit two is 0, and so forth.

|   |   |   |   |   |   |   |   |                       |
|---|---|---|---|---|---|---|---|-----------------------|
| s |   |   |   | t |   |   |   |                       |
| e |   | f | f | h |   |   |   | z                     |
| v | s | i | o | r | t | o | e |                       |
| e | i | v | u | e | w | n | r | bit position          |
| n | x | e | r | e | o | e | o |                       |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | byte representing 201 |

**FIGURE 1**

## Color Codes

The Commodore 64 provides the programmer with 16 choices of color for the screen border, the screen background, the SPRITE multi-colors, each SPRITE, and each character position on the screen. In each case, the color is controlled by POKEing a location given in the first table below with a value chosen from the second table below.

| <u>Description</u>   | <u>Location of color control</u> |
|----------------------|----------------------------------|
| Screen border        | 53280                            |
| Screen background    | 53281                            |
| SPRITE multi-color 0 | 53285                            |
| SPRITE multi-color 1 | 53286                            |
| Color of SPRITE 0    | 53287                            |
| Color of SPRITE 1    | 53288                            |
| Color of SPRITE 2    | 53289                            |
| Color of SPRITE 3    | 53290                            |
| Color of SPRITE 4    | 53291                            |
| Color of SPRITE 5    | 53892                            |
| Color of SPRITE 6    | 53293                            |
| Color of SPRITE 7    | 53294                            |
| Characters on screen | 55296-56295                      |

| <u>Color</u> | <u>Value to POKE</u> | <u>Color</u> | <u>Value to POKE</u> |
|--------------|----------------------|--------------|----------------------|
| black        | 0                    | orange       | 8                    |
| white        | 1                    | brown        | 9                    |
| red          | 2                    | light red    | 10                   |
| cyan         | 3                    | gray 1       | 11                   |
| purple       | 4                    | gray 2       | 12                   |
| green        | 5                    | light green  | 13                   |
| blue         | 6                    | light blue   | 14                   |
| yellow       | 7                    | gray 3       | 15                   |

Note: Only colors 0 thru 7 may be used when working in multi-color mode.

# Appendix D

## Disk Operating System Commands

The VIC-1541 disk drive contains its own microprocessor to control all operations involving disks. To carry out the BASIC commands CLOSE, CMD, GET#, INPUT#, LOAD, OPEN, PRINT#, SAVE, and VERIFY when the disk drive is involved, BASIC automatically communicates with the *Disk Operating System* or *DOS* associated with this microprocessor. In addition, there are commands not contained in BASIC, but understood by the disk drive, that can be issued to DOS. To issue such a command, the command channel to DOS, channel 15, first must be opened with a statement of the form OPEN N,8,15 where N is often chosen to be 15 as well. Then statements of the form PRINT#N, "*command*" cause DOS to execute *command*.

Below are the descriptions of six common commands used when working with disks. In all examples, the assumption is made that the statement OPEN 15,8,15 has been executed. (The 0: required in front of each file name below is a leftover from when one microprocessor was used to control two disk drives numbered 0: and 1:.)

### COPY

The form of the COPY command is

```
COPY 0:newfile=0:oldfile
```

This DOS command will make a second copy of the file named *oldfile*, and name it *newfile*. If the name *newfile* has already been used to name a file, the DOS error light will flash on the disk drive, and no copy will be made. The word COPY may be abbreviated to the letter C.

An extension of the COPY command allows up to four files to be copied and joined into one large file. The form of the COPY command for joining four files is

```
COPY 0:newfile=0:oldfile1,0:oldfile2,0:oldfile3,0:oldfile4
```

For example, to make a copy of the file LETTER and call it LETTER-TOJOHN requires the statement

```
PRINT#15,"COPY 0:LETTERTOJOHN=0:LETTER"
```

or

```
PRINT#15,"C 0:LETTERTOJOHN=0:LETTER"
```

## Appendix D

To copy ACCOUNT1 and ACCOUNT2 into a single file to be named MASTERACCOUNT requires the statement

```
PRINT#15,"C 0:MASTERACCOUNT=0:ACCOUNT1,0:ACCOUNT2"
```

### INITIALIZE

The INITIALIZE command returns the disk drive to the same state as when it is first turned on. This action is sometimes required when an error condition has occurred and DOS fails to execute subsequent commands as expected. The word INITIALIZE may be abbreviated to the letter I. To issue this command requires the statement

```
PRINT#15, "INITIALIZE"
```

or

```
PRINT#15, "I"
```

### NEW

The form of the NEW command is

```
NEW 0:name,id
```

This DOS command must be the first command used on a new disk. It sets up the directory and performs other formatting tasks that are necessary before a new disk can be used. *Name* becomes the name of the disk and appears as part of the first line when the directory is listed. *Name* may be up to 16 letters long. *Id* also appears in the first line of the directory and, in addition, is used to tag each data block on the disk so that DOS can tell if disks are changed in the middle of a series of read or write operations. *Id* can be a single character or a pair of characters. The word NEW may be abbreviated to the letter N.

An old disk, one previously formatted with the NEW command, can be made "new" again by clearing the directory without taking the time to reformat the entire disk. This is accomplished by giving the NEW command in the form

```
NEW 0:name
```

where *id* is omitted.

For example, to format a new disk and call it PROGRAMS with PR as the id requires the statement

```
PRINT#15, "NEW 0:PROGRAMS,PR"
```

## Appendix D

or

```
PRINT#15, "N 0:PROGRAMS,PR"
```

### RENAME

The form of the RENAME command is

```
RENAME 0:newname = oldname
```

Note that 0: is not required in front of the second file name.

This DOS command allows a file on disk to be given a new name. *Oldname* is the name of the file to be renamed as *newname*. *Newname* may be up to 16 letters long. If *newname* has already been used to name a file, the DOS error light will blink, and the file will not be renamed. The word RENAME may be abbreviated to the letter R.

For example, to change the name of the file NEWS to OLDNEWS requires the statement

```
PRINT#15, "RENAME 0:OLDNEWS=NEWS"
```

or

```
PRINT#15, "R 0:OLDNEWS=NEWS"
```

### SCRATCH

The form of the SCRATCH command is

```
SCRATCH 0:filename
```

This DOS command deletes all information about the file called *filename* from the disk. DOS does not bother to erase the contents of the file, but all information about where these contents lie is lost. The space occupied by the contents of *filename* will eventually be reused, destroying the old data. The word SCRATCH may be abbreviated to the letter S.

For example, to discard the file OLDNEWS and make the space available for later files requires the statement

```
PRINT#15, "SCRATCH 0:OLDNEWS"
```

or

```
PRINT#15, "S 0:OLDNEWS"
```



## VALIDATE

The VALIDATE command causes the contents of the disk to be "reordered" so that unused data blocks on the disk are grouped together, allowing DOS to make full use of them in saving new files. After many programs have been SAVED and SCRATCHed from a disk, unused data blocks can be scattered all over and be unaccessible for new files. VALIDATE recovers these scattered blocks. The word VALIDATE may be abbreviated to the letter V. To issue this command requires the statement

```
PRINT#15, "VALIDATE"
```

or

```
PRINT#15, "V"
```

## **HANDLING ERRORS**

**BASIC:** When an error occurs which produces a BASIC error message and program termination, care must be taken to CLOSE all disk files which were OPEN when the error occurred. This must be done before editing or rerunning the program, or the unclosed disk files will become unreadable.

**DOS:** When an error occurs within DOS, producing a blinking red light on the front of the disk drive, the nature of the error can be determined by reading from the command channel. This cannot be done in direct mode since it involves the INPUT# statement. Assuming the first line of your program is 10, you could add the following lines to your program and then type RUN to read the DOS error.

```
1 OPEN 15,8,15
2 INPUT#15, A$,B$,C$,D$
3 PRINT A$,B$,C$,D$
4 CLOSE 15
5 END
```

If your program contains a line numbered between 1 and 5, you can place the above statements, with appropriate line numbers, at the end of your program. In such cases, type RUN *n*, where *n* is the line number of the OPEN statement.

# Appendix E

## Error Messages

|                    |                                                                                                                                                                                        |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BAD SUBSCRIPT      | Use is made of an array name with a subscript whose value is greater than the limit specified in the DIM statement.                                                                    |
| BREAK              | Program execution has halted due to a STOP statement or to the pressing of the RUN/STOP key.                                                                                           |
| CAN'T CONTINUE     | The CONT command cannot be used because a program has not been run since the last occurrence of an error or the editing of a program line.                                             |
| DEVICE NOT PRESENT | One of the statements CMD, PRINT#, INPUT#, or GET# has been executed, and the required device, printer, cassette, or disk drive, is not connected to the computer or is not turned on. |
| DIVISION BY ZERO   | An attempt has been made to divide a quantity by zero.                                                                                                                                 |
| EXTRA IGNORED      | Too many items of data were entered in response to an INPUT statement. The last few items, which were not needed, have been discarded.                                                 |
| FILE DATA          | Program attempted to read a numeric value from a file, but instead found string data as the next item.                                                                                 |
| FILE NOT FOUND     | An end-of-tape marker was read on the cassette before the requested file was found, or the file does not exist on the disk.                                                            |
| FILE NOT OPEN      | The reference number used with a CMD, GET#, INPUT#, or PRINT# statement has not yet been associated with a file or device through an OPEN statement.                                   |
| FILE OPEN          | An attempt was made to execute an OPEN statement using a reference number which was                                                                                                    |

# Appendix E

|                       |                                                                                                                                                                                               |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                       | already associated with a file or device by a previous OPEN statement, and not yet disassociated by a CLOSE statement.                                                                        |
| FORMULA TOO COMPLEX   | The expression being evaluated must be broken down into several simpler expressions in order to be evaluated.                                                                                 |
| ILLEGAL DEVICE NUMBER | The device number used with a LOAD or SAVE statement was 2 or 3; numbers that are not allowed with these two statements.                                                                      |
| ILLEGAL DIRECT        | An attempt has been made to use one of the statements, GET, GET#, INPUT, or INPUT# from direct mode. These statements may be used only in program mode.                                       |
| ILLEGAL QUANTITY      | The number used as the argument of a function or statement is not within the allowed range.                                                                                                   |
| LOAD                  | There is a problem with the program on tape.                                                                                                                                                  |
| MISSING FILE NAME     | The empty string, "", has been given for the name of a file to be LOADED or SAVED on disk.                                                                                                    |
| NEXT WITHOUT FOR      | A NEXT statement has been encountered which does not match up with a previous FOR statement. The index in the NEXT statement may be in error, or several FOR loops may be incorrectly nested. |
| NOT INPUT FILE        | An attempt was made to INPUT or GET data from a file or device which has been OPENED for output only.                                                                                         |
| NOT OUTPUT FILE       | An attempt was made to PRINT data to a file or device which has been OPENED for input only.                                                                                                   |
| OUT OF DATA           | All items in DATA statements have been used, leaving no data for the current READ statement.                                                                                                  |
| OUT OF MEMORY         | No more memory is available for program lines or variables required by a program. This error also may occur when BASIC's stack is over-                                                       |

# Appendix E

flowed by too many nested FOR . . . NEXT loops, too many GOSUBs or too many nested parentheses in an expression.

|                      |                                                                                                                                                                                                                                                                                                                                |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| OVERFLOW             | The magnitude (absolute value) of the result of a computation exceeds the largest number which the computer can record, which is 1.70141884E + 38.                                                                                                                                                                             |
| REDIM'D ARRAY        | An attempt has been made to execute a DIM statement for an array which has been DIMensioned previously. If an element of an array is referenced before the array is explicitly DIMensioned, BASIC automatically DIMensions the array to a size of 11 (subscripts 0 to 10), and thus such an array cannot later be DIMensioned. |
| REDO FROM START      | Non-numeric (string) data was entered in response to an INPUT statement which was requesting numeric data. Program execution will continue when numeric data is entered.                                                                                                                                                       |
| RETURN WITHOUT GOSUB | A RETURN statement has been encountered, but no GOSUB statement is in effect.                                                                                                                                                                                                                                                  |
| STRING TOO LONG      | A string being read from cassette or disk exceeds 88 characters, or a string variable has been assigned a value of more than 255 characters.                                                                                                                                                                                   |
| SYNTAX               | A statement or command has been mistyped or otherwise formed incorrectly, and BASIC cannot recognize its meaning. Check for misspelled reserved words, reserved words included in variable names, missing or extra parentheses, etc.                                                                                           |
| TOO MANY FILES       | An attempt has been made to OPEN more than 10 files or devices. A maximum of 10 different reference numbers may be in use at any one time.                                                                                                                                                                                     |
| TYPE MISMATCH        | A number has been used where a string is expected, or vice versa.                                                                                                                                                                                                                                                              |

## Appendix E

UNDEF'D  
FUNCTION

A user-defined function has been referenced, but it has not yet been defined in a DEF FN statement.

UNDEF'D  
STATEMENT

An attempt was made to GOTO, GOSUB, or RUN a line number that does not exist.

VERIFY

The program on cassette or disk does not match the program currently in memory.

# Appendix F

## Data Files

As a model for a sequential data file, consider a roll of adding machine tape. Suppose that we recorded names and phone numbers of friends on this tape with indelible ink and just entered them one after another with no blank lines between entries and no attempt to alphabetize. Our list of data might look something like this:

```
LEONARD SLYE, 123-4567
HENRY MCCARTY, 987-6543
EHRICH WEISS, 345-9876
CLAUDE DUKENFIELD, 832-3287
```

In order to search for a name we would have to begin reading from the top of the list. We could not delete, change, or add an entry in the list unless we recopied the entire list and made the correction or addition when we came to it.

