TABLE OF CONTENTS

Chapter 1 - Introduction

If you want to learn programming, you've just made the right choice. BBC BASIC has a long and mature pedigree and is ideally suited if you are taking your first steps in making the computer do what you want, rather than what someone else thinks you want. Once you've mastered the fundamentals, the domain of this tutorial, BB4W will stay with you all the way. Unlike other dialects that seem to run out of steam after the basics, you can incorporate Windows controls and access the native Windows commands (known as the API) all with the same version.

There are two versions available: demo and full. The demo is restricted only in the size of the programs that you can write and its ability to create standalone executables. All the commands are available, giving you the chance to try absolutely everything. If and when you decide to purchase the full version, there's only one. No silver and gold versions or professional and enterprise - again one version does it all. The executables are small, fast and standalone without the baggage of other files that some require.

This tutorial assumes absolutely no prior knowledge of any programming language, but if you have dabbled before, it won't hurt. I must emphasize at the outset that its scope is purposely limited. It doesn't cover, for example, file handling or graphics. This is not because these subjects are difficult, but rather that the intention was to leave the reader with a thorough knowledge of the building blocks of all programs and know where to find pointers for the rest.

I am not going to tell you how to install BBC BASIC as this is straightforward enough. The only thing that is assumed is that you can open a simple text editor (e.g. Notepad), enter some text, use cut and paste, save it and re-open it later. If you can't do this, I suggest that you may not have had sufficient exposure to computers in general to benefit from learning to program at this moment. Go to the library, get yourself an introductory book and come back in a week.

Learning to program is an interactive experience. I have programmed in several languages over the years and bought several of those big thick books that cost £30+. They all have CDs in the back with all the examples ready to run. How does anyone

learn? I always ignore the CDs and type listings in by hand. That way you make mistakes. This is an advantage. By making and correcting mistakes, you learn far more than by just glancing over some source code and then running a precompiled example. It takes a little bit longer, but is far more worthwhile.

Everything in here was written and tested on the 8k demo version, but most of the examples are only 10 - 20 lines long anyway, so improve your touch typing, it'll be worth it. Similarly, a lot of the listings don't have example outputs. This again is to encourage you to type this stuff in and get it running on your own. The other thing is, play with the examples, improve them, prod them and find out what happens if ... Once you get the bug (no pun intended) you'll not be able to leave this alone. Have fun.

Acknowledgements

Thanks to my family who still can't understand what makes someone who programs for a living come home and do it for a hobby too.

Thanks must also go to Richard Russell for having the foresight to carry on developing and improving a language that first saw the light of day over twenty years ago and also for making suggestions and improvements to this document.

Any comments and questions please contact me at nextstep61@yahoo.co.uk

Chapter 2 - One Line Programs

One of the nice things about BBC BASIC over other languages like Visual BASIC or C is it is perfectly possible to write one line programs. Type a dozen characters into the editor, press Run and there's your working program. The program will behave like any Window, you can move it, minimize it, resize it - everything you would expect, all with just one line of code. That's pretty powerful and this power lets us experiment with some of the fundamental commands that BASIC provides immediately without three chapters of introductory pre-amble.

Start BBC BASIC. We'll now type some commands in to see how BASIC reacts. Our first command will get the computer to write to the output window. Type the following. Notice how PRINT is in upper case. All BBC BASIC commands and keywords must be typed in this way or BASIC will get upset and complain. It is possible to override this, but let's keep things simple and use the default settings.

```
PRINT 52
```

Followed by Enter or Return. Press the Run button from the toolbar, the one with the big black arrow, or press F9.

