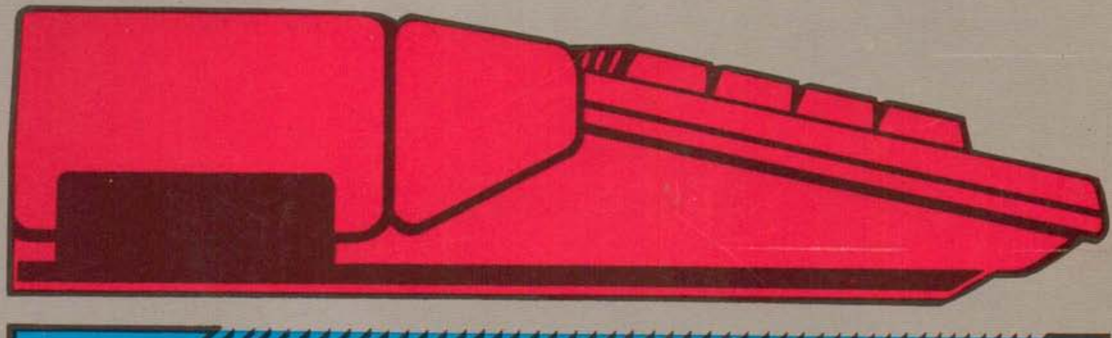
ATARI® ST™

BASIC Training Guide

Everyone's introduction to ST BASIC



BASIC TRAINING GUIDE

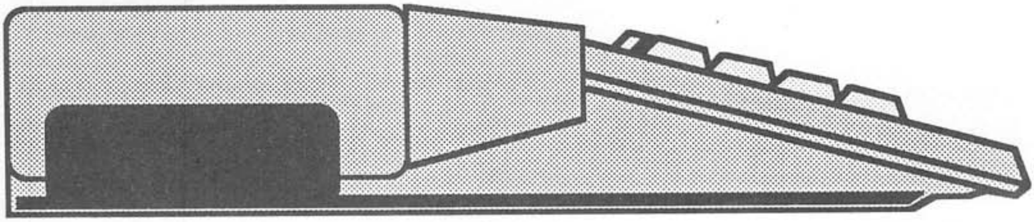
A Data Becker book published by

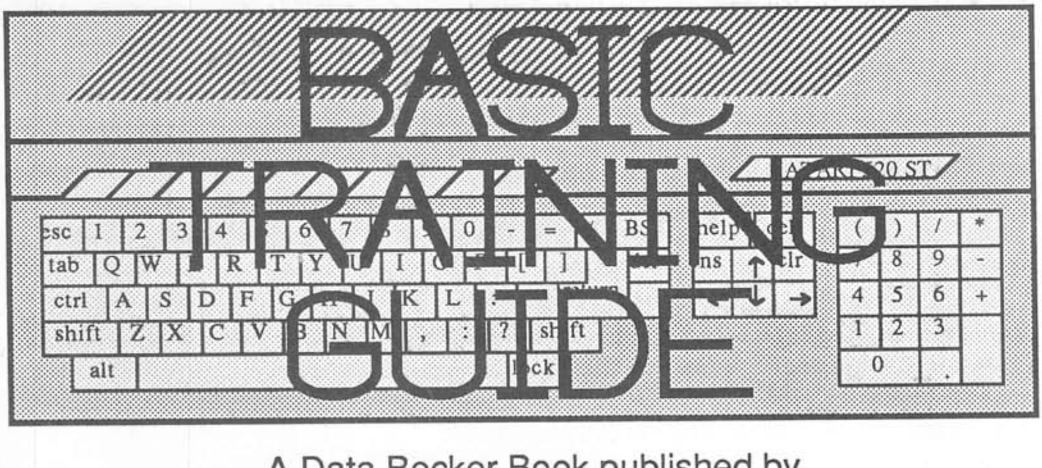
You Can Count On Abacus Software

ATARI® ST™

BASIC Training Guide

F. Kampow
and
N. Szczepanowski





A Data Becker Book published by

Abacus Software

Third Printing, June 1988
Printed in U.S.A.
Copyright ©1985 Data Becker GmbH
Merowingerstr. 30
4000 Dusseldorf, West Germany

Copyright ©1986,1987 Abacus Software, Inc.
P.O. Box 7219
Grand Rapids, MI 49510

ISBN 0-916439-57-7

Table of Contents

Chapter 2 (cont)

Chapter 4 (cont)

Foreword

The Atari ST is quite a capable computer. The purpose of this book is to make you a capable BASIC programmer. We intend to present the fundamentals of BASIC—from simple screen output with `PRINT` to complex program algorithms such as sorting. In addition, this book points out the idiosyncracies in the current version of BASIC.

Chapter 1 gives you the fundamentals of programming, such as good programming style and program documentation. In addition, you will learn the theoretical and practical foundations of data processing.

In Chapter 2 and 3 you begin your actual programming. First you will learn how various BASIC commands are used and applied by means of numerous examples. The example programs are also fairly transportable to other computers, having the same or a similar BASIC command set. This is why the programs do not use an excessive number of PEEK and POKE commands.

At the conclusion of the individual sections you will find problems for you to solve. This lets you check your comprehension of the preceding material. The problem solutions are found in Appendix C. We suggest you work through the solution to all the problems before proceeding, because the problem solution contains a transition to the following section.

Chapter 4 consists of more complex problems and hence more complex programs. Again, the chapter contains many examples and problems—you won't be just reading, but will get actual hands-on experience working with BASIC on your ST.

Chapter 5 introduces you to the principles of file management and operation of the disk drive.

Chapter 6 explains some of the sound and graphics commands of the Atari ST, and includes many examples.

And in Chapter 7, you'll learn how to use some powerful GEM functions within BASIC programs, while example programs illustrate GEM's versatility.

Appendix A is a convenient alphabetical list of all ST BASIC commands and instructions, with explanations and examples of their use. Appendix C contains the solutions to all of the sample problems in the book.

This book is based on the initial version of ST BASIC. Deviations from any subsequent versions are possible.

We'd like to wish you lots of fun and success while working with this book and with your ST.

Sincerely,

Frank Kampow
Norbert Szczepanowski

Chapter 1

FUNDAMENTALS OF PROGRAMMING

1.1 Loading ST BASIC
1.2 Algorithms and programs
1.3 The BASIC language
1.4 Data flowcharts, program flowcharts, and documentation
1.5 ASCII Codes
1.6 Number systems
1.7 Logical operators

Fundamentals of programming

1.1 Loading ST BASIC

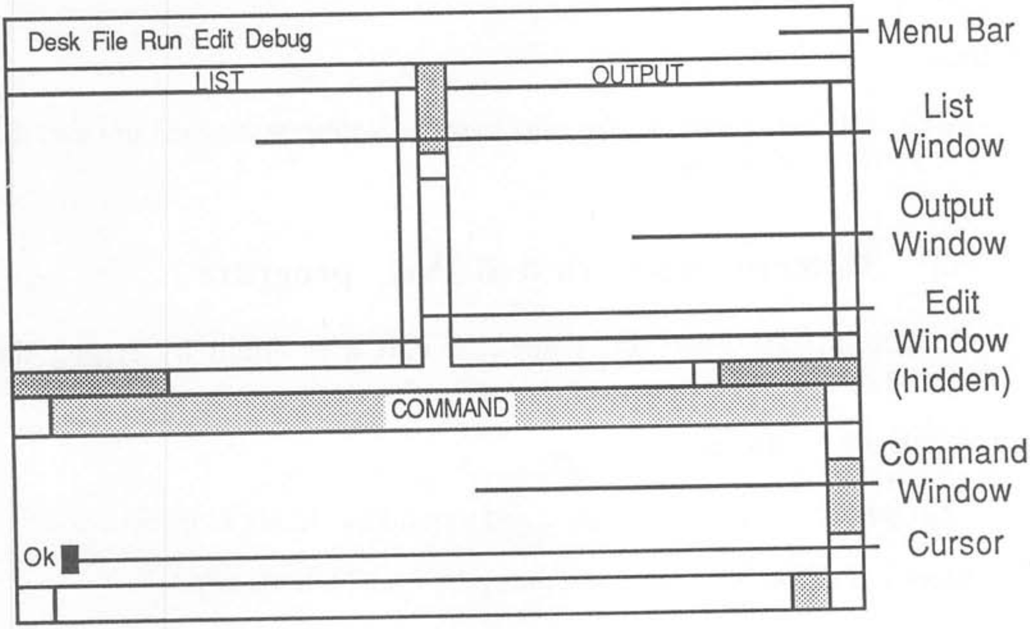
Before using ST BASIC, you must first load it into memory. Here's how to do it:

- With GEM Desktop displayed upon the screen, insert your ST BASIC disk into drive A.
- Double-click on the picture of `FLOPPY DISK A`. This opens up a window displaying the disk contents.
- Now double-click on the program `BASIC.PRG`. This loads BASIC into memory.

ST BASIC is now ready to use.

1.1.1 Getting familiar with BASIC

The following diagram shows four of the windows used by BASIC on the ST. On the next page we'll explain them in more detail.



Menu Bar

You point with the mouse to the various words (Desk, File, etc.) and a menu for that selection appears.

List Window

This is where your program is displayed when you type the command `LIST`. It will show the current program that is in memory.

Output Window

All input from a program and all screen output appears in this window.

Edit Window

All program editing is done within this window. To enter this window, type `EDIT` and your current program will appear in this window. From there you can edit your program. While in the Edit window, point to Edit on the Menu Bar and the various editing options will appear.

Command Window

This is the window where you type in your commands or enter your BASIC program.

Cursor

This little box points to the next location where text typed in from the keyboard will appear.

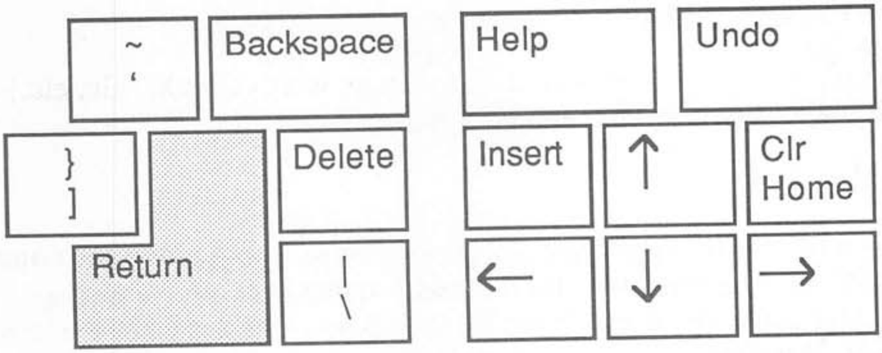
1.1.2 Entering your first BASIC program

We'll demonstrate how you enter and edit a program by typing the following example.

First type the following:

```
20 print "hi, i'm your friendly atari computer"
```

To enter it into the ST's memory, press the <RETURN> key.



The shaded key represents the <RETURN> key in the diagram above.

Now, to see if our little program works, type:

```
RUN
```

and press the <RETURN> key. After you type a command or a program line you must press the <RETURN> key to tell the computer to interpret it. In the Output window, the ST displays:

```
hi, i'm your friendly atari computer
```

The command RUN tells the ST to execute whatever program is in memory. Let's explain the program that you just typed in.

```
20 print "hi, i'm your friendly atari computer"
```

line number

To enter a program into memory, the program must have a number in front of it (20 in our example). This number is appropriately called a *line number*. The purpose of line numbers is to put the program lines into a desired order. The ST stores the program in memory in ascending sequence according to the line number.

As you enter program lines into the ST, BASIC places that line into correct sequence as determined by the line number. Therefore, you can enter program lines in any particular order and BASIC will put them into proper order in memory. Type the following:

```
10 print "this is the first line"
30 print "this is the third line"
```

To see the entire program in memory type:

```
LIST
```

Notice how the line numbers are displayed in sequence. `LIST` displays your current program in the List window. To see the entire program, click the pointer somewhere inside the List window—this activates the window. Click the pointer in the little box in the upper right corner of the List window. This action expands the List window to the size of the full screen (this also works in the other windows as well). To return the window to its normal size, click the pointer in the same upper right box.

To return to and activate the Command window so that you can enter more lines, click the pointer in the Command window.

If you enter a program line with a duplicate line number, the new program line replaces the one in memory. To demonstrate, type the following:

```
20 print "hi, i'm replacing the second line"
```

Enter the command `LIST` and notice that what you just typed replaced the second line.

If you enter just a line number followed by <RETURN>, the program line with that line number is removed or deleted from memory. Type the following to see this occur:

```
20
```

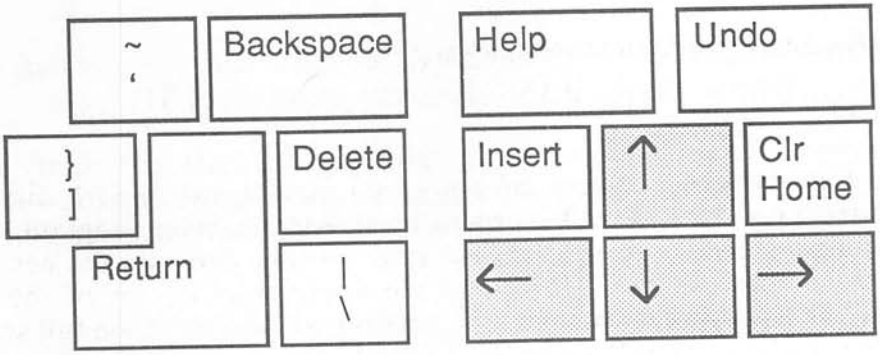
Notice that only lines 10 and 30 remain.

To change a line without having to type the whole thing over again, you are able to edit it. Enter the following command:

```
EDIT
```

Remember to press the <RETURN> key after entering a command. The Edit window is now activated. From here we can edit our program.

To edit our program we will need to use the cursor keys.



The shaded keys represent the cursor keys. The cursor is moved in the direction indicated by the arrow on the key. Press the right-cursor key (the key with the arrow pointing to the right) 15 times. The cursor should now be on the `i` in the word `is`. Press the F1 key (the diamond-shaped key above the 1 key) to insert a blank space—F2 deletes the character under the cursor. Notice how the text turned to grey. This means that a change has occurred in that line. Now type:

```
wa
```

Press the <RETURN> key to enter it into the ST's memory. The line turns black to indicate that it was entered. The line should look as follows:

```
10 print "this was the first line"
```

Press the F10 key to exit the Edit window and return to the Command window. Type RUN to execute your edited program.

1.2 Algorithms and programs

This chapter is about the fundamentals of programming. In particular, we're going to learn about the the BASIC language on the Atari ST.

But before we begin our BASIC programming, we must first clarify some terminology. We'll be giving you a little theory of programming—it may sound a little dry at first, but 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. While some computers have a programming language built into them, the Atari ST requires you to first load BASIC from diskette. But even then, you can't just type in your request at the keyboard:

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

To solve this problem with the ST, or any computer, you must first define a plan outlining how to solve the problem in a clear and logical 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.*

1.3 The BASIC language

The most widely used programming language is called BASIC. BASIC was developed in 1961 at Dartmouth College in New Hampshire. BASIC is an acronym for:

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

BASIC has its roots in the FORTRAN programming language. Since its inception, many different dialects of BASIC have been developed for different computers. For the ST we refer to the specific version developed for that computer, ST BASIC.

This ST BASIC 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 ST.

As with all interpreted languages, the ST 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 ST 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 a translation of the algorithm into a programming language, in our case ST BASIC.

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 like this:

For each of 20 values do the following:

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

A program is the translation of the algorithm into a programming language. Here's a program to solve our problem:

```
10 PI=3.14159265
20 FOR I=1 TO 20
30 INPUT"RADIUS (IN CM)";R
40 V=4*PI*R^3/3
50 PRINT"THE VOLUME IS ";V;" ccm"
60 NEXT I
```

The program works perfectly and you have your answers. 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 logic 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.

1.4 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.

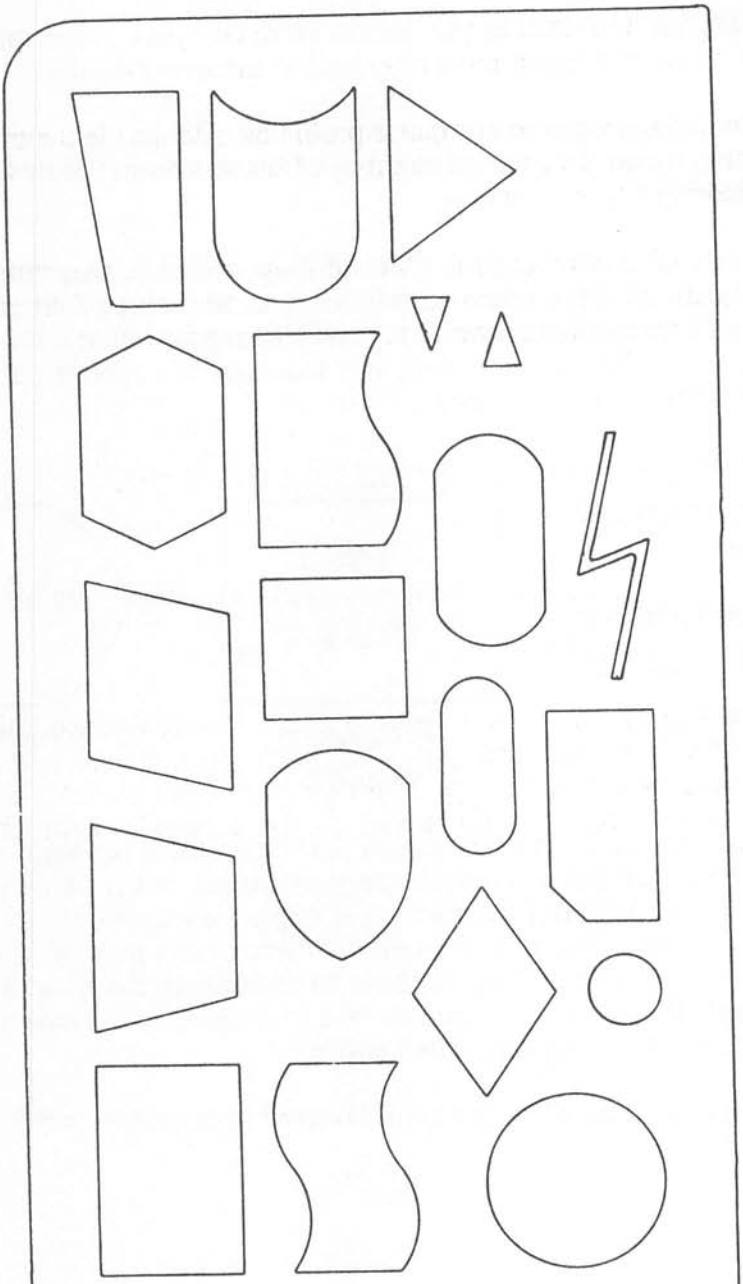


Figure 1: Programming Template

1.4.1 Data flowcharts

A data flowchart is a pictorial representation of the data elements involved in the program. Again, various different symbols represent the flow of the data within a program.

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

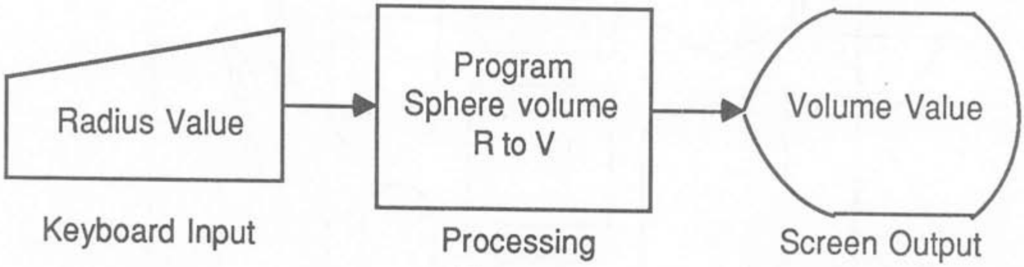


Figure 2

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.

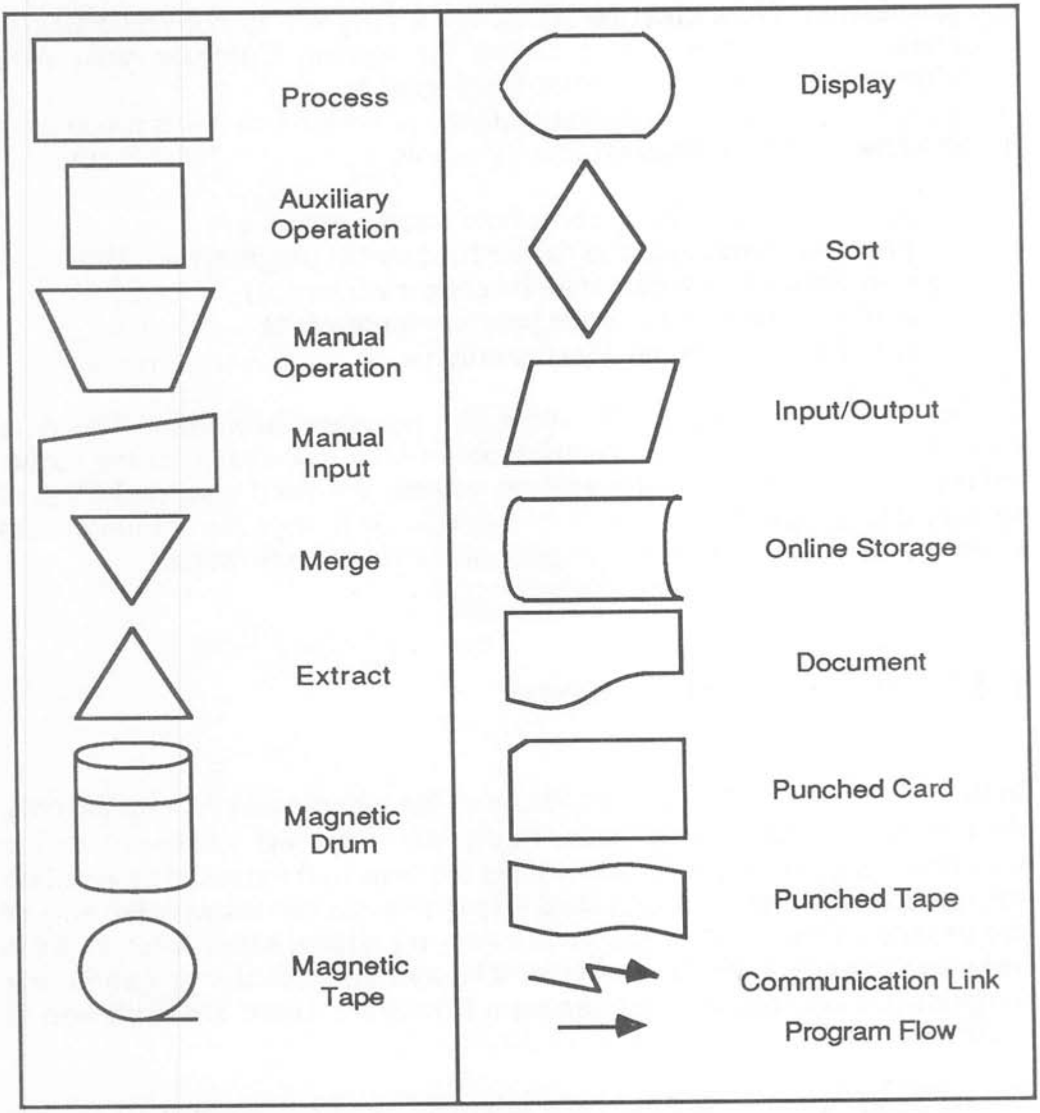


Figure 3: Data Flowchart Symbols

For practice, create a data flowchart for a program to convert miles to kilometers and display 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 that tells us the individual steps the computer uses to solve a problem. This is the purpose of the program flowchart.

1.4.2 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 the program flowchart. It shows in clear, individual steps what to do in order 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 will 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 adage "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.

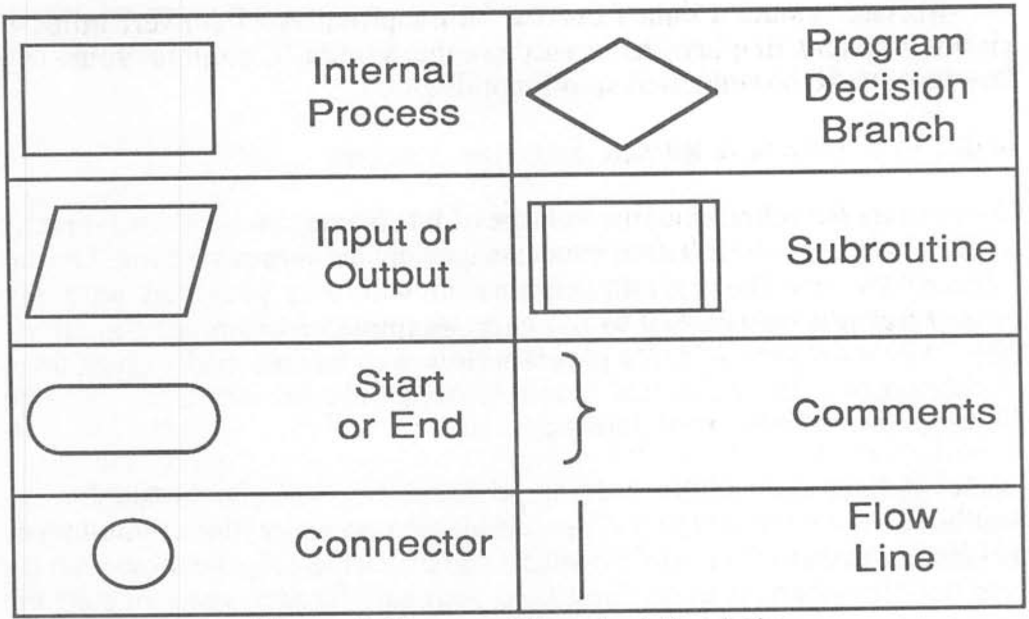


Figure 4: Program Flowchart Symbols

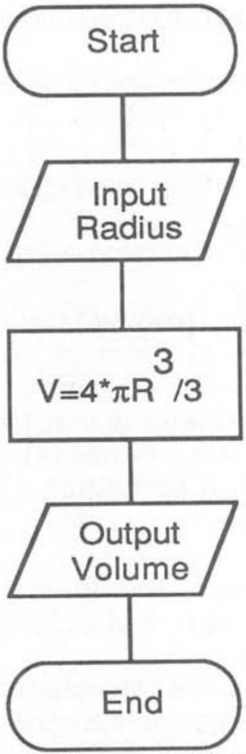


Figure 5: Program Flowchart

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. This can also be provided 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 given 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
```

Compare your result with the suggested solution in Figure 7.

The advantages of program flowcharts will 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. 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 that were put into the program, perhaps a year ago. 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.

1.4.3 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 an example 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 ST BASIC. The formula for spherical volume is from Geometric Encyclopedia, R. Chemedes, 1942."

Summarizing the steps required for good programming:

- Definition of the problem (acquire the problem statement, problem analysis)
- Development of the algorithm for solution (data and program flowcharts)
- Translation the algorithm into a programming language (creating the program)
- Test run of the program
- Documentation

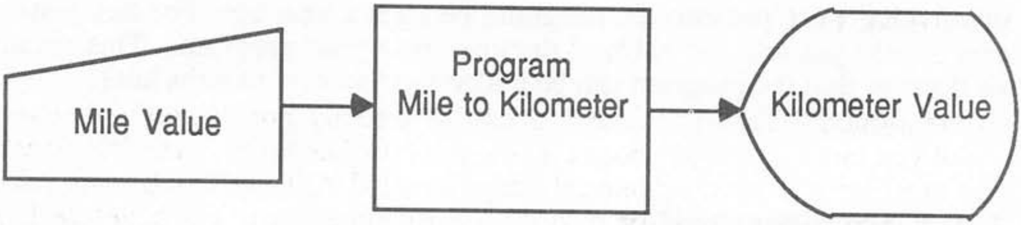


Figure 6: Example solution

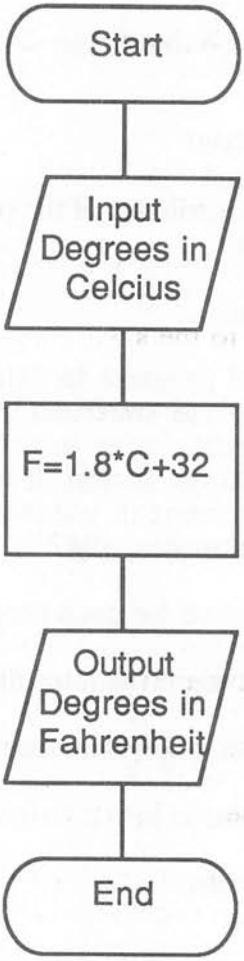


Figure 7: Example solution

1.5 ASCII Codes

As mentioned before, the Atari ST cannot directly process the characters which you enter on the keyboard. These are translated into numerical codes. The most widely used numerical code is called ASCII. ASCII stands for American Standard Code for Information Interchange. 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 works with ASCII, this value is always interpreted as the letter "A". The circumstances of the transfer make no difference. No matter whether you enter characters into the computer with the keyboard or send your data across the country with a telephone modem, as soon as the receiver gets the value 65 it will be translated into an "A". The standard ASCII code uses the values from 0 to 127.

Most computer manufacturers have decided to 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 the 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 ST is identical to the standard ASCII code for the upper- and lowercase letters. For many of the other values, Atari included different useful characters, such as foreign language symbols.

1.6 Number systems

The computer can distinguish 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. In the binary system, numbers are represented using only the digits 0 and 1. The 1 stands for the condition ON and the 0 for the condition OFF. To explain the binary system we will first start with the decimal system.

A decimal number can be converted into a number in any arbitrary number system. We can also 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.

$$(5678_{10})$$

1.6.1 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 the powers of two. If we now want to convert the binary number, we write each digit under its corresponding power of 2. The whole thing is then simply added together and we have our decimal number.

2^3	2^2	2^1	2^0
1	0	1	1

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. This then looks like this:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	0	0	1	1	1

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.

1.6.2 Bit and byte

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 processes is called a *bit* (*bi*nary dig*it*). 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 zeros 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.

1.6.3 The hexadecimal system

In the hexadecimal system the base is the number 16. For this you have 16 (including zero) different "digits." In order to be able to distinguish the digits which are to 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 will practice working with this number system using examples. We will first convert hexadecimal numbers into decimal numbers. The index 16 is used to designate the hexadecimal numbers.

$$2\ E\ 0\ C_{16}$$

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

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

You can see that here the digits 2E0C are assigned specific powers of 16. Here is another example:

$$0\ A\ B\ C_{16}$$

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

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

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 binary digits (bits) 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 case this is:

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

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

Now we have our hexadecimal number:

$$C001_{16} = 49153$$

Here's a partial listing of a conversion table to help you see the relationship between the different number systems:

Decimal	Hexa- decimal	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
	.	
	.	
	.	

1.7 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. The ST BASIC offers you three boolean operations:

NOT, AND, OR

These 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 found in various combinations in integrated circuits (such as NAND, NOR, and XOR gates).

As we 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 through 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 other than zero, 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 has the value zero and only the value zero as the result.

The two logical operators combine two values with each other, in which they are compared bit by bit (remember the binary number?). Now we'll discuss the operators individually.

1.7.1 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`

1.7.2 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
```

In order to understand this result, we take a 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).

1.7.3 OR

The 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, and OR the lowest priority. Concretely, this means that a negation is preformed 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 functions XOR (eXclusive OR), EQV (EQuiValence), and IMP (IMPlication), since they are logical operators as well.

1.7.4 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 every-day language. For example, when one friend says to another: "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 he 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 that the XOR does indeed operate in this manner by entering the following line into your computer:

```
PRINT 0 XOR 0, 0 XOR 1, 1 XOR 0, 1 XOR 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 = X **XOR** Y.

1.7.5 EQV

The logical operator EQV is the negation of the operator XOR. The table for EQV looks like this:

Operator	Value 1	Value 2	Result
EQV	0	0	-1
	0	-1	0
	-1	0	0
	-1	-1	-1

The function EQV returns a true value if the two statements have the same truth value.

1.7.6 IMP

The logical operator results in a true value in three cases and false in only one, so that the table for this operator looks like this:

Operator	Value 1	Value 2	Result
IMP	0	0	-1
	0	-1	-1
	-1	0	0
	-1	-1	-1

On the following page we've placed all of these logical operators in a table, in their order of precedence.

We've said enough for now—it's time for you to practice what you have learned. Solve the problems on page 33. If you are unsure about what to do at any point, refer back to the appropriate section in this chapter.

The solutions are found in Appendix C.

NOT	X		NOT X
	-1		0
	0		-1
AND	X	Y	X AND Y
	0	0	0
	0	-1	0
	-1	0	0
	-1	-1	-1
OR	X	Y	X OR Y
	0	0	0
	0	-1	-1
	-1	0	-1
	-1	-1	-1
XOR	X	Y	X XOR Y
	0	0	0
	0	-1	-1
	-1	0	-1
	-1	-1	0
EQV	X	Y	X EQV Y
	0	0	-1
	0	-1	0
	-1	0	0
	-1	-1	-1
IMP	X	Y	X IMP Y
	0	0	-1
	0	-1	-1
	-1	0	0
	-1	-1	-1

Table of Logical Operators

Problems

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

2.1 The first BASIC program
2.2 Variables and their use
2.3 Numerical functions
2.4 `TAB` and `SPC`
2.5 Strings
2.6 Editing programs
2.7 The screen window

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) Definition of 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

On the following page are the data flowchart Figure 8 and the program flowchart for a solution to the problem. This program flowchart 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.

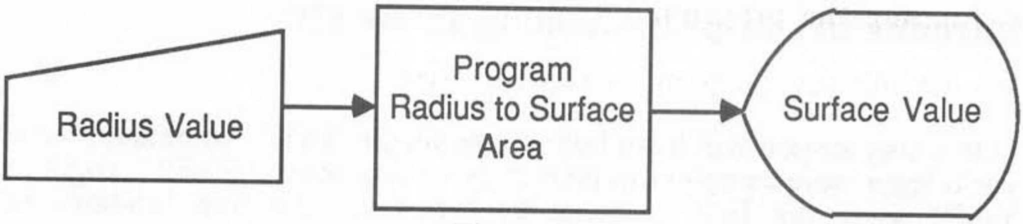


Figure 8
Data flowchart

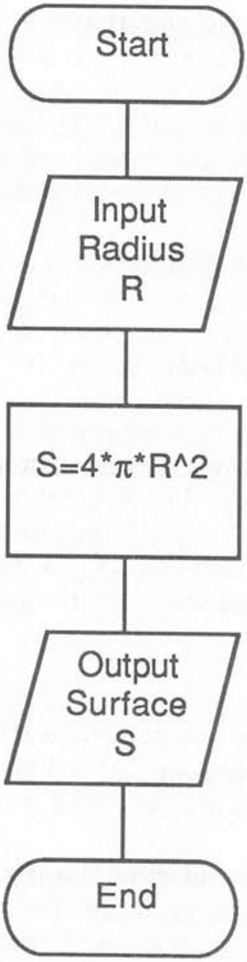


Figure 9
Program flowchart

3) Creating the program

Enter the following program from the keyboard:

```
10 INPUT"ENTER RADIUS IN CM";R
20 LET S=4*3.14159*R^2
30 PRINT S
40 END
```

4) Test run of 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, it will ease the insertion of new lines when required.

You can first enter line 20 into the computer and then line 10 if you like. The computer will sort 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                              Type: 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                Type: 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                          Type: 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) needs to be entered.

Another form of the INPUT command is the command LINE INPUT. It is distinguished from the INPUT command by the fact that if a comma is entered, it is passed to the variable.

This means that only one variable can be assigned a value with LINE INPUT, whereas a list of variables can follow INPUT. Furthermore, LINE INPUT can be used only with string variables.

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 in ST BASIC it is optional. The following examples should clarify this:

a) `LET A=10` or `A=10`

Assigns the value 10 to the variable.

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

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

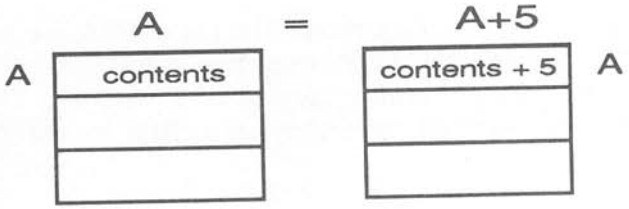
c) `LET A=A*B-8` or `A=A*B-8`

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. Only one variable may be on the left of the = sign.

Let's take another look at example b) above. Mathematically the expression is incorrect. If the value `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 which the beginner uses in programming. But at the same time it's one of the most versatile commands of ST BASIC. 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's a few examples of the use of the PRINT command. The variables to be used 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 20
```

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 in the command window. They are immediately executed after you press the <RETURN> key. If you enter:

```
PRINT A <RETURN>
```

then the value of A is immediately displayed in the output window.

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 ascending order according to the statement line number.

If you enter:

```
10 PRINT A
```

BASIC will store 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 in the output window since we previously set `A=10`.

When printing 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 will appear at the beginning of the following line.

To explain the terms carriage return and linefeed, we'll use the typewriter as an example.

Imagine that you have typed the number 10 on the paper. 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 sort of thing when it performs a carriage return and linefeed, except that you don't have to move a lever and there is 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 characters such as the <ESC> or <TAB> key. If you use these keys within the quotation marks, their symbols appear in the input window as ${}^{E}{}_{S}$ and ◷. However, this character is not visible in the output window 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 evaluated and then it is printed. Here too, a carriage return and linefeed are generated.

Example d) illustrates the ability of `PRINT` to print several variable values from one statement. The comma supresses 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 ST divides each output line into 14-character-long TAB positions. If a comma is placed between two variables, the second variable is printed at the start of the next tab, at the 15th column on the screen line. If several variables are separated by commas, they are printed at the 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 supresses both the carriage return and linefeed. It also supresses 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 39) 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 good, 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

The PRINT USING command represents a modified form of the PRINT command. You will want to learn how to use this command to output formatted numbers and strings, such as dollar amounts. The following control characters are available for PRINT USING:

For numerical output:

#	indicate position of digit
+	print + sign with positive numbers
-	print - sign with negative numbers + and - are mutually exclusive
.	designate position of decimal point
**	fill with * instead of blanks
$$	print $ as the first character
**$	fill with * and print $ as first non-numeric character
,	place a , each 3 places before decimal point
^^^^	print values in scientific notation (e.g. 1.23 E+02)
_	Suppress control function of the next character, such as for #

For text output:

!	print only the first character of string
\ \	print selected number of characters as determined by number of characters between \ \ + 2
&	print the entire string variable

Let's further clarify the use of the `PRINT USING` with some examples. First enter the following into the computer in the command window:

```
A  = 12345.678                    <RETURN>
B  = 34.34555                     <RETURN>
C  = -520                         <RETURN>
D$ = "Atari ST"                   <RETURN>
```

Enter the following statements in the command window. Press the <RETURN> key after each one.

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

As output you get the value rounded to two decimal places, displayed in the output window:

Output:

```
12345.68
```

Now input:

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

Output:

```
34.35
```

Note that the decimal point is printed in the exact same position for both values. This would not happen with the ordinary `PRINT` command.

If the specified format is exceed by the value of the number, the number is printed with a preceding percent sign. The specified format cannot be followed in this case.

The following PRINT command:

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

causes this output:

Output:
```
%12345.60
```

The percent sign tells you that the value does not fit in the format field you choose for the PRINT USING command.

If you want to emphasize positive vs. negative results, the following combinations will work:

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

Output:
```
+12345.68
```

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

Output:
```
12345.68+
```

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

Output:
```
520.00-
```

Here the negative designation is printed following the number. Normally it is placed in front of the number.

The next example shows you how to fill the output of numbers in the given format with asterisks.

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

Output:
```
**12345.68
```

```
PRINT USING :**#####.##";B
```

Output:
```
*****34.35
```

The next form of the output places a dollar sign in front of the number.

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

Output:    $12345.68
```

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

Output:    $34.35
```

Naturally, the dollar sign and asterisk sign can be combined with each other, as the following example shows.

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

Output:    *****$34.35
```

Furthermore you can have commas automatically placed at every third place before the decimal point. Additional designations like "dollars" are also possible.

```
           PRINT USING "**##,###.## dollars";A

Output:    **12,345.68 dollars
```

To output the number values in exponential notation, the following form of the `PRINT USING` command is used:

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

Output:    1.23E+04
```

You can also determine how many places appear before or after the decimal point, as the following example shows.

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

Output:    12.34E+03
```

The _ character is used in order to output control character like #.

```
                PRINT USING "FILE NUMBER _###";B

    Output:     FILE NUMBER #34
```

This concludes the explanations of the numerical output forms of `PRINT USING`. Now let's take a look at the options for text output.