Our telephone directory could be placed into a sequential file on a cassette or disk. The names and phone numbers would not be listed in a column on the cassette or disk, but would be arranged linearly. One possible arrangement is as follows:

```
LEONARD SLYE<R>123-4567<R>HENRY MCCARTY<R>987-
6543<R> . . .
```

Here, the character RETURN, denoted by <R>, is used to separate data items, and is referred to as a delimiter. RETURN is the only delimiter recognized by the INPUT# statement, which is often used to read data back from a file.

When reading a sequential file, the computer always begins with the first item and reads the pieces of information in order, one at a time, until it finds the item being searched for. To delete, change or add a piece of information to the file, the entire file is recopied, making changes where appropriate.

Sequential data files are created on cassette with statements of the form OPEN N,1,1,"*filename*" where N is a number (called the file reference number; usually 1, 2, or 3) that will temporarily identify the file. On disk, sequential data files are created with statements of the form OPEN N,8,C,"*filename,S,W*". Here, N is again a number to temporarily identify the file, and C is a number from 2 to 14 (called the channel number) that is generally the same as N. Information is entered into sequential files with the statement PRINT#. After we have finished placing information into the file, we execute the statement

## Appendix F

CLOSE N . For instance, the following program will place the first three names and phone numbers above into a file on cassette.

```
10 OPEN 1,1,1,"PHONE"
20 PRINT#1, "LEONARD SLYE"
30 PRINT#1, "123-4567"
40 PRINT#1, "HENRY MCCARTY"
50 PRINT#1, "987-6543"
60 PRINT#1, "EHRICH WEISS"
70 PRINT#1, "345-9876"
80 CLOSE 1
```

Here the file is named "PHONE", and the file is temporarily referred to as file #1.

In the above program, each item was placed in a separate PRINT# statement so that the RETURN character would follow each item in the file. This is necessary if the INPUT# statement is to be used to read the information back from the file. The following program places the same information into a disk file, but rather than using separate PRINT# statements for each item, the program assigns the RETURN character to a string variable and uses this variable to explicitly place the RETURN character after each item.

```
10 OPEN 2,8,2,"PHONE,S,W"
20 R$=CHR$(13) :REM DELIMITER IS <R>
30 PRINT#2,"LEONARD SLYE";R$;"123-4567";R$;
40 PRINT#2,"HENRY MCCARTY";R$;"987-6543";R$;
50 PRINT#2,"EHRICH WEISS";R$;"345-9876";R$;
60 CLOSE 2
```

When control characters, such as "cursor up", "reverse on", and "red foreground", are written to a file, they do not have any special effect, but are simply recorded as if in quote mode. For example, the following program places 10 control characters in the disk file SPECIAL.

```
10 OPEN 3,8,3,"SPECIAL,S,W"
20 FOR I=10 TO 19
30 PRINT#3, CHR$(I);
40 NEXT I
50 CLOSE 3
```

After this program is run, the file SPECIAL will have the following contents:

**JKLMNOPQRS**

# Appendix F

In order to read a sequential file we first must gain access to it with a statement of the form `OPEN N,1,0,"filename"` for a cassette file, or `OPEN N,8,C,"filename,S,R"` for a disk file. Then we can copy information from the file with the statements `INPUT#` and `GET#`. `INPUT#` is used to retrieve whole items that have been entered with the `RETURN` character as a delimiter, while `GET#` is used to retrieve data one character at a time, including quotation marks and any delimiters. The following program can be used to search the file `PHONE` for a telephone number.

```
10 INPUT "NAME"; N$
20 OPEN 2,8,2,"PHONE,S,R"
30 INPUT#2, A$, B$
40 IF A$ = N$ THEN GO
50 GOTO 30
60 PRINT B$
70 CLOSE 2
RUN
NAME? HENRY MCCARTY
987-6543
```

READY.

While using a file, we can check if the end of the file has been reached by using the `STATUS` function. When an attempt is made to read an item from a file in which no more items remain, the expression `(STATUS AND 64) = 64` will be true. Thus, after an `INPUT#` or `GET#` statement is executed, a statement of the form

```
IF (STATUS AND 64) = 64 THEN end of file action
```

is often appropriate.



## Generator Program for SPRITEs

You may have noted that it is quite tedious to generate the data necessary to describe a SPRITE to the computer. It is really quite senseless not to make use of the computer to do some of the work. Here is a program that lets you design a SPRITE right on the screen and then computes the data items to be included within your program.

The program displays the "+" character at each of the 24 by 21 dot locations that can form a SPRITE. Initially, the cursor, shown as a "?", is positioned in the upper left corner of the SPRITE, and all dots of the SPRITE are uncolored. You color a dot by using the RETURN and cursor control keys to move the cursor to the desired dot and then pressing the space bar. If the cursor is positioned at a dot which is already colored and you press the space bar, you will uncolor the dot. You may move the cursor freely about the SPRITE, coloring or uncoloring dots, until you are satisfied with your creation. In order to quit and see the SPRITE data, type "Q". You may then read off the numbers to be included in the data statements defining the SPRITE.

Here is the SPRITE generator program.

```
2 POKE 53280,14 :REM SCREEN BORDER LIGHT BLUE
4 POKE 53281,6 :REM SCREEN DARK BLUE
6 PRINT CHR$(154) :REM CHARACTERS LIGHT BLUE
8 DIM PIC(21,24) :REM SPRITE PIXELS
10 U$=CHR$(145) :REM CURSOR UP
12 D$=CHR$(17) :REM CURSOR DOWN
14 L$=CHR$(157) :REM CURSOR LEFT
16 R$=CHR$(29) :REM CURSOR RIGHT
18 H$=CHR$(19) :REM HOME CURSOR
20 RV$=CHR$(18) :REM INVERSE VIDEO ON
22 N$=CHR$(146) :REM INVERSE VIDEO OFF
24 PRINT CHR$(147); :REM CLEAR SCREEN
26 FOR I=1 TO 21 :REM DISPLAY PIXEL POSITIONS
28 FOR J=1 TO 24
30 PRINT "+";
32 NEXT J
34 PRINT
36 NEXT I: PRINT
38 PRINT "CURSOR CONTROLS MOVE ? MARKER."
40 PRINT "SPACE BAR COLORS OR UNCOLORS SPOT."
42 PRINT "TYPE Q TO END AND SEE SPRITE DATA.";
44 PRINT H$;"?";L$; :REM CURSOR TO ROW 1 COLUMN 1
46 R=1:C=1 :REM ROW AND COLUMN RECORD
48 GET A$: IF A$="" THEN 48 :REM INPUT A COMMAND
50 IF A$="Q" THEN 900 :REM Q MEANS QUIT, SHOW DATA
52 IF A$=U$ THEN GOSUB 100:GOTO 48
54 IF A$=D$ THEN GOSUB 200: GOTO 48
56 IF A$=L$ THEN GOSUB 300:GOTO 48
58 IF A$=R$ THEN GOSUB 400: GOTO 48
60 IF A$=" " THEN GOSUB 500:GOTO 48
62 IF A$=CHR$(13) THEN GOSUB 600
```

# Appendix G

```
64 GOTO 48
97 REM
98 REM MOVE CURSOR UP, IF POSSIBLE
99 REM
100 IF R=1 THEN RETURN
102 GOSUB 700: PRINT U$
104 R=R-1: GOSUB 800: RETURN
197 REM
198 REM MOVE CURSOR DOWN, IF POSSIBLE
199 REM
200 IF R=21 THEN RETURN
202 GOSUB 700: PRINT D$;
204 R=R+1: GOSUB 800: RETURN
297 REM
298 REM MOVE CURSOR LEFT, IF POSSIBLE
299 REM
300 IF C=1 THEN RETURN
302 GOSUB 700: PRINT L$;
304 C=C-1: GOSUB 800: RETURN
397 REM
398 REM MOVE CURSOR RIGHT, IF POSSIBLE
399 REM
400 IF C=24 THEN RETURN
402 GOSUB 700: PRINT R$;
404 C=C+1: GOSUB 800: RETURN
496 REM
497 REM COLOR OR UNCOLOR CURSOR POSITION
498 REM AND MOVE CURSOR RIGHT, IF POSSIBLE
499 REM
500 IF PIC(R,C) THEN PIC(R,C)=0: GOTO 504
502 PIC(R,C)=1
504 IF C=24 THEN GOSUB 800: RETURN
506 GOSUB 700: PRINT R$;
508 C=C+1: GOSUB 800: RETURN
596 REM
597 REM MOVE CURSOR TO BEGINNING OF NEXT
598 REM LINE, IF AT BOTTOM, GOTO TOP
599 REM
600 GOSUB 800: IF R=21 THEN 604
602 PRINT: R=R+1: C=1: GOSUB 800: RETURN
604 PRINT H$;: R=1: C=1: GOSUB 800: RETURN
696 REM
697 REM RECOLOR PIXEL TO CONDITION BEFORE
698 REM CURSOR WAS POSITIONED THERE
699 REM
700 IF PIC(R,C) THEN PRINT RV$;" ";N$;L$;:RETURN
702 PRINT "+";L$;:RETURN
796 REM
797 REM DISPLAY CURSOR ACCORDING TO
798 REM CURRENT COLOR OF PIXEL
799 REM
800 IF PIC(R,C) THEN PRINT RV$;"?";N$;L$;:RETURN
802 PRINT "?";L$;:RETURN
897 REM
898 REM COMPUTE AND DISPLAY DATA
899 REM
900 IF PIC(R,C) THEN PRINT RV$;" ";N$;H$;:GOTO 904
```

## Appendix G

```
902 PRINT "+";H$;
904 FOR I=1 TO 21
906 : PRINT SPC(24);
908 : FOR J=0 TO 2
910 : S=0
912 : FOR K=1 TO 8
914 : S=S*2+PIC(I,J*8+K)
916 : NEXT K
918 : IF S<100 THEN PRINT " ";
920 : IF S<10 THEN PRINT " ";
922 : PRINT S;
924 : NEXT J
926 : PRINT
928 NEXT I
```

# Appendix H

## Memory Allocation in the Commodore 64

| Decimal Location | Description                                                                                      |
|------------------|--------------------------------------------------------------------------------------------------|
| 0 - 1023         | 1K RAM used as data storage area by BASIC and Kernal                                             |
| 1024 - 2047      | 1K RAM for Screen Memory (character codes) and SPRITE data pointers                              |
| 2048 - 40959     | 38K RAM for writing and running BASIC programs                                                   |
| 40960 - 49151    | 8K ROM BASIC Interpreter<br>or<br>8K RAM (POKE 1,PEEK(1) OR 1)                                   |
| 49152 - 53247    | 4K RAM unused (good for user-defined machine language subroutines)                               |
| 53248 - 57343    | I/O devices and Screen Color RAM<br>or<br>4K Character ROM (POKE 1,PEEK(1) OR 4)<br>or<br>4K RAM |
| 53248 - 54271    | VIC chip registers (SPRITEs and video control)                                                   |
| 54272 - 55295    | SID chip registers (music synthesizer)                                                           |
| 55296 - 56319    | 1K RAM for Screen Memory (color nybbles)                                                         |
| 56320 - 57343    | CIA chip registers and future I/O expansion                                                      |
| 57344 - 65535    | 8K ROM Kernal (Commodore 64 operating system)<br>or<br>8K RAM (POKE 1,PEEK(1) OR 2)              |

# Appendix H

## Select Memory Locations of Interest

---

|           |                                           |
|-----------|-------------------------------------------|
| 1         | RAM versus ROM and registers control byte |
| 97 - 101  | floating-point accumulator used by USR    |
| 144       | I/O status, returned by STATUS            |
| 211       | cursor location (column)                  |
| 214       | cursor location (row)                     |
| 631 - 630 | keyboard buffer                           |
| 785 - 786 | address of subroutine for USR execution   |

# Appendix I

## Reserved Words

The following words serve as BASIC commands, statements, operators, or functions, and may not be used as any part of variable names created by the programmer.

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

Note: Reserved words preceded by an asterisk may not begin a variable name created by the programmer, but may be used within one.

