

A SHORT TOUR OF BASIC



By the Sorcerer of



13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

A SHORT TOUR OF BASIC

By

The Sorcerer of Exidy

Computer programming languages are like political parties: each language has its supporters and detractors, and there is always a lot of debate over which ones are best. Some people like an older established one, while others prefer a newer and more progressive one.

The Sorcerer of Exidy makes no judgments on which language is best; you will decide that for yourself as you gain experience with personal computing. The Sorcerer's ROM PAC™ cartridges make it easy for you to switch from one language to another — as easy as changing the cartridge.

Your first ROM PAC is Standard BASIC. The language is simple and easy to learn, yet it has many applications, and you can use it to write complex programs.

There are many books and magazines on BASIC, from "how to" instruction manuals, to collections of useful programs, to computer games. Use this manual as your tour guide to BASIC language programming, but don't deprive yourself of further exploration in other publications.

Happy Touring!

TABLE OF CONTENTS

PREFACE	5
PASSPORTS AND VISAS	
PRELIMINARY INFORMATION	
CHAPTER 1:	9
QUICK TRIPS AND LEISURELY RAMBLES	
DIRECT AND INDIRECT MODE	
PRINT, RUN, LET, REM	
CHAPTER 2:	17
CHANGING YOUR ITINERARY	
EDITING AND CORRECTING YOUR PROGRAMS	
LIST	
CHAPTER 3:	21
IMPORTS AND EXPORTS	
INPUT AND OUTPUT OF INFORMATION	
INPUT, PRINT, DATA, READ, RESTORE	
CHAPTER 4:	27
TRAFFIC CIRCLES AND CROSSROADS	
LOOPS AND BRANCHES	
FOR . . . NEXT, IF . . . THEN, GO TO, END, STOP	
CHAPTER 5:	35
CURRENCY CONVERSIONS	
NUMERICAL EXPRESSIONS AND OPERATIONS	
ON . . . GO TO	
CHAPTER 6:	45
SCHEDULES AND TIMETABLES	
NUMERICAL ARRAYS	
DIM	
CHAPTER 7:	51
FOREIGN EXPRESSIONS	
LOGIC	
NOT, AND, OR, IF . . . THEN	

CHAPTER 8:	57
TRADING AT THE BAZAAR	
NUMERICAL FUNCTIONS	
DEF, ABS, SQR, INT, RND	
CHAPTER 9:	61
NATIVE PHRASES	
STRINGS	
LEFT\$, RIGHT\$, MID\$, STR\$, LEN	
CHAPTER 10:	71
SIDE TRIPS AND EXCURSIONS	
SUBROUTINES	
GOSUB, RETURN, ON ... GOSUB	
CHAPTER 11:	75
PACKAGED TOURS	
RECORDING AND REPLAYING PROGRAMS ON	
CASSETTE TAPE	
CSAVE, CLOAD, CSAVE *, CLOAD*	
CHAPTER 12:	79
CUSTOMS CHECK	
TIPS ON DEBUGGING	

APPENDIXES

A. LIST OF RESERVED WORDS	A-1
B. LIST OF COMMANDS AND STATEMENTS	B-1
C. LIST OF INTRINSIC FUNCTIONS	C-1
D. LIST OF ERROR MESSAGES	D-1
E. USER-DEFINED GRAPHICS	E-1
F. ONE-STROKE INSTRUCTIONS	F-1
G. LIST OF ASCII CHARACTER CODES	G-1

PREFACE: PASSPORTS AND VISAS

(Preliminary Information)



CONVENTIONS AND DEFINITIONS

Welcome aboard! You are about to take a tour of Standard BASIC. When you finish the tour, you will be able to write complex programs; yet you will find that BASIC is easy to learn, even if you have no mathematical or scientific background.

Before the tour starts, we must go through some preliminaries. First, you should read through **A GUIDED TOUR OF PERSONAL COMPUTING**. Don't be discouraged if you find sections you don't understand. There is some technical information there which you won't need at first, and some which we will cover in more detail in this manual.

Second, there are some conventions we will follow:

1. Items enclosed in angle brackets (<>) must be supplied as explained in the text.

For example, the format for an INPUT statement is:

```
INPUT <list of variables>
```

This means that you must supply a list of variables, as explained in the text.

2. Items enclosed in square brackets ([]) are optional.

For example, the RUN command has the format:

RUN [<line number>]

This means that you may supply a line number if you wish, but you don't have to. (Of course, the RUN command won't give you the same result with the line number as without.)

3. Instructions to the computer are printed in capital letters and must be typed into the keyboard exactly as shown; this includes all punctuation, and proper placement of spaces.

Also, there are some special terms you should know. Here are their definitions:

CHARACTERS: The letters, numbers, punctuation, mathematical symbols and graphical symbols, taken together, are called *characters*. The letters and numbers, taken together, are called *alphanumeric* characters.

COMMAND LEVEL: After Sorcerer prints READY, it is at the command level. This means it is ready to accept commands.

COMMANDS AND STATEMENTS: Instructions in Standard BASIC are loosely divided into two classes, Commands and Statements. Commands are instructions normally used only in direct mode (see Chapter 1). Some commands, such as CONT, may only be used in direct mode since they have no meaning as program statements. Some commands, such as NEW, are not normally used as program statements because they cause a return to command level. But most commands will find occasional use as program statements. Statements are instructions that are normally used in indirect mode. Some statements, such as DEF, may only be used in indirect mode.

CURSOR: Sorcerer places a small underline (___) on the monitor screen when it is ready to accept information. This mark is called the cursor, and it shows where the next character you type will appear.

FORMAT OF AN INSTRUCTION: The correct way to write the instruction, so that Sorcerer will understand it.

INTEGER: A whole number, positive, negative or 0.

PROMPT: When Sorcerer expects information from you, it will often place a special symbol or message in front of or above the cursor. The symbol or message is called the prompt, and it tells you what kind of information Sorcerer wants. These are the prompts you should know about:

READY— means Sorcerer is in Standard BASIC at command level, and wants a command.

> — means Sorcerer is in the Power-On Monitor, and wants a Monitor command.

? — means Sorcerer has just read an INPUT statement, and or **??** wants data (see Chapter 3).

RESERVED WORDS: Some words or combinations of letters are reserved for use as statements, commands, or intrinsic functions. You may not use these as part of the name of any variable, array, or function name.

SOME SPECIAL BUTTONS ON THE KEYBOARD

CLEAR — This wipes-off the screen, and puts the cursor in the upper left corner, but doesn't re-set Sorcerer's memory.

CTRL C — Press these two keys simultaneously to stop a program. You can re-start the program with the command **CONT**.

ESC — Press this key simultaneously with the two **RESET** keys to re-start Standard BASIC without wiping the current program out of memory.

HOME — This puts the cursor in the upper left corner of the screen, without wiping the screen display or clearing the memory.

RETURN — One of the most important keys on the board. Sorcerer will not execute any command, or put any instruction or data into its memory, until you press this key. This key corresponds to the carriage return on a typewriter.

RESET — Press both **RESET** keys simultaneously to re-start Standard BASIC and wipe the current program out of memory. Same as the command **NEW**.

RUN/STOP — Temporarily stops program execution or screen display, as long as you depress the key. Release the key, and the program or display will continue.

OTHER VERSIONS OF BASIC

There are many versions of BASIC. Standard BASIC (also called Microsoft 8K BASIC) is one of the most common, and is becoming the standard first programming language for home computer owners.

There are many books and magazines for home computer enthusiasts; in these publications, you will find BASIC programs that you will want to run on your Sorcerer. However, if they are not written in Standard BASIC, you will have to translate them.

It is not usually difficult to translate a program into Standard BASIC from another version of BASIC. Find out which version the program is written in, and study an instruction manual for that version. After mastering this manual, you should have no trouble learning the other version.

The differences between versions are often minor. Some versions use a backslash (\) instead of a colon (:) to separate two or more statements on the same line; some versions have different limitations on the names of variables.

Sometimes you may have to re-write the program slightly. For example, in most BASICs, the random number functions behave differently from Standard BASIC's function RND.

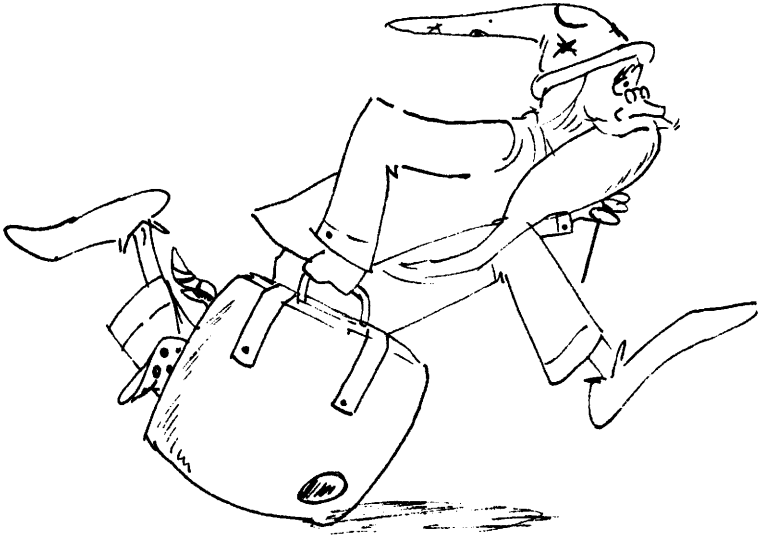
SOME SPECIAL FEATURES OF THE SORCERER

User-Defined Graphics — Sorcerer lets you create your own alphabets and symbols. You can choose which key on the keyboard you want to store your symbol under, and can call up that character whenever you wish, by pressing the **GRAPHIC** AND **SHIFT** keys, and your character's key, simultaneously. Appendix E tells you how to do this.

One-Stroke Instructions — Most of the keys on the Sorcerer's keyboard have a Standard BASIC instruction or function as an additional meaning. For example, you can enter the command CLOAD by pressing the **GRAPHIC** and **S** keys simultaneously, or the statement PRINT by hitting **?**. This is explained in Appendix F. To avoid confusion, we will not use this feature in the text.

CHAPTER 1: QUICK TRIPS AND LEISURELY RAMBLES

(Direct and Indirect Mode)



Sorcerer uses Standard BASIC in two different ways: The *direct mode* and the *indirect mode*. In the direct mode, Sorcerer executes each statement or command as you enter it on the keyboard. The results of arithmetical or logical operations will appear on the screen, and they are also saved in memory for later use. However, Sorcerer will forget each instruction after execution.

Example:

You type: PRINT 1+2

Then hit: **RETURN**

Sorcerer
replies: 3
 READY

The number 3 is now stored in memory, but the instruction PRINT 1+2 has been forgotten.

In direct mode, you can use Sorcerer like a pocket calculator.

Example:

Total this list of household expenses

rent	\$250.00
food	\$119.72
gas	\$ 36.17
electricity	\$ 8.05
telephone	<u>\$ 21.30</u>

You type: PRINT 250.00+119.72+36.17+8.05+21.30

Sorcerer
replies: 435.24
READY

When Sorcerer is prepared to accept direct mode commands, we say that it is at the **command level**. You can always tell when Sorcerer is in this condition, because that is the only time you get the READY prompt.

INDIRECT MODE

In the indirect mode, Sorcerer stores all the commands and statements of a program in its memory. It will not execute any of these instructions until you tell it to run the program. Every line of instruction must begin with a **line number**, which must be an integer from 0 to 65529. This number serves five purposes:

1. It tells the Sorcerer that the instruction is in indirect mode, rather than direct mode.
2. It tells the Sorcerer where to put the instruction in its memory.
3. It gives Sorcerer the sequence in which the instructions are to be executed.
4. It enables you to branch or skip from one part of the program to another.
5. It enables you to call up a given instruction from the memory so that you can make corrections or changes in the program.

When you write your programs, you should not number the lines consecutively (line 1, line 2, line 3 etc.). Leave an increment of 5 or 10 between the line numbers of successive lines. This will allow you to insert new lines between the existing ones, if you have to alter the program.

Example:

```
You type: 10 PRINT 1+2
           20 PRINT 3+4
```

Sorcerer stores these instructions in memory.

```
Now
type:      RUN                (Note: this is a command
                               in direct mode)
```

```
Sorcerer
replies:   3
           7
           READY
```

The program is still in memory. You can run it again, or change it if you like.

```
You type: 15 PRINT 2+3
```

This inserts a new line (line number 15) *between* lines 10 and 20.

```
Now
type:      RUN
```

```
Sorcerer
replies:   3
           5
           7
           READY
```

As you have just seen, typing the direct mode command RUN causes Sorcerer to run the program currently in memory. To clear a program out of memory, give the command NEW.

Example:

You type: NEW

Sorcerer
replies: READY

You are now ready to write a new program. First we will introduce a new instruction, the LET statement (also called an **assignment** statement).

Example:

You type: 10 LET A=1
 20 LET B=2

Here, A and B are **variables**. You will learn more about variables in Chapter 5. For now, it is enough to know that a variable is the name of a storage location in memory. The LET statement tells Sorcerer what value to assign to the variable.

The most general LET statement for numerical variables looks like this:

LET <name of variable> = <numerical expression>

Don't let the = sign confuse you. It does **not** mean "is the same as." It is just a shorthand expression for "assigned the value of."

The numerical expression can contain variables as well as numbers — even the variable whose value is being assigned.

Examples:

The assign-
ment
statement

means

LET A = 1

put the number 1 into storage location A

LET A = B + 1

find the number in storage location B, add 1 to it, and store the result in location A

LET A=A+1 find the number currently in location
A, add 1 to it, and stuff the result
back into location A

NOTE: LET A=A+1 does *not*
mean "A=A+1"

Now to continue our program:

You type 30 LET C=A+B
 40 PRINT C
 RUN

Sorcerer 3
replies: READY

You type: 50 PRINT C+1
 RUN

Sorcerer 3
replies: 4
 READY

So far, you have seen how to instruct Sorcerer to print the value of a numerical expression (such as 1+2, C, or C+1). Sorcerer will also print text and comments. The instruction for this is

PRINT "<text or comment>"

The expression inside the quote marks is called a **string**. You will learn more about strings in Chapter 9. For now, it is enough to know the string can be any combination of letters, numbers, spaces, punctuation marks, or mathematical symbols, provided that:

1. None of the characters is a quote mark (")
2. There are no more than 255 characters in all (including spaces)

Now give the command NEW to clear out the old program. Besides clearing the old program out of memory, the command NEW resets the values of all variables to zero.