```
                PRINT USING "!";D$

    Output:     A
```

By using the exclamation point, only the first character of the string variable `D$` is printed. Remember that we assigned the string `Atari ST` to `D$`.

The next form allows us to output an arbitrary number of characters of a string.

```
          PRINT USING "\   \";D$:REM THREE SPACES

    Output:     Atari
```

A total of five characters are given between the quotation marks (including the two slashes). This causes the first five characters of D$ to be printed. This use of the `PRINT USING` command is related to the `LEFT$` function, which will be discussed later.

The last option outputs the entire string.

```
                PRINT USING "&";D$

    Output:     Atari ST
```

This could also be done with:

```
                PRINT D$
```

You may need the `&` character within a `PRINT USING` instruction sometime, though.

For the sake of thoroughness we should also mention the command:

```
WRITE
```

This command also has a close relationship to the PRINT command. Its output is somewhat different, however. Enter the following command in the command window:

```
WRITE "Atari ST"
```

Then press the <RETURN> key. "Atari ST" is displayed in the output window. The quotation marks are not displayed by the PRINT command. If numbers or variables separated by commas follow the WRITE command, the comma does not have a tab function, but is also displayed.

```
WRITE 2,3,4
```

Output:

```
2,3,4
```

Now we have described all the commands that occurred in our example program, including two close "relatives." You shouldn't have any great difficulty using these commands.

In order to make programs understandable to others—and also for yourself—we will take a look at the REM instruction.

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 will be ignored by the computer, including other BASIC commands.

We will now expand our example program and also make it more understandable for others. This is part of documenting the program, by the way. Here is the modified program listing 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.*3.14159527*R^2
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. In line 30 the input of the data is done 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 computation of the mathematical expression. Line 60 makes references to the output of the CM^2 in line 70. The text and value of the variable are printed in line 70. Line 80 concludes the program with the `END` command.

2.2 Variables and their use

Before we give you some sample problems to solve, we must discuss the various types of variables.

In ST BASIC there are three types of variables, each designated in a different way. The first type of variable is the integer variable. This variable type can represent only whole numbers. The designation is made with the percent sign, which is simply appended to the name of the variable (such as `A%` or `C4%`). If this variable type is assigned a non-integer value, only the places before the decimal point are taken into account. A further restriction is placed on the values of variables of this type, in that only values between -32767 and 32767 are allowed.

The second variable type, the `REAL` variable, is used to represent decimal numbers. The variable designation is an exclamation point placed after the variable name, although this is not absolutely necessary. Examples of the permitted designations are `A!` or `B2!` (`A` or `B2`).

The third variable type is designated by the dollar sign. These are the string variables. These variables can store arbitrary strings of characters. No more than 255 characters may be placed in a string variable or the following error message will be printed:

```
    Strings cannot be longer than 255 characters
```

Certain things must be taken into account when using variable designations. Atari ST BASIC recognizes a variable only by the first 31 characters of the name. The variable name can be any length, but only the first 31 characters are significant. Many computers cannot distinguish between such long variable names. This makes programs easier to read and understand, because variables can be assigned intelligent designations like `payment`, `exchange`, or `name`. BASIC keywords may also be found within variable names. The variable name `LAND` *is* legal, despite the fact that the keyword `AND` is contained within it.

Furthermore, digits may be used in the name with the limitation that the first character must be a letter. Names like `Amount1`, `A9`, and other are allowed. It is not permitted to have a number in the first location. For instance, a name like `9Amount` is illegal.

You must also be careful not try to use BASIC keywords as variable names, like OR, FN, or ABS.

The Atari ST also uses the following variable names for internal functions:

```
GB, AS, ALL
```

You may not use these within your programs.

2.2.1 Calculations with variables

If you want to perform calculations with the variables in your programs, you must first become acquainted with 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 problems 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 how they apply in programs.

Problems

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 to now we have concerned ourselves with assignments of values to variables. The mathematical functions of the Atari ST were not used; only the four basic computations were used.

In this section we'll talk about the built-in functions like `COS(X)` or `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 user's manual for the ST you'll note that the arguments for trigonometric 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. It takes some getting used to the actual calculation with radians at the beginning since you can more easily imagine 90 degrees than $\pi/2$ radians.

This short excursion into mathematics is enough for now. Let's write and use a couple of 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
25   PI=3.1415927
30   SI=SIN(DG*PI/180)
40   PRINT USING "THE SINE OF ####.##"+CHR$(248);DG;
45   REM THE CHARACTER CODE FOR DEGREE IS 248
50   PRINT" IS =";SI
60   END
```

Start the program by typing RUN and 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. Since the variable π is not directly available on the ST, it must first be defined in line 25. 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.

In order to make clear the difference from radians, 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 25 is omitted from our previous example and line 30 has been changed a little. Start the program and enter the value 1.57079633 (which corresponds to π/2). Again, the result should be 1.

The use of the other functions is quite simple. As explained in the ST BASIC 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=2.71827183 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:

```
2.71828
```

Repeat the same process with the following command:

```
PRINT LOG(2.71828183)
```

You again get the value 1. If you want to calculate the logarithm of base 10, you need only replace `LOG(X)` with `LOG10(X)`.

The following example program calculates both the logarithm base 10 and the natural logarithm.

```
10  INPUT "INPUT THE VALUE";N
20  REM CALCULATE NATURAL LOG
30  LN=LOG(N)
40  REM CALCULATE LOG BASE 10
50  LO=LOG10(N)
60  PRINT "ln(";N;") =";LN
80  PRINT
90  PRINT"log(";N;") =";LO
110 END
```

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 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.

You can also round off numbers with the function `CINT`. But here the numbers may only be in the range from -32768 to +32767.

```
PRINT CINT(-35.6)
```

Output: `-36`

The function `CSNG` and `CDBL` convert double-precision variables (4 bytes) to single precision (2 bytes) and back again.

With the function `MOD` you can calculate the remainder of a division. Enter:

```
PRINT 32 MOD 7
```

Output: `4` (32=4*7 rem 4).

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 more clear.

```
10  REM DEFINITION FUNCTION
20  DEF FNF(X) =X^2 + 2*X+4
30  REM INPUT PARAMETER
40  INPUT"ENTER VALUE";X
50  REM OUTPUT
60  PRINT FNF(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 of the ST 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
15   RANDOMIZE 0
20   A = INT (6 * RND(1)) + 1
30   PRINT A
40   END
```

Start the program with RUN and <RETURN>. Execute the program several times in a row and note the numbers printed. You will not be able to detect any pattern in the order of output.

A combination of RND and INT was used in line 20 in order 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.

If you want to pass a new start value (seed) to the random number generator, the command:

```
RANDOMIZE
```

is available in ST BASIC. It's very simple to use, as the following example shows.

```
RANDOMIZE 3
```

If you enter this command, the random number generator will be assigned the new seed value 3. Now we can assign a new random number to A with the command:

```
A=RND(1)
```

2.3.3 More commands for variables

The comprehensive BASIC of the Atari ST has a set of commands that can be used to affect variables or variable formats.

Recall the chapter on number systems. The ST offers two functions that allow you to convert decimal numbers into hexadecimal or octal.

The function:

```
HEX$(X)
```

converts decimal numbers into hexadecimal numbers. Here X stands for the number to be converted. X may assume values between -32768 and +32767.

Example:

```
PRINT HEX$(60)
```

Output:

```
3C
```

The function:

```
OCT$(X)
```

converts decimal numbers to numbers in the octal system (base 8). Again, X stands for the number to be converted. A value between -32768 and +32767 must be passed to the function. For example:

```
PRINT OCT$(16)
```

Output: `20`

In both examples, variables can be used instead of constants. As already mentioned, there are three different types of variables in ST BASIC. If you want to assign specific variable names with specific variable types within a program, the following commands are at your disposal:

```
DEFDBL, DEFINT, DEFSNG, DEFSTR
```

For example, if you define:

```
DEFSTR A-B
```

within a program, all variables that start with A or B will be treated as string variables. You need not append a dollar sign to these variables within a program to use them as string variables. The instruction:

```
DEFSNG C-D
```

defines all variables starting with C or D as real (single-precision) variables. The other commands are used similarly.

If you want to output only the integer portion of a number, you can use:

```
FIX
```

FIX works similarly to the INT command, but FIX performs a strict truncation of the number. INT is actually the greatest-integer function of mathematics (where one is added to a negative number).

```
PRINT FIX(-3.99)
```

Output: `-3`

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

The ST 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 you have the ability to output any character of the character set. Enter the following in the direct mode:

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

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

The function ASC represents the reverse 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 the 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 Atari ST user's manual.

To give you a chance to use your newly-acquired knowledge, we want to give you some problems to solve. Compare your results with the suggested solutions and explanations in Appendix C. 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 here.

Problems

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

These two functions 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. In the next example you will see the difference. Enter the following commands:

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

After pressing the <RETURN> key the word TEST 1 will appear 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 will 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 not interpreted or interpreted as SPC. 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 ST'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 numeric variables. The sole difference is the characters are enclosed in quotation marks. The following example shows a valid assignment:

```
A$="Atari ST"
```

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

```
Types of values do not match
```

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 more 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 will get to this, but not until the compare commands are discussed (see IF...THEN...ELSE). When comparing, only strings may be compared with strings. Comparing a string variable to a numerical variable is not legal.

2.5.1 LEFT$

ST BASIC 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 to make this more clear:

```
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 next function is similar to the LEFT$ command in the way it works. 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 we want to discuss.

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
```

Output:
```
343.35
```

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

Output:
```
0
```

c)

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

Result:

```
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 which is 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$
```

Output:

```
1234
```

b)

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

Output:

```
-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

ST BASIC offers us another useful function for working with strings:

```
INSTR
```

This allows you to search through a string for a desired substring of characters. The syntax of the function looks like this:

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

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 C=INSTR(A$,B$)
40 PRINT C
50 END
```

`RUN` the program. You'll get the value 6 as the result for variable `C`. 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 STRING$

This function creates a string which contains a sequence of the same character. For example, enter the following into the computer:

```
A$=STRING$(40,"*"): PRINT A$
```

This prints a line with 40 asterisks as the output. Here again, the upper limit is 255.

2.5.9 SPACE$

The SPACE$ function is similar to the STRING$ function.

```
A$=SPACE$(40)
```

The command line above causes A$ to be filled with 40 spaces. This statement can be used for exact positioning of output, as the following example shows:

```
PRINT SPACE$(12);"Atari ST"
```

This example prints 12 spaces followed by the string "Atari ST"

Before we move on to the next chapter, you should solve the following problems so that you learn to use the new commands.

Problems

1. What difference is there in the output that the following two command sequences produce? Don't enter them in the computer, but try to answer the question.

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

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

   a) The only difference is that the first command sequence will cause the string "TEST 1" to be printed one more character to the right.

   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 A$ if A$="ROTOR"?

```
A$=LEFT$(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"?

2.6 Editing Programs

Before we continue to develop larger programs, we will first look at the commands that make programming somewhat easier. The term *editing* includes anything having to do with changing a program—whether it's deleting or inserting program lines, or correcting syntax errors. We assume that you know how to work with the edit window, as well as insert or delete lines. If you still have difficulty with these, read the corresponding section in the ST manual.

We already mentioned that the line numbering should be done in steps of ten. A command that helps you do this is:

```
AUTO
```

If, for example, you enter `AUTO 10,10` into the computer and press the <RETURN> key, the computer will automatically provide you with line numbers in increments of 10. The first value passed to the `AUTO` command specifies the first program line. The second value determines the increments, i.e. the distance between individual program lines.

If you want to turn the automatic line numbering off again, press the keys <CONTROL> G.

If you're writing a program, sooner or later you'll need:

```
RENUM
```

Lines almost always have to be inserted when developing programs. The numbering could soon look like this:

```
10...
12...
19...
20...
21...
```

Now enter `RENUM` from the command window and all program lines will be renumbered in steps of 10, so that the example above would have line numbers from 10 to 60. This command also takes the destinations of `GOTO, GOSUB`, etc. into account so that jumps still work correctly after the command.

You can also specify a different line increments as well as a range of lines to be numbered when you renumber your program by entering:

```
RENUM X,Y,Z
```

Here X stands for the line number with which the new numbering is supposed to begin. Y stands for the line number through which the lines will be renumbered, and Z determines the step width. The command:

```
RENUM 200,100,5
```

would renumber the program beginning at old line number 100 in steps of five, starting with the new line number 200. The RENUM command writes a file called BASIC.WRK. Be sure that the disk in the drive is not write-protected.

It may also occur that you want to delete only certain lines from a program. To do this you would enter:

```
DELETE 100-200
```

into the computer and all lines between 100 and 200 (inclusive) will be erased. You can also vary this command by entering:

```
DELETE -200
```

or

```
DELETE 200-
```

You can also delete program lines up to a certain line number or at certain line numbers. Use this command carefully, however, because you won't get a second chance. Once you press the <RETURN> key, the lines are gone for good.

If you are testing a program and want it to stop at a certain program line, perhaps to check to see if it had run correctly up to that point, you just insert the command:

```
STOP
```

at that point. If the interpreter encounters this command in a program, it stops with the message:

Stop at line (*line number*)

With the command:

```
CONT
```

you can cause the program to continue at the same place, while retaining all of the variable contents. In contrast, the command RUN sets all variables to zero, even if you use RUN with a line number. This is an important difference from the CONT command. You can't use CONT if your program stopped because of an error.

Sometimes it is useful to follow the course of a program by means of the line numbers, perhaps to make comparisons to the program flowchart. After entering the command:

```
TRON
```

(TRace ON) each line number is output in the command window in square brackets before the line is executed. The command:

```
TROFF
```

(TRace OFF) disables TROFF.

The command:

```
TRACE
```

shows each program line which is currently being processed. By specifying a line number you can also have only specific lines appear.

```
TRACE 40
```

causes only line 40 to be printed. You have the same combination possibilites for TRACE as with DELETE and LIST.

This aid is disabled with the command:

```
UNTRACE
```

If you want to know the contents of variables in the individual program lines, you can use:

```
FOLLOW
```

`FOLLOW A` shows you the contents of the variable `A` with line number throughout the execution of the program, for example.

By specifying:

```
UNFOLLOW
```

the `FOLLOW` command is turned off.

The command:

```
NEW
```

erases the program currently in memory. You have already used this command in your problems—we mention it again only for the sake of thoroughness. With the command:

```
LIST
```

you can display the program in the screen list window. You can vary this command in the same manner as the `DELETE` command. You can output the program to the printer with `LLIST`.

Two more commands are available for erasing variables. The command:

```
CLEAR
```

clears all variables and arrays. If you are not yet familiar with the term *array*, don't worry. It will be explained in detail in a later chapter.

If you want to erase only arrays or one array, you must use:

```
ERASE
```

For example, `ERASE A` causes the array previously created with `DIM A(20)` to be erased. The array can then be redimensioned.

By entering:

```
EDIT
```

you switch directly to the `EDIT` window.

These editor commands offer you an easy-to-use means of creating and correcting your programs.

2.7 The screen windows

The individual screen windows (Edit, List, Output, and Command) can be closed, opened, and erased from BASIC. The individual commands to do this are discussed briefly here. The individual windows are assigned the following numbers:

0 = Edit window
1 = List window
2 = Output window
3 = Command window

The command:

```
FULLW 2
```

sets the output window to the full screen size.

To erase the contents of a window, use the command:

```
CLEARW
```

CLEARW 3, for instance, erases the contents of the command window.

The command:

```
OPENW
```

opens a previously closed window.

```
CLOSEW
```

CLOSEW can close a window. The corresponding window disappears from the desktop completely.

CAUTION! If you close **all** windows, the computer will no longer accept any input. You're then forced to restart the system.

This concludes the chapter. In the next chapter we will move on to extended programming structures, and programming with loops.

Chapter 3

EXTENDED PROGRAM STRUCTURES

3.1 Unconditional program jumps
3.2 Conditional program jumps
3.3 Calculated jump commands
3.4 Reading the keyboard
3.5 `FRE`, `POS`, `CALL`, and `WAIT`
3.6 `READ`, `DATA`, and `RESTORE`

Extended program structures

Up to now we have limited ourselves to linear programs. Now we'll move on to programming program jumps, or *branches*. Linear programs have the disadvantage that the program runs once, and must then be restarted in order to get a new result. No branches of any kind take place. If there were no commands to perform such program branches, you would be limited to writing only very simple programs in BASIC.

3.1 Unconditional program jumps

The first and simplest type of a program branch is the GOTO command. This command allows the program to deviate from linear execution as determined by the sequence of line numbers. It is called an *unconditional program branch* because it is not tied to any condition—that is, the program performs this jump under all circumstances.

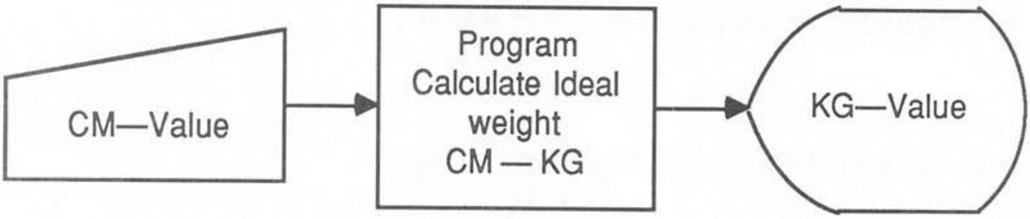
The disadvantage of unconditional program branches is that you can create only "infinite" loops with them. Once the program is started, it can be stopped only with <CONTROL> G or clicking BREAK.

We will use this command with an example we've already discussed: the problem where we had to calculate a person's ideal weight.

Imagine you're giving a party and want to use this problem as a little gag. Each of the guests will learn his or her ideal weight. Without the GOTO command, the program would have to be restarted every time. Therefore we place a GOTO command before the END command. This tells the computer to jump to the start of the program. Our program would then look like this:

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

This program calculates a person's ideal weight in one line, line 30. After outputting the result, the program encounters the GOTO command in line 70 and branches to line 10. A new value for the calculation can then be entered. Line 80 does not have to be entered because the program never reaches this line as a result of the GOTO command. The data flowchart is not affected by this command. The figure below shows how it would look.

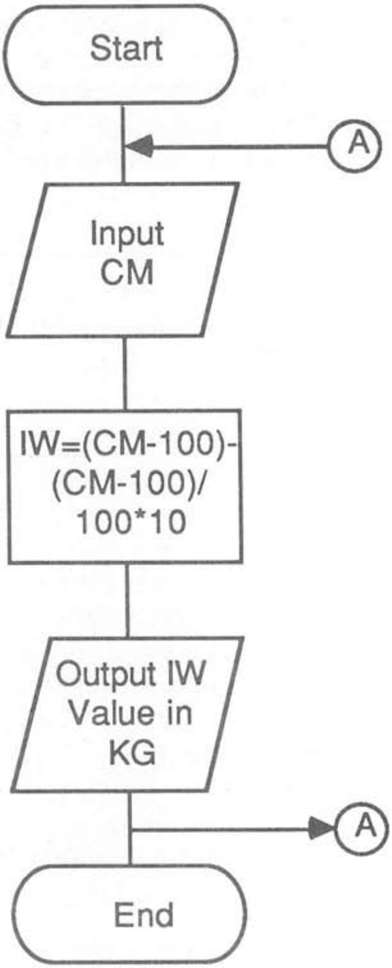


Data flow chart

The program flowchart does change as a result of this command. A symbol is added to it. This is the *connector*. It designates the location at which the program is to continue when the jump connector is reached. The jump connectors stand at the place where the GOTO command is found in the program. The entry connector is located directly after the start symbol. Both connectors are designated with an A, because they form a connector pair. The program flowchart is shown in the following diagram.

The following program lines can be added to convert from metric to English units.

```
10 INPUT "ENTER YOUR HEIGHT IN INCHES";IN
15 CM=IN*2.54:REM CONVERT TO CM
45 IW=IW*2.2:REM CONVERT TO LBS
50 PRINT "YOUR IDEAL WEIGHT IS";IW;"LBS"
```



Program flowchart

The end symbol can be omitted, but was included here for the sake of thoroughness. This example shows that only infinite loops can be created with the GOTO command—the program can't end without a checking condition. Conditional program jumps remedy this situation.

3.2 Conditional program jumps

One of the strengths of a computer lies in its ability to make logical decisions or comparisons. For example, it can test to see if a variable is greater or less than zero, and then branch within the program depending on whether the result is True or False. The `IF...THEN...ELSE` command is one that performs a comparison.

3.2.1 `IF...THEN...ELSE`

If the computer encounters an `IF...THEN...ELSE` command when executing a program, it checks the condition that follows the `IF`. If the condition is True (fulfilled), it executes the instructions or commands that follow the `THEN`. If the condition after `IF` is not fulfilled—if it is False—the computer continues with the next program line, or the commands following the `ELSE`. All instructions or commands following the `THEN` are ignored. `ELSE` is optional; it need not follow the `IF...THEN`.

Logical operators, strings, variables, comparisons, numbers, or their combinations can follow `IF`. Usually a line number follows the `THEN` to which the program is to branch. Assigning new values to a variable is also possible. We will talk more about this in a later section. Let's first take a look at a simple example of the use of the `IF...THEN` command.

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

With this program you can enter a number and the computer will tell you if this number is greater than, less than, or equal to zero. While this is a trivial program, this simple example will clarify the use of the IF...THEN command in a program and the computer's reaction to it.

Suppose you enter a number less than zero. The computer comes to line 20—here a check is made to see if the number is greater than zero. This condition is not fulfilled, so the computer continues with the execution of the next program line. There a check is made to see if the number entered is less than zero. This condition is true, so the computer jumps according to the instruction following the THEN, to line 70. Line 70 outputs the message on the screen that the number is less than zero. In line 80, the computer encounters the unconditional jump command GOTO and jumps to line 100, where the program ends. If you want the program to run continuously, you need only replace the END command in line 100 with GOTO 10.

Following this simple example, we want to turn to a more complicated program. No doubt you're familiar with the game where someone thinks of a number and someone else has to guess it. After each question the person guessing is told if the number is larger, smaller, or equal to the number in mind. We'll recreate this game on the computer. To do this, enter the following program:

```
10  REM NUMBER GUESSING
20  CLEARW 2:PRINT:RANDOMIZE 0
30  PRINT"ENTER TWO NUMBERS FOR"
40  PRINT"THE UPPER AND LOWER BOUNDARIES"
50  PRINT"LOWER LIMIT";L
60  PRINT"UPPER LIMIT";U
70  N=INT((U+1-L)*RND)+L
80  INPUT"YOUR GUESS";NG
90  IF NG < N THEN 120
100 IF NG > N THEN 140
110 IF NG = N THEN 160
120 PRINT"THE NUMBER IS LARGER"
130 GOTO 80
140 PRINT"THE NUMBER IS SMALLER"
150 GOTO 80
160 CLEARW 2:PRINT"HURRAY! YOU GUESSED IT!"
170 PRINT"PLAY AGAIN (YES/NO)";
180 INPUT A$
190 IF A$="YES" THEN 20
200 END
```

The first lines of this program consist of a remark and a randomizing function. The interval for the number to be guessed is prompted in lines 30 to 60. Line 70 determines the number sought with the entered range.

If line 70 gives you difficulty, refer back to the section on random numbers. Line 80 asks you to enter a number. This number is compared with the random number, stored in the variable N, in lines 90 to 110. According to whether the number is greater, less than, or equal to the intended number, the computer branches to the corresponding line. There it continues with the program. If you guessed the number, the program jumps to line 160. In line 170 you are asked if you want to play again. If you enter YES, the condition in line 190 is fulfilled. The program starts all over again.

In lines 90 to 110 the IF...THEN command is used to make comparisons between the number the player guesses (NG) and the random number (N). In line 190, the IF...THEN command is used to make a comparison to a string variable. Note that in order for the condition to be true, both strings must be exactly the same. You could enter Y in line 180, but the program will end because the branch to line 20 is made only if the characters Y, E, and S are entered—the string YES.

With the IF...THEN command we now have the ability to program controlled loops. Controlled means that the loop will not be executed continuously, but will be executed only as long as a specified condition is fulfilled. The following program shows you how such a controlled loop is programmed. If you want to output the times table for three, for instance, you would write a program like this:

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

In line 10, the variable A is first initialized with the value 3. Line 20 outputs the current value of A on the screen. In line 30, a counter is used. It always adds the value 3 to the current contents of variable A. In line 40 a check is made to see if A has exceeded the value 30. As long as A is less than or equal to 30, the program continues with the GOTO command in line 50. We have created a loop that is executed exactly 10 times. We have now learned a way to create loops that are executed a certain number of times.

The above example is well-suited to using ELSE. We can then get rid of line 50. Here is the modified program with ELSE:

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

When you start this program, you will see that you get the same results as with the previous example. Since in the first 10 comparisons variable A is not greater than 30, the condition is not fulfilled, the command behind the ELSE is executed. The program then branches to line 20. As soon as A has the value 33, the program is ended at line 60. We retained the line number 60 to emphasize the fact than line 50 is missing.

3.2.2 Labels

In ST BASIC you have the ability to assign labels to jump destinations. These labels can then be jumped to with GOTO or GOSUB. You don't have to remember the numbers of the lines at which routines in program start. Instead of:

```
50 GOTO 250
```

you could simply use:

50 GOTO *(Label)*

The corresponding line 250 must then look like:

250 *(Label)*:

The label must be terminated with a colon.

You can name individual sections of the program by assigning labels, making your program easier to read and understand. For example, if you want to call a sort routine in your program you need not search for the line number, but simply enter:

```
GOTO Sort
```

or

```
GOSUB Sort
```

Labels may not begin with a number. The first character of a label must always be a letter.

Here are a few exercises to help you become familiar with the new commands.

Problems

1. Write a program that calculates an income tax of 33 or 51 percent based on the annual income. The border should be at an annual income of $50,000. All amounts larger than $50,000 must be taxed at 51 precent. The output of the result should be done with accompanying text.

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

3. Write a program that outputs 6 random numbers in the range 1 to 49.

4. Which numbers are printed by the following program? Solve the problem without entering the program.

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

5. Write a program that searches for a given substring in a given string. As a test use the string A$="INFORMATION" and search for and output the substring B$="FORMAT". The difficulty of this problem is that you may not use the INSTR function. Now write a program in BASIC that replaces the INSTR function.

3.2.3 FOR...TO...NEXT

Up to now we have created loops with the IF...THEN command. Here a counter is used and its value is incremented or decremented. The value of the counter is checked at certain points in the program, and the program jumps to another line depending on the result of the test (true or false). Creating a loop like this is rather complicated, since the counter and the test are extra programming statements. You probably suspect that BASIC offers a simple solution. It does, in the form of FOR...NEXT loops.

Look at an example program introducing this method of creating loops:

```
10 REM OUTPUT THE FIRST 10 PERFECT SQUARES
20 CLEARW 2:PRINT
30 PRINT "THE FIRST 10 PERFECT SQUARES"
40 FOR I=1 TO 10
50 PRINT "SQUARE OF";I;"=";I*I
60 NEXT I
70 PRINT "DONE"
```

Enter the program into your computer and RUN it. It works similar to that of the IF...THEN command. Programming loops with FOR...NEXT is simply more elegant and also saves memory space.

The index variable is called I, and it is assigned an intiial value. In our case this is 1. The initial value is then incremented by 1 until the end value is exceeded. Every command appearing between FOR and NEXT will be repeated as often as the loop is executed. The initial and end values can be numbers, variables, or arithmetic expressions.

Some examples:

```
10 A=10:B=20
20 FOR N=A TO B
30 PRINT N;
40 NEXT N
50 END
```

In this example the variables A and B are first initialized. Line 20 begins the loop using these variables. Line 30 outputs the values of N until the index variable is greater than 20. This is comparable to the IF...THEN command.

It might look like:

```
IF N > 20 THEN 50
```

The FOR...NEXT loop in our case is executed until N is greater than 20. You can check this by entering the command:

```
PRINT N
```

in the direct mode after the program is done. As the result for N you get the value 21! The next example will show that arithmetic expressions can also be used.

```
10 A=10:B=15:C=5
20 FOR N=1 TO A+B-C
30 PRINT Z;
40 NEXT X
50 END
```

The only difference from our first example is that the end value is calculated from the expression A+B-C.

If you want to use an increment other than 1, the step size must be specified with STEP. The following example outputs the even numbers between 2 and 20, using steps of 2.

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

The starting and ending values may also be negative or fractional numbers, as may be the step width. As an example we will program a countdown.

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

After you start the program, the output quickly flashes before your eyes. Normally a countdown counts down in increments of thousandths of a second. There is a solution for this hitch. We can "nest" FOR...NEXT within each other. The next example shows what this means.

```
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
```

RIGHT!

Enter the program and run it, noting that it now counts down in increments of almost exactly one second. This is taken care of by a *delay loop* in lines 40 to 60. Such delay loops often used to display text on the screen for specific periods of time.

The delay loop in our program is intended only to clarify the nesting of FOR...NEXT loops. Naturally other BASIC commands could be in this nested loop.

What happens in this program? In line 20 the first loop starts with I=20. Line 20 outputs the current value of I. In line 40 the second loop starts—its NEXT is found in line 50. This second loop is processed completely before the first loop starts through its second pass. The second loop is completed as often as I is printed.

You must make sure that your nesting of FOR...NEXT loops is legal. You may not "cross" loops. That is, the first loop opened must be the last closed, and the last loop opened must be closed first. The program above shows correct nesting of loops. The following example is intended to show how loops may *not* be nested.

```
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!

If you have several loops nested within each other and want to close them all at once, you don't need a special NEXT for each FOR. One NEXT suffices. To this NEXT are appended the individual index variables *in the proper order*. The variables must be separated from each other by commas. The following example will clarify this.

```
10 FOR I=1 TO 10
20 FOR Z=1 TO 10
30 PRINT I;Z
40 NEXT Z,I
50 END
```

Enter this program into the computer and RUN it. Only one NEXT is used in line 40 to close both loops. Here again, the loop last opened must be the first closed. This is why the variable Z follows the NEXT first, and then I.

One mistake beginners often make is jumping into a loop. That means a jump is not made to the FOR...TO instruction, but somewhere in between FOR and NEXT. Since a loop usually contains several program lines, sometimes you jump to a line in the loop where everything turns out all right. The error is usually gone unnoticed until the program is run for the first time. The result is the termination of the program with the following error message:

```
    You're trying to jump to a loop at line (line #)
```

If you had first made a detailed flowchart, such an error probably wouldn't have occurred. Further, if the starting value is larger than the ending value, the step value must be negative. If you forget to specify the step value, the loop will be exited immediately, as the following example shows:

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

In line 10 there is no specification of the step width, such as `STEP -1`. Hence, no output is given. Ending a loop prematurely is done by setting the index variable to an ending value. This can be done independently on certain variables, or other conditions that can be tested within the program. Normally the start and end values are placed in variables. If the values of these variables change within the program, different loop lengths are possible.

In conclusion, we'd like to show you a program that interrupts the loop prematurely by setting the index variable to the end value. Then we'll present a program that determines the different loop lengths through variables. The values of the variables are determined with the string functions. Such applications are frequently found in database programs —when searching for character combinations, for instance.

```
10 REM PREMATURE LOOP END
20 FOR A=0 TO 20
30 PRINT A
40 IF A=12 THEN A=20
50 NEXT A
60 END
```

The program doesn't make a whole lot of sense, since it's so short. It is intended only to show how a loop can be prematurely ended. Normally the loop would count to 20. However, in line 40, once A reaches 12 it is set to 20. This causes only values up to 12 to be printed. This method of changing the value of the index variable within the loop is seldom used.

More often the start and end values of a loop are placed in variables. This allows the loop to be controlled more easily. The following example illustrates this:

```
10 INPUT"ENTER 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, enter your name, and press the <RETURN> key. You see that you get the same results as we got from the program in the section on string functions. But here we used the FOR...NEXT loop and made it dependent on the length of the string entered. This means that the passes through the loop are controlled by the length of the string. Take a close look at the example and try to understand all of it.

Let's summarize the most important points of working with FOR...NEXT loops.

1. Exactly one NEXT instruction belongs to each FOR instruction. A NEXT instruction can close several nested loops if the index variables follow this NEXT instruction in the proper order, separated by commas.

2. You may not jump into a loop, because the program will terminate with an error message.

3. The starting value may not be larger than the ending value if the step width is positive, or the loop will not be executed. The same applies for negative step widths.

4. Generally, a FOR...NEXT loop is executed until the value of the index variable is greater than the end value.

These rules apply only to ST BASIC, and shouldn't be generalized to other computers. There are some differences in the use of FOR...NEXT loops among the different BASIC dialects.

3.2.4 Loops with WHILE...WEND

The command combination WHILE...WEND offers you another option for constructing loops within a program. This form of loop control is more flexible than the FOR...NEXT loops. You don't have to specify a set increment for WHILE...END.

The start of the loop is indicated with WHILE, and the end with WEND. The logical expression following WHILE is tested before each pass through the loop. As long as the expression is true, the loop is executed up to WEND. If the condition behind WHILE is no longer fulfilled, the execution of the program is continued after the WEND. You terminate a loop with this command combination quite randomly, as the following example shows:

```
5  RANDOMIZE 0
10 WHILE A < 100
20 A=INT(101*RND)
30 Z=Z+1
40 IF A=100 THEN EXIT
50 WEND
60 PRINT A,Z
70 END
```

Here the WHILE...WEND loop is executed until the value of A reaches the value 100 by chance. If the statement in line 40 is true, the next command that follows the WEND is executed. In our case this is program line 60. Here A and the counter Z are printed. The counter tells you how many times the loop was executed. This example should make the operation of the WHILE...WEND loop clearer.

We can make the following general rules for the use of loops:

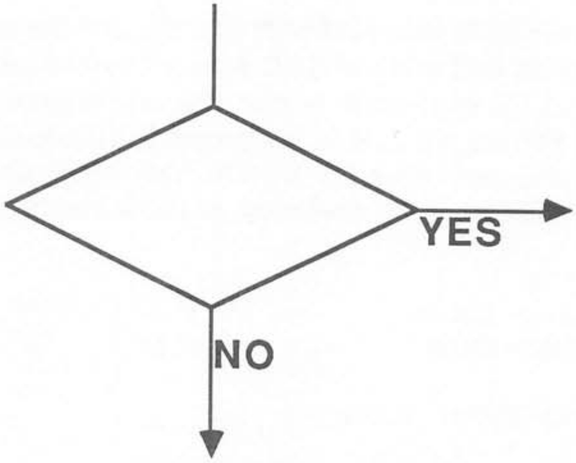
a) If the number of repetitions of the loop is known from the beginning, we use the FOR...NEXT loop.

b) If the number of repetitions of the loop is unknown, we construct the loop with IF...THEN or WHILE...WEND.

We have already seen an exception to these rules, in the last example program of Section 3.2.3. These rules are only intended to be guidelines.

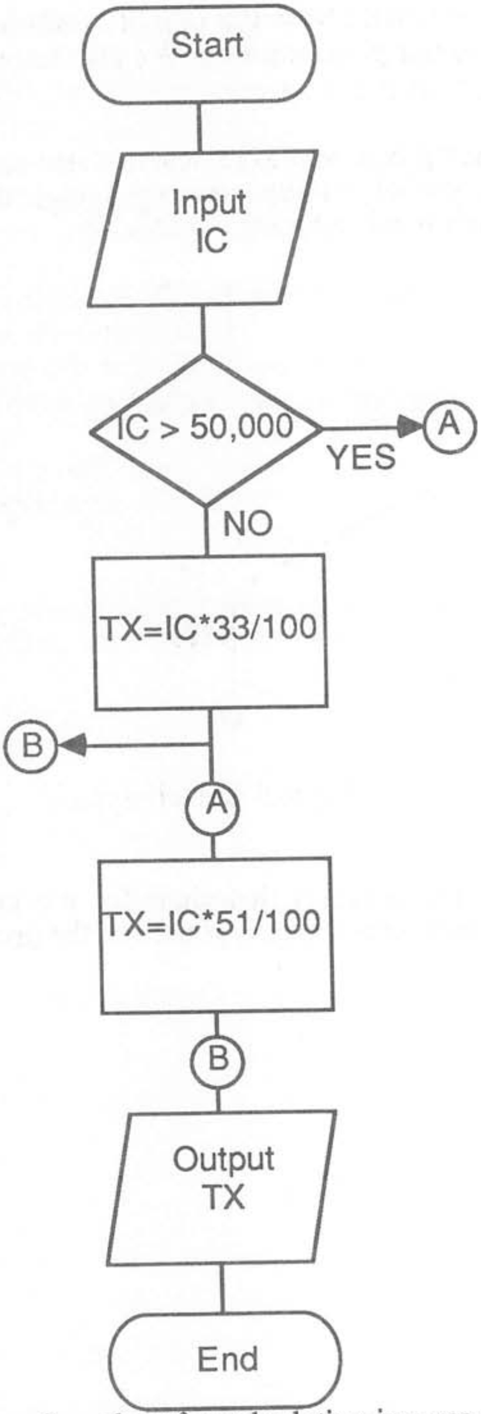
So far we have learned about the use of conditional and unconditional program jumps in our programming. We also know about program loops, especially the FOR...NEXT loop.

What is still missing is a way to represent these structures in the program flowchart. The symbol for representing a logical branch in a program flowchart is the diamond. It looks like this:



Logical Branch Symbol

First we'll look at a program flowchart for the program that calculates income tax. On the next two pages you'll see the program flowchart and its explanation.



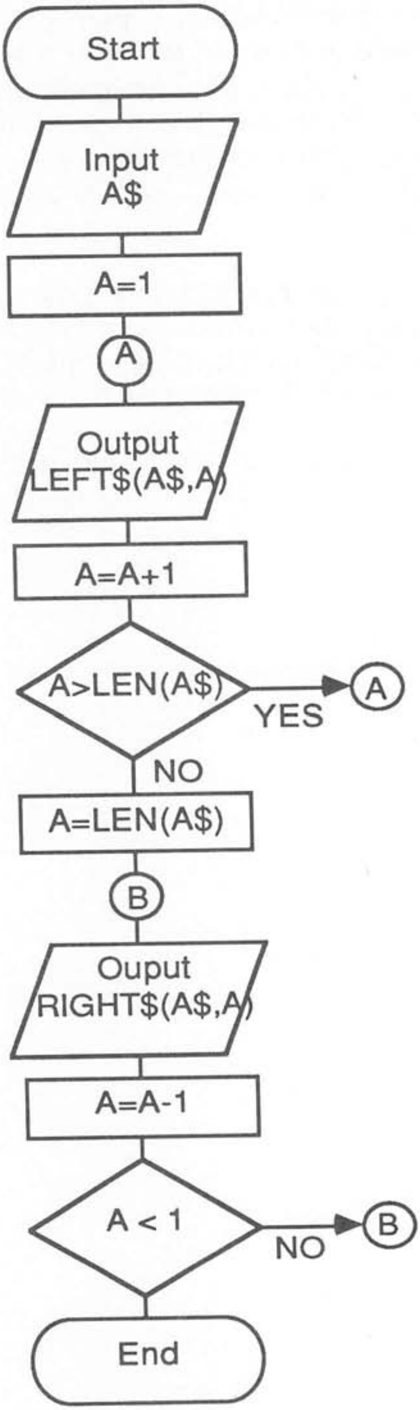
Program flowchart for calculating income tax

We recognize the *start* and *end* symbols in our program flowchart. The diamond, as we said before, is the symbol for a logical branch. It has a YES branch and a NO branch. If the condition is fulfilled, a branch is made via the jump connector A to the corresponding entry connector A. In our example the branch would take place via the YES branch. This could also be the NO branch, depending on the type of program. The calculation of the tax of 51 percent and output of the value follow the entry connector A.

If the condition is not fulfilled, the 33 percent is calculated. After the calculation comes jump connector B. It designates an unconditional jump to entry connector B. It should be noted that the jump connector B comes before the entry connector A, or the flowchart would contain a logical error.

This program flowchart showed how the IF...THEN command is represented in a flowchart. What we now need to know is how a FOR...NEXT loop is represented.

For this we'll use the program on page 96 that asks you to input your name, and write a flowchart for it. The next two pages contain the program flowchart and its explanation.



Program flowchart for the program on page 96

The symbols in this flowchart should be familiar to you. In the first rectangle, the starting value of the loop is set to one. Next we isolate the substring with `LEFT$(A$,A)`. The counter is then incremented by one. In the diamond the counter is tested to see if it is greater than the number of characters in `A$`. If this is not the case, a branch is made to entry connector A via jump connector A. The loop is thereby created in the program flowchart.

If the counter is greater than `LEN(A$)`, the second loop comes into action. The second loop has the same arrangement of symbols as the first loop. The connectors have the different designations in order to avoid confusion. However, the course is the same as the one described above.

With this information you should be in a position to create program flowcharts for any program on your own. Remember, practice makes perfect.

3.3 Calculated jump commands

The calculated jump commands have the advantage of making the actual program more flexible. Up to now we have seen only jump commands that jump to a specific program line. The line numbers in the GOTO command cannot be changed—GOTO 100 always continues execution at line 100.

But it would be nice if you could enter a value at the start of a program and the program would then branch based on the value. This could be achieved with some IF...THEN comparisons, of course. But this would require a jump command for each program line, for every comparison. A simple example will clarify this:

```
10 REM JUMP TO CERTAIN LINES
20 PRINT "ENTER A NUMBER BETWEEN"
30 PRINT "1 AND 4"
40 PRINT
50 INPUT "WHAT NUMBER";Z
60 IF Z = 1 THEN 100
70 IF Z = 2 THEN 200
80 IF Z = 3 THEN 300
90 IF Z = 4 THEN 400
100 PRINT "JUMPED TO LINE 100"
110 GOTO 410
200 PRINT "JUMPED TO LINE 200"
210 GOTO 410
300 PRINT "JUMPED TO LINE 300"
310 GOTO 410
400 PRINT "JUMPED TO LINE 400"
410 END
```

In this program, a branch is made to program line 100, 200, 300, or 400, depending on the input of a number 1 to 4. Programming these tests with IF...THEN is rather complicated in such applications, and is relatively slow in execution speed. BASIC offers a more flexible solution for such cases. The command has the following syntax:

ON (*variable*) GOTO (*line number or label*)

This extended GOTO command with ON allows the program to branch to one or several line numbers or labels following the GOTO. The range of the variables extends from zero to the number of line numbers give. If the variable does not have an integer value, the non-integer portion is ignored. Negative values are also ignored.

If the variable has a value that is larger than the number of line number available following the GOTO, the command following the ON...GOTO command is executed. Here are some simple examples:

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

If the variable Z in this example has the value 1, the program jumps to line 100. If Z runs through the values 2 to 4 in a loop, for instance, the program jumps to lines 200, 250, and 300 in succession. Z designates the positions of the individual line numbers that follow the GOTO. If Z assumes values larger or smaller than the number of line numbers behind the GOTO, the program continues with the next command following the GOTO. This is the PRINT command in our example. The ON...GOTO command is simply skipped.

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

You see that an arithmetic expression can be used instead of a variable. The advantage of this ON...GOTO command is that it can replace several IF...THEN commands. This saves programming time, as well as memory space. Our previous short program could have the following form:

```
10 REM JUMP TO CERTAIN LINES
20 PRINT "ENTER A NUMBER BETWEEN"
30 PRINT "1 AND 4"
40 PRINT
50 INPUT "WHAT NUMBER";Z
60 ON Z 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
```

We saved three program lines in this short program. With larger programs in which comparisons with `IF...THEN` can occur, you can save even more lines.

This program uses a programming technique that can be very helpful, especially with large programs. If you create a program flowchart for this program, the jumps to the various lines are symbolized with horizontal branches. Since we don't know yet what line numbers these lines will have, we choose extra-large numbers. This is how to make room within the program for other program segments.

The destination line numbers of `ON...GOTO` represent certain program segments within the program in which special tasks are generally performed. It helps to designate these with "smooth" line numbers. This can be done in steps of one hundred, as in our example program. This improves the readability of the individual program segments.

3.3.1 Example program—MATH TUTOR

We have now learned a relatively large number of BASIC commands. This is a good excuse to try a larger project.

Suppose you want to write a math drill program for your kids that features the four basic operators on the ST. The drill has the following properties:

1. One of the four operators is selected or the program is ended.
2. A given problem must be solved in a maximum of 3 attempts.
3. After the third failed attempt, the correct result is displayed.
4. After each problem, a prompt asks if more problems of the same calculation type are to be solved.

Take a look at the math lesson program listing starting below. Don't worry if individual lines are not always separated from each other by steps of exactly ten. Since the program is relatively long, we'll now give you the commands for saving a program, even though we haven't discussed them yet.

Insert a formatted disk in the drive. Enter the following command into the computer:

```
SAVE MATH
```

The program will now automatically be saved on the diskette. If you want it again, you need only call it up from the appropriate disk. This is done with the `LOAD` or `OLD` command. Simply replace `SAVE` with `LOAD` at the appropriate time and the program will be loaded into the computer.

Here is the program listing:

```
5      REM **** MENU ****
10     FULLW 2:CLEARW 2: 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"
77     ER=A1-A2
80     PRINT
90     PRINT TAB(12)"[2] - SUBTRACTION"
100    PRINT
110    PRINT TAB(12)"[3] - DIVISION"
120    PRINT
130    PRINT TAB(12)"[4] - MULTIPLICATION"
140    PRINT
```

```
145   PRINT TAB(12)"[5] - END"
148   PRINT
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 ********
210   REM ADDITION
220   REM ********
230   CLEARW 2
240   PRINT TAB(10)"INPUT THE LARGEST NUMBER"
250   PRINT
260   PRINT TAB(10)"FOR ADDITION"
270   PRINT
280   PRINT TAB(10);:INPUT"LARGEST";GR
299   REM
300   REM CREATE RANDOM NUMBERS
301   RANDOMIZE 0
310   A1 = INT (GR*RND)+1
320   A2 = INT(GR*RND)+1
329   REM
330   REM COMPUTE RESULT
340   ER = A1 + A2
350   CLEARW 2
360   PRINT
370   PRINT "HOW MUCH IS "A1"+" A2 " = ";
380   INPUT ES
390   IF ES=ER THEN PRINT:PRINT TAB(10)"CORRECT!"
      :F=0:GOTO 470
400   PRINT:PRINT TAB(10)"WRONG!"
410   FOR I = 1 TO 2000:NEXT I
420   F=F+1
430   IF F<= 2 THEN 350
440   PRINT
450   FOR I = 0 TO 2000:NEXT I
460   PRINT"THE ANSWER IS "ER
470   FOR I = 0 TO 3000:NEXT I
480   PRINT TAB(5)"ANOTHER PROBLEM? (Y/N)";
490   INPUT A$
500   IF A$="Y" THEN F=0:GOTO 300
510   GOTO 10
600   REM ***********
610   REM SUBTRACTION
620   REM ***********
```

```
630   CLEARW 2
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
699   REM
700   REM CREATE RANDOM NUMBERS
701   RANDOMIZE 0
710   A1 = INT (GR*RND)+1
720   A2 = INT(GR*RND)+1
729   REM
730   REM COMPUTE RESULT
740   IF A1 < A2 THEN I = A1:A1=A2:A2=I
750   CLEARW 2
760   PRINT
770   ER=A1-A2
780   PRINT "HOW MUCH IS "A1"-" A2 " = ";
790   INPUT ES
800   IF ES=ER THEN PRINT:PRINT TAB(10)"CORRECT!"
      :F=0:GOTO 880
810   PRINT:PRINT TAB(10)"WRONG!"
820   FOR I = 1 TO 2000:NEXT I
830   F=F+1
840   IF F<= 2 THEN 750
850   PRINT
860   FOR I = 0 TO 2000:NEXT I
870   PRINT"THE ANSWER IS "ER
880   FOR I = 0 TO 3000:NEXT I
890   PRINT TAB(5)"ANOTHER PROBLEM? (Y/N)";
900   INPUT A$
910   IF A$="Y" THEN F=0:GOTO 710
920   GOTO 10
1000  REM ********
1001  REM DIVISION
1002  REM ********
1010  CLEARW 2
1020  PRINT TAB(10)"INPUT THE LARGEST NUMBER"
1030  PRINT
1040  PRINT TAB(10)"FOR DIVISION"
1050  PRINT
1060  PRINT TAB(10);:INPUT"LARGEST";GR
1070  REM
```

```
1080  REM CALCULATE RANDOM NUMBER
1081  REM
1085  RANDOMIZE 0
1090  A1=INT(GR*RND)+1
1100  A2 = INT(GR*RND)+1
1109  REM
1110  REM CALCULATE DIVISION
1111  REM
1120  ER =A1*A2
1130  CLEARW 2
1140  PRINT
1150  PRINT "WHAT IS "ER" / " A1 "= ";
1160  INPUT ES
1170  IF ES=A2 THEN PRINT:PRINT TAB(10)"CORRECT !"
      :F=0:GOTO 1260
1180  PRINT :PRINT TAB(10)"WRONG!"
1190  FOR I = 0 TO 2000:NEXT
1200  F=F+1
1210  IF F<=2 THEN 1130
1220  PRINT
1230  FOR I = 0 TO 2000: NEXT
1240  PRINT TAB(5)" THE ANSWER IS "A2
1250  FOR I= 0 TO 3000:NEXT
1260  PRINT TAB(5)"ANOTHER PROBLEM Y/N";
1270  INPUT A$
1280  IF A$="Y" THEN F=0: GOTO 1090
1290  GOTO 10
1300  REM **************
1301  REM MULTIPLICATION
1302  REM **************
1310  CLEARW 2
1320  PRINT TAB(10)"INPUT THE LARGEST NUMBER"
1330  PRINT
1340  PRINT TAB(10)"FOR MULTIPLICATION"
1350  PRINT
1360  PRINT TAB(10);:INPUT"LARGEST";GR
1370  REM
1380  REM CALCULATE RANDOM NUMBER
1381  REM
1385  RANDOMIZE 0
1390  A1=INT(GR*RND)+1
1400  A2 = INT(GR*RND)+1
1409  REM
```

```
1410  REM CALCULATE MULTIPLICATION
1411  REM
1420  ER =A1*A2
1430  CLEARW 2
1440  PRINT
1450  PRINT "WHAT IS "A1" * " A2 " =";
1460  INPUT ES
1470  IF ES=ER THEN PRINT:PRINT TAB(10)"CORRECT !"
      :F=0:GOTO 1550
1480  PRINT :PRINT TAB(10)"WRONG!"
1490  FOR I = 0 TO 2000:NEXT
1500  F=F+1
1510  IF F<=2 THEN 1430
1520  PRINT
1530  FOR I = 0 TO 2000: NEXT
1540  PRINT TAB(5)" THE ANSWER IS "ER
1550  FOR I= 0 TO 3000:NEXT
1560  PRINT TAB(5)"ANOTHER PROBLEM Y/N";
1570  INPUT A$
1580  IF A$="Y" THEN F=0: GOTO 1390
1590  GOTO 10
1600  CLEARW 2
1610  END
```

We will now discuss the most important lines of this program listing. The MENU is displayed by lines 10 to 150. The MENU allows you to select from various choices.

Line 150 requests you enter a number, which represents a menu selection. Line 160 checks to see if a valid number was entered. This line is a good example of the use of a logical operator. If a number is either less than 1 *or* greater than 5, a branch is made to line 10. Only one of these conditions needs to be fulfilled, so the operator OR is used.

Line 170 illustrates the use of the ON...GOTO command. If Z has the value 1, a branch is made to line 200. If Z has the value 2, the program branches to line 600, and so on. The individual computation operators are selected by entering a number. Here, with the ON...GOTO command we saved 5 IF...THEN comparisons.

Lines 200 to 220 visually separate the program segment for addition. It is also a memory aid for the programmer.

In larger programs you learn to appreciate such REM statements, because they make the program much easier to read. You don't have to search through the entire program when the program needs to be changed.

Line 230 clears the screen. The next lines request a number to set the upper limit of the sums for addition. After this, two random numbers A1 and A2 are generated, from which the addition problem is formed. The problem is printed on the screen in line 370. The semicolons between text and variables are not necessary, as you can see in this example. In line 380 the result is taken from the user. Line 390 checks to see if the value entered matches the value calculated in line 340. If the value entered is false, a blank line is first created on the screen by line 400. Following this, the message WRONG! is printed on the screen. In line 410 a delay loop is processed so that the user can read the message.

Line 420 sets a counter that checks how many wrong answers the user has already given for this problem. Remember, the program displays the result after three incorrect answers. Line 430 checks to see if three wrong answers have been given. If this is not the case, the program branches to line 350 and redisplays the question again. If three incorrect answers were given, the program continues with the execution of line 440.

If the right answer was given in the meantime, the program jumps from line 390 to line 450. After the delay loop in line 450 runs out, the result is printed in line 460. After another delay loop, line 480 queries if another problem is to be asked. If the user enters Y here, the counter F is first set to zero and then a branch is made to line 300. There the two new random numbers required for the new problem are generated. If the user presses a key other than the Y key, the program jumps back to line 10, where the menu is reconstructed. The counter F must therefore be set back to zero, because the user must be allowed three attempts for the new problem. If we forget this, the old value of F is carried along. Accordingly, the result may be printed after one or two wrong answers. Remember this if you use counters in your own programs.

The other sections of the program—Subtraction, Division, and Multiplication—are constructed on the same principle. They differ only slightly in the creation of their respective problems. Let's take a look at subtraction.

Line 740 is special in its calculation of the result. This line ensures that we get only positive results by subtracting only smaller numbers from larger. If A1 is larger than A2, the problem statement in line 780 is correct. But if the reverse is true, the values in the variables must be exchanged or a larger number will be subtracted from a smaller in line 780. This is the purpose of line 740.

If A1 is smaller than A2, the value of A1 is stored temporarily in I. If we did the following:

```
A1=A2 : A2=A1        < WRONG
```

the value of A1 would be lost. Since A1 is first set equal to A2, variable A1 now contains the value of A2. After this we try to set A2 equal to A1, but A1 already has the value of A2. We don't exchange two variables in this way. First we must "save" one value—that is, store it in another variable.

First the variable I receives the value of A1:

```
I=A1
```

After this the variable A1 is assigned the value of A2:

```
A1=A2
```

Now we need:

```
A2=I
```

since I has the value of A1, and the exchange is complete. This temporary storage technique is very important. Make sure you understand this principle, so you can use it later in your own programs.

The ST has another simple solution for this problem—but it's not an instruction found on all computers. (If you adapt your programs to non-ST BASICs, you may have to use the previous solution). The instruction:

```
SWAP(X,Y)
```

exchanges the contents of the variables X and Y. You can replace the corresponding lines in the program with this instruction.

This point was the important one in the subtraction section. In the division segment a little trick was also used in order to get only integer results. In line 1120, as with the multiplication, the result of A1 and A2 is formed.

Then in the problem statement the result ER is divided by the value of A1. The result can only be an integer number since the result is generated from two whole numbers, namely A1 and A2.

The section dealing with multiplication has no special features. In construction, it is identical to the program segment for addition.

Before moving on to the self-test problems, here are some guidelines for using ON...GOTO:

1. The value that follows ON (which can be a number, a variable, or an arithmetic expression) determines the position of the line number in the list that follows the GOTO. The first line number is the branch destination for a value of 1, the second line number the branch destination for the value of two, and so on.

2. If this value is greater or less than the number of line numbers in the list, the next command, the one following the GOTO is executed.

3. Several IF...THEN comparisons can be grouped together with ON...GOTO.

3.3.2 Program jumps with ON...ERROR

This special type of the ON...GOTO command is used to automatically manage errors that occur in a program—in the program itself. A typical program line could look like this:

```
100 ON ERROR GOTO 1000
```

A small error-handling routine can be located at line 1000 to react to the error accordingly. But how do we know what error has occurred, and in which line number?

ST BASIC has two variables called *system variables* that are tested to tell us this information. These are the variables:

ERR and ERL

With ERR, you can determine the error code of the error. ERL is the line number in which the error occurred. The termination of a error-handling routine is designated by the command:

RESUME (*line number*)

With this RESUME command, you determine the program line at which the program is to continue execution after the error message. RESUME NEXT causes execution to continue with the command following the one that caused the error.

You also have the ability to let the ST error messages be printed from within the program. You must use the following command:

```
ON ERROR GOTO 0
```

Unfortunately, the ON ERROR GOTO command does not catch every error. You can check this on your ST with the following program:

```
10  ON ERROR GOTO 100
20  GOTO 40
30  FOR I=1 TO 20
40  PRINT I
50  NEXT
60  END
100 PRINT ERL, ERR
110 RESUME NEXT
```

When you RUN the program, it terminates with the error message:

```
You are trying to jump into a loop at line 20
```

despite line number 10. The error number as well as the number of the line containing the error are passed in the two variables ERL and ERR.

To see the values of these variables, enter in the command window:

```
PRINT "ERROR LINE"ERL;"ERROR LINE" ERR
```

and you'll see their values in the output window.

On the next page are some problems for your to solve. Remember to keep the five programming rules in mind when working with these problems.

Problems

1. Write a program that adds up the "harmonic series" `(1 + 1/2 + 1/3 + 1/4 + 1/5 + ... + 1/n)` up to a given number. After each 50 additions the number of additions should be printed. In conclusion, the number of required sums should be printed.

2. Write a program that calculates the real zeros of a quadratic equation of the form:

   $$AX^2+BX+C=0$$

   The solutions can be obtained from the following formula:

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

   There is no real solution for $B^2-4AC < 0$. Take this into account in your program.

3. What happens in the following program if the value 4 is entered for `Z`? Solve the problem without entering the program into the computer.

```
10 REM TEST OUTPUT WITH ON...GOTO
20 INPUT"ENTER A NUMBER";Z
30 ON Z GOTO 100,150,400:CLEARW 2
40 PRINT"ILLEGAL VALUE"
50 END
100 PRINT"LINE 100"
110 END
150 PRINT"LINE 150"
160 END
400 PRINT"LINE 400"
410 END
```

3.4 Reading the keyboard

ST BASIC has various options for reading the keyboard. You are already familiar with the simplest option: reading data using INPUT and assigning it to a variable.

The INKEY$ function is also provided, but does not work correctly in the current version of BASIC. If, for example, you enter the following program line:

```
   10 A$=INKEY$ : IF A$="" THEN 10
```

and start this little program, you will note that it does not end even if you press a key. Your only option is to break out of the program with <CONTROL> C.

Another option for selecting keys involves the use of INPUT$.

```
10 INPUT$(1)
20 A=ASC(A$)
30 PRINT A
```

The parameter in parentheses specifies how many characters you want to read in. In our example a character is read and its ASCII value determined. The advantage of this function is that the <RETURN> key need not be pressed.

All of the functions discussed so far have one disadvantage: they cannot read all of the keys on the keyboard. The cursor keys and the function keys do not transmit a value when these functions are used.

The function:

```
INP(2)
```

can help us here. The parameter 2 is a *device number*, and represents the keyboard. Valid parameters for INP are as follows:

0 = Printer
1 = RS-232
2 = Console (keyboard and screen)
3 = MIDI interface
4 = Keyboard processor

The following program shows you the codes of all the keys. You can exit the program by pressing the <ESC> key.

```
10 A=INP(2)
20 PRINT A,CHR$(A)
30 IF A=27 THEN END
40 GOTO 20
```

This covers all of the major functions for inputting characters from the keyboard. Now let's take a look at some commands and functions not used very often in programs. You should know what you're dealing with when you encounter such a command in a program.

3.5 FRE, POS, CALL, and WAIT

The above commands and functions are, as already mentioned, seldom used in BASIC programs. This has little to do with their importance. Let's take a look at them in order:

FRE

The FRE function is required to determine the amount of free memory space. The syntax of the function looks like this:

```
FRE(X)
```

If you want to know the free memory space in the computer, you can enter:

```
PRINT FRE(X)
```

in the direct mode. If there is no program in memory, you get the maximum free space as the value.

POS

You will seldom encounter this command within a program. It is used to determine the current cursor position on the screen. The following examples should explain more about the function. Enter into the computer in the direct mode:

```
PRINT "TEST" POS(X); "TEST A" POS(X)
```

and press the <RETURN> key. As output you get:

```
TEST 4 TEST A 13
```

The number 4 indicates the position of the cursor after the first execution of the print command. Correspondingly, the number 13 shows the cursor position after the second execution of the PRINT command. You can check this by entering the line again without the first POS command. You will then see that TEST A is printed directly after TEST, that the first character begins at the fourth position on the line. With POS you can determine the position in the line at that the next output with PRINT will take place.

CALL

This instruction lets you jump to an address in the computer at which a machine language program of your own or a system routine begins. The microprocessor is then no longer controlled by the BASIC commands via the interpreter. It is now controlled directly by machine code. In principle, you can address any memory location in the computer with CALL. There are many addresses that will cause the system to "crash," however. This instruction presumes a sound knowledge of the ST's operating system.

WAIT

Through the WAIT command you can cause a program to wait until a given memory location reaches a certain value. More specifically, the program waits with WAIT for a specific bit pattern in the memory location. This command is also seldom used.

3.6 READ, DATA, and RESTORE

Up until now we have discussed reading data from the keyboard. The data is stored in variables and then processed further.

If a program requires a large number of data, numerical values or strings, it is very tedious to have to enter these values each time the program is started. To get around this, you can use the READ and DATA statements.

The DATA statement is composed of a list of data items, where the individual values are separated by commas. The type of data that can be placed in a DATA statement can be either numerical or character.

With READ you can assign the individual data items in the DATA statements to variables. The variable type that follows the READ must correspond to the type of data contained in the DATA statement. You may not read a string into a numerical variable.

The DATA lines are not required to appear at any specific location within the program. They can be at the beginning, in the middle, or at the end. When the program encounters a READ command, it automatically searches for the DATA statement. Let's take a look at a simple example. First enter NEW —this command should **always** be used before you start a new program. Then enter the following program:

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

The program displays the the value 50. In line 10 the numerical variable X is assigned the value 50 with a READ commmand. If the program encounters the READ command, it searches for the corresponding DATA line and reads the first value. This value is assigned to the variable that follows the READ. In line 20 the contents of the variable X are printed. Line 30 has no further influence on the program. Change the program in the following manner:

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

After you have started the program you get the following output:

```
10        20        30
```

READ first assigned the first value in the DATA line to the variable X. Then it assigned the second value read to the variable Y, and then the third value to the variable Z.

Upon each READ access to the data in the DATA lines, the next value is always read. A pointer is maintained inside the computer that is always advanced each time an item read. This pointer always points to the next element to be read. At the start of a program this pointer points to the first element in the DATA line. The next lines should clarify this. The pointer is represented by the ↑ character.

```
30 DATA 10,20,30
        ↑
```

When the program encounters a READ, the pointer is incremented by one, and so points to the second element.

```
30 DATA 10,20,30
           ↑
```

When this element is read, the pointer is again incremented by one. When the pointer reaches the end of a list of DATA instructions, it is **not** automatically set back to the first element, but points after the last element. If you then try to access the list again with READ, the computer outputs the error message:

```
READ statement ran out of data at line (line #)
```

What if you want to access the data more than once? There is a solution for this. It's called:

```
RESTORE
```

The RESTORE command sets the pointer back to the very first element of DATA. This gives you the ability to read the data in the DATA lines as often as you like. Enter the following program in order to see what happens when the program tries to read 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
```

After the values 10, 20, 30 are printed, the error message:

```
READ statement ran out of data at line 40
```

appears. In line 40 an attempt was made to read the fourth element of DATA, an element that does not exist. To eliminate this error, you can either append three more values to the DATA statement or reset the pointer with RESTORE. Try this out once. Enter the following line in the computer and press the <RETURN> key.

```
35 RESTORE
```

Now enter the LIST command. Your program should look like this:

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

If you start the program now, the error message does not appear and you get the following output:

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

The pointer was again set to the first data element with the command RESTORE. Therefore the numerical variables D, E, and F were assigned the values 10, 20, and 30. The command:

```
READ A,B,C
```

causes three values to be read from the DATA line simultaneously. The values can also be read one at a time, of course, as the next example shows.

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

In this example, the commands READ X and PRINT X are placed in a FOR...NEXT loop that is executed a total of three times. On each pass through the loop, a new value is read from the DATA line, assigned to X, and printed.

As already mentioned, you can also put strings in the DATA lines. Normally these strings do not have to be placed in quotation marks.

As usual, there are exceptions here too. Any string with a comma must be placed in quotes. Remember that a string cannot be assigned to a numerical variable.

If you have a long list of mixed data, a wrong assignment can occur quite quickly. The following example should make the problem clear:

```
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 correctly through the second pass of the loop. Up to that point, the assigment of data values the variables that follow the READ command. According to READ, two numerical variables and then one string variable are to be READ. The order of the data in line 50 corresponds to the variable assignment behind READ, but only up to the seventh item—to the value 50. After this the program tries to read the string "TEST 3" into the numerical variable B. Since this is not possible, the program stops after two passes through the loop with the error message:

```
Function call not allowed at line 20
```

Be extremely careful when combining different variable types in a READ command.

We've learned a lot about how we can get data into the computer or program. We've also learned how to read in data from the keyboard with `INPUT`, `INPUT$`, and `INP`. The second method of inputting data is by statements so that they do not have to be manually input all the time. In this way, the data value contained in the `DATA` statements are saved as part of the program. To save data entered with `INPUT`, these values must be saved separately on a disk file.

One common use of this combination of `READ` and `DATA` is when a program in machine language is to be generated by BASIC. This is usually done by placing the numerical values of the machine codes in `DATA` statements, then reading them with a `FOR...NEXT` loop and writing them into memory with the `POKE` command. The machine language program is then started with the `CALL` command.

In the next chapter you will learn about how you write more complex programs in BASIC. The generation of arrays plays an important role in these programs. You will see that the commands `READ` and `DATA` will also have important applications.

Chapter 4

ADVANCED BASIC APPLICATIONS

4.1 Arrays
4.2 Subroutines
4.3 Menu techniques
4.4 Sorting methods

Advanced BASIC applications

4.1 Arrays

The programming and management of arrays is one of the most difficult concepts for the beginning BASIC programmer. The more complex the arrays, the more difficult it is to work with them. Even advanced programmers have problems with array management.

But a beginner can learn how to work with arrays. It's all a matter of practice. We'll begin with very simple examples.

4.1.1 One-dimensional arrays

Imagine that you want to write a program that calculates your average monthly salary. We start with 12 monthly totals. In this first example a loop is used to read in the amounts for the monthly totals. We can write a program as follows:

```
5   CLEARW 2:FULLW 2
10  REM AVERAGE MONTHLY INCOME
20  REM CALCULATE FOR 12 MONTHS
30  FOR I = 1 TO 12
40  PRINT "INCOME FOR MONTH" I;
45  INPUT M
50  S=S+M
60  NEXT I
70  D=S/12
80  D=INT(D*100)/100
90  PRINT "AVERAGE INCOME IS ";
100 PRINT "$ ";D
110 END
```

After you have entered this program into the computer, `RUN` it and enter 12 values. As the results you get the average monthly income, rounded off to two places after the decimal. In this program the individual monthly salaries are read in a FOR...NEXT loop with INPUT. The sum of the salaries is also formed within the loop (line 50). Line 70 calculates the average monthly income and the amount is rounded to 2 places in line 80. The program should be understandable to you.

We've now calculated the average monthly income. But what if we want to know later exactly how much money we made in May? In the last example the individual monthly values are lost.

There's no problem in solving that, you say. We'll use 12 variables instead of one and assign a monthly total to each one. Let's change the program:

```
5   CLEARW 2:FULLW 2
10  REM AVERAGE MONTHLY INCOME
20  REM SAVE INDIVUDAL MONTHS
30  INPUT "INCOME FOR MONTH 1";M1
40  INPUT "INCOME FOR MONTH 2";M2
50  INPUT "INCOME FOR MONTH 3";M3
60  INPUT "INCOME FOR MONTH 4";M4
70  INPUT "INCOME FOR MONTH 5";M5
80  INPUT "INCOME FOR MONTH 6";M6
90  INPUT "INCOME FOR MONTH 7";M7
100 INPUT "INCOME FOR MONTH 8";M8
110 INPUT "INCOME FOR MONTH 9";M9
120 INPUT "INCOME FOR MONTH 10";M10
130 INPUT "INCOME FOR MONTH 11";M11
140 INPUT "INCOME FOR MONTH 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 INCOME IS ";
190 PRINT "$ ";D
200 END
```

With these changes you have already moved a good deal closer to programming with arrays. If you now want to add to the program, you can refer back to the individual month values at any time. If, for example, you want to know your total income for the month of May, you need only read variable M5 and you have your answer—assuming that the month numbers correspond to the numbers behind the variable name.

We now have the month value available in our program, but at the cost of additional program lines. And you've got to admit that it's rather cumbersome having to work with 12 variables. If we had to output these values again, we couldn't do it in a loop since it involves 12 different variables. You would have to use a PRINT command for each individual variable or one PRINT command followed by all 12 variables. It would look like this:

```
.
.
100 PRINT M1,M2,M3,M4,M5,M6,M7,M8,M9,M10,M11,M12
.
.
```

This is hard to read and inelegant—a Quick-n-Dirty job, as we say in the business. It would be nice if we had a variable with a running index, something like this:

```
A(I)
```

This would make it possible to output the monthly values in a loop, and also access them through the specification of I. As you probably guessed, this arrangement is known as an *array*.

What do we mean by the term *array*?

Earlier in the book we compared a variable to a drawer that could hold numerical values or strings, depending on the type of variable. Imagine an array as a chest of drawers, with the drawers stacked on top of each other. Each drawer is designated with a number. This number has nothing to do with the actual contents of this drawer.

This number is called an *index*. This index is placed in parentheses and thereby is separated from the actual variable. We'll show you the notation again:

```
A(1)
```

Such a variable is called an *array variable* or *indexed variable*, since it is more accurately specified through the index. The index is the value in the parentheses.

Do not confuse this notation with the variable `A1`! There is a great difference between the two. The following picture should clarify the structure of such an array.

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

You see that such an array is very similar to a table (as in table of contents), in that the individual values are written under each other. Our array has 12 individual "drawers," each of which is assigned a value. If we want to know the contents of the third elements, we need only use the index 3. This could look like this:

```
PRINT A(3)
```

In our case we would get:

```
2345.65
```

as the output. If we want to use such arrays in our programs, we must first tell the computer how large our array is to be, i.e. how many elements it is to contain. This is the purpose of the `DIM` instruction in BASIC. It has the following syntax:

`DIM array name(`*number of elements*`)`

For our array we would have to use the following syntax:

```
DIM A(12)
```

A is the name of the array and 12 is the maximum number of elements. The DIM instruction is usually located at the beginning of a program. Once an array is dimensioned, it may not be redimensioned again in the program with DIM. Otherwise the computer will output the error message:

```
?You defined an array more than once
```

In our example, an array of floating-point variables was defined. You can also use arrays of string or integer variables. Take a look at the following examples:

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

These instructions create arrays of string, integer, and floating-point variables. You can also dimension several arrays at once with one DIM instruction. It looks like this:

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

The following facts about arrays must be noted.

1. If you need no more than 10 elements in an array, you don't have to use a DIM instruction. When you access an element, such as A(4), the computer automatically executes a DIM A(10) instruction.

2. DIM A(10) dimensions 11 elements in the array A(). The indices start with zero, not one, and consequently you have to count A(0) in the total as well.

The graphic representation of our array has to be expanded by the element A(0). We will not use this 0th element in our program, however.

Fact 1) states that dimensioning need not take place when 11 or fewer elements are used. If you know that you will need only 6 elements, for instance, omit the instruction DIM X(5) or DIM X(6) to save memory.

Let's take a look at the program that calculates the average monthly income using an array.

```
5   CLEARW 2:FULLW 2
10  REM AVERAGE MONTHLY INCOME
20  REM WITH ONE ARRAY
30  DIM M(12)
40  FOR I = 1 TO 12
50  PRINT "INCOME FOR MONTH";I;
55  INPUT M(I)
60  S=S+M(I)
70  NEXT I
80  D=S/12
90  D=INT(D*100)/100
100 PRINT "AVERAGE INCOME IS ";
110 PRINT "$ ";D
120 END
```

Using an array adds only one additional program line when compared to the original example. By adding one additional line, the monthly values are available in the rest of the program. For example, if you want to output the monthly values once more before the output of the average monthly income, you could change the program as follows:

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

By using an array we have made the monthly incomes available throughout the rest of the program—but we haven't made the program more complicated.

Arrays which have the following general syntax:

```
A(X)
```

are called one-dimensional arrays.This is because they have only one index.

The index does not have to be a constant. It can be a variable or a numerical expression. Imagine that you want to change our example program so that it will work with a variable number of months. The following modification to the program would be possible:

```
.
.
30 INPUT"HOW MANY MONTHS";N
35 DIM M(N)
40 FOR I=1 TO N
.
```

The number of months input will determine the size of the array. Use this little trick, you can adapt the program exactly to current requirements, as well as make optimal use of the memory of the computer.

If you try to access an element that lies outside the array dimensioned with DIM(X), the computer displays the following error message:

```
    Subscript refers to elements outside of array
```

If you have defined an array with DIM A(15) and try to access the element A(16), this error message would be printed.

We now want to practice working with some examples. We'll use the arrays X and Y$ with 6 elements each. We won't use the 0th element.

4.1.2 Examples of one-dimensional arrays

Normally you can assume that after the DIM statement the individual elements of the array are empty. But within a program you may have to clear an entire array. With numerical arrays you can do this by assigning the individual elements with a zero value. The following program shows how this is done.

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

We have assumed we are working with an array containing 6 (or 7) elements. The FOR...NEXT loop has the starting value 1 and the ending value equal to the maximum number of elements in the array—6 in our example. By running through this loop I takes the value 1, 2, 3, 4, 5, and 6 in succession. This causes all elements of the array to be set to zero, since the index of X is incremented each time. Written out this would look like this:

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

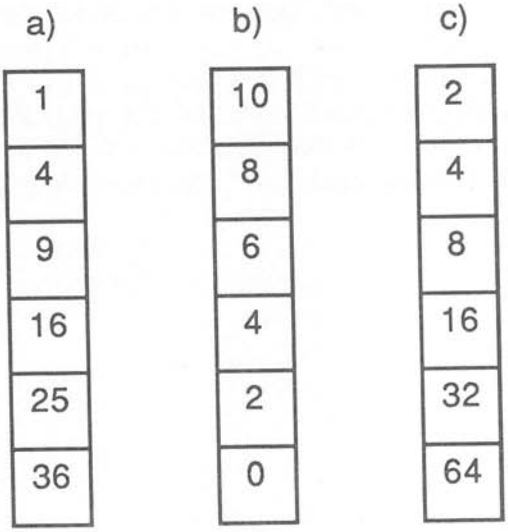
When you clear a variable containing a string, you must remember not to fill the strings with zeros. This is because the zero is treated as a character. You may be able to assign a space to each array element. But if you concatenate strings in your program, the space is included as one of the characters in the string. This can lead to errors in your program, such as when you use the LEN function, for example. Therefore, you should assign the elements of a string with "null strings."

It would look like this:

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

The method is the same as for clearing a numerical array. Note that the individual elements are assigned null strings in line 40. Make sure that there are no spaces between the two quotation marks.

Now we come to examples using the index in connection with FOR...NEXT loops. We assume three arrays with 6 elements each. The elements have the following contents:



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

How can these three arrays be used in a `FOR...NEXT` loop?

A little study will show that the elements of example a) exactly match the squares of the indices. The program could look like this:

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

The function of this loop will become more familiar if we write down the 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 = 4 : X(4) = 4*4 =  16      16
     I = 5 : X(5) = 5*5 =  25      25
     I = 6 : X(6) = 6*6 =  36      36
```

`I` is incremented by one each pass through the loop. `I` is multiplied by itself and the result is assigned to the element whose index is `I`. This fills the array with the squares of each value from 1 to 6.

The same principle of using the loop variable for calculation is used in example b) as well.

With example b) we decrease the values of the array elements by 2 with every increase of the index. The starting value of the first array is 10. To create this effect, we cannot change the FOR...NEXT loop. This is because the elements are accessed via the loop index. We must work out an assignment rule that decreases the value of the elements in steps of two with every increase of the index. The solution to this problem might look like this:

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

In line 30 we have our assignment rule. Again, we've incorporated the index into the calculation. If you run I from 1 to 6 in your mind, you see that it produces exactly the sequence of numbers in example b). You can check this by expanding the program with the following lines. The program will then output the entire array.

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

If you have difficulty understanding these programming solutions, use the method in example a). Try writing the course of the program down on paper.

Now to example c). You probably recognized a pattern here as well. The numbers are, of course, the powers of two. You probably remember these numbers from the section on number systems.

It shouldn't be any problem to find an assignment rule for this example. We use the 2 as a constant and the loop variable or index as the power. The program then looks like this:

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

Check the accuracy of this assignment by appending the program lines from example b). You can also work the process out on paper to make it clearer. With practice the principle of this technique will become quite familiar. And when you encounter such techniques or applications in more complex programs, you should be able to figure them out.

Up to now we have looked only at numerical arrays. Now we want to turn to string arrays. There is usually no pattern to the layout of the individual elements in string arrays. Assignments for the string arrays are often made by the user via the keyboard. But another possibility is using the commands READ and DATA to initialize an array.

String arrays can be used to store such data as names, addresses, or numbers that exist as strings. First we'll create an array in which we store our friends' first names in the computer.

Because we often don't know the total number of friends, the size of the array is unspecified. We'll have to specify the size of the array. But, if the capacity of the array is reached, the program will output a message on its own, instead of an error message. Our example will show how such a program is created:

```
5   FULLW 2:CLEARW 2
10  REM LIST OF FIRST NAMES
20  DIM Y$(6)
30  Z=Z+1
40  INPUT "FIRST NAME";Y$(Z)
50  PRINT "MORE INPUT Y/N ?"
60  A$=CHR$(INP(2))
70  IF A$ <> "Y" THEN 100
80  IF Z < 6 THEN 30
90  PRINT "LIST IS FULL!"
100 END
```

To make the program easier to read, an array of only 6 (or 7) elements is used here (line 20). Line 30 contains the counter. This counter is incremented by one for each input. Since we don't know the exact number of names, we cannot use a FOR...NEXT loop. Line 40 asks for the entry of a name with INPUT and assigns this to the element with the index of Z. In line 50 you're asked if additional input is yet to be made. The function line 60 should be familiar to us. Line 70 compares the character entered to Y. If the character is not equal to Y the program is ended at line 100. If more input is to take place, line 80 compares the counter to 6 to see if it is smaller. If the counter already has the value 6, the message LIST IS FULL! is printed in line 90, and the program is ended. If we had not incorporated this check, the program would have terminated on a value greater than 6 with the error message:

```
Subscript refers to element outside the array in
                    line 40
```

If Z reaches the value 7, line 40 attempts to access the element Y$(7). Because of the DIM statement, this element does not exist. Unwanted program interruptions such as this should be avoided whenever possible. The same effect could be achieved with ON ERROR GOTO and a corresponding error-handling routine.

The program still doesn't do a whole lot, but the principle should become clear. We could now add a question to see if the user wants to output the whole list. You can make this enhancement yourself by incorporating the appropriate IF...THEN test into the program. You can then display the contents of the array with a FOR...NEXT loop.

Dimensioning an array with DIM D$(200) can probably handle most requirements. But a one-dimensional array is probably sufficient for all your programming needs.

First we come to an example where an array is filled with READ and DATA commands. And imagine that you need the days of the week in your program. It would require a lot of work to enter this information each time the program was started. Why not put this data in DATA lines, then read them into an array at the program's start with READ? Take a look at the following example program:

```
5   FULLW 2:CLEARW 2
10  REM 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 YES/NO
80  PRINT "OUTPUT THE ARRAY Y/N"
90  A$=CHR$(INP(2))
100 IF A$<>"Y" THEN 140
110 FOR I = 1 TO 7
120 PRINT WD$(I)
130 NEXT I
140 END
```

This program is very similar to the one for the name list. The major difference is found in line 40, where the data is read with READ instead of INPUT. The DATA line is self-explanatory. At the conclusion of the program is an option to output the entire array.

You've now learned how to transfer data to an array with the READ and DATA commands. Before we turn to two-dimensional arrays, brush up on your knowlege of one-dimensional arrays with the problems on the following page.

Problems

1) Write a program that reads six names and places them in an array. Furthermore, the program should output the name that would come first in an alphabetical listing. Test the program with the names Thomas, James, Russell, Julie, Arnie, and Janet. Remember that strings can be compared with each other to see if they are equal, less than, or greater. The result will be the name "Arnie."

2) Write a program that creates 6 random numbers and places these numbers in an array. The largest of these numbers should be printed. The random numbers should occur in the range 50 to 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 and develop an assignment rule that creates this array. Output the array as a check. Don't worry that we listed the array elements horizontally this time. This isn't important when you're working with one-dimensional arrays.

4.1.3 Multi-dimensional arrays

Up to now we have used only one-dimensional arrays. We compared these arrays to a drawer in which the data elements were stacked one above the other. These drawers or lists usually don't consist of horizonal or vertical data, but a combination of the two. They are composed of rows and columns. Imagine that you wanted to expand the program that read a person's first name into an array, so that the last names and the birthdates were also available under the same index.

One way to do this is create three arrays in which the data can be stored. Array `A$(X)` can hold the first names, array `B$(X)` the last names, and array `C$(X)` the birthdates. You have created three arrays with different names. These arrays can be easily filled with data, but working with them within the program is rather complicated. We therefore have the same problem we had at the introduction to one-dimensional arrays.

Why shouldn't it be possible to use just one array instead of three? The solution to our problem is called a *multi-dimensional array*.