## 0 to 255 in Decimal and Binary

|    |          |    |          |     |          |
|----|----------|----|----------|-----|----------|
| 0  | 00000000 | 48 | 00110000 | 96  | 01100000 |
| 1  | 00000001 | 49 | 00110001 | 97  | 01100001 |
| 2  | 00000010 | 50 | 00110010 | 98  | 01100010 |
| 3  | 00000011 | 51 | 00110011 | 99  | 01100011 |
| 4  | 00000100 | 52 | 00110100 | 100 | 01100100 |
| 5  | 00000101 | 53 | 00110101 | 101 | 01100101 |
| 6  | 00000110 | 54 | 00110110 | 102 | 01100110 |
| 7  | 00000111 | 55 | 00110111 | 103 | 01100111 |
| 8  | 00001000 | 56 | 00111000 | 104 | 01101000 |
| 9  | 00001001 | 57 | 00111001 | 105 | 01101001 |
| 10 | 00001010 | 58 | 00111010 | 106 | 01101010 |
| 11 | 00001011 | 59 | 00111011 | 107 | 01101011 |
| 12 | 00001100 | 60 | 00111100 | 108 | 01101100 |
| 13 | 00001101 | 61 | 00111101 | 109 | 01101101 |
| 14 | 00001110 | 62 | 00111110 | 110 | 01101110 |
| 15 | 00001111 | 63 | 00111111 | 111 | 01101111 |
| 16 | 00010000 | 64 | 01000000 | 112 | 01110000 |
| 17 | 00010001 | 65 | 01000001 | 113 | 01110001 |
| 18 | 00010010 | 66 | 01000010 | 114 | 01110010 |
| 19 | 00010011 | 67 | 01000011 | 115 | 01110011 |
| 20 | 00010100 | 68 | 01000100 | 116 | 01110100 |
| 21 | 00010101 | 69 | 01000101 | 117 | 01110101 |
| 22 | 00010110 | 70 | 01000110 | 118 | 01110110 |
| 23 | 00010111 | 71 | 01000111 | 119 | 01110111 |
| 24 | 00011000 | 72 | 01001000 | 120 | 01111000 |
| 25 | 00011001 | 73 | 01001001 | 121 | 01111001 |
| 26 | 00011010 | 74 | 01001010 | 122 | 01111010 |
| 27 | 00011011 | 75 | 01001011 | 123 | 01111011 |
| 28 | 00011100 | 76 | 01001100 | 124 | 01111100 |
| 29 | 00011101 | 77 | 01001101 | 125 | 01111101 |
| 30 | 00011110 | 78 | 01001110 | 126 | 01111110 |
| 31 | 00011111 | 79 | 01001111 | 127 | 01111111 |
| 32 | 00100000 | 80 | 01010000 |     |          |
| 33 | 00100001 | 81 | 01010001 |     |          |
| 34 | 00100010 | 82 | 01010010 |     |          |
| 35 | 00100011 | 83 | 01010011 |     |          |
| 36 | 00100100 | 84 | 01010100 |     |          |
| 37 | 00100101 | 85 | 01010101 |     |          |
| 38 | 00100110 | 86 | 01010110 |     |          |
| 39 | 00100111 | 87 | 01010111 |     |          |
| 40 | 00101000 | 88 | 01011000 |     |          |
| 41 | 00101001 | 89 | 01011001 |     |          |
| 42 | 00101010 | 90 | 01011010 |     |          |
| 43 | 00101011 | 91 | 01011011 |     |          |
| 44 | 00101100 | 92 | 01011100 |     |          |
| 45 | 00101101 | 93 | 01011101 |     |          |
| 46 | 00101110 | 94 | 01011110 |     |          |
| 47 | 00101111 | 95 | 01011111 |     |          |

## Appendix J

|     |          |     |          |     |          |
|-----|----------|-----|----------|-----|----------|
| 128 | 10000000 | 176 | 10110000 | 224 | 11100000 |
| 129 | 10000001 | 177 | 10110001 | 225 | 11100001 |
| 130 | 10000010 | 178 | 10110010 | 226 | 11100010 |
| 131 | 10000011 | 179 | 10110011 | 227 | 11100011 |
| 132 | 10000100 | 180 | 10110100 | 228 | 11100100 |
| 133 | 10000101 | 181 | 10110101 | 229 | 11100101 |
| 134 | 10000110 | 182 | 10110110 | 230 | 11100110 |
| 135 | 10000111 | 183 | 10110111 | 231 | 11100111 |
| 136 | 10001000 | 184 | 10111000 | 232 | 11101000 |
| 137 | 10001001 | 185 | 10111001 | 233 | 11101001 |
| 138 | 10001010 | 186 | 10111010 | 234 | 11101010 |
| 139 | 10001011 | 187 | 10111011 | 235 | 11101011 |
| 140 | 10001100 | 188 | 10111100 | 236 | 11101100 |
| 141 | 10001101 | 189 | 10111101 | 237 | 11101101 |
| 142 | 10001110 | 190 | 10111110 | 238 | 11101110 |
| 143 | 10001111 | 191 | 10111111 | 239 | 11101111 |
| 144 | 10010000 | 192 | 11000000 | 240 | 11110000 |
| 145 | 10010001 | 193 | 11000001 | 241 | 11110001 |
| 146 | 10010010 | 194 | 11000010 | 242 | 11110010 |
| 147 | 10010011 | 195 | 11000011 | 243 | 11110011 |
| 148 | 10010100 | 196 | 11000100 | 244 | 11110100 |
| 149 | 10010101 | 197 | 11000101 | 245 | 11110101 |
| 150 | 10010110 | 198 | 11000110 | 246 | 11110110 |
| 151 | 10010111 | 199 | 11000111 | 247 | 11110111 |
| 152 | 10011000 | 200 | 11001000 | 248 | 11111000 |
| 153 | 10011001 | 201 | 11001001 | 249 | 11111001 |
| 154 | 10011010 | 202 | 11001010 | 250 | 11111010 |
| 155 | 10011011 | 203 | 11001011 | 251 | 11111011 |
| 156 | 10011100 | 204 | 11001100 | 252 | 11111100 |
| 157 | 10011101 | 205 | 11001101 | 253 | 11111101 |
| 158 | 10011110 | 206 | 11001110 | 254 | 11111110 |
| 159 | 10011111 | 207 | 11001111 | 255 | 11111111 |
| 160 | 10100000 | 208 | 11010000 |     |          |
| 161 | 10100001 | 209 | 11010001 |     |          |
| 162 | 10100010 | 210 | 11010010 |     |          |
| 163 | 10100011 | 211 | 11010011 |     |          |
| 164 | 10100100 | 212 | 11010100 |     |          |
| 165 | 10100101 | 213 | 11010101 |     |          |
| 166 | 10100110 | 214 | 11010110 |     |          |
| 167 | 10100111 | 215 | 11010111 |     |          |
| 168 | 10101000 | 216 | 11011000 |     |          |
| 169 | 10101001 | 217 | 11011001 |     |          |
| 170 | 10101010 | 218 | 11011010 |     |          |
| 171 | 10101011 | 219 | 11011011 |     |          |
| 172 | 10101100 | 220 | 11011100 |     |          |
| 173 | 10101101 | 221 | 11011101 |     |          |
| 174 | 10101110 | 222 | 11011110 |     |          |
| 175 | 10101111 | 223 | 11011111 |     |          |



## Printer Control Characters

Below is a list of all of those characters, given in terms of CHR\$, which control special effects on the VIC-1525 Graphics Printer, along with a description of these effects.

| Control Character | Description of effect                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CHR\$(8)          | <p>Switches to <i>graphics mode</i>. The ASCII/Commodore value of each subsequent character is used to print a single vertical line of dots. This mode is cancelled by CHR\$(15).</p> <p>In each vertical line of dots, there are 7 positions in which dots can be printed. Think of the 7 positions as bits in a binary number, with bit 0 the top dot and bit 6 the bottom dot of the vertical line. To print a specific pattern of dots, number the 7 positions by powers of 2, i.e., 1, 2, 4, 8, 16, 32, 64, starting with 1 at the top. For each position which is to have a dot printed, write down its power of two. Add up the numbers written down and then add 128 to this total. By printing CHR\$ of this total, the desired vertical line of dots will be printed. For example, if the printer has been accessed by the statement OPEN 1,4 , then the statement PRINT#1, CHR\$(8); CHR\$(255); CHR\$(193); CHR\$(193) CHR\$(193); CHR\$(193); CHR\$(255) prints 6 vertical lines of dots which form a little square.</p> <p>Line spacing also changes to 9 lines per inch in graphics mode. Thus, there is no space between characters on consecutive lines.</p> |
| CHR\$(10)         | Causes a new line to be started. Same effect as CHR\$(13). On some other printers, CHR\$(10) causes a line feed but does not cause the print head to return to the left edge of the paper.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| CHR\$(13)         | Causes a new line to be started. On some other printers, CHR\$(13) causes a carriage return to the left edge of the paper but does not cause the print head to advance to a new line.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| CHR\$(14)         | Causes subsequent characters to print in <i>double-width</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

# Appendix K

| Control Character | Description of effect                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                   | mode. Double-width characters are twice as wide as standard characters. Thus only 40 double-width characters will print on a line. CHR\$(14) is cancelled by CHR\$(15).                                                                                                                                                                                                                                                                                                                        |
| CHR\$(15)         | Cancels graphics and double-width character modes. Causes subsequent characters to be printed in standard form.                                                                                                                                                                                                                                                                                                                                                                                |
| CHR\$(16)         | Causes the printer to tab to the column given by the next two characters, which therefore must be digits. For the purpose of this control code, a line on the printer has 80 positions numbered 0 to 79, even in double-width mode. For example, if the printer has been accessed by the statement OPEN 1,4 , then the statement PRINT#1,CHR\$(16); "05ABC"; CHR\$(14); CHR\$(16); "50DEF" will print ABC beginning in column 5, and DEF beginning in column 50 using double-width characters. |
| CHR\$(17)         | Switches to the Lowercase/Uppercase character set for subsequent characters. Thus, even if the printer was accessed to print in the Uppercase/Graphics character set by a statement of the form OPEN N,4 or OPEN N,4,0 this control code allows the character set in use to be altered to Lowercase/Uppercase. CHR\$(145) reverses the effect of CHR\$(17), putting the printer into the Uppercase/Graphics character set.                                                                     |
| CHR\$(18)         | Causes subsequent characters on the current line to be printed in reverse image mode. This control code is cancelled by CHR\$(146) or whenever a new line is started by CHR\$(13) or CHR\$(10). (Recall that PRINT statements which do not end with a semicolon or comma automatically print CHR\$(13).)                                                                                                                                                                                       |
| CHR\$(26)         | Used in graphics mode to print the same vertical line of dots several times. See the discussion of CHR\$(8). If graphics mode is in effect, then the sequence CHR\$(26);CHR\$(N);CHR\$(P) causes the vertical line of dots represented by CHR\$(P) to be printed N times. For example, if the printer has been accessed by the statement OPEN 1,4 , then the statement                                                                                                                         |

# Appendix K

| Control Character       | Description of effect                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                         | <code>PRINT#1,CHR\$(8); CHR\$(26); CHR\$(96); CHR\$(255)</code> will print 96 solid vertical lines, creating a horizontal bar 2 inches long. (There are 48 vertical line positions per inch.) The value of N may not exceed 255; thus a maximum of 255 repetitions of a vertical line may be specified at one time.                                                                                                                                                                                                                                                                                                                                    |
| <code>CHR\$(27)</code>  | Used before <code>CHR\$(16)</code> to cause tabbing to a specific vertical line position rather than to a character position. The printer recognizes 480 vertical line positions numbered 0 to 479 from left to right across the standard print line. Whether in standard character or graphics mode, a sequence of the form <code>CHR\$(27); CHR\$(16); CHR\$(H); CHR\$(L)</code> positions the printer to begin printing the next character or vertical line of dots at vertical line position number $256 * H + L$ . The value of L may range from 0 to 255, while H may be 0 or 1. The resulting value of $256 * H + L$ must be between 0 and 479. |
| <code>CHR\$(145)</code> | Switches to the Uppercase/Graphics character set for subsequent characters. Thus, even if the printer was accessed to print in the Lowercase/Uppercase character set by a statement of the form <code>OPEN N,4,7</code> this control code allows the character set in use to be altered to the Uppercase/Graphics set. <code>CHR\$(17)</code> reverses the effect of <code>CHR\$(145)</code> , putting the printer into the Lowercase/Uppercase character set.                                                                                                                                                                                         |
| <code>CHR\$(146)</code> | Cancels reverse image character mode invoked by <code>CHR\$(18)</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

The following table summarizes these control codes.

|                        |                                                                  |
|------------------------|------------------------------------------------------------------|
| <code>CHR\$(8)</code>  | Switch to graphics mode.                                         |
| <code>CHR\$(10)</code> | RETURN on VIC-1525 printer.<br>Line feed on some printers.       |
| <code>CHR\$(13)</code> | RETURN on VIC-1525 printer.<br>Carriage return on some printers. |
| <code>CHR\$(14)</code> | Switch to double-width mode.                                     |

# Appendix K

| Control Character | Description of effect                                           |
|-------------------|-----------------------------------------------------------------|
| CHR\$(15)         | Cancel graphics or double-width mode.                           |
| CHR\$(16);“ab”;   | Tab to column ab, ab = 00 to 79.                                |
| CHR\$(17)         | Switch to Lowercase/Uppercase set.                              |
| CHR\$(18)         | Print reverse-image characters.                                 |
| CHR\$(26)         | Repeat graphics dot pattern.                                    |
| CHR\$(27)         | Used with CHR\$(16) to tab to specific line position, 0 to 479. |
| CHR\$(145)        | Switch to Uppercase/Graphics set.                               |
| CHR\$(146)        | Cancel reverse image printing.                                  |

## Logical Operators

The three logical operators NOT, AND, and OR are used primarily in two ways. In Part I we explore their role in building compound conditions for IF statements. In Part II we show how these three operators along with “xor” operate on bytes; that is, on the 8-bit binary form of numbers stored in each memory location.

