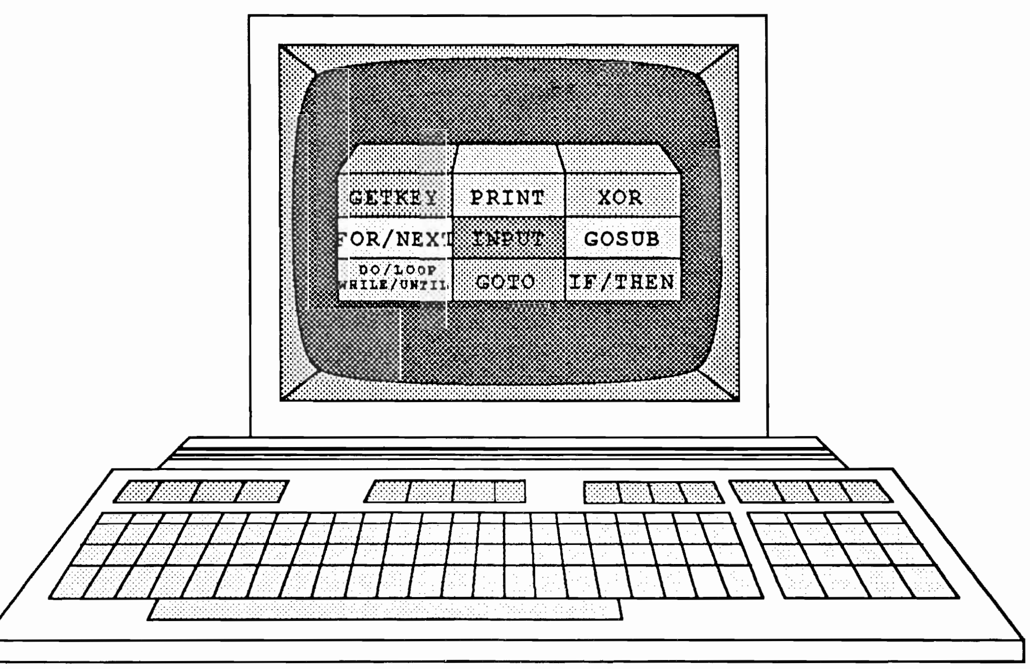
COMMODORE® BASIC

TRAINING GUIDE

Everyone's guide to learning BASIC on the C-128

C-128 BASIC TRAINING GUIDE



Frank Kampow

A Data Becker book from

Abacus Software

First Printing, October 1986
Printed in U.S.A.
Copyright © 1985 Data Becker GmbH
Merowingerstr.30
4000 Dusseldorf, West Germany
Copyright © 1986 Abacus Software, Inc.
P.O. Box 7219
Grand Rapids, MI 49510

This book is copyrighted.No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photcopying, recording or otherwise without the prior written permission of Abacus Software or Data Becker, GmbH.

Commodore, C-64, C-128, 1541, 1571, Datasette and BASIC 7.0 are trademarks or registered trademarks of Commodore International Ltd.

ISBN 0-916439-64-X

Table of Contents

Chapter 1	**BASIC Programming**	1
1.1	Algorithms and programs	3
1.2	The BASIC language	3
1.3	Data flowcharts, program flowcharts and documentation	5
1.3.1	Data flowcharts	7
1.3.2	Program flowcharts	9
1.3.3	Documentation	13
1.4	ASCII Codes	14
1.5	Number systems	15
1.5.1	The binary system	15
1.5.2	Bits and bytes	17
1.5.3	The hexadecimal system	17
1.6	The logical operators	21
1.6.1	`NOT`	22
1.6.2	`AND`	23
1.6.3	`OR`	24
1.6.4	`XOR`	25
	Exercises	27
Chapter 2	**Introduction to programming in BASIC**	29
2.1	The first BASIC program	31
2.1.1	Entering values with `INPUT`	34
2.1.2	Value assignment with `LET`	35
2.1.3	Output with `PRINT`	36
2.1.3.1	`PRINT USING`	39
2.1.3.2	`PUDEF` —altering `PRINT USING`	42
2.1.4	Comments with `REM`	43
2.2	Variables and their use	44
2.2.1	Calculations with variables	45
	Exercises	46
2.3	Numerical functions	47
2.3.1	Functions with `DEF FN`	51
2.3.2	Random numbers	51
2.3.3	More commands for variables	53
2.3.4	`ASC(X$)` and `CHR$(X)`	53
	Exercises	55

Chapter 2(continued)

2.4	TAB and SPC	56
2.5	Strings	57
2.5.1	LEFT$	58
2.5.2	RIGHT$	59
2.5.3	MID$	59
2.5.4	LEN(X$)	61
2.5.5	VAL(X$)	61
2.5.6	STR(X$)	62
2.5.7	INSTR	63
2.5.8	TI$	63
2.6	Editing programs	65
	Exercises	67
Chapter 3	**Extended Program Structures**	69
3.1	Unconditional program jumps	71
3.2	Conditional program jumps	73
3.2.1	IF...THEN...ELSE	73
3.2.2	BEGIN...BEND	76
	Exercises	77
3.2.3	FOR...TO...NEXT	78
3.2.4	Looping with DO...LOOP	83
3.2.4.1	DO...LOOP with UNTIL and WHILE	83
3.3	Computed jump commands	88
3.3.1	Sample program –**Math Tutor**	90
3.3.2	Program jumps with TRAP	97
	Exercises	99
3.4	Program control with GET	100
3.4.1	Data entry with GET	100
3.4.2	Reading the keyboard with GETKEY	102
3.4.3	Function key layout	104
3.4.4	Reading function keys with GET	105
3.5	FRE, POS, SYS, USR(X) and WAIT	107
3.6	PEEK and POKE	108
3.6.1	PEEK	108
3.6.2	POKE	109
3.7	READ, DATA and RESTORE	110

Chapter 4	**Advanced BASIC Applications**	115
4.1	Arrays	117
4.1.1	One-dimensional arrays	117
4.1.2	Examples of one-dimensional arrays	122
	Exercises	128
4.1.3	Multi-dimensional arrays	129
4.2	Subroutines	136
4.3	Menu techniques	149
4.3.1	Using `GET` routines in menus	152
4.3.2	Cursor positioning with `CHAR`	157
4.3.3	Cursor control with `CHR$`-codes	158
4.4	Window techniques	165
4.5	Sort routines	168
Chapter 5	**Principles of File Management**	171
5.1	Common forms of data storage	173
5.2	Different file types	174
5.3	The file	174
5.4	Relative file management	178
Chapter 6	**Music and Graphics**	181
6.1	Music	183
6.2	Graphics	187
6.2.1	Analog clock	189
Chapter 7	**BASIC Internals**	191
7.1	The MONITOR	193
7.2	The variable pointer	195
Chapter 8	**Utilities**	197
8.1	Hardcopy text	199
8.2	Binary conversion	200
8.3	Output with leading zeroes	200
8.4	Variable flag	200
8.5	Program listing to diskette	201
8.6	Reading a sequential file	201

Chapter 9 **Solutions to exercises** 203

Appendices 215

Appendix A: Command Overview–BASIC 2.0 217
Appendix B: Command Overview–BASIC 7.0 238
Appendix C: Data Management Commands in BASIC 7.0 263
Appendix D: Graphics and Sound Commands and Functions 273
Appendix E: Reserved Words in BASIC 7.0 289

Index 293

Chapter 1

BASIC Programming

1. BASIC Programming

1.1 Algorithms and programs

This section explains the fundamentals of programming using the BASIC language on the Commodore C-128.

But before we begin programming, let's first clarify some terminology. We'll be giving you some programming theory—it may sound a little dry at first, but it will be necessary for solving more complex problems later.

Just what is programming?

A computer is a "dumb" machine, unable to do anything unless it's carefully instructed. It has a programming language built into it—but even so, you can't just type in your request at the keyboard:

```
"Calculate the surface of a sphere"
```

To solve this problem with the C-128, or any computer, you must first define a plan outlining how to solve the problem in a clear and logically ordered set of instructions. This plan is called an *algorithm*. Next you must convert this plan into the commands of the computer language. This set of commands is called a *program*. To write a program, you have to use a programming language.

1.2 The BASIC language

The most widely used programming language is called BASIC. Developed in 1961 at Dartmouth College, BASIC is an acronym for:

Beginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode

Since it was developed, many different dialects of BASIC have been created for different computers. For the C-128, Commodore BASIC 7.0 is the version that we'll use.

BASIC 7.0 is an interpreted language and, while it retains most of the language elements of BASIC from other computers, most programs from other computers must be modified to run on the C-128.

As with all interpreted languages, the C-128 cannot immediately understand a BASIC command. A command must first be converted to a form which the computer can understand—*machine language*. The conversion is performed by the BASIC interpreter. When you type in a BASIC command at the keyboard and press the <RETURN> key, the interpreter converts the command into machine code. Only after the C-128 has done this preliminary work can it understand and execute the command.

To recap, an algorithm is an ordered set of instructions to solve a problem. A program is the translation of an algorithm into a programming language, in our case BASIC 7.0.

Let's take a specific problem to further illustrate these two concepts. Suppose you want to determine the volume of a sphere knowing only its radius. Let's carry it a bit further, and say you want to do this for twenty different radii. Remember, an algorithm is an ordered set of instructions to solve a problem. In this case you might proceed as follows.

Do the following for each of 20 values:

- input the radius
- calculate the volume of the sphere
- display that volume

Here's a program to solve our problem:

```
20 FOR I=1 TO 20
30 INPUT"RADIUS (IN CM)";R
40 V=4*π*R^3/3
50 PRINT"THE VOLUME IS ";V;" ccm"
60 NEXT I
```

The program works perfectly and supplies answers to the input values. You decided upon an algorithm and then translated it into BASIC. This all appears very straightforward and easy. Because of the simplicity of your problem, you were able to formulate a solution quickly.

But this does not hold true as you tackle more complex problems. Even the smallest logical or translation error may lead to incorrect results.

A more general approach to computer problems is to divide the problem into small subprograms. You can then think of these smaller pieces as smaller problems having easier solutions.

One difficulty of this method is that you have to make sure that all of the pieces fit together again afterwards. The next section describes two tools that help us do this—*data flowcharts* and *program flowcharts*.

1.3 Data flowcharts, program flowcharts and documentation

Data flow and *program flow* are terms that describe a programming solution to a complex problem. We'll describe both in detail.

A *flowchart* is a pictorial representation of the programmed solution to a problem. Flowcharts are made up of different geometrical symbols. Each symbol represents a specific type of program element. A calculation is one program element; printing the result to the screen is another program element.

Figure 1 is a program flowchart template.

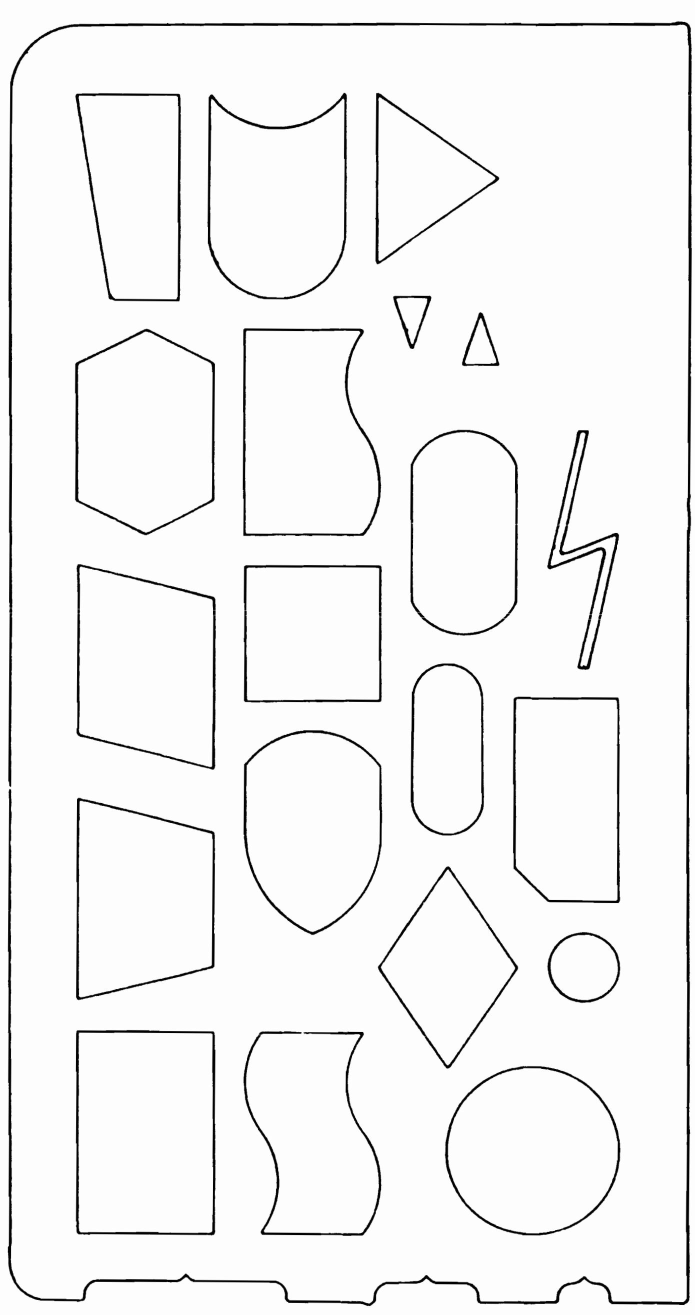


Figure 1: Program flowchart template

1.3.1 Data flowcharts

A *data flowchart* is a pictorial representation of the data elements in the program.

A data flowchart has various symbols to represent the flow of data within a program. But more precisely, it shows which data item is being used (radius value), how it is entered into the computer (by keyboard input), what calculations are performed with the data (finding the volume of the sphere) and how the results are output (to the screen).

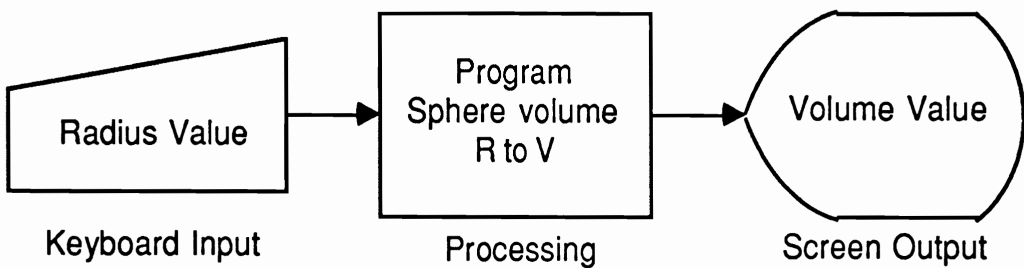


Figure 2: Data Flowchart

As you can see, we've created a data flowchart for a problem as small as this one. You may think it unnecessary or trivial, but it helps you see the overall program. Without such a tool, it might be impossible to write more complex programs. For longer programs these charts may be several pages long. In these cases, they make it easier to understand the flow of the data to be processed. If you become accustomed to creating data flowcharts, it will make your programming task much easier.

Figure 3 illustrates the different symbols used in data flowcharts.

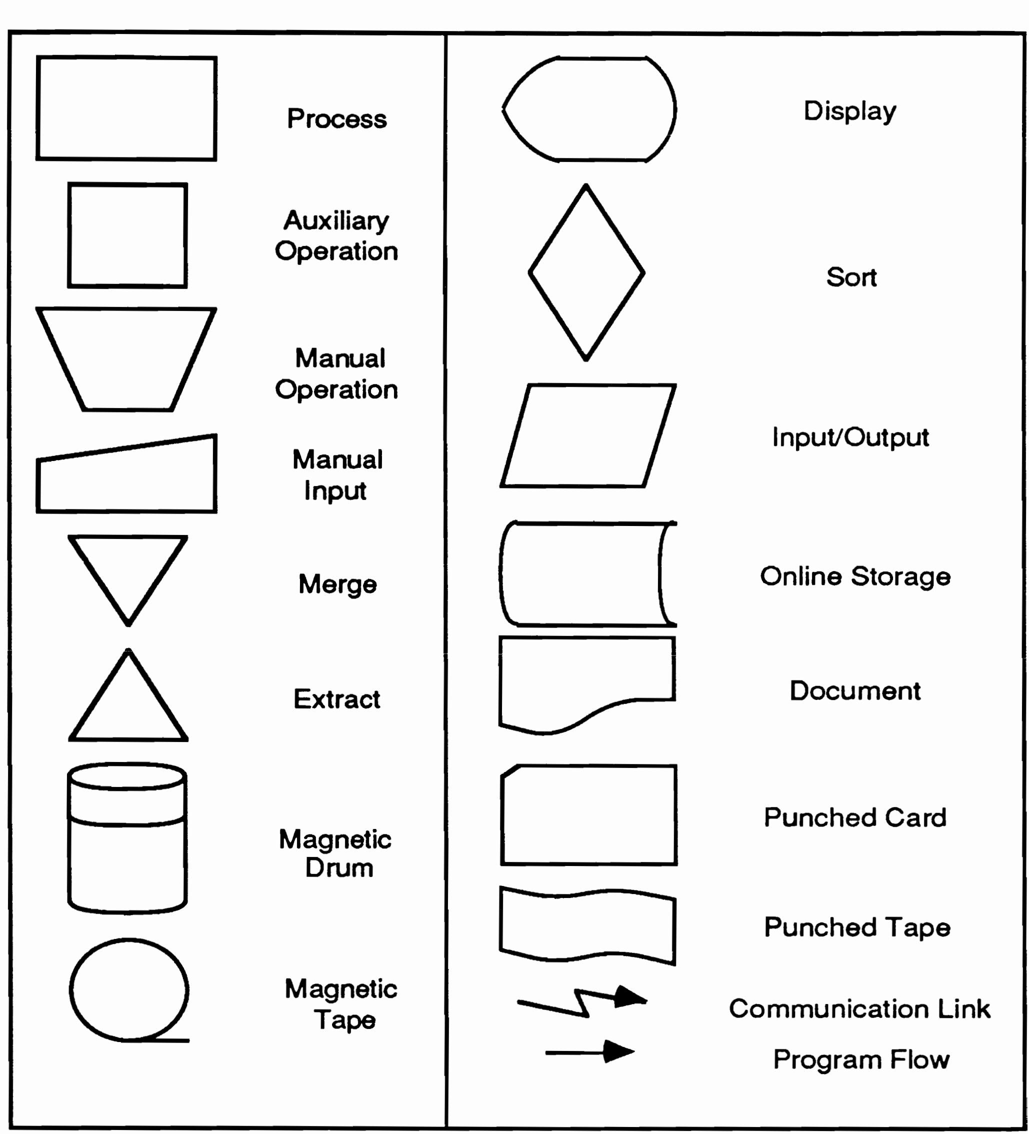


Figure 3: Data Flowchart Symbols

For practice, create a data flowchart for a program that converts miles to kilometers and displays the result on the screen. Compare your data flowchart with the suggested solution in Figure 6.

In this section we have learned:

- data flowcharts clearly show how data is used in a program—which data items are used by the program
- how the data is entered into the computer (source)
- how the data is used by the program (processed)
- how the data is output (destination)

In the next section we'll talk about the program flowchart. The data flowchart does not give information about how, for example, the radius values are converted into the volume values. We need a second form of symbolic representation to tell us the individual steps the computer uses to solve a problem. This is the purpose of the program flowchart.

1.3.2 Program flowcharts

In the data flowchart for the calculation of the volume of a sphere, the only item listed for "processing" was `Program sphere volume R to V`. There is no information about what happens to the data . The problem has not been divided into individual steps. You can do this with the help of a *program flowchart*. A program flowchart shows in clear, individual steps the operations that must be performed to solve a specific problem. The symbols on the programming template are also used for the program flowchart. These are explained in Figure 4.

We'll use our previous example to create our first program flowchart. You should practice making program flowcharts for small examples so that you don't run into difficulties when making flowcharts for larger programs. The old saying "practice makes perfect" applies here.

Program flowcharts are always drawn from top to bottom. When you reach the bottom of the page, you can use a connector symbol to indicate the continuing page. The connector is placed at the lower end of the chart and designated with a number or letter. The second connector is designated with the same letter and placed at the start of the second section. Take a look at the following example of a program flowchart in Figure 5.

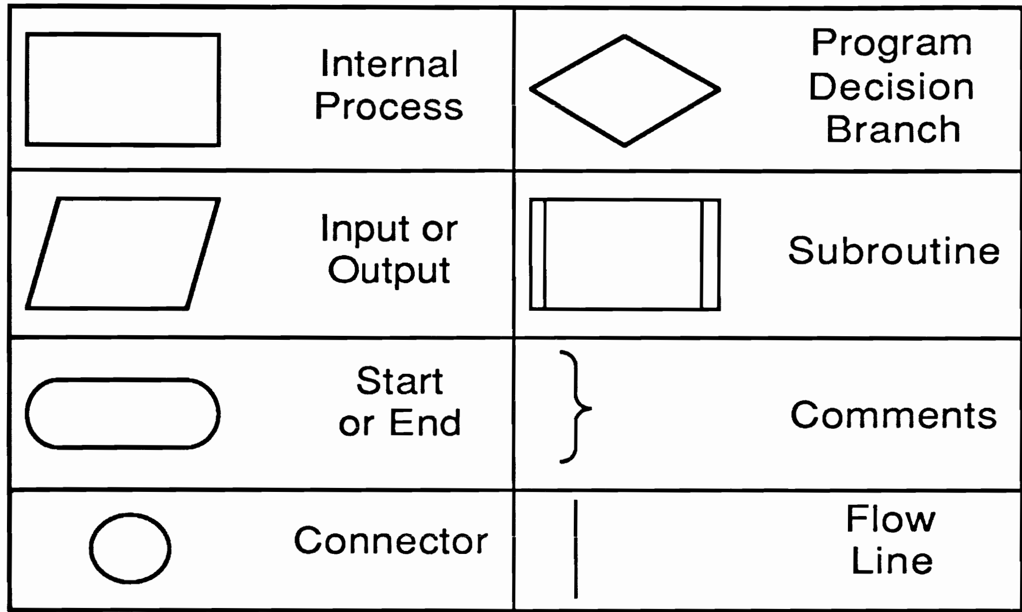


Figure 4: Program Flowchart Symbols

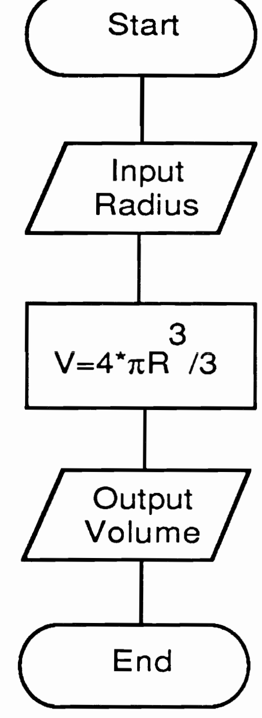


Figure 5: Program Flowchart

The start/end symbol does not have to be translated into BASIC. The input symbol `Input radius` can be translated into the BASIC command `INPUT`, included with a prompt like:

```
ENTER RADIUS IN CM?
```

The formula for calculating the volume of the sphere can be placed directly in the symbol for the internal processing. For the output symbol "Output volume" we use the `PRINT` command, which is provided with the appropriate text. In contrast to our short example program, a `FOR...NEXT` loop is not used here. When a program flowchart has reached a certain level of refinement, the individual symbols need only be translated into the corresponding language statements.

Once you have reached this point in programming, you can think about the first test run of your program. This is done first on paper—that is, you follow the data by means of the data flowchart and check the program flow with the program flowchart. If everything is to your satisfaction, you can start the program by typing `RUN`.

Try to draw your own flowchart for the following problem:

Write a program to convert temperatures from Celsius into Fahrenheit. The formula for this is:

```
F=1.8*C+32
```

When you're done, compare your result with the suggested solution in Figure 7.

The advantages of program flowcharts become clear with larger programs. They are easy to read because of their graphic representation, something that can't necessarily be said of a program listing. Another advantage that's often overlooked is that flowcharts are independent of specific computers. The end result of this is that your flowchart is usable on any computer.

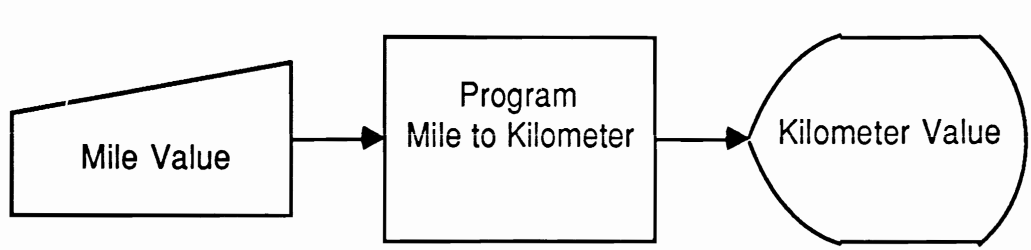


Figure 6: Data Flowchart example solution

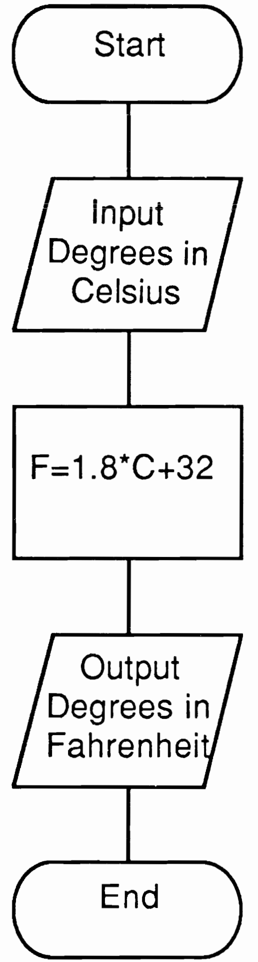


Figure 7: Program Flowchart example solution

Furthermore, it represents a useful tool for documenting your programs. *Documentation* is a narrative description of the program. The writer describes the program's approach in plain English.

All too many programs lack documentation. But if a program has to be modified some time after it's written, even the original programmer may not be able to understand it. This is because you simply cannot remember all the details of a program a year after you wrote it. For this reason, you should get into the habit of documenting your programs. This should be done so that the program can be understood several months later.

1.3.3 Documentation

Documentation is another tool to help with problem-solving by computer. To be precise, program and data flowcharts are a type of documentation for a program—but documentation also includes a narrative of the program.

The *narrative* is an English–language description of the program. It describes:

- the problem being solved
- the approach being used
- any special or unique attributes of the problem
- results to be expected

Here's a sample narrative:

> "This is a generalized program to determine the volume of a sphere. It calculates the spherical volume from the radius entered at the keyboard, for up to twenty different radii. The result is displayed on the screen. It is written in BASIC 7.0. The formula for spherical volume is from Geometric Encyclopedia, R. Chemedes, 1942."

Summarizing the five steps required for good programming:

1) Define the problem (acquire the problem statement and analyze the problem)

2) Develop the algorithm for solution (using data and program flowcharts)

3) Translate the algorithm into a programming language (creating the actual computer program)

4) Test run the program

5) Documentation

1.4 ASCII Codes

As mentioned before, the Commodore 128 cannot directly process the characters which you enter on the keyboard. These are translated into a numerical code called ASCII. ASCII stands for:

American **S**tandard **C**ode for **I**nformation **I**nterchange.

It was developed to standardize the exchange of data between different information carriers. For example, the character "A" always has the ASCII value 65. If this number is sent to a computer or printer that also uses ASCII, this value is always interpreted as the letter "A". Whether you enter characters into the computer with the keyboard or send your data across the country with a modem, as soon as the receiver gets the value 65 it will be translated into an "A". Standard ASCII code uses the values from 0 to 127.

Most computer manufacturers use an extended ASCII code so that other characters can be represented as well. This code is also called ASCII, although not all of the values agree with the standard ASCII.

In standard ASCII the values 32-90 are used for uppercase letters and the values 91-127 for lowercase letters and other characters. The ASCII code of the C-128 is different than the standard ASCII code for upper- and lowercase letters. Commodore included different useful characters, such as graphic symbols, for the remaining values.

1.5 Number systems

A computer can distinguish between only two conditions in its electronic circuits—namely ON and OFF. These two conditions must be transformed into a number system. The *binary system* is used for this. Numbers are represented using only the digits 0 and 1 in the binary system. The 1 stands for the condition ON and the 0 for the condition OFF. To further explain the binary system we'll start with the decimal system.

A decimal number can be converted into a number in any number system. We can thus write the decimal number 5678 like this:

$$5678 = 5*1000 + 6*100 + 7*10 + 8*1$$

(or)

$$5678 = 5*10^3 + 6*10^2 + 7*10^1 + 8*10^0$$

Note: In mathematics, a number raised to the power of zero is always 1. In the decimal system the numbers can be represented as a sum of individual products of base 10. Each digit is assigned a specific power of ten.

power->	10^3	10^2	10^1	10^0
	5	6	7	8

This number is often represented with the subscript $_{10}$ to distinguish it from the other number systems in this section (5678_{10}).

1.5.1 The binary system

The binary system is based on the same principle of individual powers but with the difference that the base is 2. The result is that only the digits 0 and 1 are used. To convert the binary number 1011_2 into a decimal number, we proceed as follows:

The places of the individual digits, as in the decimal system, correspond to individual powers, in this case powers of two. If we now want to convert a binary number, we write each digit under its corresponding power of 2. Then all are added together to get the decimal number.

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ 1 & 0 & 1 & 1 \end{array}$$

The result is the following sum of products:

$$1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 11$$

(or) $1*8 + 0*4 + 1*2 + 1*1 = 11$

The result is the decimal number 11. To convert a decimal number into a binary number, we proceed as follows.

Say we want to convert the decimal number 167 into a binary number. First determine the highest power of 2 in this number. In our case it's:

$$2^7 = 128$$

This value is subtracted from the number to be converted. The same thing is done for the remainder of 39. The highest power of 2 here is:

$$2^5 = 32$$

The highest power of 2 is then:

$$2^2 = 4 \text{ rem } 3 \text{ etc.}$$

Once we have found all of the powers of 2 in the number, write a 1 under the powers of 2 which are in the number. A zero is written under all other powers of 2. The result looks like this:

$$\begin{array}{cccccccc} 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{array}$$

If we then form the sum of the products of the powers of 2 under which a 1 stands, we get our decimal number back—namely 167.

1.5.2 Bits and bytes

Above we used a decimal number less than 256. It required 8 digits in the binary system, or 8 powers of base 2. The smallest unit of information which a computer can process is called a *bit* (*bi*nary digi*t*). A bit can have two conditions or values:

> A set bit has a value of 1. A cleared bit has a value of 0.
> All eight bits together make up one *byte*.

A large number composed of only zeroes and ones is difficult for us to read. For this reason, a number system that is easier for us to read is usually used when working with computers.

1.5.3 The hexadecimal system

In the *hexadecimal system* the base is the number 16. For this you have 16 (including zero) different "digits." To distinguish between the digits that represent values greater than 9, the letters A-F are used. The following sequence of decimal numbers:

`0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18` etc.

becomes the following in hexadecimal notation:

`0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12` etc.

We'll practice working with this number system using examples. We'll first convert hexadecimal numbers into decimal numbers. (The index 16 is used to designate the hexadecimal numbers).

$$2E0C_{16}$$

$$= 2*16^3 + 14*16^2 + 0*16^1 + 12*16^0$$

$$= 2*4096 + 14*256 + 0*16 + 12*1 = 11788_{10}$$

You can see that here the digits `2E0C` are assigned specific powers of 16.

Here is another example:

$$0ABC_{16}$$

$$= 0*16^3 + 10*16^2 + 11*16^1 + 12*16^0$$

$$= 0*4096 + 10*256 + 11*16 + 12*1 = 2748_{10}$$

It is no problem to convert from binary numbers if we make a detour via the hexadecimal numbers. The following examples clarify this.

Examples:

$$0101\ 1011_2 = 5B_{16} = 5*16^1 + 11*16^0 = 91_{10}$$

$$1100\ 0011_2 = C3_{16} = 12*16^1 + 3*16^0 = 195_{10}$$

$$1010\ 1010_2 = AA_{16} = 10*16^1 + 10*16^0 = 170_{10}$$

Notice that a string of eight bits (known as a *byte*) is divided into two halves. Each half is converted into one hexadecimal digit. In the first case, the first and third bits were set in the left half. This yields:

$$5_{16}$$

In the right half the first, second, and fourth bits were set, which yields:

$$B_{16}$$

So we get the hexadecimal value of 5B. The two-place hexadecimal number can be easily converted to a decimal number.

Note: These halves of four bits each are also called *nybbles* or *nibbles* (both spellings are currently in use).

In conclusion, we'll show you how to convert decimal numbers into hexadecimal numbers. The method uses the same principle as that for converting decimal numbers to binary numbers.

Say you want to convert the number 49153 into its hexadecimal equivalent. First find out the largest power of 16 contained in the number. In this particular example it's:

$$16^3 \text{ (or) } 4096$$

The number 49153 is then divided by 16^3. This results in:

12 with a remainder of 1

Now we have almost reached our goal. The values of 16^2 and 16^1 are not contained in the number. The only thing left is 16^0 which is present once. Here is the notation in the number representation:

$$49153 = 12*16^3 + 0*16^2 + 0*16^1 + 1*16^0$$

12_{10} corresponds to hexadecimal C
10_{10} corresponds to hexadecimal A
0_{10} corresponds to hexadecimal 0
1_{10} corresponds to hexadecimal 1

This gives us the hexadecimal number:

$$C001_{16} = 49153$$

The following page has a partial listing of a conversion table to help you see the relationship between the different number systems.

Decimal	Hexadecimal	Binary
0	00	000 0000
1	01	000 0001
2	02	000 0010
3	03	000 0011
4	04	000 0100
5	05	000 0101
6	06	000 0110
7	07	000 0111
8	08	000 1000
9	09	000 1001
10	0A	000 1010
11	0B	000 1011
12	0C	000 1100
13	0D	000 1101
14	0E	000 1110
15	0F	000 1111
16	10	001 0000
17	11	001 0001
18	12	001 0010
19	13	001 0011
20	14	001 0100
21	15	001 0101
22	16	001 0110
23	17	001 0111
24	18	001 1000
25	19	001 1001
26	1A	001 1010
27	1B	001 1011
28	1C	001 1100
29	1D	001 1101
30	1E	001 1110
31	1F	001 1111
32	20	010 0000
33	21	010 0001
34	22	010 0010
35	23	010 0011
36	24	010 0100
37	25	010 0101
38	26	010 0110
39	27	010 0111
.	.	. .
.	.	. .

1.6 The logical operators

The logical operators (also called boolean operators after English mathematician George Boole) are encountered in almost every program. Comparisons and bit manipulations are made possible by these operators. BASIC 7.0 offers you four boolean operations:

NOT, AND, OR, XOR

The first three operators are sufficient to attain the most complicated logical combinations. The function XOR is simply a combination of these three operators.

Note: In digital electronics, these three operators are used in various other combinations (such as NAND, NOR, and XOR gates) in integrated circuits.

As you already know, the computer can distinguish between just two states: ON and OFF. Because of this, the computer has only a two-value predicate logic. It can determine only if a statement is true or false. A *statement* is something like:

2 < 3 (two is less than three)

This statement is a true statement. The computer does not tell us this decision by outputting true or false, but with a corresponding number. If the statement is true, as in the previous example, the computer outputs a value other than zero.

Enter the following sequence of commands into the computer:

```
PRINT 2<3
```
(<RETURN>)

Output: -1

The value is not a zero, so the computer views the statement as true. In most cases, a true statement results in a value of -1. Let's create a false statement:

```
PRINT 3<2
```

Output: 0

The value is equal to zero—the computer indicates that the statement is false. A false statement results in the value zero, and only the value zero.

The logical operators combine two values with each other, where they are compared bit by bit (remember the binary number system?).

Now we'll discuss the operators individually.

1.6.1 NOT

The operator NOT has the result that a true statement returns a false result and a false statement returns true. The following overview should clarify this:

Operator	Value	Result
NOT	-1	0
	0	-1

Examples:

```
PRINT NOT 0
```

Output: -1

```
PRINT NOT -1
```

Output: 0

1.6.2 AND

The operator AND returns a true result only if both conditions are true.

Operator	Value 1	Value 2	Result
AND	0	0	0
	0	-1	0
	-1	0	0
	-1	-1	-1

Example:

```
PRINT 0 AND 0, 0 AND 1, 1 AND 0, 1 AND 1
```

Output: 0 0 0 1

Another example will clarify the function of AND.

Example:

```
PRINT 23 AND 12
```

Output: 4

To understand this result, look at the bit patterns of the values 12 and 23. The bit pattern of 23 is:

00010111

The bit pattern of 12 is:

00001010

These two bit patterns are then combined with AND:

```
00010111
         > AND
00001010

=00000010 = 4
```

Notice that only when both bits are true (1) is the result true (1).

1.6.3 OR

The logical operator OR yields a true result if one or both of the two statements is true.

Operator	Value 1	Value 2	Result
OR	0	0	0
	0	-1	-1
	-1	0	-1
	-1	-1	-1

Example:

```
PRINT 0 OR 0, 0 OR 1, 1 OR 0, 1 OR 1
```

Output: 0 1 1 1

Another example should clarify the function of OR.

Example:

```
PRINT 23 or 12
```

Output: 31

In order to understand this result, we will take another look at the bit patterns of the values 12 and 23. The bit pattern of 23 is:

```
00010111
```

The bit pattern of 12 is:

```
00001010
```

These two bit patterns are then combined with OR:

```
  00010111
           > OR
  00001010

= 00011111 = 31
```

Just like the mathematical operators, the logical operators also have a priority. NOT has the highest priority, AND the second highest. OR has the lowest priority. Concretely, this means that a negation is performed first, before AND or OR. Naturally, the order can be changed by parenthesizing the logical expressions.

To complete this section on logical operators, we'll discuss the function XOR (eXclusive OR), since it is a logical operator as well.

1.6.4 XOR

As already mentioned, this function is a combination of three operators. Let's first take a look at the function of XOR.

Exclusive OR is something we generally mean when we use "or" in everyday language. For example, when a friend says "I'll ride by bike or I'll take my car," these two options are exclusive, because he can't both ride his bicycle and drive his car. Either he drives the car, in which case he doesn't ride his bike, or he rides his bike and doesn't drive his car. The result of a XOR function is true if only *one* of the two statements is true—if the two statements have different truth values.

You can check out the function of the XOR operator by entering the following line into your C-128:

```
PRINT  XOR(0,0),XOR(0,1),XOR(1,0), XOR(1,1)
```

Output: 0 1 1 0

The table for the XOR function looks like this:

Operator	Value 1	Value 2	Result
XOR	0	0	0
	0	-1	-1
	-1	0	-1
	-1	-1	0

As said before, the XOR function can be created using NOT, AND, and OR:

```
Q = (X AND NOT Y) OR (NOT X AND Y)
```

Q is the result of the operation and X and Y are the two operators. This is the equivalent of the statement Q = **XOR** (X,Y).

Exercises

1. Convert the following binary numbers into hexadecimal:

a) `01101100` b) `10010010`
c) `10111010` d) `11110000`
e) `00001100` f) `11001001`

2. Convert the following hexadecimal numbers into decimal:

a) `F0CA` b) `1268`
c) `35A0` d) `0255`
e) `F000` f) `0800`

3. Convert the following binary numbers to decimal:

a) `10110111` b) `00110011`
c) `11111110` d) `00010101`
e) `01010101` f) `10101010`

4. Convert the following decimal numbers into hexadecimal:

a) `63280` b) `24576`
c) `32769` d) `43981`
e) `65534` f) `18193`

Chapter 2

Introduction to Programming in BASIC

2. Introduction to programming in BASIC

In this chapter you will learn how to use simple BASIC commands. Later you'll learn more complex commands by writing several BASIC programs. You'll write the first program by following the five fundamental programming rules presented in Chapter 1.

2.1 The first BASIC program

Let's assume that you want to calculate the surface of a sphere for 10 different radii. Since you already solved a similar problem in the previous chapter, you should have a good feeling for the approach to a solution.

First you'll have to define the problem.

1) Define the problem

Determine the surface area S of a sphere from the given radius, specified in centimeters. The formula for the spherical surface is:

$$S = 4\pi r^2$$

2) Develop the algorithm

a) Start
b) Input r
c) Calculate $S = 4\pi r^2$
d) Output S on the screen
e) End

The data flowchart (Figure 8) and the program flowchart (Figure 9) for the problem solution are on the following page. Figure 9 is a linear flowchart—that is, there are no branches in the form of subroutines or loops. If the terms subroutine and loop are unfamiliar to you, it doesn't matter at the moment. They will be explained in the next section.

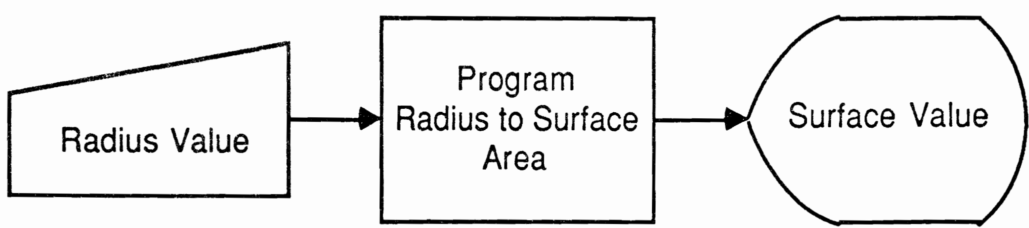


Figure 8: Data flowchart

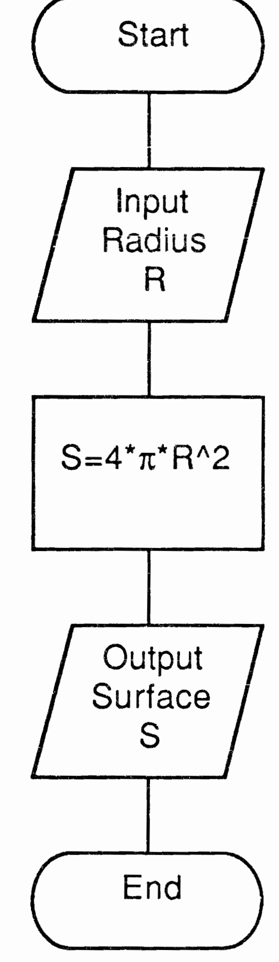


Figure 9: Program flowchart

3) Create the program

Enter the following program from the keyboard:

```
10 INPUT"ENTER RADIUS IN CM";R
20 LET S=4*π*R^2
30 PRINT S
40 END
```

4) Test run the program

Next you must check the data flowchart and program flowchart to verify if all details of the program were planned in a sound, logical manner. Then you can start the program with RUN. Since our program can be checked at a glance, we can enter RUN directly.

5) Documentation

The documentation of a program should be written so that other programmers will be able to understand the program and be able to make changes. For our short program, the data and program flowcharts and the short description under **1)** (definition of the problem) suffice for documentation.

As you have seen, each command is written on a separate line in our program. This greatly increases the readability of programs. Avoid putting multiple statements on a line. With larger programs you won't be able to understand your commands later on. If you become accustomed to using line numbers in increments of ten, inserting new lines later when required will be easier.

If you like, you can first enter line 20 into the computer, and then line 10. The computer sorts out the line automatically according to the size of the line numbers. The computer also executes them in this order, provided other BASIC commands do not change this order through program jumps.

Now to the discussion of the commands used in our program: INPUT, LET, PRINT and END.

2.1.1 Entering values with INPUT

The INPUT command is used in a program to read values from the keyboard during a program run. By entering values the user can affect the results of the program.

If a string of text within quotation marks follows the text INPUT, then this is displayed on the screen as a prompt. This prompt assists the user, by asking him to enter a particular value. The program waits until the user enters the value or values and presses the <RETURN> key.

Here are a few examples.

a) `INPUT S` *Enter:* 1

Here the variable S is assigned the value 1. No prompt is displayed, only a question mark (?). A question mark is automatically displayed when the computer is waiting for data input.

b) `INPUT"ENTER RADIUS";S` *Enter:* 3

The following text appears on the screen as a prompt:

```
ENTER RADIUS?
```

If you type 3 and <RETURN>, the value 3 is assigned to the variable S.

c) `INPUT A,B,C` *Enter:* 4.3,.5,4

In this case the variables A, B, C are assigned the values 4.3,.5,4 one after the other. The comma serves to separate the values entered as well as the variables. Notice that without a prompt it is sometimes difficult for the user to determine what value (or how many values) need to be entered.