You can PRINT a blank line by giving the instruction

<line number>PRINT

Example:

RUN this program:

```
10 PRINT "THIS IS A TEST"  
20 PRINT  
30 PRINT "I AM STILL TESTING"  
40 PRINT  
50 PRINT "I AM NOW TIRED OF TESTING"
```

Sorcerer will reply:

```
THIS IS A TEST  
  
I AM STILL TESTING  
  
I AM NOW TIRED OF TESTING
```

Your programs will be easier for you and for others to understand if you include remarks and explanations. Do this with the REM statement.

<line number>REM<your remarks >

Example:

```
5 REM THIS PROGRAM SHOWS HOW  
10 REM TO PRINT A BLANK LINE  
15 PRINT "BELOW THIS LINE IS A BLANK LINE"  
20 PRINT  
25 PRINT "ABOVE THIS LINE IS A BLANK LINE"
```

Sorcerer knows that a REM statement doesn't instruct it to do anything, but just contains your remarks and comments. Whenever it sees a REM statement, it skips to the next line. However, the REM statements remain in Sorcerer's memory, and you can call them back with LIST command. You will see how to do this in the next chapter.

Note: You can write more than one statement on a line if you separate the statements by colons (:). As long as there are no more than 64 characters in the line all told, it can contain as many statements as you please. They all have the same line number, and Sorcerer executes them in order from left to right, when it comes to the line. But notice that if one of the statements is a REM, any following

statements on the same line will just be considered part of the remark, and won't be executed. To keep things simple, none of the programs in this book have more than one statement per line.

[Illegible text in right margin]

CHAPTER 2: CHANGING YOUR ITINERARY

(Editing & Correcting Your Programs)



If you make a typing mistake while entering your program, you can correct it easily.

CHARACTER ERASE

Push the **SHIFT** and **RUB** keys simultaneously, to erase the last character of the current line. Push **SHIFT** **RUB** twice to erase the last two characters. By pushing **SHIFT** **RUB** repeatedly, you can erase as many characters as you wish from the end of the current line.

Example:

You intend to type: `10 PRINT 1+2`

You have actually
typed: `10 PRIMT`

Press **SHIFT** **RUB**
twice. The
result is:

```
10 PRI
```

Now continue
typing the
correct
characters:

```
10 PRINT 1+2
```

The result in
Sorcerer's
memory is:

```
10 PRINT 1+2
```

LINE ERASE OR CORRECTION

Push the **@** key to erase the entire current line. (Note: you cannot delete a line with the **@** key *after* you have hit the **RETURN** key, because hitting **RETURN** establishes a new current line.)

To correct a previous line of the program (not the current line), just type its line number, followed by the corrected instruction. To delete an old line entirely, type its line number and then press **RETURN**.

Example:

You have entered:

```
10 PRINT 1+2
20 PRITM 2+3
30 PRINT 3+4
```

to correct line
20, type:

```
20 PRINT 2+3
```

and enter it
with the **RETURN**
key

To delete line
20 entirely,
type:

```
20
```

and then push **RETURN**

LINE INSERTION

To insert a new line between two old lines, just assign it a line number between the numbers of the old lines.

Example:

You have entered: 10 PRINT 1+2
 20 PRINT 2+3
 30 PRINT 3+4

If you enter: 15 PRINT
 25 PRINT

The result in
Sorcerer's
memory will be: 10 PRINT 1+2
 15 PRINT
 20 PRINT 2+3
 25 PRINT
 30 PRINT 3+4

THE LIST COMMAND

Use the command LIST to get a listing of the program currently in Sorcerer's memory. Sorcerer will list all the instructions in line-number order, until it reaches the end of the program.

Example:

You enter: 10 PRINT "TEST PROGRAM"
 20 PRINT 1+1
 30 PRINT 2+2
 15 PRINT
 25 PRINT
 40 PRINT "END OF PROGRAM"
 35 PRINT
 LIST

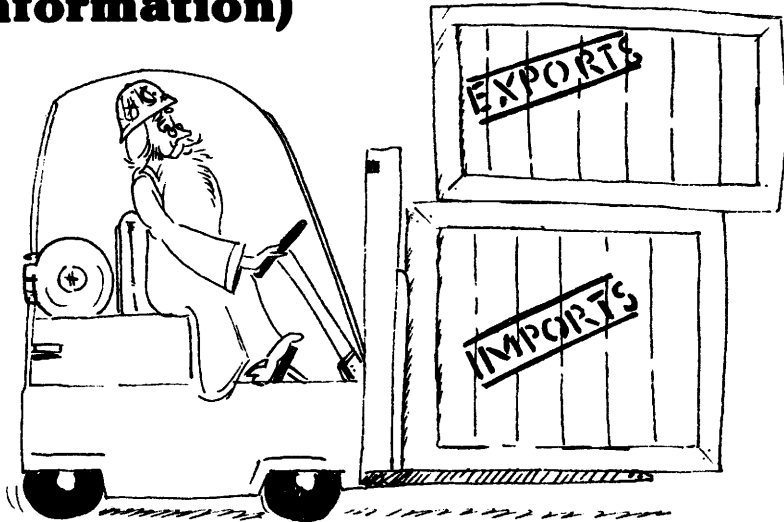
Sorcerer's
reply:

```
10 PRINT "TEST PROGRAM"  
15 PRINT  
20 PRINT 1+1  
25 PRINT  
30 PRINT 2+2  
35 PRINT  
40 PRINT "END OF PROGRAM"
```

You can make Sorcerer pause in the middle of the listing by pressing **RUN/STOP**. Release **RUN/STOP**, and the listing will continue. To stop the listing completely, push the **CTRL** and **C** keys, simultaneously.

CHAPTER 3: IMPORTS AND EXPORTS

(Input and Output Information)



This chapter will teach you how to put information into a program while it is running, and how to get information out of the program.

THE INPUT STATEMENT

The **INPUT** statement requests information from the keyboard. When Sorcerer executes an **INPUT** instruction, it prints a question mark (?) on the screen, to tell you it is waiting for information. This question mark is called the **INPUT prompt**. Sorcerer will not continue the program until you give it the data it wants. If you don't give it as much information as requested, Sorcerer will print ?? and wait for more information. If you give it too much information, it will give the message "EXTRA IGNORED" and will continue the program. If you give the wrong kind of data (a string instead of a number) you will get a SN error message (SYNTAX ERROR) and the program will stop.

The simplest **INPUT** statements look like this:

```
INPUT <list of variables>
```

The variables in the **INPUT** list must be separated by commas.

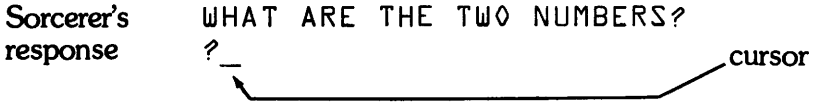
Example:

```

10 REM SUM OF TWO NUMBERS
20 PRINT "WHAT ARE THE TWO NUMBERS?"
30 INPUT A,B
40 PRINT
50 PRINT "THE SUM IS"
60 PRINT A+B

```

Now RUN this program:



Next type two numbers separated by a comma

```

You type:      3.5, 7.5

Sorcerer's response: THE SUM IS
                    11

```

You can make Sorcerer print a message in front of the INPUT prompt. Use this format for your instruction:

```
INPUT"<prompt string>";<variable list>
```

Replace lines 20 and 30 in the program above with a new line 20:

```
20 INPUT "WHAT ARE THE TWO NUMBERS"; A,B
```

Now when you run the program, Sorcerer's response will be

```
WHAT ARE THE TWO NUMBERS?_ ← cursor
```

MORE ABOUT PRINT STATEMENTS

You can make Sorcerer print several different things on the same line of the monitor screen. The instruction is:

```
PRINT<list of strings or numerical expressions>
```

You must separate the items in the list by commas (,) or semicolons (;).

Examples:

```
27 PRINT "THE ANSWER IS"; 27.5*X, "NOT"; Y
```

```
42 PRINT "3+4" ; 3+4
```

Sorcerer divides each line of printing into zones 14 characters long. If you separate two items in the PRINT list by a comma, Sorcerer will print the first item, and move to the next zone before printing the second item. If you separate two items by a semicolon, Sorcerer will print the second immediately after the first.

Examples:

```
You type: PRINT "A";"B", "C"
```

```
Sorcerer  
replies: AB C
```

```
You type: PRINT "A", "B"; "C"
```

```
Sorcerer  
replies: A BC
```

If you put a comma or semicolon at the end of a PRINT list, the next PRINT instruction will print on the same line, either in the next 14-character zone (if a comma) or right after the end of the first PRINT (if a semicolon).

Example:

```
10 PRINT "A";
```

```
20 PRINT "B",
```

```
30 PRINT "C"
```

Run this program and Sorcerer will print:

```
AB C
```

There is a special tabulator function that works like the tab key on a typewriter.

Examples:

You type: PRINT "ABC";TAB(10);"DEF"

Sorcerer
replies: ABC DEF

You type: PRINT TAB(5); "****"

Sorcerer
replies: ***

If you want Sorcerer to space over to a certain space before printing an item in a PRINT list, put

TAB (<number of the space>)

in front of the item in the PRINT instruction. The spaces are numbered 1 through 64, from left to right on the screen. Sorcerer won't backspace on TAB — if it has already gone past the space you called for, it will simply ignore the TAB.

DATA, READ, AND RESTORE STATEMENTS

So far, you have fed data into your programs with INPUT statements (which tell the Sorcerer to ask you for information) and LET statements (which assign values to variables, one at a time). You can also write lists of data directly into your programs, and instruct Sorcerer to read data from the lists.

A DATA statement looks like this:

DATA <list of constants>

The numbers in the list must be separated by commas.

Example:

25 DATA 27.5,3.2,-10

When Sorcerer sees a DATA statement, it stores the numbers in memory. You can retrieve these numbers with a READ statement:

```
READ <list of variables >
```

Again, the variables in the list must be separated by commas.

Example:

```
47 READ X,F,Y
```

When Sorcerer sees a READ statement, it assigns values from the DATA lists to the variables in the READ list. The assignments are made from left to right until the READ list is exhausted. If there are more variables in the READ list than numbers in the DATA lists, you will get an OD error message (OUT OF DATA). If there are more numbers in the DATA list than variables in the READ list, then the next READ instruction will begin at the first unassigned DATA item.

Example:

You type:

```
10 DATA 1,2,3,4
20 READ A,B,C
30 READ D
RUN
```

Sorcerer makes these assignments:

```
A=1 B=2 C=3 D=4
```

A single READ statement may assign numbers from several DATA lists, and one DATA list may supply values for several READ statements.

Example:

You type:

```
10 DATA 1,2
20 DATA 3,4
30 READ A,B,C,D
40 PRINT A,B,C,D
RUN
```

Sorcerer
replies:

```
1           2           3           4
```

Sorcerer has assigned variables A and B the values in the first DATA list, assigned C and D the values in the second DATA list, and printed the values of A, B, C, and D on the same line, in different 14-character zones.

Use the RESTORE statement to re-set the DATA list. The RESTORE statement tells Sorcerer that the next READ statement should start at the first item of the first DATA list, rather than at the first unused piece of DATA.

Example:

You type:

```
10 DATA 1,2,3,4,5
20 READ A,B,C
30 RESTORE
40 READ D,E
RUN
```

Sorcerer makes these assignments:

A=1 B=2 C=3 D=1 E=2

CHAPTER 4: TRAFFIC CIRCLES & CROSSROADS

(Loops & Branches)



A **loop** is a section of a program that tells Sorcerer to repeat an action a certain number of times. A **branch** is a place in a program where Sorcerer must make a decision, and act according to that decision.

THE FOR . . . NEXT LOOP

The FOR . . . NEXT loop starts with a FOR statement and ends with a NEXT statement. In between, you can write any instructions you like — even another FOR . . . NEXT loop.

The simplest FOR statement has this form:

```
FOR <variable> = <first value> TO <final value>
```

The NEXT statement is written;

```
NEXT <variable>
```

or more simply,

```
NEXT
```

Examples:

```
10 FOR X=1 TO 10    This loop prints the integers from
20 PRINT X          1 to 10
30 NEXT
```

```
100 FOR Y=1 TO 10   This loop prints the letter Y
110 PRINT "Y"       10 times
120 NEXT Y
```

```
61 FOR A=-10 TO 5   This loop prints 16 blank lines
62 PRINT
63 NEXT
```

When Sorcerer sees a FOR instruction, it assigns the *first value* to the variable. It then steps through each of the following instructions until it reaches the last statement of the loop (the NEXT statement). It then jumps back to the FOR statement, increases the value of the variable by 1, and compares the result to the *final value*. If the new value is more than the final value, Sorcerer skips to the instruction after the NEXT statement. Otherwise, it goes through the instructions again. Eventually the variable will become larger than the final value, and the loop will end.

Example:

```
Program: 100 FOR X=1 TO 3
          200 PRINT X
          300 NEXT X
          400 PRINT "END"
```

Sorcerer's actions:

1. Set X to 1
2. PRINT the value of X (X=1)
3. Go back to line 100
4. Set X to 2
5. 2 is not larger than 3, so PRINT the value of X (X=2)
6. Go back to line 100
7. Set X to 3
8. 3 is not larger than 3, so PRINT the value of X (X=3)
9. Go back to line 100
10. Set X to 4
11. 4 is larger than 3, so go to line 400

The first and final values do not have to be specific numbers; you can use numerical expressions if you wish. When Sorcerer first sees the FOR instruction, it will compute the values of the expressions and use those values as the first and final values for the loop variable.

Examples:

```
25 FOR X=C TO 100
30 PRINT X+C
35 NEXT X
```

```
105 FOR A=X TO Y
110 PRINT "A"
115 NEXT
```

You can also have Sorcerer increase the FOR variable by more than 1 unit at a time, or even decrease it. To do so, use this FOR statement:

```
FOR<variable> = <first value> TO <final value> STEP <step value>
```

The <step value> can be a number, positive or negative, or a numerical expression.