### PART I CONDITIONS FOR IF STATEMENTS

Some common types of conditions involve the following relationships between numbers or strings.

| <u>Relationship</u> | <u>Numbers</u>              | <u>Strings</u>                             |
|---------------------|-----------------------------|--------------------------------------------|
| =                   | is equal to                 | is identical to                            |
| <                   | is less than                | precedes alphabetically                    |
| >                   | is greater than             | follows alphabetically                     |
| <>                  | is not equal to             | is not identical to                        |
| <=                  | is less than or equal to    | precedes alphabetically or is identical to |
| >=                  | is greater than or equal to | follows alphabetically or is identical to  |

Conditions involving relationships between numbers or strings are either true (T) or false (F). Some examples of simple conditions, along with their truth values are

| <u>Condition</u>              | <u>Truth value</u> | <u>Condition</u> | <u>Truth value</u> |
|-------------------------------|--------------------|------------------|--------------------|
| $2 < 3$                       | T                  | “Y” <> “X”       | T                  |
| $2 \uparrow 3 = 3 \uparrow 2$ | F                  | “A” >= “B”       | F                  |

Compound conditions can be formed by negating and/or combining simple conditions with the three logical operators. The truth value of a compound condition can be determined by the truth values of the component simple conditions and the logical operators used. In the following table, the conditions *cond1* and *cond2* are used to form compound conditions. For each of these compound conditions, the table specifies when the compound condition will have the truth value T.

| <u>Compound Condition</u>     | <u>Requirements for a Truth Value of T</u>                    |
|-------------------------------|---------------------------------------------------------------|
| NOT <i>cond1</i>              | true only if <i>cond1</i> is false                            |
| <i>cond1</i> AND <i>cond2</i> | true if both <i>cond1</i> and <i>cond2</i> are true           |
| <i>cond1</i> OR <i>cond2</i>  | true if one or both of <i>cond1</i> and <i>cond2</i> are true |

# Appendix L

Some examples of compound conditions, along with their truth values, follow:

|                     |   |
|---------------------|---|
| NOT ("A" >= "B")    | T |
| 2 < 3 AND 2↑3 = 3↑2 | F |
| 2 < 3 OR 2↑3 = 3↑2  | T |

Compound conditions can involve more than one logical operator. If so, the order in which the operations are executed is first NOT, then AND, and finally OR. For instance, the condition NOT ("A" >= "B") OR (2 < 3) is the same as the condition (NOT ("A" >= "B")) OR (2 < 3) and hence has the truth value T.

BASIC assigns the number -1 to true conditions and the number 0 to false conditions. The following examples illustrate this feature.

## Examples

```
1. PRINT 2<3, "B"<"A"
 -1 0
```

READY.

```
2. 10 INPUT "LETTER"; L$
 20 ON (L$<"N")+2 GOSUB 40, 60
 30 END
 40 PRINT "THIS LETTER IS IN THE FIRST HA
 LF OF THE ALPHABET."
 50 RETURN
 60 PRINT "THIS LETTER IS IN THE SECOND H
 ALF OF THE ALPHABET."
 70 RETURN
 RUN
 LETTER? Q
 THIS LETTER IS IN THE SECOND HALF OF THE
 ALPHABET.
```

READY.

In this instance, the value of L\$ was "Q" and so the false condition L\$<"N" received the value 0. Hence the value of (L\$<"N")+2 was 2 and therefore line 20 branched to the second subroutine.

## PART II OPERATIONS ON BYTES

Most memory locations in the Commodore 64 store one byte of information; that is, they store 8 bits corresponding to an integer from 0 to

# Appendix L

255. The discussion below shows how the logical operators AND, OR, NOT, and "xor" are used to alter the value stored in a memory location. The operator "xor" is not available in BASIC, but since it is involved with the WAIT statement, it is included in this discussion. We assume that the reader is familiar with the binary representation of numbers. (See Appendix B for details.)

For the moment, let's consider just the two integers 0 and 1. The following tables give the definitions of the logical operators AND, OR, NOT, and "xor" for these 2 values.

| AND | 0 | 1 |
|-----|---|---|
| 0   | 0 | 0 |
| 1   | 0 | 1 |

| OR | 0 | 1 |
|----|---|---|
| 0  | 0 | 1 |
| 1  | 1 | 1 |

| NOT |   |
|-----|---|
| 0   | 1 |
| 1   | 0 |

| "xor" | 0 | 1 |
|-------|---|---|
| 0     | 0 | 1 |
| 1     | 1 | 0 |

These definitions make sense if we think of 0 as false and 1 as true. Note that NOT requires only a single quantity, while the other three operators require two quantities in order to produce a result.

The logical operators above can be extended to eight-tuples of zeros and ones by operating on corresponding entries of the eight-tuples individually. For instance, 01000111 OR 10010011 is 11010111 and 01101010 "xor" 11010100 is 1011110.

A common application of logical operators on bytes occurs in understanding statements of the form POKE A, PEEK(A) *op* B, where *op* is either AND or OR. PEEK(A) will be an integer from 0 to 255, as should B. To determine the result of PEEK(A) *op* B, first represent each of the two values in binary notation as an eight-tuple of zeros and ones. (After obtaining the binary representation of the numbers, just append some zeros to the left, if necessary, to obtain eight-tuples.) Then apply the logical operators to the eight-tuples. The resulting eight-tuple can be converted back to an integer, though quite often the individual bits of an eight-tuple are used as "on/off switches", and thus the result of POKEing this value into memory is easiest to interpret when the value is left as an eight-tuple.

## Examples

```
1. PRINT 135 AND 11
 3
```

```
READY.
```

# Appendix L

The numbers 135 and 11 correspond to the binary eight-tuples 10000111 and 00001011. Now, 10000111 AND 00001011 is 00000011 which is the binary representation of 3.

2. The following program allows the user to specify two “byte-sized” values and one of the operations AND, OR, or XOR. The program then performs the operation on the values and states the complete problem and answer in both decimal and eight-tuple form.

```
10 PRINT "{SHIFT-CLR/HOME}" :REM CLEAR SCREEN
20 INPUT "FIRST NUMBER (0 TO 255)";F
30 INPUT "SECOND NUMBER (0 TO 255)";S
40 INPUT "OPERATOR: AND, OR, XOR";L$
50 IF L$="AND" THEN A=F AND S: GOTO 100
60 IF L$="OR" THEN A=F OR S: GOTO 100
70 IF L$="XOR" THEN A=(F OR S) AND
NOT (F AND S): GOTO 100
80 GOTO 40
100 PRINT
110 C=F: GOSUB 200: PRINT F, ,B$
120 PRINT " ";L$, ,L$
130 C=S: GOSUB 200: PRINT S, ,B$
140 PRINT " IS", , "IS"
150 C=A: GOSUB 200: PRINT A, ,B$
160 END
200 B$=""
210 FOR I=7 TO 0 STEP -1
220 IF C AND 2^I THEN B#=B$+"1": GOTO 240
230 B#=B$+"0"
240 NEXT I
250 RETURN
```

3. The byte in location 54276 controls several attributes of voice 1 in the music synthesizing chip. In particular, the rightmost bit of this byte controls whether or not voice 1 is on. To turn on voice 1, we could use the statement `POKE 54276,1`. Since 1 equals 00000001, the rightmost bit would become 1, as required to turn on voice 1. But in the process, all the other bits at location 54276 have been set to 0, destroying, among other things, information about which waveform is to be used in generating the sound for voice 1. In order not to destroy information represented by other bits at location 54276, we need to use an OR operation with the value already present. The statement `POKE 54276, PEEK(54276) OR 1` does the job. We can represent the eight-tuple as `XXXXXXXX OR 00000001` which gives `XXXXXXXX1`, where the Xs are 0 or 1 depending on the original value in location 54276.



## Appendix L

Releasing voice 1 requires that we simply AND the rightmost bit at location 54276 with 0. In order that the value of the other bits not be altered, they must be ANDed with 1s. The required eight-tuple can be represented by `XXXXXXXX AND 11111110` which gives `XXXXXXXX0`, where again the Xs are 0 and 1 depending on the original value in location 54276. Since 11111110 is equal to the decimal number 254, the required statement to release voice 1 is `POKE 54276, PEEK(54276) AND 254`.

# Appendix M

## Mathematical Concepts

### PART I THE NUMBER $\pi$

The number  $\pi$  appears in nearly every branch of mathematics. The value of  $\pi$  to 9 significant digits is 3.14159265. The number  $\pi$  occurs in geometry as the area of the circle of radius 1. It occurs in statistics as one of the constants in the formula for the normal curve. The number is obtained on the Commodore 64 by shifting the key containing the upward arrow.

### PART II RADIAN MEASURE OF ANGLES

The ancient Babylonians introduced angle measurement in terms of degrees, minutes, and seconds, and these units are still generally used today for navigation and practical measurements. The unit of angle measurement in the metric system and in most personal computers is called the *radian*.

Consider a unit circle, that is, a circle of radius  $r = 1$ . The length of the circumference (the outer edge) of a circle is known from geometry to be  $2*\pi*r$ , which in this case is  $2*\pi*1$  or  $2*\pi$ . The radian system measures angles in terms of the distance around the circumference of the unit circle. Thus, for example, sweeping out 360 degrees, or one full revolution, corresponds to moving along the circumference a distance of  $2*\pi$ . So 360 degrees corresponds to  $2*\pi$  radians. Here are some useful comparisons between degrees and radians:

|                               |                          |
|-------------------------------|--------------------------|
| 360 degrees = $2*\pi$ radians | (one full revolution)    |
| 180 degrees = $\pi$ radians   | (one half-revolution)    |
| 90 degrees = $\pi/2$ radians  | (one quarter-revolution) |
| 45 degrees = $\pi/4$ radians  | (one eighth-revolution)  |

In general, the radian measure of an angle is the length of its corresponding arc on the unit circle.

One degree is  $\pi/180$  radians. To convert from degrees to radians, multiply the number of degrees by  $\pi/180$ . The number  $\pi/180$  is .0174532925 to 9 significant digits.

One radian is  $180/\pi$  degrees. To convert from radians to degrees, multiply the number of radians by  $180/\pi$ . The number  $180/\pi$  is 57.2957795 to 9 significant digits.

Figure 1 shows the degree and radian measures of several familiar angles.

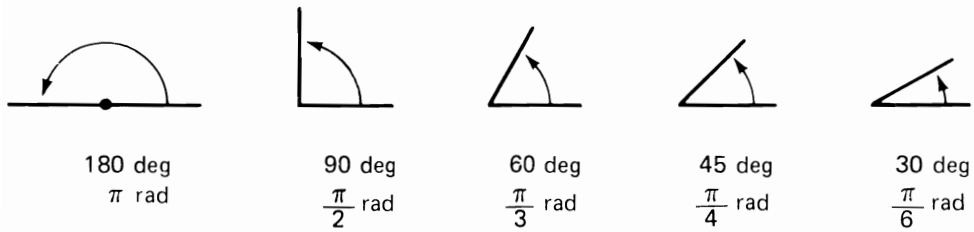


FIGURE 1

## PART III THE TRIGONOMETRIC FUNCTIONS

Suppose that we are given an acute angle  $A$ , i.e., an angle of less than  $\pi/2$  radians. We define three descriptive numbers associated to the angle  $A$ . These three numbers are called the sine, cosine, and tangent of  $A$  and are denoted  $\text{SIN}(A)$ ,  $\text{COS}(A)$ , and  $\text{TAN}(A)$ , respectively.

Consider the acute angle shown in Figure 2a. In Figure 2b we have drawn a right triangle with  $A$  as one of its angles. In Figure 2c we have labeled the other sides of the triangle by their relationship to  $A$ .