2.1.2 Value assignment with LET

The LET command assigns a value to a specific variable. The expression to the right of the = sign is evaluated, and the variable to the left of the = sign receives the value of that expression. The LET command is also called *value assignment*. The keyword LET is often omitted from assignment statements and is therefore considered optional. Some BASIC dialects do require its use, but it is optional in Commodore BASIC 7.0. The following examples should clarify this:

a) LET A=10 (or) A=10

Both assign the value 10 to the variable A.

b) LET A=A+5 (or) A=A+5

Both add 5 to the variable A. The new value is again stored in A.

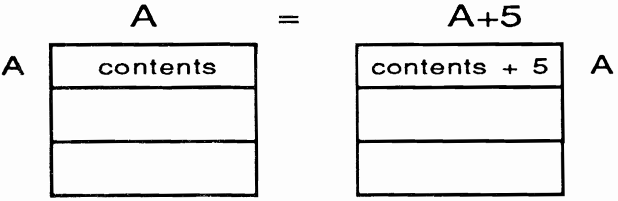
c) LET A=A*B-8 (or) A=A*B-8

The value of A is multiplied by the value of B. Eight is then subtracted from the result. This new result is again assigned to the variable A.

In assignment statements, any arbitrary mathematical expression may stand to the right of the = sign. However, only one variable may be to the left of the = sign.

Let's take another look at example b) above. Mathematically the expression is incorrect. If the value of A is 4, then A=A+5 is false. How can we explain this? In BASIC, the expression is not considered an equality. Rather, it is an *assignment*. You might think of variables as a dresser full of drawers. Each drawer has a label on it. Then the assignment A=A+5 means:

> *Take the contents of drawer A, add five to the contents and put the result into drawer A.*



2.1.3 Output with PRINT

The PRINT command is one of the first commands used by a beginning programmer. But at the same time it's one of the most versatile commands of BASIC 7.0. You can use this command to output text or the values of variables. You can combine the two and output the values of variables and text.

Here are a few examples of the use of the PRINT command. The variables used in the examples have the following values:

A=10   B=20   C=30

Examples:

	Command	Output
a)	PRINT A	10
b)	PRINT "A"	A
c)	PRINT A*B	200
d)	PRINT A,B	10 20
e)	PRINT B;C	20 30
f)	PRINT "B";B	B 20
g)	PRINT "A EQUALS";A	A EQUALS 10

You will learn more about the uses of the PRINT command in later programs. Before we discuss the examples above, we want to say something about the notation in the examples.

In the first mode, the *direct mode*, you enter the statements from the keyboard on the screen. They are immediately executed after you press the <RETURN> key. If you enter:

PRINT A <RETURN>

then the value of A is immediately displayed on the screen.

In the second mode, the *program mode*, the statements are prefixed with a line number. These statements are not executed at this time. Instead they are entered into the computer's memory as part of a program. The statement is stored in memory in sequential order according to the statement line number.

If you enter:

```
10 PRINT A
```

then BASIC stores this statement in memory. The statement is not executed until the program is later RUN.

Now we come to example a) above. This notation of PRINT is used to output the value of a variable. The number 10 appears on the screen since we previously set A=10.

When you print numbers, note that a position is always reserved for the number sign (+/-). If the number is positive, a space is placed before the number. If the number is negative, a minus sign is placed before the number. The numbers 10 and -10 always occupy the same number of positions.

Since no characters follow the variable A in example a), a *carriage return* and *linefeed* are performed automatically. This has the result that the next output appears at the beginning of the following line.

We'll use a typewriter as an example to explain the terms carriage return and linefeed.

Imagine that you have typed the number 10 on a sheet of paper in a typewriter. You then move the lever on the platen to the right. The roller moves the paper one line forward through the machine, and the typewriter carriage is returned to the start of the line. You can then start typing at the start of this new line.

The computer does the same thing when it performs a carriage return and linefeed, except that you don't have to move a lever and there's no carriage to move to the right.

In example b) the character A appears between quotation marks. This causes the character A to be printed, and not the value of the variable A. All characters enclosed in the quotation marks are printed verbatim except for certain special control characters such as the <CLR/HOME> or <INST/DELETE> keys. If you use these keys within the quotation marks, their symbols appear as reversed S or reversed T on the screen. However, these characters are not visible on the screen when the string of characters is printed.

Example c) shows you that calculations can also be performed within a PRINT command. First the product of the variables A*B (10*20) is calculated before it is printed. Here again, a carriage return and linefeed are performed.

Example d) illustrates the ability of PRINT to print several variable values from one statement. The comma suppressed the carriage return/linefeed after printing the value of the first variable (A). It also affects the position at which the values are displayed.

The C-128 divides each line into 10-character-long TAB positions. If a comma is placed between two variables, the second variable is printed at the next TAB start, at the tenth column on the screen line. If several variables are separated by commas, they are printed at successive TAB positions.

Enter the following into the computer:

```
PRINT "1","2","3","4","5","6","7","8"
```

When you press the <RETURN> key, the positions of the individual tabs are displayed. If we had not put the numbers in quotation marks, they would have been moved one position to the right because of the space reserved for the sign (+,-).

Example e) illustrates the effect of the semicolon. The semicolon suppressed both the carriage return and linefeed. It also suppressed the TAB function. The characters are printed in succession in the order in which they appear in the PRINT statement. This makes it possible to include descriptive text following the value of a variable.

Example f) is similar and shows you how to display descriptive text before the value of a variable. In this case the descriptive text is the name of the variable whose value follows. Again, the semicolon separates the descriptive text from the variable.

Example g) is similar to f) and merely shows you that the descriptive text can be of any arbitrary length. The text can be as detailed as you wish.

The END command in the last program line of our example program (on page 33) designates the logical end of the program. This command is usually found at the end of the program. It can also be placed elsewhere within the program.

If you have written a program and do not place the END command in the last line of the program, no harm is done. This is because the computer automatically indicates the end of the program in memory (and not in the program listing itself). Despite this, you should become accustomed to using the END command for the sake of complete programming style.

Our next topic is the PRINT USING command, since it is very closely related to the PRINT command.

2.1.3.1 PRINT USING

PRINT USING is a modified form of the PRINT command. This command is primarily used for "formatting" numbers, variables and/or strings. BASIC 7.0 has the following PRINT USING parameters available:

#	placeholder for characters to be output
.	decimal point position
$	indicates currency
^^^^	scientific notation (e.g., 1.23 E+02)
=	strings will be centered within format fields
>	strings will be right-justified
+	will give positive numbers
-	negative numbers

(Note: + and - signs are printed only as they are encountered)

Let's try some examples of PRINT USING on your computer. First, type these in:

```
A  =  12345.678     <RETURN>
B  =  34.3455       <RETURN>
C  =  -128          <RETURN>
D$ =  "CBM 128"     <RETURN>
```

Next, type in the following PRINT USING sequence, ending with <RETURN>:

```
PRINT USING "#####.##";A    <RETURN>
```

Result:
```
12345.68
```

```
PRINT USING "#####.##";B    <RETURN>
```

Result:
```
   34.35
```

Note that the decimal point is in the same column position onscreen—something not possible with a simple PRINT command.

If the format is too small for the the number, asterisks are printed instead. So if we type in PRINT USING "####.##";A, we will get ******* instead. In other words, the PRINT USING sequence doesn't adapt itself to the situation—you have to make the format large enough to accommodate all possible numbers involved.

Positive or negative numbers are arrived at like this:

```
PRINT USING "+#####.##";A
```

Result:
```
+12345.68
```

```
PRINT USING "#####.##+";A
```

Result:
```
 12345.68+
```

```
PRINT USING "#####.##-";C
```

Result:
```
   128.00-
```

Next, the almighty dollar sign:

```
PRINT USING "#$#####.##";A
```

Result:
```
$12345.68
```

```
PRINT USING "#$#####.##";B
```

Result:
```
   $34.35
```

Note that the dollar sign is preceded by a number sign. This is used to insure the sign's appearance directly before any numbers. Placing the dollar sign first left-justifies the sign:

```
PRINT USING "$#####.##";B
```

Result: `$    34.35`

You can also insert commas to make numbers more readable. Here is an example using DM (Deutsche Marks):

```
PRINT USING "##,###.## DM";A
```

Result: `1,345.68 DM`

Output of European writing styles are discussed further in the next section, which explains the PUDEF command.

Exponential numbers use the following PRINT USING form:

```
PRINT USING "#.##^^^^";A
```

Result: `1.23E+04`

Do this if you want to print more than one place before the decimal point:

```
PRINT USING "###.##^^^^";A
```

Result: `123.46E+02`

> and = are used to format strings. The > right-justifies the text:

```
PRINT USING ">##########";D$
```

Result: `    CBM 128`

The = centers the designated test string:

```
PRINT USING "=##########";D$
```

Result: `CBM 128`

2.1.3.2 PUDEF —altering PRINT USING

As we mentioned above, PUDEF can be used to alter the PRINT USING output. The PUDEF command should be followed by a string of characters or string variable (maximum of 4 characters):

```
PUDEF " ,.$"
```

The first character is a blank space, the second a comma, the third a decimal point, and the last a dollar sign.

This setup changes individual characters. If you want to change the fourth character only, you have to type in the sequence with the first three characters, as before. You can even put these characters into a string variable (e.g. A$=" ,.$") and call them up with PUDEF A$.

The European method of writing out numbers is different from the U.S. Decimal places are designated with a comma, while numbers in the thousands are written to the left of a period. Type in PUDEF " .,", and use this sequence:

```
PRINT USING "###,###.## DM";A
```

Result: 12.345,68 DM

The second and third characters in PUDEF switch the punctuation around. The first character (the blank space) does nothing, but must be included in the quotes.

Suppose we add a multiplication sign—PUDEF would look like this:

```
PUDEF "*"
```

Now give the PRINT USING command again:

```
PRINT USING "###,###.## DM";A
```

Result: *12.345,68 DM

2.1.4 Comments with REM

REM allows you to place comments in your program at any desired location. Everything that follows a REM instruction is ignored by the computer, including other BASIC commands.

Now we'll expand our example program and also make it more understandable for others. This is a vital part of documenting the program, by the way. The modified program listing is on the following page, along with descriptions of the individual program lines:

```
1    REM 2.1.4
10   REM CALCULATE SURFACE OF A SPHERE
20   REM INPUT RADIUS IN CM
30   INPUT "INPUT RADIUS (IN CM)";R
40   REM CALCULATE SURFACE
50   LET S=4.*π*R^2:REM ^ IS UP-ARROW
60   REM OUTPUT SURFACE IN CM^2
70   PRINT" THE SURFACE IS ";S;"CM^2"
80   END
```

Lines 10-20 serve to tell the reader what the program does and what input is required.

Line 30 inputs the data with INPUT, whereby a prompt is also printed for the user. This could also have been done by outputting the text with the PRINT command in a separate program line and the INPUT command by itself in a program line.

The comment in line 40 refers to the calculation in line 50.

In line 50 the variable S is assigned the value of the surface through the mathematical formula.

Line 60 makes references to the output of the CM^2 in line 70.

Line 70 prints the text and value of the variable.

Line 80 concludes the program with an END command.

2.2 Variables and their use

Before giving you some homework, we'll take a look at the different types of variables.

Commodore BASIC 7.0 offers three different types of variables. The first is the *integer variable*. As its name implies, an integer variable can only represent integers or whole numbers. The indicator for an integer variable is the percent sign (%) following the variable name (e.g. A% or C4%). This type of variable is limited to the range between -32768 and +32767.

The second type of variable, a *real variable*, can represent decimal numbers. No "suffix" character is required (e.g. A or B2).

The third type is the *string variable*, which is identified by a dollar sign ($). This variable can handle any string up to 255 characters in length. A longer variable produces:

```
?STRING TOO LONG ERROR
```

There are a few things you have to consider when you use variable identifiers. The computer only recognizes the first two characters of a variable. You could use even longer variable names (say, FILER), but the computer won't be able to tell this from the variable FIRETRAP.

The identifier WAND is illegal, since it contains the logical operator AND.

You can even use numbers as identifiers, but they must be placed at or after the second character of the variable name. For instance, A1 or K9 would be alright, but 1A or 9B would be illegal.

You cannot use identifiers that are identical to two-character commands or syntactical terms—for example, OR, FN and IF aren't usable.

The C-128 uses additional variables for internal functions: TI, TI$ and ST. These cannot be used for your own variables, although you can call them up within a program.

2.2.1 Calculations with variables

If you want to perform calculations with the variables in your programs, you must first learn the rules of the individual computation operations. The order of execution of the operations is just like it would be in algebra. The following listing gives more information.

Operator	Precedence	Meaning
^	First	Exponentiation
*	Second	Multiplication
/		Division
+	Third	Addition
-		Subtraction

There is also a hierarchy for the logical operators, as we have seen.

You now have enough knowledge to solve the following problems. They contain questions about specific sections as well as small programming problems for you to solve on your own. First try to solve the exercises on your own without referring to the corresponding sections. It won't hurt if you make mistakes, since we learn best from our own mistakes. And you won't be graded here. If you are unsure, you can work through the appropriate sections again.

Try your best to follow the five steps of programming when solving the problems. Before each new program you type in, enter the command `NEW` to clear the BASIC memory and erase the old program. In the following section we'll learn some new commands and see how they apply in programs.

Exercises

1. Test the following variable names for validity and give reasons for your decision.

a) `X1`	b) `WORLD$`	c) `AUTO`
d) `ORR%`	e) `IF`	f) `GB`
g) `4NAME%`	h) `255`	i) `MONDAY`

2. Write a program that reads the four values A, B, C, and D and outputs the values A and B on a line followed by the values C and D in the next line.

3. Write a program that calculates the surface of a right-angle triangle in square inches and include appropriate text with the output (Area=1/2 base*height).

4. Write a program that calculates the ideal weight (height in cm minus 100 minus 10 percent) of a person. The height input should be required in cm and the output of the body weight should appear in kilograms. (We're using metric figures to keep the calculations simple here. After you've written the program, try modifying it by converting the measurements to US Customary: one inch = 2.54 cm, one pound = 2.2 kilograms).

5. Write a program that calculates the number of liters in an aquarium after the program has asked for the length, height, and width in cm.

6. Change problem 2 so that the program prints each value on a separate line, together with the name of the variable.

2.3 Numerical functions

Up until now we have only looked at assignments of values to variables. The mathematical functions of the C-128 were not used; only the four basic computations were used.

In this section we'll talk about the built-in functions like `COS(X)` and `SIN(X)`. To do this we'll take a short excursion into mathematics. But don't worry—we won't hit you with any long-winded mathematical proofs, or beat you over the head with formulas. This is a BASIC book, and will remain such.

In many BASIC books and in the C-128 user's manual, you'll note that the arguments for trigonometrical functions like `SIN(X)`, `COS(X)`, or `TAN(X)` are specified in *radians*. What are radians?

Most of us know that a circle can be divided into 360 degrees. One degree is 1/360th of a circle. Ninety degrees is a quarter of the circle, 180 degrees is a half circle, and so on.

Radian measurement is based on the circumference of a circle. Recall that the circumference of a circle is:

$$C=2\pi r$$

If we use a circle with a unit radius (radius=1), the calculation is simplified to:

$$U=2\pi 1 \quad \text{(or)} \quad U=2\pi$$

A circle therefore has 360 degrees, or 2π (6.2831...) radians. Ninety degrees would be $2\pi/4$ or $\pi/2$ radians. The advantage of radians is that you can directly determine the length of the arc cut off by the angle. Actual calculations with radians takes some getting used to, since it's easier to visualize 90 degrees than it is to visualize $\pi/2$ radians.

This short excursion into mathematics is enough for now. Let's write and analyze some example programs so that these ideas become clearer to you.

Enter the following example program into your computer:

```
10 INPUT "ENTER IN DEGREES";DG
20 REM CALCULATE THE SINE
30 SI=SIN(DG*π/180)
40 PRINT "THE SINE OF";DG;"DEG";
50 PRINT" IS =";SI
60 END
```

Start the program by typing RUN. Then enter the value 90 for the angle. You should get the value 1 as the result. The programs expects the angle in degrees and calculates the corresponding sine. If you want to enter the angle in degrees, you must use the conversion in line 30. For calculating the cosine, simply replace SIN with COS. The variable SI can stay the same.

To make the difference from radians clear, enter the program in the following version. But first enter NEW, and press <RETURN>.

```
10 INPUT "ENTER IN RADIANS";RD
20 REM CALCULATE THE SINE
30 SI=SIN(RD)
40 PRINT"THE SINE OF";RD;"RAD ";
50 PRINT"IS =";SI
60 END
```

As you see, line 30 has been changed a little. Start the program and enter the value 1.57079633 (which corresponds to $\pi/2$). Again, the result is 1.

The use of the other functions is quite simple. As explained in the BASIC 7.0 manual, these numerical functions are passed just one variable, which is then used to make a calculation. SQR(X) calculates the square root of X and ATN(X) the arctangent of X. The functions EXP(X) and LOG(X) calculate the Xth power of e=2.71828183 and the natural logarithm of X (base e), respectively. The one function is the inverse of the other. Enter the following command in the direct mode and press <RETURN>:

```
PRINT EXP(1)
```

As a result you get the number:

```
2.71828183
```

Repeat the same process with the following command:

```
PRINT LOG(2.71828183)
```

You again get the value 1. You see that it is relatively simple to make use of these functions in programs. The only difficult part is the conversion of the values.

The following program calculates both natural and decimal logarithms.

```
10 INPUT"INPUT NUMBER";Z
20 REM CALC. NATURAL LOGARITHM
30 LN=LOG(Z)
40 REM CALCULATE DEC. LOGARITHM
50 LO=LOG(Z)/LOG(10)
60 PRINT"THE NATURAL LOGARITHM ";
70 PRINT"FROM";Z;"IS";LN
80 PRINT
90 PRINT"THE DECIMAL LOGARITHM ";
100 PRINT"FROM";Z;" IS";LO
110 END
```

You see for yourself how simple it is to use these functions in programming. The only real difficulty emerges when you have to perform the reverse functions.

NOTE: The trigonometric functions expect values in radians; calculations done in degrees must be converted from radians.

The function `SGN(X)` returns the sign of `X`. The result is 1 if `X` is positive, 0 if `X`=0, and -1 if `X` is negative. Any number can be used in place of `X`.

The function `INT(X)` is another very useful function for rounding numbers. With an appropriate routine, we can round to any number of places after the decimal. The following program should clarify this.

```
10 INPUT" HOW MANY PLACES DECIMAL PLACES";X%
20 INPUT "WHICH NUMBER";N
30 REM ROUND OFF
40 N=INT(N * 10^X% + .5) / 10^X%
50 REM OUTPUT ROUNDED OFF NUMBER
60 PRINT N
70 END
```

The program is quite simple, but we would like to explain the most important lines. Line 10 asks for the number of places to which the decimal number will be rounded. This value is assigned to the variable X%. This is an integer variable, since only integers can be used as input.

Line 20 asks for the number to be rounded off. Enter a decimal number here having more places after the decimal than the number to be rounded.

The actual rounding off is performed in line 40. N is first multiplied by 10 to the X% power. This moves all of the places to be rounded out in front of the decimal point. Then .5 is added in order to round off the final place, since INT simply truncates the number at the decimal point. The integer value of this number is then taken. This is divided by 10 to the X% power to move the digits back behind the decimal point—but this time only the digits that were moved out in front by the multiplication.

Start the program with RUN and the <RETURN> key. Enter some values in order to see what results you get. Carefully look at line 40, the line in which the number is rounded off. You can use such routines in your own programs later.

2.3.1 Functions with DEF FN

The DEF FN function is a practical way of saving space. With it you can assign complex mathematical functions to the expression FN. This expression is called when needed. At the same time, a parameter used as an argument to the function is passed to it. The following example should make this clearer:

```
10 REM DEFINITION FUNCTION
20 DEF FN F(X) =X^2 + 2*X+4
30 REM INPUT PARAMETER
40 INPUT"ENTER VALUE";X
50 REM OUTPUT
60 PRINT FN F(X)
70 END
```

In line 20 the mathematical function X^2+2X+4 is assigned to the expression FN F(X). The values of X in FN F(X) determine the result of the function. If other variables are used within the function, they are not affected by X—they retain their current values.

This function gives you the ability to create your own functions within a program. You can then call up your function with a parameter just like the built-in functions. This saves typing and memory space, and also makes formulas involving the custom function easier to read.

2.3.2 Random numbers

The BASIC 7.0 of the C-128 has a built-in random number generator that can be called with the function RND(X). This function is required in certain types of simulation in which chance plays a role. This function is often used in games in order to introduce random elements. The function is quite simple to use. The assignment A=RND(1) returns a value between 0.0 and 1.0 (zero and one, exclusive) in A. The same sequence of random numbers is always returned for negative values of X. The following example simulates a die. Each time the program is started, a random number between 1 and 6 is chosen.

```
10 REM GENERATE RANDOM NUMBER
20 A = INT (6 * RND(1)) + 1
30 PRINT A
40 END
```

Start the program with RUN and <RETURN>. Execute the program several times and note the numbers printed. You won't be able to detect any pattern in the order of output.

A combination of RND and INT was used in line 20 to output numbers between 1 and 6, since we need integer numbers. The 1 is added so that zero does not occur (lower bound) and the maximum value of 6 can be reached.

With this type of random number generation you can create random numbers in any range. The 6 represents the upper bound of the interval and the +1 is the lower bound. If you want to create random numbers in the range from 100 to 150, line 20 must look like this:

```
20 A=INT((50+1)*RND(1))+100
```

or in general notation, in which U represents the upper bound and L the lower bound:

```
A=INT((U+1-L)*RND(1))+L
```

In the simpler examples you don't always recognize this general form. The following line:

```
20 A=INT(6*RND(1))+1
```

should be written as:

```
20 A=INT((6+1-1)*RND(1))+1
```

in general form. But since the lower bound is one, the formula can be simplified.

2.3.3 More commands for variables

BASIC 7.0 has a large collection of commands that let you influence variables and variable formats. We'll look at two more in this section.

You'll recall the chapter on number systems. There are two functions in the C-128 that let you convert decimal numbers into hexadecimal, and hex numbers into decimal.

The function `HEX$(X)` takes `X` and converts it into a hexadecimal number. `X` must be a number between 0 and 65535. Any other value calls up the error message `?ILLEGAL QUANTITY ERROR`.

For example, `PRINT HEX$(60)` outputs `003C`. A variable can be inserted instead of a number: i.e. `A=1024:PRINT HEX$(A)` gives the result `0400`.

`DEC("X")` converts hexadecimal numbers into decimal. `X` is a hex number between 0000 and FFFF. Examples: `PRINT DEC("C200")` outputs `49664`, and `A$="ABCD":PRINTDEC(A$)` outputs `43981`.

Remember: When using `DEC("X")`, string variables can be no more than four characters (blank spaces excepted).

2.3.4 ASC(X$) and CHR$(X)

The C-128 can output numbers, letters, and certain special characters on its screen when these symbols are placed between the quotation marks in a `PRINT` command.

But not all characters can be printed with this method. The `CHR$` function is used here. With this function lets you output any character of the character set. Enter the following line in direct mode:

```
PRINT CHR$(65)
```

When you press the <RETURN> key, the character `A` appears on the screen.

The function ASC is the opposite of the CHR$ function. If, for example, you want to know the ASCII value of the letter A, you would enter the following command into the computer in direct mode:

```
PRINT ASC("A")
```

If you now press the <RETURN> key, the value 65 appears on the screen. You can also look up the corresponding ASCII values in the C-128 user manual.

To give you a chance to use your newly-acquired knowledge, we'll give you some problems to solve. Compare your results with the suggested solutions and explanations in Chapter 9. Then you can work through the next chapter.

In these problems you will have to use commands that were discussed on the preceding pages. Also, remember to take the five rules of programming into account.

Exercises

1. Write a program that simulates throwing two dice. The results should be printed, with appropriate spacing, on a single line.

2. Write a program that calculates the surface of a triangle according to the formula:

   ```
   F=SQR(S*(S-A)*(S-B)*(S-C))
   ```

   where `S=1/2(A+B+C)` (and `A`, `B`, and `C` are the lengths of the three sides). Note that the formula cannot be directly inserted into the program in the form it stands. The program should ask for the values of `A`, `B`, and `C` in inches. The result should be labeled appropriately.

3. Write a program that asks for the input of a character and then prints the ASCII value of this character, together with the input character, on the same line.

4. Write a program that calculates the height from the time an object takes to fall from that height. The measured fall time should be requested. Air resistance will not be taken into account. The formula is $D=1/2gt^2$. The value of the constant g is 9.81. The result is to be printed in meters/second.

5. Write a program that calculates the gasoline consumption per 100 miles using the following formula:

   usage per 100 = total usage / miles travelled * 100

2.4 TAB and SPC

The two functions TAB and SPC are used to output data or characters at specific positions on a screen line. TAB and SPC are very similar in their uses, but quite different in their effects. The TAB function and the parameter in parentheses always position the output relative to the start of the line.

Enter the following command sequence in the direct mode:

```
PRINT TAB(15) "TEST"
```

The output you get is the word TEST at position 15 of the screen line. Now write a new sequence of commands using the SPC function instead of TAB. After pressing the <RETURN> key you get the same result. When using the commands in this manner, they both have the same effect. The next example shows you the difference between the two. Enter the following commands:

```
PRINT TAB(5)"TEST 1" TAB(20)"TEST 2"
```

After pressing the <RETURN> key, the word TEST 1 appears at the fifth position, and the word TEST 2 at the 20th position. Enter the command sequence again and change the second TAB to SPC. Your line should then look like this:

```
PRINT TAB(5)"TEST 1" SPC(20)"TEST 2"
```

Now when you press the <RETURN> key, you see the difference in the output on the screen. The second word TEST 2 is not printed at the 20th position from the start of the line, but at the 20th position from the last character of TEST 1. This means that the TAB function always works with the *absolute* position in the screen line, and the SPC function with the *relative* position from the last character printed.

The values passed to both functions may not be larger than 255.

When using these functions in connection with output on the printer, TAB has no real use. This is because, in conjunction with the PRINT# command, it is either interpreted as SPC or ignored altogether. For this reason, the TAB function should only be used with the normal PRINT command.

2.5 Strings

A *string* refers to a string of characters that can contain up to 255 of the characters in the 128's character set. The string variable is designated with the dollar sign. A$ would represent a normal designation of a string. The assignment of a character to a string variable is done in the same manner as with the numerical variables. The sole difference is the characters are enclosed in quotation marks. The following example shows a valid assignment:

```
A$="Commodore 128"
```

If you try to assign a numerical value to a string variable (e.g. A$=2), the following error message appears:

```
TYPE MISMATCH
```

The same error message is printed if you try to assign a string to a numerical variable:

```
A="TEST"
```

When using strings, the plus sign is the only computation operator allowed. This character chains two strings together. If we define A$="DISK " and B$="DRIVE", the computation with + yields the string "DISK DRIVE". A short example program will make this clear:

```
10 A$="DISK ":B$="DRIVE"
20 DL$=A$+B$
30 PRINT DL$
40 END
```

In line 10 the string variables A$ and B$ are first initialized. Line 20 assigns the concatenation (linked series) of variables A$ and B$ to the variable DL$. Line 30 then outputs the new string.

Not only can you combine strings with each other, but check for equality or compare the number of characters. We'll get to this, but not until the compare commands are discussed (see IF...THEN...ELSE). Strings may only be compared with other strings. Comparing a string variable to a numerical variable is not legal.

2.5.1 LEFT$

BASIC 7.0 has other functions for manipulating strings. The first command we'll look at is LEFT$. This function returns a portion of the designated string. Enter the following program:

```
10  A$="COMPUTER"
20  B$=LEFT$(A$,1)
30  C$=LEFT$(A$,2)
40  D$=LEFT$(A$,3)
50  E$=LEFT$(A$,4)
60  F$=LEFT$(A$,5)
70  G$=LEFT$(A$,6)
80  H$=LEFT$(A$,7)
90  I$=LEFT$(A$,8)
100 PRINT B$:PRINT C$:PRINT D$:PRINT E$
110 PRINT F$:PRINT G$:PRINT H$:PRINT I$
120 END
```

Start the program with RUN. The result of the program is shown below. This example clearly shows how LEFT$ works. In line 10 the character string COMPUTER is assigned to the string variable A$. Line 20 forms a left partial string of A$ with one character and assigns it to B$. Line 30 then forms a string containing the 2 leftmost characters of A$. Lines 40 to 90 are interpreted in the same manner. This means that the statement LEFT$(A$,X) generates the leftmost X characters of A$. Lines 100 to 110 output the results to the screen.

Here is the result of the program:

```
C
CO
COM
COMP
COMPU
COMPUT
COMPUTE
COMPUTER
```

As you see, we can have some fun with this command. But it's also intended for serious applications as well, especially in data processing.

2.5.2 RIGHT$

The RIGHT$ function works similarly to the LEFT$ command. It differs from LEFT$ only in that is takes the characters starting at the right end of the string instead of the left. Let's change all references to LEFT$ in the previous example program to RIGHT$, starting in line 20 with RIGHT$(A$,1). Start the program again with RUN. You should get the following output on the screen:

```
R
ER
TER
UTER
PUTER
MPUTER
OMPUTER
COMPUTER
```

Now change the order of the numbers in the RIGHT$ statement. Starting with eight and then counting backwards to one so that you get the reverse result. You first get the expression COMPUTER, and last just the letter R. These examples should clarify the use of these functions.

2.5.3 MID$

One of the more interesting functions used in string processing is MID$. With this function you can isolate one or more characters of a string. First we want to look at the operation of this command by means of simple examples.

Enter the following program into your computer:

```
10  A$="THIS IS A SAMPLE STRING"
20  B$=MID$(A$,1,4)
30  C$=MID$(A$,6,4)
40  D$=MID$(A$,11,6)
50  E$=MID$(A$,12,6)+MID$(A$,20,4)+MID$(A$,11,1)
60  PRINT A$
70  PRINT B$
80  PRINT C$
90  PRINT D$
100 PRINT E$
110 END
```

Run the program and take a close look at the result. With the MID$ function you can isolate a specific number of characters from a specific position in a string. These are then placed in a new string. The general syntax is:

```
MID$(M$,X,Y)
```

M$ is the name of the string, X designates the position at which character it will begin, and Y determines the number of characters. The positions are always counted from left to right. So line 20 assigns the substring to B$. A new string is generated from A$ which is to contain four characters and starts with the first character of A$.

The string C$ is formed in the same manner. Here we start with the sixth character so that the string IS A is generated. Line 40 is self explanatory. Line 50 is interesting. Here again we use a *concatenation*, or linked series, to make a string not directly readable from the original string—namely AMPLE RINGS.

You can see that the LEFT$ and RIGHT$ functions can be replaced by the MID$ function. In our examples the position and number of characters were designated by constants. It is also possible to specify these through variables and arithmetic expressions. In addition, you can not only read characters within a string with MID$, but also change or reassign them. For example, write:

```
MID$(A$,3,2)="AT"
```

This changes the third and fourth characters to A and T, respectively, creating "THAT IS A SAMPLE STRING".

2.5.4 LEN(X$)

Before we look at the next function, try to figure out how many characters (without quotation marks) are contained in the string in our last example. The answer? It's 23 characters. You've probably guessed that this has something to do with the next function discussed.

You can determine the length of a string with `LEN(X$)`. The result is numerical and can be assigned to a corresponding variable. If you have not yet entered `NEW` (erasing the program and variables), and `CLR` (setting the variables to zero) since the last example program, enter this in direct mode:

```
PRINT LEN(A$)
```

and press <RETURN>. The result should be 23. You have determined the number of characters in `A$`. When using this function it doesn't matter what characters the string is made up of. All characters in the string are counted, including spaces.

2.5.5 VAL(X$)

The `VAL(X$)` function converts a string `X$` into a numerical value. If the string starts with a character that cannot be converted to a number, such as a letter, the result will be zero. If a letter or other characters which cannot be converted to a number are found within the string, only the first part of the string is converted to a number. The following examples should clarify this:

a)
```
10 A$="343.45"
20 A=VAL(A$)
30 PRINT A
```

Result: `343.35`

b)
```
10 B$="D38.47F"
20 B=VAL(B$)
30 PRINT B
```

Result: `0`

c)
```
10 C$="234FFC54"
20 C=VAL(C$)
30 PRINT C
```

Output: 234

d)
```
10 D$=33,221"
20 D=VAL(D$)
30 PRINT D
```

Result: 33

Enter these examples into your computer and try them out. Example a) shows what happens when an entire string can be converted. The string in example b) starts with a character which cannot be converted into a number, and is therefore interpreted as zero. Example c) shows a "mixed" string, in which only the first group of digits is converted. Example d) is intended only to show that the comma is simply seen as a non-convertable character, and only the first group of digits is converted.

2.5.6 STR$(X)

The `STR$(X)` function has exactly the opposite effect of `VAL$`—it converts a numerical expression into a string. Note that the string produced may start with a space. If the number is positive, the first character will be space. Two examples should clarify this:

a)
```
10 A=1234
20 A$=STR$(A)
30 PRINT A$
```

Result: ` 1234`

b)
```
10 B=-1234
20 B$=STR$(B)
30 PRINT B$
```

Result: -1234

Both of the strings contain five characters each. The values of the numbers themselves could also be converted to strings instead of assigning them to variables first. `STR$(1234)` could be used in example a) instead of `STR$(A)`.

2.5.7 INSTR

BASIC 7.0 offers us another useful function for working with strings: `INSTR`. The `INSTR` function allows you to search through a string for a desired substring of characters. The syntax of the function looks like this:

```
INSTR(A$,B$,X)
```

Here the `X` stands for the position at which the string `A$` is to be searched. `B$` stands for the substring to be searched for. The result is the position of the substring within the string. If the substring is not found within the target string, the result is zero. The following program should clarify the function.

```
10 A$="THIS IS A SAMPLE STRING"
20 B$="IS"
30 Z=INSTR(A$,B$)
40 PRINT Z
50 END
```

`RUN` the program. You'll get the value 3 as the result for variable `Z`. Note that this function searches for the exact string. This means that the substring `is` would not have been found, since is contains lowercase letters.

2.5.8 TI$

This "string" / string variable is a very special—it controls the C-128's internal clock. This clock is reset to 00.00.00 on power-up, and runs as long as the computer is turned on. The time can be read from the variable `TI$`. This string is made up of six characters: two for hours, two for minutes and two for seconds. This variable is system-reserved exclusively. You can alter the value. Type the following program into your computer:

```
10 PRINT CHR$(19);TI$
20 GOTO 10
```

RUN the program—note the HOME area of the screen (the upper-left-hand corner). You'll see a clock running in seconds. These six characters are kept in the same area by the CHR$(19) code (HOME).

Now that you've "programmed" the 128's digital clock, stop the program with the <RUN/STOP> key. The message BREAK IN [*line number*] appears. Change the above program slightly:

```
10 TI$="120000"
20 PRINT CHR$(19);TI$
30 GOTO 20
```

When this program is RUN, the clock starts at the time specified: 12.00.00. Remember to use six characters only, or you'll get an ?ILLEGAL QUANTITY ERROR message. Again, the program can be interrupted with the <RUN/STOP> key.

The GOTO command, which you've used twice in the last two pages, allows you to jump to other line numbers for program execution. We'll discuss the GOTO command in more detail in the next chapter.

2.6 Editing programs

Before going on to writing larger programs, let's take a look at the commands used for helping you write programs.

Editing is the process of altering a program in any way, whether it's deleting or inserting program lines or fixing any ?SYNTAX ERRORs. You're probably familiar with the use of the cursor keys and the <INST/DEL> key. If not, refer to your C-128 user manual.

BASIC 7.0 has a special command for automatic line numbering in steps of 10 called AUTO. Typing in AUTO 10 <RETURN> switches on automatic program lines in steps of ten, beginning with line 10. Pressing <RETURN> at the end of a program line calls up the next line number. For example, if you're typing in line 100, the next line will be 110.

When you want to turn off AUTO mode, type in AUTO without any parameter (e.g. AUTO instead of AUTO 10).

On those occasions when a program is written without autonumbering, or even during the editing stages, you'll probably find yourself using the RENUMBER command. Program development can involve a lot of added lines, so after a while, a program can look a lot like this:

```
10-----
12-----
15-----
19----
20-----
21-----
```

Typing in RENUMBER in direct mode renumbers these lines in steps of 10, from 10 to 60. In addition, all program jumps (GOTO and GOSUB) are adjusted accordingly. Parameters can be added to this command.

For example, in RENUMBER X,Y,Z, the X equals the line number at which the program will begin; Y equals the step size (e.g. 10); and Z equals the old starting line of the program. Accordingly, the command RENUMBER 200,5,100 goes to line 100, changes it to 200, and arranges the new line numbers in steps of 5.

There will be times when you will need to scratch lines out of programs. Then you would use the DELETE command. DELETE 100-200 erases the program lines from 100 to 200. DELETE -200 scratches all lines up to and including 200; and DELETE 200- deletes all lines from 200 on up.

When you want to stop a program in "mid-stream" during test runs, the STOP command within a program line will output the screen message BREAK IN [*line number*]. The CONT command continues the program execution where it left off at the STOP while retaining the current variable contents. RUN, on the other hand, resets the variables to zero, even if you give a line number with RUN. This the important difference between RUN and CONT.

By the way, CONT cannot be used to continue a program halted by an error.

You may want to follow program flow during a run. TRON (TRace ON) displays the line number currently running, encased in brackets []. Since the line number appears only briefly onscreen, you may wish to output this to a printer, and examine the results later. TROFF (TRace OFF) turns the trace mode off.

The NEW command deletes the program currently in the C-128's internal memory. Be very careful when you use this command.

LIST lets you look at program lines. The syntax for LIST is very much like that of the DELETE command. There is one very important distinction between BASIC 7.0's LIST command and Commodore's previous BASIC interpreters: BASIC 7.0 allows the use of LIST within a program, without interrupting the program for the LISTing. This is handy for demo programs, for example. Later we'll use it to mix graphics on the 40-column screen and program code on the 80-column screen.

You are always going to run into errors, whether they're logical or syntactical. Inputting HELP brings you to the "bad" line, and either highlights the error in reverse video (in 40-column mode) or underlines it (in 80-column mode).

Exercises

1. What difference is there in the output that the following two command sequences produce? Try to answer the question without typing them into the computer.

```
PRINT SPC(5)"TEST 1" TAB(15)"TEST 2"

PRINT TAB(5)"TEST 1" TAB(15)"TEST 2"
```

   a) No difference.

   b) For the first command sequence, five spaces will first be printed followed by the output. After fifteen more spaces the second output appears. For the second command sequence five spaces are first printed but the second output does not occur until 10 more spaces later because the TAB command makes reference to the start of the current line.

   c) For the first command sequence, five spaces are first printed, then the second output follows 10 spaces later. For the second command sequence, five spaces are also printed, but the second output does not occur until 15 spaces later.

2. What expression do you get in B$ from the following command sequence if the string A$="DRILL PRESS"?

```
B$=MID$(A$,1,3)+MID$(A$,7,1)+MID$(A$,10,1)
```

3. What expression does one get with the following command sequence for B$ if A$="ROTOR"?

```
B$=LEFT$(A$,3)+RIGHT$(A$,2)
```

4. What must the command sequence look like if one wants to get the result B$="MANY" if A$="ELEMENTARY"?

Chapter 3

Extended Program Structures

3. Extended Program Structures

Up to now, we've been looking at linear programming. But from this point on we'll be using program *jumps*, or *branches*. The disadvantage of using linear programming is that you have to start the program again and again to attain new results. Also, without program branches you would be able to write only the simplest of BASIC programs.

3.1 Unconditional program jumps

The first and simplest kind of jump is a GOTO. This command does just what its name implies—it tells the program to *go to* a specific line number within the program. Unconditional jumps are jumps that have no comparisons involved—the system jumps regardless of the status of the program at the time. More on this later.

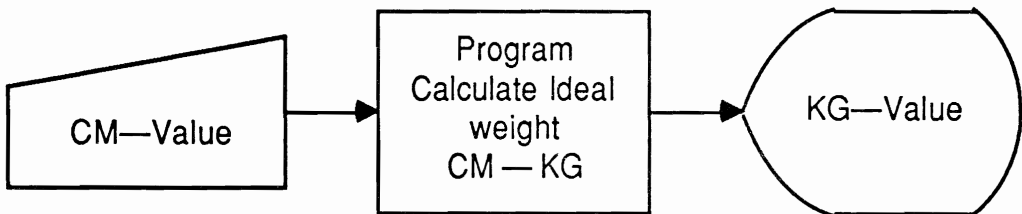
You've already found out that you can only stop the internal clock program using the <RUN/STOP> key. This is a major disadvantage of unconditional jumps—you are stuck with "infinite loops".

Say you run the program below, which calculates the ideal weight of a person. Then let's assume you have a party, and just for laughs, every guest wants to know his or her ideal weight (must be a pretty dull party!). Without the GOTO command, you'll have to keep restarting the program for each new calculation, to the great annoyance of your friends. So, we make a minor change to line 70, and add a GOTO command:

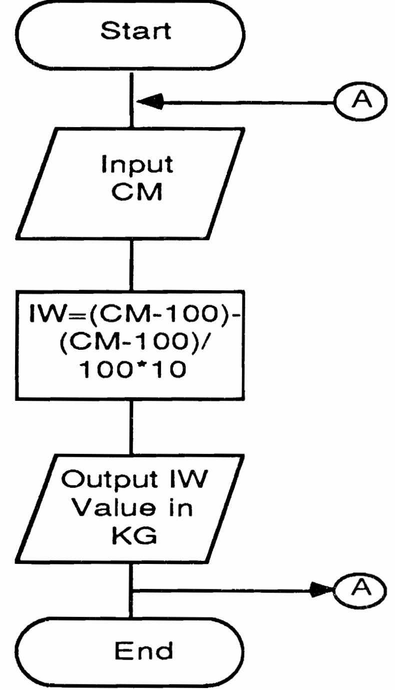
```
10 INPUT"INPUT HEIGHT IN CM";CM
20 REM CALCULATE IDEAL WEIGHT
30 IG=(CM-100)-(CM-100)/100*10
40 REM OUTPUT
50 PRINT"YOUR IDEAL WEIGHT IS";IW;"KG"
60 REM UNCONDITIONAL JUMP W/ GOTO
70 GOTO 10
80 END
```

After the result is printed onscreen, the program goes back to line 10,

thanks to the GOTO in line 70. To stop the program while it's waiting for the next input, you must press <RUN/STOP><RESTORE>. Line 80 says END, but the GOTO command won't let the program flow to line 80. Let's take another look at the original data flowchart.



The program flowchart will look a bit different when the alterations are made (see below). Note the *connector*, which tells where the branch begins and ends. This is our GOTO command. The connector begins right before the end, and jumps to just after the start of the program. Since both connectors have A identifiers, they are treated as a pair of connectors:



By adding a GOTO command, we have essentially created an endless loop—that is, the program continues cycling through until the <RUN/STOP> key is pressed.

3.2 Conditional program jumps

The strength of a computer lies in its ability to handle logical discrimination and comparison of different factors. For example, it can test whether a variable is greater than or less than zero, and go to another program branch, regardless of the result being true or false. The next command does just that.

3.2.1 IF...THEN...ELSE

When the computer runs across the above command, the condition following the `IF` is tested. If the condition is true, the commands/ instructions following `THEN` are executed. If the condition of the `IF` is false, the commands following the `ELSE`, or the next program line, are executed. In the case of false, the procedures following `THEN` are ignored.

The `IF` can follow logical operations, strings, variables, comparisons, numbers or combinations of these. Most of the time these are followed by `THEN [line #]`. It is possible to put new variables or jumps into a subroutine call following `THEN` (more on this later). Here is a simple example of `IF...THEN`:

```
10 INPUT"INPUT A NUMBER";N
20 IF N<0 THEN 50
30 IF N>0 THEN 70
40 IF N=0 THEN 90
50 PRINT"YOUR NUMBER IS LESS THAN ZERO."
60 GOTO 100
70 PRINT"YOUR NUMBER IS GREATER THAN ZERO."
80 GOTO 100
90 PRINT"YOUR NUMBER IS EQUAL TO ZERO."
100 END
```

This program asks for a number. When you input a number and press <RETURN>, the computer tells you whether the number is less than, equal to or greater than zero. Let's face it—you already know that much about the number you input. This is simply a brief demonstration of `IF...THEN`.

