11 Variables and expressions

What is a variable?

A variable is a piece of memory which is given a name, like Fred or Number or X or Y or virtually anything you want, and this memory is set aside for storing information. It is rather like a box where you and the computer can put useful items of information until they are needed at a later stage. All the computer has to be told is what the box is called, and what kind of information it can expect to find inside. Not only that, but the contents of a box can be changed at any time; so the computer can go to the box to store information, retrieve it, use it, change it, then put it back inside again as many times as you instruct the computer to do so.

There are three types of ‘boxes’ or variables which the computer can use, and these are used to store three types of information. Briefly, these are:

- A ‘real’ variable, which can store numbers or fractions, eg 123.654.
- An ‘integer’ variable, which can store only whole numbers, eg 123.
- A ‘string’ variable, which can store ‘strings’ of characters such as words.

Each type is distinguished by the last character of the variable name. A name by itself, like BERT, signifies a real variable, BERT% an integer variable and BERT$ a string variable.

Real variables

Press **BREAK** and type the following program (the line numbers are show, but you will not need to type them if you are using AUTO).

```
>10 PRINT 3+2,3-2,3*2,3/2
>20 A=3
>30 B=2
>40 PRINT A+B,A-B,A*B,A/B
```

If you run this program you will see the numbers

```
5         1         6         1.5
5         1         6         1.5
```

are printed on the screen. The first row shows the results of the calculations performed by the **PRINT** instruction. The second row again shows the results, except that this was arrived at by lines 20 to 40 which use real variables.

Line 20 tells the computer that there is a variable in the program called A, and sets the current value to 3.

Line 30 tells the computer that there is another variable called B, and its current value is 2. Now that the computer is aware of these two variables, you can tell it to use them in calculations. Thus in line 40, the computer looks for the number stored in each variable, performs the necessary calculations, and the **PRINT** instruction prints the results on the screen just like it did for line 10.

Operators and expressions

Things like 3 + 2, A*B, (FRED – 4)*B are called expressions. In general, an expression is a sequence of numbers and variables together with mathematical symbols like +, *, /. These symbols, which are called the ‘arithmetic operators’, have their normal mathematical meaning, except that in BASIC, * is used for ‘multiply’ and / for ‘divide’.

Here is a list of the arithmetic symbols or ‘operators’ used in Electron BASIC:

+	Addition
–	subtraction
*	multiplication
/	division
ˆ	raise to the power
.	decimal point

For a description of operator precedence, see chapter 12.

Rules for variable names

The rules for variable names are:

- There must be no spaces in the name.
- The name must start with a letter
- There must be no punctuation marks in the name and no arithmetic operators. Underline characters may be used.
- The name must not begin with a BASIC keyword (such as **LIST** or **RUN**).

All the following names are acceptable:

```
X = 6.6

SMALL = -30

small = -60

Xy = 4*3

height6 = 5/11

William1 = 1066

space_rocket_speed = 25.000
```

Note that capital and small letters are regarded as different by Electron BASIC, so that **SMALL** and **small** are two different variables. Underlines take the place of spaces, which are not allowed.

The following are not acceptable:

`6teen = 16`	(begins with a number)
`TOTAL = 77`	(begins with TO)
`see-seaw = 16`	(contains a minus sign)
`LOW LINE = 3.3333`	(contains a space)
`How! = 1`	(contains punctuation mark)

A variable does not have to be specified in terms of numbers; it may be specified in terms of other variables, or a mixture of variables and numbers. A statement of the form ‘variable = expression’ is called an *assignment statement*: it assigns the value of the expression to the variable. For example:

```
X = Y

Monday = Tomorrow

AGE = HEIGHT - 100

TALL = TALL + 1
```

The last assignment of this group is very common. It has the effect of increasing the value of the variable TALL by 1. It is read as ‘Add 1 to the number contained in TALL, and store it in TALL again’.

Integer variables

The variables described so far in this chapter are called real variables. This means that they can represent both whole numbers (integers) and decimal fractions. There are variables called integer variables which can be used on the Electron, and these are used for storing only whole numbers. They are signified by the % symbol after the variable name. For example,

```
SCORE% = 20

Hour% = 3600

Z% = -747
```

A% to Z%

The 26 integer variables A% to Z% are called resident integer variables, because they are not cleared when the program is **RUN**, or when **NEW** or **BREAK** is used. This means that values can be passed from one program to another. They also have special uses when you come to look at Assembly Language programming (see chapter 29).