In this particular case, we need a two-dimensional array. That's because we want to store the individual first names in the same row as the data for the last name and birthdate. Therefore, our array requires rows and columns. The structure of such an array looks like this:

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

The DIM statement for this array is:

```
DIM A$(5,3)
```

This reserves an array with 5 lines and 3 columns. (This DIM statement actually genetrates an array of 6 lines and 4 columns. However, we will not use the 0th elements).

If you do not execute the DIM statement and use one of the elements from this array, the computer automatically creates an array of size (10,10). It's wise to use the DIM statement for multi-dimensional arrays, because you can save quite a bit of memory space.

How is such an array used? Imagine that you want to fill the first three columns of the first line of this array with data. You could use the following program line:

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

This requires the input of three elements (first name, last name, birthdate), separated from each other by commas. However, this command is rather hard to read within a program. Instead, we should use a total of three INPUT commands written on three separate program lines. These lines look like this:

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

This assignment is easier to read. It also helps to avoid input errors, because there is a prompt for the input of each element.

Why isn't the input made in a loop—where the first name, last name, and birthdate are read in succession with just one INPUT command? In our example, each INPUT command has its own prompt describing the value to be entered. This is why the three INPUT commands cannot be replaced by a single INPUT, so a loop cannot be used.

We want to read exactly six first names, last names, and birthdates. We can use a FOR...NEXT loop for this, as the following example shows:

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

You can find program segments similar to this in all small data management programs.

Data for multi-dimensional arrays is not input from the keyboard only. Data from DATA statements read with the READ command can furnish data for a program as well. You are familiar with this technique from the one-dimensional arrays. We can use nested FOR...NEXT loops in programs of this type. The following example shows how a two-dimensional array of size (3,4) would be filled with data from DATA statements:

```
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 A$=CHR$(INP(2))
110 IF A$ <> "Y" THEN 170
```

```
120 FOR R=1 TO 3
130 PRINT X(R,1);X(R,2);X(R,3);X(R,4)
140 NEXT R
170 END
```

This example has an array with 3 rows and 4 columns which will be filled by two nested FOR...NEXT loops. The inner loop (40 FOR C=1 TO 4) causes all elements in a row to be filled. Once this loop is completed the outer loop (30 FOR R=1 TO 3) fills all three rows in succession. The following figure shows how the array is filled with data.

```
                         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
```

If you want to output the array, you need only press the Y key. The array will be printed in the form you see above. This output is accomplished with line 130. All four columns are printed on one line simultaneously. Only the output of the three rows is accomplished by the FOR...NEXT loop.

The next example shows you another way of outputting the array with this technique (the first lines of the program are omitted):

```
.
.
80  REM ARRAY OUTPUT
90  PRINT"DISPLAY ARRAY (Y/N)?"
100 A$=CHR$(INP(2))
110 IF A$ <> "Y" THEN 170
120 FOR R=1 TO 3
130 FOR C=1 TO 4
140 PRINT X(R,C);:ZZ=ZZ+1
150 IF ZZ=4 THEN ZZ=0:PRINT
```

```
150 IF ZZ=4 THEN ZZ=0:PRINT
160 NEXT C,R
170 END
```

You can use two nested FOR...NEXT loops if you modify your program this way. The semicolon in line 140 causes the values of the individual elements to be printed one after the other. In order to construct the structure of the array on the screen, a new line must be started after each four elements are printed. This is the reason for the additional counter ZZ. This counter registers how often an output has been made with PRINT. Line 150 tests to see if this counter has the value 4. If this is the case, the counter is set back to zero and another PRINT command is executed, causing the next element to appear at the start of the next screen line. You can make the output easier to read by using two PRINT commands, as in our example.

We now come back to the first example. We said that similar program segments were found in data management programs. In this example, we assumed that we would record the data of only 6 people. But the total number of people is rarely known ahead of time. Usually we know only the number of data elements for each person to be recorded—the last name, first name, and telephone number, for instance.

Say you want to write a program to replace your telephone directory. In most cases, you don't know the number of persons, only the number of data elements. So you can only approximate your dimensions. Let's assume that you want to record about 100 telephone numbers. The DIM statement X$(120,3) should be sufficient in this case. Here's an example:

```
10  REM READ IN DATA
20  DIM X$(50,3)
30  FULLW 2:CLEARW 2
40  Z=Z+1
50  INPUT "FIRST NAME";X$(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 A$=CHR$(INP(2))
140 IF A$="Y" THEN 30
150 END
```

The preceding program could be solved more elegantly, of course. We just want you to understand the principle used here.

Since we don't use a `FOR...NEXT` loop, the counter in line 40 must be inserted to increment the index by 1 for each new input. The data is then read with `INPUT` commands that assign the data to the appropriate elements. If additional data is to be entered, the program branches to line 30. Otherwise the program is ended. This is where a jump to a main menu in a data management program might occur—the place where the user could then select additional options.

A comparison is made in line 130 with `IF...THEN` to see if more data is to be entered. This is different from the first program, where this task was assumed by a `FOR...NEXT` loop.

Remember: We use `IF...THEN` when the exact number of data is *not* known ahead of time.

These examples show how to use multi-dimensional arrays. The ST can create arrays with up to 15 indices. This means that not only two-, three-, or four-dimensional arrays can be created, but arrays with as many as fifteen dimensions. Arrays beyond three dimensions become more difficult to picture graphically, but this does not mean that they can't be used in solving certain problems.

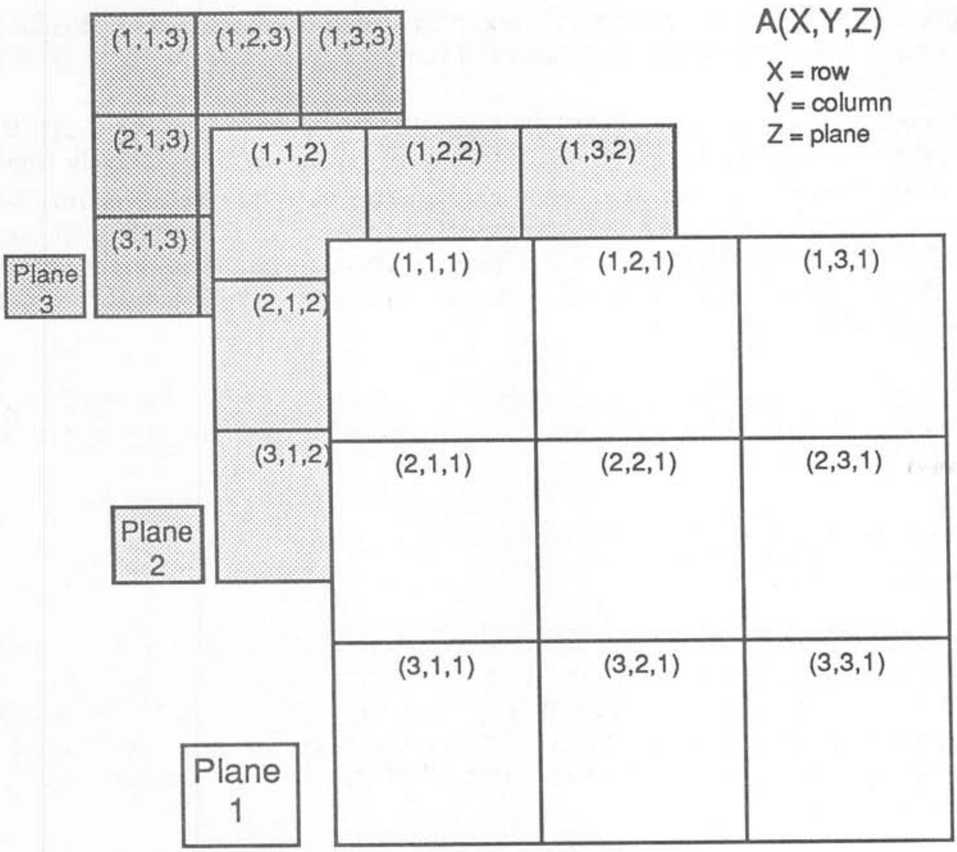
Let's take a look at an example of a three-dimensional array. We will define the indices as follows:

`X` = Row

`Y` = Column

`Z` = Depth

We will now create a three-dimensional array that we'll represent with 3 stacked planes. This three-dimensional array can best be imagined as three two-dimensional arrays, one after the other. The following illustration should clarify this concept:



To create this array, the `DIM` statement looks like this:

```
DIM C(X,Y,Z)
```

or, as in our example:

```
DIM A(3,3,3)
```

This array has a total of 27 individual elements (3*3*3=27). Actually, this array has 64 elements if we count the zero elements (4*4*4=64) .

Your problem now is writing a program to fill this array with data. The number of data is known. Assume that, in this 3D array, first the column values of a row will be filled, then the rows themselves (as for a two-dimensional array), and finally the individual "planes". Write the program to create the array shown in the illustration above. The value assignments of the individual planes is intended. You don't have to create the spatial effect in the output.

Our solution follows directly this time. Your program should 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
```

You've got to remember the initial `DIM` statement, as in line 20. What applies to a two-dimensional array also holds true for a three-dimensional array. If you forget the `DIM` statement, the computer will create an array of 10*10*10=1000 elements, or 11*11*11=1331 elements! That would be an enormous waste of memory space—you only need 27 elements.

Take a look at the three `FOR...NEXT` loops, opened in lines 30 through 50. The outer loop causes the individual planes of the cube to be filled. The other two loops fill a two-dimensional slice of the array. Line 70 closes all three loops at once. Make sure that the loop variables are placed after the `NEXT` in the proper order. The array is not printed in a spatial representation, but the "planes" should be recognizable.

This concludes the section on multi-dimensional arrays. You should now be able to use either one-dimensional or multi-dimensional arrays in your programs. One and two-dimensional arrays are the ones most often used in programs. The predominant application of arrays is data management.

4.2 Subroutines

A *subroutine* is a program segment that replaces several often-used, similar program sections. A subroutine is an independent segment of a program, and is usually located at the start or end of the the main program. Using a subroutine is made with the command:

GOSUB (*line number*)

GOSUB is a an abbreviation for GOto SUBroutine. The line number indicates the location where the subroutine begins. When the computer encounters this program command, it makes note of the line number from where it's branching. Then it branches to the line number of the subroutine. The computer continues with program execution there until it encounters the return command:

```
RETURN
```

Program execution then returns to and continues with the statements following the GOSUB command. If the program encounters the RETURN command without first having received the GOSUB command, the program stops with the error message:

```
RETURN statement needs matching GOSUB at line
                    (line #)
```

Whenever you call a subroutine you must use the GOSUB command. A common error is branching to a subroutine with the GOTO command. This is often done with IF...THEN, as in the following:

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

In this example, a branch is made to a subroutine with a GOTO in line 110 (if A is less than 1). When GOTO is improperly used in such a subroutine, the program would terminate with the error message:

```
RETURN statement needs matching GOSUB at line 150
```

The proper syntax of line 110 must be:

```
110 IF A < 1 THEN GOSUB 130
```
< **CORRECT**

The following example shows another error. This error is difficult to recognize in larger programs using subroutines:

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

Enter this program and start it. After the 15th or 16th execution of the subroutine, the program will terminate with the error message:

```
?You've nested subroutine calls too deep at line 40
```

The error lies in the call to the subroutine. The program creates a blank line after the start (line 20). Then the current value of the variable is printed—Z serves as a counter for the number of times the subroutine is executed.

Line 40 makes the call to the subroutine with `GOSUB 70`. The program jumps to the subroutine in line 70, and starts execution of the `FOR...NEXT` loop. Line 90 outputs the value of the loop variable `I`.

Line 100 breaks two rules at once. First, you should never exit a `FOR...NEXT` loop with `GOTO`. This can lead to problems with the internal management of this loop. The second error is the more serious error of the two: you *cannot* exit a subroutine without using the `RETURN` command. Line 100 requires a jump out of the subroutine, for the case in which `I` is greater than or equal to 15. Instead of `THEN 50`, we should write `THEN RETURN`.

You can branch back and forth *within* a subroutine with GOTO as with normal programs. However, you cannot simply *exit* a subroutine with GOTO. In our example, the program stops after the 16th execution of the subroutine.

If you make an error like this in any of your programs, it may appear to run correctly at first. Then, under certain conditions it suddenly will crash. For this reason you should always check your subroutines for this error. A program carefully thought out and well planned will help you avoid such errors.

Now we come to a practical application of subroutines. You might recall the program MATH TUTOR. Look through this program listing for program segments that are repeated in the program. You will probably discover that some program segments could be combined into subroutines. The lines that are often repeated are listed again here:

```
10  REM THIS LINE IS NEEDED TO BE ABLE TO RUN
230 CLEARW 2
240 PRINT TAB(10)"INPUT THE LARGEST"
250 PRINT
260 PRINT TAB(10)"NUMBER FOR ADDITION."
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 ER=A1+A2
350 CLEARW 2
360 PRINT
370 PRINT "HOW MUCH IS" A1 "+" A2 "= ";
380 INPUT ES
390 IF ES=ER THEN PRINT:PRINT TAB(10)"CORRECT!"
    :F=0:GOTO 470
400 PRINT:PRINT TAB(10)"WRONG."
410 FOR I=0 TO 2000:NEXT I
420 F=F+1
430 IF F<=2 THEN 350
```

```
440 PRINT
450 FOR I=0 TO 2000:NEXT I
460 PRINT TAB(5)"THE CORRECT ANSWER IS" ER
470 FOR I=0 TO 3000:NEXT I
480 PRINT TAB(5)"ANOTHER PROBLEM Y/N";
490 INPUT A$
500 IF A$="Y" THEN F=0:GOTO 300
510 GOTO 10
```

These repeated program lines can be grouped into specific blocks. The first block is the lines from 230 to 290, which requires the input of the largest number for the calculation type. The problem with this segment is that each calculation type for which the largest number is read must be named. The output:

```
        ENTER THE LARGEST NUMBER FOR ADDITION
```

must be made more flexible in the subroutine, specifically regarding the type of calculation.

We access the individual program segments such as addition, subtraction, etc. with `ON X GOTO` in the menu. Therefore, it's advisable to use `X` as an index. We'll see why we do this shortly.

At the start of the program we can create an array that contains the terms addition, subtraction, division, and multiplication in the order in which these terms appear in the menu. We can also output the menu itself with the contents of this array. Let's look at the modified program start:

```
10   REM *****************
20   REM * PROGRAM START *
30   REM *****************
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
.
.
.
```

In line 50 the two arrays RA$ and BE$ are generated. The two arrays are loaded with the FOR...NEXT loop in line 60. The array RA$ is initialized with the terms addition, subtraction, etc. The array BE$ is initialized with the corresponding characters of the calculation types. After the program is started, the arrays are filled with this data.

Let's take a look at lines 570 to 790 of the modified program to see how the array RA$ is used to construct the menu.

```
.
.
.
570    REM **********
580    REM *  MENU   *
590    REM **********
600    CLEARW 2: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    E$=CHR$(INP(2))
780    P=VAL(E$):IF P < 1 OR P > 5 THEN 770
790    ON P GOTO 800,890,990,1090,1180
.
.
.
```

In contrast to the previous version of the program, the menu selections addition, subtraction, etc. are not mentioned individually, but are read from the array RA$ in lines 660 to 720. The array elements were accessed individually here for the sake of readability. You could also achieve this with a FOR...NEXT loop. The following example illustrates this:

```
.
.
660    FOR I=1 TO 4
670    PRINT TAB(12)I" FOR "RA$(I)
680    PRINT
690    NEXT I
700    REM *** DELETE LINES 700 - 730 ***
.
.
```

When a character is read in line 770, it is first checked in line 780 to see if it is a valid character (a digit between 1 and 5). If this is the case, the numerical value is determined with `VAL(E$)` and assigned to the variable `P`. Branches are made to the appropriate lines based on `P`. If addition is selected, `P` has the value 1. This causes a branch to line 800. There begins the program segment for addition. Let's take a look at the program lines for this:

```
.
.
800 REM ************
810 REM * ADDITION *
820 REM ************
830 GOSUB 110
840 GOSUB 310
850 ER=A1+A2
860 GOSUB 390
870 IF A$="Y" THEN 840
880 GOTO 580
.
.
```

In line 830 the first subroutine is called. This is the segment in which the highest number to be used in the calculations is entered. The next lines constitute this first subroutine:

```
.
.
110 REM **************
120 REM * SUBROUTINE *
130 REM **************
140 REM ************************
150 REM * INPUT LARGEST NUMBER *
160 REM ************************
170 CLEARW 2:A$="":B$=""
180 PRINT TAB(10)"INPUT THE LARGEST NUMBER FOR"
190 PRINT
200 PRINT TAB(10)"FOR "RA$(P)"."
210 PRINT
220 PRINT TAB (10)"LARGEST NUMBER?"
230 FOR I=1 TO 3
240 A$=CHR$(INP(2))
250 IF ASC(A$)<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 should be familiar. But why are the variables A$ and B$ set to null in the second half? Since B$ is combined with A$ in line 260, it always "drags" its contents along. If the calculation type was changed and this subroutine called again, the old contents of the variable would be appended to the newly-read values. This would give a six-digit number for the calculation of the random numbers. This is why the two variables are set to null at the start of the subroutine.

The input of the three digits is done with INP (see the section on data input with INP). In our current example, a couple of changes are necessary. For one, only digits between 0 and 9 can be entered (line 250). The digits entered are displayed in line 260 with PRINT A$. If you want to use just a two-digit number, you must first enter a zero and then the remaining digits.

Line 200 is quite interesting. Here the value of P is used as an index. This index enables the computer to read the suitable calculation type from the array RA$. You can see that it makes a lot of sense to use indexed variables.

We assume that the data was placed in the array in the proper order. If we selected addition, `P` has the value of 1. The term "addition" is located in `RA$(1)`. For this reason, the character for calculation type is also placed in a different array with the same index. With this little trick we have made our subroutine match each of the four calculation types.

The next program line for addition, line 840, calls the subroutine for the creation of the random numbers. This is the smallest 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
.
.
```

No further explanation should be needed for this subroutine.

More important is the next subroutine, `PROBLEM SET-UP`. Here are the lines for the following subroutine:

```
.
.
360 REM *******************
370 REM * PROBLEM SET-UP *
380 REM *******************
390 CLEARW 2
400 PRINT
410 PRINT "HOW MUCH IS";A1;BE$(P);A2;"= ";
420 INPUT ES
430 IF ES=ER THEN PRINT:PRINT TAB(10)"RIGHT!"
    :F=0:GOTO 500
440 PRINT:PRINT TAB(10)"WRONG!!"
450 FOR I=1 TO 2000: NEXT I
460 F=F+1
470 IF F<=2 THEN 390
480 PRINT
490 FOR I=1 TO 2000: NEXT I
```

```
500 PRINT:PRINT TAB(5)"THE ANSWER IS"ER
510 F=0
520 FOR I=1 TO 3000:NEXT I
530 PRINT
540 PRINT TAB(5)"ANOTHER PROBLEM Y/N";
550 INPUT A$
560 RETURN
.
.
```

Lines 390 and 400 need no explanation. Line 410 of this subroutine is interesting. Here the statement of the problem is formulated. First the three words:

```
HOW MUCH IS
```

are printed. Then the values of the variables A1, BE$(P), and A2 are printed. These variables are followed by an equal (=) sign. We could formulate the general form of this screen output like this:

```
HOW MUCH IS A1 + (or -,*,/) A2 = ?
```

The proper calculation character BE$(P) is printed, depending on the calculation type selected via the index P.

We must ensure that A1 and A2 always contain the *correct* values independent of the calculation type. By *correct* values we mean than the answer is always a positive whole number. We must prepare the values of the variables A1 and A2 in the individual program selections of addition, subtraction, etc., so that our subroutine remains valid for all calculation types.

The remaining program lines to line 540 should be familiar to you from the original MATH TUTOR program. In line 550, A$ accepts the answer from the question in line 540. Line 560 ends the subroutine with the RETURN command. The contents of variable A$ are processed in the program segments corresponding to the individual calculation types.

This concludes the program segment containing the subroutines. You've no doubt noticed that we placed the subroutines at the start of the program. Many books advise you to place the subroutines at the end of the main program. Our decision was made in consideration of a program library.

This example sorts subroutines according to line numbers. This is so the subroutine for rounding a number is a line 50000, and so on. If you write a new program, you can load these routines with a special command and append them to the program. This MERGE instruction is usually offered as a BASIC extension.

Why is it advantageous to give the subroutines low line numbers, thereby placing them at the start of the program? It doesn't matter whether a subroutine is appended to a program or the program is appended to the subroutines. The results are equivalent.

A subroutine causes the computer to branch to the start of a program. There it begins searching for the line number at which the subroutine starts. If the subroutines are located at the start of the program, the execution time of the program is reduced. For small programs this will hardly be noticeable, if at all. But once the program has reached a moderate length, this trick will make the program several seconds faster.

We said that the variables A1 and A2 for the subroutine PROBLEM SETUP must be prepared in the individual program segments for the calculation type. This applies only to subtraction and division, since the values for the variables from addition ER=A1+A2 and multiplication ER=A1*A2 can be passed directly to the subroutine.

For subtraction, you must make sure that A1 is always larger than A2, to ensure that no negative values arise. This is accomplished with the program line 940. If A1 is smaller than A2, the two values are exchanged in temporary storage.

When division takes place, we want only integer results. For this reason the result ER is first calculated by multiplying the variables A1 and A2. In the original program MATH TUTOR, we could state the problem as follows:

```
HOW MUCH IS ER / A1 = ?
```

The previously-calculated result ER is divided by the variable A1. This must lead to an integer value, namely the variable A2.

Our subroutine cannot use this method directly, since it is generalized for all four arithmetical computations. This must be done in the Division program section. The following program lines are used:

```
.
.
1040 ER=A1*A2
1050 I=ER:ER=A1:A1=I
.
.
```

Here too the result is first calculated with the multiplication. Line 1050 assigns the correct values for the subroutine PROBLEM SETUP. Since we can't switch the variable designations when the variable is output, we must reassign the values of the variables.

To accomplish this, the technique of temporary storage is used. The variable I initially contains the value of the variable ER. ER is then assigned the value of A1. Finally, A1 contains the value of I—that is, the old content of ER. The contents of the two variables are exchanged in this manner. This is how we get the proper assignments in the problem statement subroutine.

Now we'll list the complete program:

```
10   REM *****************
20   REM * PROGRAM START *
30   REM *****************
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
110 REM **************
120 REM * SUBROUTINE *
130 REM **************
140 REM ************************
150 REM * INPUT LARGEST NUMBER *
160 REM ************************
170 CLEARW 2:A$="":B$=""
180 PRINT TAB(10)"INPUT THE LARGEST NUMBER FOR"
190 PRINT
200 PRINT TAB(10)"FOR "RA$(P)"."
210 PRINT
220 PRINT TAB (10)"LARGEST NUMBER?"
230 FOR I=1 TO 3
```

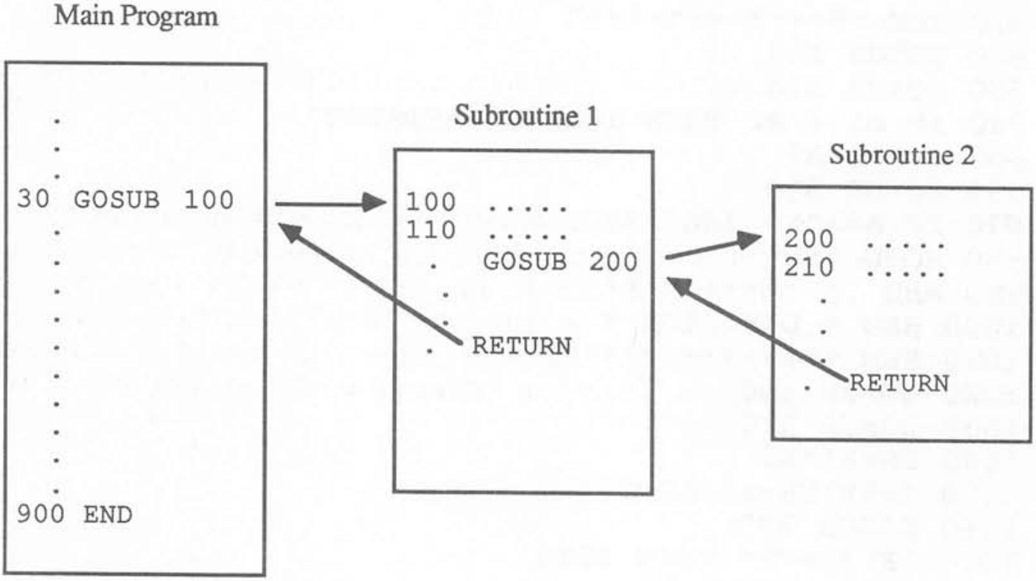
```
240 A$=CHR$(INP(2))
250 IF ASC(A$)<48 OR ASC(A$)>57 THEN 240
260 B$=B$+A$:PRINT A$;
270 NEXT I
280 GR=VAL(B$)
290 RETURN
300 REM *************************
310 REM * CREATE RANDOM NUMBERS *
320 REM *************************
330 A1=INT(GR*RND(1))+1
340 A2=INT(GR*RND(1))+1
350 RETURN
360 REM ******************
370 REM * PROBLEM SET-UP *
380 REM ******************
390 CLEARW 2
400 PRINT
410 PRINT "HOW MUCH IS";A1;BE$(P);A2;"= ";
420 INPUT ES
430 IF ES=ER THEN PRINT:PRINT TAB(10)"RIGHT!"
    :F=0:GOTO 500
440 PRINT:PRINT TAB(10)"WRONG!!"
450 FOR I=1 TO 2000: NEXT I
460 F=F+1
470 IF F<=2 THEN 390
480 PRINT
490 FOR I=1 TO 2000: NEXT I
500 PRINT:PRINT TAB(5)"THE ANSWER IS"ER
510 F=0
520 FOR I=1 TO 3000:NEXT I
530 PRINT
540 PRINT TAB(5)"ANOTHER PROBLEM Y/N";
550 INPUT A$
560 RETURN
570 REM **********
580 REM *  MENU  *
590 REM **********
600 CLEARW 2:F=0
610 PRINT
620 PRINT TAB(12)"MATH TUTOR"
630 PRINT:PRINT
640 PRINT TAB(12)"CHOOSE:"
650 PRINT
```

```
660 FOR I=1 TO 4
670 PRINT TAB(11)I" FOR "RA$(I)
680 PRINT
690 NEXT I
740 PRINT TAB(12)"5 TO END"
750 PRINT
760 PRINT TAB(12)"WHICH NUMBER?"
770 E$=CHR$(INP(2))
780 P=VAL(E$):IF P < 1 OR P > 5 THEN 770
790 ON P GOTO 800,890,990,1180
800 REM ************
810 REM * ADDITION *
820 REM ************
830 GOSUB 110
840 GOSUB 310
850 ER=A1+A2
860 GOSUB 390
870 IF A$="Y" THEN 840
880 GOTO 580
890 REM ***************
900 REM * SUBTRACTION *
910 REM ***************
920 GOSUB 110
930 GOSUB 310
940 IF A1 < A2 THEN I=A1:A1=A2:A2=I
950 ER=A1-A2
960 GOSUB 390
970 IF A$="Y" THEN 930
980 GOTO 580
990 REM  ************
1000 REM * DIVISION *
1010 REM ************
1020 GOSUB 110
1030 GOSUB 310
1040 ER=A1*A2
1050 I=ER:ER=A1:A1=I
1060 GOSUB 390
1070 IF A$="Y" THEN 1030
1080 GOTO 580
1190 REM ******************
1100 REM * MULTIPLICATION *
1110 REM ******************
1120 GOSUB 110
```

```
1130 GOSUB 310
1140 ER=A1*A2
1150 GOSUB 390
1160 IF A$="Y" THEN 1130
1170 GOTO 580
1180 REM *******
1190 REM * END *
1200 REM *******
1210 CLEARW 2
1220 END
```

By using three subroutines we saved 43 program lines—despite our using more `REM` statements! You can see that it's not only *easier* to use subroutines, but it also helps save memory space. Also, you've no doubt recognized the purpose of line 100. The subroutines are skipped and a branch is made directly to the menu.

Not only is it possible to call the subroutines from the main program, it is also possible to call them from a subroutine. Graphically this would look as follows:



When the program is run, this routine works as follows:

The program encounters the GOSUB command in line 30. It then jumps to Subroutine 1 at line 100. While in Subroutine 1, a GOSUB 200 calls Subroutine 2. Subroutine 2 is executed until the RETURN command.

The RETURN jumps the program back to Subroutine 1, where it continues execution with the instruction following the GOSUB. The Subroutine 1 is then executed through the RETURN command. From there, execution returns to the main program, i.e. the instruction following the GOSUB (line 40).

Remember: You can "nest" subroutines in much the same way as you nest FOR...NEXT loops.

Furthermore, we have already learned the ON X GOTO command. This command sequence also works with GOSUB. Here is an example of the notation:

```
ON P GOSUB 800,890,990
```

Remember that after the jump to the subroutine, the program continues on at the line after this statement.

We hope that the examples given have clarified the technique of using subroutines. To see how well you have understood all this, try soving the following:

There's another way of using subroutines in the new version of MATH TUTOR–the ON GOSUB command. Your goal is to modify the program in this form. You don't have to rewrite the program for this. First consider which program segments must be changed. You don't have to place the new subroutines at the beginning of the main program this time. Again we have placed the solution to this problem following so that we can discuss the solution now.

Solution

```
10    REM *****************
20    REM * PROGRAM START *
30    REM *****************
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
110   REM **************
120   REM * SUBROUTINE *
130   REM **************
140   REM ************************
150   REM * INPUT LARGEST NUMBER *
160   REM ************************
170   CLEARW 2:A$="":B$=""
180   PRINT TAB(10)"INPUT THE LARGEST NUMBER FOR"
190   PRINT
200   PRINT TAB(10)"FOR "RA$(P)"."
210   PRINT
220   PRINT TAB (10)"LARGEST NUMBER? ";
230   FOR I=1 TO 3
240   A$=CHR$(INP(2))
250   IF ASC(A$)<48 OR ASC(A$)>57 THEN 240
260   B$=B$+A$:PRINT A$;
270   NEXT I
280   GR=VAL(B$)
290   RETURN
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
360   REM ******************
370   REM * PROBLEM SET-UP *
380   REM ******************
390   CLEARW 2
400   PRINT
```

```
410    PRINT "HOW MUCH IS";A1;BE$(P);A2;"= ";
420    INPUT ES
430    IF ES=ER THEN PRINT:PRINT TAB(10)"RIGHT!"
       :F=0:GOTO 510
440    PRINT:PRINT TAB(10)"WRONG!!"
450    FOR I=1 TO 2000: NEXT I
460    F=F+1
470    IF F<=2 THEN 390
480    PRINT
490    FOR I=1 TO 2000: NEXT I
500    PRINT:PRINT TAB(5)"THE ANSWER IS"ER
510    F=0
520    FOR I=1 TO 3000:NEXT I
530    PRINT
540    PRINT TAB(5)"ANOTHER PROBLEM Y/N";
550    INPUT A$
560    RETURN
570    REM **********
580    REM *  MENU  *
590    REM **********
600    FULLW 2:CLEARW 2:F=0
610    PRINT
620    PRINT TAB(12)"MATH TUTOR"
630    PRINT:PRINT
640    PRINT TAB(12)"CHOOSE:"
650    PRINT
660    FOR I=1 TO 4
670    PRINT TAB(11)I" FOR "RA$(I)
680    PRINT
690    NEXT I
740    PRINT TAB(12)"5 TO END"
750    PRINT
760    PRINT TAB(12)"WHICH NUMBER?"
770    E$=CHR$(INP(2))
780    P=VAL(E$)
790    IF P < 1 OR P > 5 THEN 770
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    ER=A1+A2
910    RETURN
920    REM ***************
930    REM * SUBTRACTION *
940    REM ***************
950    IF A1 < A2 THEN I=A1:A1=A2:A2=I
960    ER=A1-A2
970    RETURN
980    REM ************
990    REM * DIVISION *
1000   REM ************
1010   ER=A1*A2
1020   I=ER:ER=A1:A1=I
1030   RETURN
1040   REM ******************
1050   REM * MULTIPLICATION *
1060   REM ******************
1070   ER=A1*A2
1080   RETURN
1090   REM *******
1100   REM * END *
1110   REM *******
1120   CLEARW 2
1130   END
```

You probably found the solution quickly. A total of five program lines always appear in the program lines listing the four calculation types. For example, the following lines are for addition:

```
830 GOSUB 110
840 GOSUB 310
860 GOSUB 390
870 IFA$="Y" THEN 840
880 GOTO 580
```

The only difference in the individual program sections lies in the calculation itself. This is why lines 830, 840, 860, 870, and 880 are placed after the menu. The test of `P=5` is made separately, since the end of the program does not involve a subroutine.

This way we can write the calculations for addition, subtraction, etc. as subroutines and call them with the command:

```
ON P GOSUB
```

We save nine program lines in this manner.

This solution concludes the section on subroutines. Here is a summary of the most important things to remember when working with subroutines:

1. Subroutines combine several repetitive sections of a program.

2. Subroutines are called with `GOSUB` and terminated with `RETURN`. They may never be called or exited with `GOTO`. The `GOTO` command may be used *within* the subroutine, however.

3. Subroutines may be "nested," in the sense that one subroutine may call another. Since the return addresses are stored in the computer, the total number of nested subroutines is limited. Be sure that each `GOSUB` command has a corresponding `RETURN`.

4. Subroutines may also be called with `ON X GOSUB`.

In the next section we will take a look at the structure and use of menus.

4.3 Menu techniques

Once you have become proficient in BASIC programming, you will probably want to write larger programs on your own. You may want to write sizable programs for fun—or even profit. If you do write a program intended for sale in the highly competitive software market, it's a real advantage if your program is *user-friendly*. What do we mean by this?

A program is written to perform certain tasks, like calculate formulas or draw charts. But the user must know exactly how to use the program—what keys to press to, say, recalculate a column of numbers or change a bar graph's fill pattern. *User-friendly* means that someone who isn't familiar with the program will still be able to use it without a great deal of explanation or training. Naturally, no large program is complete without a user's manual. But good programmers try to write programs so that the user won't have to refer to the manual for every minor function.

Don't worry. We don't expect you to write any instruction manuals. For now it's sufficicient for you to master the principles of menu techniques.

We've already mentioned the term *menu* in connection with the program `MATH TUTOR`; in fact, you worked with a menu in that program. For our purposes, we'll define the term menu as follows:

> **Menu**: a list of the individual points in the program that the user can choose from by pressing a letter or number key on the keyboard.

What a menu looks like is up to you. But a menu should always be as clear and readable as possible. Separating the different program functions should be as concise and easy to understand as possible, not overwhelming. And you should always try to make your menus aesthetically pleasing.

We'll next show you how to construct a menu step-by-step. As a working example we'll create a mathematical table of commonly-used computations.

We first clearly determine the purpose of the program. Our mathematical table will require the following computational functions:

```
Square root
Sine
Cosine
Natural logarithm
Base 10 logarithm
```

The user will select one of these five functions to be performed.

This initial menu lacks something—the program end option. You should write your programs so that they can be ended with such an option, not just with <Control>C or the on/off switch. This gives our menu six options.

We also need a statement in our program that will prompt the user to enter a number or a letter, something like:

```
ENTER YOUR SELECTION (1-6)
```

The planning for our menu is now almost complete. We want to add a heading and a border as aesthetic improvements. We will write the program so the title will be on the screen during execution of each program segment. A subroutine is a good way to create a title. This illustrates how the menu will later appear on the screen.

```
****************************************
*                                      *
*          MATHEMATICAL TABLE          *
*                                      *
****************************************


   1 SQUARE ROOT
   2 SINE
   3 COSINE
   4 NATURAL LOGARITHM
   5 LOGARITHM BASE 10
   6 END PROGRAM

INPUT THE NUMBER OF YOUR CHOICE (1-6)
```

We'll need to read the number in the last line, as well. At first we'll use the `INPUT` command for the input. Later we'll see how to use the `INP` command in this position. Here's a listing of a program to create this menu:

```
10   REM *****************
20   REM * PROGRAM START *
30   REM *****************
40   REM
50   CLEARW 2:FULLW 2
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 LOGARITHM BASE 10"
140  DATA "   6 END PROGRAM"
150  GOTO 330
160  REM ***************
170  REM * SUBROUTINES *
180  REM ***************
190  REM
200  REM ***********
210  REM * HEADLINE *
220  REM ************
230  CLEARW 2
240  FOR I=1 TO 40:PRINT"*";:NEXT I:PRINT
250  PRINT "*(38 blank spaces)*"
260  PRINT "*(10 spaces)MATHEMATICAL TABLE(10 spaces)*"
270  PRINT "*(38 blank spaces)*"
280  FOR I=1 TO 40:PRINT"*";:NEXT I:PRINT
290  RETURN
300  REM ********
310  REM * MENU *
320  REM ********
330  GOSUB 230
390  FOR I=1 TO 2:PRINT:NEXT I
400  FOR I=1 TO 6
410  PRINT M$(I)
420  NEXT I
430  PRINT
450  PRINT "INPUT THE NUMBER OF YOUR CHOICE (1-6)";
460  W$=CHR$(INP(2))
```

First we will explain the individual program lines. Line 50 clears the screen. In lines 60 through 80, the array M$ is initialized with the data from the DATA statements from lines 90 to 140. The program then branches past the subroutines and continues execution with line 330. From there the program branches to line 230, where the menu heading is created. After leaving the subroutine, program execution continues with line 340. Line 390 outputs three blank lines so that the menu options don't appear right below the heading. Lines 400 to 420 display the array with the individual menu options. Line 450 displays the prompt for the user to enter a value. Since the PRINT command in this line is terminated with a semicolon, the input is expected directly behind the parentheses (1-6).

These are all fundamentals of creating a menu. We'll skip the variable input and program branch, as we have already covered these subjects in previous sections. The principle is the same as for MATH TUTOR. Since we are working with INP, remember to check the entered values for validity.

Note the subroutine HEADER. You can call this subroutine whenever you want to reconstruct the screen. When you branch to the program segment for square root calculation, the first call there should be:

```
GOSUB 230
```

You can then ask for the input of the value to be calculated.

Finish writing this program for the practice—but first we want to explain about cursor positioning.

4.3.1 Cursor positioning with GOTOXY

GOTOXY sets the cursor to a specific position on the screen.

The notation of GOTOXY is:

```
GOTOXY X,Y
```

The parameters are:

X = column number (0-39)
Y = line number (0-24)

The positioning is usually followed by output with PRINT. The following example should clarify this:

```
10 CLEARW 2
20 GOTOXY 19,12
30 PRINT "OUTPUT WITH GOTOXY"
40 END
```

This gives us an easy-to-use method for outputting text at arbitrary locations on the screen. This can prove quite useful as used in the following section.

4.3.2 Using input routines in the menu

The input operations we previously used with INP were very primitive. If you entered three characters or numbers, they were automatically accepted without discrimination. Also, three characters of input were always required, forcing the user to enter the digit sequence 054 for the number 54. Furthermore, it wasn't possible to enter a number larger than 999. We had to resort to the INPUT command.

We should mention again that the INPUT command is usually sufficient for your own applications. But if you want to eliminate error conditions in your program, eventually you'll have to use the INP instruction.

Now we'll deal with the development of our own input routine. Once completed, it can be adapted for use in your own programs. The first line should look familiar:

```
10 A$=CHR$(INP(2))
```

With this you can read any character from the keyboard and assign it to A$. In this example we want to read numbers only. Therefore, input other than numbers must be disregarded. We accomplish this with an IF...THEN test:

```
10 A$=CHR$(INP(2))
20 IF ASC(A$) < 48 OR ASC(A$) > 57 THEN 10
```

The ASCII values 48 through 57 represent the numbers from 0 to 9. If the ASCII value entered is less than 48 or greater than 57, the input is ignored and the program branches back to line 10. To limit the number to a certain number of digits, we must count valid characters. If the largest number is to be 4 digits, we must test the counter to see if it's larger than 4. We need two additional lines—one in which the counter is incremented, and one in which the counter is checked for the value 4.

```
10 A$=CHR$(INP(2))
20 IF ASC(A$) < 48 OR ASC(A$) > 57 THEN 10
30 Z=Z+1
40 IF Z>4 THEN 10
```

The counter Z in line 30 is incremented only when a valid value is entered. If Z already has a value of 4, no more values are accepted and the program jumps back to line 10.

We now tell our routine that the entered value is acceptable. As with the INPUT command, we will use the <RETURN> key. What ASCII code does the <RETURN> key have? From the table we learn it has the value 13. We need only test ASC(A$) for the value 13. But where do we put this test? Since 13 is less than 48, we can't put this test *after* line 20. Otherwise the <RETURN> key will be ignored. This test must have a line number less than 20. Let's use the number 15.

Where should the routine branch when the <RETURN> key is pressed? We really don't know yet. However, we can see that the routine probably won't get very large, so we'll branch to line 100.