Examples:

```
10 FOR Z=1 TO 100 STEP 2
20 PRINT Z
30 NEXT
```

This program prints all the odd numbers (skipping the even ones) from 1 to 100

```
110 FOR A=10 TO 1 STEP -1
120 PRINT A
130 NEXT A
```

This prints the numbers 1 to 10, in decreasing order

Try running this program:

```
10 PRINT "COUNTDOWN"
20 PRINT
30 FOR X=10 TO 1 STEP -1
40 PRINT X
50 PRINT
60 NEXT X
70 PRINT "BLASTOFF!"
```


Now insert these lines into the COUNTDOWN program:

```
52 FOR Y=1 TO 130
54 NEXT Y
```

These instructions force Sorcerer to count from 1 to 130 during each pass through the FOR X loop — this slows down the program, so that the printed numbers move more slowly across the screen (about 1 second between counts).

If you put a FOR . . . NEXT loop inside another FOR . . . NEXT loop, you should use the longer form of the NEXT statement:

```
NEXT <variable>
```

Otherwise Sorcerer will be confused about which NEXT statement ends which loop.

You can sometimes use a single NEXT statement to end two or more loops. For example, in the COUNTDOWN program you could replace

```
54 NEXT Y
60 NEXT X
```

with

```
60 NEXT Y,X
```

However, as a beginning programmer, you should use a separate NEXT statement to end each FOR . . . NEXT loop. That way you will avoid confusing Sorcerer (and yourself).

THE NUMERICAL IF . . . THEN STATEMENT

Let's play a game. You think of an integer from 1 to 10 and Sorcerer will guess what your number is.

```
10 REM GUESSING GAME
20 PRINT "PICK A NUMBER FROM 1 TO 10"
30 PRINT "I WILL GUESS YOUR NUMBER"
40 PRINT "IF I GUESS CORRECTLY, TYPE 1"
50 PRINT "IF I GUESS INCORRECTLY, TYPE 2"
60 FOR X=1 TO 10
70 PRINT "IS YOUR NUMBER";X;"?"
80 INPUT Y
90 IF Y=2 THEN 110
```

```
95 PRINT
100 NEXT X
110 PRINT "I KNEW IT ALL ALONG"
```

Note: Be careful not to type $Y=2$ in line 90.

The IF . . . THEN instruction at line 90 tells Sorcerer to decide whether or not a certain arithmetical expression ($Y-2$) has value 0. If it does not, Sorcerer must skip to a certain line (line 110). If the value is 0, Sorcerer must go to the next line (line 95).

The general format for this kind of numerical IF . . . THEN statement is:

IF <expression> THEN <line number>

When Sorcerer comes to this instruction, it evaluates the numerical expression. If the expression has value 0, Sorcerer just moves on to the next instruction. But if the expression has any value other than 0, it skips to the instruction numbered <line number> and continues the program from that point.

Example:

```
10 INPUT A,B
20 IF A*A + B*B THEN 40
30 PRINT "BOTH A AND B ARE ZERO"
40 PRINT "END OF PROGRAM"
```

If you enter 0 for both A and B, then $A * A + B * B$ will have value 0, so that Sorcerer will go on to the next instruction (line 30) and print "BOTH A AND B ARE ZERO," before going on to line 40. Otherwise, Sorcerer skips past line 30 and prints "END OF PROGRAM."

Another kind of numerical IF . . . THEN statement looks like this:

IF <expression> THEN <statement>

The <statement> can be any Standard BASIC program statement.

When Sorcerer comes to this IF . . . THEN instruction, it evaluates the <expression>, and moves to the next instruction if the value is 0.

But if the value is not 0, it executes the <statement>, and then goes on to the following instruction.

Example:

```
10 INPUT X,Y
20 IF X-Y THEN PRINT "X AND Y ARE DIFFERENT"
30 PRINT "END OF PROGRAM"
```

If you enter two different numbers for X and Y, Sorcerer will print "X AND Y ARE DIFFERENT"; otherwise it will not. In either case, it will then go on to the next instruction, and print "END OF PROGRAM".

Notice that the first kind of numerical IF . . . THEN *may* make Sorcerer skip over one or more lines of the program, while the second kind never does.

THE GO TO STATEMENT

This statement tells Sorcerer to go directly to a certain line in the program (without passing Go, and without collecting \$200). The format is:

GO TO <line number>

(you can also write it GOTO, without the space between words).

As an example, let's use a GO TO statement to make our guessing game program more sophisticated. Add these lines:

```
120 PRINT
130 GO TO 20
```

When Sorcerer gets to line 130, it jumps right back to line 20 and starts the guessing game again. It will continue the game forever, on an endless loop, until you stop the program with **CTRL C**.

You can use IF . . . THEN and GO TO statements to make Sorcerer skip to any line of a program — even to a REM instruction.

END AND STOP

Suppose you cheat at the guessing game. You might pick a number outside the range 1 to 10, or you might lie when Sorcerer guesses your number. If you cheat, Sorcerer will go through each of the numbers 1 to 10 and not get a 1 response from you on any of them.

Add these lines to the program:

```
102 PRINT "YOU CHEATED. I DON'T PLAY"
```

```
104 PRINT "WITH CHEATERS."
```

```
106 STOP
```

The STOP instruction tells Sorcerer to stop the program, even though it hasn't come to the last instruction. This is important because when you write branching instructions into a program, Sorcerer may never get to the last instruction.

We could get the same effect by replacing line 106 (the STOP instruction) with

```
106 GO TO 140
```

and adding

```
140 END
```

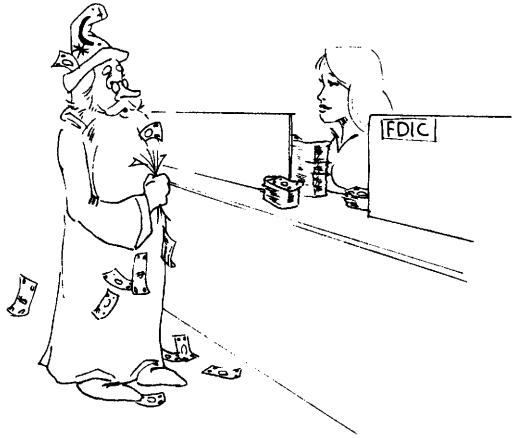
The END instruction tells Sorcerer that it has reached the end of the program and should therefore stop. We don't usually write END statements in this manual, because Sorcerer always assumes there is one tacked to the end of your program.

Notice that in the guessing game, the normal branch of the program (the non-cheater's branch) never reaches the END statement, because of the GO TO instruction at line 130.

A STOP statement can come anywhere in your program; when Sorcerer comes to a STOP, it will print a message telling you at what line it has stopped. The program and all variable values remain in memory, and you can continue the program with the command CONT (see Appendix B). An END statement should occur only as the last line in a program.

CHAPTER 5: CURRENCY CONVERSIONS

(Numerical Expressions and Operations)



FLOATING POINT NUMBERS

In Standard BASIC, Sorcerer handles all numbers as decimals. Even the integers (whole numbers) are considered to be decimal numbers with zero fractional parts.

Example:

The integer	is handled as
1	1.0000
0	0.00000
-3	-3.00000

Ordinary decimal notation is sometimes called **fixed point** notation because we consider the decimal point fixed, and place the digits around the point, according to the size of the number.

Examples of numbers in fixed point notation:

- 1.23456
- 1.0506
- 0.0
- 238.5
- 0.000123
- 4560.0

Sorcerer handles all decimal numbers in a different way, called **floating point** notation. In this notation the digits of a number are written in fixed places, and the decimal point is moved around the digits, according to the size of the number.

Examples:

This fixed point number	is written this way in floating point notation	to convert from floating to fixed point, move the decimal point
1.23	1.23 E 00	zero places
123.0	1.23 E 2	2 places to the right
-74.5	-7.45 E 1	1 place to the right
0.07324	7.324 E -2	2 places to the left
-0.003	-3.0 E -3	3 places to the left

The E notation tells Sorcerer how many places to move the decimal point to the left or to the right, to convert a floating point number to fixed point notation. Moving the point to the right is the same as multiplying by positive powers of 10, and moving the point to the left is the same as multiplying by negative powers of 10.

Examples:

$$\begin{array}{lclcl} 1.23 & = & 1.23 \text{ E } 0 & = & 1.23 \times 1 & = & 1.23 \times 10^0 \\ 123.0 & = & 1.23 \text{ E } 2 & = & 1.23 \times 100 & = & 1.23 \times 10^2 \\ -74.5 & = & -7.45 \text{ E } 1 & = & -7.45 \times 10 & = & -7.45 \times 10^1 \\ 0.07324 & = & 7.324 \text{ E } -2 & = & 7.324 \times 1/100 & = & 7.324 \times 10^{-2} \\ -0.003 & = & -3.0 \text{ E } -3 & = & -3.0 \times 1/1000 & = & -3.0 \times 10^{-3} \end{array}$$

The first (left-most) non-zero digit of a fixed point number (which is its first digit in floating point notation) is called the **first significant digit**. Standard BASIC only pays attention to the first six significant digits (that is, the first non-zero digit, and the next five digits). If you feed in a number with fewer than six significant digits, Sorcerer will fill in enough zeros to make six digits. If you feed in a number with more than six significant digits, Sorcerer will round off to six.

Examples:

You enter this number	Sorcerer uses this number
1.23	1.23000 E 00
91.82736	9.18274 E 01
123.0	1.23000 E 02
-74.5	-7.45000 E-02
0.07654321	7.65432 E-02
-0.003	-3.00000 E-03

The largest number that Standard BASIC will accept is 1.70141 E 38. The smallest positive number is 2.93874 E-39.

Though Sorcerer handles all numbers in floating point notation, it does not print all numbers that way.

1. Negative numbers are printed with a minus sign (-) before the first digit. Positive numbers are printed with a space (**not** a + sign) before the first digit.

2. Whole numbers (i.e. zero fractional part) between -999999 and 999999 are printed as integers (no decimal point).
3. Non-integer numbers between $.1$ and 999999 or between $-.1$ and -999999 are printed in fixed point notation (no E notation).
4. All other numbers are printed in floating point notation.

Put this program on the Sorcerer:

```

10 REM THIS PROGRAM SHOWS HOW SORCERER PRINTS
20 REM NUMBERS IN STANDARD BASIC
30 INPUT "WHAT IS YOUR NUMBER";X
40 PRINT
50 PRINT "I PRINT YOUR NUMBER AS";X
60 PRINT
70 GO TO 20

```

Now run the program and try as many different numbers as you can think of.

Example:

```

Sorcerer:      WHAT IS YOUR NUMBER?
You:           .5
Sorcerer:      I PRINT YOUR NUMBER AS .5
                WHAT IS YOUR NUMBER?

```

Try these numbers on the program.

+1	-1
6523	-23.460
1 E 20	-12.345 E-7
1.234567 E-10	1000000
999999	.1
.01	.000123
	38

When you get tired of the program, stop it by pressing **CTRL C**

NUMERICAL CONSTANTS AND VARIABLES

A **numerical variable** is a mathematical quantity which can be assigned different number values, but has no definite value until one is assigned. A **numerical constant** is a definite number, such as 17, 34.5, $-2\frac{1}{2}$, 3.000000 E -6, and so forth.

When you use a variable in a program, Sorcerer finds an empty storage location in memory and assigns your variable as a name for that location. If you use the word OSCAR as a variable, Sorcerer will find a storage location and call it OSCAR. When you assign a value to the variable with a LET statement, Sorcerer will put the proper number into the location named OSCAR.

You can use any combination of letters and numbers as a numerical variable, provided that:

1. The first character of the name is a letter.
2. No reserved words appear in the name.

Example:

You can use these
as variables

A1

OSCAR

OSCAR1

RUM

You cannot use
these

1A first character is
a number

TOM contains the reserved
word TO

ATOM contains the reserved
word TO

RUN contains the reserved
word RUN

Sorcerer only pays attention to the first two characters of a variable, so make certain no two variables in your program have the same first two characters.

Example:

Sorcerer cannot
tell the difference
between

and

OSCAR

OSWALD

A7

A71

JAB

JA2

NUMERICAL OPERATIONS

Standard BASIC uses the ordinary arithmetical operations (addition, subtraction, multiplication, and division) together with the operations of **negation** (changing the sign of a number) and **exponentiation** (raising a number to a power).

To add or subtract two numbers, use the + and - signs on the main keyboard or the 16-key pad. To multiply two numbers, use the x sign on the 16-key pad or the * sign (asterisk) on the main keyboard. To divide two numbers, use the ÷ sign on the 16-key pad or the / sign (slash) on the main keyboard.

When you use the division operator in an instruction, be sure that the divisor (the expression you are dividing **by**) does not have the value 0 at the time the instruction is executed. If this happens, your program will stop, and you will get a /0 error message.

Use parentheses to group your expressions.

Examples:

$(1 + A)/2$ means add 1 to A, then divide by 2

$1 + A/2$ means divide A by 2, then add 1

To **negate** an expression, put a minus sign (-) in front of it. (Negation just changes the sign of a number from positive to negative, or from negative to positive.)

Examples:

$-(-1)$	changes	the -1 to 1
$-(A+B)$	means:	add A and B , and then change the sign of the result

For exponentiation, use the \wedge sign on the main keyboard.

Examples:

$A \wedge 2$	means	A^2 (that is, $A \times A$)
$2 \wedge 0.5$	means	$2^{0.5}$
$B \wedge (-A)$	means	B^{-A}
$2 \wedge A$	means	2^A

Remember that exponentiation by a positive integer is just repeated multiplication, while exponentiation by a negative integer is just repeated division.

Examples:

$A \wedge 3$	is just	$A \times A \times A$
$(-B) \wedge 2$	is just	$(-B) \times (-B)$
$B \wedge (-2)$	is just	$1/(B \times B)$
$10 \wedge (-3)$	is just	$1/(10 \times 10 \times 10)$ that is $1/1000$ or $.001$

Standard BASIC interprets exponentiation as the inverse of a logarithm function. Therefore, Sorcerer will compute exponentiation by an integer more quickly than it would compute the equivalent repeated multiplication.

Any number to the power 0 is just 1 (that is, $\text{NUMBER} \wedge 0$ is just 1). If you try to raise 0 to a negative power ($0 \wedge (-1)$, say) you will get a /0 error message.

ORDER OF PRECEDENCE

Even fairly simple numerical expressions may contain several operators. When you compute the value of an expression, the result you get often depends on which operations you perform first.

Example:

Divide 3 by 2 and add 1 $3 / 2 + 1$

— if you divide 3 by 2 first, and then add 1, the result is
 $1.5 + 1$, or 2.5 i.e. $(3/2) + 1$

— if you add 2 and 1 first, and then divide into 3, the result is
 1 or $3 / (2 + 1)$ i.e. $3/3$

To avoid this kind of confusion, Sorcerer performs numerical operations in a special order of precedence. This insures that each numerical expression has a single, well-defined value (when values are assigned to all variables in the expression).