Real versus integer variables

The reasons for using integer variables are:

- They occupy slightly less memory than do real variables.
- They are absolutely accurate provided you do not let them get out of range. Real variables are only accurate to nine figures.
- They are much quicker for the computer to process and carry out arithmetic functions.

However:

- Decimal fractions can only be stored in real variables.
- Much larger and much smaller numbers can be stored in real variables. Real numbers can have values up to approximately 170,000,000,000,000,000,000,000,000,000,000,000,000 or 1.7 x 10^{38} (though they are only accurate to the first nine numbers or nine significant figures).

The range and accuracy of real and integer variables are shown in the following table:

	Integer	**Real**
Example	64	1.732
Typical variables	A%	A
Maximum size	2,147,483,647	1.7 xx 10^{38}
Accuracy	absolute	9 sig figs
Stored in	32 bits	40 bits

DIV and MOD

There are two special arithmetic operators which give integer results. These are called **DIV** and **MOD**.

DIV is an integer division function. It gives the whole number part of a division, for example 9 **DIV** 2 is 4, 10.5 **DIV** 3 is 3.

When decimal numbers are used, such as in the second example above, the computer truncates the number (meaning that it ignores the decimal part) before it carries out the division: 8.1 **DIV** 2.9 is 4.

MOD stands for modulo, and is used to give the remainder after an integer division. For example: 9 **MOD** 2 is 1, 17 **MOD** 7 is 3.

Once again, decimal numbers are truncated before the division takes place. For example: 16.1 MOD 3.8 is 1.

The TIME integer variable

There is also a special integer variable, resident in the computer, which is called TIME. TIME is an elapsed-time clock: it ticks away in hundredths of a second. Every 1/100 of a second its value is increased by 1, and it is used for timing programs.

```
10 T%=TIME
20 PRINT TIME -T%
```

will print the time taken to execute one line of program, in hundredths of a second.

TIME may be assigned a starting value, or it can be zeroed, just as any other variable:

```
TIME=0
```

TIME runs continually for as long as the computer remains switched on. You will understand better how to use it when you look at some of the programs later in the book.

String variables

You have seen that a variable is a name which can be assigned a value either directly or by an assignment statement. The computer will store this value in its memory as a binary number - a series of zeroes and ones. Characters are also stored in the computer as binary numbers, and each character has a code. This code is called ASCII, standing for ‘American Standard Code for Information Interchange’. If you look at Appendix F, you will see a table of ASCII codes showing all the letters, symbols, and numbers each with their corresponding ASCII code number.

When you use the **PRINT** instruction to put a message on the screen, as for example:

```
PRINT "ASCII"
```

the quotation marks each side of the message tell the computer that what is in between them is a string of characters and not a variable. So each of the characters in the message ‘ASCII’ is stored as a binary number, corresponding to 65, 83, 67, 73, 73 in decimal, as you can see from the ASCII chart in Appendix F.

There are special variables, called string variables, which hold characters as opposed to numbers. String variables are signified by a $ sign after the variable name. So we can say:

```
A$="ACORN"
fish$ = "TWO COD"
Birthday$ = "Monday 23rd August"
```

It is very important for you to understand how this last assignment is stored. Notice that the string contains a number, 23. Because of the quotation marks this number is not stored as 23 in binary, but as the ASCII code for 2 followed by the ASCII code for 3. This knowledge is very useful when you come to manipulate strings using their code values.

For example:

```
PRINT "23"
```

and

```
PRINT 23
```

both have the same effect.

But

```
PRINT "23*6"
```

and

```
PRINT 23 * 6
```

show the different ways in which numbers and strings are stored. As you can see from the ASCII table in Appendix F, every number has its own ASCII code.

You can use the computer to find out the ASCII code of a character.

```
PRINT ASC "Q"
```

will give the ASCII value of Q which is 81.

The opposite function is given by

```
PRINT CHR$ 81
```

which converts the ASCII code 81 into its corresponding character which is Q.