The computer responds by opening a new window which displays:

```
52
```

That's it: your first one line program, not worth sending your CV to Bill Gates just yet but it is a fully functional program. Close the window by clicking the little cross in the top right corner, just as you would any other window. You must always close the window before BASIC will allow you to alter the code in the editor. Place the cursor before the 52 and press delete a couple of times, now retype the line so it looks like this:

```
PRINT 50+2
```

Press Run and BASIC responds:

```
52
```

Now we're onto something. We can use this as a calculator. Close the window again. From here on we'll take the 'type - run - close window' cycle as read. Again, edit the line so it looks like below, don't delete PRINT, just alter the numbers:

```
PRINT 50+50-2
```

Response:

```
98
```

Try other mathematical expressions, using + (addition), – (subtraction), * (multiplication) and / (division).

```
PRINT (50*2)/4+8.1
```

The answer is:

```
33.1
```

So, PRINT will take whatever follows and write it to the output window. More importantly, if the values after PRINT form a mathematical expression, PRINT will calculate them and display the result. Try the next line, taking notice of the quotes:

```
PRINT "50+2"
```

Response:

```
50+2
```

If we enclose what we want in quotes, it will print exactly as written; hence, we can output text as well as numbers:

```
PRINT "Hello, world"
```

If in entering any of the above, BASIC responds:

```
Mistake
```

or

```
Missing "
```

or something similar, it means you have typed it incorrectly and need to re-enter it. Just in case you haven't seen this yet, go back into the editor and delete the final quote:

```
PRINT "Hello, world
```

PRINT is one of the most flexible commands in BASIC and you can do far more with it than output single statements. To begin with, it is possible to print more than one thing at once:

```
PRINT "50*2 = " ;50*2
```

Gives us:

```
50*2 = 100
```

Each separate item must be delimited by either a semicolon (;) or a comma (,). A comma adds more space.

```
PRINT "50*2 = " ,50*2
```

Gives:

```
50*2 =          100
```

PRINT on its own gives us a blank line. If we want several blank lines, BASIC allows us to put a single quote (') after the initial PRINT. Each quote will produce a blank line.

```
PRINT '''
```

or

```
PRINT "Hello" '' "World"
```

Each statement you have seen so far tells BASIC to print on the next line down. You can tell BASIC to print anywhere on the screen. PRINT will work in conjunction with certain other keywords one of which is TAB. TAB comes in two guises. With the first, it is supplied with one parameter (or argument as programmers like to call them). This will shift the output along a number of spaces:

```
PRINT TAB (10);"Hello"

          Hello
```

The brackets here are important. The first bracket must be immediately after the word TAB (no spaces) and the closing bracket must be after the number, although calculations are allowed:

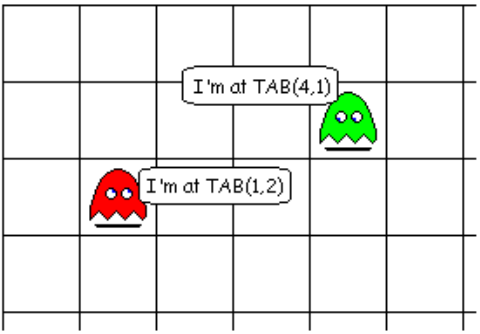
```
PRINT TAB (5*2);"Hello"
```

The closing bracket should always be followed by a semicolon.

The second variant of TAB takes two parameters and allows us to print anywhere on the screen.

```
PRINT TAB (10,20);"Hello"
```

The first parameter is the column (across the screen left to right), the second is the row (down the screen top to bottom); both start counting at zero. If you imagine a grid overlaying the window, each square in the grid can contain one character. The TAB statement tells BB4W which square to write the first character in. The others are added to the right of this. If the text you print exceeds the length of the line, the text wraps round to the start of the next line and if this occurs on the bottom line, the whole window will scroll up if necessary.



It is possible to get the program to run more than one statement on a line by separating each statement with a colon (:). This works a bit like a full stop in English and effectively tells BASIC, that command has now finished, here's the next.

```
PRINT TAB (10,20)"Hello" : PRINT TAB (10,21)"World"
```

There is a wonderful version of Tetris in the examples that come with BBC BASIC. It is all written in one line using this technique. As an intellectual exercise and a bit of fun, that's fine. Usually we want our programs to be more readable and so we need to expand into multi-line programs.