Inputting a number greater than 0 causes a reaction at line 20, which tests whether N is less than 0. Since this is not the case, the computer continues on to the next IF...THEN (in line 30). Since our test number is larger than 0, THEN tells the system to jump to line 70. Line 70 confirms that the number is greater than 0, and the program jumps to line 100 (END of program). If you want the program to run continuously, just change line 100 from END to the unconditional jump GOTO 10.

Time for a more complex routine. You know the children's guessing game of trying to guess a number and having the other person tell you "too high", "too low" or "that's right." Here's another version for your C-128:

```
10 REM NUMBER GUESS
20 SCNCLR:PRINT
30 PRINT"INPUT TWO NUMBERS FOR THE ";CHR$(17)
40 PRINT"UPPER AND LOWER LIMITS."
50 INPUT"LOWER LIMIT";L
60 INPUT"UPPER LIMIT";U
70 N=INT((U+1)*RND(1))+L
80 INPUT"YOUR GUESS";GN
90 IF GN<N THEN 120
100 IF GN>N THEN 140
110 IF GN=N THEN 160
120 PRINT"THE MYSTERY NUMBER IS LARGER"
130 GOTO 80
140 PRINT"THE MYSTERY NUMBER IS SMALLER"
150 GOTO 80
160 SCNCLR:PRINT"YIP YIPPEE! YOU GUESSED IT!"
170 PRINT"ANOTHER ROUND (Y/N)?"
180 INPUT A$
190 IF A$="Y" OR A$="YES" THEN 20
200 END
```

The first two lines are not worth explaining, but have a look at line 30. The CHR$(17) is the ASCII value for moving the cursor down one line:

CHR$(17) = cursor down one line

Lines 30 through 60 ask for the boundaries for the number to be randomized (see line 70—the RND function). If line 70 looks confusing, have another look at the chapter on random numbers. Line 80 asks for your number guess. This guess is checked in against the random number —the

variable N—in lines 90-110. Depending on whether the guess (GN) is higher, lower or equal to N, the computer jumps to the corresponding line. If your guess is correct, the program goes to line 160. Line 170 asks if you want to play again; typing Y or YES sends the program to line 190, then back to the beginning of program.

The IF...THEN commands in lines 90-110 compare your guess (N) with the random number (GN). The IF...THEN in line 190 checks a string variable. In order for the condition to be true, the string you type in must be equal to the string expected. Put another way, typing in Y or YES cycles the program through again. But typing in something like OUI or JA terminates the program, unless those strings have been provided for in the input line.

We can program in *controlled loops* using IF...THEN commands. Controlled loops are those that terminate when a certain condition is reached, rather than repeat themselves continually (infinite loops). Let's take what we've learned up until now, and use it in our programming.

For example, let's say you want to create a program that prints out multiples of three. We can do this with controlled loops:

```
10 A=3
20 PRINT A
30 A=A+3
40 IF A > 30 THEN 60
50 GOTO 20
60 END
```

Line 10 starts out the variable A with a value of 3. Line 20 displays the current value of A onscreen. Line 30 is the *counter*—it increments (that's computerese for adds) the value of A by 3. Line 40 checks to see if A has reached 30. If A is less than 30, the program goes to line 50, and loops back to line 20. We've created a loop that cycles through the program 10 times.

The above example can be extended with ELSE. Change line 40 of that program, and delete line 50. The "new" version should look like this:

```
10 A=3
20 PRINT A
30 A=A+3
40 IF A > 30 THEN 60:ELSE 20
60 END
```

This program has the same effect as the previous program—but it works more "elegantly". In this case, line 40 tests for the condition A > 30—if not, the ELSE command sends the program back to line 20. This new command helps you save program lines and, in the long run, memory.

3.2.2 BEGIN...BEND

BASIC 7.0 is Commodore's best BASIC language to date, and includes a number of new commands. One such command set is BEGIN...BEND. These commands give you the option of calling up a block of commands and/or instructions, as opposed to IF...THEN, which limits you to a program line.

Programming BEGIN...BEND (and IF...THEN) permits something almost unheard-of in BASIC programming—*structured programming*. These blocks make the program listing more "readable", and the blocks lower the odds of logical errors. The DO...LOOP instruction supports this structured programming concept. More on this later.

Let's go back to the NUMBER GUESS program a few pages back, and utilize BEGIN...BEND:

```
10 REM NUMBER GUESS
20 SCNCLR:PRINT
30 PRINT"INPUT TWO NUMBERS FOR THE ";CHR$(17)
40 PRINT"UPPER AND LOWER LIMITS."
50 INPUT"LOWER LIMIT";L
60 INPUT"UPPER LIMIT";U
70 N=INT((U+1)*RND(1))+L
80 INPUT"YOUR GUESS";GN
90 IF GN<>N THEN BEGIN
100 : IF GN>N THEN PRINT "MYSTERY # IS SMALLER."
110 : IF GN<N THEN PRINT " MYSTERY # IS LARGER."
150 BEND:GOTO 80
160 SCNCLR:PRINT" HURRAH! YOU GUESSED IT!"
170 PRINT"ANOTHER ROUND (Y/N)?"
180 INPUT A$
190 IF A$="Y" OR A$="YES" THEN 20
200 END
```

The first few lines are unchanged. Line 90 is the first altered spot—IF...THEN has been changed to BEGIN. Lines 90-150 make up the block that is initialized if GN is unequal to N. When BEND is reached (line 150), the routine returns to line 80. When you finally give a number equal to GN, the program executes lines after BEND.

You don't have to put that GOTO command in the same line as BEND—you could add another line (155, for example) with GOTO instead. You'll still get the same result.

Exercises

1. Write a program which calculates annual earnings after taxes (33 percent or 51 percent). The ceiling amount is at $50,000; i.e., all amounts over 50,000 must be in the 51% bracket. Output should appear with accompanying text explaining amounts.

2. Write a program that computes the sum of all numbers from 1 to 100.

3. Write a program that gives you six random numbers from 1 to 49.

4. Which numbers are output by the following program? Answer without typing the program into the computer.

```
10 A=7
20 A=A+5:Z=Z+1
30 IF Z<9 THEN 20
40 PRINTA,Z
50 END
```

5. Write a program that searches for a string within a larger string. Let the string to be searched (A$) equal "INFORMATION", and the partial string (B$) equal "FORMAT". The catch is—we don't want you to use INSTR.

3.2.3 FOR...TO...NEXT

We have already created a loop using the IF...THEN command set. You'll recall that it was used to count up numerically, until the number stated was passed. Dependent on the condition tested by the loop (true or false), the routine went to a certain line in the program. Of course, for numbers and input, this can get tedious. A solution: the FOR...NEXT loop.

An example of a FOR...NEXT loop:

```
10 REM OUTPUT SQUARES OF FIRST TEN NUMBERS
20 SCNCLR:PRINT
30 PRINT"OUTPUT OF FIRST TEN SQUARES";CHR$(17)
40 FOR I = 1 TO 10
50 PRINT I;"SQUARED =";I*I
60 NEXT I
70 PRINT"DONE."
```

The principle here is similar to that of the IF...THEN function, but FOR...NEXT makes loops and counters much simpler to program.

I is the floating variable, assigned a starting value (1). The starting value is incremented by 1 until the end value is surpassed. Every command between FOR and NEXT is executed for as many times stated by the loop. Starting and ending values can be numbers, variables and arithmetic expressions.

Some examples:

```
10 A=10:B=20
20 FORZ=A TO B
30 PRINT Z;
40 NEXT Z
50 END
```

A and B are initialized as variables; line 20 puts these variables into a loop. Line 30 gives the value of Z until Z is greater than 20. This procedure can also be done with the IF...THEN command:

```
IF Z > 20 THEN 50
```

You can check to see what Z is doing after the program is done. Typing PRINT Z in direct mode returns a value of 21 for Z. The next example also uses arithmetical expressions:

```
10 A=10:B=15:C=5
20 FOR Z=A TO A+B-C
30 PRINT Z;
40 NEXT Z
50 END
```

The major difference between this and the first example is the end value—A+B-C.

If you use an increment larger or smaller than 1, use STEP. Look at the example below:

```
10 REM EVEN NUMBERS BETWEEN 2 AND 20
20 FOR I=2 TO 20 STEP 2
30 PRINT I
40 NEXT I
50 END
```

The floating variable, the starting value, the end value and even the value of the step size can be conveyed as negative numbers. How about a countdown?

```
10 REM COUNTDOWN
20 FOR I=20 TO 0 STEP -1
30 PRINT I
40 NEXT I
50 END
```

Starting the program will send the countdown winging by, faster than your eye can see; normal countdowns are marked in seconds. We can fix that—with another FOR...NEXT loop. We can place loops inside one another (this is known as *nesting loops*):

```
10 REM COUNTDOWN
20 FOR I=20 TO 0 STEP -1
30 PRINT I
40 FOR Z=0 TO 1000
50 REM DELAY LOOP
60 NEXT Z
70 NEXT I
80 END
```

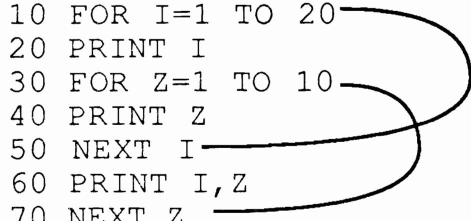
Type this program in (without the extra graphics) and RUN it. Now it moves in seconds, thanks to a timeloop in lines 40-60. Timeloops are often used in text screen programming, so that the reader has time to read the text before the program goes on to the next screen.

There's an alternative to FOR..NEXT timeloops, peculiar to BASIC 7.0: The SLEEP command halts the computer for a specified number of seconds (up to 65535). So SLEEP 10 stops the system for 10 seconds.

Timeloops within a program should only be used if you have an understanding of nested FOR...NEXT loops. It is evident that other BASIC commands can be placed within a loop. Let's look at the above program. Line 20 starts the loop with I=20. Line 30 shows the current value of I. Line 40 is the start of the second loop, the NEXT of which can be found at line 50. This second loop is finished before the first loop.

One thing you must remember: Never cross loops! That is, first loop must be closed last, and the last to be opened is the first to close. Here is a "non-program" to show you what NOT to do:

WRONG!!



```
10 FOR I=1 TO 20
20 PRINT I
30 FOR Z=1 TO 10
40 PRINT Z
50 NEXT I
60 PRINT I,Z
70 NEXT Z
```

WRONG!!

When you have several nested loops closing at the same time, there is no need to keep typing NEXT:NEXT:NEXT. You can combine closures using the variable names for each loop in the correct sequence.

```
10 FOR I=1 TO 10
20 FOR Z=1 TO 10
30 PRINT I;Z
40 NEXT Z,I:REM NOTE THE SEQUENCE—IN PROPER ORDER
50 END
```

Line 40 has only one NEXT, but acts for two loops using the variable names Z and I.

One mistake often made by beginners is made in jumping into a loop between FOR and NEXT, rather than before FOR. In most cases, a loop containing several lines of code can be jumped to, as long as it can be jumped from again. The error usually shows up the first time a program is run:

```
?NEXT WITHOUT FOR ERROR IN [line number]
```

If the programmer had used a program flowchart, this would not have happened. Another slipup occurs in neglecting to put a negative step value where necessary:

```
10 FOR A=5 TO 1
20 PRINT A
30 NEXT A
40 END
```

The STEP-1 has been left out of line 10; all this program will give is a value of 5. If you wish to end a loop prematurely, you can raise the floating variable. Normally, there is a starting value and an ending value. We can, by the use of an IF...THEN construct, shift the variable up to the value of the end number:

```
10 REM INCREMENT THE FLOATING VARIABLE
20 FOR A=0 TO 20
30 PRINT A
40 IF A=12 THEN A=20
50 NEXT A
60 END
```

This program really makes no sense, when you come right down to it; normally the loop would count up to 20. In line 40, though, when A reaches 12, A is immediately increased to 20, and the program halts. This method is rarely used. More practical is the ability to input starting and ending values for a loop:

```
10 INPUT"INPUT A WORD";A$
20 FOR A=1 TO LEN(A$)
30 PRINT LEFT$(A$,A)
40 NEXT A
50 FOR A=LEN(A$) TO 1 STEP -1
60 PRINT RIGHT$(A$,A)
70 NEXT A
80 END
```

Start the program, and type in your name <RETURN>. You'll see an output similar to those we worked with in the chapter on string functions, only here we use a FOR...NEXT loop to perform the string-length adjustments. That means that the loops control the length of the string.

The most important points about FOR...NEXT loops:

1. For every FOR, there must be a NEXT command as well. Several loops can be closed by a single NEXT command, if variable names are used in proper sequence and separated by commas (e.g., FOR A... FOR Z... FOR X/ NEXT X,Z,A).

2. Loops should not be jumped into in "midstream", or error messages can be expected.

3. The starting value of a positive step must not be larger than the end value, or the loop will only run once. The same goes for negative step values.

4. FOR...NEXT loops will normally run until the floating variable value is greater than the end value.

These rules only apply to BASIC 7.0 on the Commodore 128. Other BASIC dialects may have some minor differences—check the manuals on other machines for their unique usages.

3.2.4 Looping with DO...LOOP

This combination is yet another loop command grouping. The major difference between this and FOR...NEXT lies in increased flexibility. No specific increment (step size) is needed in DO...LOOP.

The simplest form of this loop begins with DO and ends with LOOP. All instructions inside this DO...LOOP block are executed permanently (or until you press <RUN/STOP> or <RESET>, or switch the power off). An endless loop is avoided by inserting the condition EXIT. This allows the loop to stop, and execute the next program command. We have already mentioned the flexibility of DO...LOOP: Let's look at a sample program:

```
10 DO
20 :A=INT((101)*RND(1))
30 :Z=Z+1
40 :IF A=100 THEN EXIT
50 LOOP
60 PRINT A,Z
70 END
```

DO...LOOP continues cycling through until A randomly reaches the number 100. When this occurs, the next command after LOOP is executed (in this case line 60, which prints the values of A and Z). Z is a *counter variable* here, denoting the number of times the loop cycles. Lines 20-40 begin with colons, which indicate the enclosed commands, thus making it easier to see the block structure of a loop.

The above example has only scratched the surface of this command's power. Two other instructions exist with DO...LOOP.

3.2.4.1 DO...LOOP with UNTIL and WHILE

UNTIL allows you to loop until a conditional or logical expression is verified as true. Example:

```
10 DO UNTIL A=100
20 : A=INT((101)*RND(1))
30 : Z=Z+1
50 LOOP
60 PRINT A,Z
70 END
```

Note that we've deleted line 40 from our last program, and added this UNTIL statement as the first line. Again, the loop will run until A is equal to 100. We could even place the UNTIL with the LOOP instead of at the beginning:

```
50 LOOP UNTIL A=100
```

The result is still the same.

WHILE is slightly different—the loop continues as long as a conditional or logical expression is true:

```
10 DO WHILE A<100
20 : A=INT((101)*RND(1))
30 : Z=Z+1
50 LOOP
60 PRINT A,Z
70 END
```

Now the DO...LOOP is executed as long as Z is less than 100.

We hope that the distinctions between EXIT, UNTIL and WHILE are clear to you now.

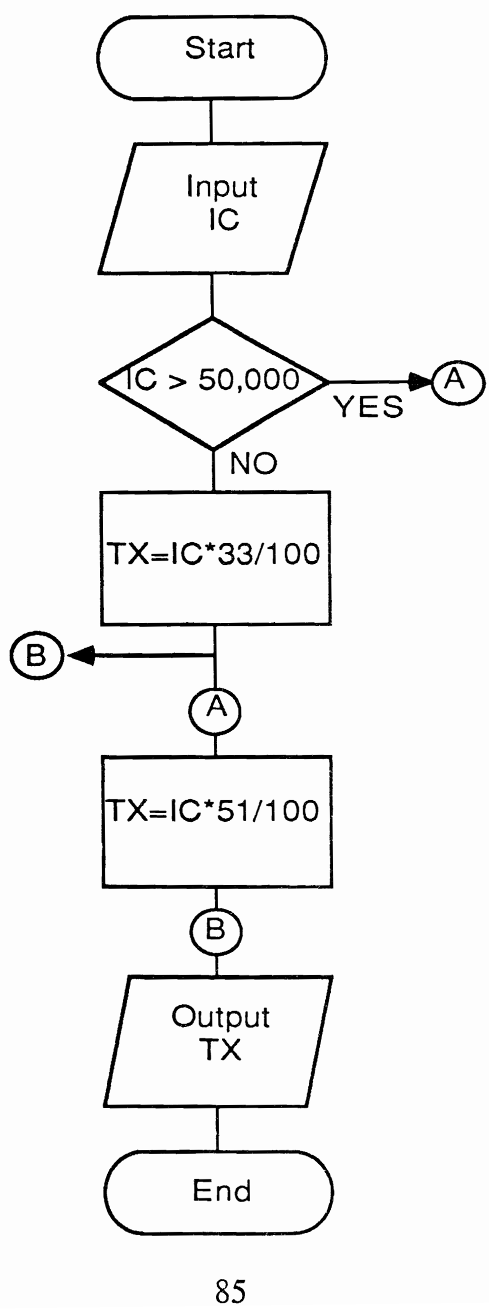
Here are a couple ground rules for using loops:

1) If the number of loops to be repeated must be stated at the outset, use the FOR...NEXT loop.

2) If the number of loop repetitions is unknown, use loops with IF...THEN or DO...LOOP.

There are exceptions—for an example, see the last program in Section 3.2.3. The rules here are simply stated as general guidelines.

Until now we have only worked with programming conditional and unconditional program jumps. Also, we have worked with materials within FOR...NEXT loops. Great—but how do we put these in our program flowcharts? The symbol used in logical branching is the diamond. Let's draw up a flowchart for our tax calculation program from the previous set of exercises.



You recall the START symbol and the INPUT symbol from earlier flowcharts. As already mentioned, a diamond is the symbol for a logical branch—there is a YES arm and a NO arm. In this example, a YES goes to the output connector A, and over to the input connector A. A NO creates a branch to out connector B, then to in-connector B, then to the end of the program.

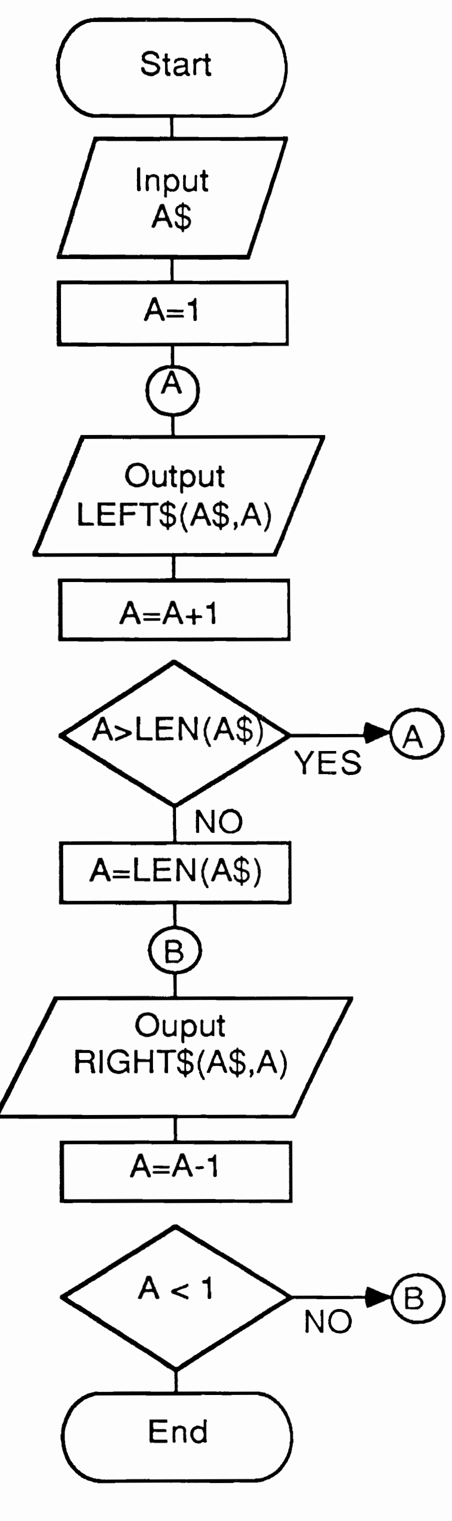
If the condition is false, 33% is calculated by way of connector B. Note that output B is placed before input A. Otherwise logical errors would occur.

So much for IF...THEN. Now let's put a FOR...NEXT loop into a flowchart. See the diagram on the next page.

These symbols should all be familiar to you now. The first triangle states the opening value of the loop at 1. Next follows the output of partial strings according to LEFT$(A$,A). The counter is incremented by one. The diamond checks to see whether the counter is larger than the number of characters in A$. If this is not the case, connectors A loop the program back.

When the counter is larger than LEN(A$), the second loop is called into play. The second loop arranges things in the opposite manner of the first loop (RIGHT$(A$,A), decrementing A), although it works the same as the first.

We have mentioned before that you should try to write flowcharts for every program. As the old saying goes, "Experience is the best teacher."


Start
Input
A$
A=1
A
Output
LEFT$(A$,A)
A=A+1
A>LEN(A$)
YES
A
NO
A=LEN(A$)
B
Ouput
RIGHT$(A$,A)
A=A-1
A < 1
NO
B
End

3.3 Computed jump commands

These jumps have the advantage of flexibility within a program itself. So far we have only seen jump commands that jump to a specific line, such as GOTO 100. IF...THEN requires a jump command for each new line number desired. For example:

```
10  REM JUMP TO SPECIFIC LINE
20  PRINT"GIVE ME A NUMBER BETWEEN";
30  PRINTCHR$(17)"1 AND 4"
40  PRINT
50  INPUT"WHICH NUMBER"N
60  IF N=1 THEN 100
70  IF N=2 THEN 200
80  IF N=3 THEN 300
90  IF N=4 THEN 400
100 PRINT"JUMP TO LINE 100"
110 GOTO 410
200 PRINT"JUMP TO LINE 200"
210 GOTO 410
300 PRINT"JUMP TO LINE 300"
310 GOTO 410
400 PRINT"JUMP TO LINE 400"
410 END
```

Note the IF...THEN constructs for each number between 1 and 4. This is adequate, but it's not great BASIC writing—those IF...THENs are slow going for the computer. An alternative is to use ON (*variable*) GOTO (*line number(s)*). The ON addition gives the program the option of branching to one of several line numbers. The variable can range from zero to the number of given line numbers. If the variable is not a whole number, the decimal places are ignored. Negative numbers give you an ?ILLEGAL QUANTITY ERROR. Values higher than the number of line numbers available cause the GOTO to pick up at the first command following the ON...GOTO command set. Here are a few short examples:

a)
```
10 ON Z GOTO 100,200,250,300
20 PRINT
 .
 .
 .
```

If the variable Z is equal to 1, the program jumps to line 100. Values 2 to 4 cause jumps to 200, 250 and 300, respectively. Z indicates the position of the individual line numbers. If Z is greater or less then the number of line numbers behind GOTO, the next command after GOTO is executed (the PRINT command in line 20, in our case).

b)
```
10 ON Z+3/4 GOTO 100,200,300
20 PRINT
 .
 .
 .
```

Here we have an mathematical expression instead of a variable... you can't do this with IF...THEN without including several such expressions. This way, you save programming time and memory. Let's rewrite the previous large program using ON...GOTO:

```
10  REM JUMP WITH ON..GOTO
20  PRINT"GIVE ME A NUMBER BETWEEN";
30  PRINTCHR$(17)"1 AND 4"
40  PRINT
50  INPUT"WHICH NUMBER"N
60  ON N GOTO 100,200,300,400
100 PRINT"JUMP TO LINE 100"
110 GOTO 410
200 PRINT"JUMP TO LINE 200"
210 GOTO 410
300 PRINT"JUMP TO LINE 300"
310 GOTO 410
400 PRINT"JUMP TO LINE 400"
410 END
```

Note that we have shortened the program considerably by using ON...GOTO. That may not look like much to you now, but when you're working on larger programs later on, you'll find ON...GOTO's compactness to be a godsend.

There is another programming technique used above that you'll find useful in larger programs—different program jumps are represented in horizontal lines on a flowchart. But first you need a rough idea of when each jump occurs. When you're putting it into the system, it's wise to have extra-large line numbers for inserting other program sections.

Another good usage of ON...GOTO is in the choice of line numbers. Using "smooth" line jumps (such as 100, 200, 300 in our example) seems to make the program quite a bit more readable to the human eye.

3.3.1 Sample program –Math Tutor

We've spent a lot of time doing nothing but learning BASIC. Now it's time to put our knowledge to work, and write a good-sized project. For example, say you want to write a math tutor program for your kids (or godchildren, or the neighbor's kids) that will work on the Commodore 128, and teach the four basic forms of arithmetic. Here are the features you want the program to have:

1) Choice of addition, subtraction, multiplication or division, or else the program ends.
2) The student has three chances to get the answer right.
3) The answer is displayed after three failed tries.
4) The student can input of the largest number that he/she is comfortable with.
5) The program should prompt the student whether to continue with the same form, or to try another.

The program listing for this math tutor follows. Don't be disturbed that the program numbers don't always run in steps of 10. Since this program is relatively long, we'll now give you commands for storing programs (although you probably know these already). When using a DATASETTE, type the following into your computer to SAVE this program to tape:

```
SAVE"MATH TUTOR"<RETURN>
```

The message PRESS RECORD AND PLAY ON TAPE appears—press these keys down on the DATASETTE.

The disk drive commands are a bit simpler: Put a formatted disk into the drive, and type in either

```
DSAVE"MATH TUTOR"<RETURN>
```

or

```
SAVE"MATH TUTOR",8<RETURN>
```

The program automatically SAVEs to diskette.

Calling the saved program back from your tape or diskette is also relatively easy. The LOAD command works much the same as the SAVE commands: LOAD"*file*" for tape, DLOAD"*file*" or LOAD"*file*",8 for disk.

Now for the **Math Tutor** program.

```
5 REM MATH TUTOR 3.3.1
10 SCNCLR:F=0
20 PRINT
30 PRINT TAB(12)"MATH TUTOR"
40 PRINT:PRINT
50 PRINT TAB(12)"YOUR CHOICES:"
60 PRINT
70 PRINT TAB(12)"[1] - ADDITION"
80 PRINT
90 PRINT TAB(12)"[2] - SUBTRACTION"
100 PRINT
110 PRINT TAB(12)"[3] - DIVISION"
120 PRINT
130 PRINT TAB(12)"[4] - MULTIPLICATION"
135 PRINT
140 PRINT TAB(12)"[5] - END PROGRAM"
150 PRINT TAB(12);:INPUT"WHICH NUMBER";Z
160 IF Z<1 OR Z>5 THEN 10
170 ON Z GOTO 200,600,1000,1300,1600
200 REM **********
230 SCNCLR
240 PRINT TAB(10)"INPUT THE LARGEST NUMBER"
250 PRINT
260 PRINT TAB(10)"FOR ADDITION."
270 PRINT
290 PRINT TAB(10);:INPUT"LARGEST";GR
300 REM CREATE RANDOM NUMBERS
301 REM
310 A1=INT(GR*RND(1))+1
320 A2=INT(GR*RND(1))+1
330 REM COMPUTE RESULT
```

```
350 SCNCLR
360 PRINT
370 PRINT"HOW MUCH IS" A1 "+" A2 "=";
380 INPUT ES
390 IF ES=RE THEN PRINT:PRINT TAB(10)"RIGHT!!":F=0:GOTO 450
400 PRINT:PRINT TAB(10)"WRONG"
410 FOR I=0 TO 500:NEXT I
420 F=F+1
430 IF F<=2 THEN 350
440 PRINT
450 FOR I=0 TO 500:NEXT I
460 PRINT TAB(5)" THE ANSWER IS";RE
470 FOR I=0 TO 500:NEXT I
480 PRINT"ANOTHER PROBLEM (Y/N)?";A$
490 INPUT A$
500 IF A$="Y" THEN F=0:GOTO 300
510 GOTO 10
600 REM
610 REM SUBTRACTION
630 SCNCLR
640 PRINT TAB(10)"INPUT THE LARGEST NUMBER"
650 PRINT
660 PRINT TAB(10)"FOR SUBTRACTION."
670 PRINT
690 PRINT TAB(10);:INPUT"LARGEST";GR
700 REM CREATE RANDOM NUMBERS
701 REM
710 A1=INT(GR*RND(1))+1
720 A2=INT(GR*RND(1))+1
730 REM COMPUTE RESULT
740 IF A1<A2 THEN I=A1:A1=A2:A2=I
750 SCNCLR
760 PRINT
770 RE=A1-A2
780 PRINT"HOW MUCH IS" A1"-" A2"=";
790 INPUT ES
800 IF ES=RE THEN PRINT:PRINT TAB(10)"RIGHT!!":F=0:GOTO 860
810 PRINT:PRINT TAB(10)"WRONG!!"
820 FOR I=0 TO 500:NEXT I
830 F=F+1
840 IF F<=2 THEN 750
850 PRINT
860 FOR I=0 TO 500:NEXT I
```

```
870 PRINT TAB(5)"THE ANSWER IS";RE
880 FOR I=0 TO 500:NEXT I
890 PRINT"ANOTHER PROBLEM (Y/N)?";A$
900 INPUT A$
910 IF A$="Y" THEN F=0:GOTO 710
920 GOTO 10
1000 REM
1001 REM DIVISION
1010 SCNCLR
1020 PRINT TAB(10)"INPUT THE LARGEST NUMBER"
1030 PRINT
1040 PRINT TAB(10)"FOR DIVISION."
1050 PRINT
1070 PRINT TAB(10);:INPUT"LARGEST";GR
1079 REM
1080 REM CREATE RANDOM NUMBERS
1081 REM
1090 A1=INT(GR*RND(1))+1
1100 A2=INT(GR*RND(1))+1
1110 REM COMPUTE RESULT
1111 REM
1120 RE=    A1*A2
1130 SCNCLR
1140 PRINT
1150 PRINT"HOW MUCH IS" RE "/"A1"=";
1160 INPUT ES
1170 IF ES=A2 THEN PRINT:PRINT TAB(10)"RIGHT!!":F=0:
     GOTO 1240
1180 PRINT:PRINT TAB(10)"WRONG!!"
1190 FOR I=0 TO 500:NEXT I
1200 F=F+1
1210 IF F<=2 THEN 1130
1220 PRINT
1230 FOR I=0 TO 500:NEXT I
1240 PRINT TAB(5)"THE ANSWER IS";A2
1250 FOR I=0 TO 500:NEXT I
1260 PRINT"ANOTHER PROBLEM (Y/N)?";A$
1270 INPUT A$
1280 IF A$="Y" THEN F=0:GOTO 1090
1290 GOTO 10
1300 REM
1301 REM MULTIPLICATION
1310 SCNCLR
```

```
1320 PRINT TAB(10)"INPUT THE LARGEST NUMBER"
1330 PRINT
1340 PRINT TAB(10)"FOR MULTIPLICATION."
1350 PRINT
1370 PRINT TAB(10);:INPUT"LARGEST";GR
1379 REM
1380 REM CREATE RANDOM NUMBERS
1381 REM
1390 A1=INT(GR*RND(1))+1
1400 A2=INT(GR*RND(1))+1
1410 REM COMPUTE RESULT
1420 RE=A1*A2
1430 SCNCLR
1440 PRINT
1450 PRINT"HOW MUCH IS" A1"*"A2"=";
1460 INPUT ES
1470 IF ES=RE THEN PRINT:PRINT TAB(10)"RIGHT!!":
     F=0 :GOTO 1540
1480 PRINT:PRINT TAB(10)"WRONG!!"
1490 FOR I=0 TO 500:NEXT I
1500 F=F+1
1510 IF F<=2 THEN 1430
1520 PRINT
1530 FOR I=0 TO 500:NEXT I
1540 PRINT TAB(5)"THE ANSWER IS";RE
1550 FOR I=0 TO 500:NEXT I
1560 PRINT"ANOTHER PROBLEM (Y/N)?";A$
1570 INPUT A$
1580 IF A$="Y" THEN F=0:GOTO 1390
1590 GOTO 10
1600 SCNCLR
1610 END
```

Let's examine the important sections of this program.

Lines 100-150 comprise the "menu" section. A menu gives the user a choice of routines/options, often at the press of one key. No good program should be without one.

Line 150 accepts numerical input representing the menu selection desired. Line 160 checks the input for allowable numbers. This is a good example of a logical operator—if the number is less than 1 or greater than 5, the program branches back to line 10. It can fulfill only one condition thanks to the combination with OR.

Line 170 uses the ON...GOTO command. If Z=1, then the system goes to line 200; Z=2, then 600, etc. ON...GOTO has just saved us from writing five separate IF...THEN sequences.

Lines 200-220 point out the addition section for the reader. REMs make programs readable—and also easier to debug and modify. Remember, though, to put REMs at the start of such a line. Syntax errors can occur if the system runs into just "ADDITION", or a bunch of asterisks.

Line 230 clears the screen. The next line asks the user for a number, which will be the upper limit for terms in addition. Next, the two random numbers A1 and A2 are added, and the problem displayed onscreen (line 370). The semicolons aren't absolutely necessary, as you can see in this example. The user types in the answer (line 380); line 390 checks the sum for accuracy. If the answer is wrong, a blank line is placed onscreen, followed by the message WRONG!!. Line 410 is a timeloop, giving the user half a second to look at the screen.

You could just as easily have used SLEEP 1 instead, but we're still trying to write in "generic BASIC". The counter in line 420 counts the number of wrong answers given by the user. Until the counter reaches three, the system will branch back to line 350, and do the problem again. Three wrong tries sends the program on to line 440.

A correct answer jumps the program execution from line 390 to line 450. The program momentarily goes into a delayed timeloop (line 450), the answer is displayed (line 460) and the user is asked if he/she would like to do another problem (line 480). Typing in Y resets variable F to zero, and returns the program execution back to line 300. Two new random numbers are created, and a new problem is formulated. Pressing any key other than Y

returns the program execution to the menu block (line 10). Here too, the number F is reset to zero. If we forgot to reset F, the counter might give the answer away after only two or even one wrong answer. Keep this counter reset in mind whenever you use counters in your programming.

The remaining sections (subtraction, division and multiplication) work on essentially the same principle as the addition section. The following paragraphs focus on the differences, starting with subtraction.

Line 740 calculates the result of the subtraction. Since we only want positive results (a positive number), we want to be sure that a lesser number is subtracted from a greater number (in other words, A1 must be greater than A2). If it happens that A2 is greater than A1, the values must be exchanged with one another. When this is the case, the value of A1 is temporarily stored into I.

This exchange isn't simply a matter of writing A1=A2:A2=A1. All you would get then is an error. This equation makes A1=A2, and A2=A1, but now A2 is already equal to A1.

So, we temporarily put A1 into another variable: I=A1. Now we exchange values between A1 and A2: A1=A2. Finally, we call the contents of I back: A2=I. This "interim storage" is important—we'll come back to it later.

In the division section, we use a little trick to get an integer result. Line 1120 finds the answer to A1*A2. Next, problem is displayed as RE/A1. The end result can only be a whole number.

There are no problems in the multiplication section—it's pretty much the same as the addition section.

Now we've had some experience with ON...GOTO, and covered the essentials of this math program. Before quiz time, we'll look at one more item—the TRAP command. First, though, a few ground rules for using ON...GOTO.

1) The value following ON can be a number, a variable or a mathematical expression. This value determines the position of the line number in the set of numbers that follows GOTO (1 is the first line number, 2 the second, etc.).

2) If this value is larger or smaller than the amount of line numbers listed, then the system goes on to the next command after GOTO.

3) Values <0 or >255 call up an ?ILLEGAL QUANTITY ERROR IN (*line number*).

4) ON...GOTO can replace several IF...THEN constructs.

3.3.2 Program jumps with TRAP

TRAP is a special variant of ON...GOTO. It lets you handle and/or change errors within a program. TRAP is usually used at the start of a program. The computer "memorizes" the line number to which the TRAP will go. All errors and error messages that occur after TRAP send program execution to the specified line.

Unlike ON...GOTO, TRAP only makes allowances for one line number. This is enough to handle a small error routine—but when several errors occur, how do we know which is which? BASIC 7.0 has two special variables—ER and EL. Say you type in PRILT instead of PRINT and press the <RETURN> key. You'll get a syntax error message. Now type PRINT ER—the error number is displayed (in this case, 11).

ER gives an error message without interrupting the program. EL gives the program line in which the error occurred.

This sort of error handling has an advantage—steps can be taken toward correcting the problems as the program is running. For example, if an error occurs in a database program, the error can automatically cue the closing of all files (so that no data is lost), before the error message is displayed.

It's not usually necessary to end the program after an error. You can continue the program right where it stopped after the error with RESUME. RESUME (*line number*) picks up the program execution at the specified line number where the error occurred. The program below is an illustration:

```
10   TRAP 1000
20   FOR I=-1 TO 3
30   ON I GOTO 200,300
40   NEXT I
100 END
200 SCNCLR
210 PRINT"LINE 210"
220 SLEEP 3
230 GOTO 40
300 SCNCLR
310 PRINT"LINE 310"
320 SLEEP 3
330 GOTO 40
1000 REM ERROR HANDLING
1010 SCNCLR
1020 PRINT"ERROR";ER;" IN LINE";EL
1030 SLEEP 3
1040 RESUME 40
```

There's a program error in line 20: the -1 in variable I. Normally the ON...GOTO branches to the line numbers determined by I, but as we already know, ON...GOTO doesn't work with negative numbers. Thus, the system jumps to the error trap at line 1000, presents us with the error number and line number, and resumes at line 40. It has yet to run lines 200 and 300, ending abruptly at line 100.

As you can see from our example, the principle of TRAP is fairly simple. Then again, the use of ER can be confusing. BASIC 7.0 gives us another option. The ERR$(X) function displays an "original error message" during a program run. The X in ERR$(X) is the error number, in a range between 1 and 41.

```
10 FOR I=1 TO 41
20 PRINT ERR$(I)
30 NEXT I
40 END
```

The previous program can use ERR$ by changing line 1020:

```
1020 PRINT "?";ERR$(ER);" ERROR IN LINE";EL
```

Isn't that a lot easier to follow?

It's time to test what you've learned in this chapter. Check your answers to the following problems against those at the end of the book. Take another look at the five points of programming, and reread this chapter if necessary.

Exercises

1) Write a program which sums up a given value in "harmonic sequence" (1 + 1/2 + 1/3 + 1/4 + 1/5 + ... + 1/n). After every 50 additions, the number of additions should be displayed (50,100,150 additions, etc.). In addition, the necessary results will be given.

2) Write a program which calculates the working zero-place of a quadratic equation in the form $AX^2+BX+C=0$. You can get the result through these formulae:

   $$x1 = (-B + SQR(B^2-4AC))/2A$$
   $$x2 = (-B - SQR(B^2-4AC))/2A$$

   There is no real solution to B^2-4AC; bear that in mind when writing this program.

3)
```
10  REM QUIZ RE ON...GOTO
20  INPUT"INPUT A NUMBER";N
30  ON N GOTO 100,150,400:SCNCLR
40  PRINT"NUMBER YOU GAVE IS ILLEGAL."
50  END
100 PRINT"LINE 100"
110 END
150 PRINT"LINE 150"
      .
      .
```

   What will this program do if you enter a 4 for N? Figure it out without typing this program into your computer.

3.4 Program control with GET

So far we've used INPUT to send data to our programs. INPUT has the advantage of being easy to integrate into a program. Also, it can be used for text output. INPUT's disadvantage lies in its lack of recovery—say, inputting an improper character, or a letter instead of a number. These errors result in the error message ?REDO FROM START.

Furthermore, you can't input commas unless the INPUT statement is set up for multiple variables. This is not to denigrate the INPUT statement—it's fine for some applications, but there are times when GET can handle input much more elegantly.

3.4.1 Data entry with GET

GET, like INPUT, belongs to the class of commands that simply cannot be used in direct mode—in other words, commands that only work within a program. Trying to use such commands in direct mode only results in the error message ?ILLEGAL DIRECT ERROR.

The Commodore 128 has a keyboard buffer, as did its predecessors (PET, VIC-20, C-64, Plus/4). The keyboard buffer has a small amount of memory capable of temporarily holding 10 characters at a time. This means that if the computer is busy doing something else, you can enter a maximum of 10 characters on the keyboard, which the computer will deal with as soon as its current task is done. Put the following "program" into the computer:

```
10 FOR I=0 TO 10000:NEXT I
```

RUN the program (a ten-second timeloop). While the computer is running, type in T E S T. These letters will appear at the conclusion of the loop. TEST was stored in the keyboard buffer, and executed as soon as possible.

GET looks for characters in the keyboard buffer. If it discovers any characters there, the first character is read and put into the variable following GET. An empty buffer puts a zero into the variable. The GET statement delays the program flow until characters are sensed in the keyboard buffer,

then continues the program. As we stand, we cannot use GET for any data transmission—but we can write a loop to show the characters seen by GET:

```
10 GET A$:IF A$ = "" THEN 10
20 PRINT A$
```

Give this program a try. Please note that GET, unlike INPUT, does not display a cursor. This program patiently waits at line 10 until you press a key. Whatever key is pressed, the GET statement reads it from the buffer. The IF...THEN in line 10 looks to see if a key has been pressed (the two quotes with nothing between them, ""). Note that there are two quotation marks and nothing more. Pressing a key prints the character onscreen, and sends the program back to line 10.

To see how fast GET really is, let's make a small change to the above lines:

```
10 GET A$:Z=Z+1:PRINT CHR$(19) Z:IF A$=""THEN 10
20 PRINT A$
```

We have added a counter (Z) to line 10, which counts the number of loops made until a keypress is "seen". PRINT CHR$(19) does the same thing as the <CLR/HOME> key.

Now we're in a position to select certain keys, i.e., we can only allow certain keys to be used for GET. Exceptions to the rule are <RUN/STOP> and <RESTORE>, as they aren't readable as CHR$ codes.

```
10 REM KEY SELECT WITH GET
20 SCNCLR
30 GET A$:IFA$="" THEN 30
40 IF A$=CHR$(65)THEN PRINT"A KEY":GOTO 30
50 IF A$=CHR$(66)THEN PRINT"B KEY":GOTO 30
60 IF A$=CHR$(69)THEN END
70 PRINT"YOU CAN ONLY PRESS THE A KEY,"
75 PRINT"THE B KEY, OR THE E KEY (THE"
80 PRINT"E KEY ENDS THE PROGRAM)."
90 GOTO 30
```

Delete the old program with NEW, and type this one in. When you start it, you will only have access to three keys—the rest won't do anything (including <SHIFT><CLR/HOME>).

Pressing any key other than A, B or E cycles the program through lines 70 and 80, then sends the program back to line 30. You have an infinite choice of functions after the THENs in this routine—i.e., THEN can be recoded to perform any task.

The next program displays the ASCII value of a character, along with the character itself. Control characters and keys (color, cursor movement, etc..) are set up in a special line for color and text conditions.

```
10 REM DISPLAY THE ASCII VALUES
20 SCNCLR
30 GET A$:IF A$=""THEN 30
40 PRINT TAB(6)A$ TAB(12)ASC(A$)
50 REM OLD SCREEN CONDITION
60 PRINT CHR$(153) CHR$(142)
70 GOTO 30
```

Lines 10 through 30 should look pretty familiar to you by now. Line 40 is new, though. The A$ (the character) is indented six spaces, when the character is a visible one. If you change the text color to the screen color, obviously the character will be invisible. The second part of line 40 prints the ASCII value of A$. Line 60 retains the old text condition (CHR$(153)) and keeps the program switched into upper case/graphic mode (CHR$(142)). Line 70 jumps back to line 30.

3.4.2 Reading the keyboard with GETKEY

GETKEY works a lot like GET, except the former will allow you to input several individual keys.

```
10 GETKEY A$,B$
20 PRINT A$,B$
```

Line 10 waits for two characters (arranged as A$ and B$); line 20 prints these characters onscreen. The next example reads four characters into B$, and prints them out.

```
10 REM FOUR CHARS READ IN WITH GETKEY
20 FOR I= 1 TO 4
30 GETKEY A$
40 B$=B$+A$
50 NEXT I
60 PRINT B$
70 END
```

This program designates the four characters with the help of a FOR...NEXT loop. Line 30 looks for keypresses and/or characters in the keyboard buffer. Press a key, and the character is placed in A$; line 40 joins the characters from A$ together to form B$. Line 50 ends the FOR...NEXT loop; when four rounds are completed, B$ is printed in its entirety.