```
10 A$=CHR$(INP(2))
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 that chains the entered characters together into a string. This is done in line 50. Furthermore, we want to be able to see our input on the screen. Line 60 prints the characters at the current cursor position, one after the other (as a result of the semicolon):

```
10 A$=CHR$(INP(2))
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 only need to convert the string we have assembled in B$ into a numeric value, and assign it to a numeric variable. This can be done after the <RETURN> key is pressed. Also, we must remember to set counter Z back to zero after the <RETURN> key. Otherwise the old value will be retained the next time the routine is called, and you won't be able to enter a four-digit number.

If the routine is to be a subroutine, the last line must contain a RETURN command. First we'll take a look at the complete routine. After you have entered it, you can experiment with it a little. Maybe, in the last line, you want to output the numerical value assigned to the variable:

```
10 A$=CHR$(INP(2))
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
```

Now we have a GET routine that will read a number of up to four digits and display it. If you want to be able to enter larger numbers, change the value in line 40.

This routine is already easier to use than the one in MATH TUTOR. We are still missing the function that allows us to delete values already entered. This function is one of the more complicated features of such a GET routine. The following routine listing contains this function:

```
10  REM GET-ROUTINE
12  LN=16 : REM LINE NUMBER LOCATION
15  GOTOXY 0,LN
20  A$=CHR$(INP(2))
30  IF ASC(A$) = 13 THEN 130
40  IF ASC(A$) <> 8 THEN 70
50  IF LEN(B$) < 1 THEN 20
55  GOTOXY 0,LN:FOR I=1 TO LEN(B$):?" ";:NEXT
60  B$=LEFT$(B$,LEN(B$)-1):Z=Z-1:GOTOXY 0,LN
65  PRINT B$; :GOTO 20
70  IF ASC(A$) < 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 PRINT B
150 END
```

We'll discuss the program's new lines. Line 40 checks to see if the <BACKSPACE> key was pressed. The ASCII value of this key is 8. If this key was not pressed, the program branches to line 70. If <BACKSPACE> key was pressed, line 50 checks to see if the string B$ still contains characters.

If the string is empty, the <BACKSPACE> key is ignored and the program branches back to line 20.

Line 55 sets the cursor position to zero on our input line—designated in line 12. Then it erases the input by replacing it with spaces.

Deleting the character takes place in line 60. The string in B$ is shortened by one character by the command sequence:

```
B$=LEFT$(B$,LEN(B$)-1)
```

The function LEFT$(B$,X) creates a string consisting of the X leftmost characters of B$. In most cases, a number takes the place of X. Here we use the LEN function instead of X. The calculation LEN(B$)-1 is first executed.This means a value exactly one less than the current length of B$ is used. Substring of B$ is assigned this new value. This substring is exactly once character shorter than the original string B$.

This substring is then assigned to B$. This process is something similar to the following operation for numerical variables:

```
A=A-1
```

Here the value 1 is subtracted from the variable A and the result is again assigned to A. This command sequence deletes the last character of the string B$.

In addition, the counter Z must be decremented by 1 in line 60 since the number of digits in the entire number has been decreased. We want to enter a maximum number of four digits. Therefore, Z is used as a counter to count valid characters already entered. If we delete a character, not only must the string B$ be shortened by one character, but 1 must also be subtracted from the value of Z. If you forget to decrement the counter, it would no longer be possible to enter anything after four valid characters. The string B$ would be shortened by one character each time the <BACKSPACE> key was pressed, but since the counter had reached the value four, the program would branch in line 90 back to line 20 again.

The last command in line 60 sets the cursor back to the beginning of the line. Line 60 prints the new contents of B$—shortened a character. Then it branches to the character input.

We now have a routine that lets us input short or long strings, depending on the setting of the variable Z. The routine is quite similar to the INPUT command, but we can edit characters and limit the length.

You are now in the position to adapt this routine to your programs. That is, you can determine yourself which keys will be allowed through the corresponding IF...THEN tests.

4.4 Sorting procedures

Many programs require data sorted according to various ordering criteria: size, alphabetical order, etc. There are a variety of different procedures for sorting. All differ from each other in performance and degree of difficulty. We will become familiar with a simple procedure that will at least give you an introduction into this subject. The more complicated sorting techniques can be more frightening than exciting to a beginner. However, if you're interested in advanced sorting procedures, there are many books available on the subject.

We'll use what's known as the *bubble sort*. Bubble sorts are so named because individual elements to be sorted "bubble up" to the top according to size. The principle of the bubble sort procedure involves comparing two neighboring elements. If one element is larger than the other, an exchange is made. All elements are compared with each other in succession.

To demonstrate the bubble sort, we'll fill an array with random numbers, and then sort and print it. We will use an array of 6 elements. The first program lines dimension the array and fill it with values:

```
10      REM GENERATE ARRAY
20      DIM F(6):RANDOMIZE 0:CLEARW 2
30      FOR I=1 TO 6
40      A=INT(50*RND)+1
50      F(I)=A
60      NEXT I
.
.
```

The comparisons and exchanges will be accomplished with `IF...THEN` tests. This could also be done with a `FOR...NEXT` loop, but the procedure wouldn't be so clear. Once you understand the procedure, you'll be able to perform it with a `FOR...NEXT` loop.

Now the program lines for the sort procedure:

```
.
.
100    REM BUBLE 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    NEXT I
260    PRINT
270    END
```

Line 110 first sets `Z` to zero. You will see why this is done later on in the program. Line 120 performs the first comparison. If the contents of the element `F(6)` are already greater than or equal to `F(5)`, no exchange need be made and a branch is made directly to line 140. If `F(6)` is smaller than `F(5)`, an exchange is performed in line 130.

The principle of the exchange should already be familiar to you. We use the element `F(0)` for temporary storage of a variable value. Next the value of the element `F(5)` is assigned to element `F(6)`. Finally, `F(5)` is assigned the value of `F(0)`—that is, the old value of `F(6)`. This principle is used in the other program lines as well.

`Z` is then set to one because an exchange does occur. We can learn from `Z` whether or not an exchange took place. If `Z` has the value 1, an exchange was made. Conversely, if `Z` has the value 0, no exchanges were made. This is common programming technique to check if specific processes occurred or not. Variables like `Z` are known as *flags*.

If the flag Z retains the value of zero throughout a pass, we know that no exchanges taken place. Therefore the array is sorted. The advantage flags have under these conditions is that they terminate sorting after one pass—*if* the elements of the array happen to be in the right order already. This procedure is also called a *bubble sort with switch.* Its difference from a standard bubble sort is that it terminates the sorting process as soon as the array is sorted.

The last lines of the sort routine output the sorted array. If you want to see how the sorting takes place in the individual steps, change the last program lines as follows:

```
.
.
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 procedure works well for use with up to about 100 elements. Try to gain a solid understanding of bubble sorts. This knowledge will come in handy when you later work with more complicated, efficient sort procedures. This concludes our discussion of the subject.

Chapter 5

WORKING WITH THE DISK DRIVE

5.1 Program management

Few computer users turn their computers off and then back on again and still expect their entered program lines to remain in the computer's memory. The internal memory of the ST, like the memory of all computers, is only temporary. If it isn't supplied with electrical current, its contents are lost. For this reason it's necessary to store programs on an external storage medium. For this purpose, Atari ST is equipped with floppy disk drive.

This disk drive allows you to store up to 360 Kbytes or 720 Kbytes of your data, either programs or other files, and retrieve your data at any time. As with many other commands, the BASIC menu bar has all the commands needed to save and load programs. In this section we will take a look at the commands available for program management.

5.1.1 Saving programs

The ST has two ways of a saving a program. The first command for saving programs is the command `SAVE` . The `SAVE` command stores a program on the diskette, provided there's no other file on the disk by that name:

`SAVE` (*name, line range*)

Parameter *name* determines the name of the program. This name is constructed as follows:

```
xxxxxxxx.yyy
```

The actual filename may consist of a maximum of 8 characters. An *extension* may be entered following the period, and can contain a maximum of 3 characters. Files are classified and easily identified with an extension. If no extension is entered, `.BAS` is automatically appended. Note that the program name is not placed in quotation marks.

The *line range* corresponds to the parameter of the `LIST` or `DELETE` command. If no line range is given, the entire program is saved.

The `SAVE` command creates an error message if the program already exists under the given name on the diskette.

Here's the second way to save a program:

`REPLACE` *name, line range*

As the name implies, this command replaces a program by the same name already on the disk. This command erases the old data with that name, and stores the new. The parameters correspond to those of the `SAVE` command.

5.1.2 Loading programs

ST BASIC also has two commands available for loading a program. They are identical in the way they work:

`OLD` *name*

(or)

`LOAD` *name*

If no filename extension is entered, the extension `.BAS` is automatically supplied. If the specified program is not found, the following error message will be printed:

```
        File not found on disk drive specified
```

5.1.3 Displaying the disk contents

The ST BASIC command to display the contents of the diskette is `DIR`. `DIR` is an abbreviation for *directory*. The syntax is as follows:

`DIR` *drive: mask*

If the command is entered without a *mask*, the entire contents of the directory are displayed. A mask allows you to display specific groups of files by modifying a DIR command. An asterisk signifies that all characters normally in place should be ignored. In other words, the asterisk substitutes for groups of characters. Some examples:

`DIR A:*.BAS`	shows all BASIC programs on drive A.
`DIR B:TEST.*`	lists all files with name TEST. Extension is ignored.
`DIR A:A*.B*`	lists all files with names beginning with A and extensions beginning with B.

Another character used with DIR command masks is the question mark (?). A question mark designates the individual characters that will be ignored in the listed filenames. For example, the mask ?FILE.BAS would list the files AFILE.BAS, BFILE.BAS, CFILE.BAS, etc.

5.1.4 Erasing files

Once a disk gets so full that no more programs can be saved on it, you may want to do some "cleaning up". Usually the disk contains old versions of programs that are no longer required.

The following command is used to erase files:

ERA *drive:name*

The command ERA A:TEST.BAS deletes the program TEST.BAS from the diskette in drive A, for instance. A mask can also be given instead of the filename, as described with the DIR command.

`ERA A:*.*` (deletes all files from drive A)

Again, there is an equivalent ST command:

KILL "*drive:name*"

The only difference between the two commands is that the latter specifies the filename in the quotation marks. The advantage of this should not be overlooked: the parameter is a string. The following command sequence is therefore possible in connection with the KILL command:

```
B$="PROG1.BAS"
KILL B$
```

The advantage of this "parameterized" deletion is that a filename can be read from the keyboard within a program, and can then be deleted. This doesn't work with the command ERA.

5.1.5 Renaming files

ST BASIC also has the ability to perform DOS (Disk Operating System) functions. One elementary DOS function is the renaming of files. The corresponding BASIC command for this is:

NAME *old file* AS *new file*

The parameters are fairly self-explanatory. For example, if we want to change the name of the program TEXT05.BAS to TEXTPRO.BAS, the following command will do it:

```
NAME TEST05.BAS AS TEXTPRO.BAS
```

5.2 Sequential file management

In previous sections, you've learned to handle different types of data, like names, numbers, and equations. This data has always been an integral part of the program, in the form of `DATA` statements. However, it should be clear that this is an awkward way of handling data. You can't expect the user of your program to modify the program lines when entering or editing data. Doing so in programming languages other than BASIC would require complete program recompilation. Obviously, we have to find a different means of managing data.

The most common way to manage the data is to form an independent file on the disk, reserved for data only. In this section we'll learn to use the best-known form of file organization for data management—*sequential files*.

In a sequential file, the data records are organized one after the other, separated from each other by the ASCII character <RETURN>. This <RETURN> is required because the corresponding command to read an entry in the sequential file reads up to this <RETURN>. The ASCII value for <RETURN> is `CHR$(13)`.

We will explain sequential file management with a few example programs.

Let's assume that we want to store an address list in the ST's memory as a two-dimensional table on diskette. Each address record consists of three fields:

`A$(n,1)`	Last name
`A$(n,2)`	First name
`A$(n,3)`	Telephone number

The whole list will have 200 entries. Managing such a file requires a continuous *record counter*. The record counter, or `RC`, always points to the last record. This `RC` is required for the new name entry, since the next free table location can be determined from `RC+1`.

We could use a subroutine like the following one to print this address table on the screen:

```
1000  REM LIST PHONE
1010  LSTPHONE:
1020  FOR X=1 TO RC
1030  PRINT "ENTRY #";X
1040  PRINT "NAME:      ";A$(X,1)
1050  PRINT "NAME:      ";A$(X,2)
1060  PRINT "NAME:      ";A$(X,3)
1070  PRINT
1080  PRINT "HIT RETURN TO CONTINUE"
1090  IF INP(2)<>13 THEN 1090
1100  NEXT X
1110  RETURN
```

This subroutine is fairly easy to read and understand, so we won't explain it any further. It's more important that we should learn how to write these data records on the diskette.

A subroutine to store the data records must first open a file in which the addresses will be written. The command to open a sequential file for writing is the following:

OPEN "O",*#file number*,"*filename*"

If the address file is to be called ADDRESS.DAT, the OPEN command is:

```
OPEN "O",#1,"ADDRESS.DAT"
```

The file number is required to distinguish between several open files. Here is a subroutine for writing the addresses:

```
2000  REM SAVE PHONE
2010  SAVPHONE:
2020  OPEN "O",#1,"PHONE.DAT"
2030  PRINT #1,RC
2040  FOR X=1 TO RC
2050  FOR Y=1 TO 3
2060  PRINT #1,A$(X,Y)
2070  NEXT Y,X
2080  CLOSE #1
2090  PRINT "PHONE NUMBERS STORED."
2100  PRINT "CONTINUE WITH RETURN"
2110  IF INP(2)<>13 THEN 2110
2120  RETURN
```

After opening the file, first the record counter is saved. It will be required later for loading the address records. The individual elements of the table are stored in a nested loop, in the following order: last name, first name, and telephone number of the first entry; then the last name, first name, and phone number of second entry; and so on. The `PRINT #1` command has a file number and is used for writing to the sequential file. Here is the subroutine for entering data into this file:

```
3000  REM LOAD PHONE
3010  LDPHONE:
3020  OPEN "I",#1,"PHONE.DAT"
3030  INPUT #1,RC
3040  FOR X=1 TO RC
3050  FOR Y=1 TO 3
3060  INPUT #1,A$(X,Y)
3070  NEXT Y,X
3080  CLOSE #1
3090  PRINT "FILE IS LOADED."
3100  PRINT "CONTINUE WITH RETURN"
3110  IF INP(2)<>13 THEN 3110
3120  RETURN
```

The mode in the `OPEN` command is `I` (input) when reading a file. After the record counter is read, it is used as the argument for the outer loop. The subroutine "knows" by the record counter how far it must read.

The following subroutine allows you to enter the data.

```
4000  REM ENTER RECORD
4010  ENTRPHONE:
4020  RC=RC+1
4025  PRINT "ENTRY #";RC
4030  INPUT "LAST NAME :";A$(RC,1)
4040  INPUT "FIRST NAME:";A$(RC,2)
4050  INPUT "TELEPHONE :";A$(RC,3)
4060  PRINT
4070  PRINT "CONTINUE WITH RETURN"
4080  IF INP(2)<>13 THEN 4080
4090  RETURN
```

And finally, our main program that pulls it all together:

```
100    REM MAIN PROGRAM
110    FULLW 2 :DIM A$(100,3)
120    MENU: CLEARW 2
130    PRINT "***********************"
140    PRINT "* TELEPHONE DIRECTORY *"
150    PRINT "***********************"
160    PRINT
170    PRINT " 1  ENTER DATA"
180    PRINT " 2  LIST DATA"
190    PRINT " 3  SAVE DATA"
200    PRINT " 4  LOAD DATA"
210    PRINT " 5  END"
220    PRINT
230    PRINT "SELECT ITEM ";
240    A$=CHR$(INP(2)):PRINT A$: A=VAL(A$)
250    ON A GOSUB ENTRPHONE,LSTPHONE,SAVPHONE,
       LDPHONE,FINISH
260    GOTO MENU
300    FINISH:
310    END
```

One command that we should mention when using subroutines is:

MERGE *filename*

By using this command, you can save all your subroutines individually and then later MERGE them together to create one total program. This way you can keep your much needed subroutines in a library on disk and later incorporate them into your programs.

These subroutines can be easily adapted to your own requirements. Using sequential files, you're now ready to create your own data management programs.

Chapter 6

SOUND AND GRAPHICS

Sound and Graphics

6.1 Sound

Computers are excellent tools for creating sound. The processor, the heart of the computer, is controlled by a clock generator. This generator creates 1 to 8 MHz (million cycles per second) depending on the computer.

The simplest method to create sound from this generator is to send it an audio speaker. But since the sound frequency produced in such a manner is much too high to be heard by humans, the clock frequency must be lowered considerably. The computers "divided" this frequency to a low enough level that it can be output to a speaker. The resulting frequencies produced audible sounds.

This was the first attempt at computer sound synthesis. The resulting sounds are rather unpleasant, comparable to a digital wristwatch's alarm. But the sound quality of personal computers is continually being improved. The result is that most home computers are now equipped with synthesizers on a chip. The sounds they create are deceptively similar to those of "real" (acoustic) musical instruments.

The sound chip in the ST cannot honestly be called a synthesizer. Important features like several different waveforms (sine, square wave, etc.) and a tone-modifying filter are missing.

But the sound generator in the ST has three voices available. It can produce three tones at once, making it polyphonic. Everything produced by the sound chip is played the speaker built into the ST's monitor. The volume can be adjusted on the monitor.

The following section will acquaint you with the BASIC commands for creating sound on the ST. For a complete discussion of the ST's sound capabilities, please refer to the reference books *Atari ST Internals* and *Atari ST Graphics & Sound* from Abacus.

6.1.1 The SOUND command

The sound generator of the ST is controlled by the command SOUND. This is the syntax of the command:

SOUND *channel, volume, note, octave, duration*

The parameters are:

•channel	sound channel (1 to 3)
•volume	from 0 to 15 (0=silent, 15=loud)
•note	from 1 to 12 (1=C, 2=C#, ... 12=B)
•octave	from 1 to 8 (1=low, 8 high)
•duration	duration of the sound in steps of 1/60th second

The musical scale consists of 12 tones, as you've probably seen the 12 different keys on a piano keyboard.

tone parameter	Tone in the musical scale
1	C
2	C#
3	D
4	D#
5	E
6	F
7	F#
8	G
9	G#
10	A
11	A#
12	B

A keyboard consists of several such groupings of 12 notes. Each grouping called an octave. The first octave comprises the low notes, the last octave the highest.

The following program converts your ST into a musical instrument:

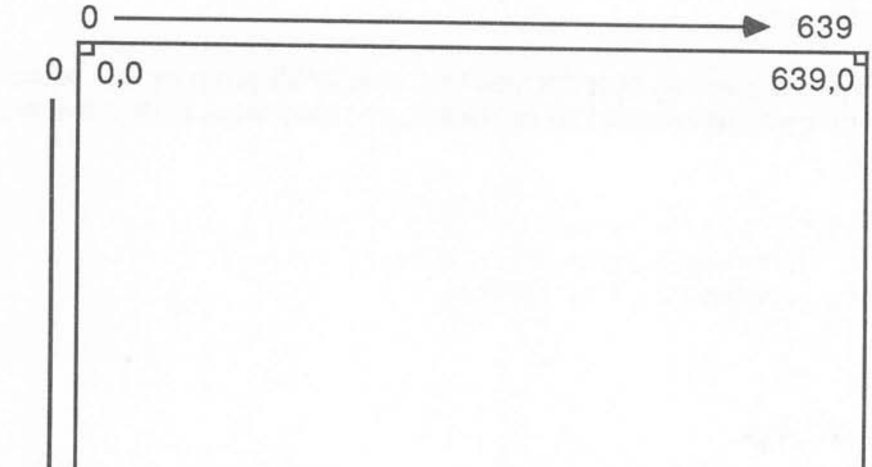
```
10   REM ST MUSIC
20   DIM TON(256)
30   TON$="Q2W3ER5T6Y7UI9O0P"
40   FOR I = 1 TO 17
50   INDEX = ASC(MID$(TON$,I,1))
60   TON(INDEX)=I
70   NEXT I
80   CLEARW 2:FULLW 2
90   PRINT "    ST MUSIC"
95   PRINT
100  PRINT
110  PRINT " 2 3  5 6 7  9 0"
115  PRINT
120  PRINT "Q W E R T Y U I O P"
140  A=INP(2)
145  IF A=13 THEN END
150  ON=TON(A)
160  OCT=3
170  IF TON>12 THEN TON=TON-12:OCT=4
180  SOUND 1,15,TON,OCT,20
190  SOUND 1,0,0,0,0
200  GOTO 140
```

Now RUN it. As you can hear for yourself, this small program has some big effects! Once you understand its operation, you may want to improve on it.

6.2 Graphics

For many home computer buyers, a computer's color display and graphics capabilities are deciding purchase factors. The ST offers excellent color and graphic features far surpassing other machines in its price range. The ST has 512 colors that can be mixed from the basic colors red, green, and blue. The resolution of the graphics in monochrome mode is 640 x 400 points—sensational for a microcomputer with its price tag.

The ST lets you select from three different graphic resolutions with different numbers of colors. The high-resolution mode consists of 640 x 400 points in black and white. So that each point can be accessed, the screen is constructed as a coordinate system. Similar to the command GOTOXY, which puts the cursor at a specific location on the screen, a position is determined with graphic commands having two values. The first value determines the position on the horizontal axis (X axis) of the screen, from left to right. The second value corresponds to the position on the vertical axis (Y axis), from top to bottom. The four corner points have the following coordinates:



Once you have become familiar with this system, you can start using the commands for creating graphics.

6.2.1 Lines

LINEF *starting x,y, ending x,y*

The parameters are self-explanatory. For example, the following command draws a line from the top left to the bottom right corner of the screen :

```
LINEF 0,0,639,399 (0,0,639,199 Color)
```

Drawing lines becomes interesting when used in conjuction with loops. Try the following example:

```
10 FULLW 2:CLEARW 2
20 FOR I=0 TO 199 STEP 5
30 LINEF 199,I,0,199-I
40 NEXT I
```

This simple program produces an interesting effect. Let's try another one:

```
10 FULLW 2:CLEARW 2
20 FOR I=0 TO 199 STEP 10
30 LINEF I,0,199,I
40 LINEF 199,I,199-I,199
50 LINEF 199-I,199,0,199-I
60 LINEF 0,199-I,I,0
70 NEXT I
```

You may want to have some fun creating graphics of your own. The difficult part is converting your intended figure into points in the coordinate system—i.e., constructing an algorithm.

6.2.2 Circles

To draw a circle, both the origin of the circle and the radius must be specified. In addition, we can create arcs by specifying of the starting and ending angles as well. The command has the following syntax:

CIRCLE *x origin,y origin,radius,start angle,end angle*

The angles are specified in tenths of a degree (0-3600).

The following example draws a circle with the radius of 50 points in the center of the screen:

```
        CIRCLE 320,200,50 (320,100,50 Color)
```

Here is a program that demonstrates the drawing of arcs:

```
10 FOR I=10 TO 100 STEP 5
20 CIRCLE 320,100,I,0,900
30 NEXT I
```

6.2.3 Ellipses

The command for drawing ellipses corresponds to `CIRCLE`, but includes a distinction of the X and Y radii.

`ELLIPSE` *x origin, y origin, x radius, y radius, start angle, end angle*

An ellipse drawn at the center of the screen with the X radius of 100 and the Y radius of 50 is drawn as follows:

```
   ELLIPSE 320,200,100,50 (320,100,100,50 Color)
```

Here is a short demonstration program for this command:

```
10 FOR X=50 TO 200 STEP 10
20 FOR Y=50 TO 100 STEP 5
30 ELLIPSE 320,100,X,Y
40 NEXT Y,X
```

6.2.4 Filled surfaces

Circles and ellipses can be filled with colors and/or patterns. The two commands are:

`PCIRCLE` and `PELLIPSE`

The parameters match those of the commands already described. Before the fill commands can be executed, certain attributes must be set. This is done with the command `COLOR`. Its syntax is given here:

`COLOR` *text color, fill color, line color, style, index*

Here is the description of the parameters:

• text color	Color of the text output
• fill color	Color of the fill pattern
• line color	Color of the lines, circles, and ellipses
• style	Fill pattern
• index	Fill type (2=dotted pattern, shaded pattern)

The first three parameters are always 1 (black) or 0 (white) on a monochrome monitor. To draw the graphic, the line color is set to black (`COLOR 1,1,1`), or white to erase it (`COLOR 1,1,0`).

The fill pattern can be set with the parameters *style* and *index*. The following program demonstrates the 24 different dotted patterns:

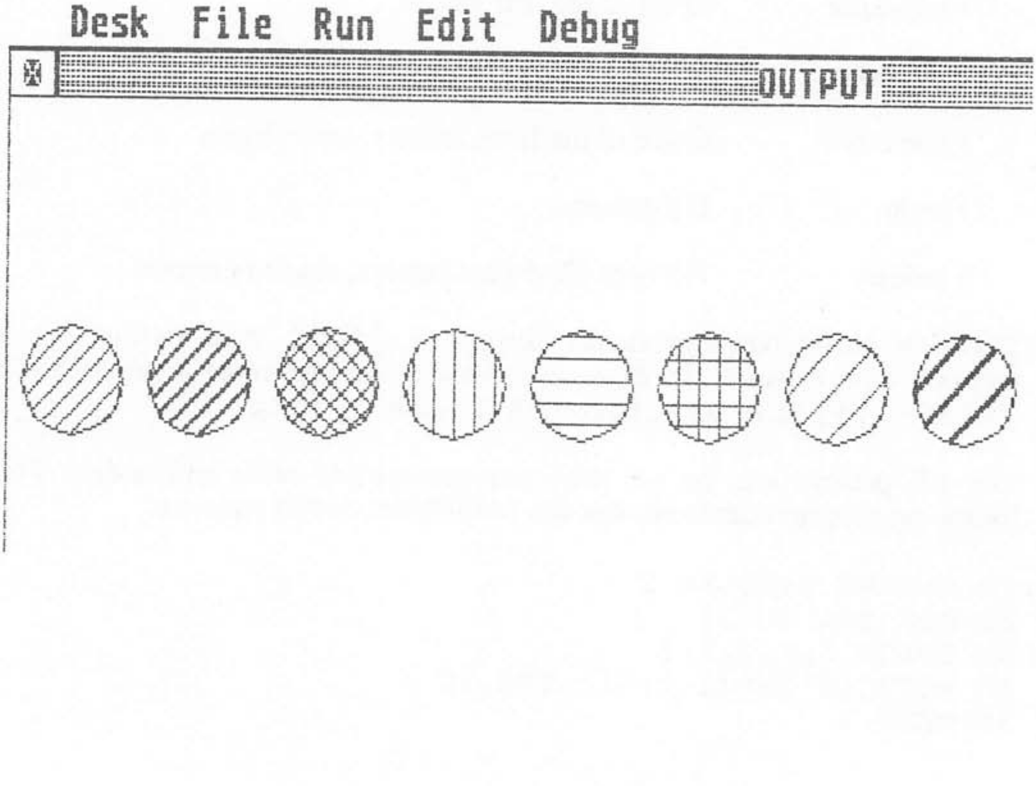
```
10 CLEARW 2:FULLW 2
20 FOR I=1 TO 24
30 COLOR 1,1,1,I,2
40 PCIRCLE 25*(I-1)+15,100,20
50 NEXT I
```

Here are the 12 shaded patterns:

```
10 CLEARW 2:FULLW 2
20 FOR I=1 TO 12
30 COLOR 1,1,1,I,3
40 PCIRCLE 50*(I-1)+15,100,20
50 NEXT I
```

Independent of the *style* parameter, the fill type 0 creates surfaces filled with white. Type 1 surfaces are filled with black. Fill type 4 contains the Atari logo.

This section should serve as a primer for your own graphic experiments. For advanced graphic and sound programming techniques, please refer to *Atari ST Graphics & Sound* from Abacus.



Chapter 7

USING GEM WITH ST BASIC

Using GEM with ST BASIC

7.1 GEM fundamentals

Before we discuss the functions of GEM—the Graphics Environment Manager—you should know how it is structured. GEM consists of the two major parts:

VDI (Virtual Device Interface)

AES (Application Environment System)

The VDI and AES are collections of subroutines. These functions are integrated into your application written in assembly language or C language during the assembly or compilation. It is not recommended you write entire GEM applications in BASIC, but various functions can be used effectively.

7.1.1 GEM VDI

The VDI contains all basic graphic functions, such as drawing lines, circles, etc. The functions which create GEM applications like windows and boxes are contained in the AES. For example, if the function to open a window is used, the AES is responsible. AES is in turn supported by the VDI routines. For example, the VDI creates the border of the window. VDI finally uses DOS functions. The hierarchy of GEM is therefore:

DOS (Disk Operating System)

VDI (Virtual Device Interface)

AES (Application Environment System)

The VDI's task is to facilitate creating graphic applications through graphic functions. This makes the functions independent of the graphics output device. The device drivers, also part of the VDI, take care of the device-specific control.

7.1.2 GEM AES

To clarify the purpose of AES, we should begin with its name: Application Environment Manager. This "environment" is graphic-oriented in GEM. This means graphic elements (like windows and icons) are used as the interface, or communication link, between the computer and the user.

The graphic environment of the AES is a particularly powerful operating system composed of several components.

The *routine libraries* offer functions with which all the elements of the AES system can be controlled and used. *Multitasking* makes it possible to let AES appear to be running several programs at the same time. The *Shell* represents the connection to the actual operating system: TOS. To manage several different screen elements, the AES uses *buffers,* which serve to temporarily store screen parts.

7.2 Passing parameters to GEM routines

The GEM routine are so complex that they require not just individual variables but whole tables (arrays) of input and output parameters. The VDI functions to open an output device comprise no fewer than 14 input and 60 output parameters! But don't worry, we will demonstrate only a few of the many GEM functions. The principle of passing and accepting the parameters should be explained first.

The VDI functions are controlled via 5 input/output arrays:

`contrl`	input/output array
`intin`	input array
`intout`	output array
`ptsin`	input array
`ptsout`	output array

Instead of the arrays `ptsin` and `ptsout`, the AES uses the arrays `addrin` and `addrout`. The AES also uses another array named `global`.

7.2.1 VDI calls

The starting addresses of the VDI arrays are found in the reserved variables `contrl`, `intin`, `intout`, `ptsin`, and `ptsout`. The input paramaters are placed into the array with the `POKE` command. Note that the arrays `contrl`, `intin`, and `intout` contain 2-byte values (words). Therefore, the parameter `intin(3)` is set with the command:

```
POKE INTIN+6,value
```

in BASIC. This must be taken into account!

The VDI functions are numbered. This function number is stored in `contrl(0)` with the command:

POKE CONTRL,*function number*

VDI is called with the command:

```
VDISYS
```

After calling the function, any parameters returned can be read. If the VDI function returns a value in `intout(2)`, for example, it can be assigned to a variable with the command:

variable=PEEK(INOUT)

7.2.2 AES calls

AES calls are not as easy to work with as VDI calls. Here not all of the array addresses are available in reserved variables. There is only the variable GB—it contains the address of the address list of the AES array. This sounds complicated, but will be thoroughly explained. Take a look at the following diagram:

```
GB  ---->      [] [] [] [] ----> contrl

               [] [] [] [] ----> global

               [] [] [] [] ----> intin

               [] [] [] [] ----> intout

               [] [] [] [] ----> addrin

               [] [] [] [] ----> addrout
```

The address of GB points to an area of 6 by 4 bytes. This area contains 6 addresses. These are the addresses of the AES arrays.

Important: The arrays `intin` and `intout` are **not** identical to the VDI arrays of the same names!

For this reason, different designations are used in the following AES initialization.

```
1000 REM AES INITIALIZATION
1010 A#=GB
1020 GCONTRL = PEEK(A#)
1030 GLOBAL  = PEEK(A#+4)
1040 GINTIN  = PEEK(A#+8)
1050 GINTOUT = PEEK(A#+12)
1060 ADDRIN  = PEEK(A#+16)
1070 ADDROUT = PEEK(A#+20)
1080 RETURN
```

After calling this subroutine, the addresses of all of the arrays are available in the variables.

Note that the arrays `addrin` and `addrout` are 4-byte (long word) arrays. `addrin(3)`, for example, is set with the command:

`POKE ADDRIN+12,`*value*

The AES is called with:

`GEMSYS` (*function number*)

Different from the VDI, the function number is not placed in `contrl(0)` for the AES but is passed along with the AES call. It is therefore unnecessary to set the function number in the control array for an AES call.

After calling the AES, return parameters are read as described for the VDI.

7.3 VDI examples

The following program draws a rectangle with rounded corners. This graphics operation is not available in BASIC.

```
1 GOSUB 10:END
10 REM ROUNDED RECTANGLE
20 COLOR 1,1,1,1,1
30 FULLW 2
40 POKE CONTRL,11       :REM FUNCT # FOR GRAPHIC OPS
50 POKE CONTRL+2,2      :REM # OF COORDINATES IN PTSIN
60 POKE CONTRL+6,8      :REM NUMBER OF INTIN ENTRIES
70 POKE CONTRL+10,8     :REM FUNCT # FOR ROUNDED RECT
80 POKE PTSIN,50        :REM X-COORD TOP LEFT CORNER
90 POKE PTSIN+2,20      :REM Y-COORD TOP LEFT CORNER
100 POKE PTSIN+4,150 :REM X-COORD LOWER RT CORNER
110 POKE PTSIN+6,100 :REM Y-COORD LOWER RT CORNER
120 VDISYS
130 RETURN
```

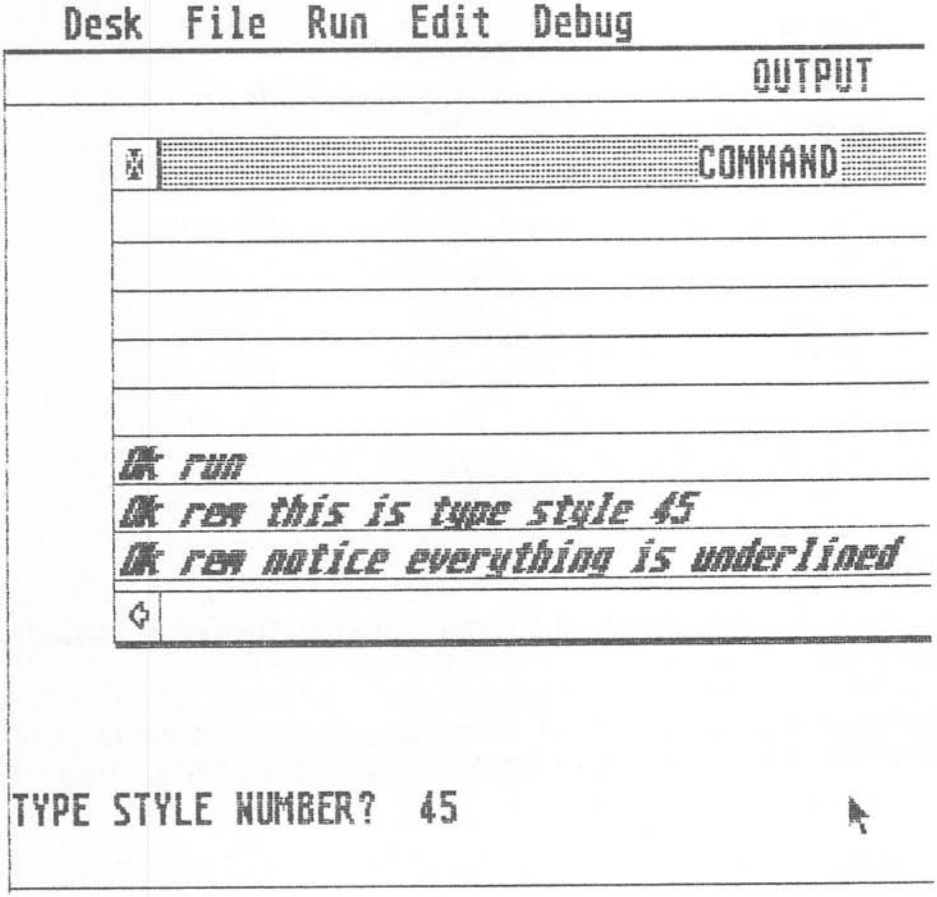
The next example program outputs different text styles on the screen. The following type styles are possible:

1	**boldface**	**2**	shaded
4	*italic*	**8**	<u>underlined</u>
16	outline	**32**	normal

Combinations of type styles are also allowed. You simply add the numbers of the type styles. Bold/italics are achieved with the style number 5, for example.

Inportant: RUN this program only in command mode. If your using the Edit window, it will change the type style back to normal before you get the chance to see the change. Here's the listing:

```
10 REM TYPE FORMATION
20 REM FULLW 2
30 INPUT"TYPE STYLE NUMBER:";A
40 IF A<1 OR A>63 THEN 30
5J POKE CONTRL,106 :REM FUNCTION NUMBER
60 POKE CONTRL+2,0 :REM NUMBER OF PTSIN ENTRIES
70 POKE CONTRL+6,1 :REM NUMBER OF INTIN ENTRIES
75 POKE INTIN,A
80 VDISYS
90 END
```



```
Ok run
Ok rem this is type style 45
Ok rem notice everything is underlined
```

```
TYPE STYLE NUMBER?  45
```

7.4 AES examples

The following demo program uses the AES routines 73 and 74 to enlarge and shrink a box. The function is often used in GEM programs to startling effect. You get the impression that the box "zooms" out of the background.

```
10     REM ENLARGE AND SHRINK A BOX
20     GOSUB 1000
30     FOR I=0 TO 7
40     READ A:POKE GINTIN+2*I,A
50     NEXT I
55     REM FOR MED-RESOLUTION
60     DATA 315,100,5,5,0,0,640,200
62     REM FOR HI-RESOLUTION
65     REM DATA 315,200,5,5,0,0,640,400
70     GEMSYS (73): GEMSYS (74)
80     IF INP(2)<>13 THEN 70
90     END
1000   REM AES INITIALIZATION
1010   A#=GB
1020   GCONTRL = PEEK(A#)
1030   GLOBAL  = PEEK(A#+4)
1040   GINTIN  = PEEK(A#+8)
1050   GINTOUT = PEEK(A#+12)
1060   ADDRIN  = PEEK(A#+16)
1070   ADDROUT = PEEK(A#+20)
1080   RETURN
```

The GINTIN array consists of 8 entries. The first four entries specify the X- coordinate of the upper left corner, the corresponding Y-coordinate, and the width and height of the starting (small) rectangle. The second four entries specify the dimensions of the resulting rectangle the same form. The program repeats itself with each keypress and terminates with <RETURN>.

The second example program changes the shape of the mouse cursor. The following forms may be entered:

0	arrow
1	cursor
2	busy bee
3	hand with index finger
4	flat hand
5	thin crosshair
6	thick crosshair
7	outline crosshair

Here's the program:

```
10     REM SET MOUSE CURSOR SHAPE
20     FULLW 2
30     INPUT "CURSOR SHAPE:";F
40     IF F<0 OR F>7 THEN 30
50     GOSUB 1000
60     POKE GINTIN,F
70     GEMSYS (78)
80     END
1000   REM AES INITIALIZATION
1010   A#=GB
1020   GCONTRL = PEEK(A#)
1030   GLOBAL  = PEEK(A#+4)
1040   GINTIN  = PEEK(A#+8)
1050   GINTOUT = PEEK(A#+12)
1060   ADDRIN  = PEEK(A#+16)
1070   ADDROUT = PEEK(A#+20)
1080   RETURN
```

7.5 Using GEM Functions in Applications—Example Program

This example gives you a feel for what you can do when you integrate GEM functions into BASIC programming. We've tried to come up with a sensible example to demonstrate this. The result of this was re-vamping a BASIC program without windows or menus intended for programmers.

The ST already has a huge command set, but these are oriented toward output windows. Graphic functions, for example, only work within these windows, so screen-absolute input and output are impossible. Try it yourself—try drawing a diagonal across the entire screen!

This section gives you a BASIC listing with subroutines for direct screen input and output. The program takes advantage of GEM functions that tell BASIC to ignore all windows. The first assignment from GEMBAS is to turn the screen white; now you can address any area of the screen. Exiting a GEMBAS program re-activates all windows. The menu line, which was cleared with the windows, is also re-enabled.

The following example is not a complete program. It is just an example to demonstrate how to create an input mask for a database. It also demonstrates how flexible BASIC can be when utilizing VDI calls.

GEMBAS programs should follow this format:

0 - 999	Program-specific initialization
1000 - 1999	GEMBAS initialization
2000 - 4999	GEMBAS routines
5000 - 9999	Program-specific routines
10000 -	Main program

Lines 10000 to the end contain a GEMBAS demo program.