The operations are done in this order:

1. Expressions inside parentheses ().
2. \wedge Exponentiation
3. $-$ Negation
4. $*$ and $/$ Multiplication and Division
5. $+$ and $-$ Addition and Subtraction

When two or more operations in an expression have the same precedence, Sorcerer will do them in order from left to right.

Examples:

$3 * (1 + 4)$ first add $1 + 4$; then multiply by 3

$3 * 1 + 4$ first multiply $3 * 1$; then add 4

$-2 + 3 \wedge 4$ raise 3 to the power 4; then add -2

THE CONDITIONAL GO TO

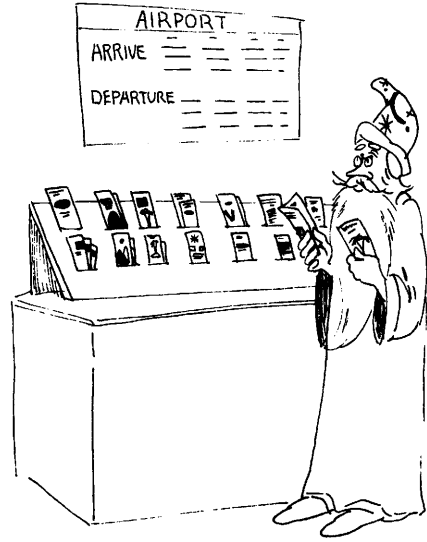
The conditional GO TO statement gives you a way to branch a program into several different paths, instead of just two (as with the IF . . . THEN statement). The format is:

ON <numerical expression> GO TO <list of line numbers>

When Sorcerer sees this instruction, it evaluates the expression and throws away the fractional part (if any) of the result. This produces an integer, say J. Now Sorcerer skips to the J-th line number on the list. If J has value 0 or is greater than the number of items on the list, Sorcerer goes to the next instruction after the ON . . . GO TO. If J is less than 0 or greater than 255, you will get an FC error message (ILLEGAL FUNCTION CALL).

CHAPTER 6: SCHEDULES AND TIMETABLES

(Numerical Arrays)



Sometimes it is useful to give a single name to an entire list of numbers, and treat the list as one thing. You do this by using **numerical arrays**. A numerical array is the name of a block of storage locations, just as a numerical variable is the name of a single storage location.

The simplest kind of array is the **linear array**. Think of a linear array as a list of numbers arranged in a line. Each **element**, or position in the line, has an **index number**. The first position (element) has index 0, the second has index 1, and so on. You can refer to a number stored in the array by giving the array name and the location index.

Example:

Suppose array A consists of the numbers

1.1, 1.2, 1.3, 1.4, 1.5

A(0) is 1.1, the first element (index number 0)

A(3) is 1.4, the fourth element (index number 3)

You can use numerical variables or other numerical expressions as indexes for arrays. For instance, when Sorcerer sees OSCAR(I), it first looks at the number in storage location I; this gives it an index number and specifies which part of array OSCAR to look at.

Example:

```
100 FOR X=0 TO 10
110 PRINT OSCAR(X)
120 NEXT X
```

This program prints the first 11 elements of the array OSCAR, in order.

The most general format for an array element is:

<Array name>(<numerical expression>)

Numerical array names obey the same rules as the names of numerical variables: the first character must be a letter, and the name cannot contain any reserved words.

How does Sorcerer tell the difference between a variable and an array, if their names are chosen in the same way? When you refer to an element of the array, you use the array name followed by the element's index in parentheses — variables don't have parentheses. But the best way to tell Sorcerer that your array really is an array is with a **dimension statement**. The format is:

DIM <array name>(<highest index number>)

Example:

```
10 DIM OSCAR (17)
```

This tells Sorcerer that OSCAR is a linear array with 18 elements, indexes 0 through 17.

You can have as many elements in an array as you want, provided there is enough memory available to store them all. If you try to dimension an array too large, you will get an OM error message (OUT OF MEMORY).

You can also use a variable or other numerical expression to dimension an array, so that the size of the array will depend on the value of the expression.

Example:

```
10 DIM OSWALD (2*YEARS + 6)
```

But remember that the value of the expression must be an integer, and cannot be less than 0, or you will get an FC error message (ILLEGAL FUNCTION CALL).

You can also dimension two or more arrays with the same DIM statement.

Example:

```
27 DIM A(17), B(X), OSCAR(Y+3)
```

If Sorcerer sees an array before it sees a DIM statement for that array, it will assume the array has 11 elements (that is, the index runs from 0 to 10). So Sorcerer always knows how large an array is supposed to be — either you tell it (with a DIM statement), or else it assumes 11 elements.

Therefore, you have to be careful about calling for array elements. Remember that OSCAR(K) is the element of OSCAR whose index is K (whatever K might happen to be), just as OSCAR(3) is the element with index 3. If you ask for an element that doesn't exist, you are asking for trouble.

Example:

```
10 DIM OSCAR(5)
20 FOR X=0 TO 10
30 LET OSCAR(X)=X
40 NEXT X
```

When you run this program, Sorcerer will assign values to the elements of OSCAR until it gets to X=6; it will then freak out. You are asking it to do something to OSCAR(6), but in line 10 you said that there weren't any OSCARs past OSCAR(5). The result is a BS error message (SUBSCRIPT OUT OF RANGE).

HIGHER DIMENSIONAL ARRAYS

Look at a table of numbers in an almanac. We can think of this table as an array of elements with two sets of indexes — one index tells which row of the table an element is in, and the other index tells which column.

Example:

Table AA

1.2	1.3	1.4	row #0 (first row)
1.5	1.6	1.7	row #1 (second row)
column #0 (first column)	column #1 (second column)	column #2 (third column)	

If we define the first index to be the number of the row and the second index to be the number of the column, then the element with index numbers 0 and 1 (written AA(0,1)) is just 1.3. The element with first index 1 and second index 0 (written AA(1,0)) is 1.5, and so forth.

This kind of array is called a **two-dimensional array**, because it has two indexes. If we stacked several pages of almanac tables, we could form an array with three sets of indexes — one to tell which row an element was in, one to tell which column, and one to tell which page. We call this a **three-dimensional array**. Similarly, we could have an array with as many different indexes as we wished — the number of different indexes is just the **dimension** of the array.

In Standard BASIC, you can have as many as 255 different index sets for an array, and each index can run up to whatever you please. But remember that all the elements of an array have to fit somewhere into

Sorcerer's memory. If you have too many dimensions, or if the indexes run to very large values, you will run out of storage locations and get an OM error message (OUT OF MEMORY).

By the way, you are probably wondering why all the indexes start with number 0 rather than number 1 (i.e., why is the first element of a linear array given sequence number 0?). This is because mathematicians and scientists are in the habit of numbering arrays in this way. Even though Standard BASIC was designed so that non-technical people could easily learn it, it was still written by computer scientists — old habits die hard.

MORE ABOUT DIM

Higher dimensional arrays should have DIM statements, just as linear arrays should. The DIM statement not only tells Sorcerer that a certain thing is an array, and what the highest values are for its indexes, but also how many dimensions it has.

Example:

```
10 DIM OSWALD (7,4)
```

This tells Sorcerer that OSWALD is a two-dimensional array whose first index runs from 0 to 7, and whose second index runs from 0 to 4.

You can use one DIM statement to dimension several arrays. Just separate the array declarations with commas.

Examples:

```
44 DIM A(17), B3(2,5), AL(3)
```

```
12 DIM X1(5), X2(3,3), OSWALD(1,2,3,4)
```

If Sorcerer runs across an array that has not been declared in a previous DIM statement, it will assume that all the indexes run from 0 to 10. If you declare an array to have some dimension (using a DIM statement) and then use the same name for an array with a different dimension, you will get a BS or DD error message. You will also get a BS error if you call for an array element with an index value outside the declared range.

Example:

```
10 DIM OSWALD(1,2,3,4)
```

```
20 LET OSWALD(2,2,2,2)=1 ← This gives you a BS  
error, because the first  
index of OSWALD  
only goes up to 1.
```

```
RUN
```

CHAPTER 7: FOREIGN EXPRESSIONS

(Logic)



NUMERICAL RELATIONS

A *numerical relation* is a statement that compares two numerical expressions. The relation is either true or false, and Sorcerer can always tell which, just by evaluating the expressions and comparing the resulting numbers.

Examples:

$1 = 1$	"1 equals 1"	This is true.
$1 = 0$	"1 equals 0"	This is false.
$X = 6$	"X equals 6"	This is true if X has the value of 6. Otherwise it is false.
$A + B = C$	"A + B equals C"	This is true if the value of A plus the value of B equals the value of C. Otherwise it is false.

Numerical relations have this form:

< first expression > Relation Operator < second expression >

The relation operators are:

=	equal	(examples: $1 = 1$ is true, $1 = 0$ is false)
< >	not equal	(examples: $2 < > 3$ is true, $2 < > 2$ is false)
<	less than	(examples: $2 < 4$ is true, $0 < -2$ is false)
>	greater than	(examples: $1 > -1$ is true, $2 > 3$ is false)
< =	less than or equal ¹	(examples: $1 < = 1$ and $6 < = 7$ are true, $1 < = 0$ is false)
= >	greater than or equal ²	(examples: $-1 = > -1$ and $-1 = > -2$ are true, $0 = > 1$ is false)

¹(This is also written = <)

²(This is also written > =)

LOGICAL OPERATORS

A numerical expression is a mathematical creature that always has a number value. Similarly, a logical expression is a creature that always has a logical value (sometimes called a **truth value**). Numerical expressions can have any number for their values, but logical expressions can only take on two different logical values: true and false.

Numerical relations are simple examples of logical expressions. Logical operators let you build more complicated expressions out of simple ones. There are three of these operators:

NOT < simpler expression >

< first simpler expression > AND < second simpler expression >

< first simpler expression > OR < second simpler expression >

The NOT expression is always true when the simpler expression is false, and vice versa.

Example:

NOT $X=1$ is true if X does **not** have the value 1, and is false if X **does** have value 1.

NOT (it is raining) is true if it is **not** raining, and is false if it **is** raining.

The AND expression is true in case **both** of the simpler expressions are true. If either or both of them are false, then the AND expression is false.

Examples:

$X=1$ AND $A=B$ is true just in case X has value 1 **and** A has the same value as B; otherwise it is false.

(it is raining) AND (your feet are wet)

is true just in case it is raining and your feet are wet. If it is not raining, or your feet are dry (or both) the expression is false.

The OR expression is true if either of the simpler expressions is true, or if they are both true. It is only false if **both** of the simpler expressions are false.

Examples:

$X < 1$ OR $X = 1$ is true if X is less than 1 or if X is equal to 1. It is false if X is greater than 1. This is just the same as the relation $X \leq 1$.

$X = 1$ OR $Y = 2$ is true if X has the value 1, or if Y has the value 2, or both. Otherwise it is false.

Note: The OR operation is sometimes called **inclusive OR**, because it includes as true the case where both simpler expressions are true.

There is another logical operator called **exclusive OR**, which Standard BASIC does not recognize. An exclusive OR expression is true if either of the simpler expressions are true, but not if both are.

Many people think of the English word “or” as an exclusive OR operator. Don’t be confused by this habit; an OR expression in Standard BASIC **is** true if both of its parts are true.

MORE ABOUT PRECEDENCE

Before Sorcerer can decide whether a logical expression is true or false, it must decide whether the numerical relations in the expression are true or false. Before it can decide whether a numerical relation is true or false, it must compute the values of the numerical expressions in the relation.

In Chapter 5 you learned the order of precedence for numerical operations. The numerical relations and logical operators are part of this order of precedence. This is the complete sequence:

1. Expressions inside parentheses ()
2. \wedge Exponentiation
3. $-$ Negation
4. $*$ and $/$ Multiplication and division
5. $+$ and $-$ Addition and Subtraction
6. Numerical Relations (all have the same precedence)

= Equal

< > Not Equal

< Less Than

> Greater Than

< = Less Than or Equal

= > Greater Than or Equal

7. NOT

8. AND

9. OR

As we said in Chapter 5, whenever two or more operations or relations have the same precedence, Sorcerer will do them in order from left to right.

MORE ABOUT BRANCHING INSTRUCTIONS

In Chapter 4 you saw how numerical IF . . . THEN statements instruct Sorcerer to make decisions based on whether or not a numerical expression has value 0. Sorcerer can also base its decisions on whether or not a numerical relation or a logical expression is true. This kind of IF . . . THEN statement is written just like the numerical IF . . . THEN, except you use a numerical relation or other logical expression instead of a numerical expression.

Examples:

```
47 IF X=Y THEN 78
```

```
12 IF (A=B OR C<>D+2) THEN PRINT "HELP!"
```

```
101 IF (OSCAR(X)=SAM(Z)) THEN STOP
```

```
200 IF (A*2=6 AND X<Y) THEN 17
```

Recall the guessing game program in Chapter 4. We could rewrite line 90 as:

```
90 IF Y=1 THEN 110
```

and the program would behave exactly as before.

CHAPTER 8: TRADING AT THE BAZAAR

(Numerical Functions)



Think of a function as a little trading booth: You give something to the merchant, and you get something else in exchange.

Examples:

The square function — you supply a number, say X , and you get back $X * X$. You give 2, you get back 4; give -3 , get back 9; give 1.5, get back 2.25, and so on.

The absolute value function —

you supply any positive number (or 0) and get the same number back. Supply a negative number, say $-X$, and get X . Give 1, get 1; give -10 , get 10; give -1.5 , get 1.5 etc.

The examples above are numerical functions — put in a number, and out comes a number. This is the only kind of function we will consider in this chapter. In Chapter 9, you will learn about string functions.

FUNCTION NOTATION

The value you feed into a function (its input value) is called the *argument*; the *function value* (its output value) is referred to in this way:

<function name> (<argument>)

Examples:

- We give the name SQR to the square root function. If X is 0 or positive, SQR(X) is just the number which, multiplied by itself, equals X. That is:

SQR(X) * SQR(X) equals X

If you try to call SQR(X) for some negative value of X, you will get an FC error message (ILLEGAL FUNCTION CALL).

- We give the name ABS to the absolute value function. The value you get when you plug in X is called ABS(X); ABS(1) is 1; ABS(2.3) is 2.3; ABS(-17) is 17.