You could use this technique to read in four-digit numbers with GET or GETKEY. To do so, you would be forced to input individual numbers, and limit yourself to four-digit numbers.

The most important point in GET/GETKEY is the selection of individual keys. This allows for simple menu control. Let's takes our "math tutor" program from a few pages back, and replace the INPUT commands with GET statements. The next page shows the first few lines of the "new" version (up to line 320). Load in the original program and enter DELETE 120-320 to remove the old lines. Now enter these new lines. Pay careful attention to the IF...THEN additions.

```
120 PRINT
130 PRINTTAB(12)"(4) MULTIPLICATION"
133 PRINT
135 PRINTTAB(12)"(5) END"
140 PRINT
150 PRINT"WHICH NUMBER?"
155 GETA$:IFA$=""THEN155
160 IF VAL(A$) < 1 OR VAL(A$) > 5 THEN 155
170 ON VAL(A$) GOTO 200,600,1000,1300,1600
200 REM ***************
210 REM ADDITION
220 REM ***************
230 SCNCLR
240 PRINTTAB(10)"INPUT THE HIGHEST NUMBER"
250 PRINT
```

```
260 PRINTTAB(10)"TO BE USED FOR ADDITION."
270 PRINT
290 PRINTTAB(10)"HIGHEST?"
291 FOR I=1TO3
293 GETA$:IFA$=""THEN293
295 IFASC(A$) < 48 OR ASC(A$) > 57 THEN 293
297 B$=B$+A$
298 GR=VAL(B$)
299 REM
300 RANDOM NUMBERS
301 REM
310 A1=INT(GR*RND(1))+1
320 A2=INT(GR*RND(1))+1
```

You can plainly see the alterations made at lines 150-170. The original program had INPUT statements here; we have changed it slightly to require one number keypress, nothing more. Line 155 contains the GET statement and a "null string" (""). Line 160 tests for valid numbers (<5 and >1)—invalid input puts the user back in line 155. The variable A$ is converted to a number using VAL(A$), and the program branches appropriately with ON...GOTO (line 170).

That's the extent of converting GET to serve instead of INPUT. RUN this program, and press a few characters or numbers other than 1 to 5. Now, press <SHIFT><CLR/HOME>. See the difference between GET and INPUT?

One small snag in using GET: We don't see where the data input appears onscreen, nor do we see what the data might be. This problem doesn't exist with INPUT, since we have a blinking cursor, and the statement waits until the <RETURN> key is pressed. We will discuss this bug in a later chapter.

3.4.3 Function key layout

The Commodore 128, like its immediate relatives (the C-64, VIC-20, etc.), has eight function keys (F1-F8). Unlike the other Commodore machines, these function keys have built-in commands. Typing the command KEY calls up the following:

```
KEY 1,"GRAPHIC"
KEY 2,"DLOAD"+CHR$(34)
KEY 3,"DIRECTORY"+CHR$(13)
KEY 4,"SCNCLR"+CHR$(13)
KEY 5,"DSAVE"+CHR$(34)
KEY 6,"RUN"+CHR$(13)
KEY 7,"LIST"+CHR$(13)
KEY 8,"MONITOR"+CHR$(13)
```

These are the onboard functions. You may be wondering about the `CHR$()` markings; `CHR$(13)` is the <RETURN> key, which causes the line to execute immediately, and `CHR$(34)` is a quotation mark for filenames (`DSAVE"` & `DLOAD"`).

You can change these functions to your own commands, simply by typing `KEY`, the function key number, and the commands required. Here, for example, is a 40/80 column toggle (<ESC>-X) on function key F6:

```
KEY 6,CHR$(27)+"X"
```

This next program alters C-64 BASIC programs so that the C-64 function key commands will work on the 128.

```
10 REM C-64 FUNCTION KEYS ON C-128
20 FORI=1 TO 8
30 KEY I,CHR$(132+I)
40 NEXT I
50 END
```

3.4.4 Reading function keys with GET

Another possibility of `GET`'s use needs some clarification. As we just saw, the C-128 has four function keys able to handle eight functions. The beginner can't really use these keys, mostly because little has been said about how to program them on your own. Here is a small sample program to give you some ideas. Run the program above to set the new definitions before running the next program.

```
10 REM READING FUNCTION KEYS WITH GET
20 GET A$:IFA$=""THEN 20
30 REM F1
40 IF A$=CHR$(133) THEN SCNCLR
50 REM F2
60 IF A$=CHR$(137) THEN PRINT CHR$(144);"F2"
70 REM F3
80 IF A$=CHR$(134) THEN PRINT CHR$(5);"F3"
90 REM F4
100 IF A$=CHR$(138) THEN PRINT CHR$(153):END
110 GOTO 20
```

This program assigns functions to F1 through F4. F1 clears the screen; F2 changes print color to black and prints "F2"; F3 changes the text to light blue and prints "F3"; and the program ends by pressing F4. F2, F4, F6 and F8 can be had by pressing <SHIFT> and the odd numbered key (F1,3,5,7).

This next application program tests your reaction time. You must press a key as quickly as possible at the given signal. To re-test, press F1.

```
5   REM 3.4.4
7   REM REACTION TEST
10 KEY 1,CHR$(133)
20 FOR I=1 TO 11:CU$=CU$+CHR$(17):NEXT I
30 FOR I=1 TO 19:CU$=CU$+CHR$(29):NEXT I
40 SCNCLR
50 FOR I=0 TO INT(10000*RND(1))+1:NEXT I
60 PRINT CHR$(7);CU$;CHR$(209):TI$="000000"
70 GETKEY A$
80 PRINT CHR$(19)TI/60;"SEC."
90 GET A$:IF A$<>CHR$(133) THEN 90
100 GOTO 40
```

Lines 20 and 30 produce a string by means of two FOR...NEXT loops. This string gives out 11 cursor-down characters and 19 cursor-right characters (i.e., the cursor moves 11 lines down, and 19 columns to the right). Line 60 drops a period into the middle of the screen, and sounds a CHR$(7) (known as a <CONTROL>G, or <BELL> tone). Line 50 causes the point to appear any time from 1 to 10 seconds. Line 60 sets the "clock" to zero, and measures the time from the appearance of the point to a keypress (lines 70 and 80). The rest of the program shouldn't be too hard to follow.

3.5 FRE, POS, SYS, USR(X) and WAIT

These commands and functions are rarely seen in BASIC programming, which doesn't necessarily mean that they're unimportant items.

FRE tells you the amount of memory available, using the syntax FRE(X). Typing in PRINT FRE(X) (X equals either 0 or 1, dependent on bank of memory used) gives you the remaining memory. If no program exists, you'll get 58109. FRE(0) reads program memory, while FRE(1) looks at variable memory.

POS is not used in programs much at all—it tells the current cursor position in a program line. This limits it to a value between 0 and 79. Type in PRINT "TEST" POS(X);"TEST A" POS(X) in direct mode; the response should be TEST 4 TEST A 13 . The numbers are the cursor positions after the PRINT command was executed. So, the 13 follows the two strings.

SYS calls up the memory address specified, where there is (or should be) a machine language routine or program. Microprocessor control is no longer in the hands of BASIC, rather directly handled in machine code. SYS allows you to access machine code routines through BASIC. It helps to know the operating system before fooling around with this command. You can give up to four parameters for the accumulator, X-, Y- and status registers.

Example: The cold-start address–SYS 4*4096. Typing this in will set the computer up as if it had been switched on. Use this command cautiously.

USR(X) works something like SYS, with the distinction that a value associated with a machine language program can be given. It's primarily used in advanced BASIC and machine code, so the beginner would have no use for the command.

WAIT causes a program to pause until a certain value is taken into a certain memory location. Or, for that matter, you could say that a certain bit pattern should appear in a certain memory location. The next chapter will tell us how to get the computer to wait for a keypress without the use of GET.

3.6 PEEK and POKE

3.6.1 PEEK

The PEEK statement is used for looking at the contents of a specific memory location. The syntax reads PEEK(*address*). The (address) can be a number anywhere from 0 to 65535.

For example, let's see what's in memory location 1024. We type in PRINT PEEK(1024), which is a screen memory address (to be specific, the upper left-hand corner of the screen). If no characters are presently in this location, the number 32 (space) is displayed in response to our PRINT PEEK() command. We can also arrange this in variable form:

```
X=PEEK(1024)
PRINT X
```

Again, if that section of the screen is blank, we'll get back 32. Next, we'll PEEK into the first 78 memory cells in BASIC memory, starting with the address 7168 (which can't be used for high-resolution graphics). If you turn on a graphic page, start-of-BASIC automatically shifts to address 16384. Type this short program in:

```
10 REM 78 BASIC MEMORY LOCATIONS
20 FOR I=7168 TO 7246
30 PRINT PEEK(I);
40 NEXT I
50 END
```

Now RUN the program. A set of 78 numbers appears. You don't need anything more than a BASIC program like this to look at the "innards" of the Commodore 128. Interesting, isn't it?

But in fact, you will seldom use PEEK in programming. Nor will you need PEEK opposite POKE very much in BASIC 7.0—which we'll discuss next.

3.6.2 POKE

POKE writes a value to a specific memory location. POKE is necessary on the Commodore 64 (and is necessary on the 128, when the machine is set in 64 mode) to change the screen and border colors (locations 53281 and 53280, respectively). You can still control these locations in 128 mode, but you must first preface the POKEs with BANK 15 to set up the proper memory configuration.

Say that you wanted to change the entire screen to black. You would type in BANK 15:POKE 53280,0: POKE 53281,0 <RETURN>. Presto! The screen and border turn to black. Typing in BANK 0 <RETURN> returns us to the original memory configuration. Now, change the color of the lettering, using <CTRL>1 to 8, <C=>1 to 8, or a POKE command.

Memory location 241 lets you change lettering color (0 to 15). POKE 241,1 turns the onscreen cursor white. See your C-128 user's manual for the color numbers (0=black, 1=white, etc.).

Remember the WAIT statement from an earlier chapter? It's used to wait for a certain key or set of keys to be pressed. Location 208 is where the number of keys pressed is registered. This program should illustrate the point:

```
10 REM READ KEYBOARD WITH WAIT
20 POKE 208,0
30 WAIT 208,1
40 PRINT"KEY PRESSED"
50 END
```

Line 20 sets the address to zero, i.e., no keys have been pressed. The WAIT in line 30 pauses until something appears in location 208 (a key is pressed). When this occurs, the KEY PRESSED message appears onscreen.

There you have it—another method of reading the keyboard, aside from GET or GETKEY. Of course, POKE can be followed by a variable, and the value of the variable is written to that memory location. But remember—the value can be no larger than 255.

3.7 READ, DATA and RESTORE

Up until now, we have been dealing only with inputting data from the keyboard (INPUT and GET), which is then put into variable space. This is fine for small amounts of data, but when our programs work with larger quantities of material (e.g. sets of numbers or strings), these values would have to be input every time the program is run. But instead, we can use the READ and DATA statements.

DATA combines a list of materials. The individual items are separated from one another by commas. Data can be numbers, strings or both in combination.

READ does just what it says—it reads the DATA statements. The variable type stated in READ must match the data type in the list. String data can't be read by a READ statement set for numbers of integers.

It doesn't really matter where the DATA lines go in a program. You can put them at the beginning, middle, or end of a program—just as long as the READ statement precedes the DATA. Here is a short example—type in NEW (do this before typing in every new program) and type in the following:

```
10 READ X
20 PRINT X
30 DATA 50
40 END
```

You'll get 50 as a result of RUNning this program. Line 10 looks for the numerical DATA (line 50). Therefore, if the program runs into a READ statement, the program looks for the first DATA value. In this case the value is set up as a variable. The contents of X are printed onscreen (line 20). The DATA in line 30 has no influence on the program run. Now, change the program like this:

```
10 READ X,Y,Z
20 PRINT X,Y,Z
30 DATA 10,20,30
40 END
```

RUNning the program yields these results:

```
10   20   30
```

READ puts the first number in the DATA line into variable X, the second into Y, and the third to Z. Every READ access to DATA reads in the next value, thanks to pointers. This particular pointer points to the next element to be read out. Starting the program sets this pointer to the first DATA element, and moves along as needed (below the pointer is represented by a ^):

```
30 DATA 10,20,30
        ^
```

During READ, the pointer moves to the next element:

```
30 DATA 10,20,30
           ^
```

When this element is read, the pointer is incremented by 1. Once the pointer reaches the end of DATA, it remains "stranded" at the last element. **Note**: It will not reset automatically!

Another attempt at reading data will produce the error message:

?OUT OF DATA ERROR IN *(line number)*

So what now? There is a solution—known as RESTORE. RESTORE resets the pointer to the first DATA element. Now you can read the DATA lines as many times as you want. In the sample below, we are READing more DATA than is available:

```
10 READ A,B,C
20 PRINT A,B,C
30 DATA 10,20,30
40 READ D,E,F
50 PRINT D,E,F
60 END
```

You will get 10, 20, 30, and an ?OUT OF DATA ERROR IN 40. There are more READ statements than there are DATA statements, and the pointer is still set to the end of DATA. Add this line:

```
35 RESTORE
```

Now LIST the program:

```
10 READ A,B,C
20 PRINT A,B,C
30 DATA 10,20,30
35 RESTORE
40 READ D,E,F
50 PRINT D,E,F
60 END
```

RUN the program again—you get the following output:

```
10  20  30
10  20  30
```

RESTORE resets the pointer, and the numerical values D, E and F are taken from DATA. But suppose we don't want all the DATA on one line. We can solve that problem with a FOR...NEXT loop, and print each element on its own line.

```
10 FOR I=1 TO 3
20 READ X
30 PRINT X
40 NEXT I
50 DATA 10,20,30
60 END
```

As mentioned above, you can also use strings for DATA elements, as long as the data type matches the variable type. There are some exceptions in string DATA, however. You can only use commas, colons, semicolons, spaces, shifted characters, graphic characters and control characters within quotation marks. You must be sure of matching data and variable types, or you'll get a ?TYPE MISMATCH ERROR IN *(line number)*. This can happen when longer lists of data are written. For example, look at the problem below:

```
10 FOR I=1 TO 3
20 READ A,B,C$
30 PRINT A,B,C$
40 NEXT I
50 DATA 10,20,TEST 1,30,40,TEST 2,50,TEST 3,OK
60 END
```

The program runs fine through the first and second loops. The data was properly arranged according to READ (*number*, *number*, *string*). The third loop runs into an error, though (*number*, *string*, *string*). The system runs into a problem with treating "TEST 3" as variable B, and you get a ?TYPE MISMATCH ERROR IN 20.

Now we've seen a number of ways to put data into a program:

1) Input using INPUT, GET or GETKEY
2) DATA statements.

This data can be within programs, or even stored separately on cassette or diskette. If you wish to store data entered with INPUT, GET or GETKEY, you need to save this material as a separate diskette or cassette file.

One extremely useful service performed by DATA is the generation of machine language in a BASIC program. This is done with READ, DATA, and a good-sized FOR...NEXT loop. A machine-code program is POKEd in from the DATA, and started by a SYS command. The next program clears the screen, sets the screen color to black, and turns the text color to orange. These machine-code generator programs are also known as BASIC loaders. This BASIC loader is pretty simple, but is just meant to be a quick demo program:

```
10 FORI=8000 TO 8016
20 READ M
30 POKE I,M
40 NEXT I
50 DATA 169,0,141,32,208,141,33,208,169,8,141
   241,0,32,66,193,96
60 END
```

The machine language program starts at memory address 8000, which is in the middle of BASIC memory. You could put it virtually anywhere that has some free memory. Your BASIC programming can't overwrite this code.

How does this program work?? The built-in FOR...NEXT loop starts at the opening memory location. READ gets the values from the DATA line, and POKEs them into the memory. The program runs, and ends with READY. All you need do is type in SYS8000, and the screen clears and turns black, and the text turns orange.

The same results can be achieved in BASIC:

```
10 BANK 15
20 REM BLACK SCREEN
30 POKE 53280,0
40 POKE 53281,0
50 SCNCLR:REM CLEAR SCREEN
60 PRINT CHR$(129):REM ORANGE
```

Now you have some working information about DATA, READ and RESTORE.

Chapter 4

Advanced BASIC Applications

4. Advanced BASIC applications

4.1 Arrays

One of the biggest problems facing the beginning programmer is programming and handling *arrays*. The more complicated the array, the harder it is to manage. Even accomplished programmers have trouble handling arrays.

You'll learn a lot about arrays in this chapter—it's just a matter of study, really.

4.1.1 One-dimensional arrays

Suppose you want to write a program that calculates your monthly wages and divides it into 12 periods. This first example uses a loop to add up the number of monthly paychecks:

```
10 REM AVERAGE MONTHLY INCOME
20 REM FIGURE OUT 12 MONTHS
30 FOR I=1 TO 12
40 INPUT"MONTH'S INCOME";M
50 S=S+M
60 NEXT I
70 D=S/12
80 D=INT(D*100)/100
90 PRINT"AVERAGE COMES TO";
100 PRINT"$";D
110 END
```

Run the program—you'll be asked for 12 amounts. The end average will be rounded to the cent. The prompt appears repeatedly due to a FOR...NEXT loop. The amounts are then added up (line 50). Line 70 calculates the average, and line 80 rounds it off to the cent.

Now, suppose we wish to backtrack—how can we find out from this program how much we typed in for May? We can't... not in the last program, at least. However, we can open that option by creating 12 separate variables. Now we'll change the above program.

```
10 REM AVERAGE MONTHLY INCOME
20 REM INPUT INDIVIDUAL AMOUNTS
30 INPUT"MONTH'S INCOME 1";M1
40 INPUT"MONTH'S INCOME 2";M2
50 INPUT"MONTH'S INCOME 3";M3
60 INPUT"MONTH'S INCOME 4";M4
60 INPUT"MONTH'S INCOME 5";M5
80 INPUT"MONTH'S INCOME 6";M6
90 INPUT"MONTH'S INCOME 7";M7
100 INPUT"MONTH'S INCOME 8";M8
110 INPUT"MONTH'S INCOME 9";M9
120 INPUT"MONTH'S INCOME 10";M10
130 INPUT"MONTH'S INCOME 11";M11
140 INPUT"MONTH'S INCOME 12";M12
150 S=M1+M2+M3+M4+M5+M6+M7+M8+M9+M10+M11+M12
160 D=S/12
170 D=INT(D*100)/100
180 PRINT"AVERAGE COMES TO ";
190 PRINT"$";D
200 END
```

Now run the program. You'll see that it does the same work as the first program. Now ask for the system to print the amount for May (M5)—you should get your answer. This is assuming that the running month number coincides with the the variable number.

There is a problem here—using 12 variables is quite a waste of memory. Every variable must be used with a PRINT command, like this:

```
.
.
100 PRINT M1,M2,M3,M4,M5,etc.
.
.
.
```

Instead, we can use a variable with a floating index. Now we create a loop, and fix the variable I to represent all 12 months. This is called an *array*.

You'll recall our discussion of variables. The variables have indicators showing whether they are string, number, or whatever. An array can be limited to a certain number of elements. This is called the *index*. An index is found in parentheses following the array name. For example, `A(1)` is an array variable, or an indexed variable. Do not confuse `A(1)` with a variable such as `A1`!! There is a world of difference between `A(1)` and `A1`. Here is a sample of how the `A(I)` array would be used in our program:

A(1)	2334
A(2)	2333
A(3)	2345.65
A(4)	2344.34
.	.
.	.
.	.
.	.
A(12)	3433.20

You see that an array is quite similar to a table or chart. Our array has 12 "cells" into which a value can be placed. If we want to see the contents of the third cell, we just call up index 3 by typing `PRINT A(3)`. The result is the value `2345.65`. To use arrays, we must tell the 128 the size of the array, and how many elements it contains. The latter requirement is accomplished by the `DIM` statement (`DIM`ension). The syntax is:

`DIM` *Array name(# of fields)*

Our own array would look like this:

```
DIM A(12)
```

`A` is the array name, and `12` is the maximum number of fields allowed in the array. The `DIM` statement usually appears at the start of the program. Once an array is dimensioned, it remains that way; if you try to change it in mid-program, the onscreen result is:

?REDIM'D ARRAY ERROR IN *(line #)*

Our example is defined as an array for floating-point variables. You can also define string and integer arrays:

```
DIM DE$(15)
DIM GZ%(20)
DIM AB(12)
```

The rules here are the same as for arrays—no strings in an integer array, no floating-point in the other two, etc. If you need to, you can put a number of DIM statements on one line:

```
DIM A(12),B$(16),S%(20)
```

Two peculiarities should be mentioned here:

1) If you are using 11 elements or less in an array, you won't need a DIM statement. An automatic limit is set (equivalent to DIM A(10)).

2) DIM A(10) gives us 11 elements rather than 10 (DIM starts at zero, not 1).

In reference to item 1): if you want to save some memory, you can make your array smaller than the standard 11 elements (e.g., DIM X(5)).

We're going to have one more look at our monthly salary averaging program. But this time, the program uses arrays. You'll find it a good deal more elegant than the second version, and more efficient than either.

```
10 REM AVERAGE MONTHLY INCOME
20 REM USING ARRAYS
30 DIM M(12)
40 FOR I=1 TO 12
50 INPUT"MONTH'S INCOME";M(I)
60 S=S+M(I)
70 NEXT I
80 D=S/12
90 D=INT(D*100)/100
100 PRINT"AVERAGE COMES TO";
110 PRINT"$";D
```

The one thing we haven't provided for is a display of which month you are inputting. If we want to do this, we add the following lines to the program:

```
  .
  .
90  D=INT(D*100)/100
100 FOR I=1 TO 12
110 PRINT"INCOME MONTH "I,M(I)
120 NEXT I
130 PRINT"AVERAGES OUT TO ";
140 PRINT"$";D
150 END
```

Arrays written like this—`A(X)`—are known as one-dimensional arrays, since these arrays only have one index. This index can be a number, variable, or arithmetical expression. You can also make the index adjustable. For example, suppose you don't want the program to be exactly 12 months. You could add the following lines:

```
  .
  .
30 INPUT"HOW MANY MONTHLY CHECKS";Z
40 DIM M(Z)
  .
  .
```

The number of monthly payments dictates the size of the array.

If you attempt to dimension an array ambiguously (`DIM(X)`, e.g., which gives you no array name), you'll get this:

`?BAD SUBSCRIPT ERROR IN` (*line #*)

Redimensioning an array (trying to give it a new size) also yields an error message.

4.1.2 Examples of one-dimensional arrays

Sometimes it is necessary to clear an array after it has been filled. An entire array can be cleared within a program. Numerical arrays can be cleared by filling the elements with zeroes with this example program:

```
10 REM CLEAR NUMERIC FIELD
20 DIM X(6)
30 FOR I=1 TO 6
40 X(I)=0
50 NEXT I
60 END
```

We're clearing a field with 6 (or 7) elements. The FOR...NEXT loop has a starting value of 1 and an ending value of the maximum number of elements. (In our case the maximum number of elements is 6). The loop takes I as 1, 2, 3, 4, 5, 6. Line 40 sets each element to 0, and the index X is incremented by 1. The following happens:

```
X(1)=0
X(2)=0
X(3)=0
X(4)=0
X(5)=0
X(6)=0
```

The above program should give you a general idea of the principle of clearing an array. If you wish to clear a string array, you change the zeroes to "null strings". The reason you do this is to avoid errors in programs using the LEN function—to fill in remaining spaces within an element.

```
10 REM CLEAR STRING ARRAY
20 DIM Y$(6)
30 FOR I=1 TO 6
40 Y$(I)=""
50 NEXT I
```

The logic of this program runs much the same as the numerical array-clearing program. The exception here is that the elements are filled with null strings, rather than zeroes. Note: The two quotation marks appear side by side—with no space between the two.

Now we come to some examples which put number sequences into arrays with the help of FOR...NEXT loops. In the cases of three arrays having six elements, the following contents should be put in using our programs below:

a)	b)	c)
1	10	2
4	8	4
9	6	8
16	4	16
25	2	32
36	0	64

How can we do all that with FOR...NEXT loops?

Explanation of Example a)

What we want to do here is fill each element with its square:

```
10 DIMX(6)
20 FOR I=1 TO 6
30 X(I)=I*I
40 NEXT I
50 END
```

The function of this loop will be a little clearer to you once you see it lined up in individual steps:

```
                                         DIM X(6)
I = 1: X(1) = 1*1 = 1                    1
I = 2: X(2) = 2*2 = 4                    4
I = 3: X(3) = 3*3 = 9                    9
I = 2: X(4) = 4*4 = 16                   16
I = 1: X(5) = 5*5 = 25                   25
I = 2: X(6) = 6*6 = 36                   36
```

Each loop cycle increments `I` by one. `I` is used as the index for each element. At the same time, `I` is multiplied by itself, and the result placed in the element being processed currently. In this way the array is filled with squares of the index.

The same principle (i.e. the floating variables calculated using the loop) is used in example b).

Explanation of Example b)

This example installs elements in "backward" steps of two. The starting value of the first array is the number 10. We can't change the `FOR...NEXT` loop to get this countdown effect, since that would foul up the indexing. Instead, we'll rearrange the computations so that the index still increases while the element values decrease by two. The solution could look something like this:

```
10 DIM X(6)
20 FOR I=1 TO 6
30 X(I)=12 - 2*I
40 NEXT I
50 END
```

Line 30 reorganizes the elements—the larger the index (`I`), the smaller the number put into the element. To see the entire array printed out, add these lines to the above program:

```
50 FOR I=1 TO 6
60 PRINT X(I)
70 NEXT I
80 END
```

Explanation of Example c)

The numerical sequence in this example is generated in powers of 2. You'll remember that we discussed these numbers back in Chapter 2. Accomplishing this task is no problem, if we use 2 as the exponent of the index. See for yourself:

```
10 DIM X(6)
20 FOR I=1 TO 6
30 X(I)=2^I
40 NEXT I
50 END
```

You might want to use the lines that we added to Example b) to read the element numbers, and then check the results on paper. You should be getting a general idea of how these examples work by now—you'll run into these techniques in more complex programs.

We've been working with numerical arrays so far. But what about string arrays? Well, they're really not much more complicated than numbers. We can input string elements from the keyboard, or "load" an array with DATA and READ.

String arrays are used whenever names, addresses, or even numbers (entered as a string) are to be stored. Needless to say, string arrays are pretty versatile. Let's put some of these arrays to work. We're going to create a program to store the first names of your best friends. We really can't go any farther now, since we have yet to learn how to store this data on a mass storage medium (tape or floppy disk). This sample program is virtually useless as a database—it's simply a demonstration.

Do you have the number of your friends memorized? Usually, this just isn't the case, but with our little demo program, a specific array size must be declared in the DIM statement. If we stay within the limits of the program, everything works fine. However, going past the specified number of elements gives us a LIST FULL--SORRY message, instead of an error message.

```
10 REM FIRST-NAME BASIS
20 DIM Y$(6)
30 Z=Z+1
40 INPUT"FIRST NAME";Y$(Z)
50 PRINT"MORE INPUT Y/N ??"
60 GETKEY A$
70 IF A$ < > "Y" THEN 100
80 IF Z < 6 THEN 30
90 PRINT"LIST FULL--SORRY"
100 END
```

We've set up an array of seven elements (counting 0) in line 20. Line 30 contains the counter which increments by one on each input. Since we don't know exactly how many friends you have, a FOR...NEXT loop wouldn't work here—hence the counter. Line 40 prompts for your input, and arranges input according to the index Z. Line 50 asks if you have more input, and the program awaits your response (line 60). Line 70 compares your response with the letter Y. If the response is anything but Y, the program ends at line 100. More input can be sent on condition that Z is less than 6 (line 80). When the counter reaches 6, then we get the LIST FULL--SORRY message, and the program grinds to a halt.

If we hadn't installed this "safety device", a value larger than 6 would give us ?BAD SUBSCRIPT ERROR IN 40, and stopped the program. In other words, if we get Y$(7), the DIM statement (set for 6) doesn't acknowledge it. Avoid this sort of program interruption as much as you can. Use TRAP and an error routine if you're unsure of the outcome of a new routine.

This program doesn't have to stop here. The principle needed next should now be clear to you, though. We still want to read out the names of our friends—this involves output of the entire array. This can be handled like the "reading" code after example b), using FOR...NEXT loops.

Dimensioning an array to, say, DIM D$(200) would be more than enough for your own database programming. However, a one-dimensional array is not enough to handle large amounts of data. Let's look at an alternate way of inputting data, which supplies the array with data through READ and DATA statements.

There are times when you need the days of the week to be entered into your programs, but this can be a pain in the neck when you have to keep typing in dates. What about putting these dates into DATA statements, and just having the array read them at the beginning of the program? Here is a sample:

```
10 REM THE DAYS OF THE WEEK
20 DIM WD$(7)
30 FOR I=1 TO 7
40 READ WD$(I)
50 NEXT I
60 DATA MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
   FRIDAY, SATURDAY, SUNDAY
70 REM OUTPUT Y/N
80 PRINT"ARRAY OUTPUT Y/N"
90 GETKEY A$
100 IF A$ < > "Y" THEN 140
110 FOR I=1 TO 7
120 PRINT WD$(I)
130 NEXT I
140 END
```

This program has some similarities to the first-name list we did a few pages ago. The major distinction between the two occurs at line 40—instead of INPUT, the elements of data are accessed with READ from the DATA statements. The program concludes with an option of viewing the data.

Before we go on to multi-dimensional arrays, give the problems on the next page a try, just to verify your knowledge of one-dimensional arrays. Good luck—see you in the next section!

Exercises

1) Write a program which reads six names and puts this data into an array. Furthermore, write it so that the names are output in alphabetical order. Test out this program with the names Russ, Arnold, Debbie, Jan, Frank and Jim. Consider that the names must be compared as strings to see if they are less than, equal to or more than one another. Consequently, the name Arnold must be the first listed.

2) Write a program which generates six random numbers, and places these numbers into an array. Plus, the largest of these numbers should be output. The random numbers should be between 50 and 150.

3) Start with the following array `X(6)`:

```
X(1)    X(2)  X(3)  X(4)  X(5)  X(6)
 0       2     6     12    20    30
```

Write a program that produces this array on its own (you program it to arrange things accordingly), and outputs it for your comparison. Do not put these numbers directly into the elements! (don't cheat). Basically, this array serves no purpose other than to test your abilities.

4.1.3 Multi-dimensional arrays

In the previous chapter we worked only with one-dimensional arrays—a list of individual data elements. Now we're working with individual elements surrounded by other elements, or a combination of data. This combination is be divided into rows and columns, not unlike a spreadsheet (such as ***Power Plan***). To illustrate how these combined elements work, we're going to write a program that contains a first name, last name and birthday in the same element.

"Easy," you say, "we make three separate arrays—`A$(X)` for first names, `B$(X)` for last names and `C$(X)` for birthdays." Well, that's fine—but now you have names and birthdays broken into three different arrays, and the system is going to have one heck of a time sorting all that out. But you figured it out as best you could with the only arrays you know about—one-dimensional arrays.

The answer to our problem is the *multi-dimensional array*. We need a two-dimensional field, consisting of the number of total groups of data, and the number elements per group. For the sake of graphic illustration, here is what this array would look like drawn out:

	Column 1	Column 2	Column 3
Row 1			
Row 2			
Row 3			
Row 4			
Row 5			

The `DIM` statement for this array would be `DIM A$(5,3)`, giving us 5 rows and 3 columns. (Actually we have 6 rows and 4 columns—we're leaving out 0 for now). If `DIM` wasn't executed, and one of the elements of this array was used, the computer would automatically create a two-dimensional array of `(10,10)`. Thus, you would use the `DIM` statement for dimensioning any array smaller or larger than `(10,10)`.

How do we use such an array? For demonstration purposes, let's fill the first three columns of row 1 with data. We can use this program line:

```
.
.
40 INPUT"FIRST NAME, LAST NAME, BORN";A$(1,1),A$(1,2),A$(1,3)
.
```

From this we get our three elements, separated by commas. If you prefer, we can put these elements into three separate program lines per entry:

```
40 INPUT"FIRST NAME";A$(1,1)
50 INPUT"LAST NAME";A$(1,2)
60 INPUT"BORN";A$(1,3)
```

This arrangement leaves more margin for error on individual entries.

You might be wondering why this input hasn't been put into a loop, into which names and birthdays are combined into one `INPUT` statement. Our example uses every `INPUT` statement as its own commentary on the value to which it writes. We can use a `FOR...NEXT` loop in this manner:

```
10 REM READ IN FIRST NAMES, LAST NAMES
20 REM  AND BIRTHDAYS
30 DIM A$(6,3)
40 FOR I=1 TO 6
50 INPUT"FIRST NAME";A$(I,1)
60 INPUT"LAST NAME ";A$(I,2)
70 INPUT"BIRTHDATE ";A$(I,3)
80 NEXT I:END
```

You can find similar routines in small BASIC databases.

Note that a multi-dimensional array isn't limited to keyboard input—you can also use this sort of array with `READ` and `DATA`. You've already used this technique with one-dimensional arrays. Here's a two-dimensional array `(3,4)`:

```
10 REM LOAD ARRAY WITH DATA LINES
20 DIM X(3,4)
30 FOR R=1 TO 3
40 FOR C=1 TO 4
50 READ X(R,C)
```

```
60 NEXT C,R
70 DATA 11,12,13,14,21,22,23,24,31,32,33,34
80 REM ARRAY OUTPUT
90 PRINT "DISPLAY ARRAY (Y/N)?"
100 GETKEY A$
110 IF A$ < > "Y" THEN 150
120 FOR R=1 TO 3
130 PRINT X(R,1);X(R,2);X(R,3);X(R,4)
140 NEXT R
150 END
```

Here we have an example of two nested FOR...NEXT loops, helping to fill in an array of 3 rows and 4 columns. The inner loop works to fill in all the elements of a row with data. Once this loop is done, the external loop runs, until three rows are done. The illustration below will help clear up confusion.

```
                        ARRAY X(3.4)

11  *  *  *    11 12  *  *    11 12 13  *    11 12 13 14
 *  *  *  *     *  *  *  *     *  *  *  *     *  *  *  *
 *  *  *  *     *  *  *  *     *  *  *  *     *  *  *  *

11 12 13 14    11 12 13 14    11 12 13 14    11 12 13 14
21  *  *  *    21 22  *  *    21 22 23  *    21 22 23 24
 *  *  *  *     *  *  *  *     *  *  *  *     *  *  *  *

11 12 13 14    11 12 13 14    11 12 13 14    11 12 13 14
21 22 23 24    21 22 23 24    21 22 23 24    21 22 23 24
31  *  *  *    31 32  *  *    31 32 33  *    31 32 33 34
```

Press Y for a visible output—you'll see what the illustration above shows. Line 130 draws this display; there are four columns of three lines, thanks to the FOR...NEXT loop. There is another way of getting the display:

```
  .
  .
80 REM ARRAY OUTPUT
90 PRINT"DISPLAY ARRAY (Y/N)?"
100 GETKEY A$
110 IF A$ < > "Y" THEN 150
120 FOR R=1 TO 3
130 FOR C=1 TO 4
```

```
140 PRINT A(R,S);:ZZ=ZZ+1
150 IF ZZ=4 THEN ZZ=0:PRINT:PRINT
160 NEXT C,R
170 END
```

Now you have two nested loops executing in the second half of the program as well. Note the semicolon in line 140. This sets the elements next to one another, until four elements are lined up according to counter ZZ (see IF...THEN in line 150). When ZZ reaches 4, it is reset to 0, then two PRINT statements space down, and the next element is printed at the start of the next line.

Now for a small tip: You may want to slow down the output with a time loop. Add this command to pause for a second:

```
155 SLEEP 1
```

We're going to spend just a little more time with the name/birthday program, and treat it as if it were a "real" database. Our array provides for only 6 entries —this, as you know, is not the case in most databases. We aren't concerned with the person, but with the amount of personal information, i.e., last name, first name and telephone number. The first number of the DIM statement designates the number of people—the second number designates the amount of information per person: DIM X$(120,3).

```
10 REM READ IN DATA
20 DIM X$(120,3)
30 SCNCLR
40 Z=Z+1
50 INPUT"FIRST NAME";X$(Z,1)
60 PRINT
70 INPUT"LAST NAME ";X$(Z,2)
80 PRINT
90 INPUT"PHONE NUMBER";X$(Z,3)
100 PRINT
110 PRINT"DO YOU WANT TO"
120 PRINT"INPUT MORE DATA (Y/N)?"
130 GETKEY A$
140 IF A$="Y" THEN 30
150 END
```

This problem can be solved a bit more elegantly. But here we're just concentrating on basic principles. Since we aren't using any FOR...NEXT loops here, we've inserted the counter at line 40, which increment after each input. Next follows the INPUT statements for data entry, with their respective array elements. If more input is required, the program branches back to line 30. Otherwise the program terminates. You could extend this little database to include a main menu, in which the user chooses his/her own program areas.

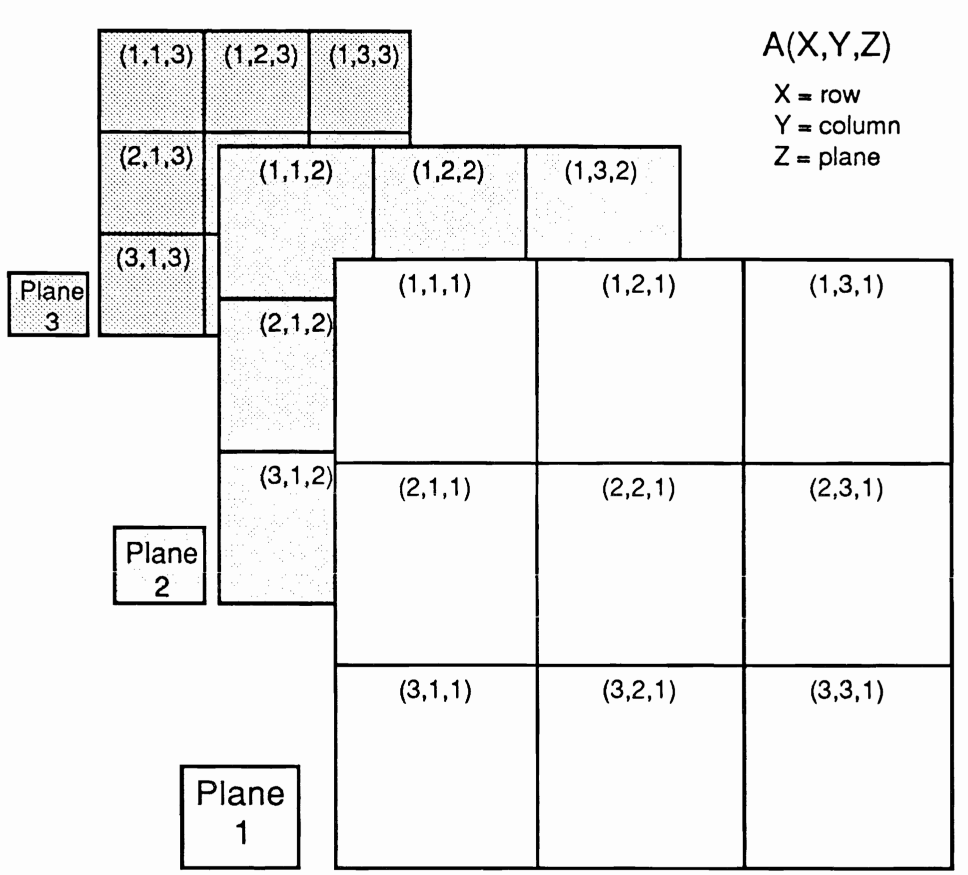
The IF...THEN comparison (line 140) tests for the need for more data entry. This is quite a change from the first program, which used a FOR...NEXT loop. Reminder: use IF...THEN when the amount of data to be given is unknown.

These examples should help you comprehend the nature of multi-dimensional arrays. Theoretically the Commodore 128 can handle arrays with up to 255 indices. This means that it not only produces two-, three- or even four-dimensional arrays, but arrays with up to 255 dimensions. The key word is theoretically —you need a huge amount of memory to handle that many "pigeon-holes". Three-dimensional arrays are relatively easy to plot out, but four-dimensional and five-dimensional arrays are a big problem to design.

Here's a sample three-dimensional array. The indices are:

X = Row
Y = Column
Z = Depth

Now we have a 3-D array, which looks like a cube. We'll give it a side length of 3.



Let's put the array into the computer using `DIM`:

```
DIM W(3,3,3)
```

This gives us a total of 27 elements (3*3*3=27) (actually 4*4*4=64 elements, if you count zeroes).

Now comes the fun part—we have to write a program that fills this array with data. We know how much data to use—now we have to figure out how the rows are set up, and how to "shift" the arrays around. Write the program so that the uppermost arrays are produced first. Don't worry about spatial effects on output.

Solution

The program would look something like this:

```
10 REM 3-D ARRAY
20 DIM W(3,3,3)
30 FOR Z=1 TO 3
40 FOR Y=1 TO 3
50 FOR X=1 TO 3
60 READ W(X,Y,Z)
70 NEXT X,Y,Z
80 DATA 1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,
   3,3,3,3,3,3,3,3,3
90 REM ARRAY OUTPUT
100 FOR Z=1 TO 3
110 FOR Y=1 TO 3
120 FOR X=1 TO 3
130 PRINT W(X,Y,Z);:ZZ=ZZ+1
140 IF ZZ=3 THEN ZZ=0:PRINT
150 NEXT X,Y,Z
160 END
```

Don't forget the `DIM` statement (line 20)—what applies to two-dimensional arrays counts even more for three-dimensional arrays. If you leave out the `DIM` statement, the computer creates a `10*10*10` (or, counting zeroes, `11*11*11`) array! All we want is 3*3*3, or 27 elements (which take up quite a bit of memory in and of themselves). Lines 30-50 are three parallel `FOR...NEXT` loops that figure out the design of the cube. Line 70 holds the `NEXT` for all three loops here. Note the next `FOR...NEXT` loop set, which is lined up in the same sequence as the first set. The array output could be literal, as in the illustration a few pages back, but we've simply lined up the three "sides" to make sure all is well numerically.

As you become more adept at programming, you'll find that one- and two-dimensional arrays will be more than sufficient for your purposes.

4.2 Subroutines

What is a subroutine? You already know about programs being divided into sections. A *subroutine* is a section of a program that is called up time and time again. It is an independent section, usually put after the END or at the beginning of the main program. The syntax:

GOSUB (*line number*)

GOSUB is short for "GOto SUBroutine". The line number identifies the beginning of the subroutine. When the program encounters this command, it looks for the line number, and continues the program at the subroutine. Once the subroutine has completed its task, it sends the execution back to the main program with the statement:

RETURN

On finding a RETURN without a GOSUB, the system responds with the message:

?RETURN WITHOUT GOSUB ERROR IN (*line number*)

There are occasions when the programmer uses GOTO commands instead of GOSUB, and errors abound. For example, this program gives the programmer just what he/she deserves...

```
.
.
110 IF A < 1 THEN GOTO 130:REM  ERROR!!!!!!
120 GOTO 90
130 REM SUBROUTINE??
140 A=A+1
150 RETURN
```

This example has line 110 ordering up a GOTO a subroutine when A is less than 1. This incorrect usage of GOTO results in a ?RETURN WITHOUT GOSUB ERROR IN 150. But if 110 would read:

```
110 IF A < 1 THEN GOSUB 130:REM  RIGHT!
```

then there would be no problem. Another error, which often occurs in the development of larger programs, occurs because of improper use of subroutines:

```
10 REM ERROR IN SUBROUTINE
20 PRINT
30 PRINT CHR$(18);Z
40 GOSUB 70
50 Z=Z+1:GOTO 20
60 END
70 REM SUBROUTINE
80 FOR I=1 TO 25
90 PRINTI;
100 IF I>= 15 THEN GOTO 50
110 NEXT I
120 RETURN
```

Type in this program and start it. After 21 or 22 runs of this subroutine, the error message ?OUT OF MEMORY ERROR IN 100 appears. This error is due to the GOTO out of the subroutine rather than the subroutine itself. The program prints a blank line at the start of the run (line 20). Then, the variable Z (the counter for the number of times the subroutine runs) is printed in reverse video (CHR$(18)). Line 40 calls the subroutine, which begins at line 70, and starts executing the FOR...NEXT loop. The variable I is displayed.

Line 100 breaks two cardinal rules: First, you should never leave a FOR...NEXT loop with a GOTO. Second (and this is really important), never, never leave a subroutine in a manner other than RETURN. Instead of just a THEN GOTO at line 100 when I reaches 15, it should read THEN RETURN.