We define  $\text{SIN}(A)$  to be the length of the side opposite  $A$  divided by the length of the hypotenuse. This is abbreviated as

$$\text{SIN}(A) = \text{OPP}/\text{HYP}$$

Similarly, we define

$$\text{COS}(A) = \text{ADJ}/\text{HYP}$$

$$\text{TAN}(A) = \text{OPP}/\text{ADJ}$$

### Examples

1. Consider an angle of .523598776 radians (that is, 30 degrees). Figure 3 shows a 30-60-90 right triangle with the lengths of the sides

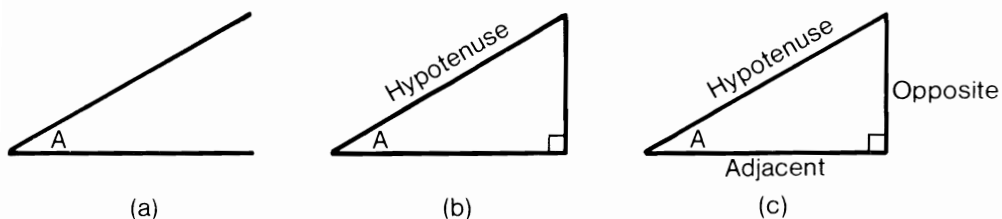


FIGURE 2

## Appendix M

indicated. (A theorem of geometry states that in any 30-60-90 triangle, the side opposite the 30 degree angle has length one-half the length of the hypotenuse. The length of the remaining side was determined from the Pythagorean theorem.) Referring to the triangle in Figure 3, we have

$$\text{SIN}(.523598776) = \text{OPP}/\text{HYP} = 1/2 = .5$$

$$\text{COS}(.523598776) = \text{ADJ}/\text{HYP} = \text{SQR}(3)/2 = .866025404$$

$$\text{TAN}(.523598776) = \text{OPP}/\text{ADJ} = 1/\text{SQR}(3) = .577350269$$

2. Consider an angle of .785398163 radians (i.e., 45 degrees). Figure 3b shows a 45-45-90 right triangle with the lengths of the sides indicated. Referring to the triangle, we have

$$\text{SIN}(.785398163) = \text{OPP}/\text{HYP} = 1/\text{SQR}(2) = .707106781$$

$$\text{COS}(.785398163) = \text{ADJ}/\text{HYP} = 1/\text{SQR}(2) = .707106781$$

$$\text{TAN}(.785398163) = \text{OPP}/\text{ADJ} = 1/1 = 1.$$

### Further Discussion

The definitions of the trigonometric functions sine, cosine, and tangent can be extended to angles other than acute angles. Consider an angle of  $A$  radians drawn with its vertex at the center of a unit circle and one side along the  $x$ -axis. See Figure 4. The other side of the angle intersects the unit circle at a point, call it  $P$ , with coordinates  $x$  and  $y$ . We define

$$\text{SIN}(A) = y \quad \text{COS}(A) = x \quad \text{TAN}(A) = y/x.$$

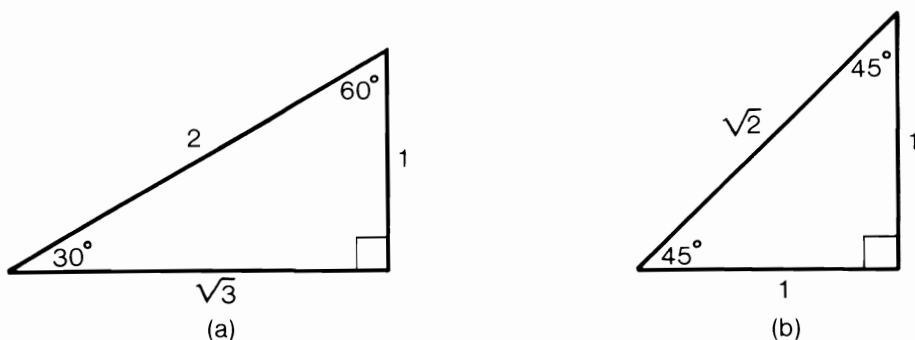
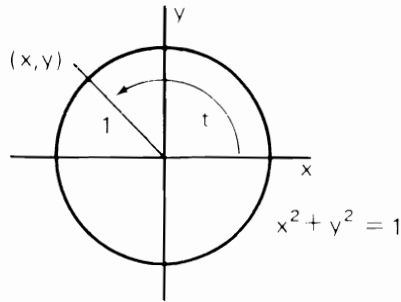


FIGURE 3



**FIGURE 4**

In other words,  $\text{SIN}(A)$  is the second coordinate of the point P,  $\text{COS}(A)$  is the first coordinate of P, and  $\text{TAN}(A)$  is a ratio of the coordinates of P. These three trigonometric functions are available in BASIC. There are three other common trigonometric functions which do not occur in BASIC, but can be computed in terms of the sine, cosine, the tangent functions. The cotangent of A is defined as  $\text{COT}(A) = x/y$  and is computed in BASIC as  $1/\text{TAN}(A)$ . The secant of A is defined as  $1/\text{COS}(A)$  and is computed in BASIC as  $1/\text{COS}(A)$ . The cosecant of A is defined as  $1/y$  and computed as  $1/\text{SIN}(A)$ .

## Further Examples

3. The following program determines the values of several trigonometric functions at an angle specified by the user.

```

10 INPUT "ANGLE IN DEGREES";A
20 R = A*π/180
30 PRINT "SINE OF";A;"DEGREES IS";SIN(R)
40 PRINT "COSINE OF";A;"DEGREES IS";COS(R)
50 PRINT "TANGENT OF";A;"DEGREES IS";
TAN(R)
60 PRINT "COTANGENT OF";A;"DEGREES IS";
1/TAN(R)
RUN
ANGLE IN DEGREES? 30
SINE OF 30 DEGREES IS .5
COSINE OF 30 DEGREES IS .866025404
TANGENT OF 30 DEGREES IS .577350269
COTANGENT OF 30 DEGREES IS 1.73205081

READY.
```

# Appendix N

## SPRITE POKE Summary

| Decimal Location | Description          | Decimal Location | Description           |
|------------------|----------------------|------------------|-----------------------|
| 2040             | SPRITE 0 data block# | 53264            | 9th bit of X pos      |
| 2041             | SPRITE 1 data block# |                  | SPRITES 0 thru 7      |
| 2042             | SPRITE 2 data block# | 53269            | SPRITE enable/        |
| 2043             | SPRITE 3 data block# |                  | disable               |
| 2044             | SPRITE 4 data block# | 53271            | Expand SPRITE         |
| 2045             | SPRITE 5 data block# |                  | vertically            |
| 2046             | SPRITE 6 data block# | 53275            | SPRITE/background     |
| 2047             | SPRITE 7 data block# |                  | display priority      |
| 53248            | X pos, SPRITE 0      | 53276            | Select/deselect       |
| 53249            | Y pos, SPRITE 0      |                  | multi-color SPRITE    |
| 53250            | X pos, SPRITE 1      | 53277            | Expand SPRITE         |
| 53251            | Y pos, SPRITE 1      |                  | horizontally          |
| 53252            | X pos, SPRITE 2      | 53278            | SPRITE/SPRITE         |
| 53253            | Y pos, SPRITE 2      |                  | collision detection   |
| 53254            | X pos, SPRITE 3      | 53279            | SPRITE/background     |
| 53255            | Y pos, SPRITE 3      |                  | collision detection   |
| 53256            | X pos, SPRITE 4      | 53285            | SPRITE multi-color 1  |
| 53257            | Y pos, SPRITE 4      | 53286            | SPRITE multi-color 2  |
| 53258            | X pos, SPRITE 5      | 53287            | SPRITE-color SPRITE 0 |
| 53259            | Y pos, SPRITE 5      | 53288            | SPRITE-color SPRITE 1 |
| 53260            | X pos, SPRITE 6      | 53289            | SPRITE-color SPRITE 2 |
| 53261            | Y pos, SPRITE 6      | 53290            | SPRITE-color SPRITE 3 |
| 53262            | X pos, SPRITE 7      | 53291            | SPRITE-color SPRITE 4 |
| 53263            | Y pos, SPRITE 7      | 53292            | SPRITE-color SPRITE 5 |
|                  |                      | 53293            | SPRITE-color SPRITE 6 |
|                  |                      | 53294            | SPRITE-color SPRITE 7 |

## Music Synthesis POKE Summary

Music synthesis on the Commodore 64 is handled by a Sound Interface Device chip, or SID for short. The registers for controlling SID correspond to memory addresses 54272 thru 54300. By POKEing appropriate values into these locations, a chorus of 3 distinct voices can be generated. The following chart summarizes the use of each location.

### Voice 1

---

|               |                                                     |
|---------------|-----------------------------------------------------|
| 54272 & 54273 | scaled frequency (lo/hi)                            |
| 54274 & 54275 | pulse width (lo/hi)                                 |
| 54276         | wave form + ring modulation + sync + on/off control |
| 54277         | 16 * attack rate + decay rate                       |
| 54278         | 16 * sustain level + release rate                   |

### Voice 2

---

|               |                                                     |
|---------------|-----------------------------------------------------|
| 54279 & 54280 | scaled frequency (lo/hi)                            |
| 54281 & 54282 | pulse width (lo/hi)                                 |
| 54283         | wave form + ring modulation + sync + on/off control |
| 54284         | 16 * attack rate + decay rate                       |
| 54285         | 16 * sustain level + release rate                   |

### Voice 3

---

|               |                                                     |
|---------------|-----------------------------------------------------|
| 54286 & 54287 | scaled frequency (lo/hi)                            |
| 54288 & 54289 | pulse width (lo/hi)                                 |
| 54290         | wave form + ring modulation + sync + on/off control |
| 54291         | 16 * attack rate + decay rate                       |
| 54292         | 16 * sustain level + release rate                   |

### General

---

|               |                                      |
|---------------|--------------------------------------|
| 54293 & 54294 | scaled filter frequency (lo 3/hi 8)  |
| 54295         | 16 * resonance + filter controls     |
| 54296         | osc. 3 detach + filter type + volume |
| 54299         | read oscillator 3                    |
| 54300         | read amplitude envelope 3            |

If a description is followed by the notation (lo/hi), then the first location is assigned  $\text{VALUE} - 256 * \text{INT}(\text{VALUE} / 256)$  and the second location is assigned  $\text{INT}(\text{VALUE} / 256)$ , where VALUE is the item described for the two locations. In the special case of the scaled filter frequency,

## Appendix O

location 54293 is assigned  $\text{VALUE} - 8 * \text{INT}(\text{VALUE} / 8)$  and location 54294 is assigned  $\text{INT}(\text{VALUE} / 8)$ .

Terms referred to in Summary:

*Scaled frequencies* can be found in Appendix S.

*Pulse width* is  $40.95 * \text{PERCENT}$ , where PERCENT is 0 to 100 percent.

*Waveform* is 128 for noise, 64 for pulse, 32 for sawtooth and 16 for triangular.

*Ring modulation* is 4.

*Sync* is 2.

*On* is 1 / *Off* is 0.

*Attack, decay, sustain, and release* tables are given under MUSIC AMPLITUDE ENVELOPE.

*Resonance* may range from 0 for none to 15 for maximum.

*Filter controls*: To activate filtering of voice 3 is 4, of voice 2 is 2, and of voice 1 is 1. Add values to filter more than one voice.

To *detach oscillator 3* from output is 128.

*Filter types*: high pass filter is 64, band pass is 32, low pass is 16.

*Volume* may range from 0 to 15. No sound is heard if volume is 0.



## RS-232 Interface Communications

The Commodore 64 has a built-in RS-232 interface for connection to any RS-232 modem, printer, or other device. Thus, the Commodore has the ability to communicate with other computers or devices other than those for which direct connections are provided. Access to the RS-232 interface is gained by a statement of the form

```
OPEN N,2,0,CHR$(A)+CHR$(B)+CHR$(C)+CHR$(D)
```

As with OPEN statements for other devices and files, N becomes the reference number to be used in GET#, INPUT#, and PRINT# statements. The value of N is normally between 1 and 127. Using values of N between 128 and 255 causes a line feed, CHR\$(10), to be added after each RETURN character, CHR\$(13), that is PRINTed.

The values of A, B, C, and D are used to specify various parameters which control exactly how data is to be transmitted and received. The values of A and B are determined as follows:

$$A = 128 * SB + 32 * WL + BR$$
$$B = 32 * PR + 16 * DP + HS$$

The charts below give the possible values and interpretations for the six parameters in these equations.

*SB* gives the number of *Stop Bits*:

- 0 - use 1 stop bit
- 1 - use 2 stop bits

*WL* gives the *Word Length*:

- 0 - word consists of 8 bits
- 1 - word consists of 7 bits
- 2 - word consists of 6 bits
- 3 - word consists of 5 bits

*BR* gives the *Baud Rate*:

- 0 - user gives rate via CHR\$(C)+CHR\$(D)
- 1 - 50 baud
- 2 - 75
- 3 - 110
- 4 - 134.5
- 5 - 150
- 6 - 300
- 7 - 600

# Appendix P

- 8 - 1200
- 9 - 1800
- 10 - 2400
- 11 - 3600
- 12 - 4800
- 13 - 7200
- 14 - 9600
- 15 - 19200

Note: CHR\$(C) + CHR\$(D) need only be included in the open statement if BR = 0. In this case, if R is the desired baud rate, then

$$D = \text{INT} ((1.02273\text{E}6/\text{R}/2-100/256)$$

$$C = 1.02273\text{E}6/\text{R}/2-100-256*D$$

*PR* gives the *Parity*:

- 0, 2, or 4 - parity disabled, none generated or received
- 1 - odd parity for receiver and transmitter
- 3 - even parity for receiver and transmitter
- 5 - mark transmitted, parity check disabled
- 7 - space transmitted, parity check disabled

*DP* gives the *Duplex*:

- 0 - full duplex
- 1 - half duplex

*HS* gives the *Handshake*:

- 0 - use 3 line handshake
- 1 - use X line handshake

## Standard Graphics

### PART I SPECIFYING COORDINATES IN GRAPHICS MODES

For graphics purposes, each point of the screen is specified by a pair of numbers, called coordinates. The point with coordinates  $(x,y)$  can be reached by starting at the upper left-hand point of the screen and moving  $x$  points to the right and then  $y$  points down. (See Figure 1(a).) This convention for labeling points arises from a mathematical coordinate system with the positive  $y$ -axis pointing downward. (See Figure 1(b).)

The values of  $y$  range from 0 to 199. The values of  $x$  range from 0 to 319 in high-resolution graphics mode and from 0 to 159 in multi-color graphics mode. The center of the screen has coordinates  $(160,100)$  in high-resolution and  $(80,100)$  in multi-color resolution graphics mode. Figure 2 shows the coordinates of several points.

### PART II GRAPHICS IN HIGH-RESOLUTION GRAPHICS MODE

High-resolution graphics mode allows each of the  $320 \times 200$  dots on the screen to be independently turned "on" or "off". The following subroutine can be placed within a program to invoke high-resolution mode and prepare the screen.