Chapter 3 - Using The Editor

The editor allows you to build up a sequence of commands by writing them on successive lines. To demonstrate this enter the following, one line at a time. Notice how when you press Enter, the editor formats the line for you and prints words it recognizes (keywords) in different colours.

```
REM My first program
PRINT "BBB   BBB    CCC"
PRINT "B  B  B  B  C   "
PRINT "BBB   BBB   C   "
PRINT "B  B  B  B  C   "
PRINT "BBB   BBB    CCC"
END
```

Select Run, the output window opens up and the message is displayed for all to see.

There are two new keywords here. REM and END. REM has been called the most important word in BASIC. Strange because it actually does nothing. It tells the computer to ignore anything that follows up to the end of the line. This is important as it lets us add text to our programs to remind us what the program does. REMs can be placed just about anywhere in the program and should be, because even the most obvious code in the world somehow manages to obscure itself when left alone for a couple of months. It's also nice for other readers as it explains what the code is intended to do. To enter a comment after a line of code, use the colon to separate the lines, just as described in the previous chapter. All comments are coloured green by default, so you can tell instantly if you are looking at active code or a comment.

The other word is END. This tells BASIC to stop executing the program. In short examples like this, it's not too important, but later on when structure is discussed, it becomes crucial. Get into the habit now of putting it in.

We will now play with the editor a little more, just to get a feel for it.

The larger a program gets, the more likely it is that we introduce mistakes or bugs.

As the editor very kindly colours words it knows, it is fairly easy to spot misspelt keywords; a common mistake is forgetting that all keywords must be in capitals. Other typing mistakes or runtime errors are harder to spot. BASIC will point these out for us though. Remove the final quote from one of the lines above and run the program. BASIC screeches to a halt and announces: *Missing "* when it reaches the offending line. To find the line on which the error occurred, go back to the editor and there it is, nicely highlighted. To edit the code, you need to close the output window or press the Stop button before the editor will let you back in again. Do this now, correct the mistake and then we can play some more.

If you've never run a program before, you might think that all this text appears in one swoop. Not so. BASIC executes the lines, one at a time in just the same order as you would read them, top to bottom. We can prove this by introducing another useful keyword. As computers do things very fast, there are times when it can be useful to slow them down a little. WAIT is an instruction that has no other function than to stop the program in its tracks for awhile. You have to give it a value to work with, which represents the amount of time to wait. Again, as computers move fast, the value is in centiseconds - hundredths of seconds or 0.01 seconds if you prefer. There are 100 of these in a second therefore 5.25 seconds would be 5.25 * 100 = 525. Let's edit our program so we can see it working. Place the cursor at the end of the first PRINT line and press Enter. A blank line is created. Now type WAIT 100 and press the down arrow, not Enter because we don't want another blank line. The program now looks like this:

```
REM My first program
PRINT "BBB   BBB    CCC"
WAIT 100
PRINT "B  B  B  B  C   "
PRINT "BBB   BBB   C   "
PRINT "B  B  B  B  C   "
PRINT "BBB   BBB    CCC"
END
```

Click the mouse at the start of the WAIT 100 line. Now hold down the shift key and press the down arrow on the keyboard. The whole line is highlighted. Right click the mouse and select Copy. Press the down arrow to move the cursor down a line. Right click and select Paste. Our text is copied into a new line, saving typing. Move the cursor down and repeat until the whole program looks like this:

```
REM My first program
PRINT "BBB   BBB    CCC"
WAIT 100
PRINT "B  B  B  B  C   "
WAIT 100
PRINT "BBB   BBB   C   "
WAIT 100
PRINT "B  B  B  B  C   "
WAIT 100
PRINT "BBB   BBB    CCC"
END
```

When the program is run, it appears one line at a time with a one second gap between each line. To make our program behave as it did before, we can delete the wait lines. Before we do that, highlight the first WAIT line and right click. Select *Add REMs* from the menu and the line is commented out. Repeat this for the other lines.

```
REM My first program
PRINT "BBB   BBB    CCC"
REM WAIT 100
PRINT "B  B  B  B  C   "
REM WAIT 100
PRINT "BBB   BBB   C   "
REM WAIT 100
PRINT "B  B  B  B  C   "
REM WAIT 100
PRINT "BBB   BBB    CCC"
END
```

As REM causes BASIC to ignore everything else on the line, the WAITs are now bypassed and the program runs as before. Using REMs like this is quite common when testing a program to block out lines of code. It is perfectly possible to REM lots of lines by selecting more than one at a time. You can remove the REMs in a similar manner by highlighting lines, right clicking and selecting *Remove REMs* .