You can pass in and out of a subroutine with GOTO—the issue we're bringing up here is that you shouldn't just LEAVE using GOTO. Larger programs with errors like these can be tough to debug, since one moment the program runs fine, and next moment an error message turns up running through the same program lines.

Now we come to practical usage of a subroutine. Remember our Math Tutor program? We're going to revise this somewhat, so that the tedious tasks are simply pulled from subroutines. This means redesigning the program:

```
230 SCNCLR
240 PRINT TAB(10)" INPUT THE LARGEST NUMBER
250 PRINT
260 PRINT TAB(10)" TO BE ADDED."
270 PRINT
290 PRINT TAB(10);:INPUT"LARGEST";GR
299 REM
300 REM RANDOM NUMBERS
301 REM
310 A1=INT(GR*RND(1))+1
320 A2=INT(GR*RND(1))+1
329 REM
330 REM COMPUTE RESULT
331 REM
340 RE=A1+A2
350 SCNCLR
360 PRINT
370 PRINT"HOW MUCH IS" A1 "+" A2 "= ";
380 INPUT ES
390 IF ES=RE THEN PRINT:PRINT TAB(10)
    "RIGHT!":F=0:GOTO 450
400 PRINT:PRINT TAB(10)"WRONG !"
410 FOR I=0 TO 500:NEXT I
420 F=F+1
430 IF F<=2 THEN 350
440 PRINT
450 FOR I=0 TO 500:NEXT I
460 PRINT TAB(5)"THE ANSWER IS" RE
470 FOR I=0 TO 500:NEXT I
480 PRINT TAB(10)"ANOTHER PROBLEM Y/N";
490 INPUT A$
500 IF A$="Y" THEN F=0:GOTO 300
510 GOTO 10
```

The most-used program lines are typed in over and over. Most of them are the "INPUT THE LARGEST..." statements. So, to increase our flexibility, we'll rewrite the program to include a number of subroutines.

The addition, subtraction, etc. sections are controlled using menu options and an ON X GOTO. The beginning of the program contains the arrays which hold the names and operators of these options (line 90).

```
10 REM ******************
20 REM * START OF PROGRAM*
30 REM ******************
40 COLOR 0,1:COLOR 4,1:COLOR 5,6
50 DIM RA$(4),BE$(4)
60 FOR I=1 TO 4
70 READ RA$(I),BE$(I)
80 NEXT I
90 DATA ADDITION,+,SUBTRACTION,-,DIVISION,/,
    MULTIPLICATION,*
100 GOTO 580
```

Line 40 has a COLOR command, which turns the entire screen to black, and the text green (we'll discuss COLOR later). Two arrays, RA$ and BE$, are generated at line 50. Line 60 loads the arrays and puts addition, subtraction, etc. into RA$. BE$ holds the character corresponding to the type of calculation. Line 100 signals a jump to line 580, where the menu routine begins.

```
570 REM ******************
580 REM *    MENU        *
590 REM ******************
600 SCNCLR:F=0
610 PRINT
620 PRINT TAB(12)"MATH TUTOR"
630 PRINT:PRINT
640 PRINT TAB(12)"CHOOSE:"
650 PRINT
660 PRINT TAB(12)"1 FOR "RA$(1)
670 PRINT
680 PRINT TAB(12)"2 FOR "RA$(2)
690 PRINT
700 PRINT TAB(12)"3 FOR "RA$(3)
710 PRINT
720 PRINT TAB(12)"4 FOR "RA$(4)
730 PRINT
740 PRINT TAB(12)"5 TO END"
750 PRINT
760 PRINT TAB(12)"WHICH NUMBER ?"
770 GETKEY E$
780 IF VAL(E$) < 1 OR VAL(E$) > 5 THEN 770
790 P=VAL(E$):ON P GOTO 800,890,990,1090,1180
```

The "new" version of the program has the expression names (addition, subtraction, etc.) stored in RA$, from which these names are called as needed. The routine could even be written like this:

```
660 FOR I=1 TO 4
670 PRINT TAB(12)I" FOR "RA$(I)
680 PRINT
690 NEXT I
```

Line 770 contains a GETKEY which checks whether the input is within the allowed number range (1-5). If so, VAL(E$) changes the string to a numerical value, and places this number into the variable P, which branches to the appropriate lines. When addition is chosen, P=1, and goes to line 800.

```
800 REM **************
810 REM * ADDITION    *
820 REM **************
830 GOSUB 110
840 GOSUB 310
850 RE=A1+A2
860 GOSUB 390
870 IF A$="Y" THEN 840
880 GOTO 580
.
.
```

Line 830 calls the first subroutine. That's the "highest number" section:

```
110 REM **************
120 REM * SUBROUTINE *
130 REM **************
140 REM ************************
150 REM * INPUT LARGEST NUMBER *
160 REM ************************
170 SCNCLR:A$="":B$=""
180 PRINT TAB(10)" INPUT THE LARGEST NUMBER FOR
190 PRINT
200 PRINT TAB(10) RA$(P)"."
210 PRINT
220 PRINT TAB(10)"LARGEST?";
230 FOR I=1 TO 3
240 GETKEY A$
```

```
250 IF ASC(A$) < 48 OR ASC (A$) > 57 THEN 240
260 B$=B$+A$:PRINT A$;
270 NEXT I
280 GR=VAL(B$)
290 RETURN
  .
  .
```

The first part of line 170 is obvious. But why fill A$ and B$ with null strings? In line 260, A$ and B$ are combined to form B$. In this case—the routine is going to be recalled time and again—the variables are wiped out at the start of the routine for fresh input. If we didn't do this, the old variables would simply be added to the new, and figures would come up wrong.

Numerical input by means of GET has already been discussed (see Chapter 3.4.1). Our example requires a few changes—we've limited the input to the number keys (line 250). Line 260 lets you see your input (PRINT A$;). If you wish to use two-digit numbers, you must first enter a 0, then the rest of the number.

Line 200 is of particular interest. Variable P is used as an index to call up the terms in RA$. RA$(1) is "addition." RA$(2) is "subtraction," and so on.

The next program line in addition (line 840) calls the subroutine for generating random numbers. This is the shortest and simplest subroutine.

```
300 REM *************************
310 REM *  CREATE RANDOM NUMBERS *
320 REM *************************
330 A1=INT(GR*RND(1))+1
340 A2=INT(GR*RND(1))+1
350 RETURN
```

The next subroutine sets up the math problem:

```
360 REM *****************
370 REM * PROBLEM SET-UP *
380 REM *****************
390 SCNCLR
400 PRINT
410 PRINT"HOW MUCH IS";A1;BE$(P);A2;"= ";
420 INPUT ES
```

```
430 IF ES=RE THEN PRINT:PRINT TAB(10)
    "RIGHT!":F=0:GOTO 500
440 PRINT:PRINT TAB(10)"WRONG!!!"
450 FOR I=0 TO 500:NEXT I
460 F=F+1
470 IF F<=2 THEN 390
480 PRINT
490 FOR I=0 TO 500:NEXT I
500 PRINT:PRINT TAB(5)"THE ANSWER IS "RE
510 F=0
520 FOR I=0 TO 500:NEXT I
530 PRINT
540 PRINT TAB(5)"ANOTHER PROBLEM Y/N";
550 INPUT A$
560 RETURN
```

Line 410 is of particular interest—we've managed to create one line that designs all four types of problems. `HOW MUCH IS` opens, and the rest of the line gets its input from `A1`, `BE$(P)` and `A2`. `A1` and `A2` supply our numbers. `BE$(P)` determines the type of calculation performed (based upon the user's choice). The subroutines for calculation also contain the "factor correction" to avoid any strange equations.

You'll recognize the lines from 540 on from the "original" version of `Math Tutor`. Line 550 takes the answer to the question in line 540.

Note that we've placed the subroutines at the beginning of the program. Most books tell you the opposite—to put the subroutines at the end. There's a reason for this. Many programmers keep a library of standard subroutines, equipped with high line numbers to ease their insertion into a program. These routines can be put into a program by a `MERGE` command (available in some versions of BASIC), or by adjusting the memory in the system and loading in the new routine.

But what happens if you put the subroutines at the start of the program? Nothing new—the subroutines work whether the `MERGE` puts them in front or in back.

Actually, putting the subroutines at the start is a little faster. When a subroutine is called, the computer goes to the first program line and searches the entire program until the subroutine is found. The less time the computer spends searching, the faster the execution time. You won't notice this in small programs, but it will be obvious in larger programs.

A1 and A2 are monitored for subtraction problems to be sure that A1 is greater than A2. If this weren't the case, we would end up with negative answers. Division problems are checked to ensure whole numbers as answers. The program multiplies A1 and A2 and moves variables around for division:

```
1040 RE=A1*A2
1050 I=RE:RE=A1:A1=I
```

Here is the complete program:

```
5   REM TUTOR 4.2
10 REM*******************
20 REM* PROGRAM START    *
30 REM*******************
40 COLOR 0,1:COLOR 4,1:COLOR 5,6
50 DIM RA$(4),BE$(4)
60 FORI=1TO4
70 READ RA$(I),BE$(I)
80 NEXTI
90 DATA ADDITION,+,SUBTRACTION,-,DIVISION,/,
   MULTIPLICATION,*
100 GOTO580
110 REM********************
120 REM* SUBROUTINES       *
130 REM********************
140 REM********************
150 REM* INPUT HIGHEST NO.*
160 REM********************
170 SCNCLR:A$="":B$=""
180 PRINTTAB(10)"INPUT THE HIGHEST NUMBER"
190 PRINT
200 PRINTTAB(10)"FOR THE "RA$(P)"."
210 PRINT
220 PRINTTAB(10)"HIGHEST?"
230 FORI=1 TO 3
240 GETKEY A$
250 IF ASC(A$)<48 OR ASC(A$)>57 THEN 240
260 B$=B$+A$:PRINTA$;
270 NEXTI
280 GR=VAL(B$)
290 RETURN
```

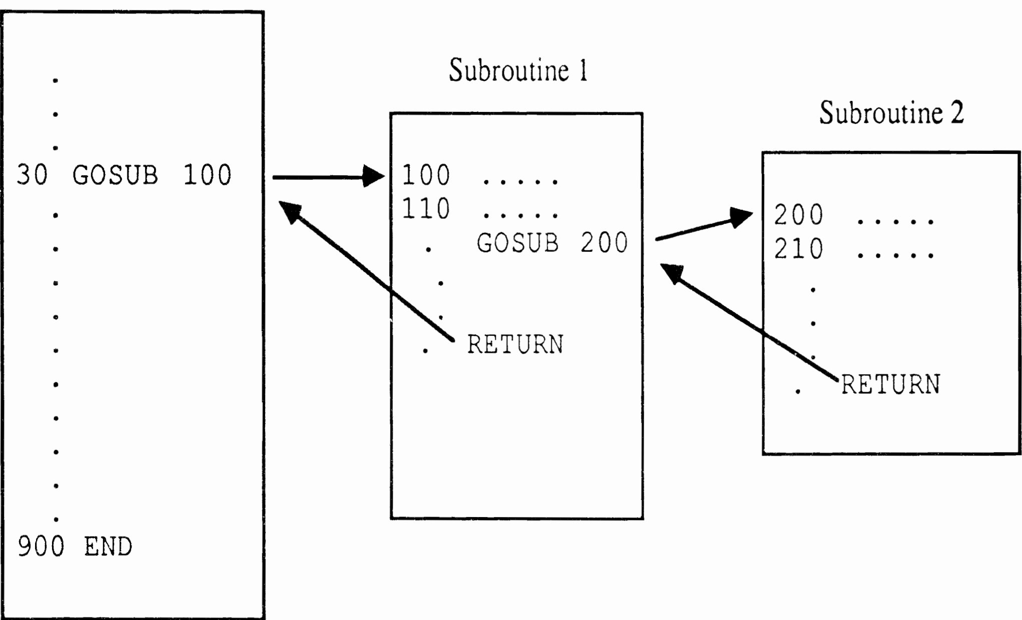
```
300 REM********************
310 REM* RND NUMBERS       *
320 REM********************
330 A1=INT(GR*RND(1))+1
340 A2=INT(GR*RND(1))+1
350 RETURN
360 REM********************
370 REM*  PROBLEM  SETUP   *
380 REM********************
390 SCNCLR
400 PRINT
410 PRINT"HOW MUCH IS";A1;BE$(P);A2;"=";
420 INPUT ES
430 IF ES=RE THEN PRINT:PRINTTAB(10)
    "RIGHT!":F=0:GOTO500
440 PRINT:PRINTTAB(10)"WRONG."
450 FOR I=0 TO 500:NEXTI
460 F=F+1
470 IF F <=2 THEN 390
480 PRINT
490 FOR I=0 TO 500:NEXTI
500 PRINT:PRINTTAB(10)"THE ANSWER IS";RE
510 F=0
520 FORI=0 TO 500:NEXTI
530 PRINT
540 PRINTTAB(5)"ANOTHER PROBLEM (Y/N)?";
550 INPUT A$
560 RETURN
570 REM*****************
580 REM*    MENU       *
590 REM*****************
600 SCNCLR:F=0
610 PRINT
620 PRINTTAB(12)"MATH TUTOR"
630 PRINT:PRINT
640 PRINTTAB(12)"CHOOSE A NUMBER:"
650 PRINT
660 PRINTTAB(12)"1)  "RA$(1)
670 PRINT
680 PRINTTAB(12)"2)  "RA$(2)
690 PRINT
700 PRINTTAB(12)"3)  "RA$(3)
710 PRINT
```

```
720 PRINTTAB(12)"4)  "RA$(4)
730 PRINT
740 PRINTTAB(12)"5)  TO END"
750 PRINT
760 PRINTTAB(12)"WHICH NUMBER?"
770 GETKEY E$
780 IF VAL(E$)<1 OR VAL(E$)>5 THEN 770
790 P=VAL(E$):ON P GOTO800,890,990,1090,1180
800 REM***************
810 REM* ADDITION     *
820 REM***************
830 GOSUB110
840 GOSUB310
850 RE=A1+A2
860 GOSUB390
870 IFA$="Y"THEN840
880 GOTO580
890 REM***************
900 REM* SUBTRACTION  *
910 REM***************
920 GOSUB110
930 GOSUB310
940 IFA1<A2THENI=A1:A1=A2:A2=I
950 RE=A1-A2
960 GOSUB390
970 IFA$="Y"THEN930
980 GOTO580
990 REM****************
1000 REM* DIVISION     *
1010 REM****************
1020 GOSUB110
1030 GOSUB310
1040 RE=A1*A2
1050 I=RE:RE=A1:A1=I
1060 GOSUB390
1070 IFA$="Y"THEN1030
1080 GOTO580
1090 REM****************
1100 REM* MULTIPLICATION*
1110 REM****************
1120 GOSUB110
1130 GOSUB310
1140 RE=A1*A2
```

```
1150 GOSUB390
1160 IFA$="Y"THEN1130
1170 GOTO580
1180 REM*********
1190 REM*  END   *
1200 REM*********
1210 SCNCLR
1220 END
```

By using three subroutines, we've saved ourselves 43 program lines and a larger number of REMs! Line 100 sends the program directly to the menu.

It's also possible to call a subroutine from a subroutine, in addition to a subroutine from the main program. Graphically it looks like this:



The program encounters the GOSUB command in line 30, which calls Subroutine #1. Subroutine #2 is called within Subroutine 1; once #2 is done, the RETURN sends the program back to Subroutine #1, which runs until it encounters the RETURN command. RETURN sends the system back to the main program.

Note: Subroutines can be "nested" like FOR...NEXT loops.

We have already seen the ON X GOTO statement. GOSUB can also function like this:

```
ON P GOSUB 800,890,990
```

The program branches conditionally to the subroutine that correlates to the value of P.

You probably have a working knowledge of subroutine technique by now. Now let's put this knowledge to the test. There is another improvement that can be made to Math Tutor with subroutines. Your job is to alter the program in the present form. You don't need to rewrite the program, just make some changes. Do not put the subroutine at the beginning of the program, or too much rewrite time would be involved.

Solution

```
.
.
780 IF VAL(E$) < 1 OR VAL(E$) > 5 THEN 770
790 P=VAL(E$)
800 IF P=5 THEN 1100
810 GOSUB 110
820 GOSUB 310
830 ON P GOSUB 880,930,990,1050
840 GOSUB 390
850 IF A$="Y"THEN 820
860 GOTO 580
870 REM ************
880 REM * ADDITION *
890 REM ************
900 RE=A1+A2
910 RETURN
920 REM ***************
930 REM * SUBTRACTION *
940 REM ***************
950 IF A1<A2 THEN I=A1:A1=A2:A2=I
960 RE=A1-A2
970 RETURN
980 REM ************
990 REM * DIVISION *
```

```
1000 REM ************
1010 RE=A1*A2
1020 I=RE:RE=A1:A1=I
1030 RETURN
1040 REM ******************
1050 REM * MULTIPLICATION *
1060 REM ******************
1070 RE=A1*A2
1080 RETURN
1090 REM *******
1100 REM * END *
1110 REM *******
1120 SCNCLR
1130 END
```

The solution can be found in five lines of code. The following lines are from the most recent revision:

```
810 GOSUB 110
820 GOSUB 310
840 GOSUB 390
850 IF A$="Y" THEN 840
860 GOTO 580
```

For every basic math expression, you need these five program lines. Unless, of course, you use `ON P GOSUB` (line 830), as we have shown in our "final revision." By using this command, we've saved ourselves nine more program lines.

Here are the most essential items to remember about subroutines:

1) Subroutines are repeated program routines.

2) Subroutines are called by `GOSUB` and ended with `RETURN`. You may not enter or leave a subroutine using `GOTO`. However, it is alright to use `GOTO` to move about within a subroutine.

3) Subroutines can be nested like loops: Subroutine 1 can have a call to Subroutine 2, and so on.

4) Subroutines can have conditional calls with `ON X GOSUB`.

4.3 Menu techniques

Say you've advanced in BASIC to the point where you want to write a large program to simplify your work, or perhaps write a program to keep track of sales. An important term to consider is *user-friendly*. What does this mean?

A program should be workable by you or any other user. Anyone should be able to know which key to press and when, and which operation does what, without reams of documentation required for explanation. In other words, a user-friendly program is one that explains itself, and can be run without running to a manual for every little detail of program operation.

With this in mind, let's talk about *menus*. We've already seen a menu in the `Math Tutor` program. It essentially lines up the program points individually, and allows the user to choose one of the options by pressing a specific key or set of keys. Menus should be clearly understandable, but not to the point of treating the user like an idiot. The old saying "What you see is what you get" applies to menus.

To let you see just how a menu is designed, we're going to write a concrete example step by step. Our sample will be built on a mathematical table. First we have to figure out what to have our program do—let's give it the following functions:

Square root
Sine
Cosine
Natural logarithm
Base 10 logarithm

Wait, we've left something out—we still need an "exit" option. Always provide a convenient way of ending a program—the user shouldn't have to resort to pressing the <RUN/STOP> key or shutting the computer off.

So we actually have six menu options. Next we need some form of prompt that specifies how the user makes his/her choice:

```
PRESS THE NUMBER OF YOUR CHOICE (1-6)
```

That is the basic outline of a menu. Now let's dress it up a bit. We'll turn the screen black, draw a border, and put a title in a subroutine that keeps it onscreen throughout the execution of the program.

The last line should be reserved for input—here we use the `INPUT` command. Now for the program listing:

```
5 REM MENU DEMO 4.3
10 REM*******************
20 REM* PROGRAM START    *
30 REM*******************
40 COLOR 0,1:COLOR 4,1
50 SCNCLR
60 DIM M$(6)
70 FOR I=1 TO 6
80 READ M$(I):NEXT I
90 DATA " 1 SQUARE ROOT"
100 DATA " 2 SINE"
110 DATA " 3 COSINE"
120 DATA " 4 NATURAL LOGARITHM"
130 DATA " 5 DECIMAL LOGARITHM"
140 DATA " 6 END PROGRAM"
150 GOTO 330
160 REM*******************
170 REM* SUBROUTINE      *
180 REM*******************
190 REM
200 REM*******************
210 REM*HEADER           *
220 REM*******************
230 SCNCLR
240 FOR I=1 TO 40:PRINT"*";:NEXT I
250 PRINT"*[38 spaces]*";
260 PRINT"*              MATH   TABLE             *";
270 PRINT"*[38 spaces]*";
280 FOR I=1 TO 40:PRINT"*";:NEXTI
290 RETURN
300 REM*************
310 REM*  MENU      *
320 REM*************
330 GOSUB 230
340 FOR I=1 TO 18
350 PRINT"*[38 spaces]*";
360 NEXT I
370 FOR Q=1 TO 40:PRINT"*";:NEXT Q
380 PRINTCHR$(19)CHR$(19);
390 FOR R=1 TO 6:PRINT:NEXT R
```

```
400 FOR I=1 TO 6
410 PRINTCHR$(29);M$(I)
420 NEXT I
430 PRINT
440 PRINT CHR$(29);
450 PRINT"INPUT YOUR CHOICE (1-6)";
460 INPUT W$
```

Let's backtrack and go over the salient points of this program. Lines 40-50 turn the screen black and clear the screen. M$(6) is generated, and the DATA from lines 90-140 is read in (lines 60-80). Then the program jumps to the routine at 330, and from there to the subroutine at line 230. Here the header is drawn, asterisks appear onscreen (lines 240-250), and a PRINT command prints the headline.

The subroutine is completed, sending the program along at line 340. A new loop (340-360) designs the menu border, while line 370 draws the bottom menu line, and 380 moves the cursor to the HOME position. Line 390 prints three blank lines, to avoid printing the menu options in the headline. Lines 400-420 display the contents of M$. Line 410 moves the cursor one position to the right, so that the menu points don't overwrite the left border. Line 450 contains a PRINT statement and a semicolon—the latter joins the INPUT statement (line 460) to 450.

We won't waste space telling you how to branch to the different options, since we have already covered that in Math Tutor. You can use INPUT to take in the options, or even GET or GETKEY.

One further note. If you wish to continually display the headline during all of the math functions, open each calculation subroutine with GOSUB 230.

4.3.1 Using GET routines in menus

In an earlier chapter we discussed the main disadvantage of GET—you don't see what was chosen. The new version of Math Tutor presented a way to view that input. It had a PRINT command after the GET to show the value in A$.

This GET input routine is still pretty primitive, though. If you had to input three numbers or characters, the program would have to be prepared for it ahead of time, without a margin for error. You would be forced to input three digits—54 would be input as 054; and what if you wanted numbers higher than 999? Why don't we just go back to the INPUT statement?

Most of the time, you'll find yourself doing just that—turning to the INPUT statement. If you want to protect your program from any "skipping" on the part of the user, though, GET is your best bet.

Presently, we're going to develop our own GET routine, suitable for merging into your own programs as a subroutine. You have the option of using either GET or GETKEY in this routine—a comparison of the first line in both versions should tell you how to tailor the program for GETKEY:

```
10 GET A$:IF A$=""THEN 10
```

or

```
10 GETKEY A$
```

Every character entered is arranged into A$. For our purposes, we only want numerical input, so line 20 checks to make sure that number keys alone will register, with the help of an IF...THEN statement:

```
10 GET A$:IF A$="" THEN 10
20 IF ASC(A$) < 48 OR ASC(A$) > 57 THEN 10
```

ASCII codes 48 to 57 are the numbers 0 to 9. Any ASCII characters out of that range will be ignored. Next, we have to set the routine to count the number of digits input, so we add a counter (Z) to watch for four digits per number. You must change the number in line 40 if a larger number is desired.

```
10 GET A$:IF A$="" THEN 10
20 IF ASC(A$) < 48 OR ASC(A$) > 57 THEN 10
30 Z=Z+1
40 IF Z > 4 THEN 10
```

Z increments itself only if allowable numbers (rather than characters) are input. Once Z reaches 4, it will no other input will be accepted, and the program jumps back to line 10.

The system must now be signalled that we are through with input. Like the INPUT statement, we will use the <RETURN> key. <RETURN> has an ASCII value of 13, so we'll need to include that in out routine. Wait a minute, that won't work—13 is less than 48, so we can't include this after line 20, or the <RETURN> key won't work. Thus, we add a line 15 (before 20, so it will be acknowledged first) with this information.

How will this routine behave when <RETURN> is pressed? Too early to say, since we haven't written line 100 yet...

```
10 GET A$:IF A$="" THEN 10
15 IF ASC(A$) = 13 THEN 100
20 IF ASC(A$) < 48 OR ASC(A$) > 57 THEN 10
30 Z=Z+1
40 IF Z > 4 THEN 10
```

What we need now is a line to convert the characters input to a complete string variable. This is handled by line 50, after which line 60 prints the string onscreen:

```
10 GET A$:IF A$="" THEN 10
15 IF ASC(A$) = 13 THEN 100
20 IF ASC(A$) < 48 OR ASC(A$) > 57 THEN 10
30 Z=Z+1
40 IF Z > 4 THEN 10
50 B$=B$+A$
60 PRINT A$;
```

The semicolon in line 60 keeps the current cursor position after the string, i.e. the cursor stays on the same line as the PRINT command. Once the characters have been displayed, the routine returns to line 10 through a GOTO 10.

```
10 GET A$:IF A$="" THEN 10
15 IF ASC(A$) = 13 THEN 100
20 IF ASC(A$) < 48 OR ASC(A$) > 57 THEN 10
30 Z=Z+1
40 IF Z > 4 THEN 10
50 B$=B$+A$
60 PRINT A$;
70 GOTO 10
```

Now, we convert the string variable B$ to a numerical value, and treat it as a number. This occurs after pressing the <RETURN> key. The counter Z is cleared and reset to zero; if it weren't zeroed out, the old value would still be there for the next call of the routine, and four-digit numbers would be out of the question.

Should you wish to use this routine as a subroutine, remember to place a RETURN statement in the last line. Rather than keep you in suspense any longer, here is the entire routine:

```
10 GET A$:IF A$="" THEN 10
15 IF ASC(A$) = 13 THEN 100
20 IF ASC(A$) < 48 OR ASC(A$) > 57 THEN 10
30 Z=Z+1
40 IF Z > 4 THEN 10
50 B$=B$+A$
60 PRINT A$;
70 GOTO 10
100 B=VAL(B$):Z=0
110 PRINT B
120 END:REM RETURN INSTEAD FOR SUBROUTINE
```

This routine is a lot simpler to use than the one we put into Math Tutor. We still haven't told you how to clear the variables for new input, though. This function can be appended to our new GET routine—here's a modified version:

```
10 REM GET ROUTINE W/ DELETE FUNCTION
20 GET A$:IF A$=""THEN 20
30 IF ASC(A$) = 13 THEN 130
40 IF ASC(A$) <> 20 THEN 70
50 IF LEN(B$)<1 THEN 20
60 LEN(B$,LEN(B$)-1):Z=Z-1:GOTO 110
70 IF ASC(A$) < 48 OR ASC(A$) > 57 THEN 20
```

```
80 Z=Z+1
90 IF Z>4 THEN Z=4:GOTO 20
100 B$=B$+A$
110 PRINT A$;
120 GOTO 20
130 B=VAL(B$):Z=0
140 PRINTB
150 END
```

One point of interest; line 40 checks to see whether the <INST/DEL> key (ASCII 20) has been pressed. If not, the program goes on to line 70. If <INST/DEL> has been pressed, line 50 checks for any more characters in B$. Once the characters have all been deleted, the clear function stops the program with:

```
?ILLEGAL QUANTITY ERROR IN 60
```

Also, the <INST/DEL> key won't work with an already empty line. The program branches back to line 20. Line 60 contains our delete routine:

```
B$=LEFT$(B$,LEN(B$)-1)
```

It deletes one character from the string B$. The statement LEFT$(B$,X) gives us a string with X leftover characters of B$. X is replaced in our routine by LEN. LEN(B$)-1 subtracts one character on the right of B$. All this is the string equivalent of the numerical operation A=A-1 . 1 is subtracted from A, and the new value becomes A.

In addition to the character deletion, the counter Z is diminished by 1 for each character deleted. This is to keep accuracy between the string and counter—if this decrementing didn't take place, the entire delete section would not work properly. The last command in line 60 sends the program to line 110, where the current contents of A$ are displayed. There you will find the results of pushing the <INST/DEL> key—a CHR$(20). The <INST/DEL> key directly affects strings in a program run. Type in:

```
PRINT"COMMODORE 129";CHR$(20);"8"
```

Press <RETURN>, and watch the result:

```
COMMODORE 128
```

The CHR$(20) deleted the "9", leaving the "8" in its place. Our GET routine does the same thing (line 110).

We'll close with a little "fine tuning" on our GET routine—we'll change the type of cursor available. It doesn't blink, but we know where the next character will appear. With the use of CHR$(164), we can put our own cursor onscreen, in the form of an "underline". The "real" cursor, represented by CHR$(157), is also used one place to the left. When a character is input, PRINT A$ displays the cursor and the character.

You may want to add some other characters, say letters or spaces. If so, we have to add two more IF...THENs, and exclude the ASCII values between the numbers and the characters (58-64). Here is the final routine:

```
10 REM **********************
20 REM * CURSOR POSITIONING *
30 REM **********************
40 SCNCLR
50 FOR I=1 TO 5
60 PRINT CHR$(17);CHR$(29);
70 NEXT I
80 PRINT CHR$(164);CHR$(157);
90 REM  **********************
100 REM * READ IN CHARACTER   *
110 REM **********************
120 GET A$:IF A$=""THEN 120
130 IF ASC(A$)=13 THEN 120
140 IF ASC(A$)=32 THEN 260
150 IF ASC(A$)<>20 THEN 200
160 IFLEN(B$)>=1 THENB$=LEFT$(B$,LEN(B$)-1):
     GOTO 210
170 IF ASC(A$) < 48 THEN 120
180 IF ASC(A$) > 90 THEN 120
190 IF ASC(A$) > 57 AND ASC(A$) < 65 THEN 120
200 B$=B$+A$
210 PRINTA$;CHR$(164);CHR$(157);
220 GOTO 120
230 REM ******************
240 REM * STRING OUTPUT   *
250 REM ******************
260 SCNCLR
270 FOR I=1 TO 15
280 PRINT CHR$(17);CHR$(29);
290 NEXT I
300 PRINT B$
```

Lines 40-70 clear the screen and move the cursor five rows down and five columns right. Our self-designed cursor appears (line 80), and the invisible cursor is moved over the underscore (CHR$(157)). Line 120 is the start of the routine proper. Line 140 is new programming. This allows for spaces (CHR$(32)) to be inserted in our string. Line 180 asks whether the input has gone beyond the letter Z in ASCII value (>90). Line 190 checks for the area between ASCII 57 and 65; the logical operator AND is used here. The program branches back to 120 if any characters within this range are pressed. If an OR were put here, then numbers and letters would be ignored. The program lines following require a little graphic explanation, to show you what the cursor does (the invisible cursor is represented by an asterisk "*"). When we input the letter A, line 210 does the following:

```
A*      Execution of PRINT A$;
A_*     Execution of CHR$(164);
A*      Execution of CHR$(157);
```

The remainder of this program should already look familiar to you.

All in all, we have a GET routine that behaves like an INPUT statement. Using IF...THENs, you can add other characters to fit your needs (function keys, control keys, etc.). Between this routine and the menu routine, you have enough material to make your programs very user-friendly.

4.3.2 Cursor positioning with CHAR

The CHAR command is mainly used for printing text on a graphic screen. This command makes it a simple matter to label bar charts or pie charts with text. However, this command limits you to 40 columns and 25 rows, i.e., the 40-column screen, so exact positioning is impossible with CHAR.

Some BASIC dialects recognize the LOCATE statement for text output in text mode. CHAR is its equivalent in BASIC 7.0. BASIC 7.0's LOCATE command merely positions the invisible graphic cursor. We can use CHAR to put our text anywhere onscreen. Its syntax is:

```
CHAR C,X,Y,T,I
```

C = Color source (0-3) >immaterial in text mode<
X = Column (0-39)
Y = Line (0-24)
T = Text to be printed
I = 0=normal video / 1 = reverse video

CHAR ,X,Y puts the cursor on the appropriate spot onscreen. The text is printed starting at this point. An example:

```
10 SCNCLR
20 CHAR ,19,12
30 PRINT "OUTPUT USING CHAR"
40 END
```

Be forewarned that when using CHAR for longer text output, no carriage returns follow CHAR text.

4.3.3 Cursor control with CHR$-codes

Here is another form of cursor control that will work with most dialects of BASIC. This is based on the idea that the cursor can be moved by CHR$(17),CHR$(29),CHR$(145) and CHR$(156). Our attentions will be turning toward CHR$(17) (cursor down) and CHR$(29) (cursor right). We'll turn these codes into strings: The first string, made up of CHR$(29), will move the cursor to the right 40 times, while the second (CHR$(17) as a basis) will move the cursor 25 lines down.

```
90 REM CURSOR DOWN
100 FOR I=1 TO 25
110 CU$=CU$+CHR$(17)
120 NEXT I
130 REM CURSOR RIGHT
140 FOR I=1 TO 40
150 CR$=CR$+CHR$(29)
160 NEXT I
```

These lines generate two strings from which the cursor takes its movement. Our work is only half-done, though; we have yet to talk about positioning itself. We'd like to be able to specify the cursor position onscreen. The syntax should be as follows:

```
S=ROW.COLUMN
```

Or, to use specific numbers:

```
S=10.12
```

S is the variable which arranges the "whole" number as row and the "decimal" number as column. The following lines will allow this:

```
300 REM CURSOR POSITIONING
310 PRINT CHR$(19);CHR$(19);LEFT$(CU$,S)
320 PRINT LEFT$(CR$,100*(S-INT(S))+.5);
330 RETURN
```

Line 310 should be easy to understand. Cursor position is sent to the HOME location (in the upper-left-hand corner). This gives you a reference point for positions. Next, the partial string CU$ is made up from the number S, which in our case moves the cursor down 10 lines. LEFT$(CU$,S) checks out the whole numbers.

Line 320 is a bit more complex. Let's begin with the inner parentheses. S-INT(S) separates the "decimal" numbers from the "whole" numbers. The "decimal", in our case .12, is multiplied by 100, and added to .5 . The calculation is rounded off by 100*(S-INT(S)). So with 12, our integer result will only be 11, rather than 11.99999. With an additional .5, the value is increased to 12.49999. The LEFT$(CR$,S) will take integers only. The PRINT statement sends the cursor 12 places to the right. Line 330 contains a RETURN so that the code can be called up as a subroutine.

This routine can also be used in to design menus. Let's look at this revised version of Math Table from Chapter 4.3. A new cursor positioning is utilized here, employing DATA statements. See the program changes below:

```
90 DATA " 1 SQUARE ROOT"
100 DATA " 2 SINE"
110 DATA " 3 COSINE"
120 DATA " 4 NATURAL LOGARITHM"
```

```
130 DATA "  5 DECIMAL LOGARITHM"
140 DATA "  6 END PROGRAM"
150 REM*****************
160 REM* CURSOR RIGHT   *
170 REM*****************
180 FOR I=1 TO 40
190 CR$=CR$+CHR$(29)
200 NEXTI
210 REM****************
220 REM* CURSOR DOWN   *
230 REM****************
240 FORI=1TO25
250 CU$=CU$+CHR$(17)
260 NEXTI
270 GOTO 510
280 REM****************
290 REM* SUBROUTINES   *
300 REM****************
310 REM
320 REM************
330 REM*HEADER     *
340 REM************
350 SCNCLR
360 FOR I=1 TO 40:PRINT"*";:NEXT I
370 PRINT"*[38 spaces]*";
380 PRINT"*            MATH   TABLE            *";
390 PRINT"*[38 spaces]*";
400 FOR I=1 TO 40:PRINT"*";:NEXTI
410 RETURN
420 REM*************
430 REM* CURSORS    *
440 REM*************
450 PRINTCHR$(19);LEFT$(CU$,S);
460 PRINTLEFT$(CR$,100*(S-INT(S))+.5);
470 RETURN
480 REM*******
490 REM* MENU *
500 REM*******
510 GOSUB350
520 FORI=1 TO 18
530 PRINT"*[38 spaces]*";
540 NEXTI
550 FORI=1TO40:PRINT"*";:NEXTI
```

```
560 PRINT CHR$(19);CHR$(19);
570 FORI=1 TO 3:PRINT:NEXTI
580 FORI=1 TO 6
590 S=4+I*2+.03
600 GOSUB450:REM CURSOR POSIT.
610 PRINTM$(I)
620 NEXTI
```

As already mentioned, the spaces are sent for the menu points by way of DATA. Lines 150-260 are new, which contain what we've been discussing here, as are lines 420-470. These lines from the original version of the program:

```
400 FOR I=1 TO 6
410 PRINT CHR$(29);M$(I)
420 NEXT I
```

have been replaced by these lines:

```
580 FOR I=1 TO 6
590 S=4+I*2+.03
600 GOSUB 450: REM CURSOR POS.
610 PRINT M$(I)
620 NEXT I
```

We want our menu choices to begin at row 6, column 3. Line 590 contains a .03 for our third column. Don't use .3 though, or the system will end up at the 30th column.

The line value begins with 6, and should be output as two values. This is where the floating variable I comes in. The row number is calculated by S=4+1*2=6; a second run gives us S=4+2*2=8, and so on.

Our previous work with menu techniques required us to press a number or letter key to run the corresponding program. We have yet another menu technique for you, using the cursor-up and cursor-down keys. You choose the menu option by positioning a cursor over the option and pressing <RETURN>. The option chosen is marked in reverse video. We'll need two arrays for this—the one to store the options in normal video, and the other in reverse video. Below is the new program listing:

```
5 REM MENU 2 4.3.3
10 REM********************
20 REM* PROGRAM START    *
30 REM********************
40 COLOR 0,1:COLOR 4,1
50 SCNCLR
60 DIM M$(6):REM MENU POINT
70 DIM MR$(6):REM MENU POINT (REVERSE)
80 REM ******************
90 REM *FILL ARRAYS     *
100 REM******************
110 FORI=1 TO 6
120 READ M$(I)
130 MR$(I)=CHR$(18)+M$(I)+CHR$(146)
140 NEXTI
150 DATA " 1 SQUARE ROOT"
160 DATA " 2 SINE"
170 DATA " 3 COSINE"
180 DATA " 4 NATURAL LOGARITHM"
190 DATA " 5 DECIMAL LOGARITHM"
200 DATA " 6 END PROGRAM"
210 REM******************
220 REM* CURSOR RIGHT   *
230 REM******************
240 FOR I=1 TO 40
250 CR$=CR$+CHR$(29)
260 NEXTI
270 REM*****************
280 REM* CURSOR DOWN   *
290 REM*****************
300 FORI=1TO25
310 CU$=CU$+CHR$(17)
320 NEXTI
330 GOTO 570
340 REM*****************
350 REM* SUBROUTINES   *
360 REM*****************
370 REM
380 REM*************
390 REM*HEADER     *
400 REM*************
410 SCNCLR
420 FOR I=1 TO 40:PRINT"*";:NEXT I
```

```
430 PRINT"*[38 spaces]*";
440 PRINT"*                MATH   TABLE             *";
450 PRINT"*[38 spaces]*";
460 FOR I=1 TO 40:PRINT"*";:NEXTI
470 RETURN
480 REM*************
490 REM* CURSORS    *
500 REM*************
510 PRINTCHR$(19);LEFT$(CU$,S);
520 PRINTLEFT$(CR$,100*(S-INT(S))+.5);
530 RETURN
540 REM********
550 REM* MENU *
560 REM********
570 GOSUB410
580 FORI=1 TO 18
590 PRINT"*[38 spaces]*";
600 NEXTI
610 FORI=1TO40:PRINT"*";:NEXTI
620 PRINT CHR$(19);CHR$(19);
630 FORI=1 TO 3:PRINT:NEXTI
640 FORI=1 TO 6
650 S=4+I*2+.03
660 GOSUB510:REM CURSOR POSIT.
670 PRINTM$(I)
680 NEXTI
690 S=6.03:GOSUB510:REM CURSOR POSIT.
700 PRINTMR$(1):Z=1
710 GET MP$:IFMP$=""THEN710
720 IFASC(MP$)<>17 OR Z>=6 THEN 760
730 S=4+Z*2+.03:GOSUB510
740 PRINTM$(Z)
750 Z=Z+1:GOTO 820
760 IF ASC(MP$) <> 145 OR Z=1 THEN 800
770 S=4+Z*2+.03:GOSUB510
780 PRINTM$(Z)
790 Z=Z-1:GOTO 820
800 IF ASC(MP$) =13 THEN 850
810 GOTO 710
820 S=4+Z*2+.03:GOSUB510
830 PRINTMR$(Z)
840 GOTO 710
850 ON Z GOSUB 1000,2000,3000,4000,5000,6000
```

```
999 END
1000 PRINTM$(1):RETURN
2000 PRINTM$(2):RETURN
3000 PRINTM$(3):RETURN
4000 PRINTM$(4):RETURN
5000 PRINTM$(5):RETURN
6000 PRINTM$(6):RETURN
```

Lines 60 and 70 dimension our arrays in normal (`M$(6)`) and reverse (`MR$(6)`) video. Lines 110-140 fill the arrays with our menu options (reverse video is produced by line 130). `CHR$(18)` turns on reverse video, while `CHR$(146)` turns on normal video. The rest of the menu routine should look familiar to you. Note line 690—here is our line which dictates the cursor starting position (row 6, column 3). Line 700 conveys the first reverse-video menu option and sets `Z` to 1. Line 710 contains the inevitable `GET` statement.

Line 720 asks whether the key pressed is different from cursor-down, or if `Z` is greater than or equal to 6. The reason for this latter comparison is the total number of menu options here (six). Upon reaching the sixth option, the cursor must be blocked from going any further down. Pressing the cursor-down key (`CHR$(17)`) while in the first option moves the cursor down to the next option, putting #1 back into normal video (lines 730-740). Line 750 increments the counter `Z` by one.

The program branches to line 820, which changes the new menu option to reverse lettering. Line 760 works very much like line 720, except that 760 checks for use of the cursor-up key (`CHR$(145)`) or `Z` equalling 1. Naturally, `Z` must not equal anything less than 1, so that the cursor cannot be moved any farther up than the first option. Here too, cursor movement "blanks" the reverse video of the previous option, and switches on the current option's reversed lettering. Pressing <RETURN> activates the chosen option. You have already seen `ON Z GOSUB`.

Hope this little lecture on menus helps your work.

4.4 Window techniques

The Commodore 128 has a command which allows you to define screen windows; this command is called, obviously enough, `WINDOW` . These windows have a myriad of uses; for example, you might have a window appear onscreen, telling the user to remove the program disk and put a data disk into the drive. Once such a window is defined, all screen output is displayed within this window, as well as input generated by `INPUT`.

Several windows can be created at once. However, all input and output moves to the window created last.

Syntax: `WINDOW uc,ur,bc,br,c`

- uc = Top left column
- ur = Top left row
- bc = Bottom right column
- br = Bottom right row
- c = 1=SCNCLR 0=No SCNCLR

The maximum values for the row/column parameters are the same as those for the respective screen. Forty-column mode gives you a column range of 0-39, and rows of 0-24. Eighty-column mode gives you 0-79 and 0-24, respectively.

Now, let's draw a window at the bottom of the screen:

```
10 WINDOW 0,24,39,24
20 PRINT "COMMAND LINE!";
```

Start the program—you won't see the message, though, just the cursor. The program is right, but defining a single-line window like that will print the message, `READY`, CR, and the cursor, all on that one line. Thus, we have to somehow get out of the window to see our message.

Let's perform some minor surgery on our two-line program from above:

```
10 WINDOW 0,24,39,24
20 PRINT"COMMAND LINE!";
30 PRINT CHR$(19);CHR$(19)
```

Line 30 sends `CHR$(19)` twice, restoring the normal screen.

Starting the program will put our message into the lowest screen line, and a `READY` message in the upper left-hand corner.