Even a space has an ASCII code.

```
PRINT ASC " "
```

gives 32.

And nothing at all (an empty string):

```
PRINT ASC ""
```

gives –1. This is not an ASCII value, but is conveniently different from all the others as to be easily distinguishable.

The instruction

```
PRINT CHR$ 81
```

has an equivalent which is easier to type:

```
VDU 81
```

is identical, so

```
VDU 81
```

gives the letter Q.

Commands operating on strings

LEN

String variables may be up to 255 characters long, and there is an instruction LEN, which gives the length of a string – the number of characters it contains.

```
PRINT LEN “ABCDEF”
```

will print 6 on the screen.

Similarly,

```
A$=“S O S”
PRINT LEN A$
```

will print 5 (because a space is a character).

Linking strings

Two or more strings may be linked together by using the ‘+’ operator, which apart from its arithmetic use, can simply link strings. The following program is an example of this.

```
10 AS = "I‘M"
20 B$ = "LEARNING"
30 C$ = "BASIC"
4B D£ = A$ + B$ + CS
50 PRINT D$
>RUN
I’MLEARNINGBASIC
```

LEFT$, RIGHT$, MID$

Not surprisingly, if the computer can link strings it can also disassemble a string to make smaller ones, using **LEFT$**, **RIGHT$**, and **MIDS**.

```
10 A$ = "INEQUITABLE”
20 B$ = LEFT$(A$,2)
30 CS = RIGHT$(A$,5)
40 D$ = MID$(A$,3,4)
50 PRINT B$
60 PRINT CS
70 PRINT D$
>RUN
IN
TABLE
EQUI
```

Notice how the three functions **LEFT$**, **RIGHT$**, and **MID$** are used:

LEFT$ (A$,2) copies the first two characters of string A$. In the program, these two characters are copied into B$.

RIGHTS (A$,5) copies the last five characters of string A$.

MID$ (A$,3,4) copies four characters from string A$, beginning at the third character from the left.

VAL, EVAL, STR$

There are three more string operating functions which convert to or from numbers: **VAL**, **EVAL**, and **STR$**.

```
10 X$ = "57/7 * SIN.6"
20 PRINT VAL X$
30 PRINT EVAL X$
```

When you run this program, **VAL** X$ gives the number with which the string X$ begins, in this case 57. If the string does not begin with a number then **VAL** returns the value 0.

EVAL X$ evaluates the string as if it were a numeric function, giving in this case 4.5978W3. EVAL will also evaluate variables in strings, provided these variables have been assigned earlier in the program.

Sometimes you need to turn a number into a string, and this is done by using the instruction **STR$**.

```
10 A=45 : B= 38
20 A$ = STR$ (A)
30 B$ = STR$ (B)
40 PRINT A + B
50 PRINT A$ + B$
```

INSTR

Another useful string function is **INSTR** (standing for IN STRING) which will compare two strings and tell you whether one of these strings is contained within the other, and at what position.

For example

```
10 A$ = "INEQUITABLE"
20 B$ = "I"
```

```
30 Z = INSTR (A$,B$,2)
40 PRINT Z
```

This program shows that **INSTR** returns the position at which B$ is the same as A$. We can start the **INSTR** comparison at any point along the string.

This program starts the comparison at the second character of string A$, and therefore indicates the second ‘I’ at position 6. If **INSTR** is used and there is no similarity between the strings, a 9 is given.

STRING$
The last string function, **STRING$**, is used when you want to make a long string which consists of repeated units. For example, if you wish to use a string to print a border made up from *–*–*–*– etc then it is easier to use the **STRING$** function than to type all the characters.

```
10 A$ = "*-"
20 B$ =STRING$(2@,A$)
30 PRINT B$
>RUN
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
```

The string B$ is made up from 20 copies of the string A$.

Comparison table of variables

Finally, here is the complete comparison table for integer, real, and string variables:

	Integer	**Real**	**String**
Example	810	1.141	“WORDS”
Typical variable	A%	A	A$
Maximum size	2,147,483,647	1.7×10^{38}	255 characters
Accuracy	absolute	9 sig figs	—
Stored in	32 bits	40 bits	ASCII values