All routines with CALL conventions:

Routine:	INITWORK
Line range:	2000-2080
Assignment:	Establish working range: lines 1-2 program name lines 2-23 input/output lines 23-25 message line
Input parameters:	none
Output parameters:	none

Routine:	SETPRGNAME
Line range:	2100-2197
Assignment:	Set program name in line 1
Input parameters:	NAME$ - program name
Output parameters:	none

Routine:	RECTANGLE
Line range:	2200-2320
Assignment:	Draw a rectangle
Input parameters:	X1 - Upper left-hand corner (0-639) Y1 - Upper left-hand corner (0-399) X2 - Lower right-hand corner (0-639) Y2 - Lower right-hand corner (0-399)
Output parameters:	none

Routine:	MESSAGE
Line range:	2400-2520
Assignment:	Output message in line 25
Input parameters:	MESGE$ - message
Output parameters:	none

Routine:	`STRINGOUT`
Line range:	2600-2740
Assignment:	Output string variable
Input parameters:	`TEXT$` - string
	`TEXTSTYLE` - typestyle
	0 normal 1 bold 2 bright 4 cursive 8 underlined 16 outlined 32 shaded
	`ROW` - line position (1-22)
	`COLUMN` - column position (1-80)
Output parameters:	none

Routine:	`CLEARSCREEN`
Line range:	2800-2850
Assignment:	Clear input/output
Input parameters:	none
Output parameters:	none

Routine:	`CHAROUT`
Line range:	2900-2995
Assignment:	Output a character
Input parameters:	`CHARS$` - character (ASCII)
	`ROW` - line position (1-22)
	`COLUMN` - column position (1-80)
Output parameters:	none

Routine:	`STRINGIN`	
Line range:	3000-3210	
Assignment:	Input routine	
Input parameters:	`TEXT1$` -	prompt string
	`TEXT2$` -	default string
	`STYLE1` -	typestyle prompt
	`STYLE2` -	typestyle default
		0 normal 1 bold 2 light 4 cursive 8 underlined 16 outlined 32 shaded
	`ROW` -	line position (1-22)
	`COLUMN` -	column position (1-80)
Output parameters:	`TEXT2$` -	given string
	`BACK` -	end key
	0 -	<RETURN>
	1 -	cursor down
	-1 -	cursor up

Now the program listing:

```
100   rem ***********************
105   rem * Data Initialization *
110   rem ***********************
120   rem ++ Input routine ++
130   dim char(255)
132   for i=0 to 255:char(i)=0:next
135   char(27)= 1: rem Escape
140   char(13)= 2: rem RETURN
145   char(200)=3: rem CRSR UP
150   char(208)=4: rem CRSR DOWN
155   char(8)  =5: rem Backspace
160   char(127)=5: rem Delete
165   for i=32 to 165 : rem possible characters
170   char(i)=6
175   next
1000  rem ++ aes-initialization ++
1010  A#=gb
1020  gcontrl=peek(a#)
1030  global=peek(a#+4)
1040  gintin=peek(a#+8)
1050  gintout=peek(a#+12)
1060  addrin=peek(a#+16)
1070  addrout=peek(a#+20)
1999  goto prgstart
2000  rem *****************************
2005  rem * Work space initialization *
2008  rem *****************************
2010  initwork:
2020  x1=0:y1=0:x2=639:y2=639
2030  findex=0:fstyle=0
2040  gosub rect
2045  seterrorarea:
2050  x1=0:y1=380:x2=639:y2=399
2060  findex=1:fstyle=1
2070  gosub rect
2080  return
```

```
2100  rem ************************
2105  rem * Set program name     *
2110  rem * name$ = program name *
2115  rem ************************
2116  setprgname:
2120  x1=0:y1=2:x2=639:y2=19
2130  findex=2:fstyle=1
2140  gosub rect
2150  poke contrl,8:poke contrl+2,1:poke
      contrl+6,len(name$)
2160  poke ptsin,(640-len(name$)*8)/2
2170  poke ptsin+2,16
2180  for i=0 to len(name$)-1
2190  poke intin+i*2,asc(mid$(name$,i+1,1))
2195  next i
2196  vdisys
2197  return
2200  rem *************************
2205  rem * Solid rectangle       *
2210  rem * x1,y1 =upper left     *
2220  rem * x2,y2 = lower right   *
2230  rem * fstyle,findex=pattern *
2240  rem *************************
2245  rect:
2248  color 1,1,1,fstyle,findex
2250  poke contrl,9:poke contrl+2,5:poke contrl+6,0
2260  poke ptsin+0,x1:poke ptsin+2,y1
2270  poke ptsin+4,x2:poke ptsin+6,y1
2280  poke ptsin+8,x2:poke ptsin+10,y2
2290  poke ptsin+12,x1:poke ptsin+14,y2
2300  poke ptsin+16,x1:poke ptsin+18,y1
2310  vdisys
2320  return
2400  rem *******************
2405  rem * Messages        *
2406  rem * mess$ = message *
2408  rem *******************
2410  message:
2420  mess$="Message: "+mess$
2430  poke contrl,8:poke contrl+2,1:poke
      contrl+6,len(mess$)
2440  poke ptsin,0:poke ptsin+2,398
2450  for i=0 to len(mess$)-1
```

```
2460  poke intin+i*2,asc(mid$(mess$,i+1,1))
2470  next i
2480  vdisys
2490  sound 1,15,12,4,10:sound 1,0,0,0,0
2500  if inp(2)<>27 then 2500
2510  gosub seterrorarea
2520  return
2600  rem *************************
2605  rem * String output         *
2606  rem * text$ =string         *
2607  rem * textstyle = type style *
2609  rem * row =1-22             *
2610  rem * column = 1-80         *
2611  rem *************************
2612  strout:
2620  rem set type style
2630  poke contrl,106:poke contrl+2,0:poke
      contrl+6,1
2640  poke intin,textstyle
2650  vdisys
2660  rem string output
2665  if row<1 or row>22 or column<1 or column>80
      then return
2666  if column+len(text$)>80 then
      text$=left$(text$,81-column)
2670  poke contrl,8:poke contrl+2,1:poke
      contrl+6,len(text$)
2680  poke ptsin,(column-1)*8
2690  poke ptsin+2,row*16+19
2700  for i=0 to len(text$)-1
2710  poke intin+i*2,asc(mid$(text$,i+1,1))
2720  next i
2730  vdisys
2740  return
2800  rem ****************
2805  rem * Clear screen *
2808  rem ****************
2810  clearscreen:
2820  x1=0:y1=20:x2=639:y2=679
2830  findex=0:fstyle=0
2840  gosub rect
2850  return
```

```
2900  rem ************************
2910  rem * Ouput character      *
2920  rem ************************
2930  rem * letter=char sent out *
2935  rem * row=1-22             *
2940  rem * column=1-80          *
2945  rem ************************
2950  chrout:
2955  poke contrl,8:poke contrl+2,1:poke contrl+6,1
2960  poke ptsin,(column-1)*8
2970  poke ptsin+2,row*16+19
2980  poke intin,letter
2990  vdisys
2995  return
3000  rem ***************************
3003  rem * Input routine           *
3004  rem * length = string length  *
3005  rem * text1$ = prompt string  *
3006  rem * text2$ = default string *
3007  rem * style1 = style prompt   *
3008  rem * style2 = style default  *
3009  rem *   /output string        *
3010  rem * row/column = position   *
3011  rem * back = repeat           *
3012  rem * 0=RET,1=down,-1=up      *
3013  rem ***************************
3015  string:
3016  if column<1 or column>80 or row<1 or row>22
      then return
3020  text$=text1$
3030  textstyle=style1
3040  gosub strout
3050  if len(text2$)>length then
      text2$=left$(text2$,length)
3055  text$=text2$+string$(length-len(text2$),"_")
3060  textstyle=style2
3070  column=column+len(text1$)
3080  gosub strout
3090  column=column+len(text2$)
3092  nxtchar:
3095  letter=95:gosub chrout
3100  letter=inp(2)
```

```
3110   on char(letter) goto
       3115,3120,3130,3140,3150,3170
3111   goto nxtchar
3115   rem ....Escape....
3116   column=column-len(text2$):text2$=""
3117   text$=string$(length,"_"):gosub strout
3118   goto nxtchar
3120   rem ++ Return ++
3125   back=0:goto jmpback
3130   rem ++ Cursor up ++
3135   back=-1:goto jmpback
3140   rem ++ Cursor down ++
3145   back=1:goto jmpback
3150   rem ++ Delete/backspace ++
3155   if len(text2$)=0 then goto nxtchar
3160   text2$=left$(text2$,len(text2$)-1)
3165   column=column-1:goto nxtchar
3170   rem ++ legal characters ++
3175   if len(text2$)=length then goto nxtchar
3180   text2$=text2$+chr$(letter)
3185   gosub chrout
3190   column=column+1:goto nxtchar
3200   jmpback:
3210   return
```

```
10000 rem *****************
10005 rem * Program start *
10008 rem *****************
10010 prgstart:
10020 gosub initwork
10030 name$="Address program":gosub setprgname
10040 mess$="Start program (ESC)":gosub message
10090 text$="Input an address:"
10095 column=20:row=4:textstyle=8:gosub strout
10100 x1=19*8:y1=6*16+17:x2=59*8:y2=16*16+17
10110 fstyle=7:findex=2:gosub rect
10120 style1=1:style2=0:length=20
10125 for i=1 to 5:adr$(i)="":next
10130 row=7:column=21
10135 text1$="First name       ":
      text2$=adr$(1):gosub string
10140 adr$(1)=text2$
10150 if back=-1 then 10130
10160 row=9:column=21
10165 text1$="Last name:       ":
      text2$=adr$(2):gosub string
10170 adr$(2)=text2$
10180 if back=-1 then 10130
10190 row=11:column=21
10195 text1$="Street address: ":
      text2$=adr$(3):gosub string
10200 adr$(3)=text2$
10210 if back=-1 then 10160
10220 row=13:column=21
10225 text1$="City/State:      ":
      text2$=adr$(4):gosub string
10230 adr$(4)=text2$
10240 if back=-1 then 10190
10250 row=15:column=21
10255 text1$="Telephone:       ":
      text2$=adr$(5):gosub string
10260 adr$(5)=text2$
10270 if back =-1 then 10220
10280 end
```

APPENDIX

Appendix A: Overview of ST BASIC Commands
Appendix B: Reserved BASIC Words
Appendix C: Problem Solutions
Appendix D: BASIC Error List

Appendix A:

Overview of ST BASIC Commands

ABS

Category: Numerical Function

Use: `ABS(X)`

Function: Returns the absolute value of the input value `X` (that is with no leading sign). The absolute value of a negative number is achieved by multiplying it by a negative one.

AND

Category: Logical Operator

Use: *<expression>* `AND` *<expression>*

Function: The logical `AND` is used in Boolean algebra to test the truth of two expressions. Result is true only if both expressions are true.

ASC

Category: String function

Use: `ASC(X)`

Function: This returns the ASCII value of the character `X`. If the input string is composed of more than one character, then only the first character of the string has its ASCII value returned.

ATN

Category: Numerical function

Use: `ATN(X)`

Function: Returns the arctangent of the value `X`. The result is in the range of $-\pi/2$ through $+\pi/2$

AUTO

Category: Command

Use: `AUTO` *(line number)*,*(increment)*

Function: This command invokes the automatic line numbering. As soon as a program line is entered after a <RETURN>, the computer displays the next line number. If we start with line 20, then the next line number will be 30. This can be changed be entering an increment after the line number in the command. The increment determines the amount of space between line numbers.

Examples: `AUTO ,10:` automatic line numbers with increments of 10.

`AUTO 20,100:` automatic line numbering with an increment of 100, starting at line 20.

The `AUTO` command is exited by entering a <CONTROL> G.

BLOAD

Category: Instruction

Use: `BLOAD` "*data*",*address*

Function: The command `BLOAD` loads data that was previously stored with a `BSAVE` into main memory. This can be used, for example, to load a picture into graphic memory. The data is loaded into main memory byte by byte, starting at the given address. If no address is given, the data is loaded into the same location from where it was `BSAVE'`d. You should be familiar with memory organization before using `BLOAD` or `BSAVE`.

Example: 10 BLOAD "*dump1*",5000

BREAK

Category: Command

Use: `BREAK` (*line number*)

Function: After the input of a `BREAK`, a running program is stopped after each line of the program. The next line of the program is run after a <CONT> or <RETURN> is pressed. If a line number is entered after the command, then the program is stopped only after that line number. The command `UNBREAK` disables the BREAK command.

Examples: `BREAK` : A program is stopped after each line.

`BREAK 50,75` : A program is stopped after lines 50 and 75.

BSAVE

Category: Instruction

Use: `BSAVE` "*data*",*address*,*length*

Function: The command `BSAVE` saves a block of main memory in a file on a diskette. The data is retrieved starting at the location address, through the region ending at address+length. You should be familiar with the organization of memory before using `BSAVE` or `BLOAD`.

Example: `BSAVE` "*dump*",5000,256

CALL

Category: Instruction

Use: `CALL ADDRESS(X,Y)`

Function: `CALL` calls a machine language subroutine. The variable `ADDRESS` contains the entry point address of the machine language routine in the memory of the computer. The parameters in the `CALL` can be changed as needed.

Example:

```
100 ADDRESS = 100000
110 CALL ADDRESS
```

In this example the machine routine must start at address 100000. In line 110 this routine is called.

CDBL

Category: Function

Use: `CDBL(X)`

Function: `CDBL` changes `X` from a single precision variable to a double precision variable. This variable will now use more space and have greater precision.

CHAIN

Category: Instruction

Use: `CHAIN` "*data*",*line*,`ALL`
`CHAIN MERGE` "*data*",*line*,`DELETE` *lines*

Function: The `CHAIN` command without `MERGE` will load the data and automatically start the program. This enters the program in memory. A program loaded in this manner is also known as an **overlay**.

When a line number is entered with the command, the program is loaded starting at the entered line number. The parameter `ALL` retains the variables from the "old" program, so that the new program can make use of the same data.

The `CHAIN MERGE` command loads the "new" program after the last line of the "old" program. The parameter `DELETE` clears the lines listed after the parameter before merging the "new" program with the "old". The line region parameter deletes ranges of lines to be deleted (for example: 100,200-300).

`CHAIN` allows a running program to use overlay techniques. Therefore a main program can load and start a subroutine.

Examples:

```
1000 CHAIN "PRINT"
```

The program `PRINT` is loaded and started.

```
1000 CHAIN "PRINT",100,ALL
```

The program `PRINT` is loaded starting at line 100. When the program is started, the variables from the "old" routine are used by `PRINT`.

```
CHAIN MERGE "LISTOUT"
```

The program `LISTOUT` is loaded immediately after the resident program; the programs are chained together and started.

CHR$

Category: String function

Use: `CHR$(X)`

Function: After calling this function the number `X` is converted to the appropriate character in the ASCII code table. The number `X` must be between 0 and 255.

CINT

Category: Function

Use: `CINT(X)`

Function: `CINT` rounds off `X` to the next higher or lower whole number. `X` must be in the range from -32768 and +32767.

Example:

```
PRINT CINT(45.69),CINT(-4.88)
```

Output: `46  -5`

CIRCLE

Category: Instruction

Use: `CIRCLE X,Y,`*rad*`,`*beg*`W,`*end*`W`

Function: Draws a circle with a center point at x and y, with a radius of *rad*. You can draw a portion of a circle (like a slice of pie) using a beginning and ending degree (*beg*W & *end*W). These are measured in 1/10ths of a degree.

Examples:

```
10 CLEARW 2
20 CIRCLE 200,100,50
```

Draws a circle with a center at `200,100` with a radius of 50.

```
10 CLEARW 2
20 CIRCLE 200,100,50,900,1800
```

This draws a quarter circle (90 degrees) starting at the 90 degree location of the circle.

CLEAR

Category: Instruction

Use: `CLEAR`

Function: This instruction clears all variables and fields, such as user defined functions.

CLEARW

Category: Instruction

Use: `CLEARW` *window#*

Function: Clears a screen window. The *window#* parameter can assume a value from 0 to 3.

0 - Edit window
1 - List window
2 - Output window
3 - Command window

CLOSE

Category: Instruction

Use: `CLOSE` *#number*

Function: The `CLOSE` instruction closes an open data line. The data buffer in memory is written to the disk drive and the line is opened. The data number is the same as the identity number used in the `OPEN` command.

The character # can be omitted. You can close more than one data line with the `CLOSE` command by entering the data lines' numbers, separated by commas. `CLOSE` without a data number closes everything.

`END`, `LOAD`, `NEW`, `OLD`, `QUIT`, `RUN`, and `SYSTEM` close all open data lines.

Example:

```
820 CLOSE 1,2
```

This closes data lines 1 and 2 which were opened previously with an `OPEN` instruction.

CLOSEW

Category: Instruction

Use: `CLOSEW` *window #*

Function: Closes a screen window. The *window #* has values from 0 through 3, with these meanings:

0 - Edit window
1 - List window
2 - Output window
3 - Command window

Warning: If you close all windows, the computer can no longer accept any input. The only way out is to restart the computer.

COLOR

Category: Instruction

Use: `COLOR` *txt,fill,line,index,style*

Function: The `COLOR` instruction sets the instructions for all graphic commands that follow. This sets the text, fill and line colors. The fill pattern is set with the parameters index and style.

The color values 0 through 15 are only for the low resolution colors. In the middle resolution then the values are only from 0 - 4. High resolution is only possible in monochrome; therefore you can only have the values 0 and 1.

The fill style 0 is a completely filled region. The style 1 fills it with all black. Style 2 allows the use of up to 24 patterns with the parameter index, where style 3 has 12 hatched patterns.

Example:

```
10 CLEARW 2:FULLW 2
20 FOR I=1 to 24
30 COLOR 1,1,1,i,2
40 PCIRCLE 25*(i-1)+15,100,20
50 NEXT I
```

The program demonstrates all 24 fill patterns of style 2:

```
10 COLOR 2,0,1:PRINT "RED SCRIPT"
```

COMMON

Category: Instruction

Use: COMMON *var1,var,...*

Function: The variables defined in a COMMON statement can be used in programs that are CHAINed together later. If all variables are to be passed to later programs, then the COMMON statement does not need to be used. The same result can be obtained using the ALL parameter to the CHAIN command. A program can have more COMMON statements than were defined in the initial program.

Example:

```
700 COMMON addresses(), character#
710 CHAIN "PRINTADR"
```

The program PRINTADR, which is loaded after the initial program, will have the array *addresses* and the variable *character#* available to it just as they were in the originating routine.

CONT

Category: Command

Use: CONT

Function: A program which has been halted with a STOP, BREAK, or CTRL-G can be resumed with the CONT command. The program continues from the point where it was stopped. You cannot use this command if the program was stopped due to an error in the program.

COS

Category: Numerical function

Use: COS(X)

Function: This command returns the cosine of the number X. X is the measure of curvature of an angle.

CSNG

Category: Function

Use: `CSNG(X)`

Function: The CSNG function returns a truncated (more simple) representation of X.

Example: `A=456.456789:PRINT CSNG(A)`

Output:

```
456.456
```

CVD, CVI, CVS

Category: Function

Use: CVD(*8-byte string*)
CVI(*2-byte string*)
CVS(*4-byte string*)

Function: These functions convert numerical values from their random access file format into a format usable in a BASIC program. **Note**: Data stored in a random access file is stored in a format different from the format used for calculations. Therefore, when a value is read in from such a file, it must be converted back to a usable format.

Example:

```
100 OPEN "R",#1,"NUMDAT"
110 FIELD #1,2 AS A$,4 AS B$, 8 AS C$
120 GET #1, SENTENCE%
130 I%=CVI(A$)
140 S!=CVS(B$)
150 D#=CVD(C$)
160 PRINT "INTEGER:";I%
170 PRINT "REAL:   ";S!
180 PRINT "D-REAL: ";D#
190 CLOSE #1
```

DATA

Category: Instruction

Use: `DATA X,Y,Z`

Function: This instruction allows you to save information in a program which can be read later using the READ command. The data can be either a number or a character. Character data must be in quotes if it contains a comma, null character, or a colon. The data is read in from left to right.

DEF FN

Category: Instruction

Use: DEF FN F(X)=X*Y

Function: You can define mathematical functions with this instruction. The functions can be used later by invoking the name of the function. Here F is the function name. (X) is a variable. X*Y is the function. The function can contain other mathematical functions.

DEF SEG

Category: Instruction

Use: `DEF SEG`

Function: `DEF SEG` affects the operation of the `PEEK` and `POKE` command. If the instruction is used with a value which is greater than zero, then `PEEK` and `POKE` are only valid with single precision values. If the value is zero, then both commands are valid with double precision values.

DEFDBL

Category: Instruction

Use: `DEFDBL X-Y`

Function: This defines a single variable or a whole range of variables as being double precision. For these variables all variables in the definition are automatically determined to be double precision.

Example: `DEFDBL A-B`

This example defines all variables that begin with an `A` or `B` as being double precision.

DEFINT

Category: Instruction

Use: `DEFINT X-Y`

Function: This defines a variable or a region of variables as being integer variables. This means all variables defined by this instruction will be automatically determined to be integer variables.

Example: `DEFINT D-E`

Defines all variables starting with D or E as integer variables.

DEFSNG

Category: Instruction

Use: `DEFSNG X-Y`

Function: This defines a character or range of characters as a real variable. All variables that lie in this range are automatically determined to be real variables.

Example: `DEFSNG F-G`

This example defines all variables beginning with the characters F or G as a single precision real variable.

DEFSTR

Category: Instruction

Use: `DEFSTR X-Y`

Function: This instruction defines a character or range of characters as string variables. Any variable in this range will automatically be determined as string variables.

Example: `DEFSTR H-I`

This example interprets all variables beginning with an H or I as string variables.

DELETE

Category: Command

Use: `DELETE` *line#-line#*

Function: This command clears one or more program lines from a program. This is especially useful if you want to write over or replace only a segment of an existing program. The syntax of this command is similar to the `LIST` command.

Examples: `DELETE 10` clears line number 10
`DELETE -100` clears all lines up to 100
`DELETE 200-500` clears lines from 200 to 500.
`DELETE 500-` clears all lines from 500 to the program end.

DIM

Category: Instruction

Use: `DIM A(X)`

Function: This instruction dimensions a field (for arrays) or for a matrix (more than one dimension). `(X)` is the index into the matrix; you may also have heard of an indexed variable, in this case `A(X)`. `A(X)` is called a one-dimensional matrix and A(X,Y) is called a multi-dimensional matrix.

DIR

Category: Command

Use: `DIR` *drive:mask*

Function: The `DIR` command gets a listing of the contents of a disk in the specified disk drive. If the command is given without a mask, the entire contents of the disk are displayed. The mask specifies certain groups of data from the disk to be displayed.

Example:

`DIR A:`	Lists all data from drive A.
`DIR A:*.BAS`	Lists everything with a .BAS extension.
`DIR B:A*.*`	Lists everything on disk drive B that begins with an A.
`DIR A:?TEST.BAS`	Lists all programs having the characters `TEST` in positions 2-5 (`ATEST.BAS, BTEST.BAS`...)

EDIT

Category: Command

Use: `EDIT` *(line#)*

Function: `EDIT` turns on the BASIC editor. You enter all input in the `EDIT` window. Here you can change line numbers or remove them. You can exit the editor with the command `EXIT`.

ELLIPSE

Category: Instruction

Use: `ELLIPSE` X,Y,*Xrad*,Y*rad*,*beg*W,*end*W

Function: This command is the same as the `CIRCLE` command, except that there are two radii (one for the `X`-axis and one for the `Y`-axis). A section of an ellipse can be done by setting the beginning and ending angles of the ellipse (in 1/10th degree increments).

Example:

```
10 CLEARW 2
20 ELLIPSE 200,100,50,40
30 ELLIPSE 210,110,50,40,450,1350
```

This draws an ellipse with a center at `200,100` with a horizontal radius of 50 and a vertical radius of 40. The second ellipse is only 90 degrees beginning at 45 degrees.

END

Category: Command

Use: `END`

Function: When a running program comes across this command the program will immediately halt, and the cursor will show in the Command window. An `END` can show up at the very end of a program or in the middle, if subroutines follow.

EOF

Category: Function

Use: `X=EOF(`*number*`)`

Function: This function determines whether or not the end of a sequential or random file has been reached. If the end of the file has been reached, the function returns the value `-1`. This lets us read a file of any length without knowing the length beforehand. The parameter is the number of the opened file.

Example:

```
100 OPEN "I",1,"FILE"
110 WHILE NOT EOF(1)
120 LINE INPUT #1,DAT$
130 PRINT DAT$
140 WEND
150 CLOSE 1
```

This example reads the contents of a line and then outputs it until the end of the file `FILE` has been reached.

EQV

Category: Logical Operator

Use: `X EQV Y`

Function: This compares the variables `X` and Y (bitwise) in a negated exclusive `OR`, according to boolean algebra theory (see Chapter 1.6).

Example: `PRINT 1 EQV 1`

Output: `-1`

ERA

Category: Command

Use: ERA *drive# : name*

Function: This command erases a file on the given drive.

Example: `ERA A:TEST1.BAS`

This erases a file `TEST1.BAS` on disk drive `A`.

ERASE

Category: Instruction

Use: `ERASE X$`

Function: Clears the fields of a previously dimensioned array. After this command the array can be redimensioned. `ERASE X$` clears the array `X$`.

ERL,ERR

Category: Function

Use: `A=ERR` or `B=ERL`

Function: `ERL` and `ERR` are system variables—you cannot use these two variables for your own use. You can only use the contents of the variables as the computer sets them. You are not allowed to change the contents (as in `IF ERR=96 THEN...`). These variables are usually used after error trapping has been enabled with `ON ERROR GOTO`. If an error occurs during a program, the line where the error occurred is in the variable `ERL` and the error number is in `ERR`.

ERROR

Category: Instruction

Use: `ERROR X`

Function: `ERROR` simulates an error like `NUMBER TOO LARGE` (error code 6). This instruction you will override the `ERR` and `ERL`. You can still use the `ON ERROR GOTO` error trapping routine to handle any errors.

EXP

Category: Numerical function

Use: `EXP(X)`

Function: This gives the `X`th power of the constant `e` (2.7182818823) where `X` cannot be greater than 43.6682.

FIELD

Category: Instruction

Use: `FIELD` # *length* `AS` *string*,...

Function: This instruction is used in conjunction with a random-access file. `FIELD` defines the structure of each record of a random access file. Next to the file number (`#`) are the field length(s) and the associated variable with each field. The `FIELD` instruction doesn't read any data from the file and doesn't effect the random access file buffer in any way.

The string variables associated with a field cannot be used like other string variables. These are managed in the string memory, rather than the random access data buffer. To put the data into a usable form, use the commands `LSET` and `RSET`. You can use a single variable (let's say `A$`) to encompass more than one other variables (like `C$` and `D$`).

Examples:

```
10 OPEN "R",#1,"CHARACTERS",122
20 FIELD #1,2 AS KNUMBER$,30 AS KNAME$,
30 AS KFIRST$, 30 AS KSTREET$,
30 AS KPLACE$
30 INPUT "Character #:";A :
   LSET KNUMBER$=MKI$(A)
40 INPUT "Name:       ";A$:
   LSET KNAME$=A$
50 INPUT "First Name: ";B$:
   LSET KFIRST$=B$
60 INPUT "Street:     ";C$:
   LSET KSTREET$=C$
70 INPUT "Place:      ";D$:
   LSET KPLACE$=D$
80 PUT #1,1
90 GET #1,1
100 KNUM%=CVI(KNUMBER$)
110 PRINT KNUM%,KNAME$,KFIRST$,
    KSTREET$,KPLACE$
```

The previous example demonstrates not only how to use the FIELD instruction, but also other instructions used with random access files. This field statement is for a random access file with a record length of 122 characters. After the description of the fields in the file is the output statement PUT #1,1 which writes the record 1 to file #1. The last few lines are to input the information from the file and print out the contents.

```
20 FIELD #1,4 AS PLZ$, 20 AS PLACE$
30 FIELD #1,24 AS PLZPLACE$
```

In this example the FIELD instructions are for input from a file. The second instruction is equivalent to the first, except for that PLZPLACE$ contains both of the previous string variables in a single variable.

FILL

Category: Instruction

Use: `FILL X, Y`

Function: The FILL command fills the area where the coordinate point X,Y is located. The region is filled with the colors and attributes of a previously defined COLOR statement.

Example:

```
10 COLOR 1,2,1
20 ELLIPSE 200,100,50,40
30 FILL 200,100
```

This program draws an ellipse filled with solid red.

FIX

Category: Functions

Use: `FI`

Function: `FIX` truncates the decimal portion of the variable `A`. The variable is not rounded.

Example:

```
PRINT FIX(45.9),FIX(45.3)
```

Output:

```
45    45
```

FLOAT

Category: Functions

Use: `A=FLOAT(X)`

Function: `FLOAT` converts an integer number `(X)` into a floating point number `(A)`. `X` must be in the range -32768 through +32767.

FOLLOW

Category: Command

Use: `FOLLOW`(variable)

Function: This command allows the contents of the variable in the parenthesis to be observed through the program. Every time the variable changes it is printed out with the line number where the change occurred. This command is turned off with the command `UNFOLLOW`.

FOR...TO...(STEP)

Category: Instruction

Use: `FOR X=1 TO 10 STEP 2`

Function: This instruction allows you to use a variable (in this case `X`) as a counter. You have to specify the starting value (in this case 1) and the ending value `(10)` and the step size `(2)`. If you don't have a step in the instruction, you will automatically have a step size of 1. The step size can be any floating point number.

FRE

Category: Numerical function

Use: FRE(X)

Function: This function outputs the number of free bytes left for you to use for your program. `X` can take on any value you want since it is not used in the calculation of the amount of free space.

FULLW

Category: Instruction

Use: `FULLW` *window#*

Function: `FULLW` sets a screen window to the full size of the screen. The values 0 through 3 determine which window:

0 = Edit window
1 = List window
2 = Output window
3 = Command window

GEMSYS

Category: Function

Use: `GEMSYS(`*function#*`)`

Function: `GEMSYS` calls a specific GEM-AES function with a function number. The communication with the AES functions is a result of a lot of pointer arrays whose address is a pointer array with the address in `GB`, a reserved variable (see Chapter 7).

GET

Category: Instruction

Use: `GET #`*number,file line*

Function: `GET` reads a line of data from the Random Access buffer, which had been defined in a `FIELD` statement. Next to the file number there must exist a record number. This number must be in the range of 1 to 32767. If the record number is not given, then the record following the last `GET` record number is read.

Example:

```
10 OPEN "R",#1,"CHARACTER",122
20 FIELD #1,2 AS KNUMBER$,30 AS KNAME$,
30 AS KFIRST$,30 AS KSTREET$,
30 AS KPLACE$
30 GET #1,15
```

This example reads the fifteenth record of the random access file `CHARACTER` from the buffer.

GOSUB

Category: Instruction

Use: `GOSUB XX` or `GOSUB` label

Function: This command allows you to jump to a subroutine within the program (at line number `XX` or at a jump marker label). When the program reaches a `RETURN` statement in the subroutine, it will automatically return to the calling procedure at the line following the call.

GOTO

Category: Instruction

Use: `GOTO XX` or `GOTO` *label*

Function: This command allows you to jump to a specific line number `XX` or to a jump marker label. Using this you can jump to (or over) any line in a program you wish.

GOTOXY

Category: Instruction

Use: `GOTOXY` *line,cloumn*

Function: This instruction positions the cursor at the specified *line* and *column.*

Example:

```
10 GOTOXY 20,10
20 PRINT "TEXT AT COLUMN 20, LINE 10"
```

This function doesn't work all of the time in the present version of BASIC. The screen columns are misinterpreted.

HEX$

Category: Function

Use: `HEX$(X)`

Function: This function changes the decimal value of X into its hexadecimal equivalent.

Example:

```
PRINT HEX$(255)
```

Output: `FF`

IF...THEN...ELSE

Category: Instruction

Use: `IF...THEN...ELSE`

Function: This command tests a statement, and if the statement is true react to it accordingly. If the result of the test is true, then the portion of code after the THEN is performed. If the test is false then the code after the ELSE is performed.

Examples:

```
110 IF A$="J" THEN A=5 ELSE GOSUB 1000
```

If the variable A$ was not equal to J then the program would call the subroutine starting at line 1000. If the test were true then the variable A is set to 5.

```
10 INPUT 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 code only accepts input of 0 through 9 and the letter E.

IMP

Category: Logical Operator

Use: `X IMP Y`

Function: This performs a bitwise comparison of the variables `X` and `Y` according to the rules of implication (see Section 1.6).

Example: `PRINT -1 IMP 0`

Output: `0`

INKEY$

Category: Function

Use: `A$=INKEY$`

Function: The `INKEY$` function reads the next character from the keyboard buffer and puts it into the variable (in this case `A$`).

INKEY$ doesn't work in the current version of BASIC.

In place of this we can use INP(2) to poll the keyboard buffer (see Section 3.4).

INP

Category: Function

Use: `X=INP(`*port*`)`

Function: Reads a character from the port which is given as a parameter.

Port numbers:
0 - Printer (parallel port)
1 - RS-232 (serial port)
2 - Consol (monitor)
3 - MIDI interface
4 - keyboard

Example:

```
10 A=INP(2):IF A=0 THEN 10
20 A$=CHR$(A)
```

This short routine reads a character from the keyboard and then places it in the variable `A$`. You can use these lines as a replacement for the `INKEY$`.

INPUT

Category: Command

Use: `INPUT X` or `INPUT"`*comment*`";X`

Function: This command allows the user to enter data into a program in an interactive mode. When a program reaches this command the program halts, prints a `?` and displays the cursor. After this the user can enter the needed data. The program continues after the user hits the <RETURN> key. You can either use `INPUT` as described, or you can have a comment printed out describing the type of data required, followed by a semicolon and the variable. This disables the printing of the `?` prompt.

INPUT

Category: Instruction

Use: `INPUT` *#number,variable,...*

Function: This instruction works similarly to the `INPUT` command preceding. In this instruction, the data doesn't come from the keyboard, but from a sequential file. The end of a line in the file is marked by a <RETURN> (ASCII 13) , LINEFEED (ASCII 10), Comma, or a maximum of 255 characters.

Example:

```
10 OPEN "I",#1,"BOOKS"
20 INPUT #1,ACCOUNT%,GACCT%,AMT%,TEXT$
```

This reads the fields of the file `BOOK`. Numeric and string fields can exist in the same file.

INPUT$

Category: Function

Use: `INPUT$`(*length,#number*)

Function: `INPUT$` reads a string of a given length from the file attached to the file number in the parameter list. If no file number is given, the INPUT$ gets its information from the keyboard without printing the characters on the screen.

Example:

```
10 PRINT "CODEWORD"[";
20 CODE$=INPUT$(6)
30 IF CODE$="JOSHUA" THEN 100 ELSE END
100 PRINT "Welcome!!"
```

INSTR

Category: Function

Use: `INSTR (`*stnr*`,`*str1*`,`*str2*`)`

Function: This function gives the position of the substring (*str2*) inside of the string (*str1*). If the substring is not found, then the return value is set to 0. The parameter *stnr* specifies a position after which the search will start.

Example:

```
10 A$="Atari ST":B$="ST":C$="ari"
20 P1=INSTR(A$,B$)
30 P2=INSTR(2,A$,C$)
40 PRINT P1,P2
```

Output: `7    3`

Line 20 searches for variable `A$` starting at the first character for the substring `B$`. The result is placed in variable `P1`. `P1` is set to 7, which means that the substring `B$` was found starting at the seventh character from the beginning. Line 30 searched the string `A4` starting at the second character for the substring `C$`. The result is 3.

INT

Category: Numerical function

Use: `INT(X)`

Function: This function returns the whole number portion of the variable `X`. No rounding off takes place. The result is always smaller or equal to the original contents of `X`.

KILL

Category: Instruction

Use: `KILL` "*name*"

Function: Erases the file "*name*" from the diskette in the default disk drive. This instruction is different than the `ERA` command in that you have to input a string with this instruction.

Example:

```
10 INPUT "FILENAME:";D$
20 KILL B$
```

LEFT$

Category: String Function

Use: `LEFT$(X$,A)`

Function: This function returns a substring of characters from `X$` starting at the leftmost character and finishing up after `A` characters. The function `LEFT$(X$,4)` returns the leftmost 4 characters of `X$`. `A` can have any value between 0 and 255.

LEN

Category: Numerical function

Use: `LEN(X$)`

Function: This function returns the number of characters in the string `X$`. It counts all characters, including spaces.

LET

Category: Command

Use: `LET X=5`

Function: This command allows a value to be placed into variable. In this case the variable `X` gets the value 5. The word `LET` can be omitted in this command (for example, `X=5`).

LINEF

Category: Instruction

Use: `LINEF X1,Y1,X2,Y2`

Function: This draws a line from the coordinate point `X1,Y1` to the point `X2,Y2`. If you want to only draw a single point, then `X1,Y1` should be the same point as `X2,Y2`.

Example:

```
10 CLEARW 2
20 LINEF 0,0,639,199
```

This program draws a diagonal line across the color screen.

LINE INPUT

Category: Instruction

Use: `LINE INPUT "`*comment*`";A$`

Function: This instruction is almost identical to the `INPUT` instruction. The difference between the two is that `LINE INPUT` returns a full line (255 characters) as the answer. This instruction doesn't have a question mark like the `INPUT` command. This instruction requires a comment be included.

LINE INPUT

Category: Instruction

Use: `LINE INPUT` #*number,string*

Function: This instruction is the same as the previous, except that it reads information from a file. The maximum length of input is 254 characters followed by a <RETURN> (ASCII 13). Therefore, this instruction can also read commas as part of the data, unlike other input methods.

Example:

```
10 OPEN "O",#1,"TEXT"
20 PRINT "Input text (**=end)";
30 LINE INPUT TEXT$
40 IF TEXT$="**" THEN 60
50 PRINT #1,TEXT$:GOTO #0
60 CLOSE #1
100 OPEN "I",#1,"TEXT"
110 WHILE NOT EOF(1)
120 LINE INPUT #1,TEXT$
130 PRINT TEXT$
140 WEND
150 CLOSE #1
```

This routine writes the text input from the keyboard to the file `TEXT$`. After all of the text has been entered, the file is closed, reopened and printed out.

LIST

Category: Command

Use: `LIST` *line#* *-line#*

Function: The `LIST` command allows you to view a single line number or a range of line numbers. If only the command `LIST` is entered, then the entire program is listed.

`LIST` Shows the entire program
`LIST 10` Shows program line 10
`LIST -100` Shows all lines up to 100
`LIST 100-` Shows all lines after 100
`LIST 10-20` Shows lines from 10 to 20

LLIST

Category: Command

Use: `LLIST` *line#-line#*

Function: This command is exactly like the `LIST` command except the output is printed to a printer rather than the screen.

LOAD

Category: Command

Use: `LOAD` *name*

Function: This command loads a program from diskette and overwrites whatever is currently in memory. If no extension is given (for example, `.DAT`), then the command will automatically search for a file with extension `.BAS`.

Example `LOAD TEST`

This command loads the program `TEST.BAS`.

LOF

Category: Function

Use: `X=LOF(#`*number*`)`

Function: This function returns the length of the file associated with the file number in the parameter list. This file must have been previously opened.

Example:

```
10 OPEN "I",#1,"TEXT"
20 IF LOF(#5)>32000THEN PRINT"FILE FULL"
```

LOG

Category: Numeric Function

Use: `LOG(X)`

Function: This function returns the natural logarithm (base e) of X, where X must be greater than 0.

LOG10

Category: Numeric Function

Use: `LOG10(X)`

Function: This function returns the decimal logarithm (base 10) of X, where X must be greater than 0.

LPOS

Category: Function

Use: `LPOS(X)`

Function: This function returns the position of the printhead in the print buffer. The value is the number of characters from the last carriage return. Due to the special print characters that the printer can handle, this function is not always completely accurate.

LPRINT

Category: Instruction

Use: `LPRINT` (*variable*) ; "*text*"

Function: This instruction is similar to the `PRINT` or `PRINT USING` instructions. The result is printed on the printer.

LSET

Category: Instruction

Use: LSET *stringvar* = *string*

Function: This instruction converts data to be used later in a PUT statement. If the string to be put in the file is less than the maximum size, the string is left justified and filled with spaces on its right. If the string is longer than the maximum, the rightmost characters are truncated to fit.

The following program writes a record to the random access file CHARACTER: Example:

```
10 OPEN "R",#1,"CHARACTER",122
20 FIELD #1,2 AS KNUMBER$,30 AS KNAME$,
   AS KFIRST$,30 AS KSTREET$,
   AS KPLACE$
30 LSET KNUMBER$=MKI$(1250)
40 LSET KNAME$="JONESMAM"
50 LSET KFIRST$="HENRYSIR"
60 LSET KSTREET$="PICCADILLY 10"
70 LSET KPLACE$ = " 212 CANDYCURREN"A
80 PUT #1,1
90 CLOSE #1
```

MERGE

Category: Command

Use: `MERGE` *name*

Function: This command brings in the program *name* from diskette into memory with the current program. Any duplicate line numbers results in the "old" line numbers being overwritten by the "new" line number. This command can be very helpful during program development.

Example: Program in memory:

```
10 PRINT "Line 10 of the old program"
20 PRINT "Line 20 of the old program"
30 PRINT "Line 30 of the old program"
```

The `PROG2.BAS` on diskette:

```
15 PRINT "Line 15 of the new program"
20 PRINT "Line 20 of the new program"
MERGE PROG2
```

Result:

```
10 PRINT "Line 10 of the old program"
15 PRINT "Line 15 of the new program"
20 PRINT "Line 20 of the new program"
30 PRINT "Line 30 of the old program"
```

MID$

Category: Instruction/Function

Use: `MID$(X$,A,B)`

Function: This function returns the characters from character position `A` for `B` characters from the string `X$`. `A` and `B` can both have values between 0 and 255.

Example:

```
10 A$="Atari ST XXX"
20 MID$(A$,10,3)="520"
30 PRINT A$
40 END
```

Output: `Atari ST 520`

MKD$, MKI$, MKS$

Category: Function

Use:

```
X$=MKD$(num)
X$=MKI$(int)
X$=MKS$(num)
```

Function: This function changes the number in the parameter list to the format needed for a random access file. This is used before the `LSET` command. The function is for 2-byte (`MKI$`), 4-byte (`MKS$`), and 8-byte (`MKD$`)

Example:

```
10 OPEN "R",#1,"NUMBERS",14
20 FIELD #1,2 AS INTEGER$,4 AS SINGLE$,
   8 AS DOUBLE$
30 LSET INTEGER$=MKI$(I%)
40 LSET SINGLE$=MKS$(S!)
50 LSET DOUBLE$=MKD$(D#)
60 PUT #1
70 CLOSE #1
```

MOD

Category: Arithmetic Operator

Use: `X MOD Y`

Function: This returns the remainder of the division of X by Y.

Example: `PRINT 64 MOD 6`

Output: `4`

NAME

Category: Instruction

Use: `NAME` "*oldname*" `AS` "*newname*"

Function: This instruction renames a file with the name *oldname* to a file with the name `newname`.

Example: `NAME "TEST1.BAS" AS "TEST2.BAS"`

NEW

Category: Command

Use: `NEW`

Function: This command deletes the current program in memory, and clears all the variables in RAM. Use this command with caution, it will completely delete the program currently in memory. As a general rule, the `NEW` command should be given at the beginning of each new program to insure that all variables are initialized properly.

NEXT

Category: Instruction

Use: `NEXT X`

Function: This instruction indicates the end of a loop that was started with a statement like `FOR X=0 TO 10`. When the program reaches this instruction, the variable `X` is increased by the step size. Then the variable `X` is compared against the exit condition—in this case, `10`. If `X`>10 then the loop is finished. Otherwise the program returns to the top of the loop. A single instruction can be used to close more than one loop. For example: `NEXT X,Y,Z`.

NOT

Category: Logical Operator

Use: `NOT X`

Function: The `NOT` operator reverses the result of a comparison that is usually true. For example, `TRUE=-1/FALSE=0` is actually reversed to `TRUE=0/FALSE=-1`.

OCT$

Category: Function

Use: `OCT$(X)`

Function: This function converts the decimal value `X` into its octal equivalent. `X` must be in the range from -32768 through +32767.

Example: `PRINT OCT$(8)`

Output: `10`

OLD

Category: Command

Use: `OLD` *file*

Function: This is identical to the `LOAD` command.

Example:

```
OLD MINE
```

This loads the program `MINE.BAS`

ON

Category: Instruction

Use:

```
ON X GOTO 10,MARK
ON X GOSUB 10,20
```

Function: This command lets you jump to different line numbers or markers for different values of `X`. If `X` has the value 1, then the program jumps to line 10 for the `GOTO` or as a subroutine for `GOSUB`. You get the same results if you use different `IF..THEN..` commands for each value.

ON ERROR GOTO

Category: Instruction

Use: `ON ERROR GOTO` *line number*

Function: The `ON ERROR GOTO` instruction automatically jumps to the line number in the parameter list whenever an error occurs in the program. Using a line number of 0 with the `ON ERROR GOTO` returns error handliing to ST BASIC.

Example:

```
10 ON ERROR GOTO 100
```

The program automatically jumps to line 100 if an error occurs.

OPEN

Category: Instruction

Use: `OPEN` *"mode", #number, "name", length*

Function: The `OPEN` command opens a sequential or random access file. The parameter `len` is for the length of a record. The *length* must be given for a random access file.

The following modes are available (Capital letters only; must be enclosed in quotation marks):

```
"O"  - sequential file, output
"I"  - sequential file, input
"R"  - random access, input & output
```

Records for a random access file begin with record 1 and must be consecutively numbered thereafter.

Example:

```
10 OPEN "O",#1,"ADDRESSES"
```

This sequential file `ADDRESSES` is opened for output.