If you specify an exact value for the argument, the function will give you a definite value. But you can also call a function with a variable or other numerical expression in the argument; then the function does not have a specific value until the variable or expression is evaluated.

Examples:

```
PRINT ABS(X*Y)
IF (SQR(A/B)>ABS(G)) THEN 32
LET HARRY = ABS(OSCAR)
```

THE DEF STATEMENT

Certain functions (called *intrinsic functions*) are built into Standard BASIC. Sorcerer recognizes their names when it sees them, without having to be told how the functions work. These names are all reserved words, so when Sorcerer sees an intrinsic function, it knows it is not dealing with an array, since no array name can contain a reserved word.

But you can also define your own functions with a special statement, the DEF statement. The DEF statement tells Sorcerer that a certain name is the name of a function, rather than an array, and it also tells Sorcerer how the function works — what kind of business the merchant conducts.

The DEF statement has this format:

```
DEF <function name> (<variable>) = < expression >
```

The function name obeys these rules:

1. The name must have at least three characters.
2. The first two characters must be the letters FN.
3. The third character must be a letter.
4. The name cannot contain any reserved words.

The expression can be any numerical expression, but it should not contain any variables other than <variable>.

Example:

```
42 DEF FNJOE(A) = SQR(A)*2.7 + 6
```

This tells Sorcerer that every time it sees the function FNJOE(<expression>) it must:

1. Evaluate the expression in the argument.
2. Take the square root of the result.
3. Multiply by 2.7.
4. Add 6.

THE FUNCTIONS INT AND RND

Here are two intrinsic functions you will find handy. The function INT is called the *greatest integer function*. For any number X, INT(X) is the biggest integer that doesn't exceed X. If X is an integer, then INT(X) is X itself. Otherwise, if X is positive you get INT(X) by throwing away the fractional part of X, and if X is negative you get INT(X) by adding -1, and *then* throwing away the fractional part.

Examples:

INT(1.5)	is 1	INT(-17)	is -17
INT(1)	is 1	INT(-3.5)	is -4
INT(478.006)	is 478	INT(-.005)	is -1

The function RND produces random numbers. Feed RND a negative number, and Sorcerer will generate a sequence of random numbers, each one between 0 and 1. You get a different sequence of random numbers for each different negative number. Now you can pick numbers out of this sequence by feeding positive numbers — for each different positive number X, RND(X) gives a different number from the sequence. RND(0) always gives you the last number that you called out of RND, whatever it was. To generate a new sequence, just put in a new negative number.

Example:

We can use RND and INT to simulate dice. RND always produces numbers between 0 and 1, so $6 * \text{RND}(X) + 1$ will always be between 1 and 7, and $\text{INT}(6 * \text{RND}(X) + 1)$ will be an integer from 1 to 6.

```
10 REM SORCERER'S DICE
20 LET R=-1
30 INPUT "DO YOU WANT TO ROLL AGAIN"; Y
35 REM ANSWER 1 IF YOU DO, AND 0 IF YOU DON'T
40 IF Y=1 THEN STOP
50 LET A=RND(R)
55 REM STARTS A NEW SEQUENCE
60 LET B=INT(6*RND(1)+1)
70 LET C=INT(6*RND(2)+1)
80 PRINT "YOU HAVE ROLLED"; B; "AND"; C
90 PRINT "TOTAL"; B+C
100 LET R=R-1
105 REM CHANGE THE SEQUENCE
110 GO TO 30
```

CHAPTER 9: NATIVE PHRASES

(Strings)



In Chapter 1 you saw how to print strings of characters, and in Chapter 3 you saw how to print a string before the prompt (?) of an INPUT. A string is just a series of characters which are stored in Sorcerer's memory. You can have as many as 255 characters in a string.

Any letter, number, or symbol on the keyboard can be a part of a string — even the graphic symbols and your own user-defined symbols. However, you will sometimes have trouble if you try to use the quote mark (") as a character in a string. This is because quote marks are used to *delimit* strings. When Sorcerer sees a bunch of characters between a pair of quote marks, it knows those characters are to be treated as a string. For instance, if you tried to use " as part of a string, Sorcerer would see three (or more) quote marks, with other characters scattered around them, and would become confused about which ones were the first and last characters. However, the situation is not hopeless; you will see there are ways to put quote marks into a string without confusing the Sorcerer.

Example:

This program uses strings of graphic characters to print a picture on the screen. Here is how to find the graphics on your keyboard:

/ is just **GRAPHIC** **B**
 \ is just **GRAPHIC** **N**
 | is just **GRAPHIC** **1**
 | is just **GRAPHIC** **8**

```

100 PRINT TAB(10);"/\"
110 PRINT TAB(10);"| |"
120 PRINT TAB(10);"| |"
130 PRINT TAB(9);"/ \ \"
140 FOR Z=1 TO 15
150 PRINT TAB(11);"*"
160 PRINT TAB(10);"*"
170 NEXT Z
180 FOR W=1 TO 30
190 PRINT
200 NEXT W

```

Run this program and you will see a little rocket fly up the screen. Now stick this program onto the end of the COUNTDOWN program in Chapter 4.

STRING CONSTANTS AND VARIABLES

A string of characters surrounded by a pair of quotes is called a **string constant**. It is similar to a numerical constant in having a definite value. Just as there are numerical variables, which can take on many

Run this program and you will see a little rocket fly up the screen. Now stick this program onto the end of the COUNTDOWN program in Chapter 4.

STRING CONSTANTS AND VARIABLES

A string of characters surrounded by a pair of quotes is called a *string constant*. It is similar to a numerical constant in having a definite value. Just as there are numerical variables, which can take on many different number values, so there are *string variables* which can take on many string values. Like the numerical variable, a string variable is just a name which Sorcerer can give to a block of memory locations — but instead of storing number constants in these locations, Sorcerer stores string constants.

You can use any combination of letters and numbers as a string variable, provided:

1. The first character is a letter.
2. No reserved words appear in the name.
3. The last character is a dollar sign (\$).

This last part is important; it is the \$ at the end which tells Sorcerer it is working with a string variable, rather than a numerical variable.

PUT AND TAKE WITH STRINGS

You can use string variables and constants in INPUT, PRINT, DATA, and READ statements, just as you would use numerical variables.

Example:

```
10 INPUT "WHAT IS YOUR STRING"; A$
20 PRINT "YOUR STRING IS"; A$
30 DATA "A", "B", "C"
40 FOR X=1 TO 3
50 READ B$
```

```
70 NEXT X
80 GO TO 10
```

When you answer an INPUT prompt (?) with a string constant, you don't have to put quotes around the string. In the program above, if you answer

```
WHAT IS YOUR STRING?
```

by typing A, Sorcerer will reply

```
YOUR STRING IS A
```

and then go on to tell you the contents of the DATA list. If you answer "A" instead of A, you will get exactly the same result.

You can get Sorcerer to accept strings containing the quote mark if you feed it the strings through INPUT statements.

Example:

Delete lines 30 through 70 from the last program, and run it. Try as many different strings as you can think of which contain one or more quote marks. Here are some suggestions:

```
A"A          "A"A"
A""A          A" "A
" " "         " "
```

STRING RELATIONS AND EXPRESSIONS

We can form string expressions from string variables and constants, just as we form numerical expressions from numerical variables and constants. However, we only have one string operation, while there are six numerical operations (addition, subtraction, multiplication, division, negation, and exponentiation).

The string operation is called **concatenation**, and we use the + sign as its symbol. The effect of concatenation is to paste the string on the right side of the + onto the end of the string on the left.

Example:

"ABC" + "DEF" is just "ABCDEF"

The double use of the + sign doesn't confuse Sorcerer. If it sees a + surrounded by string expressions, it says to itself "Aha! Concatenation!" If it sees a + surrounded by numerical expressions, it says "Addition".

There are also a number of useful string functions, which you will learn about in the next section of this chapter. Many of these functions can appear in string expressions.

In Chapter 7 you learned to form numerical relations with numerical expressions and numerical relation operators. As you might suspect, there are also string relation operators. We will only worry about two of these:

= Equal

and

< > Not equal

String relations are always either true or false, as numerical relations are; Sorcerer can always decide which, just by evaluating the string expressions and comparing them.

So it's not surprising that you can use string relations in logical expressions, and especially, you can use string relations in branching instructions.

Example:

```
10 INPUT "WHO IS BURIED IN GRANT'S TOMB"; GRANT$
20 PRINT
30 IF GRANT$="GRANT" THEN ?0
```

```

40 PRINT "WRONG, DODO. TRY AGAIN."
50 PRINT
60 GO TO 10
70 PRINT " RIGHT YOU ARE. HAVE A CIGAR."

```

STRING FUNCTIONS

A *string function* is any function that involves string expressions, either as the value of the function, or as arguments. Standard BASIC has a number of these, but we will only consider the most useful ones.

LEFT\$ and RIGHT\$ give you the leftmost and rightmost characters of a string, respectively. LEFT\$(A\$,3) is just a three-character string consisting of the leftmost three characters of A\$. If A\$ has the value "ABCDEF", then LEFT\$(A\$,3) is "ABC". Similarly, RIGHT\$(A\$,2) is a string consisting of the rightmost two characters of A\$; if A\$ has value "ABCDEF", then RIGHT\$(A\$,2) is "EF".

The general formats for these functions are:

LEFT\$(<string expression> , <numerical expression>)

and

RIGHT\$(<string expression> , <numerical expression>)

To evaluate these functions, Sorcerer first evaluates the string expression and the numerical expression, to get a string and a number. It then takes the leftmost or rightmost characters from the string.

Examples:

RIGHT\$(A\$ + "G17", 24)

LEFT\$("GRANT'S TOMB", X+1)

LEFT\$(A\$ + B\$, 3*X+Y)

Run this program and experiment with as many different strings as you can think of:

```
10 INPUT "WHAT IS YOUR STRING"; A$
20 PRINT
30 INPUT "WHAT IS YOUR NUMBER"; X
40 PRINT
50 PRINT LEFT$(A$,X), RIGHT$(A$,X)
60 PRINT
70 GO TO 10
```

You can also take characters from the middle of a string using the function `MID$`. You specify the number of characters you want to take, and which character you want to start with. For instance, `MID$(A$,3,6)` consists of the six characters of `A$` starting with the third.

Examples:

`MID$("ABCDEF", 3,2)` is "CD"

`MID$("ABCDEF", 2,3)` is "BCD"

`MID$("ABCDEF", 3,3)` is "CDE"

If you ask for too many characters from the string, `MID$` will give you as many as it can.

Example:

`MID$("ABCDEF", 5,4)` is "EF"

You can leave out the third argument of MID\$. MID\$(A\$,I) gives you all the rightmost characters of A\$, starting with I-th one.

Example:

MID\$("ABCDEF", 3) is "CDEF"

The most general format for MID\$ is:

MID\$(<string expression> , <first numerical expression> , <second numerical expression>)

The function STR\$ gives the string representation of a number. For instance, STR\$(999) is just the string of digits "999", and STR\$(-1.23456E-01) is the string of letters, digits, and symbols "-1.23456E-01". The difference between 999 and STR\$(999) is that Sorcerer handles the first as a number, and the second as a string.

The general format is:

STR\$(<numerical expression>)

Example:

This program illustrates the use of STR\$, and also the difference between numerical addition (+) and string concatenation (+).

```
10 INPUT "WHAT ARE YOUR TWO NUMBERS"; X,Y
20 PRINT
30 PRINT "THE SUM IS"; X+Y
40 PRINT
50 PRINT "THE CONCATENATION IS"; STR$(X) + STR$(Y)
60 PRINT
70 GO TO 10
```

Try this program with as many different numbers as you can think of, using integer, fixed point, and floating point notation. Notice that the first character of a positive number is always a blank space.

The function LEN tells you how many characters there are in a string.

Examples:

LEN("ABCDE") is 5

LEN("5") is 1

LEN(STR\$(5)) is 2 STR\$(5) has two characters: one the digit five, and one the leading space.

The general format for LEN is:

LEN(< string expression >)

Examples:

LEN("ABC" + G\$)

LEN(MID\$(OSCAR\$, X, Y))

LEN(A\$ + STR\$(X) + RIGHT\$(B\$, 3))

STRING ARRAYS

You can define arrays of strings, just as you define arrays of numbers. Each element of a string array can have up to 255 characters, and the array can have as many dimensions as a numerical array (that is, up to 255). You specify the dimension and highest index values of a string array with a DIM statement, just as you do for a numerical array. In fact, you can use the same DIM statement to simultaneously dimension numerical arrays and string arrays. Names for string arrays obey the same rules as names for string variables — the last character of the name must be a \$ sign.

Examples:

```
76 DIM A(1,7), JOE$(3), SAM(12,3,5), B$(4,4)
```

```
12 IF A$(X)="ABC" THEN 23
```

```
123 LET A43$(I+1)=B43$
```


CHAPTER 10: SIDE TRIPS AND EXCURSIONS

(Subroutines)



Let's try another guessing game:

```
10 REM ANOTHER GUESSING GAME
20 INPUT "WHO IS BURIED IN GRANT'S TOMB"; GRANT$
30 IF GRANT$<>"GRANT" THEN 50
40 GOSUB 200
45 GO TO 60
50 PRINT "WRONG, DODO. THE ANSWER IS GRANT."
60 INPUT "WHAT COLOR IS AN ORANGE"; ORANGE$
70 IF ORANGE$<>"ORANGE" THEN 90
80 GOSUB 200
85 GO TO 100
90 PRINT "THAT WAS DUMB. ORANGES ARE ORANGE."
100 INPUT "WHO WROTE BEETHOVEN'S FIFTH"; BEETH$
110 IF BEETH$<>"BEETHOVEN" THEN 130
120 GOSUB 200
```

```

125 GO TO 140
130 PRINT "SUCH TOWERING STUPIDITY!"
135 PRINT "IT WAS BEETHOVEN."
140 STOP
200 REM SUBROUTINE
210 FOR X=1 TO 15
220 PRINT "CONGRATULATIONS!"
230 PRINT
240 NEXT X
250 FOR Y=1 TO 30
260 PRINT
270 NEXT Y
280 RETURN

```

Lines 200 to 280 are a **subroutine** — a subprogram of a larger program. You should use a subroutine whenever there is a complex action you want to perform repeatedly during a program. In the program above, Sorcerer goes to the subroutine as many as three times.

Use the GOSUB statement to send Sorcerer to a subroutine. The format is:

GOSUB <line number>

where <line number> is the number of the first line of the subroutine.