Unfortunately, the Commodore 128 doesn't allow you to visually delineate a window on the 40-column screen. This program will solve the problem:

```
10 SCNCLR
20 REM RW$=40X CHR$(192)
30 RW$=CHR$(192)+ CHR$(192)+ CHR$(192)+
   CHR$(192)+ CHR$(192)
40 REM BL$=40X CHR$(32)
50 BL$=CHR$(32)+ CHR$(32)+ CHR$(32)+
   CHR$(32)+ CHR$(32)
60 FOR I=1 TO 3
70 RW$=RW$+RW$:BL$=BL$+BL$
80 NEXT I
90 RS$=CHR$(221)
100 REM WINDOW PARAMETERS
110 UC=4:UR=8:BC=14:BR=18
120 REM DRAW BORDER
130 CHAR,UC-1,UR-1,CHR$(176)+LEFT$
    (RW$,BC-UC+1)+CHR$(174)
140 FOR I=UR TO BR
150 CHAR ,UC-1,UR+Z,RS$+LEFT$
    (BL$,BC-UC+1)+RS$
160 Z=Z+1
170 NEXT
180 CHAR ,UC-1,BR+1,CHR$(173)+LEFT$
    (RW$,BC-UC+1)+CHR$(189)
190 WINDOW UC,UR,BC,BR
200 END
```

This program draws a border around any screen window.

Note: You should not use maximum window values. Any other definition is allowed.

The string variables in the first few lines are made up of CHR$ codes because this cuts down on mishaps and bugs. You can look up the codes if you're uncertain of what they represent.

The window "frame" is drawn around the window by the CHAR statement first, then the window is generated (line 190).

This is good subroutine material. The parameters given in line 110 can be given before the subroutine (falling to line 110). The parameters UC,UR,BC and BR can be input, and the program can jump to this subroutine.

There is a relative of the WINDOW command—RWINDOW(X). RWINDOW lets you determine the number of rows and columns making up the window, as well as the character mode (40 or 80 cols). X represents the following:

0 = Number of rows in the window
1 = Number of columns in the window
2 = 40- or 80-column, depending on the character mode

You can use this new information in context with the menu routines discussed earlier. Just keep in mind that you should keep the screen uncluttered.

All we have offered here are general suggestions for menus, windows and such. We urge you to change, develop and improve to your own taste. For openers, finish the Math Table program that we've alluded to in the last few subheadings.

4.5 Sort routines

Many programs need data sorted into some organized form (alphabetical order, numerical order, etc.). There are a number of different sorting procedures extant, which vary in functionality and degree of difficulty in programming. It goes without saying that the more complex the routine is, the more efficient and more versatile the sorting will also be. For our own purposes, though, we're going to look in on the simplest sorting procedures, and stay at an introductory level. Once you've learned these structures, you can go on to more complicated sorts on your own; there are a number of books on the subject.

We'll start with the *bubble sort*. Its name comes from the fact that the individual sorted elements "float" to the top, like bubbles rising in water. For the sake of illustration, we'll fill an array with random numbers and have them sorted. We take an array with 6 elements. The next few program lines dimension that array and fill in the values:

```
10 REM GENERATE ARRAY
20 DIM F(6)
30 FOR I=1 TO 6
40 A=INT(50*RND(0))+1
50 F(I)=A
60 NEXT I
  .
  .
```

The principle behind our sort routine lies in its comparing two elements with one another. If an element is larger than another, it is set farther down. Most of our routine is made up of `IF/THEN` constructs. You could also do this with `FOR/NEXT` loops, but the routine is less comprehensible then. Once you understand the basic routine, feel free to utilize `FOR/NEXT` instead. The sort proper:

```
100 REM * BUBBLE SORT *
110 Z=0
120 IF F(6)>F(5) THEN 140
130 F(0)=F(6):F(6)=F(5):F(5)=F(0):Z=1
140 IF F(5)>F(4) THEN 160
150 F(0)=F(5):F(5)=F(4):F(4)=F(0):Z=1
160 IF F(4)>F(3) THEN 180
170 F(0)=F(4):F(4)=F(3):F(3)=F(0):Z=1
```

```
180 IF F(3)>F(2) THEN 200
190 F(0)=F(3):F(3)=F(2):F(2)=F(0):Z=1
200 IF F(2)>F(1) THEN 220
210 F(0)=F(2):F(2)=F(1):F(1)=F(0):Z=1
220 IF Z=1 THEN 110
230 FOR I=1 TO 6
240 PRINT F(I)
250 NEXTI
260 END
```

Line 110 sets `Z` to 0; you'll see why shortly. Line 120 performs the first comparison. If the contents of element `F(6)` are already larger than `F(5)`, no rearrangement is made, and a direct branch to line 140 is performed. On the other hand, if `F(6)` is smaller than `F(5)`, the two are exchanged (line 130). `F(0)` is used for temporary storage of variables; thus `F(5)` and `F(6)` are rearranged. Temporary storage has been discussed in other contexts a few chapters ago.

Whenever `Z` is set to 1, an exchange has occurred; when `Z`=0, an exchange has not happened. We can check the condition of `Z` for exchanges. This little trick can be used in your own programs to see if the desired effect has been reached. The variable `Z` serves as a flag here. Once `Z` consistently reads 0, this means that our sorting is done. To smooth out the sort procedure, change these lines:

```
220 FOR I=1 TO 6
230 PRINT F(I);
240 NEXT I
250 PRINT
260 IF Z=1 THEN 110
270 END
```

The bubble sort routine can be adjusted to work with a total of 100 elements (this is the maximum for a relatively short sorting time).

Chapter 5

Principles of File Management

5. Principles of file management

5.1 Common forms of data storage

This chapter deals with the essentials of data management. We'll give you the most important commands and instructions, complete with brief examples. (See the Appendices for a complete list of file commands).

Before we go any further, let's cover a few points about storing information on diskette. Before you use a diskette for the first time, you must first *format* it. Formatting arranges *tracks* and *sectors* on a diskette that are compatible with the disk drive's operating system. This must be done, or you won't be able to find your data (or rather, the drive won't find it).

Format a diskette with the following command:

```
HEADER "DATA DISK",I85
```

`DATA DISK` is the name of the diskette, and `85` is the `ID` number of the diskette. The name can have a maximum of 16 characters. Type in the above command and then press <RETURN>. The computer asks:

```
ARE YOU SURE?
```

Pressing Y (yes) begins the formatting process (it takes about 80 seconds to format a diskette). When you format an old diskette, all previous information is destroyed—so be careful when you format diskettes.

The diskette has a directory (a sort of "table of contents") which tells the user what files are available on diskette. You have your choice of two commands: `DIRECTORY` or `CATALOG`. Put our freshly formatted diskette in the drive and type one of these previous two commands. The following directory should appear onscreen:

```
0 "DATA DISK  "85 2A
664 BLOCKS FREE
```

This directory is from a 1541 disk drive. A 1571 drive gives you a total of 1328 blocks free (because of its double-sided diskettes).

5.2 Different file types

When you save a program to diskette, the program name is placed in the directory. In addition, three characters are placed to the right. These characters are abbreviations for the type of file matching the filename. The abbreviations have the following meanings:

PRG = Program file
SEQ = Sequential file
USR = User file
REL = Relative file

PRG is the normal program created by DSAVE"*Program name*" or SAVE"*Program name*",8[,1]. A SEQuential file usually contains additional data for a file (say, files created by the user with a word processor). USR indicates a file created by the user for some special purposes of their own. This includes sprite data, which is laid out in binary form.

Yet another file type is indicated by the abbreviation REL (RELative file). This sort of file is distinguished from a sequential file by the enormous saving of time during data access. If you want to look at the 54th record in a sequential file, the system has to go through the first 53 records! But relative files allow you to go directly to record #54 without having to look through the others first.

5.3 The file

A file is a collection of data stored on some mass-storage medium (tape or disk drive), and accessible through the same medium (i.e., it can be loaded back into the computer). By definition, a program is a file. By rights, though, a file is a collection of names, numbers of other data stored on other media. Working with an external storage medium necessitates the use of the two commands DLOAD and DSAVE. Since diskette access is the most practical form of data storage, we'll limit our work here to those two commands. To load a program from diskette, type

```
DLOAD"PROGRAM NAME"
```

or

```
LOAD"PROGRAM NAME",DV
```

DV stands for device number. Most disk drives are device number 8 (DLOAD does not need a device number). The filename can have a maximum of 16 characters within quotation marks. To save a program, use

```
SAVE"PROGRAM NAME",DV
```

or

```
DSAVE"PROGRAM NAME"
```

See your 128 User Manual or the Command Overview in Appendix A of this book for more information on these commands. We refer you to Abacus Software's ***1571 Internals*** and ***Anatomy of the 1541 Disk Drive*** for more details on disk drive operation.

Here is an example of simple file management:

Data is read into a predimensioned array. The data in this array will be saved in its entirety on diskette. Some programs give you the option of searching for a file, loading it into the computer, altering the data, and finally re-saving the file to diskette. Before getting data from or writing data to diskette, you have to open a file. This is accomplished with the DOPEN command:

```
DOPEN#1,"ADDRESSES",W
```

opens a channel for data transfer to diskette, creates a sequential file called "ADDRESSES" and tells the system to write the information to diskette (,W). Sequential means that the data is lined up in the order in which it is sent to the diskette. Next command you'll need is the PRINT# command. A number follows the PRINT# command—this number matches the number used in DOPEN#. Typing in PRINT#1,D$ writes the contents of the variable D$ to diskette. When you are through writing data to diskette, you will need to close the file with DCLOSE. Again, the number of the open file should be given, i.e. DCLOSE#1.

Now that we know how to write a file, let's read our sample file. We must open a channel once again. We use a slightly different form of DOPEN:

```
DOPEN#1,"ADDRESSES"
```

Now the sequential file is opened for reading. To read the data, we have to use the INPUT# command. Again, the number must match the file number opened (this is a logical file number). The syntax looks like this:

```
INPUT#1,D$
```

this lets you read a file with a maximum of 80 characters. The GET# command, like the "normal" GET command, reads individual characters from diskette.

Another command worth knowing is the one used for deleting a file. That command is

SCRATCH"*Filename*"

After executing a SCRATCH command, the system asks:

```
ARE YOU SURE?
```

Pressing Y<RETURN> wipes the file from the directory. SCRATCH"ADDRESSES" scratches that filename from the diskette. The program below uses SCRATCH to delete a file before saving its revised version of the file to diskette. If two files of the same name exist, the disk drive status light flashes (red on the 1541, green on the 1571), indicating an error. Reading the variable DS$ gives you the message 63 FILE EXISTS. This program brings you to a menu that allows you to either input data, save, load or output files, or end the program. Choosing "Input Data" lets you input four first and last names.

This program is extremely simple, to make you comfortable with sequential file management. We hope you'll use these basics to write your own file management programs.

```
10 DIM D$(4,2)
20 SCNCLR
30 PRINT"WHAT DO YOU WANT TO DO:"
40 PRINT
50 PRINT"INPUT DATA?  (1)"
60 PRINT
70 PRINT"SAVE DATA?  (2)"
80 PRINT
90 PRINT"LOAD DATA?  (3)"
```

```
100 PRINT
110 PRINT"OUTPUT DATA? (4)"
120 PRINT
130 PRINT"END PROGRAM? (5)"
140 GETKEY A$
150 ON VAL(A$) GOTO 190,290,400,500,600
160 REM **************
170 REM * DATA INPUT *
180 REM **************
190 SCNCLR
200 FOR I=1 TO 4
210 INPUT"FIRST NAME";D$(I,1)
220 INPUT"LAST NAME";D$(I,2)
230 SCNCLR
240 NEXT I
250 GOTO 20
260 REM **************
270 REM * SAVE DATA  *
280 REM **************
290 SCRATCH"ADDRESSES"
300 DOPEN#1,"ADDRESSES",W
310 GOR I=1 TO 4
320 FOR Z=1 TO 2
330 PRINT#1,D$(I,Z)
340 NEXT Z,I
350 DCLOSE#1
360 GOTO 20
370 REM *************
380 REM * LOAD DATA *
380 REM *************
400 DOPEN#1,"ADDRESSES"
410 FOR I=1 TO 4
420 FOR Z=1 TO 2
430 INPUT#1,D$(I,Z)
440 NEXT Z,I
450 DCLOSE#1
460 GOTO 20
470 REM ***************
480 REM * DATA OUTPUT *
490 REM ***************
500 SCNCLR
510 FOR I=1 TO 4
520 FOR Z-1 TO 2
```

```
530 PRINTD$(I,Z)
540 NEXT Z,I
550 SLEEP 3
560 GOTO 20
570 REM *******
580 REM * END *
590 REM *******
600 SCNCLR
610 END
```

5.4 Relative file management

Relative files have the advantage of not having a permanent file order. You read the material you need, change it, and re-save it. First let's put the relative file on diskette. A relative file can have records with a maximum of 254 characters. This means that all file arrays can be no more than 254 characters. That number should be more than sufficient for your purposes.

Suppose we want a record with a maximum of 100 characters. The OPEN command for a relative file looks like this:

```
DOPEN#1,"REL ADDRESSES",L100
```

The disk drive opens within the file all records of 100 characters in length. No memory has been set aside on the diskette yet—we have to let the system know how many records sets to produce. We opt for 200 data records. BASIC 7.0 has a special command for positioning relative file pointers: RECORD. Its syntax is as follows:

```
RECORD#lfn,rn[,bnr]
```

lfn is the logical file number matching DOPEN#. *rn* is the data record number and *bnr* is the byte number of the record. This statement lets you set the file pointer to any record or relative file.

Now, let's set the record pointer to record 200, which doesn't exist per our instructions. The operating system displays 50, RECORD NOT PRESENT,00,00, and the status light blinks on the drive (see above). We can ignore this message.

Next, we write CHR$(255) to our record, which automatically reserves 200 records for us. Try this:

```
10 REM SET UP RELATIVE FILE
20 DOPEN#1,"REL-ADDRESSES",L100
30 RECORD#1,200
40 PRINT#1,CHR$(255)
50 DCLOSE#1
60 END
```

The LED on the drive will begin to blink after the program is run. Input PRINT DS$ to see the error message. Remember that DS$ and DS are status variables for you to check disk drive status.

Now call up CATALOG or DIRECTORY to see your relative file, which takes up 80 blocks.

A block on a diskette contains 256 bytes. Thus, the pointer must move two bytes to the next block. The relative file needs 200 * 100 = 20,000 bytes of memory for records. Another block (the side-sector block) is used to store the table for individual records. 20000 divided by 254 gives us 78.7. Since no half-blocks can be in the directory, 80 blocks are laid out. You can determine by this calculation (and the amount of free memory) how much memory your relative file takes up.

Chapter 6

Music and Graphics

6. Music and graphics

This chapter briefly descibes some of the most important commands in BASIC 7.0—those dealing with the music and graphics capabilities of the C-128. Since many books on the subject are finding their way to the bookstore, we'll just be covering the barest essentials here, and hope you get excited enough to continue your experiments in C-128 music and graphics.

6.1 Music

BASIC 7.0 offers you six commands or statements to help program the 128's "music department." The C-128 has the same SID (Sound Interface Device) chip as the C-64. However, the 64's SID has to be manipulated with numerous PEEKs and POKEs. The 128's SID doesn't need them.

We should talk a little bit about the structure of a musical tone. Every sound has three characteristics, whether it is a musical sound or simply noise:

- Frequency (how "high" or "low" the tone is)
- Amplitude (volume)
- Duration (how long the tone sounds)

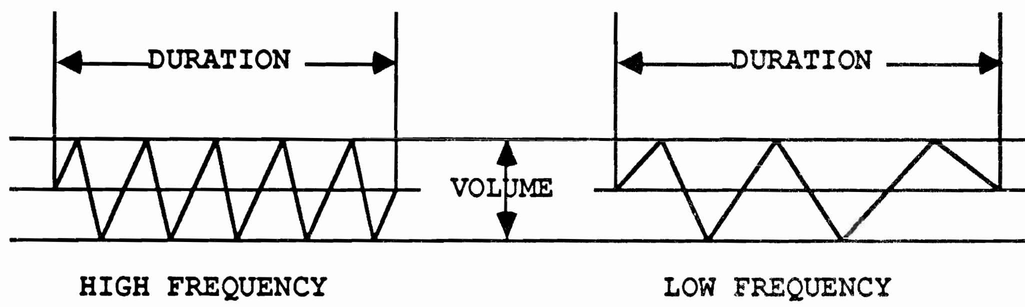
Furthermore, every sound has a waveform. The waveform dictates the tonal quality of the sound (fluty, brassy, stringy, etc.). The SID has four such waveforms:

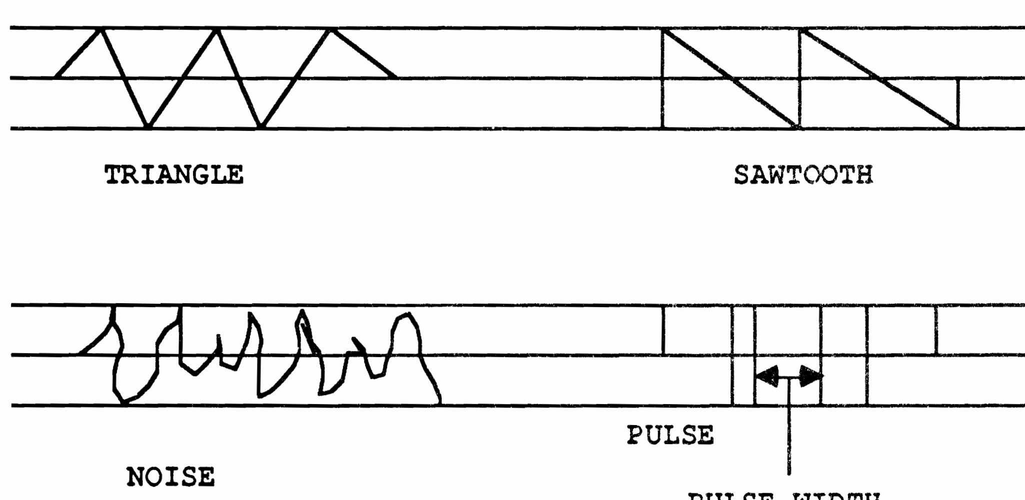
- Triangle
- Sawtooth
- Pulse
- Noise

Another factor in production of tones is the envelope. This lets you state:

- the Attack phase
- the Decay rate
- the Sustain time
- the Release time of the note.

Once you understand these terms, we can begin to use the music commands. For more information, we suggest you refer to ***C-128 Internals*** published by Abacus Software.





The simplest musical command is `PLAY`. Try typing this:

```
PLAY "C E G C"
```

The notes are rapidly played on the monitor's loudspeaker. The "long version" of command syntax looks like this:

```
PLAY "Vn,On,Tn,Un,Xn,notes"
```

`Vn` = Voice number (1-3)
`On` = Octave number (0-6)
`Tn` = Envelope (0-9)
`Un` = Volume (0-15)
`Xn` = Filter (0=off, 1=on)

Now let's have a little fun. Change the above example to this:

```
PLAY "O4 C E G O5 C"
```

Now you'll hear the `O` (octave) parameter at work (capital letter `O`, not 0). The C-128 has a 7-octave range (almost the range of an 88-key piano).

The `Tn` parameter lets you select one of 10 standard envelopes built into the C-128. These are:

0 Piano
1 Accordian
2 Calliope
3 Drum
4 Flute
5 Guitar
6 Harpsichord
7 Organ
8 Trumpet
9 Xylophone

If we wanted our little musical passage to sound like a guitar, we would call up the envelope like this:

```
PLAY "T5 O4 C E G O5 C"
```

You can also change envelope parameters. The command looks like this:

```
ENVELOPE e,a,d,s,r,wf,pw
```

e = Envelope number (0-9)
a = Attack (0-15)
d = Decay (0-15)
s = Sustain (0-15)

r = Release (0-15)
wf = Waveform (0-4)
0 = triangle
1 = sawtooth
2 = pulse (square)
3 = noise
4 = ring modulation
pw = Pulse width (0-4095)

These parameters allow you to tailor sounds to your own tastes. (Note: The *pw* parameter only applies to waveform 2).

Yet another command is used to create sound effects:

```
SOUND vc,freq,dur (,dir,min,sv,wf,pw)
```

The parameters are:

vc = Voice (1-3)
freq = Frequency (0-65535)
dur = Duration in 1/60 seconds (0-32767)
dir = Direction:
0 = ascending
1 = descending
2 = oscillating
min = Minimum if sweep (dir) is specified (0-65535)
sv = Step value for sweep (0-32767)
wf = Waveform (0-3)
pw = Pulse width (when using pulse waveform 2)

The example below makes a siren sound:

```
SOUND 1,4000,250,2,1000,100
```

The commands `TEMPO` and `VOL` control the speed and volume of the notes. `FILTER` gives you three different types of filters to "fine-tune" a sound quality. Appendix D has a description of these commands and their parameters.

6.2 Graphics

This chapter is a general overview of the most important graphics commands. It is simply an introduction to Commodore C-128 graphics.

The BASIC graphic commands work only on the 40-column screen. This screen, like the Commodore 64's, has a resolution of 320 x 200 points. You can switch to the 80-column screen with the GRAPHIC 5 command, but commands such as BOX and CIRCLE only run in 40-column mode.

C-64 users had to know its internals backwards and forwards to make use of the C-64's graphics, and tell every bit on the screen what to do. Let's face facts—the average person is not going to remember POKE 53270,(PEEK 53270) OR 16, which turns on the 64's multicolor mode. However, the average Joe can remember GRAPHIC 3, which accomplishes the same thing. The following command probably looks familiar to those of you who've owned a VIC-20 and Superexpander:

GRAPHIC

This command controls graphic modes. The syntax is GRAPHIC *m* (,*c*,*s*). The parameters in parentheses can be given, but they aren't essential. The c parameter states whether or not the system should perform an automatic SCNCLR: A 1 executes a SCNCLR, a 0 doesn't (1=on, 0=off).

The s parameter works in those modes combining text and graphics (GRAPHIC 2 or GRAPHIC 4), and states at which line the text begins (default is line 19). The m parameter calls up one of six graphic modes:

0 40-character text mode
1 Graphic mode
2 Combined graphic/text mode
3 Multicolor graphic mode
4 Combined multicolor graphic/text mode
5 80-character text mode

You must call up a graphic mode to perform any graphics commands. If no GRAPHIC command is given, and you type in BOX or SCNCLR 1, you'll get:

```
?NO GRAPHICS AREA ERROR
```

Thus, GRAPHIC must precede any graphics commands:

Wrong!	`10 SCNCLR 1:GRAPHIC 1`
Right!	`10 GRAPHIC 1:SCNCLR 1`

In the next couple of pages, we will be working on a high-resolution analog clock, which operates from TI$, and will actually draw the minute, hour and second hands. The hands are drawn by this line:

```
DRAW cs,x1,y1 TO x2,y2 TO x3,y3...
```

The color source used is declared by fs (0-1 in high- resolution mode, 0-3 in multicolor mode). The x and y parameters determine the starting and ending points (x and y axes).

High-res mode gives you a graphic page with horizontal coordinates of 0-319, and vertical coordinates of 0-199.

This program draws a point in the exact center on the screen:

```
10 GRAPHIC 1:SCNCLR
20 DRAW ,160,100
30 GETKEY A$
40 GRAPHIC 0
50 END
```

Another vital graphics command is CIRCLE. Not only can it draw simple circles, it can also do ellipses and polygons. The syntax for CIRCLE is:

CIRCLE *cs,x,y,xr,yr,sa,ea,a,inc*

cs	Color source
x,y	Coordinates for center point
xr	X-coordinate radius
yr	Y-coordinate radius
sa	starting angle of circle
ea	ending angle of circle
a	rotation in clockwise degrees (default 0)
inc	degrees between segments

6.2.1 Analog clock

Here's the `Analog Clock` program listing:

```
5 TI$="030045"
10 REM ANALOG CLOCK
15 REM CHAPTER 6.2.1
20 GRAPHIC 1:SCNCLR
30 DRAW ,162,100
40 FORI=1 TO 12
50 READ X,Y
60 CHAR ,X,Y,STR$(I)
70 NEXT I
80 FORI=100 TO 96 STEP-1
90 CIRCLE ,162,100,I,,,,,1
100 NEXT I
110 DATA 25,3,29,7,30,12,29,17,26,21,19,23,
        12,21,9,17,8,12,9,7,12,3,18,1
200 REM
210 REM HANDS
220 REM
230 DO
240 :    CHAR ,1,1,TI$
250 :    H=VAL(LEFT$(TI$,2))
260 :    M=VAL(MID$(TI$,3,2))
270 :    X= 20*SIN(M*6*π/180)
280 :    Y=-20*COS(M*6*π/180)
290 :    Y1=-10*COS((H*30+M/2)*π/180)
300 :    X1= 10*SIN((H*30+M/2)*π/180)
310 LOOP WHILE MA=M
320 CIRCLE 0,162+XA,100+YA,8,55,,,MA*6,120
330 CIRCLE 0,162+SX,100+SY,8,45,,,HA*30+MA/2,120
340 CIRCLE 1,162+X1,100+Y1,8,45,,,H*30+M/2,120
350 CIRCLE 1,162+X,100+Y,8,55,,,M*6,120
360 XA=X:YA=Y:MA=M:HA=H
370 SX=X1:SY=Y1
380 GOTO 230
400 END
```

Now for some details of the program:

Line 20 calls up graphic mode 1 and clears the screen. Line 30 places the center point on which the hands turn on the screen. Lines 40-70 fetch the positions of the numbers (for the clock face) from the DATA statements in line 110, and display the numbers with CHAR (line 60). Lines 80-100 draw the edge of the clock face (CIRCLE command). Line 240 states the time in digital form onscreen. This is also where the program gets its coordinates for hand "movement." Every minute's passage exits the loop (MA is no longer equal to M) and clears the old minute-hand (lines 320 & 330). New hands are then drawn (lines 340-350). The value of the hands at that moment are stored for clearing later, and MA is reset to equal M (lines 360-370). Finally, the program jumps back to the delay loop in line 230.

You must redefine TI$ if you want the actual time of day (TI$ counts from power-up, unless told otherwise). You could even develop the program to include a "current time" input line.

You'll find complete descriptions of the graphic commands in Appendix D.

Chapter 7

BASIC Internals

7. BASIC Internals

This chapter is not a replacement for ***C-128 Internals***. Since we are mainly concerned with BASIC, we will only touch on machine language.

7.1 The MONITOR

The Commodore C-128 has a built-in *monitor*. We're not referring to the video screen of the same name, but to a "program" that lets you view, output and change the memory contents of the C-128. This makes it possible to look at the *ROM listing* (operating system) of the C-128 in disassembled form.

The operating system has different machine language routines that handle all the internal operations of the computer, such as power-up screens, disk commands, etc. A section of the C-128's ROM looks like this:

```
F4BCC:  01 18      ORA($18,X)
F4BCE:  D0 26      BNE $4BF6
F4BD0:  24 7F      BIT $7F
F4BD2:  10 0D      BPL $4BE1
F4BD4:  20 34 4B   JSR $4B34
F4BD7:  A5 3B      LDA $3B
F4BD9:  A4 3C      LDY $3C
F4BDB:  8D 00 12   STA $1200
F4BDE:  8C 01 12   STY $1201
```

The hexadecimal numbers in the left column indicate the memory locations themselves. The middle columns are *machine code commands*, and the right-hand columns are the *assembly language commands*. This section of ROM represents the first section of the END command in BASIC. If the BASIC interpreter runs across an END, this routine is executed.

The most important monitor command is MONITOR, which switches on the monitor program. When this occurs, you'll see the following on your screen:

```
MONITOR

  PC       SR AC XR YR SP
; FB000    00 00 00 00 F8
```

Without digging any deeper into machine-code, all we'll say for now is that these are the different processor registers and their contents. The blinking cursor is waiting for some input. Type in `D F4BCC` to see our ROM listing section again. `D` means *disassemble*. The `M` command displays a section of memory, while ASCII equivalents appear at the far right. Type in `M F4BCC` to see for yourself.

Next, let's look at a BASIC program through the eyes of the monitor. Return to BASIC by typing in `X`. Now type in `NEW` and this program:

```
10 PRINT"I CAN SEE THIS IN THE MONITOR!"
20 END
```

Go to the monitor (type `MONITOR` <RETURN> or press <F8>). Now type either `M 01C00` or `M 04000`. The former address is the start-of-BASIC when no graphic pages are used. If graphics are used, the second address is start-of-BASIC, since the start is moved 9K for graphics.

`M 04000` lists memory from $4000 to $405F. What you see onscreen is the BASIC program, viewed from the machine language point of view. We see our text, which was in quotes, but there's no sign of `PRINT` or `END`. You'll recall that BASIC commands are stored as `TOKENS` (reserved numbers representing the commands, read by the interpreter). The token for `PRINT` is 153 decimal, $99 in hexadecimal. (See the Appendices for a complete list of tokens). You'll see the hex value $99 in memory location $0405—and that is our `PRINT` command.

Two locations before this, you'll see our line number in the form of $0A hex (10 decimal), followed by the low byte and high byte (30 40) of the pointer to the next program line. Change the $99 to $8F, press <RETURN> and exit the monitor.

Now list your BASIC program, which has a `REM` instead of a `PRINT` statement! You changed the tokens while in the monitor. You could handle this by `POKE`ing as well—even create a "self-modifying" program.

7.2 The variable pointer

Another handy function is POINTER (variable). This helps you find the position of the variable pointer in bank 1 of memory. Type in NEW and define the variable A$="TEST". Input PRINT POINTER(A$). The answer should be 1026 ($0402 hex). Re-enter the monitor, and ask for M 10400. Address $0402 contains a 4 (represents the four characters of TEST). The two bytes following contain the low/high byte of the location of A$'s contents (FEFA). Now give the command M 1FEFA to see the contents of A$ itself (TEST).

Chapter 8

Utilities

8. Utilities

This chapter contains some handy routines that can help you in your programming.

8.1 Hardcopy text

This first program produces a hardcopy of the screen on your printer, which is also known as a *screen dump*. You can enter this program as a subroutine, and perhaps use it for self-generating data processing or word processing. Inverse-video characters will appear on the printer as normal characters.

```
10 REM HARDCOPY TEXT
20 OPEN1,4:CMD1
30 FORI=0 TO 999
40 A=PEEK(1024+I):Z=Z+1
45 IF Z=41 THEN Z=1:PRINT#1,CHR$(13);
50 IF A > 127 THEN A=A-128
60 IF A < 32 THEN A=A OR 64:GOTO 100
70 IF A > 31 AND A < 64 THEN 100
80 IF A > 63 AND A <96 THEN A=A OR 32:GOTO 100
90 IF A > 95 AND A < 128 THEN A=A AND 31 OR 160
100 PRINT#1,CHR$(A);
110 NEXT
120 PRINT#1:CLOSE1
130 END
```

When using this program as a subroutine, remember to change the END (line 130) to a RETURN.

8.2 Binary conversion

One function that Commodore left out of the 128 is a decimal-binary converter. This routine puts the number to be converted into variable A. The equivalent binary number comes out in variable BI$.

```
100 REM BINARY CONVERSION SUBROUTINE
110 BI$=""
120 FOR D=7 TO 0 STEP -1
130 BI(D)=INT(A/2^D)
140 A=A-BI(D)*2^D
150 BI$=BI$+RIGHT$(STR$(BI(D)),1)
160 NEXT
170 RETURN
```

8.3 Output with leading zeroes

Maybe you have been writing a "transportable" BASIC program for a computer that doesn't have a PRINT USING function. This routine formats variables with leading zeroes:

```
100 PU$=RIGHT$("0000"+RIGHT$(STR$(I),LEN(STR$(I))-1),4)
```

The format will be set up with the four zeroes as a guide. For larger numbers, just increase the number of zeroes. I is the value to be formatted.

8.4 Variable flag

It can be worthwhile to have a flag jump between two values. Flags either see 0's or 1's. Often we'll see these flip-flops solved by IF...THEN and GOTO. Here is a simpler solution:

```
100 Z=1-Z
```

Now Z will be 0 to 1 to 0 to 1 ....

8.5 Program listing on diskette

This program stores a program listing to diskette in an uncompressed, straight sequential file format, rather than the usual C-128 token format. This can be quite useful when you transfer your programs to different computer systems:

```
DOPEN#1,"FILENAME",W:CMD1:LIST:DCLOSE#1
```

8.6 Reading a sequential file

The following program reads a sequential file and displays the file onscreen:

```
10 DOPEN#1,"FILENAME"
20 DO WHILE ST <> 64
30 : GET#1,A$:IFA$=""THEN 30
40 : PRINT A$;
50 LOOP
60 DCLOSE#1
70 END
```

Chapter 9

Solutions to exercises

9. Solutions to exercises

Answers to page 27

1. a) 6C b) 92
   c) BA d) F0
   e) C f) C9

2. a) 61642 b) 4712
   c) 13728 d) 597
   e) 61440 f) 2048

3. a) 183 b) 51
   c) 254 d) 21
   e) 85 f) 170

4. a) F730 b) 6000
   c) 8001 d) ABCD
   e) FFFE f) 4711

Answers to page 46

1. a) legal
   b) legal
   c) AUTO illegal
   d) illegal; contains logical operator OR.
   e) IF illegal
   f) GB legal
   g) 4NAME% illegal—the identifier begins with a number.
   h) 255 illegal (see g)
   i) legal

2.
```
10 INPUT A,B,C,D
20 PRINT A;B
30 PRINT C,D
40 END
```

Have you noticed the difference between a comma and a semicolon in output?

Note that in INPUT awaiting several items, if you input only one item and <RETURN>, two question marks appear onscreen. This means that the computer is asking,"Where's the rest of it?? I need more data."

3.
```
10 REM INPUT HEIGHT AND
20 HYPOTENUSE C IN METERS
30 INPUT"INPUT H,C";H,C
40 A=.5*H*C
50 PRINT"AREA IS ";A;" SQUARE METERS."
60 END
```

4.
```
10 INPUT YOUR HEIGHT IN CENTIMETERS';CM
20 REM CALCULATE IDEAL WEIGHT
30 IG=CM-100
40 REM 10%
50 PR=IG/100*10
60 IG=IG-PR
70 PRINT"YOUR IDEAL WEIGHT IS ";IG;"KG"
80 END
```

This program can be condensed somewhat—lines 30, 50 and 60 can be stated in <u>one</u> line, like this:

```
30 IG=(CM-100)-(CM-100)/100*10
```

This line is a lot less readable than the three separate lines above, but it also saves some memory. There are a number of ways of writing programs—"all roads lead to Rome", as the old saying goes. So you can compromise for readability or saving memory—take your choice.

5.
```
10 INPUT"HEIGHT, LENGTH, DEPTH IN CM";H,L,T
20 REM CALCULATE VOLUME
30 V=H*L*T
40 REM CALCULATE LITERS
50 V=V/1000
60 PRINT"AQUARIUM CONTAINS ";V;"LITERS"
70 END
```

Line 30 computes the volume in cubic centimeters; the volume is divided by 1000 (line 50) and we have the aquarium volume in liters.

6.
```
10 INPUT A,B,C,D
20 PRINT"A";A
30 PRINT"B";B
40 PRINT"C";C
50 PRINT"D";D
60 END
```

Answers to page 55

1.
```
10 REM CLEAR SCREEN
20 PRINT CHR$(147)
30 REM GENERATE RANDOM NUMBERS
40 D1=INT(6*RND(1))+1
50 D2=INT(6*RND(1))+1
60 REM DISPLAY RESULT
70 PRINT"DIE 1:";D1,"DIE 2:";D2
80 END
```

As we've said earlier, your programs should look something like the "answer" program, but there is no one answer. Line 20 clears the screen with a CHR$ code (if you use SCNCLR instead, it achieves the same effect). Lines 40 and 50 give us our random numbers D1 and D2.

2.
```
10 REM INPUT VALUES OF TRIANGLE SIDES
20 INPUT"INPUT A,B,C IN CM";A,B,C
30 REM COMPUTE S
40 S=.5*(A+B+C)
50 REM CALCULATE AREA
60 F=SQR(S*(S-A)*(S-B)*(S-C))
70 REM DISPLAY AREA
80 PRINT"TRIANGLE AREA";
90 PRINT"IS";F;"SQUARE CM"
100 END
```

3.
```
10 INPUT"PLEASE PRESS A KEY";A$
20 A=ASC(A$)
30 PRINT"ASCII VALUE OF";A$;"="A
40 END
```

Pressing a comma or just <RETURN> will not give the ASCII values for those "characters". Commas will turn up an `ILLEGAL QUANTITY` error (`INPUT` doesn't like commas), and <RETURN> yields an empty string.

4.
```
10 G=9.81
20 INPUT"HOW MANY SECONDS";T
30 S=.5*G*T^2
40 PRINT"THE OBJECT FELL FROM";
50 PRINT"A HEIGHT OF";S;"METERS."
60 END
```

Line 10 is of especial interest; variable initialization gives `G` a value of 9.81 . Variable initialization means simply that the variables are set up at the beginning of the program. This has the advantage that the variables are declared once and done with. You'll find this useful in larger programs—it saves time and memory.

5.
```
10 INPUT"HOW MANY LITERS USED";L
20 INPUT"HOW MANY KM TRAVELED";KM
30 V=L/KM*100
40 PRINTV;"LITERS USED PER 100 KM."
50 END
```

Answers to page 67

1. a) is correct.
2. `DRIPS`
3. `ROTOR`
4. ```
   B$=MID$(A$,4,1)+MID$(A$,8,1)+MID$(A$,6,1)
   +MID$(A$,10,1)
   ```

That's one solution.

Answers to page 77

1.
```
10 REM INPUT ANNUAL INCOME
20 INPUT"ANNUAL INCOME IN $";JV
30 IF JV>50000 THEN 70
40 REM COMPUTE 33 %
```
```

```
50 ZS=JV/100*33
60 GOTO 90
70 REM CALC. 51%
80 ZS=JV/100*51
90 PRINT"TAXES OWED:";
100 PRINT"$";ZS
110 END
```

Line 20 asks for your annual income. This is put into variable JV. Line 30 tests for income above $50,000. If not, 33% is figured out, and the taxes owed conveyed. If your income is $50,000 or over, 51% is computed.

2. There are at least two ways of doing this assignment. First, the solution with IF...THEN.

```
10 REM SUM OF 1-100
20 A=A+1
30 S=S+A
40 IF A < 100 THEN 20
50 PRINT"SUM OF 1 TO 100 IS ";S
60 END
```

The second method involves an arithmetic sequence, i.e., the difference between individual terms is constant. The sum is computed by the formula Sn=n/2(A1+An). n=the number of terms encountered, A1=the first term, and An=the last term.

```
10 INPUT"NUMBER OF TERMS";N
20 INPUT"FIRST TERM";A1
30 INPUT"LAST TERM";AN
40 REM CALC.
50 SN=N/2*(A1+AN)
60 REM OUTPUT
70 PRINT"SUM IS";SN
80 END
```

3.

```
10 REM 6 FROM 49
20 Z=Z+1
30 L=INT(49*RND(1))+1
40 IF Z> 6 THEN END
50 PRINT L;
60 GOTO 20
```

4. Keep in mind here that the last values of A and Z are to be given. Because of this, the PRINT command must sit outside the loop proper. Your answer must be:

```
52        9
```

5.
```
10 REM INPUT STRING AND PARTIAL STRING
20 INPUT"WHICH STRING";A$
30 INPUT"WHICH PARTIAL STRING";B$
40 I=I+1
50 C$=MID$(A$,I,LEN(B$))
60 IF C$=B$ THEN PRINT" INCLUDED":END
70 IF I>LEN(A$)
    THEN PRINT"NOT INCLUDED":END
80 GOTO 40
```

Granted, this problem was tough. The hardest part of this program is the comparison string at line 50. Essentially, the program takes a string and searches another string to see if the two have similarities. The MID$ function must work with the length of the string being searched. The counter in line 40 moves the position of C$ one character to the right in A$. Line 60 has the comparison which checks for agreement between the search string (B$) and the momentary string in C$. Line 70 asks if the entirety of A$ has been searched, and if B$ is not contained in A$. An illustration:

String A$="INFORMATION" is searched for B$="FORMAT". Character total of B$=6. The following partial strings are checked:

1 INFORM
2. NFORMA
3. FORMAT

That program was a hard nut to crack, huh? But once you break programs down into smaller bites, it's always easier to work them out, however complex they are.

Answers to Page 99

1.
```
10 REM HARMONIC SEQUENCE
20 SCNCLR
30 PRINT"UP TO WHICH SUM"
40 PRINT"SHOULD I ADD?"
50 PRINT
60 INPUTS
70 Z=1
80 SH=SH+1/Z
90 Z=Z+1
100 IF Z=50*INT(Z/50) THEN PRINT
     Z;"ADDITIONS"
110 IF SH < S THEN 80
120 PRINT"THE SUM "SH" COMES
     AFTER "Z" TERMS."
```

Lines 20 to 60 clear the screen and ask up to which sum the program should add. Line 70 sets the counter to 1, and line 80 forms the sums; after this, the counter is incremented by 1 (line 90). Line 100 checks whether the counter has reached 50, or if a multiple of 50 has been reached. An output occurs after 50 elements. It doesn't have to be a multiple of 50. You can change it to whatever you wish. Line 110 checks whether the sum requested at line 60 has been reached. Line 120 is the closing line, telling you how many runs the program has made.

2.
```
10 REM QUADRATIC EQUATIONS
20 SCNCLR
30 PRINT"INPUT THE COEFFICIENTS A,B,C"
40 PRINT
50 INPUT A,B,C
60 IF A=0 THEN 20:REM A MUST BE <> 0
70 D=B*B-4*A*C
80 IF D<0 THEN 140
90 X1=(-B+SQR(D))/(2*A)
100 X2=(-B-SQR(D))/(2*A)
110 PRINT"SOLUTION TO X1=";X1
120 PRINT"SOLUTION TO X2=";X2
130 GOTO 150
140 PRINT"NO ZEROES."
150 END
```

The transposition of the problems in this program should pose no problem. Watch for A=0, though, in division of 2*A; division by zero isn't allowed.

3. Hope you got this right: First the screen clears, then the "NUMBER YOU GAVE IS ILLEGAL" appears. It's important to realize the execution of the command after the GOTO.

Answers to page 128

1.
```
10 REM READ NAMES
20 DIM Y$(6)
30 FOR I=1 TO 6
40 INPUT"NAME";Y$(I)
50 NEXTI
60 REM 1ST NAME IN ALPHABETICAL ORDER
70 Y$(0)=Y$(1)
80 FOR I=2 TO 6
90 IF Y$(0)<=Y$(I) THEN 110
100 Y$(0)=Y$(I)
110 NEXT I
120 PRINT"1ST NAME";Y$(0)
130 END
```

The first section of the program should pose no difficulties. You have already seen how the FOR...NEXT loop gives us a total of 6 names. The second section is tricky, though. And if you <u>have</u> figured it out, give yourself a pat on the back! We have already discussed temporary storage of variables, numbers, etc. We'll need this technique again for this program. No data can be lost. What string variable you use is immaterial. Here we used Y$(0), since we haven't used it elsewhere. Line 70 takes the contents of Y$(1) and stores them in Y$(0). Line 80 starts the FOR...NEXT loop with a beginning value of 2. Line 90 compares the individual names in the set with one another. If the name in Y$(0) is "less than" what is in Y$(I), the program branches to the NEXT at 110, and the floating variable is incremented by 1. If, however, Y$(0)'s name is "greater than" Y$(I), then Y$(0) arranges the name from Y$(I). When the floating variable reaches 6, the name sought is found in Y$(0).

We told you it wasn't going to be easy, but after all, racking your brains now and then is fun, isn't it?

A word about comparing strings: Performing this comparison is a matter of comparing individual characters with one another (i.e., comparing ASCII values of characters). So, "HAND" is "less than" the string "HANS", since ASCII 68 ("D") is less than ASCII 83 ("S").

2.
```
10 REM READ NUMBER
20 DIM X(6)
30 FOR I=1 TO 6
40 X(I)=1 TO 6
50 NEXT I
60 REM SEARCH FOR HIGHEST NUMBER
70 X(0)=X(1)
80 FOR I=2 TO 6
90 IF X(0)>=X(I) THEN 110
100 X(0)=X(I)
110 NEXT I
120 PRINT"LARGEST NUMBER=";X(0)
130 END
```

This program has the same structure as exercise 1. If you solved #1, the odds are good that you also solved this one. The differences lie in the type of array (numerical here) and the random number generation in line 40. The comparisons to find the largest number are based on the same principle as in Problem 1. Line 90 looks for greater/lesser numbers, eventually coming up with the largest number.

3.
```
10 REM ORDINAL RULE FOR NUMBERS
20 DIM X(6)
30 FOR I=1 TO 6
40 X(I)=I*I-I
50 NEXT I
60 REM ARRAY OUTPUT
70 FOR I=1 TO 6
80 PRINT X(I)
90 NEXT I
100 END
```

The values in this problem will be formed from multiplication by itself (`I`) and subtraction by `I`. This solution is, naturally, only a suggestion. If you came to the solution in another manner, that doesn't mean your answer is wrong.