```
20 OPEN "I",#1,"ADDRESSES"
```

The sequential file `ADDRESSES` is opened for input.

```
30 OPEN "R",#1,"CHARACTERS",122
```

The random access file `CHARACTERS` is opened with a record length of 122.

OPENW

Category: Instruction

Use: `OPENW` *window#*

Function: Opens a screen window previously closed with a `CLOSEW` command. The *window#* parameter has values from 0 to 3:

0 = Edit window
1 = List window
2 = Output window
3 = Command window

OPTION BASE

Category: Instruction

Use: `OPTION BASE X`

Function: This instruction allows you to define whether the base address of an array should be 1 or 0. `X` can be assigned the values 1 or 0 only. Zero is the default for an array. If you enter `OPTION BASE 1`, the lowest element of the array is `A(1)`, rather than `A(0)`.

OR

Category: Logical Operator

Use: (*expr*) `OR` (*expr*)

Function: The logical operator can take more than two expressions as input for comparison. The `OR` operator only requires a single TRUE result out of the entire list of expressions in order for the entire expression to be TRUE.

OUT

Category: Instruction

Use: `OUT` *channel,int*

Function: The *int*eger version of a character (0-255) is output to the port defined by *channel*. There are 5 possible ports:

0 - Printer (2 parallel ports)
1 - RS-232 (serial port)
2 - Console (monitor screen)
3 - MIDI interface

Example:

```
110 OUT 0,13
```

This prints a <RETURN> to a parallel printer. It's the same as an `LPRINT CHR$(13)`, except that the `LPRINT` command automatically issues a line feed afterwards. That is why `LPRINT` cannot be used with graphics.

PCIRCLE

Category: Instruction

Use: `PCIRCLE X,Y,`*rad*`,`*beg*`W,`*end*`W`

Function: This instruction draws a circle with a center at `X,Y` and with a radius of *rad*. You can draw a portion of a circle by entering the location of the starting angle and the ending angle. The partial circle angles are entered in 1/10ths of a degree. This instruction draws a circle with the color and pattern defined by the `COLOR` command.

Example:

```
10 CLEARW 2
20 COLOR 1,2,1
30 PCIRCLE 200,100,50
```

This draws a whole circle in red with a center point at `200,100` and a radius of `50`.

```
10 CLEARW 2
20 COLOR 1,1,1,2,3
30 PCIRCLE 200,100,50,900,1800
```

This draws a black segment of a circle, starting at 90 degrees and ending at 180 degrees.

PEEK

Category: Function

Use: `A=PEEK(`*address*`)`

Function: `PEEK` reads the contents of the address in the parameter list. `PEEK` is dependent upon the last `DEF SEG` instruction (see `DEF SEG`).

PELLIPSE

Category: Instruction

Use: `PELLIPSE X,Y,`*Xrad*`,`*Yrad*`,`*begW*`,`*endW*

Function: This is the same as the instruction `PCIRCLE`, except that a radius for both the `X` and Y axes must be specified, rather than a single constant radius.

Example:

```
10 CLEARW 2
20 COLOR 1,2,1
30 PELLIPSE 200,100,50,40
40 COLOR 1,1,1,2,3
50 PELLIPSE 210,110,50,40,450,1350
```

This draws a red ellipse at `200,100` with an `X` radius of `50` and a `Y` radius of `40`. Then the program draws a 90 degree segment of a black ellipse at `210,110`, from 45 to 135 degrees.

POKE

Category: Instruction

Use: `POKE X,A`

Function: This instruction is the opposite of PEEK. It writes the contents of A to memory address X. The A parameter can have the values from 0 to 255 only. As with PEEK, the POKE instruction is dependent upon the last DEFSEG command given.

POS

Category: Function

Use: `POS(A)`

Function: POS(A) returns the cursor position where the next PRINT will start on the current line on the screen. The value in A is meaningless, much like in the function FRE(A).

PRINT

Category: Instruction

Use:

```
PRINTvar
PRINT "TEXT"
```

Function: The PRINT command is probably the most commonly used in BASIC. If a variable follows the PRINT command, the contents of the variable are printed. If a string follows the PRINT command, the string is printed.

PRINT USING

Category: Instruction

Use: `PRINT USING "format";A`

Function: This instruction is used to format the output of a PRINT command. The format of the output is defined in the string immediately following the USING.

For numeric data output:

#	number of characters to be printed
+	print a + in front of pos. numbers.
-	print a - in front of neg. numbers.
.	give the location of the decimal pt.
**	fill blanks with *'s.
,	every third position with a comma.
**$	combine a $ with *'s.
^^^^	numbers are printed in scientific notation.
_	underline the next char.

For text output:

!	print only the first character.
**	print more characters.
&	print the complete string.

Example: Print the following values:

```
10 A=34.57:B=1234:C=5421.236:D=546320
20 PRINT USING "  ####.##";A,B
30 PRINT USING "  ####.##";C,D
```

Output:

```
  34.57  1234.00
5421.24 %546320.00
```

PRINT

Category: Instruction

Use: `PRINT` *#number,output...*

Function: This instruction is like the normal `PRINT`, except that instead of writing the output to the screen, the output is sent to a specified sequential file. The sequential file is specified with the *#number* of an `OPEN` command. If you want to print more than a single item of data, you must seperate the data with commas or semicolons.

Example:

```
10 OPEN "O",#1,"TELEPHONE"
20 N$="REAGAN"
30 V$="RONALD"
40 T$="001/202-4561414"
50 PRINT #1,N$;",";V$;",";T$
60 CLOSE 1
```

This example saves names for a telephone file. The data can be read back in with an `INPUT #` instruction, with the same three fields seperated by commas. The commas in the quotes are also written to the file on the diskette. The record in the file would look like:

```
REAGAN,RONALD,001/202-4561414
```

PUT

Category: Instruction

Use: `PUT` *#number,record number*

Function: This instruction is used to read data from a random access file into the random access data buffer. The buffer is organized with a `FIELD` instruction and the data is formatted with an `LSET` instruction.

Example:

```
10 OPEN "R",#1,"CHARACTERS",122
20 FIELD #1,2 AS KNUM$,30 AS KNAME$,
30 AS KFIRST$,30 AS KSTREET$,
30 AS KPLACE$
30 LSET KNUM$=MKI$(1250)
40 LSET KNAME$="JONESMAM"
50 LSET KFIRST$="HENRYSIR"
60 LSET KSTREET$="PICCADILLY 10"
70 LSET KPLACE$=" 212 CANDYCURREN"
80 PUT #1,1
90 CLOSE #1
```

This example stores the data in the quotes into the random access file `CHARACTERS`.

QUIT

Category: Command

Use: `QUIT`

Function: The `QUIT` command corresponds to the `SYSTEM` command. It is used to exit from BASIC. A `QUIT` command closes all open files, and returns you to the GEM desktop.

RANDOMIZE

Category: Instruction

Use: `RANDOMIZE X`

Function: The RANDOMIZE instruction gives the random number generator a new initialization value. X must be in the range from -32768 through +32767.

READ

Category: Instruction

Use: `READ X`

Function: This instruction reads an element from a DATA line and assigns the value to the variable X. The type of the element in the DATA line must be the same as the variable type in the instruction.

Example:

```
10 FOR I=1 TO 4
20 READ A
30 DATA 10,20,30,40
40 PRINT A:NEXT
50 END
```

REM

Category: Instruction

Use: REM *text*

Function: A REM instruction is a remark about a segment of a program, within the program itself. It's like a notepad to leave notes to yourself about the program segment's purpose. All text immediately following a REM is ignored by the computer during program execution.

RENUM

Category: Command

Use: `RENUM` *new start line,old start line,increment*

Function: This command renumbers the program line numbers in the current program. The increments are determined by the difference between two lines. The default is 10.

Example:

```
RENUM
```

Renumbers the program starting at 10 and incrementing line numbers by 10.

```
RENUM 1000,1,20
```

The new first line is 1000, the 1 is the first line of the old program, and the increment is 20.

REPLACE

Category: Instruction

Use: `REPLACE` *file,line range*

Function: This instruction works in much the same manner as the `SAVE` command. The `REPLACE` command saves a program on disk. If a file already exists with the same name, that file is overwritten. If a line range is given, only those specified lines are replaced. If no filename is specified, then the last filename used in a `LOAD` or `OLD` command is used in its place.

Example:

```
LOAD FIRSTDAT
1000 A=X+2*Y/4
5 REM *** VERSION 2.0 ***
REPLACE
```

The preceding lines load the program `FIRSTDAT`, and change lines 1000 and 5. Finally the changes are saved in the same file by simply typing the command `REPLACE`.

RESET

Category: Instruction

Use: `RESET`

Function: This instruction copies the output window to the graphic memory region, where it can be saved on diskette if necessary (if the BUF GRAPHICS menu is activated). After clearing the window, you can return the graphic area back to the output window with an OPENW 2.

If you don't use this option, you can turn off the BUF GRAPHICS. This will increase the size of BASIC memory by 32K.

Example:

```
10 COLOR 1,1,1,1,1
20 FULLW 2:CLEARW 2
30 CIRCLE 200,100,50
40 RESET
50 FOR I=1 TO1000:NEXT
60 CLEARW 2
70 PCIRCLE 200,100,50
80 FOR I=1 TO 1000: NEXT
90 OPENW 2
```

RESTORE

Category: Instruction

Use: RESTORE *line number*

Function: This instruction sets the pointer for the DATA lines back to the first element. Therefore it is possible to READ the same value more than once from the same DATA line. You can specify the DATA line of one line number only be RESTOREd.

RESUME

Category: Instruction

Use: `RESUME` *(line number)*`/NEXT`

Function: `RESUME` is used much like the `RETURN` statement at the end of a subroutine, except that `RESUME` is used at the end of an error trapping routine. Therefore, you have to use the `ON ERROR GOTO` statement. If you enter `RESUME` alone, the program will return to the line where the error occurred. If you use the `NEXT` option, the program returns to the next line after the error. You can also specify a line number that the program is to return to at the end of the error routine.

RETURN

Category: Command

Use: `RETURN`

Function: This command marks the end of a subroutine. When a program reaches this command, control is returned to the line following the `GOSUB` command. The program continues execution there.

RIGHT$

Category: String Function

Use: `RIGHT$(X$,A)`

Function: This function returns the substring of `X$` (to the right of `X$`) through the first `A` characters. `A` can contain values ranging from 0 to 255. If the value is greater than the length of the string, then the entire string is returned.

RND

Category: Numerical function

Use: `RND(X)`

Function: This function returns a random number between 0.0 and 1.0. A positive value of X gets the next random number, a value of 0 gets the last random number generated (without regeneration). A negative value resets the random number generator and returns the first value of the new sequence.

RSET

Category: Instruction

Use: RSET *stringvar=string*

Function: This instruction prepares data for the PUT instruction, where the data goes to the random access file. This instruction is necessary because a buffer string variable (FIELD) cannot be used with a LET statement. If the string variable has fewer characters than defined in the FIELD statement, then the data is right-justified. The remaining left characters are filled with blank spaces. If the string variable is longer than the defined region, then the variable is truncated.

Example:

```
10 OPEN "R",#1,"CHARACTERS",122
20 FIELD #1,2 AS KNUM$,30 AS KNAME$,
30 AS KFIRST$,30 AS KSTREET$,
30 AS KPLACE$
30 RSET KNUM$=MKI$(1250)
40 LSET KNAME$="JONESMAM"
50 LSET KFIRST$="HENRYSIR"
60 LSET KSTREET$="PICCADILLY 10"
70 LSET KPLACE$="212 CANDYCURREN"
80 PUT #1,1
90 CLOSE #1
```

This writes a record to the random access file CHARACTERS. The numeric field is right-justified, the others are left-justified.

RUN

Category: Command

Use: `RUN` *line number*

Function: `RUN` starts program execution. You can specify the line number you want the program to start execution. Doing so automatically resets all variables to zero. If you want the variables to stay as they are, use a `GOTO` statement with the appropriate line number.

SAVE

Category: Command

Use: `SAVE`*file,line range*

Function: This command saves a program to diskette. The name of the program is specified by the file parameter (see `REPLACE`). You don't have to specify the `.BAS` extension.

Example:

```
SAVE MYPROG
```

This saves `MYPROG` to diskette.

SGN

Category: Numeric Function

Use: `SGN(X)`

Function: This returns a value that is dependent on the sign of the input value `X`. For `X>0`, the return value is `1`. For `X=0`, the return value is 0. If `X<0`, then the return value is -1.

SIN

Category: Numeric Function

Use: SIN(X)

Function: This function returns the trigonometric Sine value of the variable X.

SOUND

Category: Instruction

Use: SOUND *voice,vol,note,oct,len*

Function: Controls the sound generator chip. You can produce a tone of any volume, frequency, and length with this instruction. You have a choice of 3 *voice*s (1-3), to be played at the same time (polyphonic). The *vol*ume can be any value from between 0 and 15. The individual notes of a given octave are given in the *note* parameter. The *oct*ave of the tone must be between 1 and 8. The last parameter gives the *len*gth of the tone in 1/50th of a second intervals.

Example:

```
10 FOR OCTAVE=1 TO 8
20 FOR NOTE=1 TO 12
30 SOUND 1,15,NOTE,OCTAVE,1
40 NEXT NOTE,OCTAVE
50 SOUND 1,0,0,0,0
```

This is a test program that plays all 96 tones very quickly. After the loops are finished, line 50 silences voice 1.

SPACE$

Category: String Function

Use: `SPACE$(X)`

Function: Returns a string of X spaces. X must be between 0 and 255. `PRINT SPACE$(10)` prints out 10 spaces.

SPC

Category: Function

Use: `SPC(X)`

Function: This function allows you to jump over X characters on a given line before printing starts. X must be between 0 and 255.

SQR

Category: Numeric Function

Use: `SQR(X)`

Function: Returns the square root of the value in variable X.

STEP

Category: Command

Use: `STEP`

Function: After issuing the `STEP` command, the program stops after executing each line. The next line is executed after you hit the <RETURN> key. This continues until the program encounters the `CONT` command.

STOP

Category: Instruction

Use: `STOP`

Function: This instruction is used within a program to stop the execution for your specified reason. When `STOP` is executed, the computer will display the statement `Stop at Line X`, where `X` is the number of the line containing the `STOP` instruction. You can restart the program using the `CONT` command.

STR$

Category: String Function

Use: `STR$(X)`

Function: You can convert a numeric value in `X` to its character string representation. If `X` is positive or zero, the first character is a blank space. If the value is negative, then the first character is a hyphen (-).

STRING$

Category: Function

Use: `STRING$(X,B$)`

Function: This function returns the string in `B$` repeated `X` times. `X` must be between 0 and 255.

Example:

```
PRINT STRING$(5,"*")
```

Output: `*****`

SWAP

Category: Instruction

Use:

```
SWAP A,B
```

Function: The `SWAP` instruction switches the contents of the two input variables. The two variables must be of the same type.

Example:

```
10 X=10:Y=30
20 SWAP X,Y
30 PRINT X;Y
```

Output: `30 10`

SYSTAB

Category: Variable

Use:

```
X=PEEK(SYSTAB+offset)
```

Function: This variable is an array of system parameters and pointers. The following shows the organization of this table.

Contents	Offset	Function
READ	0	Graphic resolution (1=hi,2=mid,4=low)
READ/ WRITE	2	Type of character in Editor 0-normal 1-bold 2-light 4-italic 8-underlined 16-reverse video To get combinations of the above typestyles, simply add the numbers of the needed options.
READ	4	AES-HANDLE of the Edit window.

Contents	Offset	Function (cont'd)
READ	6	AES-HANDLE of the List window.
READ	8	AES-HANDLE of the Output window.
READ	10	AES-HANDLE of the Command window.
READ	12	EDIT flag (0=close,1=open)
READ	14	LIST flag (0=close,1=open)
READ	16	OUTPUT flag (0=close,1=open)
READ	18	COMMAND flag (0=close,1=open)
READ	20	Address to graphic memory (4-bytes)
READ	24	GEMFLAG (0=normal,1=out) Disabling the GEM activities can cause problems with file activities.

Example: `POKE SYSTAB+2,8`

Sets the characters to underlined in the edit window.

```
10 RESET
20 A#=PEEK(SYSTAB+20)
30 BSAVE DUMP.DAT,A#,32768
```

This saves the contents of the OUTPUT window in the graphic memory and saves it under the name DUMP.DAT.

SYSTEM

Category: Command

Use: SYSTEM

Function: SYSTEM is the same as the QUIT command. It closes all files and returns you back to the GEM desktop.

TAB

Category: Function

Use: `TAB(X)`

Function: This function places the cursor at any location on the ST screen. X must be in the range between -32768 and +32767. The locations begin at the upper left corner of the screen.

Example:

```
PRINT TAB(5);"Test"
```

Output:

```
     Test
```

TAN

Category: Numeric Function

Use: `TAN(X)`

Function: Returns the trigonometric function Tangent of value X.

TRACE

Category: Command

Use: TRACE *line#-line#*

Function: When you run your program, TRACE prints each line number on the screen as it is executed. You can specify a range of line numbers if you only want to check on a portion of your program.

Example:

```
TRACE 40
```

Line 40 is printed out each time it is executed.

TROFF

Category: Command

Use: `TROFF` *line#–line#*

Function: This disables the line number trace turned on by the `TRON` command.

TRON

Category: Command

Use: `TRON` *line#–line#*

Function: This command is very similar to the `TRACE` command, except that `TRON` only prints the number of the executed line, rather than printing out the actual line itself.

These commands allow you to see how your program is running, line by line. They can be quite helpful during program development.

UNBREAK

Category: Command

Use: `UNBREAK` *line#–line#*

Function: This command disables the `BREAK` command. The `UNBREAK` command also allows you to specify certain lines.

UNFOLLOW

Category: Command

Use: `UNFOLLOW X,Y`

Function: This command disables the `FOLLOW command`. If only the command is given, then the entire `FOLLOW` command is disabled. Listed variables let you stop following the altering of any variable you wish.

UNTRACE

Category: Command

Use: `UNTRACE` *line#-line#*

Function: This command undoes the `TRACE` command. If this command is given without line numbers, then `TRACE` is completely disabled. Otherwise, the `TRACE` is only stopped for the specified range of line numbers.

VAL

Category: Numeric Function

Use: `VAL(X$)`

Function: This function turns the character representation of a number into an actual number. If the function comes across a character which cannot be turned into a number (ex: `A`), then only the characters up to this character are converted. If the first character is not a number, minus sign or plus sign, then the returned value is 0.

VARPTR

Category: Function

Use: `X=VARPTR(`*var*`)`
`X=VARPTR(`*#file*`)`

Function: This function returns the location of the variable into `X`. The address is the first location of the variable. The second version of the function returns the first location of the input/output buffer of the file associated with *#file*.

Example: `X=VARPTR(ST$)`

Returns the location in memory where the string variable `ST$` begins.

VDISYS

Category: Function

Use: `VDISYS`

Function: `VDISYS` enables the GEM-VDI interface and calls a VDI function. Parameters to the VDI function are passed in the `contrl`, `intin`, and `ptsin` arrays (see Chapter 7).

WAIT

Category: Instruction

Use: `WAIT X,Y`

Function: This instruction halts program execution until the contents of X and the value `Y` are the same (bitwise).

WAVE

Category: Instruction

Use: `WAVE`*,enable,env,shape,per,del*

Function: This instruction controls the waveform of the tones created by `SOUND`.

enable mix register of the tone generator. A zero in bits 0 through 2 activates the voice (1-3), a 0 in the bits 3-5 of the voice. *t* can activate more voices than one.

env curve register. A 1 in the bits 0-2 activates the curve for voice (1-3). A curve can be used for more than one voice.

shape curve form

period frequency of the wave

delay how long to wait before activating the wave. (in 1/50th of a second intervals).

WEND

Category: Instruction

Use: `WEND`

Function: This instruction marks the end of a `WHILE ... WEND` loop. `WEND` is similar to `NEXT` in a `FOR/NEXT` loop.

Example:

```
10 WHILE A<10
20 A=A+1
30 PRINT a;
40 WEND
50 END
```

WHILE

Category: Instruction

Use: `WHILE` *expression*

Function: This is the instruction which starts off a `WHILE...WEND` loop. Commands inside of this loop are executed only if a specified condition is true. As soon as the condition is false, then execution jumps outside the loop.

Example:

```
10 WHILE X<100
20 X=INT(101*RND(1))
30 PRINT X;
40 WEND
50 END
```

Loop continues until the value `X` is greater than or equal to 100.

WIDTH

Category: Instruction

Use: `WIDTH` *width*
`WIDTH LPRINT` *width*

Function: The `WIDTH` instruction can change the width of the screen or the print width of the printer. Widths between 14 and 255 are allowed. If this instruction is used, the ST will automatically append a <RETURN> at the end of the new line width. No <RETURN> is given if the value is 255.

Example:

```
10 WIDTH 15
20 PRINT "*****************************"
```

This example outputs a line of 15 asterisks and a second line of 5 asterisks.

WRITE

Category: Instruction

Use: `WRITE` *output,output...*

Function: This instruction is very similar to a `PRINT` statement, except that any string variable is output with quotation marks on both sides of the result. Also, commas are output.

Example:

```
10 X$="TEST"
20 WRITE X$,"A","B"
```

Output: `"TEST","A","B"`

WRITE

Category: Instruction

Use: `WRITE` *#number,output,...*

Function: This instruction is similar to `PRINT #` and `WRITE`. However, the output from `WRITE #` goes to the file specified by the file number *#number*.

`WRITE #` and `INPUT #` are the easiest ways to output data to a sequential output file.

Example:

```
10 OPEN "O",#1,"TELEPHONE"
20 N$="MOORE "
30 V$="ROGER "
40 T$="00441-9302312"
50 WRITE #1,N$,V$,T$
60 CLOSE #1
100 OPEN "I",#1,"TELEPHONE"
110 INPUT #1,N$,V$,T$
120 PRINT V$;N$;T$
130 CLOSE #1
```

XOR

Category: Function (Logical operator)

Use:

```
X XOR Y
```

Function: The logical operator XOR states that two parameters X and Y are exclusive of each other. This means that if only one of the parameters is true, the statement is true. However, if neither or both parameters are true, the statement is false.

Example:

```
PRINT 0 XOR 0,0 XOR 1,1 XOR 0,1 XOR 1
```

Output:

```
0 1 1 0
```

Appendix B: Reserved BASIC Words

ABS
ALL
AND
AS
ASC
ATN
AUTO
BASE
BLOAD
BREAK
BSAVE
CALL
CDBL
CHAIN
CHR$
CINT
CIRCLE
CLEAR
CLEARW
CLOSE
CLOSEW
COLOR
COMMON
CONT
CONTRL
COS
CSNG
CVD
CVI
CVS
DATA
DEF
DEF FN
DEFDBL
DEFINT
DEFSEG
DEFSNG
DEFSTR
DELETE
DIM
DIR
DO
EDIT
ELLIPSE
ELSE
END
EOF
ERA
ERASE
ERL
ERR
ERROR
EQV
EXP
FIELD
FIELD#
FILL
FIX
FLOAT
FOLLOW
FOR
FRE
FOR
FRE
FULLW
GB
GEMSYS
GET
GET#
GO
GOSUB
GOTO
GOTOXY
HEX$
IF
IMP
INKEY$
INP
INPUT
INPUT#
INPUT$
INSTR
INT
INTIN
INTOUT
KILL
LEFT$
LEN
LET
LINE
LINEF
LIST
LLIST
LOAD
LOC
LOF
LOG
LOG10
LPOS
LPRINT
LSET
MERGE
MID$
MKD$
MKI$
MKS$
MOD
NAME
NEW
NEXT
NOT
OCT$
OLD
ON
OPEN
OPENW
OPTION
OR
OUT
PCIRCLE
PEEK
PELLIPSE
POKE

POS
PRINT
PRINT#
PTSIN
PTSOUT
PUT
QUIT
RANDOMIZE
READ
REM
RENUM
REPLACE
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SEG
SGN
SIN
SOUND
SPACE$
SPC
SQR
STEP
STOP
STR$
STRING$
SWAP
SYSDBG
SYSTAB
SYSTEM
TAB
TAN
THEN
TO
TRACE
TROFF
TRON
UNBREAK
UNFOLLOW
UNTRACE
USING
USR
USR0
USR1
USR2
USR3
USR4
USR5
USR6
USR7
USR8
USR9
VAL
VARPTR
VDISYS
WAIT
WAVE
WEND
WHILE
WIDTH
WRITE
WRITE#
XOR

Appendix C: Problem Solutions

Chapter One

Solutions to page 33

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

Chapter Two

Solutions to page 54

1. a) legal
   b) legal
   c) `AUTO` is illegal (Atari ST command)
   d) legal
   e) `IF` is illegal; see c)
   f) `GB` is illegal (system variable)
   g) `4NAME%` is illegal since it starts with a digit
   h) 255 is illegal; see g)
   i) legal

2.
```
10 INPUT A,B,C,D
20 PRINT A;B
30 PRINT C;D
40 END
```

3.
```
10 REM ENTER HEIGHT H AND
20 REM AND BASE B IN INCHES
30 INPUT"ENTER H,B";H,B
40 A=.5*B*H
50 PRINT"THE AREA IS"A"SQUARE INCHES"
60 END
```

4.
```
10 INPUT"ENTER HEIGHT IN CM";CM
20 REM CALCULATE IDEAL WEIGHT
30 IW=CM-100
40 REM CALCULATE 10 PERCENT
50 PR=IW/100*10
60 IW=IW-PR
70 PRINT"YOUR IDEAL WEIGHT IS";IW;"KG"
80 END
```

This problem could also be solved with a shorter program. Lines 30, 50, and 60 could be gathered into one line, as the following example shows:

```
30 IW=(CM-100)-(CM-100)/100*10
```

This line is harder to read since you can't tell right away what calculation is being performed. Some readers will no doubt point out that this style of programming helps to save memory space. That's right, of course—but as a programmer, you must make some compromise between readability and length of the program. If you don't have to worry about memory space, you should write your program so that it can be understood easily. This will also help you understand your own programs later if you have to make changes.

5.
```
10 INPUT"HEIGHT, LENGTH, DEPTH IN CM";H,L,D
20 REM CALCULATE VOLUME
30 V=H*L*D
40 REM CALCULATE LITERS
50 V=V/1000
60 PRINT"AQUARIUM CONTAINS";V;"LITERS"
70 END
```

In this program the variables for the height, length, and depth (H, L, and D) are first assigned values in cm. Then in line 30 the volume is calculated in ccm. In line 50 the calculated volume is divided by 1000 and we have the volume of our aquarium 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
```

If these problems gave you any trouble, read through the appropriate sections again. Then you should be able to understand and solve them.

Solutions to page 64

1.
```
10 REM CLEAR THE SCREEN
20 CLEARW 2
30 REM CREATE RANDOM NUMBERS
40 R1=INT(6*RND(1))+1
50 R2=INT(6*RND(1))+1
60 REM OUTPUT RESULT
70 PRINT"ROLL 1:";R1,"ROLL 2:";R2
80 END
```

Your program should look something like this. Obviously, the solutions we present here are only *suggested* solutions. There are many ways of solving any problem. If you repeat the program with `RUN` / <RETURN> you will see the difference in the numbers printed. In line 20 the output window is cleared with the command `CLEARW 2`.

Lines 40 and 50 assign newly-created random numbers to the variables R1 and R2. If you had problems with the upper and lower bounds, read the section on random numbers again.

2.
```
10 REM ENTER VALUES OF TRIANGLE SIDES
20 INPUT"ENTER A,B,C IN CM";A,B,C
30 REM CALCULATION OF S
40 S=.5*(A+B+C)
50 REM CALCULATE SURFACE AREA
60 A=SQR(S*(S-A)*(S-B)*(S-C))
70 REM OUTPUT AREA
80 PRINT"THE AREA OF THIS TRIANGLE ";
90 PRINT"IS";F;"SQUARE INCHES"
100 END
```

With this program you must note that S must be calculated first because it is used in the calculation of the area. The conversion of the formula into BASIC should not have presented any difficulties. Nevertheless, be sure that you don't lapse into using the mathematical notation of the formulas. This happens all too easily.

3.
```
10 INPUT"TYPE A KEY FOLLOWED BY RETURN";A$
20 A=ASC(A$)
30 PRINT"THE ASCII VALUE OF ";A$;"=";A
40 END
```

If you have already tried this program out, you may have tried to enter a comma or just a <RETURN> to find out the ASCII value of one of these "characters." But the computer printed an error message. This is one of the disadvantages of the INPUT command, since it uses the comma to separate variables, for instance. If you press just <RETURN>, *nothing* is assigned to the string variable—that is, this variable is empty. Since the computer naturally cannot determine the ASCII value of nothing, an error message is printed. A way of getting around this problem is explained in a later section (see INP).

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 DISTANCE OF";S;"METERS"
60 END
```

Line 10 is interesting here. The variable G is assigned the value 9.81 at the start of the program. This procedure is called variable initialization. This means that you assigned certain values to various variables at the start of the program. Its advantage is that only the variable has to be called up in the program and not the entire number, which can be quite long under certain circumstances. This can, in turn, save memory space in large programs with more variables.

5.
```
10 INPUT"HOW MANY GALLONS USED";G
20 INPUT"HOW MANY MILES TRAVELLED";MI
30 V=G/MI*100
40 PRINT"CONSUMPTION PER 100 MI IS"
   ;V;"GALLONS"
50 END
```

This program is self-explanatory.

If you solved all of these problems to your own satisfaction, you can now go on to the next section. If you are uncertain about anything, go through the appropriate passages again.

Chapter Three

Solutions to page 74

1. a) is correct

2. You get the expression B$ = "DRIPS"

3. You get the expression ROTOR back again.

4.
```
B$=MID$(A$,4,1)+MID$(A$,8,1)+MID$(A$,6,1)+
MID$(A$,10,1)
```

This is one possible solution.

Solutions to page 90

1.
```
10 REM ENTER ANNUAL INCOME
20 INPUT"ANNUAL INCOME IN $";IC
30 IF IC > 50000 THEN 70
40 REM CALCULATE 33 PERCENT
50 TX=IC*33/100
60 GOTO 90
70 REM CALCULATE 51 PERCENT
80 TX=IC*51/100
90 PRINT"TAX TO BE PAID:";
100 PRINT " $";TX
110 END
```

In line 20 the annual income is read. The value entered is assigned to the variable IC. In line 30 the income is checked to see if it is over $50,000. If this is not the case, 33 percent of the income is calculated and printed. If the income is greater than $50,000, 51 percent is calculated in line 80 and displayed.

2. This problem could be solved in at least two ways. First the solution which uses the command IF...THEN.

```
10 REM SUM 1 TO 100
20 A=A+1
30 S=S+A
40 IF A < 100 THEN 20
50 PRINT"SUM OF 1 TO 100 =";S
60 END
```

In line 20 we have our counter for the individual summands from 1 to 100. Line 30 calculates the sum of the values of A so far, 1+2+3+4 and so on. Line 40 performs the comparison and the sum S of the individual summands from 1 to 100 is finally printed in line 50.

The second solution results from the fact that we are dealing with an *arithmetic* series here—that is, the difference between successive terms is a constant. The sum can be calculated from the formula,

```
Sn=n/2(A1+An)
```

n is the number of terms in the series, A1 the first term, and An is the last term. According to this, the second solution offers a solution in general. The program could look something like this:

```
10 INPUT"NUMBER OF TERMS";N
20 INPUT"FIRST TERM";A1
30 INPUT"LAST TERM";AN
40 REM CALCULATION
50 SN=N/2*(A1+AN)
60 REM OUTPUT
70 PRINT"THE SUM IS";SN
80 END
```

3.

```
10 REM 6 OUT OF 49
20 Z=Z+1
30 L=INT(49*RND(1))+1
40 IF Z > 6 THEN END
50 PRINT L;
60 GOTO 20
```

In this program the END command is not placed at the end of the program. There is no need to place the END command in the last line of the program. The program should otherwise be easily understood.

4. For this problem you must note that only the last values of A and Z are printed. The PRINT command stands outside the actual loop.

Your solution therefore must be:

```
52        9
```

If you got an 8 as the second value, remember that the program jumps to line 20 as long as Z is less than 9. Not until Z equals 9 is the condition no longer fulfilled and the output in line 40 is performed.

5.
```
10 REM ENTER STRING AND SUBSTRING
20 INPUT"ENTER STRING";A$
30 INPUT"ENTER SUBSTRING";B$
40 I=I+1
50 C$=MID$(A$,I,LEN(B$))
60 IF C$=B$ THEN PRINT"FOUND":END
70 IF I > LEN(A$) THEN PRINT"NOT FOUND":END
80 GOTO 40
```

This problem was rather difficult. Your program need not match the one above to the last detail. But it should contain something similar to the formation of the comparison string in line 50, since this is the real problem. The statement of the problem makes reference to an arbitrary string—that is, independent of which and how many characters are searched for. The MID$ function must be informed as to the length of the string to be found via the LEN function. The counter in line 40 takes care of always moving the position of C$ one place to the right in A$. The comparison to see if the string to be found (B$) matches the current string in C$ takes place in line 60. Line 70 asks if the entire length of A$ has already been searched and B$ was not found. The following example will help clarify the function of the program:

String A$ = "INFORMATION" is to be searched for the string B$ = "FORMAT".

Number of characters in B$=6, so the following substrings are generated:

```
1. INFORM
2. NFORMA
3. FORMAT
```

String 3 is the string we are looking for.

That was a tough nut to crack. Make sure that you have understood all the details of this program. If you are still unsure, work through the program step-by-step once.

Chapter Four

Solutions to page 115

1.
```
10 REM HARMONIC SERIES
20 CLEARW 2
30 PRINT"ADD UP TO WHAT SUM?"
40 PRINT
50 PRINT
60 INPUT S
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"AFTER"Z"TERMS, THE SUM IS"SH
```

In lines 20 to 60 the screen is cleared and then the user is requested to enter the sum to be generated. Line 70 sets the counter to 1 and in line 80 the sum is formed from the individual terms. After this the counter is incremented by one in line 90. Line 100 checks to see if the counter has reached 50 or a multiple of 50. An appropriate output is to be made after every 50 terms. Other multiples can also be tested with this technique. The value 50 need only be replaced by the number to be tested for. If the counter is a multiple of 50, the command after the `THEN` is executed. Line 110 checks if the entered sum has already been reached. Line 120 outputs the passes required after the sum is reached, as well as the sum itself.

2.
```
10 REM QUADRATIC EQUATION
20 CLEARW 2
30 PRINT"ENTER 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 FOR X1 =";X1
120 PRINT"SOLUTION FOR X2 =";X2
130 GOTO 150
140 PRINT"NO REAL SOLUTIONS!"
150 END
```

The conversion of the problem into a program should not have presented any difficulties. Note the case in which A is zero. According to the formula, a division must be made by 2*A. Since division by zero is not allowed, we must exclude this case from the beginning.

3. First the screen is cleared and then comes the output ILLEGAL VALUE. The important thing here is that you must recognize that the command directly following the GOTO command was executed.

Solutions to page 140

1.
```
10 REM READ NAMES
20 DIM Y$(6)
30 FOR I=1 TO 6
40 INPUT"NAME";Y$(I)
50 NEXT I
60 REM 1ST ALPHABETICAL NAME
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 part of the program should be pretty straightforward, since it was presented previously in some examples. Since you know that a total of 6 names are to be read, a FOR...NEXT loop can be used.

The second part of the program was, admittedly, somewhat trickier. If you solved this problem yourself, you may now pat yourself on the back. We talked about the temporary storage of values in an earlier section. It is precisely this technique which you must use again here. Which string variable you used for this is actually not so important. The array element Y$(0) is ideal for this purpose since it has no other use in the program. So in line 70 the contents of element Y$(1) are stored in Y$(0). In line 80 the FOR...NEXT loop begins with the start value 2. We can skip the value 1 since we don't need to compare the first element with itself. In line 90 the individual names are compared with each other in order. If the name in Y$(0) is "less than" the one currently in Y$(1), a branch is made to line 110 and the loop variable is incremented by 1. If the string in Y$(0) is "greater than" that in Y$(I), then Y$(0) is assigned the name in Y$(I). Once the loop variable has reached the value 6, the desired name is in Y$(0). It is then printed in line 120.