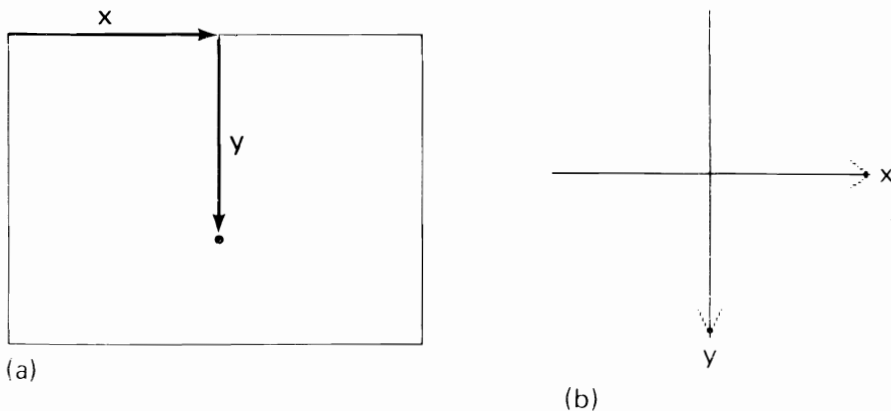
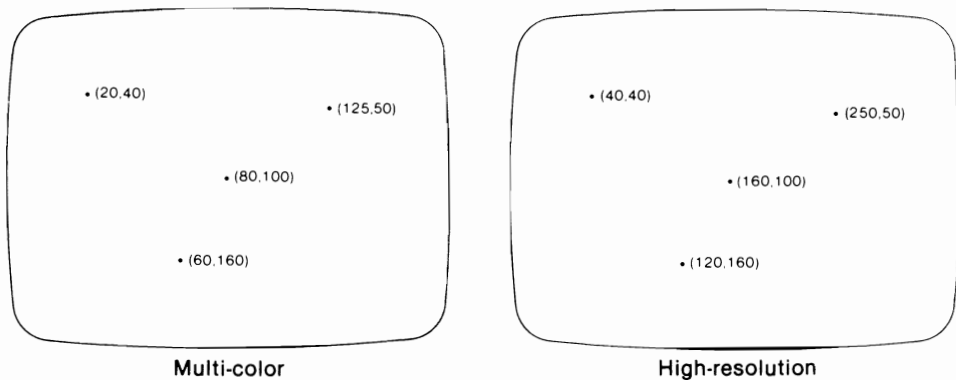


FIGURE 1

## Appendix Q



**FIGURE 2**

```
1000 POKE 53265,PEEK(53265) OR 32
1010 POKE 53272,PEEK(53272) OR 8
1020 FOR I = 8192 TO 16191
1030 POKE I,0: NEXT I
1040 DOT=13: BACK=6
1050 FOR I = 1024 TO 2023
1060 POKE I,DOT*16+BACK: NEXT I
1070 RETURN
```

Note: This subroutine requires approximately 40 seconds to execute.

Line 1000 invokes high-resolution graphics mode. Line 1010 selects 8000 memory locations to be used as the data or *map* specifying which dots are “on” and “off”. Lines 1020 and 1030 initialize these 8000 memory locations so that all dots are “off”. Line 1040 chooses the color of “on” dots to be light blue (color code 13) and the color of “off” dots to be blue (color code 6). See Appendix C for other possible color selections. Finally, lines 1050 and 1060 place the “on” and “off” colors into memory. (Note: To restore the screen to standard character mode, type RUN/STOP-RESTORE.)

The following subroutine will plot the point having coordinates (X,Y).

```
2000 BY=8192+320*INT(Y/8)+8*INT(X/8)+(Y AND 7)
2010 BI=7-(X AND 7)
2020 POKE BY,PEEK(BY) OR 2↑BI
2030 RETURN
```

Now we can incorporate these subroutines into a program that will turn on a point specified by the user.

## Appendix Q

```
10 INPUT X,Y
20 GOSUB 1000
30 GOSUB 2000
40 END
1000 ...
2000 ...
```

The following program draws a straight line between any two points specified by the user. The coordinates of the points will be taken as (X1,Y1) and (X2,Y2).

```
10 INPUT X1,Y1,X2,Y2
20 GOSUB 1000
30 IF X1=X2 THEN 100
40 S = (Y1-Y2)/(X1-X2)
50 FOR I = 0 TO (X2-X1) STEP SGN(X2-X1)
60 X=X1+I: Y=Y1+I*S: GOSUB 2000
70 NEXT I
80 GET A$: IF A$="" THEN 80
90 END
100 FOR I = 0 TO (Y2-Y1) STEP SGN(Y2-Y1)
110 X=X1: Y=Y1+I: GOSUB 2000
120 NEXT I
130 GET A$: IF A$="" THEN 130
140 END
1000 ...
2000 ...
```

The following program graphs the function given in line 10. The numbers on the x-axis will range from -A to A and the numbers on the y-axis will range from -B to B. We can graph other functions by changing the values for FNF(X), A, and B, in lines 10 and 20.

```
10 DEF FNF(X) = 30*SIN(X)
20 A=6: B=40
30 GOSUB 1000: REM INITIALIZE SCREEN
40 GOSUB 3000: REM DRAW AXES
50 FOR X = 0 TO 319
60 Y = 100-(100/B)*FNF(A*(X-160)/159)
70 GOSUB 2000
80 NEXT X
90 GET A$: IF A$="" THEN 90
100 END
1000 ...
2000 ...
```

## Appendix Q

```
3000 Y = 100
3010 FOR X=0 TO 319
3020 GOSUB 2000: REM PLOT POINT (X,100)
3030 NEXT X
3040 X = 160
3050 FOR Y=0 TO 199
3060 GOSUB 2000: REM PLOT POINT (160,Y)
3070 NEXT Y
3080 RETURN
```

## Three-Voice Chorus Program

The program below can be modified by the user to play up to 40 measures (4/4 time) of music, with one, two, or three-voice harmony. Besides specifying your own melodies in the data statements beginning at lines 603, 703, and 803, you also may wish to change the waveform assignments in lines 13 thru 15, the pulse width assignment in lines 16 and 17, and the amplitude envelope assignments in lines 73 thru 75.

The method for specifying melody data is designed to be straightforward. The only mathematics you will need is the following chart, which gives the duration number for the different note counts found in music. You can derive this table yourself, using the fact that each duration given is the number of 32nd beats for the given note count. For example, a quarter note is equal to 8 32nd beats, thus 8 is the duration for a quarter note.

| <u>note count</u> | <u>duration</u> |
|-------------------|-----------------|
| sixteenth         | 2               |
| eighth            | 4               |
| dotted eighth     | 6               |
| quarter           | 8               |
| dotted quarter    | 12              |
| half              | 16              |
| dotted half       | 24              |
| whole             | 32              |

Enter your melody in the data statements beginning with 603 for voice 1, 703 for voice 2, and 803 for voice 3. For each note to be played, give the name of the note, for example A, F, B- (B flat) or C# (C sharp), followed by its duration from the table above. For a rest, use the name P (pause) and then a duration.

There are 8 octaves of 12 notes each which can be played. Each octave ranges from C up to B. The program begins in octave 4, which is the octave containing middle C. You may change octaves at any time by placing the symbols "<" or ">" in your data statements before the notes which are to be played in the new octave. The symbol "<" moves subsequent notes up one octave, while the symbol ">" moves subsequent notes down one octave. Using "<<" will move you up two octaves, ">>>" down three octaves, and so forth.

You may use as many data statements as needed for each voice. However, the last data statement for each voice must end with the three characters P0\* . The following data statement is the translation

# Appendix R



FIGURE 1

of the 4 measures of music in Figure 1. It contains all the features mentioned above.

603 DATA <D8D4C#4D8>A8B8<D8C#8>B8A8A4B4A8F#8G8A8E16

```
1 REM CREATION AND INITIALIZATION OF VARIABLES
2 REM
3 REM
4 DIM F7(12) :REM STORAGE FOR OCTAVE 7
 SCALED FREQUENCIES
5 DIM H%(3,1400),L%(3,1400) :REM STORAGE FOR FREQUENCIES
 DURING 1400 32TH-MEASURE
6 REM INTERVALS FOR THE THREE
 VOICES
7 DIM S%(3,1400) :REM STORAGE FOR THE STATE,
 ON OR OFF, OF EACH VOICE
8 REM DURING EACH 32ND-MEASURE
 INTERVAL
9 DIM WF(3) :REM STORAGE FOR VALUE GIVING
 WAVEFORM FOR EACH VOICE
10 REM
11 FOR I=54272 TO 54296 :REM CLEAR ALL MUSIC CONTROL
 LOCATIONS
12 POKE I,0: NEXT I
13 WF(1)=16 :REM VOICE 1 USES TRIANGULAR WAVE
14 WF(2)=32 :REM VOICE 2 USES SAWTOOTH WAVE
15 WF(3)=64 :REM VOICE 3 USES PULSE WAVE
16 POKE 54288, 25*40.95-256*INT(25*40.95/256):
 REM PULSE WIDTH IS 25 PERCENT
17 POKE 54289, INT(25*40.95/256)
18 FOR I=1 TO 12 :REM READ IN OCTAVE 7 FREQUENCIES
19 READ F7(I): NEXT I
20 DATA 34334,36376,38539,40830,43258,45830
21 DATA 48556,51433,54502,57743,61176,64814
22 Z=1 :REM INITIALIZE MARKER OF NEXT
 PIECE OF NOTE DATA
 TO BE READ
23 REM
24 MC=0 :REM GREATEST NUMBER OF
 MEASURES USED BY A
 VOICE SO FAR
25 REM
26 REM
27 REM
```



# Appendix R

```
28 REM PROCESS DATA FOR EACH VOICE TO CREATE
 THE H%, L%, and S% ARRAYS
29 REM
30 FOR K=1 TO 3
31 MN=0 :REM NUMBER OF MEASURES USED
 BY CURRENT VOICE
32 OC=4 :REM INITIAL OCTAVE IS 4
33 GOSUB 100 :REM GET NEXT VALUES OF NOTE
 NUMBER AND DURATION
34 IF DR=0 THEN 57 :REM GO TO NEXT VOICE
35 IF DR>0 THEN 41
36 FOR I=1 TO -DR :REM NEGATIVE DURATION,
 NOTE IS A REST, CONTINUE RELEASE
37 MN=MN+1
38 H%(K,MN)=HF: L%(K,MN)=LF: S%(K,MN)=WF(K)
39 NEXT I
40 GOTO 33 :REM GOTO NEXT NOTE
41 FQ=F7(NN)/(2^(7-OC)) :REM COMPUTE FREQUENCY
 FROM OCTAVE 7 VALUE
42 HF=INT(FQ/256)
43 LF=FQ-256*HF
44 IF DR>1 THEN 49
45 REM DURATION IS ONLY 1 32TH
 INTERVAL, NOTE MUST BE ON
46 MN=MN+1
47 H%(K,MN)=HF: L%(K,MN)=LF: S%(K,MN)=WF(K)+1
48 GOTO 33 :REM GOTO NEXT NOTE
49 FOR I=1 TO DR-1 :REM STORE REQ. & STATE FOR
 ALL BUT LAST INTERVAL
50 MN=MN+1
51 H%(K,MN)=HF: L%(K,MN)=LF: S%(K,MN)=WF(K)+1
52 NEXT I
53 MN=MN+1
54 REM RELEASE VOICE DURING LAST
 INTERVAL OF ITS DURATION
55 H%(K,MN)=HF: L%(K,MN)=LF: S%(K,MN)=WF(K)
56 GOTO 33 :REM GOTO NEXT NOTE
57 IF MN>MC THEN MC=MN
58 PRINT "VOICE";K;"PROCESSED"
59 NEXT K :REM GOTO NEXT VOICE
70 REM
71 REM SET AMPLITUDE ENVELOPES, VOLUME, AND PLAY!
72 REM
73 POKE 54277,0: POKE 54278,240:
 REM VOICE 1 IS AN ORGAN
74 POKE 54284,10: POKE 54285,0:
 REM VOICE 2 IS A HARPSICORD
75 POKE 54291,88: POKE 54292,192:
 REM VOICE 3 IS A TRUMPET
76 POKE 54296,15 :REM OVERALL VOLUME AT MAXIMUM
77 FOR I=1 TO MC :REM PLAY MAESTRO!
78 POKE 54272,L%(1,I): POKE 54273,H%(1,I)
79 POKE 54279,L%(2,I): POKE 54280,H%(2,I)
80 POKE 54286,L%(3,I): POKE 54287,H%(3,I)
81 POKE 54276,S%(1,I): POKE 54283,S%(2,I)
82 POKE 54290,S%(3,I)
83 NEXT I
84 POKE 54296,0 :REM TURN OFF VOLUME
```