GOSUB is a lot like GO TO, with one big exception. When you send Sorcerer somewhere with a GO TO statement, it forgets where it was when you sent it (that is, it forgets the line number of the GO TO instruction). But when Sorcerer sees a GOSUB instruction, it remembers the GOSUB's line number. When it finishes the subroutine, it goes right back to the next instruction after the GOSUB. If there are several GOSUBs to the same subroutine, Sorcerer always goes back to the right one — it can't do that with a GO TO instruction.

The statement RETURN sends Sorcerer out of the subroutine, back to its proper place in the main program. The last line of your subroutine should be a RETURN statement, and you can have RETURN statements elsewhere in the subroutine (as part of an IF . . . THEN branch, for instance).

A subroutine can call other subroutines if you wish; the only limit to this is the capacity of Sorcerer's memory. But when Sorcerer sees a RETURN statement, it always goes straight to the instruction after the most recent GOSUB.

Place all your subroutines at the end of your program, so that you can find them easily when you look for them in a program listing. Then put a STOP statement at the end of your main program, ahead of your first subroutine. This will prevent Sorcerer from falling into the subroutine when it finishes the main program.

THE CONDITIONAL GOSUB

The format for this statement is:

ON <numerical expression> GOSUB <list of line numbers>

Here, the line numbers in the list are all the first line numbers of subroutines. The ON . . .GOSUB statement works the same way as ON . . .GO TO, except that when Sorcerer returns from one of the subroutines, it goes straight to the instruction following the ON . . . GOSUB.

CHAPTER 11: PACKAGED TOURS

(Storing and Loading Programs From Cassette Tape)



You can record your programs on a tape cassette using any cassette recorder and the recorder cables that come with your Sorcerer. You can also play a recorded program directly into Sorcerer's memory.

CHECKING YOUR HARDWARE

First check your recorder to see how its controls work. Some recorders will record when you depress the REC button; with others you must press the REC and PLAY buttons together. Now take the two cassette recorder cables that come with your Sorcerer, and plug them into the MIC and EAR connectors of your recorder; plug the other ends into the MIC and EAR connectors at the rear of the Sorcerer (these connectors are next to the power cord). The two recorder cables are identical, so it doesn't matter which one you hook up to the MIC connectors and which to the EAR connectors.

Now check your recorder's tone and volume settings. If these settings are wrong, your programs won't record or load properly. Each recorder has its own characteristics, and so does each brand of cassette tape. If you don't know the proper settings, try putting both the tone and volume on about 1/2 of the way full. Then try recording and loading a test program, making adjustments to the controls as needed to get good results.

You can use any tape cassette, but note that many cassettes have non-magnetic strips of tape at each end of the magnetic tape, so make sure there is magnetic tape under the recording head before you try to record a program. You will get better results with short cassettes (30 minutes or less) than with longer ones — the longer cassettes have more internal friction, and their tape won't move as uniformly past the recording head. Low noise, high frequency tape gives better results than cheaper tape. For best results, we recommend Exidy data cassettes (Part No. DP3001).

The Sorcerer can handle two cassette recorders, using the Exidy Data Cable (Part No. DP 4003). Therefore, when you instruct Sorcerer to play or record a program, you must also tell it whether to use tape unit # 1 or tape unit # 2. The instruction sheet which comes with the Data Cable explains how to tell which tape recorder is unit # 1 and which is unit # 2. Of course, if you only attach one recorder (using the two cables that come with the Sorcerer) then that recorder is tape unit # 1, and there isn't any tape unit # 2.

TO RECORD A PROGRAM

CSAVE is the command to record the program currently in Sorcerer's memory. The format is:

CSAVE<program name>[<unit number>]

Leave a space between CSAVE and the name of your program and another space between the name and <unit number>. The <program name> can be any combination of up to five letters or numbers, but the first character must be a letter. If you don't include the <unit number>, Sorcerer will use unit # 1.

Before you give this command, make sure the REC and PLAY buttons are depressed on your recorder (or the REC button alone, on some recorders). Wait five seconds, and then pick a name for your program and give the CSAVE command. When Sorcerer finishes recording, it gives you a READY prompt; you can then stop the tape.

TO RE-PLAY A PROGRAM

CLOAD is the command to load a program from the cassette tape into Sorcerer's memory. This command also wipes-out whatever program is already in memory and re-sets all variables to 0, just like the command NEW. The format is:

CLOAD<program name>[<unit number>]

Again, leave a space before the <program name> and another before the <unit number>. If you don't specify the unit, Sorcerer will use unit # 1.

Before giving this command, rewind the tape. Give the CLOAD command and then press the PLAY button on your recorder. When Sorcerer finishes loading, it will give you a READY prompt; you can then stop the tape.

Example:

Put this program into memory:

```
10 REM TEST PROGRAM
20 PRINT "TEST PROGRAM"
30 PRINT
40 GO TO 20
```

Put this on tape with the command:

```
C SAVE TEST 1
```

After Sorcerer records the program, it will reply with a READY prompt. If this doesn't happen in a reasonable time, check all your electrical connections, re-adjust your recorder's tone and volume controls, rewind the recorder, and try again.

When you get Sorcerer to record your program properly, wipe-out the version of it in memory, by giving the command NEW. Then give the RUN command to convince yourself that the program really has been wiped off the memory.

Now rewind the tape and load the program back into memory with the command:

```
C LOAD TEST 1
```

If you don't get a READY prompt in a reasonable time, re-adjust your tone and volume controls and try again. When you finally get the READY, check your results by giving the command:

```
LIST
```

You can use CLOAD as a program statement, but when the new program is loaded, the one containing the CLOAD statement is automatically wiped out of memory.

SAVING AND LOADING NUMERICAL ARRAYS

The command to save a numerical array is:

```
CSAVE * <unit number> <array name>
```

You **must** include the <unit number> and leave a space before the <array name>; otherwise, use this command just as you would use the CSAVE command.

The command to load a numerical array from a cassette recorder is:

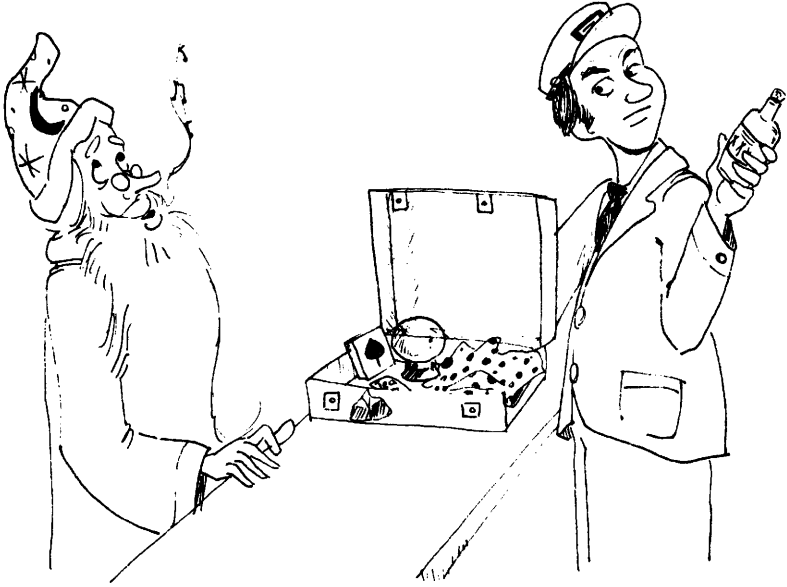
```
CLOAD * <unit number> <array name>
```

Again, you **must** include the <unit number> and leave a space before the <array name>. You must also give a DIM declaration for your array before you try to load it. Otherwise, use this command as you would use CLOAD.

Both CSAVE * and CLOAD * can be used as program statements.

CHAPTER 12: CUSTOMS CHECK

(Tips on Debugging)



By now you have probably tried a few programs of your own, and been disappointed when they wouldn't work. Don't be discouraged; even professional programmers can rarely get a new program to work on the first try. Here are some tips to help you ferret out the bugs hiding in your program.

ERROR MESSAGES

When you run your program, Sorcerer looks for certain especially gross errors; when it finds one, it stops the program and gives you an error message. The message gives the line number of the instruction that caused the problem, and the error code of the error.

The error codes are explained in Appendix D. Be especially watchful of SN errors. Incorrect punctuation, a space where none should be (or no space where there should be one) or any other typographical error can give Sorcerer apoplexy.

SIMPLE DATA SETS AND BRANCH TRACES

Once you get your program to run through without sending error messages, you next have to find out why it won't do what you want it to do. If your program is supposed to handle some data, try it first on very simple data — nice round numbers and short simple strings.

Do all the computations by hand and find out what value each variable should have at each line of the program. Also trace through the logic and see which branches the program is supposed to take when it has to make decisions. You might try temporarily replacing INPUT statements by DATA and READ statements or LET statements. This may save you having to type-in the same data on each trial run. Alternatively, you might try temporarily replacing READ or LET statements by INPUT statements, so that you can change the data more easily on different runs of the program. These INPUT statements should have prompt strings telling what their line numbers are and what data they are asking for. This brings us to our next hint.

DIAGNOSTIC PRINT STATEMENTS

Scatter PRINT statements liberally throughout your program — you can always delete them later when the bugs are gone. Each diagnostic PRINT statement should print its line number and the names and values of any variables being handled in that part of the program.

Be suspicious of any PRINT statement that regularly fails to report as you run the program, or of any that shows up too often. If this persists when you try many different sets of INPUT data, you are probably having problems with your branching statements. Sit down and trace out the logic.

There are several special commands and functions you will find helpful for debugging, once you become familiar with Sorcerer's machine language and inner workings. These are listed in the back of Appendix B.

No matter how puzzled or frustrated you get, remember that there are just so many places for the bugs to hide. If you are persistent enough you will find them. Good luck, and good hunting!

APPENDIX A: RESERVED WORDS

ABS	FOR	NEW	RUN
AND	FRE	NEXT	SGN
ASC	GO	NOT	SIN
ATN	GOSUB	NULL	SPC
BYE	GOTO	ON	SQR
CHR\$	IF	OR	STOP
CLEAR	INP	OUT	STR\$
CONT	INPUT	PEEK	TAB
COS	INT	POKE	TAN
DATA	LEFT\$	POS	THEN
DEF	LEN	PRINT	USR
DIM	LET	READ	VAL
END	LIST	REM	WAIT
EXP	LOG	RETURN	
FN	MID\$	RND	

APPENDIX B: COMMANDS AND STATEMENTS

COMMANDS

Commands direct Sorcerer to arrange memory and input/output facilities, to list and edit programs and to handle other housekeeping details in support of program execution. Sorcerer accepts commands after it prints READY and is at command level. The table below lists the commands in alphabetical order.

BYE

Takes Sorcerer out of Standard BASIC, into the Power-On Monitor. To get back into BASIC, give the Monitor command PP.

CLEAR

Sets all program variables to zero.

CLEAR [<expression>]

Same as CLEAR but sets string space to the value of the expression. If no argument is given, string space will remain unchanged. Normally, string space is set to 50 bytes.

CLOAD<string expression>[<unit number>]

Causes the program on cassette tape designated by the <string expression> to be loaded into memory from designated tape unit. A NEW command is issued before the program is loaded. The string expression cannot contain more than five characters, and the first character must be a letter. If no tape unit is designated, unit # 1 will be used.

CLOAD * <unit number> <array name>

Loads the specified array from the designated cassette unit. You must include the <unit number> (1 or 2); leave a space between the <unit number> and the <array name>. May be used as a program statement.

CONT

Continues program execution after a **CTRL C** has been typed or a STOP statement has been executed. Execution resumes at the statement after the break occurred unless input from the terminal was interrupted. In that case, execution resumes with the reprinting of the prompt (? or prompt string). CONT is useful in debugging, especially where an infinite loop is suspected. An infinite loop is a series of statements from which there is no escape. Typing **CTRL C** causes a break in execution and puts Sorcerer in command level. Direct mode statements can then be used to print intermediate values, change the values of variables, etc. Execution can be restarted by typing the CONT command, or by executing a direct mode GOTO statement, which causes execution to resume at the specified line number. Execution cannot be continued if a direct mode error has occurred during the break, or if the program was modified during the break.

CSAVE <string expression> [**<unit number>**]

Causes the program currently in memory to be saved on the designated cassette tape under the name specified by the first five characters of <string expression>. If no tape unit is specified, the program is saved on unit #1.

CSAVE * **<unit number>** **<array name>**

Causes the array named to be saved on the designated cassette tape. You must include the **<unit number>** (1 or 2); leave a space between the **<unit number>** and the **<array name>**. May be used as a program statement.

LIST

Lists the program currently in memory starting with the lowest numbered line. Listing is terminated either by the end of the program or by typing **CTRL C**.

LIST[**<line number>**]

Prints the current program beginning at the specified line.

NEW

Deletes the current program and clears all variables. Used before entering a new program.

NULL <integer expression>

Sets the number of nulls to be printed at the end of each line. For 10 character per second tape punches, <integer expression> should be ≥ 3 . For 30 cps punches, it should be 2 or 3. When tapes are not being punched, <integer expression> should be ≥ 3 for Teletypes* and Teletype compatible CRTs. It should be 2 or 3 for 30 cps hard copy printers. The default value is 0. If Sorcerer is not connected to a paper tape punch or a line printer, you won't have to worry about this command.

RUN[<line number>]

Starts execution of the program currently in memory at the line specified. If the line number is omitted, execution begins at the lowest line number.

*Teletype is a registered trademark of the Teletype Corporation.

STATEMENTS

In the following table, X and Y stand for any expressions, L stands for any logical expression, I and J stand for expressions whose values are truncated to integers, and V and W are any variable names. The format for a Standard BASIC line is as follows:

<nnnnn> <statement> [:<statement> . . .]

where nnnnn is the line number.

Name	Format
-------------	---------------

DATA	DATA<list>
------	------------

Specifies data to be read by a READ statement. List elements can be numbers or strings. List elements are separated by commas.

DEF DEF FNV(<W>)= <X>

Defines a user-defined numerical function. Function name is FN followed by a legal variable name. Definitions are restricted to one line (64 characters).

DIM DIM <V>(<I>[,J . . .]), . . .]

Allocates space for array variables. More than one variable may be dimensioned by one DIM statement up to the limit of the line. The value of each expression gives the maximum subscript possible. The smallest subscript is 0. Without a DIM statement, an array is assumed to have maximum subscript of 10 for each dimension referenced. For example, A(I,J) is assumed to have 121 elements, from A(0,0) to A(10,10) unless otherwise dimensioned in a DIM statement.