Appendices

Appendix A

COMMAND OVERVIEW—BASIC 2.0

ABS

Category: Numeric function

Syntax: `ABS(X)`

Function: Gives the absolute value of X (no leading character + or –). The absolute value of a negative number is reached by multiplying X by -1.

AND

Category: Logical operator

Syntax: `(expression) AND (expression)`

Function: Logical AND is used in Boolean algebra to compare for truth between two expressions. The result is true only when both expressions are true.

ASC

Category: String function

Syntax: `ASC("X")`

Function: Gives the ASCII value of X. It's handled as a string function to accommodate letters and other characters. A null string ("") gives an ?ILLEGAL QUANTITY ERROR.

ATN

Category: Numeric function

Syntax: `ATN(X)`

Function: Gives the angle with the tangent of `X` (in radians).

CHR$

Category: String function

Syntax: `CHR$(X)`

Function: Calling this function converts any number for `X` into an ASCII character. The number must be between 0 and 255.

CLOSE

Category: Command

Syntax: `CLOSE X`

Function: Closes file `X`, whereby `X` represents the logical file number. `X` should be equal to the file number used in the `OPEN` command (see `OPEN`).

CLR

Category: Command

Syntax: `CLR`

Function: This command clears all variables, arrays, and user-defined functions. Not to be confused with `NEW`, which destroys everything.

CMD

Category: Command (output)

Syntax: `CMD X`

Function: Sends data which usually appears onscreen to another peripheral, e.g., printer, disk drive or cassette drive. X is the file number from the OPEN command used to communicate with the peripheral device. PRINT and LIST transmit the data to the peripheral.

CONT

Category: Command

Syntax: `CONT`

Function: CONT can continue the run of a program halted by the STOP key, STOP command or END command. The program picks up after the interruption. This command cannot be used when program changes have been made, nor when an error message has appeared.

COS

Category: Numeric function

Syntax: `COS(X)`

Function: This command gives the cosine of number X. X is the radian measure of an angle.

DATA

Category: Statement

Syntax: `DATA X,Y,Z`

Function: This statement can feed in program information, which can be read by a `READ` command. This information can consist of numbers or strings (letters/numbers). Certain characters, such as commas, spaces and colons, can be used in `DATA` elements, but only if these characters are placed in quotation marks. `DATA` is read from left to right.

DEF FN

Category: Statement

Syntax: `DEF FN F(X)=X*Y`

Function: This statement lets the programmer design a mathematical function which can be called just by calling the function name. `F`=function name; `(X)`=variable; `X*Y` the function itself. The function can be ANY mathematical statement.

DIM

Category: Statement

Syntax: `DIM A(X)`

Function: This statement dimensions an array or a matrix (multidimensional array). `(X)` is the index; you can also talk about indexed variables, in this case `A(X)`. `A(X)` names a one-dimensional array, and `A(X,Y)` names a multi-dimentional array.

END

Category: Command

Syntax: `END`

Function: This command ends a program at this point, and responds with `READY:` Similar to `STOP`. `CONT` can be used after both commands.

EXP

Category: Numeric function

Syntax: `EXP(X)`

Function: Yields the `X`th exponent of the constant `e` (2.718281823), as long as `X` is no larger than 88.0296919.

FN

Category: Numeric function

Syntax: `FN A(X)`

Function: Gives the function value of `X` declared in `DEF FN A(X)`.

FOR...TO...(STEP)

Category: Command

Syntax: `FOR X=1 TO 10 STEP 2`

Function: Gives you the ability to use a variable `(X)` as a counter. You must give the starting value of the variable (here 1), the ending value (here 10) and the step size (here 2). No step value sets an increment of 1. Any floating-point number can be used.

FRE

Category: Numeric function

Syntax: `FRE(X)`

Function: Gives you the number of bytes free available for your programming. `X` can be either 0 or 1.

GET

Category: Command

Syntax: `GET X$`

Function: Reads individual characters from the keyboard. Characters read in by `GET` are arranged into a variable (here `X$`). If the variable is numeric `(X)` and a character other than a number is read, a `"SYNTAX ERROR"` occurs. Best avoided by accepting numbers as string input and converting that input to numeric values later.

GET#

Category: Command (input)

Syntax: `GET#1,X$`

Function: Reads individual characters from a peripheral device opened by an `OPEN` command. Here 1 is the logical file number, and `X$` is the variable into which the characters are placed. This command works virtually the same as `GET`.

GOSUB

Category: Command

Syntax: `GOSUB XX`

Function: This command branches to a subroutine within a program. The XX is the line number at which this subroutine begins. The subroutine ends with RETURN, which sends the program to the point immediately after the GOSUB command. **NOTE:** Do not GOTO a branch outside the subroutine, or the computer runs into the subroutine again (see the appropriate chapter).

GOTO

Category: Command

Syntax: `GOTO XX`

Function: The program branches to a section indicated by XX (=line number). Program run is influenced by these line numbers.

IF...THEN

Category: Command

Syntax: IF *(expression)* THEN *(expression)*

Function: Tests for a condition within a program. Once this condition is true, the instruction after THEN is executed. If the condition is false, then what follows THEN is ignored, and the program goes on to the next line. The expression after IF is a comparison in most cases (e.g., IF A=10 THEN), and the expression after THEN can be any BASIC command.

INPUT

Category: Command

Syntax: `INPUT X` or `INPUT"comment";X`

Function: Offers you the option of entering your own data to be used by the program. When the program encounters this command, a question mark and the cursor appear onscreen. Now the user data can be entered, ended with a <RETURN>. The `INPUT` statement can be followed immediately by a variable, or a comment can be inserted between `INPUT` and variable (variable must then be preceded by a semicolon). `INPUT` can only be used in a program.

INPUT#

Category: Command (input)

Syntax: `INPUT#1,X$`

Function: `INPUT#` has a similar function to `GET#`, with the distinction that a maximum of 80 characters can be put here, rather than one character at a time. 1 indicates the logical file number used to open the peripheral (with `OPEN 1`). `X$` is the variable into which the data is organized. `INPUT#` ends when it runs into a semicolon, comma, colon or <RETURN> (`CHR$(13)`).

INT

Category: Numeric function

Syntax: `INT(X)`

Function: Gives an integer value for `X`, i.e., no decimal numbers regardless of the size of the fractional numbers. The result is always slightly larger or smaller than `X`. This function also works with negative numbers, e.g., `INT(- 1.23)` gives -2.

LEFT$

Category: String function

Syntax: `LEFT$(X$,A)`

Function: Gives the characters to the left of X$, starting with the leftmost character in X$. A is the number of characters to be read from X$. LEFT$(X$,4) reads the first four characters of X$. A is a number between 0 and 255. Any number beyond this range gives up an ILLEGAL QUANTITY ERROR.

LEN

Category: Numeric function

Syntax: `LEN(X$)`

Function: Gives the number of characters in X$: Counts all characters in the string, including spaces.

LET

Category: Command

Syntax: `LET X=5`

Function: The LET command assigns a string or value to a variable. In this case, X is equal to 5. LET is seldom used in programming, however—you would simply say X=5.

LIST

Category: Command

Syntax: LIST *(line no.)* - *(line no.)*

Function: The LIST command makes it possible to see the BASIC program lines in memory at that moment, either all at once or

selected lines. `LIST` alone gives the entire program. Giving `LIST` *line number* shows the line requested. Used in conjunction with `CMD`, the list can be sent to another peripheral, e.g., printer or diskette. `LIST` used within a program stops the program run in C-64 mode.

`LIST` (gives complete program listing)
`LIST 10` (gives program line 10)
`LIST -100` (gives program listing up to line 100)
`LIST 100-` (gives program listing from line 100 to the end)
`LIST 10-20` (lists lines 10 to 20)

Program run does not cease when a `LIST` is called in 128 mode!

LOAD

Category: Command

Syntax: `LOAD"`*program name*`",DV,SA`

Function: `LOAD` brings a program into memory from cassette drive or disk drive. `LOAD` alone calls up the next program on tape. Typing in `LOAD`, the program name in quotes and a device number (`DV`=8) calls the program in from the disk drive and starts the program at decimal memory address 2048. When the secondary address (`SA`=1) is given, the program is loaded in at the memory location at which it was saved.

LOG

Category: Numeric function

Syntax: `LOG(X)`

Function: Gives you the natural logarithm of `X` (based on the constant `e`). `X` must be greater than zero.

MID$

Category: String function

Syntax: `MID$(X$,A,B)`

Function: This function defines a partial string from `X$`; `A` is the starting position of the partial string, `B` is its length. `A` and `B` are numbers between 0 and 255.

NEW

Category: Command

Syntax: `NEW`

Function: `NEW` destroys the program currently in memory. You should ideally type this in before any program input. Be careful with this command; people have been known to wipe out programs accidentally before saving them.

NEXT

Category: Command

Syntax: `NEXT X`

Function: `NEXT` identifies the end of the loop opened with `FOR X= 0 TO 10`. When a program runs across a `NEXT` command, `X` is incremented by 1 until the end value is reached (10 here). Several loops can be ended at once with one `NEXT` by stating variables in reverse order and separating them with commas, e.g., `NEXT X,Y,Z`.

NOT

Category: Logical operator

Syntax: `NOT X`

Function: `NOT` performs a comparative operation—TRUE= -1, FALSE = 0.

ON

Category: Command

Syntax: `ON X GOTO 10,20,30 / ON X GOSUB 100,200`

Function: This gives us a conditional branch option, according to the value of `X`—the system branches to another program line (or subroutine, in the case of `ON X GOSUB`). If `X`=1, the above examples would branch to line 10 and line 100, respectively. Use of this command set can save a lot of time and memory instead of `IF/THEN`.

OPEN

Category: Command (input/output)

Syntax: `OPEN X,DV,SA`

Function: `OPEN` makes data exchange between the computer and peripherals possible. `X` is the logical file number (see `CLOSE`, `INPUT#` and `GET#`); it can be between 0 and 255, but most of the numbers above 128 are for specific peripheral functions. Use no logical file number, then, over 128. `DV` stands for `DeVice` number, e.g., 8=disk drive 1, 9=disk drive 2, 1=cassette drive, 4=printer. After `DV`, a comma and the secondary address (`SA`) follows. The secondary addresses of different peripherals have different functions. See your handbook for secondary addresses.

OR

Category: Logical operator

Syntax: `(expression) OR (expression)`

Function: The logical operator OR checks for validity between two or more expressions, much like AND. The difference here is that only one expression in OR need be "true" for the entire value to be "true".

PEEK

Category: Numeric function

Syntax: `PEEK(X)`

Function: Gives a number between 0 and 255, which represents the contents of memory address X. X must lie in the range between 0 and 65535.

POKE

Category: Command

Syntax: `POKE X,A`

Function: This function is the reverse of PEEK; POKE writes A into location X (A is between 0 and 255—X can be a location between 0 and 65535).

POS

Category: Numeric function

Syntax: `POS(A)`

Function: This function gives you the screen line of the cursor at which the next PRINT command will occur.

PRINT

Category: Command

Syntax: `PRINT (`*`variable`*`) or PRINT "`*`TEXT`*`"`

Function: The `PRINT` command is probably the most versatile command in Commodore BASIC (or in any BASIC, for that matter). It is primarily used for displaying data or information onscreen. `PRINT(`*`variable`*`)` displays the variable; `PRINT "TEXT"` shows the string `TEXT` onscreen.

PRINT#

Category: Command (output)

Syntax: `PRINT#X,"DATA"`

Function: `PRINT#` will write data to a file first opened by `OPEN`. `X` is the logical file number, and "`DATA`" stands here for special commands or variables which should be written to the file.

READ

Category: Command

Syntax: `READ X`

Function: Reads the elements of a `DATA` line and puts them into variable `X`. The elements to be read must match the given variable type—`X` only applies to numeric values. If more `READ` commands are given than there are `DATA` elements, an `?OUT OF DATA ERROR` is displayed.

REM

Category: Statement

Syntax: REM *TEXT*

Function: REM is reserved for putting comments within a program, to explain individual sections, program flow, or instructions. Programs will ignore lines with REM statements in them, so do not put any commands after REMs.

RESTORE

Category: Command

Syntax: RESTORE

Function: READ sets the pointers to the next DATA element to be read. RESTORE resets the pointer to the start of DATA. This allows you to read elements multiple times.

RETURN

Category: Command

Syntax: RETURN

Function: This command concludes a subroutine. On finding a RETURN, the system jumps to the main program, to a point immediately after the GOSUB command.

RIGHT$

Category: String function

Syntax: `RIGHT$ (X$,A)`

Function: Gives a partial string on the right side of X$, the length of which is stated by A. A can be a value between 0 and 255. If the value of A is larger than the total string length, the entire string is given.

RND

Category: Numeric function

Syntax: `RND(X)`

Function: RND produces a random number between 0.0 and 1.0. The value of X has no meaning if negative. Positive X- values give random numbers with a starting value of X.

RUN

Category: Command

Syntax: RUN (*line number*)

Function: RUN starts a program. You can also give a line number to start the program at a point other than the beginning. All variables are automatically set to zero. If you'd prefer to avoid resetting the variables, use GOTO (*line number*) instead.

SAVE

Category: Command

Syntax: `SAVE "NAME",DV`

Function: SAVE stores a program on tape or diskette. SAVE without a secondary address defaults to cassette. SAVE"*prg*",8 saves the program to disk (8 is the disk drive address).

SGN

Category: Numeric function

Syntax: `SGN (X)`

Function: This function gives you an integer, dependent on the size of X. If X > 0 you get 1; X = 0 yields 0; and X < 0 results in -1.

SIN

Category: Numeric function

Syntax: `SIN (X)`

Function: Gives you the sine of the angle from X (X is in radians).

SPC

Category: String function

Syntax: `SPC (X)`

Function: SPC is used in conjunction with PRINT; the cursor moves X spaces to the right, and prints at that point. X can be between 0 and 255.

SQR

Category: Numeric function

Syntax: `SQR (X)`

Function: Gives the square root of X. X cannot be negative.

STEP

Category: Command

Syntax: `STEP X`

Function: STEP is used in loop programming. It gives the step value for the loop. X can be any value except zero. When STEP is unused, the step value is equal to 1.

STOP

Category: Command

Syntax: `STOP`

Function: STOP is used within a program to stop the program flow at that particular point. BREAK IN *line number* appears onscreen, listing the place of interruption. CONT continues the program run.

STR$

Category: String function

Syntax: `STR$ (X)`

Function: Converts X into a string. If X is positive or null, the string begins with a space.

SYS

Category: Command

Syntax: `SYS X`

Function: `SYS` calls a machine language program from BASIC, at address `X`. `X` can be a number between 0 and 65535. One well-known `SYS` command is the system cold-start (`SYS 64738` in C-64 mode, `SYS 16384` in C-128 mode).

TAB

Category: String function

Syntax: `TAB (X)`

Function: Tab moves the cursor `X` columns to the right, beginning from the leftmost position of the current line. `X` must be between 0 and 255. `TAB` is used in conjunction with `PRINT`, although `PRINT#` cannot interpret `TAB`.

TAN

Category: Numeric function

Syntax: `TAN (X)`

Function: Gives the tangent of the angle of `X` (`X` is in radians).

USR

Category: Numeric function

Syntax: `USR (X)`

Function: `USR` branches to a machine-code subroutine which must start at addresses 785-786 (64 mode) and 4633-4634 (128 mode). This works something like a `POKE` command. The value `X` is given to the machine language program, and another value is returned to BASIC. This parameter set-up does not exist in `SYS`.

VAL

Category: Numeric function

Syntax: `VAL (X$)`

Function: `VAL` converts `X$` into a numeric value. When encountering a string of letters, `VAL` converts the first character into numeric representation. If the first character is a space, plus or minus sign, the value returned is null.

VERIFY

Category: Command

Syntax: `VERIFY "FILE",DV`

Function: `VERIFY` ensures that a program saved to disk or tape matches the program still in memory. `DV` is the device number. `VERIFY"*",8` automatically verifies the last program saved to disk, and checks it with the program in memory.

WAIT

Category: Command

Syntax: `WAIT X,Y`

Function: The program pauses until memory location `X` has bit pattern `Y`.

Appendix B

COMMAND OVERVIEW–BASIC 7.0

AUTO

Category: Command

Syntax: `AUTO` *(step size)*

Function: This command switches on automatic line numbering. Pressing <RETURN> after a line number generates a new line number. If you've typed in `AUTO 10`, you have to type in the first line number. After that, though, the completion of line 10 <RETURN> generates line number 20, then 30, and so on. If the opening line is 25, the next will be 35, 45, etc.

`AUTO  10` - Automatic line numbering in steps of ten.

`AUTO 100` - Automatic line numbering in steps of 100.

`AUTO` - Switches `AUTO` off.

BANK

Category: Statement

Syntax: `BANK` *(no.)*

Function: This allows you to choose one of the 64K banks of memory, used in `PEEK`, `POKE` and `WAIT` instructions.

Example: `BANK 0`

`BANK 1` is chosen for `PEEK`, `POKE` and `WAIT`.

```
10 BANK 0:POKE 8000,255
20 BANK 1:POKE 8000,100
```

```
30 BANK 0:PRINT PEEK(8000)
40 BANK 1:PRINT PEEK(8000)
```

Start this short program to see how important BANK is.

Many of you probably know about POKE from the Commodore 64, particularly from changing screen color (e.g. POKE 53280,0 = black border). This address works on the C-128 when BANK 4 is called up only.

BEGIN...BEND

Category: Command

Syntax: IF..THEN..BEGIN*(instruction)*..BEND:ELSE

Function: BEGIN/BEND is a powerful extension of IF..THEN. Branching in IF..THEN required a program line after THEN—a limitation not found in BEGIN/BEND.

BEGIN/BEND recognizes a block of statements which are controlled by IF..THEN, and all this sequence can be set into one line. BASIC 7.0 permits entire program sections to be set in an IF..THEN construct with BEGIN/BEND. The C-64 does not have this command.

It should be self-evident that IF..THEN..BEGIN/BEND constructs can be nested if necessary. Example:

```
10 A=INT(50*RND(1))+1
20 PRINT "YOU HAVE TO GUESS A NUMBER"
30 PRINT"BETWEEN 1 AND 50."
40 INPUT"YOUR GUESS";R
50 IF R<>A THEN BEGIN:
60 :  IF R<A THEN PRINT"TOO SMALL."
70 :  IF R>A THEN PRINT"TOO BIG."
80 BEND:GOTO 40:ELSE PRINT"RIGHT!!":END
```

After typing this program into the computer, input a number between 1 and 50 when the prompt asks for it. The computer will see if your number is larger than, smaller than

or equal to random number A, and you are told how you did (TOO SMALL, TOO LARGE or RIGHT). This process is run in lines 50-80. You can see the BEGIN/BEND construct in these lines. Here is another example:

```
10    INPUT"HOW MANY RUNS";A
20    IF A<100 THEN BEGIN
30    : PRINT"THE NUMBER CHOSEN WAS ";A
40    : FOR Z=1 TO A
50    : PRINT"BEGIN..BEND DEMO"
60    : NEXT Z
70    : PRINT"HERE'S THE END!"
80    BEND:ELSE PRINT"NUMBER TOO LARGE!":END
```

BOOT

Category: Command

Syntax: BOOT"*FILENAME*" *(,DR#,UNIT#)*

Function: This command is used to load and run binary files. Obviously the files must be binary files the computer can read.

Example: BOOT "MACH III" loads and runs the file "MACH III". BOOT can also load and run binary files saved with BSAVE; the files are loaded at the starting address at which they were saved by BSAVE.

BOOT without a program name causes the system to look at the first track on the diskette. If a certain byte sequence is found—for instance, CBM—the system initializes that program.

For an in-depth explanation of BOOT and autostart routines, refer to the Abacus book *Tricks and Tips for the C-128.*

DEC

Category: Function

Syntax: `DEC (hex expression)`

Function: This function converts hexadecimal numbers into their decimal equivalents. This can come in handy for quick hex/dec conversions, say if you need an address in ROM in decimal.

Example: `PRINT DEC("0AFF")` gives you 2815 decimal. The following program converts $00-$FF to 0-255 decimal.

```
10   FOR Z=48 TO 70
20   IF Z=58 THEN Z=65
30   FOR I=48 TO 70
40   IF I=58 THEN I=65
50   Z$=CHR$(Z)+CHR$(I)
60   GOSUB 100
70   NEXT I:NEXT Z
90   END
100  PRINT "$";Z$;" =";DEC(Z$):RETURN
```

DELETE

Category: Command

Syntax: `DELETE (line no.)-(line no.)`

Function: This command deletes one or more BASIC program lines. This command is particularly useful when you want to delete a number of lines from a pre-existing program. The syntax rules for this command are similar to `LIST`.

`DELETE 10`	deletes line # 10.
`DELETE -100`	deletes all lines up to 100.
`DELETE 200-250`	deletes lines 200 to 250.
`DELETE 500-`	deletes all lines from 500.

DO/LOOP/WHILE/UNTIL/EXIT

Category: Statement

Syntax: `DO:(UNTIL` *instruction* / `WHILE` *instruction: instruction:* `EXIT)`
`LOOP (UNTIL` *instruction* / `WHILE` *instruction*`)`

Function: `DO..LOOP` offers you a simple method of designing program loops which were previously available only with structured languages. The simplest form is a loop starting with `DO` and ending with `LOOP`. `LOOP` can then be followed by `WHILE` and/or `UNTIL`. An `EXIT` can be put inside this `DO..LOOP`; when `EXIT` is found, the system executes the next command following `LOOP` (something like `FOR...NEXT`).

```
10   DO: PRINT"COMMODORE 128"
20   Z=Z+1
30   LOOP UNTIL Z=25
40   END
```

The loop runs until `Z=25` (line 30). `UNTIL`, then, is used to wait for a specific goal or condition.

```
10   DO:PRINT"COMMODORE 128"
20   Z=Z+1
30   LOOP WHILE Z<25
40   END
```

This example shows an application for `WHILE`. The loop runs as long as `Z` < 25 (line 30). `WHILE` is used during a specific goal or condition. Once the condition is false (`Z>25`) the program ends.

```
10   DO
20   PRINT"WHICH COMPUTER DO YOU PREFER?"
30   INPUT A$
40   LOOP UNTIL A$="COMMODORE 128"
50   PRINT"GOOD CHOICE."
60   END
```

The loop runs until you give the right answer (COMMODORE 128). Add these lines to install an EXIT:

```
35 IF A$="COMMODORE 128" THEN EXIT
40 LOOP
```

ERR$

Category: Function

Syntax: ERR$(*X*)

Function: This function describes an error number in string form. It can be used in connection with TRAP, RESUME and system variables ER and EL to see error messages without leaving a program.

```
10   FOR I=1 TO 41
20   PRINT ERR$(I)
30   NEXT I
40   END
```

This program gives you all the available error strings.

Use ERR$ in your error routines if you want the original error messages.

FAST/SLOW

Category: Command

Syntax: FAST

Function: FAST shifts the C-128 to a speed of 2 mHz (twice the speed of normal SLOW mode). This is advantageous for performing long computations. Unfortunately, FAST mode affects the 40-character screen—any input after this command will not be seen. The 80-character screen remains unaffected.

SLOW returns the C-128 to 1 mHz, and reactivates the 40-column screen. You can use FAST to render the 40-column screen invisible while a graphic is being drawn, and put a "PLEASE WAIT" message in 80 columns; then use SLOW to bring things back to normal. Here is a sample:

```
10   REM 1 MHZ
20   FOR I=1 TO 5000:NEXT I
30   GETKEY A$
40   FAST : REM 2 MHZ
50   FOR I=1 TO 5000: NEXT I:SLOW:END
```

FETCH

Category: Statement

Syntax: FETCH *a,b,c,d*

Function: FETCH takes a complete memory range from banks 1-15 and loads the range into BASIC memory (bank 1). This means that it's possible to store variable contents in another bank and pull it back into working memory (cf. STASH).

The parameters *a,b,c* give the number of bytes, starting and goal addresses. Parameter *d* is the bank to be read. *a,b* and *c* can values each up to 65535 and *d* can be between 1 and 15.

```
FETCH 144,24576,16381,15
```

GETKEY

Category: Statement

Syntax: GETKEY *A$*

Function: GETKEY works similarly to GET, with the distinction that GETKEY automatically waits for a keypress, while GET must have program code written around it specifying the wait.

GET must be written like this:

```
10 GET A$:IFA$=""THEN 10
```

This is unnecessary with GETKEY. GETKEY waits until a key is pressed, then assigns the value to a variable:

```
10 GETKEY A$
```

If you want the program to react to an "A", do this:

```
.
.
.
70 GETKEY A$
80 IF A$ <> "A" THEN 70
```

GO 64

Category: Statement

Syntax: GO 64

Function: Switches the computer from 128 mode to 64 mode. After typing in GO 64, the computer asks the user to confirm:

```
ARE YOU SURE?
```

Press Y, and the 64 mode power-up screen appears. Now you can use the wealth of software written for the '64. Changing back to 128 mode is a matter of pressing the <RESET> button or (less graceful) switching the computer off and on.

HELP

Category: Command

Syntax: HELP

Function: Press this key whenever you get an error message; the system displays the line at which the error occurred. The

error will be in reverse video (40-cols.) or underlined (80-cols.).

```
10 SCNCLR
20 GOSUP 100 REM INITROUTINE
.
```

?SYNTAX ERROR IN 20 appears—press <HELP> and the bad line will be shown:

```
20 GOSUP 100 REM INITROUTINE
```

HEX$

Category: Function

Syntax: HEX$ (*X*)

Function: This function converts a decimal number from 0 to 65535 to an equivalent hex number.

Example: PRINT HEX$(49152) gives C000. Used in conjunction with DEC, you can easily convert number bases back and forth.

```
10   OPEN 1,4:CMD1
20   FOR I=0 TO 255
30   PRINT#1,I;"= ";HEX$(I)
40   NEXT I:CLOSE 1
```

IF/THEN/ELSE

Category: Statement

Syntax: IF....THEN....:ELSE

Function: The importance of IF..THEN has already been stressed in the BASIC 2.0 overview. BASIC 7.0 offers the completing statement ELSE, which lets the user branch on an "if not" condition. For example:

```
110 IF A$="Y" THEN A=5:  ELSE GOSUB 1000
```

Any string but "Y" sends the program to the subroutine at line 1000.

NOTE: ELSE must be separated from THEN by a colon.

Example:

```
10 GETKEY A$
20 IF ASC(A$) > 47 AND ASC(A$) < 58 THEN
   30:ELSE IF ASC(A$)=69 THEN 30:ELSE 10
30 PRINT A$;
```

This example is only affected by inputting numbers between 0 and 9, or the <E> key to end the program.

INSTR

Category: Function

Syntax: INSTR (*str1,str2(,stnr)*)

Function: This function gives the position at which string 2 (*str2*) can be found in string 1 (*str1*). If string 2 doesn't exist, the value returned is 0. *stnr* tells the starting position at which string 1 should be searched.

Example:

```
10   A$="COMMODORE 128":B$="128":C$="DORE"
20   P1=INSTR(A$,B$)
30   P2=INSTR(A$,C$,3)
40   PRINT P1,P2
```

Output: 11   6

Line 20 looks for B$ within A$ starting at the first character; the result is put into P1. P1 contains 11, which is the position at which B$ was found. Line 30 searches A$ for C$. The result is 6, stored in P2.

JOY

Category: Function

Syntax: `JOY (X)`

Function: JOY reads the joystick ports. X signifies the joystick port to be read (1=port 1, 2=port 2). The values for each positon have the following meanings:

0 no function
1 up
2 up/right
3 right
4 down/right
5 down
6 down/left
7 left
8 up/left
128 fire button

Values over 128 signify the fire button plus a stick direction. This command takes a lot of effort out of programming joystick control—the C-64 required a great deal of POKEing to use joysticks.

KEY

Category: Command

Syntax: `KEY (KEY #,"FUNCTION")`

Function: This command serves two purposes: KEY alone gives a list of functions assigned to the function keys (F1-F8). KEY and a number allows you to redefine that function key.

```
KEY
```

```
KEY 1,"GRAPHIC"
KEY 2,"DLOAD"+CHR$(34)
KEY 3,"DIRECTORY"+CHR$(13)
KEY 4,"SCNCLR"+CHR$(13)
```

```
KEY 5,"DSAVE"+CHR$(34)
KEY 6,"RUN"+CHR$(13)
KEY 7,"LIST"+CHR$(13)
KEY 8,"MONITOR"+CHR$(13)
```

`KEY 6,CHR$(27)+"X"` defines function key 6 for an <ESC> X sequence (40/80 toggle).

MID$

Category: Function

Syntax: `MID$(A$,X,Y)`

Function: MID$ is a sensible development in BASIC 7.0. You can reorganize individual characters into a string.

```
10   A$="COMMODORE 64 "
20   MID$(A$,11,3)="128"
30   PRINT A$
40   END
```

Output: `COMMODORE 128`

This is possible in the example because we have deliberately made A$ thirteen characters long. If the space after 4 wasn't there, this reorganizing would not have worked.

MONITOR

Category: Command

Syntax: `MONITOR`

Function: Activates the onboard machine language monitor, allowing you to easily program in machine code. You have such commands as A(ssemble), D(isassemble), C(ompare), F(ill), H(unt), M(emory), R(egister), T(ransfer), etc. at your disposal. X returns you to BASIC 7.0.

G run a program (GO)
L load a file
S save a memory range to peripheral
V verify file
X exit to BASIC
> change memory
; change processor register
@ drive status

See Chapter 7 for more on the monitor.

PEN

Category: Function

Syntax: `PEN (X)`

Function: Determines the current position of a lightpen. `PEN(0)` gives the X-coordinate; `PEN(1)` gives the Y-coordinate of the pen.

The lightpen is used for applications or even for menu options. You move the pen to the desired spot onscreen and press the button (or <CTRL> key in some cases); it is at this moment that the lightpen position is registered.

The values must be properly prepared. The X values must be in the range between 60 and 320; the Y-coordinate must be in the range from 50 to 250 (Note: not in the ranges 0-319 or 0-250, respectively).

POINTER

Category: Function

Syntax: `POINTER (variable)`

Function: The pointer address of a variable name is returned. As long as a variable is defined, whether in direct mode or program mode, the pointer can read the variable. You can see the

contents of strings, for example, and find out the memory location of the variable.

Example:

```
A$="TEST"

PRINT POINTER(A$)
```

Output: `1034`

POT

Category: Function

Syntax: `POT (X)`

Function: `POT (X)` reads the paddles.

`X=1` Position of paddle 1
`X=2` Position of paddle 2
`X=3` Position of paddle 3
`X=4` Position of paddle 4

The following program moves sprite 1 horizontally; the direction is dependent upon the position of paddle 1.

```
10   SPRITE 1,1,1:REM SPRITE 1 ON
20   PA=POT(1)    :REM READ PADDLE 1
30   MOVSPR 1,PA,90
40   GOTO 20
```

PRINT USING

Category: Statement

Syntax: `PRINT USING "FORMAT";A`

Function: `PRINT USING` allows formatted output of strings and/or numbers. The format is set up in quotation marks, and the

variables follow that format. Formatting is defined as follows:

#	mark number of digits
+	positive number output
-	negative number output. Plus or minus signs appear only if specifically requested by user
.	decimal point indicator
$	dollar sign
^^^^	number is to be printed in scientific notation (e.g.,1.23 E+02)
=	string centered in format array
>	string right-justified in format array

Example:

`A` signifies the following:

```
A=34.57: A=1234: A=5421.236: A=54632
```

```
PRINT USING "####.##";A
```

Output:

```
  34.57
1234.00
5421.24
*******
```

A number larger than the given format will be replaced with asterisks in the format array.

The same values can be conveyed in scientific notation:

```
PRINT USING "#.##^^^^";A
```

Output:

```
3.46E+01
1.23E+03
5.42E+03
5.46E+04
```

PUDEF

Category: Statement

Syntax: `PUDEF "(1-4 chars)"`

Function: `PUDEF` allows you to change the characters in the `PRINT USING` instruction. The first position in the string (max. 4 characters) is defined as a space. The second is a comma, the third a decimal point and the fourth, a dollar sign.

Example: Normally the format would be filled in by spaces. Here we'll use an asterisk instead (e.g., for "footnote" purposes). Further, we want to add a period for the thousand place, and turn the decimal point into a comma (European style).

```
10   PUDEF "*., "
20   A=4325.674
30   PRINT USING  "###,###.##";A
```

Output: `**4.325,67`

RCLR

Category: Function

Syntax: `RCLR (X)`

Function: Gives us current color of color source. `RCLR` is a value between 1 and 16, matching the color values (1=black, etc.).

The following values are permissible for `X`:

0 = 40-column screen color
1 = graphic character color
2 = graphic multicolor character color 1
3 = graphic multicolor character color 2
4 = 40-column border color
5 = 40/80-column text color
6 = 80-column screen color

This function serves to test for color values in different text and graphic modes. No memory location is mentioned—you'll have to PEEK around a bit for that once you find the mode. Any mode can be read at any time with RCLR(*X*), regardless of the mode you are in at the time.

RDOT

Category: Function

Syntax: RDOT (*X*)

Function: This function supplies the position of the invisible graphic cursor, as well as the current color source at that cursor point.

RDOT(0) gives the X-position of the graphic cursor.
RDOT(1) gives the Y-position of the graphic cursor.
RDOT(2) gives the current color memory.
(0 background color=no point set)
(1 foreground color=point set)

Example:

```
10   GRAPHIC 1,1:REM CLR & GRAPHIC MODE 1
20   GRAPHIC 0
30   FOR I=0 TO 2
40   PRINT RDOT(I),
50   NEXT I
60   GRAPHIC 1:LOCATE 150,120:GRAPHIC 0
70   FOR I=0 TO 2
80   PRINT RDOT(I),
90   NEXT I
```

RENUMBER

Category: Command

Syntax: `RENUMBER` (*new starting line, increment, old starting line*)

Function: Renumbers program lines using given parameters. *increment* is the distance between program lines (default is 10).

Example: `RENUMBER` changes program to line numbers in steps of 10, beginning with line 10 (i.e., 10,20,30,etc.).

`RENUMBER 1000,20,1` 1 is the original starting line of the program; the program's new first line number becomes 1000, and later lines are incremented by 20s.

When you want to change line numberings from line 120 to increments of 10, do this: `RENUMBER 120,10,120`.

RESUME

Category: Statement

Syntax: `RESUME` (*line number* / `NEXT`)

Function: `RESUME` is used in connection with `TRAP`, to continue a program run. `RESUME` without line number or `NEXT` reruns the program at the line where the error occurred, thus giving you an endless loop. `RESUME` *line number* runs the program from that line number stated (hopefully, a line after the error). `RESUME NEXT` runs the program immediately after the error. Example:

```
10 TRAP 1000
20 PRINT "THERE IS A SYNTAX ERROR IN LINE 30."
30 PRINT:PRIEMT
40 PRINT"RESUME NEXT SENDS PROGRAM TO LINE 40."
50 PRINT"AND PROGRAM CONTINUES DESPITE ERROR."
60 END
1000 RESUME NEXT
```

RGR

Category: Function

Syntax: `RGR (A)`

Function: Gives the current graphic mode in numeric form (i.e. between 0 and 5).

Example: `PRINT RGR (A)` gives you 5. graphic mode 5=80-character text mode.

This function lets you find out exactly which mode you are in. This comes in handy for programs which switch between 40- and 80-column mode. **NOTE:** The `A` in `RGR (A)` is handled as a dummy argument, i.e., the current contents of `A` are of no consequence.

RREG

Category: Statement

Syntax: `RREG(a,x,y,s)`

Function: After a `SYS` call, this instruction gives the contents of the accumulator (*a*), the X-register (*x*), the Y-register (*y*) and the status register (*s*).

Type `SYS 828`, which jumps into the monitor and displays a `BREAK`. Look at the registers:

```
SR  AC  XR  YR

33  FF  00  00
```

Type `X` to exit the monitor. Now input `RREG A,X,Y,S`. You can now read these registers in decimal form with a `PRINT` statement:

```
PRINT A,X,Y,S
```

Output: `255 0 0 51`

RWINDOW

Category: Statement

Syntax: `RWINDOW (X)`

Function: `RWINDOW` gives the row and column of a window, as well as character mode (40/80). `X` has the following meanings:

0 gives number of rows in the window
1 gives the number of columns in the window
2 gives 40 or 80, dependent on character mode

```
PRINT RWINDOW (0)
```

Output: `24` (24 rows)

You can determine the size of the window you created with the `WINDOW` command, and use this knowledge to adapt or format your screen output.

SLEEP

Category: Command

Syntax: `SLEEP` *x*

Function: `SLEEP` is a simplified time-loop command (x=1 to 65535). *x* is the duration in seconds.

Example: `SLEEP 10` provides a 10-second time loop. BASIC 2.0 and 4.0 had no `SLEEP` command; you had to set up time loops with `FOR I=1 to x:NEXT`. `SLEEP` is a simple alternative.

STASH

Category: Statement

Syntax: `STASH` *a,b,c,d*

Function: `STASH` permits complete memory ranges to be taken from BASIC memory (bank 1) and moved to external memory banks. Thus, variable contents can be moved to another bank, and stored temporarily for later retrieval (cf. `STASH`). Naturally, `STASH` can only be used for writing or storing memory. Be careful that you don't accidentally overwrite variables or program registers.

The parameters *a,b,c* give the starting address and goal addresses. Parameter *d* determines the bank to which the memory is to be written. *a,b* and *c* have maximum values of 65535; *d* is in a range between 1 and 15.

Example: `STASH 1000,16831,16831,4` writes 1000 bytes from address 16831 in bank 1 to the same memory address in bank 4.

SWAP

Category: Statement

Syntax: `SWAP a,b,c,d`

Function: Switches complete memory ranges between bank 1 (BASIC working memory) and other memory banks. Thus it's possible to exchange variable contents between two banks—contents that may have originally been placed in another bank by `STASH`. `SWAP`, of course, can only perform successful swaps to free RAM areas, lest occupied ranges be overwritten (cf. `STASH`).

Parameters *a,b,c* give the number of bytes, starting address, and goal starting address. Parameter *d* determines the bank by which the exchange will be made. *a,b* and *c* can be any number up to 65535 and *d* must be between 1 and 15.

Example: `SWAP 1000,16831,16831,4` exchanges 1000 bytes starting at 16831, bank 1, with 1000 bytes of bank 4 at the same address.

SYS

Category: Statement

Syntax: `SYS address ,a,x,y,s`

Function: BASIC 7.0 lets you convey additional parameters in `SYS` commands. These parameters control the accumulator *a*, the X-register *x*, the Y-register *y*, and the status register *s*. Example: Enter the monitor and type in this little m/l program beginning at $04000.

```
STA $D020
STX $D021
RTS
```

Exit the monitor (X) and type in BANK 4. The SYS command below will now change the screen and border colors.

SYS 4*4096,0,1 (black border, white screen)

TRAP

Category: Statement

Syntax: TRAP *line number*

Function: This can initiate a reaction to any program errors within a program. Any error will send the program to the line number specified; RESUME will continue the program (cf.). You could have read the system variable EL, as well as ERR$(ER). TRAP can also be used to detect a non-functional peripheral.

Example:

```
10    TRAP 100
20    DOPEN#1,"ADDRESSES"
30    INPUT#1,A$
40    DCLOSE#1
50    PRINTA$;
60    END
100   IF ER=5 AND EL=30 THEN PRINT"PLEASE
       TURN ON YOUR DISK DRIVE AND PRESS A KEY."
110   GETKEY B$:RESUME
```

TROFF

Category: Statement

Syntax: TROFF

Function: TROFF turns TRACE mode off (TRace OFF). The program runs normally, without displaying program flow.

TRON

Category: Command

Syntax: `TRON`

Function: This command switches on TRACE mode (TRace ON). TRACE mode shows the program line number being executed while the program is running.

These two commands, TRON and TROFF, are very practical tools for the programmer, allowing him/her to follow program flow, and hunt for sources of errors.

WINDOW

Category: Command

Syntax: `WINDOW` *uc,ur,bc,br,c*

Function: This command generates a window on the current screen. The maximum parameters are those of the screen size (40 or 80 columns).

uc = upper left column
ur = upper left row
bc = bottom right column
br = bottom right row
c = is an optional parameter. c=1 automatically clears screen in the window.

Example:

```
10 WINDOW 30,15,39,24,1
```

This line defines a window in the upper right screen. This could be used for sub-menus.

Note: Windows can also be defined with <ESC> T and <ESC> B, and cursor positioning.

XOR

Category: Function (logical operator)

Syntax: `XOR (X, Y)`

Function: Parameters `X` and `Y` are compared by this "exclusive `OR`" operator. Values between 0 and 65535 should be used.

Example:

```
PRINT XOR(0,0),XOR(0,1),XOR(1,0),XOR(1,1)
```

Output: `0 1 1 0`

```
PRINT XOR(23,56)
```

Output: `47`

23 = 010111 binary
>> XOR = 101111 = 47
56 = 111000 binary

Appendix C

DATA MANAGEMENT COMMANDS IN BASIC 7.0

APPEND

Category: Command

Syntax: `APPEND #lfn,"filename"Dr# on U#`

Function: This command opens the file "*filename*" with the logical file number `lfn`, sets the file pointer to the end of the file, making additional appending of data to this file possible. Double drives can utilize the parameters `Dr#` (drive number) and `U#` (unit number).

Example:

```
10 DOPEN#1,"TEST 1",D0,W
20 FOR I = 33 TO 65:PRINT#1,CHR$(I);:NEXT
   I: DCLOSE#1
40 DOPEN#1, "TEST 1"
50 GET#1,A$: IF A4="" THEN 50
60 PRINT A$:IF ST<>64 THEN 50:ELSE
   DCLOSE#1:END
100 APPEND#1,"TEST 1",D0 ON U8
110 FOR I = 66 TO 90
120 PRINT#1,CHR$(I);
130 NEXT:DCLOSE#1:END
```

This program writes `CHR$` codes from 33 to 65 to the file `TEST 1`; after this, the file is read and the characters displayed onscreen. Typing in `RUN 100` will add `CHR$(66)` to `(99)`, and `RUN 60` will read the file out again.

BACKUP

Category: Command

Syntax: `BACKUP Dr# to Dr# ,ON U#`

Function: This command copies an entire diskette onto another diskette. The BACKUP command only works with dual-disk-drive systems. Example:

```
BACKUP D0 TO D1, ON U8
```

This example copies the contents of the disk in drive D0 onto the diskette in drive D1. The double disk drive here has a device number of 8.

BACKUP makes an exact copy of the original diskette, i.e., any data already on the destination diskette will be destroyed. BACKUP will format the destination diskette before copying, so all data on that diskette will be overwritten.

This command is handy for making a backup copy of a data disk. BACKUP is not useful for copying only a few files, though: For that you use COPY.

BLOAD

Category: Command

Syntax: BLOAD"*filename*" *ON* Bnk,Psa

Function: Loads a binary file. Psa is the starting address; Bnk is the bank number into which the binary file is to be loaded.

Example: BLOAD "BPROG" ON B1, P4000 loads BPROG into bank 1 starting at address 4000. This syntax is also allowed: BLOAD "BPROG",B1,P4000.

BLOAD"SPRITE",B0,P3584 loads a binary file into sprite memory; sprite data must go here.

BSAVE

Category: Command

Syntax: `BSAVE "`*`filename`*`",Bnk,Psa TO Pea`

Function: Saves a file "*filename*" as a binary file. `Bnk`=bank number, `Psa`=starting address and `Pea`=ending address.

Example: `BSAVE"MACH 1",B1,P16384 TO P17408` saves the memory registers from 16384 to 17408, bank number 1, to disk as file "`MACH 1`".

You can load and instantly run programs saved from the monitor using this command and `BOOT`. In addition, you can save sprite data with `BSAVE` and call the data back with `BLOAD`.

To save sprite data:

```
BSAVE"SPRITES",B0, P3584 TO P4096
```

CATALOG

Category: Command

Syntax: `CATALOG Dr#,U#,`*`wild cards`*

Function: Displays a disk directory onscreen. If `CATALOG` is given alone on a double-drive system, both drives will be searched for directories. *Wild Card* parameters let you look for similar file names. Example: `CATALOG D0,U8,"??B?R"` displays all five-character file entries with similar third and fifth characters, such as `AMBER`, `UMBER`, `IMBAR`, `FUBAR`, `TABER`, and so on, that are on the disk. `CATALOG "TES*"` shows all filenames beginning with `TES`.