A word about comparing strings: If two strings are compared for greater or less than, each letter of the two strings are compared with each other. The deciding factor is the ASCII values of the individual characters. The string WIND is less than the string WINS because the ASCII value of D=68, and S=83.

2.
```
10 REM READ NUMBERS
20 DIM X(6)
30 FOR I=1 TO 6
40 X(I)=INT(100*RND(1))+50
50 NEXT I
60 REM FIND LARGEST 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 the program from problem 1. If you got the solution to problem 1, you then have the solution to problem 2. The difference is only in the type of array (numerical) and the random number generation in line 40. The comparisons for finding the largest number are based on the same principle as in problem 1. In line 90 we test only for greater than/equal to, since we want to find the largest number.

In problem 3, where you had to find the assignment rule for the array:

3.
```
10 REM ASSIGNMENT RULE FOR SEQUENCE
20 DIM X(6)
30 FOR I=1 TO 6
40 X(I)=I*I-I
50 NEXT I
60 REM OUTPUT ARRAY
70 FOR I=1 TO 6
80 PRINT X(I)
90 NEXT I
100 END
```

The values in this problem are created by the multiplying the loop variable `I` with itself and then subtracting `I` from that total. This solution is intended only to be a suggestion. If you got the same results in a different manner, naturally your solution is right as well.

Appendix D: BASIC Error List

This book was originally written using BASIC Version 7/18/85, with a total size of 138944 bytes. This version contains a number of errors. You may want to check these errors against later versions you might have.

Arrays

Before putting arrays into a program, the entire set of arrays must be first cleared with 0 or a null string, or else a number will be received (`ERASE` command).

Accuracy

The accuracy of the present version is inconsistent. For example, division such as 53/100 results in 0.529999 rather than 0.53. Extremely simple commands like `PRINT 8.4` won't execute accurately.

Variable Declaration

This is a strange one: Declaring values between 77312 (e.g. x=77312) and 77823 results in the `Function not yet done` error message. Some values (e.g. x=77500) lock up the entire system!

GOTOXY

This divides the column position by two (e.g. column 10=column 5). This is only the case in high-res mode.

INP

This command can only be interrupted by pressing <CONTROL>C and moving the mouse simultaneously.

LINE INPUT#

This command doesn't recognize the `EOF(n)` function.

ON ERROR GOTO

When this command is used, an EOF (end-of-file) results in a system crash.

OPTION BASE

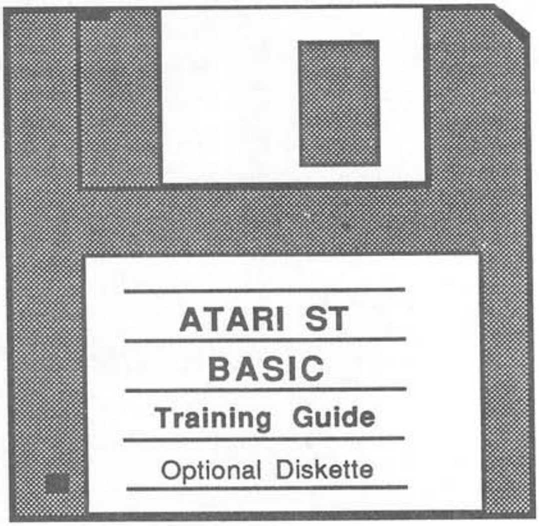
This command doesn't work at all.

VAL

The VAL function gives a nullstring with the last VAL value. This would be right if the output were a zero.

INDEX

Optional Diskette



For your convenience, the program listings contained in this book are available on an SF354 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
5370 52nd, Street SE
Grand Rapids, MI 49508

Or for fast service, call **1- 616 / 698-0330.**

AssemPro

Machine language development system for the Atari ST

"...I wish I had (AssemPro) a year and a half ago... it could have saved me hours and hours and hours."

—Kurt Madden
ST World

"The whole system is well designed and makes the rapid development of 68000 assembler programs very easy."

—Jeff Lewis
Input

AssemPro is a complete machine language development package for the Atari ST. It offers the user a single, comprehensive package for writing high speed ST programs in machine language, all at a very reasonable price.

AssemPro is completely GEM-based—this makes it easy to use. The powerful integrated editor is a breeze to use and even has helpful search, replace, block, upper/lower case conversion functions and user definable function keys. **AssemPro**'s extensive help menus summarizes hundreds of pages of reference material.

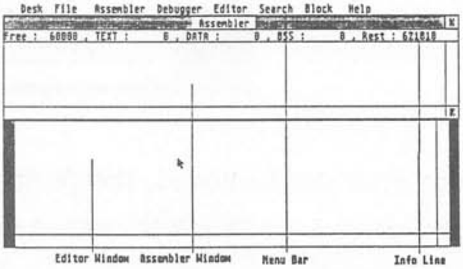
The fast macro assembler <u>assembles object code to either disk or memory.</u> If it finds an error, it lets you correct it (if possible) and continue. This feature alone can save the programmer countless hours of debugging.

The debugger is a pleasure to work with. It features single-step, breakpoint, disassembly, reassembly and 68020 emulation. It lets users thoroughly and conveniently test their programs immediately after assembly.

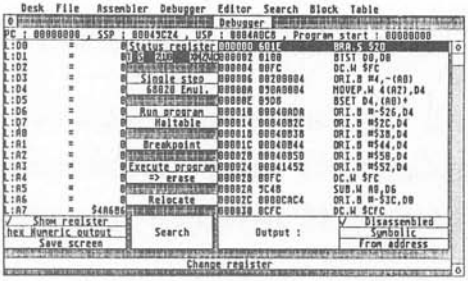
AssemPro Features:

- Full screen editor with dozens of powerful features
- Fast 68000 macro assembler assembles to disk or memory
- Powerful debugger with single-step, breakpoint, 68020 emulator, more
- Helpful tools such as disassembler and reassembler
- Includes comprehensive 175-page manual

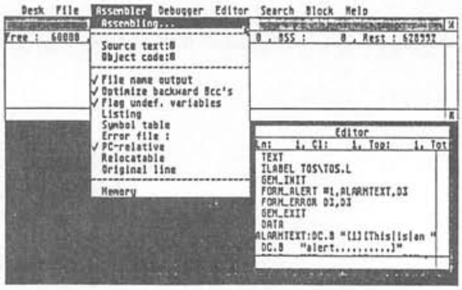
AssemPro Suggested retail price: **$59.95**



```
Free :  60000 , TEXT :       0 , DATA :       0 , BSS :       0 , Rest : 621818
```



```
PC : 00000000 , SSP : 00043C24 , USP : 0004A0C8 , Program start : 00000000
L:D0   =      0  000000 601E       BRA.S $20
L:D1   =      0  000002 0100       BTST D0,D0
L:D2   =      0  000004 00FC       DC.W $FC
L:D3   =      0  000006 00200004   ORI.B #4,-(A0)
L:D4   =      0  00000A 030A0004   MOVEP.W 4(A2),D4
L:D5   =      0  00000E 09D0       BSET D4,(A0)+
L:D6   =      0  000010 00040ADA   ORI.B #-$26,D4
L:D7   =      0  000014 0004082C   ORI.B #$2C,D4
L:A0   =      0  000018 00040838   ORI.B #$38,D4
L:A1   =      0  00001C 00040844   ORI.B #$44,D4
L:A2   =      0  000020 00040850   ORI.B #$50,D4
L:A3   =      0  000024 00041452   ORI.B #$52,D4
L:A4   =      0  000028 00FC       DC.W $FC
L:A5   =      0  00002A 3C48       SUB.W A0,D6
L:A6   =      0  00002C 0000CAC4   ORI.B #-$3C,D0
L:A7   = $4A686  000030 0CFC       DC.W $CFC
```



```
Free :  60000 ,
0 , BSS :       0 , Rest : 628992
Ln:    1, Cl:    1, Top:    1, Tot
 TEXT
 ILABEL TOS\TOS.L
 GEM_INIT
 FORM_ALERT #1,ALARMTEXT,D3
 FORM_ERROR D3,D3
 GEM_EXIT
 DATA
ALARMTEXT:DC.B "[1][This|is|an "
 DC.B   "alert..........]"
```

Selected Abacus Products for the ATARI® ST™

Chartpak ST

Professional-quality charts and graphs on the Atari ST

In the past few years, Roy Wainwright has earned a deserved reputation as a topnotch software author. **Chartpak ST** may well be his best work yet. **Chartpak ST** combines the features of his **Chartpak** programs for Commodore computers with the efficiency and power of GEM on the Atari ST.

Chartpak ST is a versatile package for the ST that lets the user make professional quality charts and graphs fast. Since it takes advantage of the ST's GEM functions, **Chartpak ST** combines speed and ease of use that was unimaginable til now.

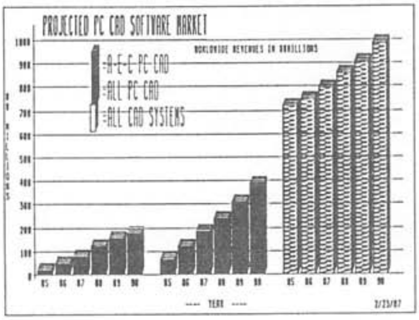
The user first inputs, saves and recalls his data using **Chartpak ST**'s menus, then defines the data positioning, scaling and labels. **Chartpak ST** also has routines for standard deviation, least squares and averaging if they are needed. Then, with a single command, your chart is drawn instantly in any of 8 different formats—and the user can change the format or resize it immediately to draw a different type of chart.

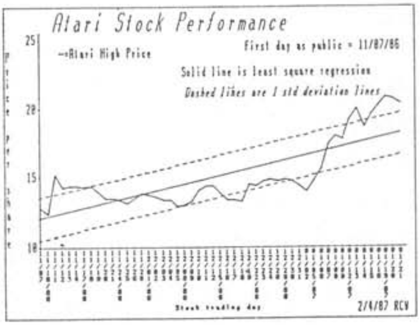
In addition to direct data input, **Chartpak ST** interfaces with ST spreadsheet programs spreadsheet programs (such as **PowerLedger ST**). Artwork can be imported from **PaintPro ST** or DEGAS. Hardcopy of the finshed graphic can be sent most dot-matrix printers. The results on both screen and paper are documents of truly professional quality.

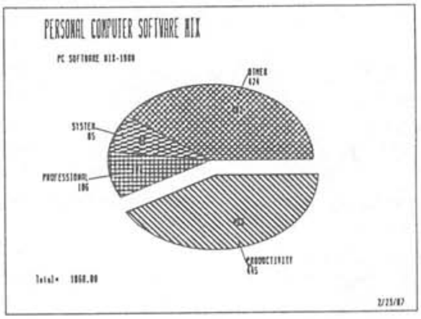
Your customers will be amazed by the versatile, powerful graphing and charting capabilities of **Chartpak ST** .

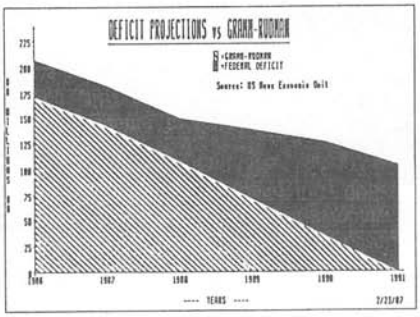
Chartpak ST works with Atari ST systems with one or more single- or double-sided disk drives. Works with either monochrome or color ST monitors. PWorks with most popular dot-matrix printers (optional).

Chartpak ST Suggested Retail Price: **$49.95**









Selected Abacus Products for the ATARI® ST™

DataRetrieve

(formerly FilePro ST)

Database management package for the Atari ST

"DataRetrieve is the most versatile, and yet simple, data base manager available for the Atari 520ST/1040ST on the market to date."

—Bruce Mittleman
Atari Journal

DataRetrieve is one of Abacus' best-selling software packages for the Atari ST computers—it's received highest ratings from many leading computer magazines. **DataRetrieve** is perfect for your customers who need a powerful, yet easy to use database system at a moderate price of $49.95.

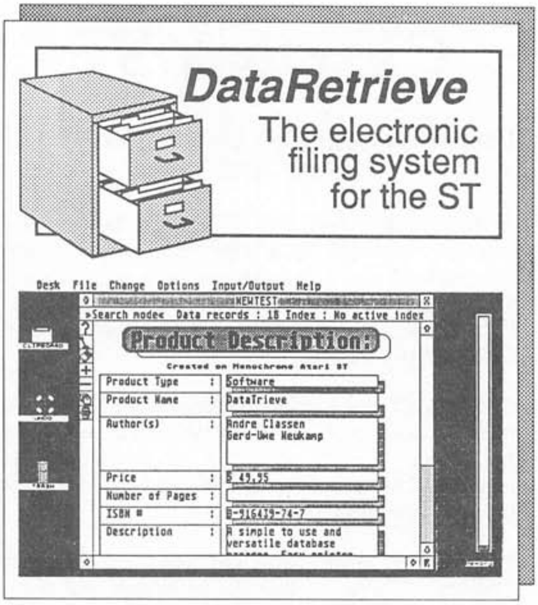
DataRetrieve's drop-down menus let the user quickly and easily define a file and enter information through screen templates. But even though it's easy to use, **DataRetrieve** is also powerful. **DataRetrieve** has fast search and sorting capabilities, a capacity of up to 64,000 records, and allows numeric values with up to 15 significant digits. **DataRetrieve** lets the user access data from up to four files simultaneously, indexes up to 20 different fields per file, supports multiple files, and has an integral editor for complete reporting capabilities.

DataRetrieve's screen templates are paintable for enhanced appearance on the screen and when printed, and data items may be displayed in multiple type styles and font sizes.

The package includes six predefined databases for mailing list, record/video albums, stamp and coin collection, recipes, home inventory and auto maintenance that users can customize to their own requirements. The templates may be printed on Rolodex cards, as well as 3 x 5 and 4 x 5 index cards. **DataRetrieve**'s built-in RAM disks support lightning-fast operation on the 1040ST. **DataRetrieve** interfaces to **TextPro** files, features easy printer control, many help screens, and a complete manual.

DataRetrieve works with Atari ST systems with one or more single- or double-sided disk drives. Works with either monochrome or color monitors. Printer optional.

DataRetrieve Suggested Retail Price: **$49.95**


DataRetrieve
The electronic filing system for the ST
Desk File Change Options Input/Output Help
NEWTEST

>Search mode< Data records : 18 Index : No active index

Product Description:

Created on Monochrome Atari ST

Product Type :	Software
Product Name :	DataTrieve
Author(s) :	Andre Classen Gerd-Uwe Neukamp
Price :	$ 49,95
Number of Pages :	
ISBN # :	0-916439-74-7
Description :	A simple to use and versatile database

DataRetrieve Features:

- Easily define your files using drop-down menus
- Design screen mask size to 5000 by 5000 pixels
- Choose from six font sizes and six text styles
- Add circles, boxes and lines to screen masks
- Fast search and sort capabilities
- Handles records up to 64,000 characters in length
- Organize files with up to 20 indexes
- Access up to four files simultaneously
- Cut, past and copy data to other files
- Change file definitions and format
- Create subsets of files
- Interfaces with **TextPro** files
- Complete built-in reporting capabilities
- Change setup to support virtually any printer
- Add header, footer and page number to reports
- Define printer masks for all reporting needs
- Send output to screen, printer, disk or modem
- Includes and supports RAM disk for high-speed 1040ST operation
- Capacities: max. 2 billion characters per file
  max. 64,000 records per file
  max. 64,000 characters per record
  max. fields: limited only by record size
  max. 32,000 text characters per field
  max. 20 index fields per file
- Index precision: 3 to 20 characters
- Numeric precision: to 15 digits
- Numeric range $\pm 10^{-308}$ ti $\pm 10^{308}$

Selected Abacus Products for the ATARI® ST™

Forth/MT

Powerful Multi-tasking Language for the Atari ST

Forth is not only a programming language, but also an operating environment—the user can program, assemble and edit. Since Forth is fast, compact, flexible and efficient., it's particularly well-suited to the solution of real time problems. In use for more than fifteen years in industrial and scientific applications, Forth dramatically reduces program development time compared to programming in assembly language or other higher-level languages.

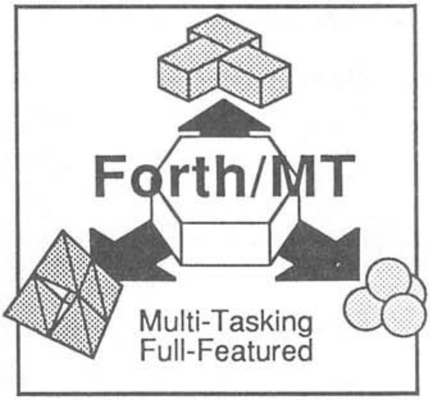
The powerful multi-tasking **Forth/MT** package was designed to make the fullest use of the ST's features for Forth programming.

Forth/MT features include:

- Over 750 words in the Kernal
- Complete TOS and LINE-A commands available
- Over 1500 words (disk accessible)
- Complete 32-bit implementation based on Forth-83 standard
- Machine language sections added for speed
- Many utilities: full screen editor, monitor, disk monitor and Forth macro assembler
- Utility descriptions stored on disk-you can change them to suit your needs
- Multitasking capability
- Machine language sections added for high-speed operation

Forth programmers will love the ease of use of this excellent package. **Forth/MT** the perfect tool for unleashing the power of the Forth programming language on the Atari ST line of computers.

Forth/MT Suggested retail price: **$49.95**



```
POINTER NEW-MOUSE <CR> (DEFINE BUFFER HEADER )
0 W, ( MASK COLOR ) 1 W, (MOUSE COLOR ) <CR>
BIN 0000000000000000 W, <CR>  (  1ST MASK LINE )
    0000000000000000 W, <CR>  (  2ND MASK LINE )
    0001111001111000 W, <CR>  (  3RD MASK LINE )
    0001111001111000 W, <CR>  (  4TH MASK LINE )
    0001001001001000 W, <CR>  (  5TH MASK LINE )
    0001001001001000 W, <CR>  (  6TH MASK LINE )
    0001001001001000 W, <CR>  (  7TH MASK LINE )
    0000001000001000 W, <CR>  (  8TH MASK LINE )
    0000000000000000 W, <CR>  (  9TH MASK LINE )
    0000101010100000 W, <CR>  ( 10TH MASK LINE )
    0000011111100000 W, <CR>  ( 11TH MASK LINE )
    0000001001000000 W, <CR>  ( 12TH MASK LINE )
    0000000000000000 W, <CR>  ( 13TH MASK LINE )
    0000000000000000 W, <CR>  ( 14TH MASK LINE )
    0000000000000000 W, <CR>  ( 15TH MASK LINE )
    0000000000000000 W, <CR>  ( 16TH MASK LINE )
    0000000000000000 W, <CR>  (  1ST MOUSE LINE )
    0001111001111000 W, <CR>  (  2ND MOUSE LINE )
    0010000110000100 W, <CR>  (  3RD MOUSE LINE )
    1010000110000101 W, <CR>  (  4TH MOUSE LINE )
    1110110110110111 W, <CR>  (  5TH MOUSE LINE )
    1110110110110111 W, <CR>  (  6TH MOUSE LINE )
    1110110110110111 W, <CR>  (  7TH MOUSE LINE )
    0111110111110110 W, <CR>  (  8TH MOUSE LINE )
    0111111111111110 W, <CR>  (  9TH MOUSE LINE )
    0011010101011100 W, <CR>  ( 10TH MOUSE LINE )
    0001100000011000 W, <CR>  ( 11TH MOUSE LINE )
    0001110110111000 W, <CR>  ( 12TH MOUSE LINE )
    0000111111110000 W, <CR>  ( 13TH MOUSE LINE )
    0000001111000000 W, <CR>  ( 14TH MOUSE LINE )
    0000001111000000 W, <CR>  ( 15TH MOUSE LINE )
    0000000000000000 W, <CR>  ( 16TH MOUSE LINE )
NEW-MOUSE TRANSFORM <CR>  ( SET NEW MOUSE )
SHOW <CR>                 ( AND DISPLAY )
```

Selected Abacus Products for the ATARI® ST™

PaintPro

Design and graphics software for the ST

PaintPro is a very friendly and very powerful package for drawing and design on the Atari ST computers that has many features other ST graphic programs don't have. Based on GEM™, **PaintPro** supports up to three active windows in all three resolutions—up to 640x400 or 640x800 (full page) on monochrome monitor, and 320 x 200 or 320 x 400 on a color monitor.

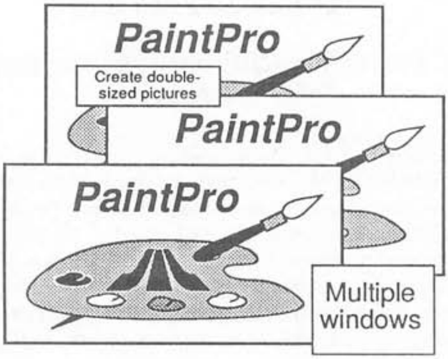
PaintPro's complete toolkit of functions includes text, fonts, brushes, spraypaint, pattern fills, boxes, circles and ellipses, copy, paste and zoom and others. Text can be typed in one of four directions—even upside down—and in one of six GEM fonts and eight sizes. **PaintPro** can even load pictures from "foreign" formats (ST LOGO, DEGAS, Neochrome and Doodle) for enhancement using **PaintPro**'s double-sized picture format. Hardcopy can be sent to most popular dot-matrix printers.

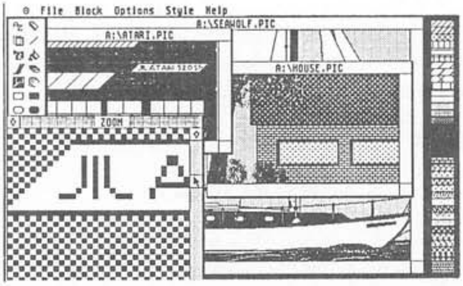
PaintPro Features :

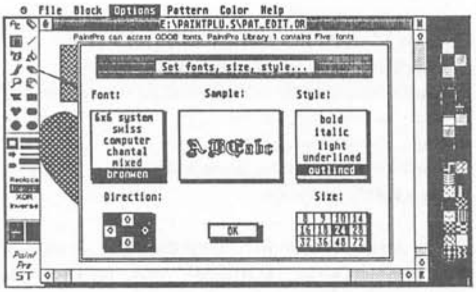
- Works in all 3 resolutions (mono, low and medium)
- Four character modes (replace, transparent, inverse XOR)
- Four line thicknesses and user-definable line pattern
- Uses all standard ST fill patterns and user definable fill patterns
- Max. three windows (dependng on available memory)
- Resolution to 640 x400 or 640x800 pixels (mono version only)
- Up to six GDOS type fonts, in 8-, 9-, 10-, 14-, 16-, 18-, 24- and 36-point sizes
- Text can be printed in four directions
- Handles other GDOS compatible fonts, such as those in **PaintPro Library # 1**
- Blocks can be cut and pasted; mirrored horizontally and vertically; marked, saved in LOGO format, and recalled in LOGO
- Accepts **ST LOGO, DEGAS, Doodle & Neochrome** graphics
- Features help menus, full-screen display, and UNDO using the right mouse button
- Most dot-matrix printers can be easily adapted

PaintPro works with Atari ST systems with one or more single- or double-sided disk drives. Works with either monochrome or color ST monitors. Printer optional.

PaintPro Suggested Retail Price: **$49.95**







File Block Options Pattern Color Help
E:\PAINTPLU.S\PAT_EDIT.DR
Replace
Transparent
XOR
Inverse
Paint Pro ST

PaintPro can access GDOS fonts. PaintPro Library 1 contains Five fonts

Set fonts, size, style...

Font:
6x6 system
swiss
computer
chantal
mixed
bronwen

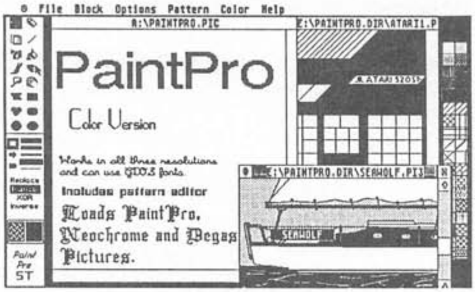
Sample:
ABCabc

Style:
bold
italic
light
underlined
outlined

Direction:

Size:
8 9 10 14
16 18 24 28
32 36 48 72

OK



PaintPro

Color Version

Works in all three resolutions
and can use GDOS fonts.

Includes pattern editor

Loads PaintPro,
Neochrome and Degas
Pictures.

Selected Abacus Products for the ATARI® ST™

PCBoard Designer

Interactive CAD Package for printed circuit board layout on the Atari ST

PCBoard Designer is an interactive, computer-aided design package for creating electronic printed circuit boards. It drastically reduces the cost, time and tedium of making one or two-sided pc boards. The advanced features of **PCBoard Designer** can improve a designer's productivity ten-fold.

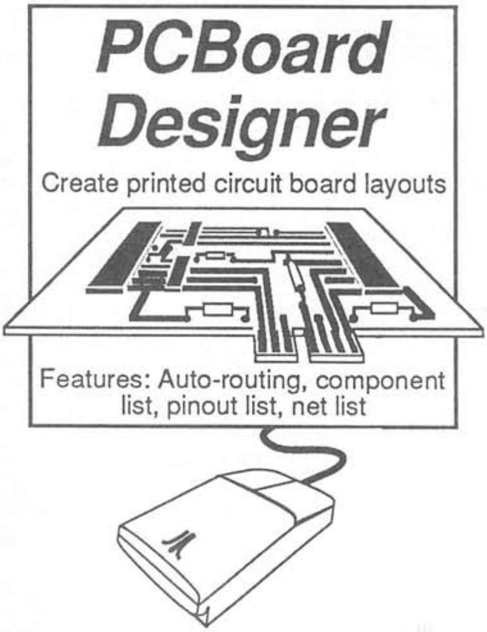
PCBoard Designer is easy to use. Design parameters are conveniently entered and modified at the computer. The user can position the components interactively by moving them on the screen using the mouse. This lets the user compare alternative component placement with no extra effort.

As the user position the components on the screen using the mouse, **PCBoard Designer** displays the new connections! Automatic routing is fast and precise.

The most powerful feature of **PCBoard Designer** is its fast automatic routing capability. Traces are automatically and precisely drawn on the screen. If the user changes the design, the traces can be immediately redrawn—this feature alone can save an enormous amount of time and money. In addition, the user has options of 45° or 90° angle traces, different trace widths, routing from pin to pin, pin to BUS, BUS to BUS, as well as two-sided boards. The rubberbanding feature lets you see the user-defined components during placement—and the user can reposition your components at any time during the design process.

PCBoard Designer prints the completed layout to any Epson/compatible dot matrix printer and Hewlett-Packard plotters at 2:1. The high-quality printout is camera-ready for final photo-etching. **PCBoard Designer** also prints the component layout, and lists every component and connection as well.

In conjuction with the Atari ST computer, **PCBoard Designer** is the most affordable PC board CAD package available. It boasts features that not available on systems costing thousands of dollars.

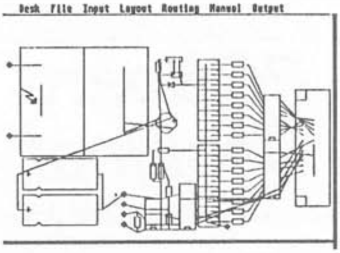

PCBoard Designer
Create printed circuit board layouts
Features: Auto-routing, component list, pinout list, net list

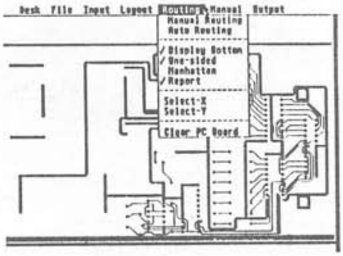
How PCBoard Designer works

There are basically four steps in creating a working pc board:

- **Specify the components:** For example, IC4 is an integrated circuit that fits in a 14-pin dual-in-line socket. You can also define custom component types, for example a 99-pin circular IC.
- **Specify the connections:** For example, pin 2 of integrated circuit IC4 is connected to lead 1 of transistor Q7. You can change the connections at any time.
- **Position the components:** Move the components to their desired position on the screen by using the Atari ST's mouse. You can reposition them at any time. **PCBoard Designer** automatically routes the connections when you're done.
- **Output the design:** The finished board can be printed on any Epson/compatible printer or Hewlett-Packard plotter. The printout is suitable for photoetching. You can also print the component layout (for silkscreening), the component list, and the list of connections.

Selected Abacus Products for the ATARI® ST™





"I was thoroughly impressed... a powerful, multi-featured design tool that can be easily learned and used."

—Bill Marquardt
Input magazine

"What makes this program especially easy to use is that the components are drawn to scale on the screen. This comes in handy when it's time for the user to position the components.

*"The author invested a lot of blood, sweat and tears writing this portion of the program. **PCBoard** Designer has a wide selection of options here that allow for flexible design. Either all of the connections or an individual connection can be routed at the click of the mouse button.*

"One thing is clear, though: author Florian Sachse has produced a first-class software package. This program will undoubtedly be a godsend to the engineer and electronic hobbyist alike.

—**DATA WELT** Magazine
APRIL 1986

Abacus Software, Inc.
5370 52nd St. S.E.
Grand Rapids, MI 49508

(616) 698-0330

PCBoard Designer (continued)

PCBoard Designer Features:

- PC boards may be one-sided or two-sided
- Components are drawn to scale on the screen
- Custom components may be used
- Component positioning is flexible and interactive
- Components may be roatated in 90º increments
- Traces are drawn using sophisticated and fast automatic routing techniques—the user has the ability to make 45º and 90º angle traces, variable trace widths, pin to pin, pin to bus and bus to bus routing
- "Blockades" may be inserted onto the board to handle special cases
- Printout is high quality and suitable for photo-reproduction
- Features are clearly displayed and are selectable from the drop-down menus

Hardware Requirements:

Computer: Atari 520ST or 1040ST computer and monochrome monitor with one or more single-sided, double-sided, or hard disk drives.

Printers/Plotters: PCBoard Designer prints your completed layout to any Epson or Epson-compatible dot matrix printer at 2:1. Epson FX-80, FX-100, Toshiba, NEC P6 and P7 or compatible printersrequired for photo-ready traces. Also works on Hewlett/Packard plotters.

Package: Includes 100 page manual in 3-ring slipcase binder and program diskette.

Free phone support to registered users.

PCBoard Designer can dramatically improve design productivity by eliminating many redundant steps and time-consuming alterations. With all of its advanced time-saving capabilities, **PCBoard Designer** pays for itself after the first successfully designed board.

PCBoard Designer

Suggested Retail Price:

$195.00

Selected Abacus Products for the ATARI® ST™

PowerLedger ST

(formerly PowerPlan ST)

Spreadsheet/Graphics package for the Atari ST

"A superior spreadsheet program for weekend bookeeping to the heavyweight job costing applications, (Powerledger ST) is a definite winner."

—Judi Lambert
ST World

Ever since VisiCalc and Lotus 1-2-3 stormed the personal computer market, the computer has become an important planning tool. **PowerLedger ST** brings the power of electronic spreadsheets to the Atari ST line of computers—it lets the user quickly perform hundreds of calculations and "what-if" analyses for business applications, and crunch raw data into meaningful, comprehensible information, to keep track of budgets, expenses and statistics.

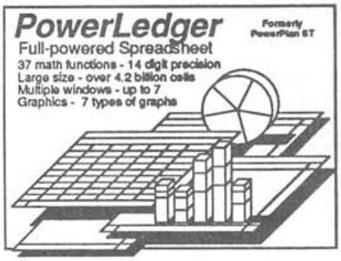
PowerLedger ST is a powerful analysis package that features a large spreadsheet (65,536 X 65,536 cells—over 4 billion data items). It also contains a built-in calculator, online notepad, and integrated graphics.

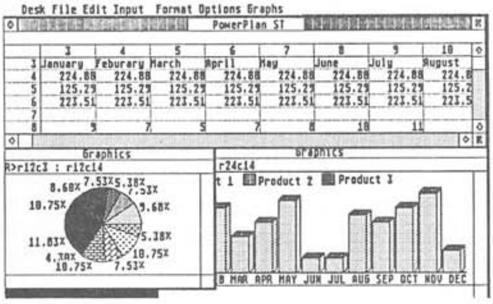
PowerLedger ST is also very easy to learn, since it uses the familiar GEM features built into the ST. And PowerLedger ST can use multiple windows—up to seven. Data from the spreadsheet can be graphically summarized in in pie charts, bar graphs and line charts, and displayed simultaneously with the spreadsheet. For example, one window can display part of the spreadsheet; a second window a different part; and a third window, a pie or bar chart of the data.

PowerLedger ST works hand-in-hand with our **DataTrieve** data management package and our **TextPro** wordprocessing package.

PowerLedger ST's extraordinary combination of data and graphic power, ease of use and low price makes it a perfect tool for every ST owner's financial planning needs.

PowerLedger ST works with Atari ST systems with one or more single- or double-sided disk drives. Works with either monochrome or color ST monitors. Works with most popular dot-matrix printers (optional).


PowerLedger
Formerly PowerPlan ST
Full-powered Spreadsheet
37 math functions - 14 digit precision
Large size - over 4.2 billion cells
Multiple windows - up to 7
Graphics - 7 types of graphs



	3	4	5	6	7	8	9	10
3	January	Feburary	March	April	May	June	July	August
4	224.88	224.88	224.88	224.88	224.88	224.88	224.88	224.8
5	125.29	125.29	125.29	125.29	125.29	125.29	125.29	125.2
6	223.51	223.51	223.51	223.51	223.51	223.51	223.51	223.5
7								
8	9	7	5	7	8	10	11	

PowerLedger ST Features:

- Familiar drop-down menus make PowerPlan easy to learn and use
- Large capacity spreadsheet serves all the user's analysis needs
- Convenient built-in notepad documents your important memos
- Flexible online calculator gives you access to quick computations
- Powerful options such as cut, copy and paste operations speeds the user'swork
- Integrated graphics summarize hundreds of data items
- Draws pie, bar, 3D bar, line and area charts automatically (7 chart types)
- Multiple windows emphasize the user's analyses
- Accepts information from DataTrieve, our database management software
- Passes data to **TextPro** wordprocessing package
- Capacities: maximum of 65,535 rows
  maximum of 65,535 columns
  variable column width
  numeric precision of 14 digits
  maximum value 1.797693 x 10^{308}
  minimum value 2.2 x 10^{-308}
  37 built-in functions

PowerLedger ST Suggested Retail Price: **$79.95**

Selected Abacus Products for the ATARI® ST™

TextPro

Wordprocessing package for the Atari ST

"TextPro seems to be well thought out, easy, flexible anf fast. The program makes excellent use of the GEM interface and provides lots of small enhancements to make your work go more easily... if you have an ST and haven't moved up to a GEM word processor, pick up this one and become a text pro."

—John Kintz
ANTIC

"TextPro is the best wordprocessor available for the ST"
—Randy McSorley
Pacus Report

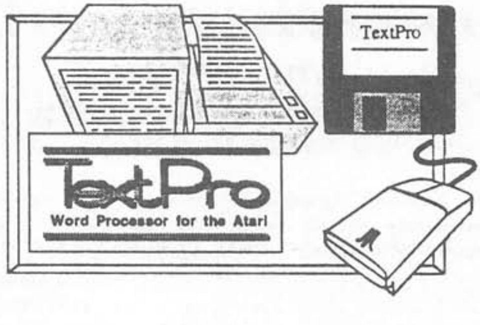
TextPro is a first-class word processor for the Atari ST that boasts dozens of features for the writer. It was designed by three writers to incorporate features that they wanted in a wordprocessor—the result is a superior package that suits the needs of all ST owners.

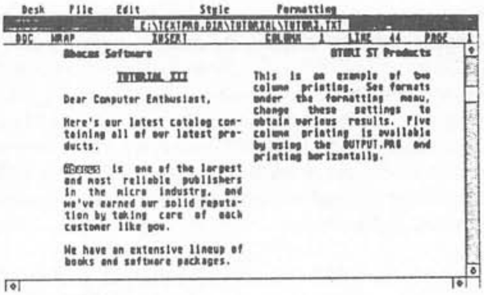
TextPro combines its "extra" features with easy operation, flexibility, and speed—but at a very reasonable price. The two-fingered typist will find **TextPro** to be a friendly, user-oriented program, with all the capabilities needed for fine writing and good-looking printouts. **Textpro** offers full-screen editing with mouse or keyboard shortcuts, as well as high-speed input, scrolling and editing. **TextPro** includes a number of easy to use formatting commands, fast and practical cursor positioning and multiple text styles.

Two of **TextPro**'s advanced features are automatic table of contents generation and index generation —capabilities usually found only on wordprocessing packages costing hundreds of dollars. **TextPro** can also print text horizontally (normal typewriter mode) or vertically (sideways). For that professional newsletter look, **TextPro** can print the text in columns—up to six columns per page in sideways mode.

The user can write form letters using the convenient Mail Merge option. **TextPro** also supports GEM-oriented fonts and type styles—text can be **bold**, underlined, *italic*, $^{\text{superscript}}$, outlined, etc., and in a number of point sizes. **TextPro** even has advanced features for the programmer for development with its Non-document and C-sourcecode modes.

TextPro Suggested Retail Price: **$49.95**




Desk File Edit Style Formatting
E:\TEXTPRO.DIR\TUTORIAL\TUTOR3.TXT
DOC WRAP INSERT COLUMN 1 LINE 44 PAGE 1

Abacus Software ATARI ST Products

TUTORIAL III

Dear Computer Enthusiast,

Here's our latest catalog containing all of our latest products.

Abacus is one of the largest and most reliable publishers in the micro industry, and we've earned our solid reputation by taking care of each customer like you.

We have an extensive lineup of books and software packages.

This is an example of two column printing. See formats under the formatting menu, change these settings to obtain various results. Five column printing is available by using the OUTPUT.PRG and printing horizontally.

TextPro ST Features:

- Full screen editing with either mouse or keyboard
- Automatic index generation
- Automatic table of contents generation
- Up to 30 user-defined function keys, max. 160 characters per key
- Lines up to 180 characters using horizontal scrolling
- Automatic hyphenation
- Automatic wordwrap
- Variable number of tab stops
- Multiple-column output (maximum 5 columns)
- Sideways printing on Epson FX and compatibles
- Performs mail merge and document chaining
- Flexible and adaptable printer driver
- Supports RS-232 file transfer (computer-to-computer transfer possible)
- Detailed 65+ page manual

TextPro works with Atari ST systems with one or more single- or double-sided disk drives. Works with either monochrome or color ST monitors.

TexPro allows for flexible printer configurations with most popular dot-matrix printers.

How to Order

5370 52nd Street SE Grand Rapids, MI 49508

All of our ST products—applications and language software, and our acclaimed 14 volume **Atari ST Reference Library**—are available at more than 2000 dealers in the U.S. and Canada. To find out the location of the Abacus dealer nearest to you, call:

(616) 698-0330
8:30 am-8:00 pm Eastern Standard Time

Or order from Abacus directly by phone with your credit card. We accept Mastercard, Visa and American Express.

Every one of our software packages is backed by the **Abacus 30-Day Guarantee**–if for any reason you're not satisified by the software purchased directly from us, simply return the prooduct for a full refund of the purchase price.

Order Blank

Send your completed order blank to:

Abacus Software
5370 52nd Street SE
Grand Rapids, MI 49508

Your order will be shipped within 24 hours of our receiving it

Name: ____________________

Address: ____________________

City ____________ State ________ Zip ______ Country ____________

Phone: ______ / ____________

Qty	Name of product	Price
	Mich. residents add 4% sales tax	
	Shipping/Handling charge **(Foreign Orders $12 per item)**	$4.00
	Check/Money order TOTAL enclosed	

Credit Card# □□□□□□□□□□□□□□□□□

Expiration date □□/□□   Cardholder Signature ____________________

For extra-fast 24-hour shipment service, order by phone with your credit card

ATARI® ST™

BASIC Training Guide

A functional, educational and well-written introduction to ST BASIC. From problem analysis, to algorithms, to BASIC commands, you'll learn programming quickly. Quizzes throughout the book help you "learn to think" in BASIC, while getting practical grounding in the language.

Topics include:

- Data-flow and program flowcharts
- Advanced programming techniques
- Menus
- Multi-dimensional arrays
- Sort routines
- File management
- BASIC under GEM™
- and much, much more

About the authors:
Frank Kampow is an expert in data processing, with many years of experience in programming and writing for Data Becker. Co-author Norbert Szczepanowski is a highly experienced data-processing specialist and also a bestselling book author (GEM Programmer's Reference, Anatomy of the 1541, others).

ISBN 0-916439-57-7

A Data Becker book published by

You Can Count On
Abacus Software

P.O. Box 7211 Grand Rapids, MI 49510 - Telex 709-101 - Phone 616/241-5510

ATARI
ST

ATARI ST BASIC Training Guide

Kampow
Szczepanowski

9

ABACUS