END END

Terminates execution of a program.

FOR FOR<V> = <X> TO <Y> [STEP <Z>]

Allows repeated execution of the same statements. First execution sets $V=X$. Execution proceeds normally until NEXT is encountered. Z is added to V, then IF $Z<0$ and $V>=Y$, or if $Z>0$ and $V<=Y$, Sorcerer branches back to the statement after FOR. Otherwise, execution continues with the statement after NEXT.

GOTO GOTO <nnnn>

Unconditional branch to line number nnnn.

GOSUB GOSUB <nnnn>

Unconditional branch to subroutine beginning at line nnnn.

IF . . . GOTO IF <X> GOTO <nnnn>

Same as IF . . . THEN except GOTO can only be followed by a line number and not another statement.

IF . . . THEN

IF <X> THEN <line number>

or IF <X> THEN <statement> [:<statement> . . .]

or IF <L> THEN <line number>

or IF <L> THEN <statement> [:<statement> . . .]

If value of X is not 0, or if L is true, Sorcerer branches to line number or statement(s) after THEN. If X is 0 or L is false, goes to the line after IF . . . THEN.

INPUT INPUT <V> [, <W> . . .]

Causes Sorcerer to request input from terminal. Values typed on the terminal are assigned to the variables in the list.

LET LET <V> = <X>

Assigns the value of the expression to the variable. The word LET is optional.

NEXT NEXT [<V>, <W> . . .]

Last statement of a FOR loop. V is the variable of the most recent loop, W of the next most recent and so on. NEXT without a variable terminates the most recent FOR loop.

ON . . . GOTO ON <I> GOTO <list of line numbers>

Branches to line whose number is Ith in the list. List elements are separated by commas. If I=0 or > number of elements in the list, execution continues at next statement. If I<0 or > 255, an error results.

ON . . . GOSUB ON <I> GOSUB <list>

Same as ON . . . GOTO except list elements are initial line numbers of subroutines.

PRINT PRINT <X> [, <Y> . . .]

Causes values of expressions in the list to be printed on the terminal. Spacing is determined by punctuation.

Punctuation Spacing — next printing begins:
, at beginning of next 14 column zone
; immediately
other or none at beginning of next line.

String literals may be printed if enclosed by quote marks.
String expressions may also be printed.

READ READ<V>[,<W> . . .]

Assigns values in DATA statements to variables. Values are assigned in sequence starting with the first value in the first DATA statement.

REM REM[<remark>]

Allows insertion of remarks. Not executed, but may be branched into.

RESTORE RESTORE

Allows data from DATA statements to be reread. Next READ statement after RESTORE begins with first data of first DATA statement.

RETURN RETURN

Terminates a subroutine. Branches to the statement after the most recent GOSUB.

STOP STOP

Stops program execution. Sorcerer enters command level and prints BREAK IN LINE nnnnn.

SPECIAL DIAGNOSTIC INSTRUCTIONS AND FUNCTIONS

You won't find these very useful in writing simple programs in Standard BASIC, but if you become familiar with Sorcerer's machine language, you will find them helpful in debugging programs.

WAIT

The status of input ports can be monitored by the WAIT command which has the following format:

WAIT<I,J>[,<K>]

where I is the number of the port being monitored and J and K are integer expressions. The port status is exclusive OR'd with K and the result is AND'ed with J. Execution is suspended until a non-zero value results. J picks the bits of port I to be tested and execution is suspended until those bits differ from the corresponding bits of K. Execution resumes at the next statement after the WAIT. If K is omitted, it is assumed to be zero. I, J and K must be in the range 0 to 255. Examples:

WAIT 20,6 Execution stops until either bit 1 or bit 2 of port 20 are equal to 1. (Bit 0 is least significant bit, 7 is the most significant.) Execution resumes at the next statement.

WAIT 10,255,7 Execution stops until any of the most significant 5 bits of port 10 are one, or any of the least significant 3 bits are zero. Execution resumes at the next statement.

POKE, PEEK

Data may be entered into memory in binary form with the POKE statement whose format is as follows:

POKE<I,J>

where I and J are integer expressions. POKE stores the byte J into the location specified by the value of I. I must be less than 32768. J must be in the range 0 to 255. Data may be POKEd into memory above location 32768 by making I a negative number. In that case, I is computed by subtracting 65536 from the desired address. To POKE data into location 45000, for example, I is $45000 - 65536 = -20536$. Care must be taken not to POKE data into the storage area occupied by Standard BASIC (addresses 0100 to BFFF), or the system may be POKEd to death, and you will have to hit **RESET** and start again.

The complementary function to POKE is PEEK. The format for a PEEK call is as follows:

PEEK(<I>)

where I is an integer expression specifying the address from which a byte is read. I is chosen in the same way as in the POKE statement. The value returned is an integer between 0 and 255. A major use of PEEK and POKE is to pass arguments and results to and from machine language subroutines.

OUT, INP

The format of the OUT statement is as follows:

OUT<I,J>

where I and J are integer expressions. OUT sends the byte signified by J to output port I. I and J must be in the range 0 to 255.

The INP function is called as follows:

INP(<I>)

INP reads a byte from port I where I is an integer expression in the range 0 to 255. Example:

```
20 IF INP(J)=16 THEN PRINT "ON"
```

APPENDIX C: INTRINSIC FUNCTIONS

Standard BASIC provides several commonly used algebraic and string functions which may be called from any program without further definition. In the following list, X and Y stand for any numerical expressions, I and J stand for integers (or expressions whose values are always integers), and X\$ and Y\$ stand for string expressions.

Function	Call Format
-----------------	--------------------

ABS	ABS(X)
-----	--------

Returns absolute value of expression X. $ABS(X) = X$ if $X \geq 0$, $-X$ if $X < 0$.

ASC	ASC(X\$)
-----	----------

Returns the ASCII code of the first character of the string X\$. ASCII codes are in Appendix G.

ATN	ATN(X)
-----	--------

Returns arctangent (X). Result is in radians in range $-\pi/2$ to $\pi/2$.

CHR\$	CHR\$(I)
-------	----------

Returns a string whose one element has ASCII code I. ASCII codes are in Appendix G.

COS	COS(X)
-----	--------

The cosine function: returns $\cos(X)$. X is in radians.

EXP	EXP(X)
-----	--------

Returns e to the power X. X must be ≤ 87.3365 .

FRE	FRE(0)
-----	--------

Returns number of bytes in memory not being used by Standard BASIC. If argument is a string, returns number of free bytes in string space.

|

| INP INP(I)

| Reads a byte from port I.

| INT INT(X)

| Returns the largest integer $\leq X$.

| LEFT\$ LEFT\$(X\$,I)

| Returns leftmost I characters of string X\$.

| LEN LEN(X\$)

| Returns length of string X\$. Non-printing characters and blanks are counted.

| LOG LOG(X)

| Returns natural log of X; $X > 0$.

| MID\$ MID\$(X\$,I [,J])

| Without J, returns rightmost characters from X\$ beginning with the Ith character. If $I > \text{LEN}(X\$)$, MID\$ returns the null string. $0 < I < 255$. With 3 arguments, returns a string of length J of characters from X\$ beginning with the Ith character. If J is greater than the number of characters in X\$ to the right of I, MID\$ returns the rest of the string. $0 \leq J \leq 255$.

| RND RND(X)

| Returns a random number between 0 and 1. $X < 0$ starts a new sequence of random numbers. $X > 0$ gives the next random number in sequence. $X = 0$ gives the last number returned. Sequences started with the same negative number will be the same.

| POS POS(I)

| Returns present column position of terminal's print head. Leftmost position = 0.

| RIGHT\$ RIGHT\$(X\$,I)

| Returns rightmost I characters of string X\$. If $I = \text{LEN}(X\$)$, returns X\$.

|

SGN SGN(X)

If $X > 0$, returns 1, if $X = 0$ returns 0, if $X < 0$, returns -1 . For example, ON SGN(X) + 2 GOTO 100,200,300 branches to 100 if X is negative, 200 if X is 0 and 300 if X is positive.

SIN SIN(X)

Returns the sine of the value of X in radians.
 $\text{COS}(X) = \text{SIN}(X + 3.14159/2)$.

SPC SPC(I)

Prints I blanks on terminal. $0 <= I <= 255$.

SQR SQR(X)

Returns square root of X. X must be $>= 0$

STR\$ STR\$(X)

Returns string representation of value of X.

TAB TAB(I)

Spaces to position I on the terminal. Space 0 is the leftmost space, 64 the rightmost. If the carriage is already beyond space I, TAB has no effect. $0 <= I <= 255$. May only be used in PRINT statements.

TAN TAN(X)

Returns tangent (X); X is in radians.

USR USR(X)

Calls the user's machine language subroutine with argument X.

VAL VAL(X\$)

Returns numerical value of string X\$. If first character of X\$ is not +, -, or a digit, VAL(X\$) = 0.

APPENDIX D: ERROR MESSAGES

After an error occurs, Sorcerer prints an error message, returns to direct mode, and then gives the READY prompt. The program instructions and all variables and arrays remain in memory, but the program cannot be continued by the CONT command. All GOSUB and FOR context is lost, but you can continue the program with a direct mode GO TO command. When an error occurs in a direct mode command, no line number is printed in the error message. The format for error messages is:

Direct mode error ?XX ERROR

Indirect mode error ?XX ERROR IN YYYYY

Here, XX is the error code, and YYYYY is the line number where the error occurred.

These are the error codes and their meanings:

BS SUBSCRIPT OUT OF RANGE

An attempt was made to reference an array element which is outside the dimensions of the array. This error can occur if the wrong number of dimensions are used in an array reference. For example:

```
LET A(1,1,1) = Z
```

when A has already been dimensioned by DIM A(10,10).

CN CAN'T CONTINUE

Attempt to continue a program when none exists, an error occurred, or after a modification was made to the program.

DD REDIMENSIONED ARRAY

After an array was dimensioned, another dimension statement for the same array was encountered. This error often occurs if an array has been given the default dimension of 10 and later in the program a DIM statement is found for the same array.

FC ILLEGAL FUNCTION CALL

The parameter passed to a math or string function was out of range. FC errors can occur due to:

1. a negative array subscript (LET A(-1) = 0)
2. an unreasonably large array subscript (> 32767)
3. LOG with negative or zero argument
4. SQR with negative argument
5. A^B with A negative and B not an integer
6. a call to USR before the address of a machine language subroutine has been entered.
7. calls to MID\$, LEFT\$, RIGHT\$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STR\$, or ON . . . GOTO with an improper argument.

ID ILLEGAL DIRECT

INPUT and DEF are illegal in the direct mode.

NF NEXT WITHOUT FOR

The variable in a NEXT statement corresponds to no previously executed FOR statement.

LS STRING TOO LONG

An attempt was made to create a string more than 255 characters long.

OD OUT OF DATA

A READ statement was executed but all of the DATA statements in the program have already been read. The program tried to read too much data or insufficient data was included in the program.

OM OUT OF MEMORY

Program is too large, has too many variables, too many FOR loops, too many GOSUBs or too complicated expressions.

OS OUT OF STRING SPACE

String variables exceed amount of string space allocated for them. Use the CLEAR command to allocate more string space or use smaller strings or fewer string variables.

OV OVERFLOW

The result of a calculation was too large to be represented in Standard BASIC's number format. If an underflow occurs, zero is given as the result and execution continues without any error message being printed.

RG RETURN WITHOUT GOSUB

A RETURN statement was encountered before a previous GOSUB statement was executed.

SN SYNTAX ERROR

Missing parenthesis in an expression, illegal character in a line, incorrect punctuation, etc.

ST STRING FORMULA TOO COMPLEX

A string expression was too long or too complex. Break it into two or more shorter ones.

TM TYPE MISMATCH

The left hand side of an assignment statement was a numeric variable and the right hand side was a string, or vice-versa; or a function which expected a string argument was given a numeric one or vice-versa.

UF UNDEFINED USER FUNCTION

Reference was made to a user defined function which had never been defined.

UL UNDEFINED LINE

The line reference in a GOTO, GOSUB, or IF . . . THEN was to a line which does not exist.

/0 DIVISION BY ZERO

0 to a negative power also causes DIVISION BY ZERO error.

APPENDIX E: USER-DEFINED GRAPHICS

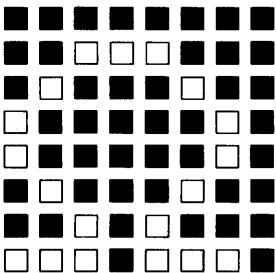
You can program Sorcerer to use special graphic characters that you design yourself. If you want to store or print information in Russian, Arabic, Hebrew, or Greek, you can train Sorcerer to use those alphabets; if you just want to use a graphic symbol that is not in the standard keyboard graphic set, you can do that, too.

Each character is stored in eight successive memory addresses in the memory block FE00 to FFFF (hexadecimal). For example, the first key (the **F** key on the main keyboard) uses FE00 to FE07. So first you have to figure out which eight addresses go with the key in which you want to store your new character.

Your character will be represented as an 8 x 8 array of dots and blanks, so get out a pencil and paper, and work out the representation. Quarter-inch grid graph paper is useful here. Each eight-place row of this array will fit in one of the eight addresses of your character's key; the top line goes into the first address, the next line goes into the second, and so on.

Example:

One way to represent the Greek letter omega is:



Now take each line of your array, and change it into a two-character code in this way: First split the eight-place line in two. Then convert each of these two four-place lines into code with the following table:

Four-Place Line Code

■ ■ ■ ■ = 0	□ ■ ■ ■ = 8
■ ■ ■ □ = 1	□ ■ ■ □ = 9
■ ■ □ ■ = 2	□ ■ □ ■ = A
■ ■ □ □ = 3	□ ■ □ □ = B
■ □ ■ ■ = 4	□ □ ■ ■ = C
■ □ ■ □ = 5	□ □ ■ □ = D
■ □ □ ■ = 6	□ □ □ ■ = E
■ □ □ □ = 7	□ □ □ □ = F

Examples: The eight-place line:

□ □ ■ ■ □ ■ □ ■ is coded as CA

The eight lines of our letter omega are:

■ ■ ■ ■ ■ ■ ■ ■	00
■ ■ □ □ □ ■ ■ ■	38
■ □ ■ ■ ■ □ ■ ■	44
□ ■ ■ ■ ■ ■ □ ■	82
□ ■ ■ ■ ■ ■ □ ■	82
■ □ ■ ■ ■ □ ■ ■	44
■ ■ □ ■ □ ■ ■ ■	28
□ □ □ ■ □ □ □ ■	EE


What you have just done is to consider each eight-place line of your character as an eight-digit binary number (white dots are 1s and blanks are 0s), and to re-write that number in hexadecimal notation. So you now have a two digit hexadecimal number for each of the eight lines of your new character; you must now get these numbers into the eight memory locations that belong to your character's key.