# Appendix R

```
85 END

100 REM
101 REM DETERMINE NOTE NUMBER (NN)
 AND DURATION (DR) OF NEXT NOTE
102 REM
103 IF A$=">" THEN GOSUB 300: GOTO 113
104 IF A$="<" THEN GOSUB 400: GOTO 113
105 IF A$="P" THEN GOSUB 500: RETURN
106 IF A$="A" THEN NN=10:GOTO 120
107 IF A$="B" THEN NN=12:GOTO 120
108 IF A$="C" THEN NN=1:GOTO 120
109 IF A$="D" THEN NN=3:GOTO 120
110 IF A$="E" THEN NN=5:GOTO 120
111 IF A$="F" THEN NN=6:GOTO 120
112 IF A$="G" THEN NN=8:GOTO 120
113 GOSUB 200: GOTO 103
120 GOSUB 200
121 IF A$<>"#" THEN 126 :REM IS NOTE TO BE SHARP?
122 IF NN<12 THEN 125 :REM IS NOTE AT TOP OF OCTAVE?
123 IF OC=7 THEN 132 :REM IS THE CURRENT OCTAVE
 THE HIGHEST?
124 OC=OC+1: NN=0: GOTO 132 :REM MOVE UP AN OCTAVE TO
 MAKE NOTE SHARP
125 NN=NN+1: GOTO 132 :REM MAKE NOTE SHARP
126 IF A$<>"-" THEN 133 :REM IS NOTE TO BE FLAT?
127 IF NN>1 THEN 129 :REM IS NOTE AT BOTTOM
 OF OCTAVE?
128 IF OC=0 THEN 132 :REM IS THE CURRENT OCTAVE
 THE LOWEST?
129 OC=OC-1: NN=11: GOTO 132 :REM MOVE DOWN AN OCTAVE
 TO MAKE NOTE FLAT
130 NN=NN-1 :REM MAKE NOTE FLAT
131 REM DETERMINE DURATION OF NOTE
132 GOSUB 200
133 DR=0
134 IF A$<"0" OR A$>"9" THEN 138
135 DR=DR*10+ASC(A$)-ASC("0")
136 GOSUB 200
137 GOTO 134
138 RETURN :REM RETURN TO MAIN PROGRAM
 WITH NN AND DR

200 REM
201 REM GET A CHARACTER OF DATA
202 REM
203 IF Z>LEN(M$) THEN Z=1: READ M$
204 A$=MID$(M$,Z,1) :REM GET NEXT CHARACTER OF DATA
205 Z=Z+1
206 RETURN

300 REM
301 REM HANDLE DECREASE IN OCTAVE
302 REM
303 IF OC=0 THEN RETURN
304 OC=OC-1
305 RETURN
```

## Appendix R

```
400 REM
401 REM HANDLE INCREASE IN OCTAVE
402 REM
403 IF OC=7 THEN RETURN
404 OC=OC+1
405 RETURN

500 REM
501 REM RETURN NEGATIVE DURATION FOR PAUSE
502 REM
503 GOSUB 200
504 DR=0
505 IF A$<"0" OR A$>"9" THEN 509
506 DR=DR*10-ASC(A$)+ASC("0")
507 GOSUB 200
508 GOTO 505
509 RETURN

600 REM
601 REM DATA FOR VOICE 1
602 REM
603 DATA P8D4D4F4F4P4E4P4E8P12C4C4E4E4P4D4P4D8P0*
700 REM
701 REM DATA FOR VOICE 2
702 REM
703 DATA P16D4D4P4C4P4C8P20C4C4P4>A4P4A8P0*
800 REM
801 REM DATA FOR VOICE 3
802 REM
803 DATA D4D4F4F4A4A4G20C4C4C4E4E4G4G4F20P0*
```

# Appendix S

## Music Note Scaled Frequencies

| N<br>o<br>t<br>e | <u>Octave</u> |      |      |      |       |       |       |       |
|------------------|---------------|------|------|------|-------|-------|-------|-------|
|                  | 0             | 1    | 2    | 3    | 4     | 5     | 6     | 7     |
| C                | 268           | 536  | 1072 | 2145 | 4291* | 8583  | 17167 | 34334 |
| C#/D-            | 284           | 568  | 1136 | 2273 | 4547  | 9094  | 18188 | 36376 |
| D                | 301           | 602  | 1204 | 2408 | 4817  | 9634  | 19269 | 38539 |
| D#/E-            | 318           | 637  | 1275 | 2551 | 5103  | 10207 | 20415 | 40830 |
| E                | 337           | 675  | 1351 | 2703 | 5407  | 10814 | 21629 | 43258 |
| F                | 358           | 716  | 1432 | 2864 | 5728  | 11457 | 22915 | 45830 |
| F#/G-            | 379           | 758  | 1517 | 3034 | 6069  | 12139 | 24278 | 48556 |
| G                | 401           | 803  | 1607 | 3215 | 6430  | 12860 | 25721 | 51443 |
| G#/A-            | 425           | 851  | 1703 | 3406 | 6812  | 13625 | 27251 | 54502 |
| A                | 451           | 902  | 1804 | 3608 | 7217  | 14435 | 28871 | 57743 |
| A#/B-            | 477           | 955  | 1911 | 3823 | 7647  | 15294 | 30588 | 61176 |
| B                | 506           | 1012 | 2025 | 4050 | 8101  | 16203 | 32407 | 64814 |

\* Middle C

# Index

Page numbers shown in bold face provide the primary information for the subject.  
Page numbers shown in italics provide an example of the subject's use in a program.

- ABS, **13-14**, 49
- Absolute error, 98
- Absolute value, 13
- ADSR table, 286
- Alphabetize, 69, 95
- Amplitude envelope, **285-288**
- AND, 97, 333
- Animation, 257, 264
- Arccosine, 21, 38
- Arccotangent, 22
- Arccosecant, 22
- Arcsecant, 22
- Arcsine, 21, 211
- Arctangent, 20
- Array, 50
- ASC, **15-19**, 24, 195
- ASCII/Commodore values, 3, 7, 15, 23, 24, **295-302**
- ATN, **20-22**, 39, 211, 234
- Attack, 285, 286, 288, 341
- Automatic DIMensioning, 53, 54
  
- Binary numbers, 305, 325, 326
- Bisection method, 49
- Bit, 305
- Byte, 305, 332
- Byte arithmetic, 333, 334
  
- Cassette, 9
  - file, 115
  - player, 7, 218
- Chaining, 137
- Channel number, 155
- Character sets, 5, 303, 304
- CHR\$, 16, **23-25**, 165, 319
- Clock, 237, 239
- CLOSE, **26-28**, 311
- CLR, **29-30**, 53, 136, 147, 205
- CMD, **31-34**, 133
- Cold start, 240
- Collision detection of SPRITEs, **275-277**
- Color codes, 307
- Comma, 173, 230
- Command channel, 308
  
- Common logarithmic function, 139
- Comparing programs, 246
- Compound conditions, 331
- Computed GOSUB, 86
- Computed GOTO, 92
- Concatenation, 3
- Conditions, 3, 94, 331
- CONT, **35-37**, 56, 221, 222
- Control characters, 15, 172, 178, 181, 317
- Control characters, printer, **327-330**
- Coordinates in graphics modes, 345
- COPY, 308
- Copying statements, 134
- COS, **38-40**, 337
- Cosecant, 339
- Cosine, 337
- Cotangent, 339
  
- DATA, **41-44**, 186, 192
- Data file, 8, 78, 108, 154, **316-318**
- Datasette recorder, 9
- Deactivated FOR . . . NEXT loop, 70, 71
- Debugging programs, 37, 87, 93, 106, 134, 164, 180, 191, 200, 223
- Decay, 285, 286, 288, 341
- DEF FN, 29, **45-49**, 205
- Default values, 103
- DELeTe, 25
- Deleting disk files, 310
- Delimiter, 316, 317
- Device, 7, 108, 154, 181
- Device number, 7, 108, 154, 181
- DIM, 30, **50-55**, 205
- Direct mode, 1, 85
- Directory of files, 137
- Disk drive, 7, 218
- Disk id, 309
- Disk name, 309
- Disk operating system (DOS)
  - commands, **308-311**
  - errors, 311
  - COPY, 308
  - INITIALIZE, 309

Disc operating system (DOS) (*continued*)  
   NEW, 309  
   RENAME, 310  
   SCRATCH, 310  
   VALIDATE, 311  
 Display priorities of SPRITES, **273-274**  
 DOS commands (see Disk operating system commands)  
 Double-width print mode, 327  
  
 Empty string, 17, 81  
 Enabling SPRITES, 258, 263, **265-266**, 340  
 END, 35, **56-58**  
 End-of-file, 81, 115, 182, 218, 318  
 End-of-tape, 159, 218  
 Entering, 9  
 Erasing old disk files, 310  
 Error messages, **312-315**  
   BAD SUBSCRIPT, 30, 53, 55  
   CAN'T CONTINUE, 35, 36, 223  
   DEVICE NOT PRESENT, 184  
   EXTRA IGNORED, 76, 80, 105  
   FILE DATA, 34, 109, *110*, 113  
   FILE NOT FOUND, 158  
   FILE NOT OPEN, 32, 158, 184  
   FILE NOT OUTPUT FILE, 184  
   FILE NOT PRESENT, 158  
   FILE OPEN, 157, *158*  
   ILLEGAL DEVICE NUMBER, 247  
   ILLEGAL DIRECT, 48, 77, 82, 106  
   NEXT WITHOUT FOR, *70*  
   NOT OUTPUT FILE, 163  
   OUT OF DATA, 188, 193  
   OUT OF MEMORY, 29, 48, 54, 86  
   OVERFLOW, 61, 101  
   REDIM'D ARRAY, 53, 54  
   REDO FROM START, 101, *106*  
   RETURN WITHOUT GOSUB, 83, 85  
   STRING TOO LONG, 111, 115  
   TOO MANY FILES, 157  
   TYPE MISMATCH, *127*, *130*, 188  
   UNDEF'D FUNCTION, 48  
   UNDEF'D STATEMENT, 86, 92, 153, 206  
   VERIFY, 246  
 EXP, **59-61**, 140  
 Expanded screen, 269  
 Expanding SPRITES, 258, 263, **267-268**, 272, 340  
 Exponential function, 59  
 Exponential random variable, 204  
  
 File buffer, 26, 29, 184, 205  
 Filename, 8, 158  
 Flag, 43  
 Floating-point  
   constant, 2, 140  
   notation, 2  
   variable, 128, 189  
 FOR . . . NEXT, 30, **62-71**, 210  
 Forgotten file or device, 27, 28, 34, 136, 157  
 Format new disk, 309  
 FRE, **72-73**  
 Functions, 45  
  
 GET, 17, 26, **74-77**, 82, 92, 255  
 GET#, **78-82**, 160, *161*, *164*, 219, 318  
 GOSUB, 19, 29, 56, **83-89**, *144*, 149, 222, *319*, *352*  
 GOSUB and RETURN, **83-89**  
 GOTO, 56, **90-93**, 95, 149, 205, 222  
 Graphics mode, printer, 327  
 Graphs of functions, 232  
  
 Hash sort, 18  
 High-resolution  
   graphics, 345  
   mode, 253  
  
 IF, 91, **94-99**  
 IF . . . GOTO, 91, 95  
 IF . . . THEN, **94-99**  
 Index, 62, 64  
 Infinite loop, 90, 91  
 INITIALIZE, 309  
 INPUT, 8, **100-107**, 121, 143  
 INPUT#, 26, **108-119**, *156*, 160, *162*, 316, 318  
 INSert, 25  
 Insert mode, 7, 179  
 Instruments, 287  
 INT, 19, 101, **120-122**, *201*, 202, *209*, *270*, *283*, *291*, *350*  
 Integer  
   constant, 2  
   variable, 128, 189  
  
 Jiffy clock, 237, 239  
  
 Keyboard, 8  
 Keyboard buffer, 74, 118, 162, 165, 249, 323  
  
 LEFT\$, 88, **123-125**

LEN, **126-127**, 144, 170, 214, 233  
 LET, 8, **128-131**  
 LIFO stack, 83  
 Line input, 75, 79  
 LIST, 31, **132-134**  
 Listening, 33  
 LOAD, 30, **135-138**  
 LOG, 59, 60, **139-142**, 204  
 Logical operators, 3, 97, **331-335**  
 Logical screen line, 7, 170  
 Lowercase/Uppercase mode, 5, 15, 17, 23, 159, 183, **299-302**  
  
 Machine infinity, 61, 101  
 Machine language subroutine, 226, 242  
 Machine zero, 142  
 Memory allocation, 322  
 Memory locations, 165, 167  
 MID\$, 16, 19, 124, 125, 127, **143-146**, 178, 193  
 Monte Carlo calculations, 203  
 Moving SPRITEs, **269-272**  
 Multi-color mode, 259, 277  
 Multi-colors, 259, 263, 307  
 Multiple choice, 75  
 Multiple PRINT statement, 173  
 Multiple-statement lines, 1, 32, 96  
 Music  
   amplitude envelope, **285-288**  
   note selection, **281-284**  
   on/off control, 341, **289-292**  
   synthesizer, 281  
   volume, 283, 284, **285-288**  
   waveforms, 283, 284, **289-292**, 341  
  
 Natural logarithmic function, 59, 139  
 Nested loops, 69, 70  
 NEXT, **62-71**  
 NEW, **147-148**  
 NEW (DOS), 309  
 Noise waveform, 289, 291  
 NOT, 97, 333  
 Note selection, **281-284**  
 Null  
   character, 117, 184  
   effect, 15  
   string, 17  
 Numeric  
   constant, 2, 41  
   expression, 2  
   operator, 2  
   variable, 2, 128  
 Nybble, 305  
  
 ON, **149-153**, 209  
 ON . . . GOSUB, **149-153**  
 ON . . . GOTO, **149-153**, 209  
 On/off control, **289-292**, 341  
 OPEN, 26, 34, 109, 119, **154-164**, 182, 308  
 Operations on bytes, 333  
 OR, 97, 333  
 Order of operations, 3, 97  
  