`CATALOG` doesn't overwrite the program presently in memory. `LOAD "$",8` could not perform this function.

COLLECT

Category: Command

Syntax: `COLLECT Dr# ON U#`

Function: This command opens up space on a diskette taken up by improperly-closed files (known as "splat files"; the filenames are usually ended by an asterisk "*"). `COLLECT` also "shuffles" the disk space to make room (similar to `VALIDATE` in BASIC 2.0).

Examples: `COLLECT D0 ON U8` or `COLLECT D0,U9` or `COLLECT`.

Any one of these will work to free up disk space. However, do not use this command within a program:

```
10 OPEN 15,8,15;PRINT# 15,"V":CLOSE 15
```

CONCAT

Category: Command

Syntax: `CONCAT Dr#, "Sourcefile" TO Dr#, "Dest.file" ON U#`

Function: This command joins sequential files—source file to destination file.

Example: `CONCAT D0,"FILE 1" TO D0,"FILE2" ON U8` links the two files.

```
10 DOPEN#1,"DATA1,W"
20 PRINT#1,"0123456789"
30 DCLOSE#1
40 DOPEN#!,"DATA2",W
50 PRINT#1,"987654321"
60 DCLOSE#1
70 CONCAT "DATA1" TO "DATA2"
```

"DATA2" now contains the data '9876543210123456789'.

COPY

Category: Command

Syntax: `COPY Dr#,"Source file" TO Dr#,"New file" ON U#`

Function: Copies files from one drive to another, or from one disk to another in the same drive.

Examples: `COPY D0,"CON" TO D0,"CONBACKUP" ON U8` makes a copy of "CON" with a new filename "CONBACKUP". `COPY D0,"CON" TO D1,"TEST"` copies CON from drive 1 to drive 2, and names the drive 2 file "TEST" (double drive).

You can use the COPY command when one or more files need to be copied. Data on the destination diskette will remain untouched, and not be overwritten (see BACKUP). Be mindful, however, of the fact that the same filename cannot be used in COPY (i.e., `COPY D0,"JOETTE" TO D0,"JOETTE"` is illegal).

DCLEAR

Category: Instruction

Syntax: `DCLEAR Dr# ON U#`

Function: Resets all disk drive channels.

WARNING!: Be sure that you DCLOSE# all files before doing a DCLEAR, or you'll be stuck with still-open files (i.e., you won't be able to get into them)!

Example:

```
10 DOPEN#1,"ADDRESS",W
20 PRINT#1,"ADDRESS 1"
30 PRINT#1,"ADDRESS 2"
40 DCLOSE#1
50 DCLEAR
```

DCLOSE

Category: Instruction

Syntax: `DCLOSE #lfn ON U#`

Function: Closes all files open at the moment. `#lfn` is the logical file number, and `U#` the unit number.

Example: `DCLOSE#1` closes all files opened by `DOPEN#1`. `DCLOSE` closes all open files.

```
10   DOPEN#1,"ADDRESSES",W
20   DOPEN#2,"ADDRESSES-GE"
30   PRINT#1,"ADDRESS 1"
40   PRINT#1,"ADDRESS 2"
50   INPUT#2,AD$
60   DCLOSE#1:DCLOSE#2
70   DCLEAR
```

Line 60 could be replaced with just a `DCLOSE`.

DIRECTORY

Category: Command

Syntax: `DIRECTORY Dr#,U#,"Filename"`

Function: Displays a disk directory onscreen (see "`CATALOG`"). "`Filename`" can have wild cards (*, ?) here also.

Examples:

`DIRECTORY` — shows complete directory.

`DIRECTORY D0,"?S*"` — shows all entries with "`S`" as second character.

`DIRECTORY D1,"TEXT*"` — shows all entries beginning with "`TEXT`".

`DIRECTORY` allows you to view a directory without destroying the present memory contents, unlike the BASIC 2.0 `LOAD "$",8`.

DLOAD

Category: Command

Syntax: `DLOAD"`*`Filename`*`" ,Dr#,U#`

Function: Loads a program from diskette into the computer. `DLOAD` can also be used within a program.

Examples: `DLOAD"GRAPHICS"` loads the program `"GRAPHICS"` into memory. `DLOAD(A$)` loads the program named by the contents of `A$`.

```
10 INPUT "PROGRAM NAME";A$
20 DLOAD (A$)
```

DOPEN

Category: Instruction

Syntax: `DOPEN #lfn,"`*`filename`*`",Dr#,U#,Lx/W`

Function: Opens a file for reading or writing. If the file is to be relative, `L` (record length) has to be used. Any other files being written require a `W` suffix. (`lfn`=logical file number, `Dr#`=drive number, and `U#`= unit number).

Examples: `DOPEN#1,"ADDRESSES"` opens the file `"ADDRESSES"` for writing. `DOPEN#1,  "TEST",L15,D1,U9` opens drive 1, device 9 for writing to a relative file 15 records long. `DOPEN#1,"ADDRESSES"` opens `"ADDRESSES"` for reading.

DSAVE

Category: Command

Syntax: `DSAVE "`*`filename`*`",Dr#,U#`

Function: Saves a program to diskette. You should use `SAVE` for cassette storage.

cassette storage.

Examples: `DSAVE "TEST"` saves the file "TEST" to disk. `DSAVE"TEST",D1,U9` saves the file "TEST" to disk (drive 1, unit 9).

"TEST" can also be conveyed as a string variable—`A$="TEST"`, and saved as `DSAVE(A$)`. `DSAVE` is designed to save you typing time as with the "old" BASIC (`SAVE"TEST",8`).

DVERIFY

Category: Command

Syntax: `DVERIFY Dr#,U#`

Function: `DVERIFY` compares the program in memory with the program on diskette. If the program was correctly saved, the `OK` message is displayed. If the two programs don't match, a `?VERIFY ERROR` message will appear. BASIC programs saved on the C-64 can be loaded into the 128 without further ado. Type in `DVERIFY`, though; the starting address in the 128 is different than the address on the diskette, so you get an error. To avoid this, be sure that the program is saved in 128 mode, then everything will be OK.

HEADER

Category: Command

Syntax: `HEADER "DISK NAME",Iid,Ddr#,U#`

Function: Formats a new disk. Any data on an old disk will be destroyed. `Iid` is a two-character ID marker. If a disk is already formatted, you can leave the ID out, which will clear the directory and give the disk a new name. Names can have a maximum of 16 characters.

Examples: `HEADER "DAA DISK",I85,D0` gives the disk the name "DAA DISK" and an ID of 85. This procedure takes about

80 seconds. HEADER "DATA DISK 1",D0 erases the directory and renames the disk (fast format). This can only be done to already-formatted disks. Fast formatting takes about 2 seconds.

RECORD

Category: Instruction

Syntax: `RECORD #lfn,dnr,bnr`

Function: Sets a relative file pointer to any byte (bnr=byte number) of any record (dnr=record number).

Example:

```
10   DOPEN#1,"ADDRESSES"
20   RECORD#1,5,10
30   INPUT#1,A$:DCLOSE#1
40   PRINT A$;:END
```

`This program opens the relative file "ADDRESSES" for reading. The file pointer is set to the 10th byte of the 5th record. INPUT# reads and outputs the appropriate field.

The RECORD command simplifies reading material in a relative file. Only the record number need be given of all the above parameters. Setting the pointer to a record on the C-64 required a lot of roundabout programming. For example:

```
PRINT#1,"P"+CHR$(2)+CHR$(10)+CHR$(1)+CHR$(5)
```

RENAME

Category: Command

Syntax: RENAME "*old name*" TO "*new name*",Dr#,U#

Function: Renames files.

Example: RENAME "ADDRESSES" TO "HOUSE NUMBERS",D0 changes the old name "ADDRESSES" to a new filename

"HOUSE NUMBERS". Note that no other file on the disk can have the same filename as the new name. You can also perform this function in BASIC 2.0, albeit not as gracefully:

```
10 OPEN 1,8,15,"R:HOUSE NUMBERS=ADDRESSES":CLOSE 1
```

SCRATCH

Category: Command

Syntax: SCRATCH "*filename*" ,Dr#,U#

Function: This command deletes a file from the disk directory. The computer asks ARE YOU SURE? Press Y to scratch the file.

Example:

```
SCRATCH "ADDRESSES"
ARE YOU SURE?
```

A Y scratches the file.

WARNING: The SCRATCH command also works with wild cards. SCRATCH "TA?" wipes out TAN, TAP, TAB or any other three-character word starting with TA; SCRATCH "TA*" destroys these plus any other word starting with TA, such as TANGENTS, TABER, TARARABOOM, etc. Whatever you do, don't use SCRATCH "*"—it scratches the entire disk!

APPENDIX D

GRAPHICS AND SOUND COMMANDS AND FUNCTIONS

Sprites

The major distinction between sprites and shapes lie in that sprites are controlled by the VIC chip. Thus, sprites are faster and can be moved about the screen more easily than shapes. Furthermore, sprites can collide and be read for this collision.

BUMP

Category: Function

Syntax: `BUMP (X)`

Function: `BUMP` determines which sprite collision has occurred since the last reading. `BUMP(1)` handles the information of which sprites have collided with one another, and `BUMP(2)` determines which sprites have collided with the background.

The values presented by `BUMP` are conveyed by bit positions. In other words, if `BUMP(1)=129`, then sprites 1 and 8 have touched. If `BUMP(2)=15`, then sprites 1,2 and 3 have all hit a background character.

COLLISION

Category: Statement

Syntax: `COLLISION (A,ln)`

Function: This statement presses the reaction to a sprite/sprite or sprite/background collision; additionally, `COLLISION` checks for lightpen activity. When a collision ocurs, the program jumps to a user-written subroutine at line number

ln (remember to end the subroutine with a RETURN). A can be one of the following values:

1 Sprite/sprite collisions
2 Sprite/background collisions
3 Lightpen

Example:

```
90   ...
100  COLLISION 1,1000
.
1000 SOUND 3,965,60
1010 RETURN
```

A sprite/sprite collision branches the program to line 1000; a noise is made, and the program returns to the main section.

MOVSPR

Category: Statement

Syntax: MOVSPR *s,x,y* or MOVSPR *s,a#f*

Function: MOVSPR lets you easily place a sprite at any X- and Y-coordinates onscreen. Alternatively, you can give the sprite continuous movement at a certain angle and speed.

Examples: MOVSPR 1,120,90 positions sprite 1 at x=120 and y=90.

MOVSPR 3,90#5 sends sprite 3 moving constantly at a 90-degree angle, at speed 5.

This command gives you quick and painless sprite movement—a feature not available on the C-64.

RSPCOLOR

Category: Function

Syntax: `RSPCOLOR (X)`

Function: This function tests which multicolor values were last used by sprites. It returns to you the values defined by the `SPRCOLOR` command. `RSPCOLOR (1)` gives the first parameter and `RSPCOLOR(2)` the second, defined by `SPRCOLOR a,b`.

Example: `PRINT RSPCOLOR(1)` Output: 2

You can also see the previously-defined values from `SPRCOLOR 2,12` by including in a program:

```
PRINT RSPCOLOR(1),RSPCOLOR(2)
```

or you could organize the lines as variables:

```
100 A%=RSPCOLOR(1):B%=RSPCOLOR(2)
```

RSPPOS

Category: Function

Syntax: `RSPPOS (`*`sprite number,`*` X)`

Function: You can read the X-position, Y-position or the speed of any sprite. X can equal one of these values:

0 Current X-position
1 Current Y-position
2 Speed (0-15)

Examples: `PRINT RSPPOS (1,0)` gives the X-position of sprite 1.
`PRINT RSPPOS (3,1)` gives sprite 3's Y-position.
`PRINT RSPPOS (2,2)` gives the speed of sprite 2 (see `MOVSPR`).

RSPRITE

Category: Function

Syntax: `RSPRITE (A,X)`

Function: RSPRITE reads the different parameters of a sprite. A= the sprite number, and X equals one of the following:

0 Sprite on or off
1 Sprite color
2 Background priority (1=Y/0=N)
3 X-direction expansion (1=Y/0=N)
4 Y-direction expansion (1=Y/0=N)
5 Multicolor mode (1=Y/0=N)

Examples:

`PRINT RSPRITE (2,1)` Output:1 (sprite 2 is black)
`PRINT RSPRITE (5,5)` Output:0 (sprite 5 is not in multicolor mode)

SPRCOLOR

Category: Command

Syntax: `SPRCOLOR A,B`

Function: This command sets the multicolor values for all sprites. The parameter A=multicolor value 1, B=multicolor value 2.

The value can be later read with RSPCOLOR(1) and RSPCOLOR(2) (cf.).

Example:

```
10   SPRITE 1,1,,,,,1
20   SPRCOLOR 3,1
30   MOVSPR 1,145#8
```

(NOTE: Be sure to define a multicolor sprite with SPRDEF before running this program; if you do not, the sprite will not

appear as multicolor.) This program switches on sprite #1 and multicolor mode for sprite 1. Line 20 defines multicolor mode 1 as red, and multicolor mode 2 as black. Line 30 sets the sprite moving at a speed of 8 and at a 145-degree angle.

SPRITE

Category: Statement

Syntax: `SPRITE nr,a,c,p,x,y,m`

Function: This function lets you define sprite parameters. Here are the parameters:

nr	Sprite number
a	on (=1) or off (=0)
c	Color (1-16)
p	Foreground (=0) or background (=1) priority
x	X-direction expansion (=1, normal -- x=0)
y	Y-direction expansion (=1, normal -- y=0)
m	Multicolor mode on (m=1) / off (m=0)

Example:

```
110  SPRITE 1,1,1,0,1,1,0
120  SPRITE 2,1,3,1
130  SPRITE 4,0
```

Sprites 1 and 2 are switched on, 4 off, and the first two sprites are defined.

SPRSAV

Category: Statement

Syntax: `SPRSAV A$,1`

Function: Saves a string variable into sprite memory. This command is used mainly to save the graphic information produced by

SSHAPE. You can save this information to sprite memory, and treat the shape as a sprite. Alternately, SPRSAV 1,A$ lets you save sprite information into A$, and treat the sprite as a shape.

Examples: SPRSAV X$,1 saves the contents of X$ in sprite 1 memory. SPRSAV 1,A$ saves sprite 1 in A$.

SSHAPE/GSHAPE

Category: Statement

Syntax: SSHAPE A$,*x1*,*y1*,*x2*,*y2*

Function: SSHAPE turns a graphic section into a string variable. This variable can be no larger than 255 characters; if that limit is overstepped, a STRING TOO LONG ERROR occurs. GSHAPE can put the graphic information on any spot onscreen.

Examples: SSHAPE SP$,10,10,30,30 sets up the graphic section specified in SP$. GSHAPE SP$,100,120 puts SP$ onscreen at coordinates 100/120.

```
50   GRAPHIC1,1:FOR I=0 TO 320 STEP 2:
          GSHAPE SP$,I,100:NEXT
```

This line moves SP$ around the screen.

SPRDEF

Category: Command

Syntax: SPRDEF

Function: SPRDEF switches on the sprite editor, giving the user a simple method of sprite generation. The sprite is designed within a 21 X 24 matrix. The sprite data is set aside for later saving to diskette (see BSAVE) or inclusion in a program.

Functions:

<CLR>	Clears entire editor
M	Multicolor on/off (toggle)
<CTRL> 1-8	Colors 1-8
C= 1-8	Colors 9-16
1	Clear point
2	Set a point
3	Same as 1, but for multicolor
4	Same as 2, but for multicolor
A	Automatic cursor control on/off
X	X-expansion
Y	Y-expansion
C	Copy sprite (e.g., sprite 1 into sprite 2)

Cursor is controlled by the CRSR keys; <SHIFT> <RETURN> stores the information in the corresponding sprite memory.

Graphic commands

BOX

Category: Statement

Syntax: `BOX (cs,x1,y1,x2,y2,a,p)`

Function: Draws a box at any point onscreen.

cs	Color source (0-3)
x1,y1	Upper left-hand corner
x2,y2	Lower right-hand corner
a	Angle of rotation
p	Paint box (1=Y/0=N)

Example:

```
10  SCNCLR
20  GRAPHIC1:Z=5
```

```
30  FOR I=0 TO 361 STEP 5
40  Z=Z+1
50  BOX 1, 160+Z,100+Z,160-Z,100-Z,I
60  NEXT
70  END
```

CHAR

Category: Statement

Syntax: CHAR (*cs,x,y,tx,vd*)

Function: CHAR lets you put text strings on graphic screens at the row and column stated.

cs	Color source (0-3)
x	Column number
y	Row number
tx	Text to be printed
vd	0=normal video/ 1=reverse video

Example:

```
10  CHAR 1,18,12,"TEST OUTPUT"
20  SLEEP 1
30  CHAR 1,18,12,"TEST OUTPUT",1
40  SLEEP 1
50  GOTO 10
```

CIRCLE

Category: Statement

Syntax: CIRCLE (*cs,x,y,xr,yr,sa,ea,r,i*)

Function: The CIRCLE command is one of the most versatile in BASIC 7.0. You can draw circles, ovals, ellipses, etc., anywhere onscreen.

cs	Color source (0-3)
x,y	Midpoint coordinates
xr	Radius in X-direction
yr	Radius in Y-direction
sa	Starting angle of arc
ea	Ending angle of arc
r	Angle of rotation
i	Degrees between circle segments

Example:

```
10   SCNCLR:GRAPHIC1:Z=5
20   FOR I=1 TO 361 STEP 5:Z=Z+1
30   CIRCLE ,160+Z,100+Z,160-Z,100-Z,I
40   NEXT
```

COLOR

Category: Statement

Syntax: `COLOR` *cs,cn*

Function: `COLOR` sets up one of the seven color sources (*cs*)with the value *cn*. *cn* can be a value from 1 to 16.

Examples: `COLOR 0,1` turns the 40-column screen to black.
`COLOR 6,8` turns the 80-column screen to yellow.
`COLOR 1,2` turns the foreground white in graphic mode.

fs can be one of the following values:

0 40-col. background
1 foreground (graphic mode)
2 multicolor 1 (graphic mode)
3 multicolor 2 (graphic mode)
4 40-col. border
5 Character color (40 or 80 cols.)
6 background (80 cols.)

DRAW

Category: Statement

Syntax: `DRAW` *cs,x1,y1* `TO` *x2,y2* `TO` *x3,y3 ...*

Function: `DRAW` allows you to draw any figure by stating starting and ending points on each line. *cs* is the color source, and *x* and *y* are the starting and ending points.

Examples: `DRAW 1,10,20 TO 300,20 TO 150,190 TO 10,20`
Draws a triangle.

```
10   SCNCLR:GRAPHIC1
20   X=160:Y=100:R=50
30   FOR A=1 TO 360
40   DRAW 1,X+R*COS(A),Y+R*SIN(A)
50   NEXT A
60   END
```

This routine draws a circle without the `CIRCLE` command.

GRAPHIC

Category: Statement

Syntax: `GRAPHIC` *m,c,s*

Function: `GRAPHIC` allows the choice of one of six graphic modes. The second parameter *c* determines whether an automatic `SCNCLR` will take place (if *c*=1, yes; *c*=0, no). The parameter *s* determines the line at which text begins in combined text/graphic modes (`GRAPHIC 2` or `4`). The default for *s* is 19.

0 40-col. text mode
1 Graphic mode
2 Combined graphic/text mode
3 Multicolor graphic mode
4 Combined multicolor graphic/text mode
5 80-col. text mode

Example: `GRAPHIC 2,1,15` turns on graphic/text mode and clears the screen. The first line of text appears at line 15.

LOCATE

Category: Statement

Syntax: `LOCATE` *x,y*

Function: This command moves the invisible graphic cursor to any point on the graphic screen.

Example: `LOCATE 160,100` puts the graphic cursor dead center on the screen. These locations can be read with the help of the `RDOT` function. Type in `PRINT RDOT(0),RDOT(1):` You'll get the locations 160,100.

PAINT

Category: Statement

Syntax: `PAINT` *cs,x,y,m*

Function: `PAINT` fills in an area with a color. *x* and *y* are the coordinates at which `PAINT`ing should begin. If the area is not completely enclosed, the whole screen will be painted in.

The parameter *m* determines from which color source the color can be drawn (foreground or background).

Example:

```
10 SCNCLR:GRAPHIC 1
20 BOX ,40,40,100,100
30 PAINT,50,50
```

A square is drawn then colored in. If you'd drawn a three-quarter circle with `CIRCLE ,160,100,40,,0,270` in line 20 instead, then called up `PAINT,160,100`, the entire screen would have been colored in.

SCALE

Category: Statement

Syntax: `SCALE n`

Function: SCALE *n* changes between scales in grahics mode. SCALE 1 changes the graphic screen to horizontal and vertical coordinates of 0-1023. Don't get your hopes up—you can't get a resolution of 1024 x 1024. It's more like 320 x 200, but now a horizontal point is running at 3.2, and a vertical point at 5.12.

There are quite a few programs out there using graphics scaled for 1024 X 1024 screens. You can actually use these on the 128, with some adaptation. SCALE 0 will put us back to our normal 319 (159) X 199 resolution.

Example:

```
10  GRAPHIC 1,1
20  SCALE 0
30  CIRCLE ,160,100,60
40  SCALE 1
50  CIRCLE ,160,100,60
```

SCNCLR

Category: Statement

Syntax: `SCNCLR n`

Function: This command clears the screen. The parameter *n* adapts SCNCLR to the proper graphic page (*n*=0 for 40-char. text mode, *n*=1 for graphic mode, etc.).

WIDTH

Category: Command

Syntax: `WIDTH n`

Function: WIDTH determines the character width of graphics. WIDTH 1 indicates normal width; WIDTH 2 indicates double width.

Example:

```
10  SCNCLR:GRAPHIC1
20  BOX,20,20,60,60
30  WIDTH 2
40  BOX,60,60,110,110
50  END
```

Sound commands

ENVELOPE

Category: Statement

Syntax: `ENVELOPE nr,a,d,s,r,w,p`

Function: This command calls up the predefined envelopes in the C-128. You can change the envelope parameters if you wish. You have a total of 10 envelopes available (0-9) which already imitate musical instruments. Changing these parameters will give you strange new sounds -- or better versions of the old ones.

The parameters:

nr	Envelope number (0-9)
a	Attack (0-15)
d	Decay (0-15)
s	Sustain (0-15)
r	Release (0-15)
w	Waveform 0 = Triangle wave 1 = Sawtooth wave 2 = Pulse wave 3 = Noise 4 = Ring modulation
p	Pulse width for w2 (0-4096)

FILTER

Category: Statement

Syntax: `FILTER` *fr,lp,bp,hp,re*

Function: You can switch the various SID filters on and off (high-pass, band-pass and low-pass), thus influencing and refining tones.

Parameters:

fr	Frequency limit for practical filtering
lp	Low-pass filter (0=off/1=on)
bp	Band-pass filter (0=off/1=on)
hp	High-pass filter (0=off/1=on)
re	Resonance (0-15)

Resonance declares how strident or mellow a tone is to be.

PLAY

Category: Command

Syntax: `PLAY "Vc O En Vl Fl notes"`

Function: PLAY supplies you with a simple method of playing tunes and harmonies—it's almost as simple as using a PRINT command! The strings above set up necessary controls for the SID, and abbreviations for notes. You are limited to 255 characters per string output.

Vc	Voice (n=1-3)
O	Octave (n=0-6)
En	Envelope (n=0-9)
Vl	Volume (n=0-15)
Fl	Filter (0=off/1=on)

notes C,D,E,F,G,A,B

Notes can be adjusted for length, tempo and accidentals:

#=half-tone sharp $=half-tone flat
W=whole note H=half note
Q=quarter note I=eighth note S=sixteenth note
.=dotted note value.
R=rest
M=wait until all voices have finished.

SOUND

Category: Command

Syntax: `SOUND v,f,d,dir,m,sc,w,p`

Function: This is a quick method of playing music or just making noise. You don't have all the neat extras here that you did in PLAY (i.e., ENVELOPE, FILTER, etc.), but this command will suffice for simple sounds.

Parameters:

v	Voice (1-3)
f	Frequency (0-65535)
d	Duration in 1/60 second (0-32767)
dir	Direction of sound (0=up, 1=down, 2=oscillate)
m	Minimum frequency (for r -- 0-65535)
sc	Step value for r (0-32676)
w	Waveform (0-3)
p	Pulse width

TEMPO

Category: Statement

Syntax: `TEMPO x`

Function: `TEMPO` dictates the speed at which the notes are to be played in `PLAY`. *x* can be a number between 0 and 255. The formula

`D=19.22/x` seconds

gives the duration; the larger that *x* is, the faster the notes. The default value for *x* is 8.

VOL

Category: Statement

Syntax: `VOL x`

Function: The statement `VOL` controls the amount of volume at which notes are to be played. *x* can be between 0 and 15, with 15 being the loudest level.

Appendix E

RESERVED WORDS IN BASIC 7.0

A

ABS, AND, APPEND, ASC, ATN, AUTO

B

BACKUP, BANK, BEGIN, BEND, BLOAD, BOOT, BOX,
BSAVE, BUMP

C

CATALOG, CHAR, CHR$, CIRCLE, CLOSE, CLR, CMD,
COLLECT, COLLISION, COLOR, CONCAT, CONT, COPY

D

DATA, DCLEAR, DCLOSE, DEC, DEF, DELETE, DIM,
DIRECTORY, DLOAD, DO, DOPEN, DRAW, DS, DSAVE, DS$,
DVERIFY

E

EL, ELSE, END, ENVELOPE, ER, ERR$, EXIT, EXP

F

FAST, FETCH, FILTER, FN, FOR, FRE

G

GET, GETKEY, GO64, GOSUB, GOTO, GRAPHIC, GSHAPE

H

HEADER, HELP, HEX$

I

IF, INSTR, INPUT, INPUT#, INT

J

JOY

K

KEY

L

LEFT$, LEN, LET, LIST, LOAD, LOCATE, LOG, LOOP

M

MID$, MONITOR, MOVSPR

N

NEW, NEXT, NOT

O

OFF, ON, OPEN, OR

P

PAINT, PEEK, PEN, PLAY, POINTER, POKE, POS, POT, PRINT, PRINT#, PUDEF

Q

QUIT

R

RCLR, RDOT, READ, RECORD, REM, RENAME, RENUMBER, RESTORE, RESUME, RETURN, RGR, RIGHT$, RND, RREG, RSPCOLOR, RSPPOS, RSPRITE, RUN, RWINDOW

S

SAVE, SCALE, SCNCLR, SCRATCH, SGN, SIN, SLEEP, SLOW, SOUND, SPC, SPCOLOR, SPRDEF, SPRITE, SPRSAV, SQR, SSHAPE, ST, STASH, STEP, STOP, SWAP, SYS

T

TAB, TAN, TEMPO, THEN, TI, TI$, TO, TRAP, TRON, TROFF

U

UNTIL, USING, USR

V

VAL, VERIFY, VOL

W

WAIT, WHILE, WIDTH, WINDOW

X

XOR

Index

ABS 217
algorithm 3-5, 31
Analog Clock program 189
AND 21, 23,217
APPEND 263
arrays 117–135
ASC 53- 54, 207, 217, 247
ASCII 14
ATN 48, 218
AUTO 65, 238

BACKUP 264
BANK 109, 238-239
BEGIN...BEND 76-77, 239
binary system 15-17
BLOAD 264
boolean operators 21
BOOT 240
BOX 279-280
BSAVE 265
BUMP 273

CATALOG 173, 265
CHAR 157-158, 280
CHR$ 53- 54, 158-162, 218
CIRCLE 188-190, 280-281
CLOSE 218
CLR 218
CMD 219
COLLECT 266
COLLISION 273-274
COLOR 281
CONCAT 266
CONT 219
COPY 267
COS 47-48, 189, 219, 282

DATA 110-113, 220
DCLEAR 267
DCLOSE 268
DEC 241
DEF FN 51, 220-221
DELETE 66, 241
DIM 119-120, 220
DIRECTORY 173, 268
DLOAD 174-175, 269
documentation 13-14, 33
DO...LOOP 83-84, 242
DO..WHILE 83, 242
DO..UNTIL 83-84, 242
DOPEN 175, 178, 269
DRAW 282
DSAVE 175, 269–270
DVERIFY 270

EL 97, 98, 243, 260
END 38-39, 221
ENVELOPE 185-186, 285-286
ER 59, 96-98, 243, 260
ERR$(X) 98, 243
EXP 221

FAST 243
FETCH 244
FILTER 186, 286
flowcharts 5-13, 32, 71, 72, 81-89
FN 44, 51, 220, 221
FOR...NEXT 78-82, 221
FRE 107, 222
function keys 104-106, 248-249

GET 100-106,152-157, 222
GETKEY 102-104, 244-245
GOSUB 136-148, 223
GOTO 71-74, 223, 239, 251, 280
GSHAPE 278
GRAPHIC 187-189, 282
HEADER 173, 270-271
HELP 66,245
HEX$ 246
IF...THEN 73-75, 88-89, 223
IF...THEN...ELSE 73-75, 246-247
INPUT 34, 224
INPUT# 224
INSTR 63, 247
INT 224

JOY 248

KEY 248-249, 260

LEFT$ 58, 225, 248
LEN(X$) 61, 225
LET 35-36, 265
LIST 66, 225-226
LOCATE 283
logical operators 21-26

Math Tutor 90-96, 143-145
menus 149-167
MID$ 59-60, 227, 249
MONITOR 193-194, 249
MOVSPR 274

NAND 21
NOR 21
NOT 21, 22, 228

ON...GOSUB 147-148, 228
ON...GOTO 89-90, 97-99, 228
OPEN 228
OR 21, 24-25, 229

PAINT 283
PEEK 108, 229
PEN 250
PLAY 184-185,287
POINTER 195, 250
POKE 109, 229
POS 107, 229
POT 251
PRINT 36-39, 230
PRINT# 230
PRINT USING 39-41, 251-252
PUDEF 42, 253

random numbers 51-53, 239
RCLR 253-254
RDOT 254
READ 110-113, 230
RECORD 178-179, 271
relative files 178-179
REM 43, 231
RENAME 271-272
RENUMBER 65, 255
RESTORE 110-113, 231
RESUME 97-99, 255–256, 260
RETURN 136-148, 231
RGR 256
RIGHT$ 59, 232
RND 51-53, 239
RREG 257
RSPCOLOR 275
RSPPOS 275
RSPRITE 276
RWINDOW 167, 257–258

SCALE 284
SCNCLR 284
SCRATCH 272
screen dump 199
SIN 233
SLEEP 258
sort routines 168-169
SOUND 186, 287-288
SPC 56, 233

SPRCOLOR 276-277
SPRDEF 278-279
SPRITE 277
SPRSAV 277-278
SQR 234
SSHAPE 278
STASH 258
STEP 234
STR$(X) 62-63, 234
subroutines 136-148
SWAP 259
SYS 107, 259

TAB 56, 235
TAN 235
TEMPO 186, 288
TI$ 63-64
TRAP 97-99, 260
TROFF 261
TRON 260

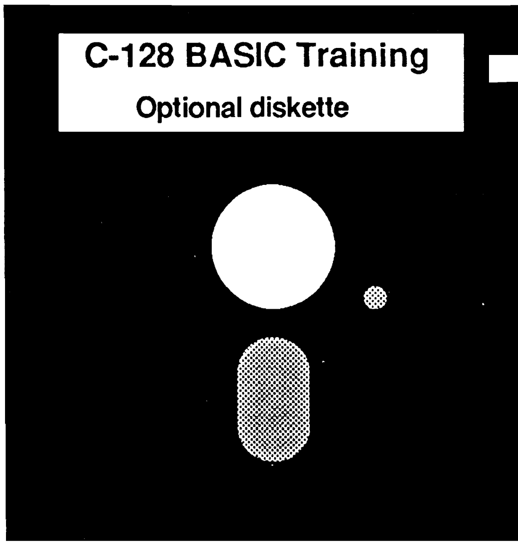
UNTIL 83-88, 242-243
USR 107, 236

VAL(X$) 62, 236
variables 44-45, 47-50, 53-54
VOL 186, 288

WAIT 107, 237
WHILE 83, 241-244, 260
WINDOW 165-166, 261

XOR 21, 25-26, 262

Optional Diskette



For your convenience, the program listings contained in this book are available on a 1541 formatted floppy disk. You should order the diskette if you want to use the programs, but don't want to type them in from the listings in the book.

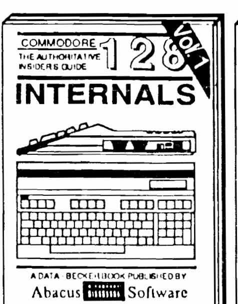
All programs on the diskette have been fully tested. You can change the programs for your particular needs. The diskette is available for $14.95 plus $2.00 ($5.00 foreign) for postage and handling.

When ordering, please give your name and shipping address. Enclose a check, money order or credit card information. Mail your order to:

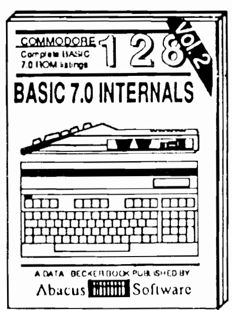
Abacus Software
P.O. Box 7219
Grand Rapids, MI 49510

Or for *fast* service, call **(616) 241-5510.**

C-128™ AUTHORITATIVE and C-64™ BOOKS

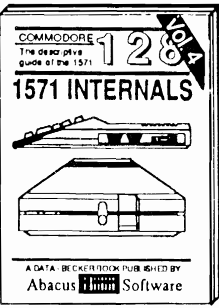


Detailed guide presents the 128's operating system, explains graphic chips, Memory Management Unit, 80 column graphics and commented ROM listings. 500pp $19.95

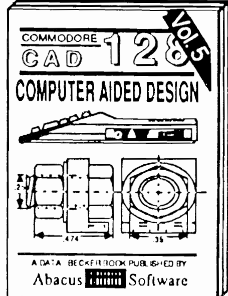


Get all the inside information on BASIC 7.0. This exhaustive handbook is complete with commented BASIC 7.0 ROM listings. Coming Summer '86. $19.95

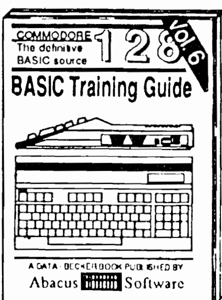
Filled with info for everyone. Covers 80 column hi-res graphics, windowing, memory layout, Kernal routines, sprites, software protection, autostarting 300pp $19.95



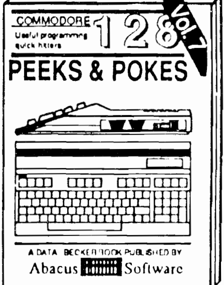
Insiders' guide for novice & advanced users. Covers sequential & relative files, & direct access commands. Describes DOS routines. Commented listings. $19.95



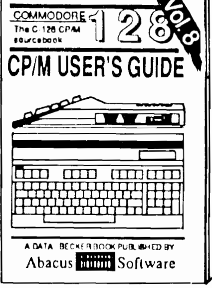
Learn fundamentals of CAD while developing your own system. Design objects on your screen to dump to a printer. Includes listings for '64 with Simon's Basic. 300pp $19.95



Introduction to programing; problem analysis; thorough description of all BASIC commands with hundreds of examples; monitor commands; utilities; much more $16.95



Presents dozens of programming quick-hitters. Easy and useful techniques on the operating system, stacks, zero-page, pointers, the BASIC interpreter and more $16.95



Essential guide for everyone interested in CP/M on the 128. Simple explanation of the operating system, memory usage, CP/M utility programs, submit files & more. $19.95

ANATOMY OF C-64 Insider's guide to the '64 internals. Graphics, sound, I/O, kernal, memory maps, more. Complete commented ROM listings. 300pp $19.95

ANATOMY OF 1541 DRIVE Best handbook on floppy explains all. Many examples and utilities. Fully commented 1541 ROM listings. 500pp $19.95

MACHINE LANGUAGE C-64 Learn 6510 code write fast programs. Many samples and listings for complete assembler, monitor, & simulator. 200pp $14.95

GRAPHICS BOOK C-64 - best reference covers basic and advanced graphics. Sprites, animation, Hires, Multicolor, lightpen, 3D-graphics, IRQ, CAD, projections, curves, more. 350pp $19.95

TRICKS & TIPS FOR C-64 Collection of easy-to-use techniques: advanced graphics, improved data input, enhanced BASIC, CP/M, more. 275pp $19.95

1541 REPAIR & MAINTENANCE Handbook describes the disk drive hardware. Includes schematics and techniques to keep 1541 running. 200pp $19.95

ADVANCED MACHINE LANGUAGE Not covered elsewhere: - video controller, interrupts, timers, clocks, I/O, real time, extended BASIC, more. 210pp $14.95

PRINTER BOOK C-64/VIC-20 Understand Commodore, Epson-compatible printers and 1520 plotter. Packed: utilities; graphics dump; 3D-plot; commented MPS801 ROM listings, more. 330pp $19.95

SCIENCE/ENGINEERING ON C-64 In depth intro to computers in science. Topics: chemistry, physics, biology, astronomy, electronics, others. 350pp $19.95

CASSETTE BOOK C-64/VIC-20 Comprehensive guide; many sample programs. High speed operating system fast file loading and saving. 225pp $14.95

IDEAS FOR USE ON C-64 Themes: auto expenses, calculator, recipe file, stock lists, diet planner, window advertising, others. Includes listings. 200pp $12.95

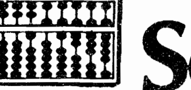
COMPILER BOOK C-64/C-128 All you need to know about compilers: how they work; designing and writing your own; generating machine code. With working example compiler. 300pp $19.95

Adventure Gamewriter's Handbook Step-by-step guide to designing and writing your own adventure games. With automated adventure game generator. 200pp $14.95

PEEKS & POKES FOR THE C-64 Includes in-depth explanations of PEEK, POKE, USR, and other BASIC commands. Learn the "inside" tricks to get the most out of your '64. 200pp $14.95

Optional Diskettes for books For your convenience, the programs contained in each of our books are available on diskette to save you time entering them from your keyboard. Specify name of book when ordering. $14.95 each

C-128 and C-64 are trademarks of Commodore Business Machines Inc.

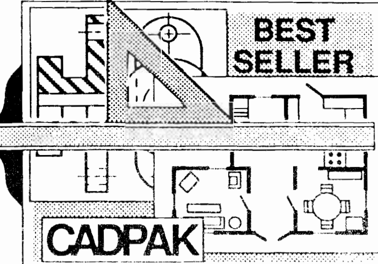
Abacus  **Software**

P.O. Box 7219 Dept. H8 Grand Rapids, MI 49510 - Telex 709-101 - Phone (616) 241-5510

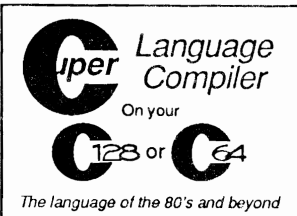
Optional diskettes available for all book titles - $14.95 each. Other books & software also available. Call for the name of your nearest dealer. Or order directly from ABACUS using your MC, Visa or Amex card. Add $4.00 per order for shipping. Foreign orders add $10.00 per book. Call now or write for your **free** catalog. Dealer inquires welcome--over 1400 dealers nationwide.

SUPER SOFTWARE

The *complete* compiler and development package. Speed up your programs 5x to 35x. Many options: flexible memory management; choice of compiling to machine code, compact p-code or both. '128 version: 40 or 80 column monitor output and FAST-mode operation. '128 Compiler's extensive 80-page programmer's guide covers compiler directives and options, two levels of optimization, memory usage, I/O handling, 80 column hi-res graphics, faster, higher precision math functions, speed and space saving tips, more. A great package that no software library should be without. **128 Compiler $59.95**
64 Compiler $39.95

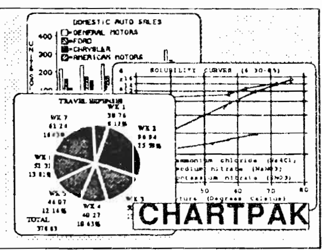


Remarkably easy-to-use interactive drawing package for accurate graphic designs. New dimensioning features to create exact scaled output to all major dot-matrix printers. Enhanced version allows you to input via keyboard or high quality lightpen. Two graphic screens for COPYing from one to the other. DRAW, LINE, BOX, CIRCLE, ARC, ELLIPSE available. FILL objects with preselected PATTERNS; add TEXT; SAVE and RECALL designs to/from disk. Define your own library of symbols/objects with the easy-to-use OBJECT MANAGEMENT SYSTEM–store up to 104 separate objects. **C-128 $59.95**
C-64 $39.95



For school or software development. Learn C on your Commodore with our in-depth tutorial. Compile C programs into *fast* machine language. C-128 version has added features: Unix™-like operating system; 60K RAM disk for fast editing and compiling Linker combines up to 10 modules; Combine M/L and C using CALL; 51K available for object code; Fast loading (8 sec. 1571, 18 sec. 1541); Two standard I/O librarys plus two additional libraries—math functions (sin, cos, sqrt, etc.) & 20+ graphic commands (line, fill, dot, etc.). **C-128 $59.95**
C-64 $59.95

Not just a compiler, but a complete system for developing applications in Pascal with graphics and sound features. Extensive editor with search, replace, auto, renumber, etc. Standard J & W compiler that generates fast machine code. If you want to learn Pascal or to develop software using the best tools available-**SUPER Pascal** is your first choice. **C-64 $59.95**



Easily create professional high quality charts and graphs without programming. You can immediately change the scaling, labeling, axis, bar filling, etc. to suit your needs. Accepts data from CalcResult and MultiPlan. C-128 version has 3X the resolution of the '64 version. Outputs to most printers. **C-128 $39.95**
C-64 $39.95

PowerPlan
One of the most powerful spreadsheets with integrated graphics. Includes menu or keyword selections, online help screens, field protection, windowing, trig functions and more. PowerGraph, the graphics package, is included to create integrated graphs and charts. **C-64 $39.95**

Technical Analysis System for the C-64	$59.95
Ada Compiler for the C-64	$39.95
VideoBasic Language for the C-64	$39.95

OTHER TITLES AVAILABLE:

COBOL Compiler
Now you can learn COBOL, the most widely used commercial programing language, and learn COBOL on your 64. COBOL is easy to learn because it's easy to read. COBOL Compiler package comes complete with Editor, Compiler, Interpreter and Symbolic Debugger. **C-64 $39.95**

BEST SELLER

Personal Portfolio Manager
Complete portfolio management system for the individual or professional investor. Easily manage your portfolios, obtain up-to-the-minute quotes and news, and perform selected analysis. Enter quotes manually or automatically through Warner Computer Systems. **C-64 $39.95**

Xper
XPER is the first "*expert system*" for the C-128 and C-64. While ordinary data base systems are good for reproducing facts, XPER can derive knowledge from a mountain of facts and help you make expert decisions. Large capacity. Complete with editing and reporting. **C-64 $59.95**

C-128 and C-64 are trademarks of Commodore Business Machines Inc
Unix is a trademark of Bell Laboratories

Abacus Software

P.O. Box 7219 Dept.M9 Grand Rapids, MI 49510 - Telex 709-101 - Phone (616) 241-5510

Call **now** for the name of your nearest dealer. Or to order directly by credit card, MC, AMEX of VISA call **(616) 241-5510**. Other software and books are available–Call and ask for your **free** catalog. Add $4.00 for shipping per order. Foreign orders add $12.00 per item. Dealer inquires welcome–1400+ nationwide.

COMMODORE® 128™ BASIC TRAINING GUIDE

128 BASIC Training is a thorough guide to Commodore BASIC for the C-128. You'll learn programming quickly and easily with this book. There are many examples that demonstrate the power of BASIC. For the seasoned programmer, there is a complete list of commands in both BASIC 2.0 and BASIC 7.0. Topics include:

- Program and data flowcharts
- Advanced programming
- Menus and how to create them
- Graphics and sound
- Multi-dimensional arrays
- Programming windows
- Utilities
- BASIC internals
- Loops
- Structured BASIC
- Number systems
- Disk commands

About the author:
Frank Kampow is employed by West Germany's DATA BECKER Software Division as a programmer. He is highly qualified to write this book on 128 BASIC, since he has spent many years lecturing on computer science and has long-standing practical experience as a programmer.

ISBN 0-916439-64-X

Commodore 128™ is a trademark of Commodore Electronics Ltd.

Part of the continuing series of informative books from