Get into the Power-On Monitor by giving the command `BYE` (leave the ROM PAC in place). Then give the Monitor command `EN XXXX` where `XXXX` is the first memory address of the desired key. For example, the first address of the **2** key on the main keyboard is `FE08`, so

the proper command for that key is EN FE08. Sorcerer will respond with the address you gave it, followed by a colon.

Example:

You type: EN FE08

Sorcerer
replies: FE08:  cursor

Type in the two-digit hexadecimal code of your character's first line, and hit **RETURN**. Sorcerer will reply with the next address in memory, and wait for more data. Type in the hex-code for your character's second line, and repeat this process until you have entered all eight lines of your character into memory. Sorcerer will then give you the address of the first memory location for the next key on the keyboard. Type in a slash (/) and hit **RETURN**.

Your new character is now stored in the Sorcerer. Just press **GRAPHIC** and **SHIFT**, and your character's key, and your new graphic will appear on the screen.

Note: Your special graphic characters normally live in memory addresses FE00 to FFFF. Addresses FC00 to FDFF contain the standard graphic characters; these are the characters you get using the **GRAPHIC** key, but not the **SHIFT**. You can put your own characters into these addresses if you wish; the procedure is the same as above. Sorcerer will still respond to the one-key BASIC command for your character's key (see Appendix F).

You can get back into Standard BASIC with the Monitor command PP. When you turn off Sorcerer's power, or hit **RESET**, all your user-defined characters will be lost. To save them for future use, pass to the Power-On Monitor and save the contents of the character's memory addresses on cassette, using the Monitor command SA. You can load the characters back into memory from the tape, using the Monitor command LO.

APPENDIX F: ONE-STROKE INSTRUCTIONS

Most of Sorcerer's keys have a Standard BASIC instruction or intrinsic function as an additional meaning. For example, you can use the **6** key on the main keyboard to enter an INPUT statement, or the **Q** key to enter the statement RETURN.

To use this feature, press the key for your function or instruction, simultaneously with the **GRAPHIC** key. A graphic symbol will appear on the screen. But that symbol is not necessarily what goes into Sorcerer's memory. If Sorcerer expects an instruction or function when you hit the keys, it will interpret what you gave it as an instruction or function. The figure on page F-2 shows the additional meanings of the keys.

Examples:

You type: 10

Then hit: **GRAPHIC** and **P**

Then hit: **RETURN**

The screen shows: 10 —

Note: If you have stored your own graphic symbol under key **P**, it will appear here instead of —.

Now type: **GRAPHIC** and **]**

This gives the LIST command.

Then hit: **RETURN**

Sorcerer replies: 10 PRINT

Now try again:

You type: 10

Then hit: **GRAPHIC** and **P**

	END	FOR	NEXT	DATA	BYE	INPUT	DIM	READ	LET	GOTO	RUN	IF	RESTORE			ABS	USR	FRE	INP
GOSUB	RETURN	REM	STOP	OUT	ON	NULL	WAIT	DEF	POKE	PRINT	CONT	LIST				POS	SCR	5	RND
	CLEAR	CLOAD	CSAVE	NEW	TAB(TO	FN	SPC(THEN	NOT	STEP	+	-		LOG	EXP	COS	SIN	
		*	/	^	AND	OR	>	=	<	SGN	INT				TAN	ATN	PEEK	LEN	

One-Stroke BASIC Instructions

Then type: "

Then hit: GRAPHIC P

And type: "

And hit: RETURN

The screen shows: 10—"—"

Now press: GRAPHIC and]

And hit: RETURN

Sorcerer replies: 10 PRINT"—"

Next hit: GRAPHIC and * This gives the RUN
command.

And hit: RETURN

Sorcerer replies by printing —, or whatever graphic you have stored under the P key.

Note: The RETURN key does *not* give the instruction RETURN. It terminates a line of typed instructions or data, and feeds the line into Sorcerer's memory. GRAPHIC Q gives the instruction RETURN (part of a subroutine) but does *not* give a carriage return.

The keys for intrinsic functions work the same way as the keys for instructions. Some of these keys supply a left parenthesis for the function argument, and some do not. You still have to type in the argument and any needed parentheses.

The ? key's additional meaning is INT, the greatest integer function. This key also has a second additional meaning: SHIFT ? gives a PRINT instruction in the same way that GRAPHIC P does.

APPENDIX G: ASCII CHARACTER CODE

ASCII stands for American Standard Code for Information Exchange.

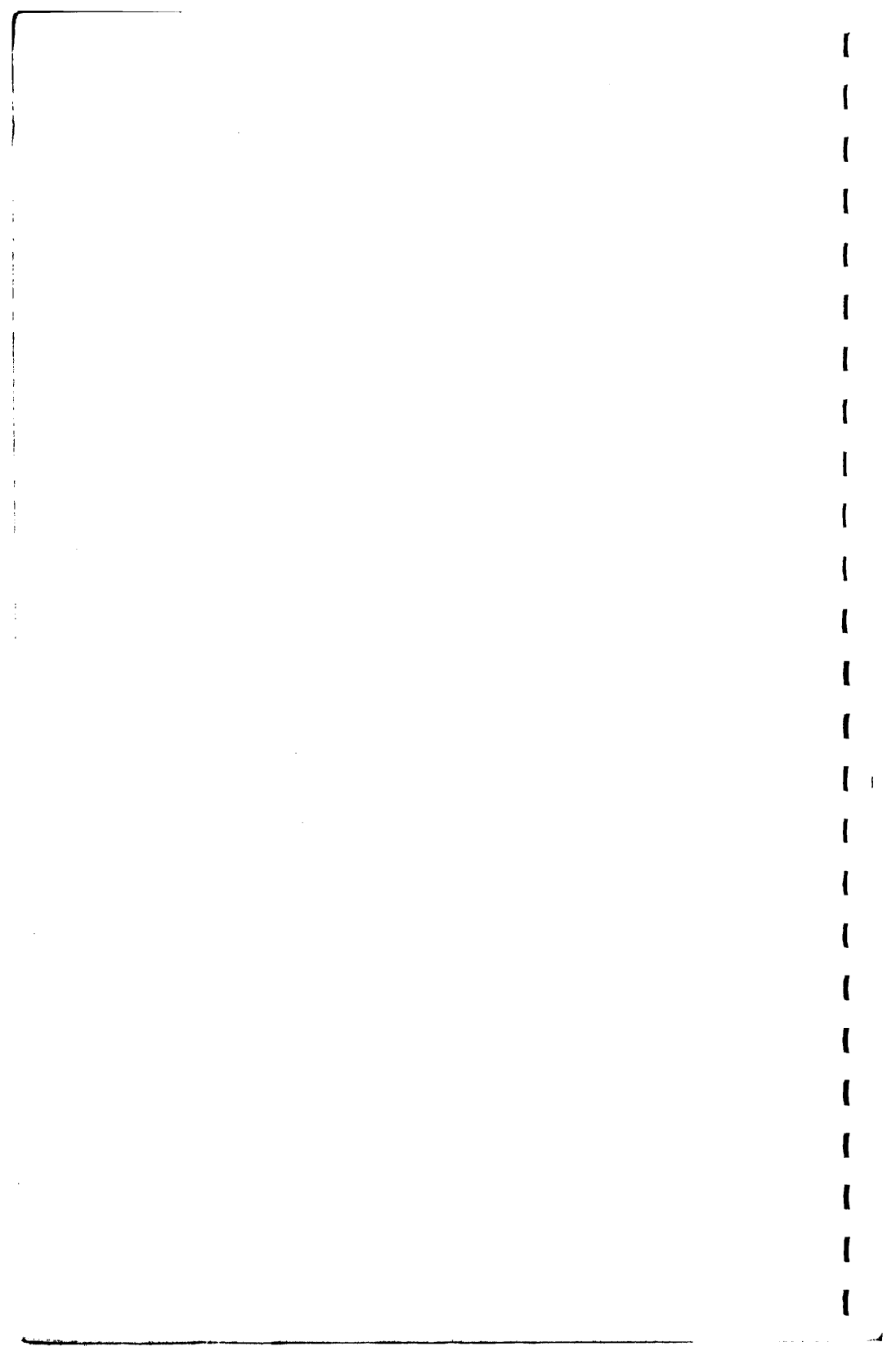
Table 1. Decimal Designations of the Standard Characters

DECIMAL	CHAR.	DECIMAL	CHAR.	DECIMAL	CHAR.
000	NUL	031	US	062	>
001	SOH	032	SPACE	063	?
002	STX	033	!	064	@
003	ETX	034	"	065	A
004	EOT	035	#	066	B
005	ENQ	036	\$	067	C
006	ACK	037	%	068	D
007	BEL	038	&	069	E
008	BS	039	'	070	F
009	HT	040	(071	G
010	LF	041)	072	H
011	VT	042	*	073	I
012	FF	043	+	074	J
013	CR	044	,	075	K
014	SO	045	-	076	L
015	SI	046	.	077	M
016	DLE	047	/	078	N
017	DC1	048	0	079	O
018	DC2	049	1	080	P
019	DC3	050	2	081	Q
020	DC4	051	3	082	R
021	NAK	052	4	083	S
022	SYN	053	5	084	T
023	ETB	054	6	085	U
024	CAN	055	7	086	V
025	EM	056	8	087	W
026	SUB	057	9	088	X
027	ESC	058	:	089	Y
028	FS	059	;	090	Z
029	GS	060	<	091	[
030	RS	061	=	092	\

DECIMAL	CHAR.	DECIMAL	CHAR.	DECIMAL	CHAR.
093]	105	i	117	u
094	^	106	j	118	v
095	<	107	k	119	w
096	'	108	l	120	x
097	a	109	m	121	y
098	b	110	n	122	z
099	c	111	o	123	{
100	d	112	p	124	}
101	e	113	q	125	~
102	f	114	r	126	DELETE
103	g	115	s	127	(Rubout)
104	h	116	t		

Table 2. Standard Abbreviations for ASCII characters 0 through 31

ACK = Acknowledge	FF = Form Feed ✓
BELL = Bell	FS = Form Separator
BS = Backspace ✓	GS = Group Separator
CAN = Cancel	HT = Horizontal Tab ✓
CR = Carriage Return ✓	LF = Line Feed ✓
DC1 = Direct Control 1 ✓	NAK = Negative Acknowledge
DC2 = Direct Control 2 ✓	NUL = Null
DC3 = Direct Control 3	RS = Record Separator
DC4 = Direct Control 4	SI = Shift In
DLE = Data Link Escape	SO = Shift Out
EM = End of Medium	SOH = Start of Heading
ENQ = Enquiry	STX = Start Text ✓
EOT = End Of Transmission ✓	SUB = Substitute
ESC = Escape	SYN = Synchronous Idle
ETB = End Transmission Block	US = Unit Separator
ETX = End Text ✓	VT = Vertical Tab



WARRANTY REGISTRATION CARD

PLEASE MAIL THIS PORTION OF FACTORY WARRANTY
AS SOON AS POSSIBLE

Model No. _____ Serial No. _____

Date of Purchase _____

Customer Name _____

Address _____

City, State, Zip _____

Name of Dealer Where Purchased _____

You do not have to return this card. Failure to return the card will not
affect your Warranty rights.

PASS THIS CARD ON TO A FRIEND WHO IS INTERESTED IN FURTHER INFORMATION ABOUT THE EXIDY SORCERER.

Name _____

Address _____

City _____ State _____ ZIP _____

Phone Number: (_____) _____

- Please send more information on the Sorcerer.
- Please have one of your salesmen or dealers contact me.
- My applications include:
 - I heard about the Sorcerer from (television, radio, magazine, a friend, etc.):
 - My previous computer experience includes:



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL
FIRST CLASS PERMIT #1096 SUNNYVALE, CA

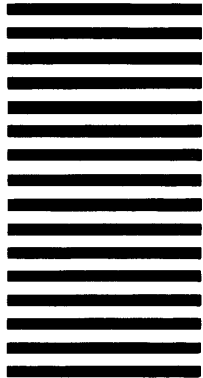
POSTAGE WILL BE PAID BY ADDRESSEE

EXIDY INCORPORATED
Data Products Division
390 Java Drive
Sunnyvale, California 94086

WARRANTY REGISTRATION



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL
FIRST CLASS PERMIT #1096 SUNNYVALE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

EXIDY INCORPORATED
Data Products Division
390 Java Drive
Sunnyvale, California 94086

LIMITED WARRANTY

EXIDY warrants to the registered owner for a period of 90 days from the date of purchase that the computer components described herein shall be free from defects in material and workmanship under normal use and service. This warranty shall be void if the computer enclosure is opened or if the unit is altered or returned as outlined in Customer Service Information. (Refer to A GUIDED TOUR OF PERSONAL COMPUTING, p. 5.) Customer's sole and exclusive remedy in the event of defect is expressly limited to the correction of the defect by adjustment, repair or replacement at EXIDY's election and there shall be no obligation to replace or repair items which by their nature are expendable. No representation or other affirmation of fact, including but not limited to statements regarding capacity, suitability for use, or performance of the equipment, shall be deemed to be a warranty or representation by EXIDY, for any purpose, nor give rise to any liability or obligation of EXIDY what-so-ever.

EXCEPT AS SPECIFICALLY PROVIDED IN THIS AGREEMENT, THERE ARE NO OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OR MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE AND IN NO EVENT SHALL EXIDY BE LIABLE FOR LOSS OF PROFITS OR BENEFITS, INDIRECT, SPECIAL, CONSEQUENTIAL OR OTHER SIMILAR DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR OTHERWISE.

NOTICE

ALL EXIDY SOFTWARE IS DISTRIBUTED ON AN "AS IS" BASIS WITHOUT WARRANTY

EXIDY shall have no liability or responsibility to customers or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by computer equipment or software sold or distributed by EXIDY, including but not limited to any interruption of services, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer or software.

\$9.95



REORDER DP 5002