 PEEK, 137, **165-166**, 167, 168, 271, 275  
 Peripheral device, 7, 108, 154, 181  
 Physical screen line, 7, 170  
 Pi, 2, 336  
 Plotting lines, 347  
 Plotting points, 346  
 POKE, 137, 166, **167-169**, 226, 242, 254-273, 282-291, 319, 350  
 POS, **170-171**  
 Precision, 2, 189  
 Prime numbers, 135  
 PRINT, **172-180**, 213, 229  
 Print zones, 173, 177, 181, 213  
 PRINT#, 163, 155-164, **181-185**, 308, 316, 317  
 Printer, 8, 159, 218  
 Printer control characters, **327-330**  
 Program file, 8  
 Program mode, 1  
 Prompt, 100  
 Pseudorandom numbers, 199  
 Pulse waveform, 289, 291  
 Pulse width, 290, 341  
 Pushdown stack, 83  
  
 Quotation marks, 42  
 Quote mode, 6, 178, 295, 297, 299, 301  
  
 Radian measure, 336  
 Random access memory (RAM), 72, 165, 167, 305, 322  
 Random number, 198  
 READ, 41, **186-189**, 192  
 Recovering lost disk space, 311  
 Redirected output, 31  
   printer, 31  
   screen, 34  
 Reference number, 26, 154, 157, 163, 205, 206, 316  
 Relational operators, 3, 331  
 Relative error, 98  
 Release, 284, 285, 286, 288, 289, 341  
 REM, **190-191**, 350

RENAME disk files, 310  
 Replacing files, 160, 208  
 Reserved words, 129, **324**  
 RESTORE, 189, **192-194**  
 RETURN, **83-89**  
 RETURN character, 25, 79  
 Reverse image print mode, 328  
 Reverse video, 6, 179  
 Reverse video character, 8  
 RIGHT\$, **195-197**  
 RND, **198-204**  
 Rounding, 122, 209  
 Rounding errors, 98  
 RS-232 interface, 8, 163, 343  
 RUN, 29, 30, 56, 91, **205-206**

**SAVE, 207-208, 246**  
 Sawtooth waveform, 289, 291  
 Scaled frequency, 281, 283, 341, 354  
 Scientific notation, 2  
 SCRATCH, 310  
 Screen, 8, 170  
     border and background, 167, 307  
     input, 117, 162  
     memory, 322  
     output, 162, 229  
 Scrolling, 223  
     slowed, 133  
 Secant, 339  
 Semicolon, 173, 175, 230  
 Sequential data file, 316  
 Serial bus, 157  
 SGN, 49, 150, **209-210**  
 SHIFT-RETURN character, 25, 114  
 Signal, 43  
 Simple condition, 3  
 Simulation, 203  
 SIN, **211-212, 337**  
 Sine, 337  
 Sound Interface Device, 341  
 SPC, **213-215, 321**  
 SPRITE collision detection, **275-277, 340**  
 SPRITE color, 253, 254, 259, 261-263, 307, 340  
 SPRITE data, 253-257, 259-263, 319, 340  
 SPRITE design, **253-264**  
     high-resolution, **253-258**  
     multi-color, **259-264**  
     program, 319  
 SPRITE display priorities, **273-274**  
 SPRITE pointer, 254, 259  
 SQR, **216-217**  
 ST, 220

Stack, BASIC's, 70, 86  
 Standard display mode, 6  
 Standard graphics, 345  
 STATUS, 115, 161, **218-220, 318, 323**  
 STOP, 35, 57, **221-223**  
 Stopwatch, 92, 239  
 STR\$, 124, 127, 196, **224-225, 245**  
 String, 2  
     constant, 2, 41, 102  
     encoding, 18  
     expression, 3  
     operator, 3  
     variable, 2, 128  
 Structured programming, 87, 319  
 Subroutine, 84, 87, 191  
 Subscript, 50  
 Sustain, 283, 284, 285, 286, 288, 341  
 Switching character sets, 25  
 Symbols

|       |               |
|-------|---------------|
| \$    | 2             |
| %     | 2             |
| ,     | 173, 230      |
| ;     | 173, 175, 230 |
| ??    | 105           |
| @0:   | 160, 207, 208 |
| E     | 2             |
| { }   | 8             |
| $\pi$ | 2, 336        |

**SYS, 226-228**

TAB, 182, 193, 201, 215, **229-233**  
 Tab, printer, 328  
 TAN, 20, 337, **234-235**  
 Tangent, 337  
**THEN, 94-99**  
 Three-voice chorus, 349  
 TI, 237  
 TI\$, 239  
 TIME, 92, **237-238**  
 TIME\$, 92, 238, **239-241**  
 Top-Down programming, 87  
 Trailer value, 43, 193, 218  
 Transparent, 254  
 Triangular waveform, 289, 291  
 Trigonometric function, 38, 211, 234, 337  
 Truth values, 331

Unlisten, 33  
 Uppercase/Graphics mode, 5, 15, 17, 23, 159, **295-298**  
 User-defined functions, 45  
 User-friendly, 17, 225  
 USR, **242-243, 323**

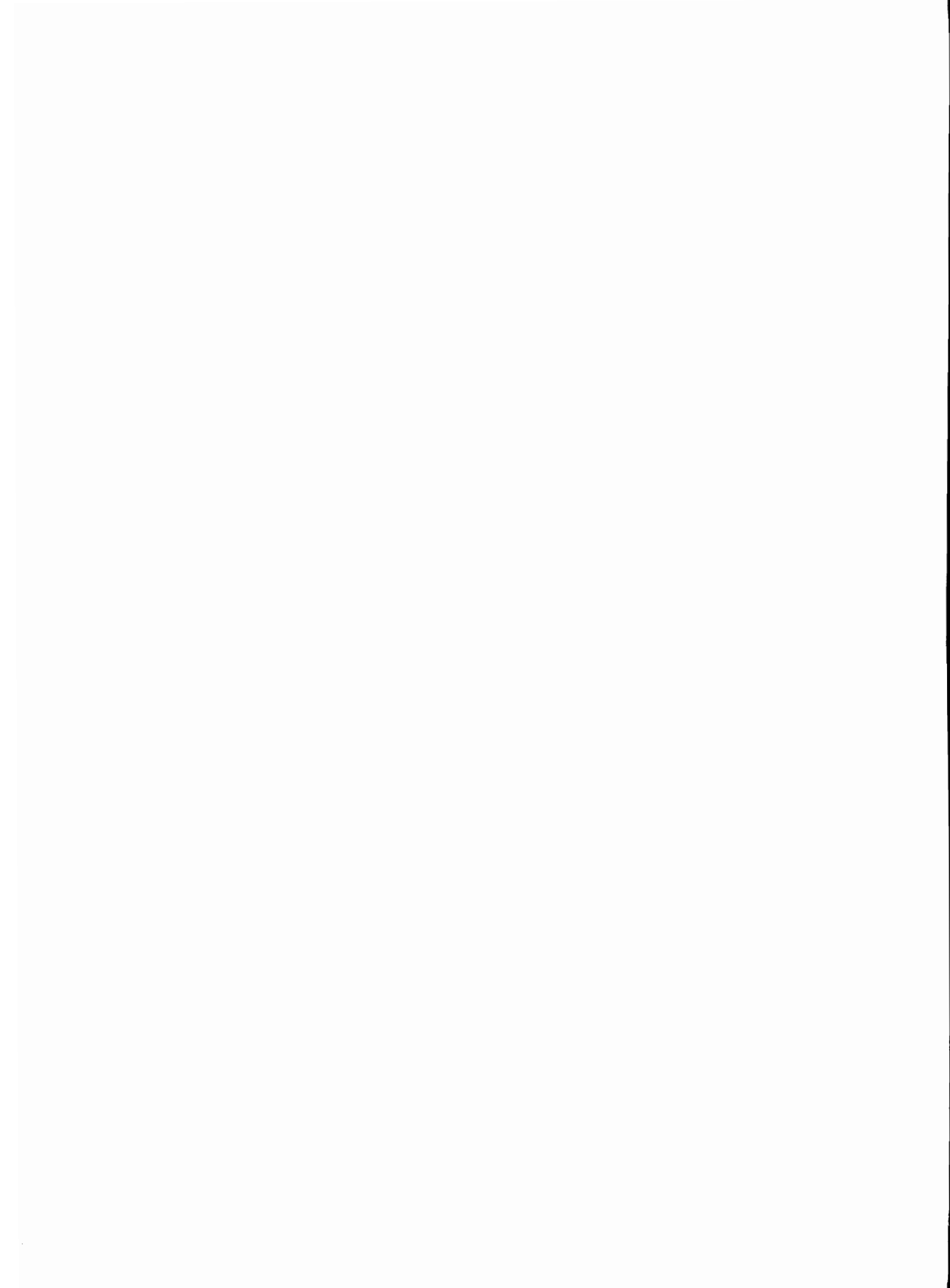


VAL, 18, 102, 124, *144*, 196, 225, **244-245**  
VALIDATE, 311  
VERIFY, **246-247**  
VIC-1541 disk drive, 9, 308  
Voices, 282, 349  
Volume, 283, 284, **285-288**

WAIT, **248-250**

Warm start, 81, 117, 160, 249  
Waveforms, 283, 284, **289-292**, 341  
Wild card, 137  
Wraparound (SPRITEs), 271, 272  
Wrapping around, 177

Xor, 249, 333



## Some Useful ASCII/Commodore Values

|       |                                                             |
|-------|-------------------------------------------------------------|
| 8     | disable SHIFT-Commodore (prevent switching character modes) |
| 9     | enable SHIFT-Commodore (allow switching character modes)    |
| 13    | RETURN                                                      |
| 14    | switch to Lowercase/Uppercase character display mode        |
| 34    | " (quotation mark)                                          |
| 48-57 | 0 to 9                                                      |
| 65-90 | unSHIFTed letters                                           |
| 141   | SHIFT-RETURN                                                |
| 142   | switch to Uppercase/Graphics character display mode         |

## Powers of Two

|     |     |      |       |
|-----|-----|------|-------|
| 2↑0 | 1   | 2↑8  | 256   |
| 2↑1 | 2   | 2↑9  | 512   |
| 2↑2 | 4   | 2↑10 | 1024  |
| 2↑3 | 8   | 2↑11 | 2048  |
| 2↑4 | 16  | 2↑12 | 4096  |
| 2↑5 | 32  | 2↑13 | 8192  |
| 2↑6 | 64  | 2↑14 | 16384 |
| 2↑7 | 128 | 2↑15 | 32768 |
|     |     | 2↑16 | 65536 |

| <u>Color</u> | <u>Keys</u> | <u>Poke Code</u> | <u>Color</u> | <u>Keys</u> | <u>Poke Code</u> |
|--------------|-------------|------------------|--------------|-------------|------------------|
| Black        | CTRL-1      | 0                | Orange       | ⌘-0         | 8                |
| White        | CTRL-2      | 1                | Brown        | ⌘-1         | 9                |
| Red          | CTRL-3      | 2                | Lt Red       | ⌘-2         | 10               |
| Cyan         | CTRL-4      | 3                | Gray 1       | ⌘-3         | 11               |
| Purple       | CTRL-5      | 4                | Gray 2       | ⌘-4         | 12               |
| Green        | CTRL-6      | 5                | Lt Green     | ⌘-5         | 13               |
| Blue         | CTRL-7      | 6                | Lt Blue      | ⌘-6         | 14               |
| Yellow       | CTRL-8      | 7                | Gray 3       | ⌘-7         | 15               |

## Some Useful Operations

Break out of infinite loop: RUN/STOP

Terminate program when it is waiting for input: RESTORE-RUN/STOP

Load and run next program from cassette: SHIFT-RUN/STOP

Load directory of disk: LOAD "\$",8

List current program on printer: OPEN 1,4:CMD 1:LIST

Prepublication reviewers say:

*"The organization is extremely good . . . it's the best instruction guide to the Commodore 64 I've ever seen!"*

*"The best book of its type . . . an excellent reference that should quickly become 'the bible' . . ."*

## **HANDBOOK OF BASIC FOR THE COMMODORE 64**

*Frederick E. Mosher and David I. Schneider*

Now—a complete and easy-to-use BASIC reference manual for the Commodore 64 that's designed for programmers of all levels! Organized alphabetically by BASIC statement, this handy guide allows you to pinpoint information without wading through entire chapters. Each statement is discussed in terms of its capabilities and limitations . . . and includes numerous examples designed to show you how each command is used!

Unlike the BASIC reference manual that accompanies the Commodore 64, this book assumes very little knowledge of BASIC—yet covers all the material from the original manual to accommodate experienced users as well!

Inside you'll find:

- Numerous examples of how each BASIC command is used
- Discussion of the subtleties and variations of each command
- Plus—applications to illustrate further uses of each function

### **CONTENTS**

Preface • Preliminary Material • BASIC Commands, Statements, and Functions • SPRITES • Music • ASCII/Commodore Values and the Character Sets • Binary Representation of Numbers • Color Codes • Disk Operating System Commands • Commodore 64 Error Messages • Data Files • Generator Program for SPRITES • Memory Allocation on the Commodore 64 • Reserved Words • 0 to 255 in Decimal and Binary • Printer Control Characters • Logical Operators • Mathematical Concepts • SPRITE POKE Summary • Music Synthesis POKE Summary • RS-232 Interface Communications • Graphics • Three-Voice Chorus Program • Music Note Scaled Frequencies • Index

ISBN 0-89303-505-X