To delete the lines completely, place the cursor at the start of the line, hold down shift and press the down arrow once. Press the delete key and the line disappears. The editor shuffles the remaining lines up to fill the gap. Delete the WAIT lines now, you'll need the original program for the exercise later on.

Programs grow in size, almost as you look at them. An important thing to bear in mind with programs of any size is readability. BBC BASIC for Windows allows us a couple of useful techniques to add space to our program and make it easier on the eye. Neither of these alters the way BASIC runs the code, but makes it easier for us poor humans to follow.

You can include blank lines in your programs. They are ignored by BASIC but do make things easier to read. Our little program could be easily written:

```
REM My first program

PRINT "BBB   BBB    CCC"
PRINT "B  B  B  B  C   "
PRINT "BBB   BBB   C   "
PRINT "B  B  B  B  C   "
PRINT "BBB   BBB    CCC"

END
```

It would run exactly the same, but just looks that bit clearer.

BBC BASIC allows up to 251 characters on each line, but when lines get this long, they can be a pain to read as it involves scrolling across the screen. It is possible to split a line using the backslash (\ ). To indicate that the line is incomplete, put the \ at the point you want to split the line. The next line must start with \ as the very first character to indicate this is the continuation. As BASIC ignores everything after the \ at the end of the line, you can insert comments there if you want.

Tip: Splitting long PRINT lines

You may remember from the first section that we can use a semicolon to print items next to each other. We can use the semicolon to split a line with lots to print on it like this:

```
REM Splitting long lines
PRINT "This is a very, very, very, very," ; \
\  " very, very, very, very, very, very," ; \
\  " very long line."
END
```

Exercises

1) Modify the first program to read "BB4W" instead of "BBC".
2) Insert TABs into the print statements to print BB4W starting at row 10, column 10 in the output window.

Chapter 4 - Storing Data in Variables

You can now use BBC BASIC to print to the screen and as a calculator. Wow, you might think, at least my calculator has a memory so I can store a result to use it again later. Well, BASIC has too. In fact it's got lots of memory areas you can use to store results, and it can store the information in different formats. This section introduces the concept of variables: memory locations within the computer that can be used by a program.

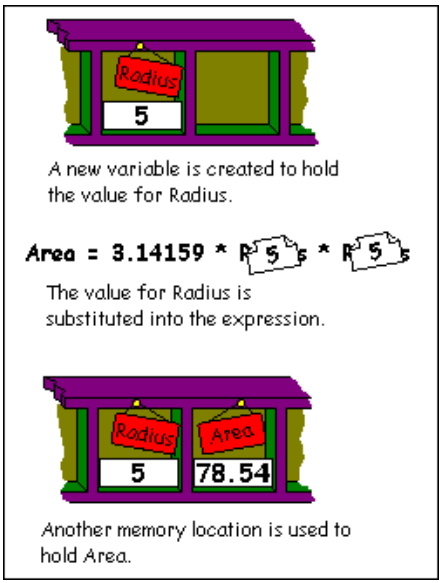
What we need is a little program to demonstrate this. If you have anything in the editor, get rid of it by pressing New. You will be prompted if you have not saved your previous masterpiece, so save it or discard it as you wish. Now type:

```
REM Area of a circle
Radius=5
Area=3.14159*Radius*Radius
PRINT "The area of your circle is " ;Area
END
```

Run it, just to see it works. Save it because we'll be using it a lot later. Anybody who ever went to school once should understand the principles at work here, it's a bit like algebra. If you change the second line for different values of *Radius* and re-run the program, it will give you a different result each time. What is happening? When the program runs, it gets to the second line and sees we are trying to assign a value of 5 to something called *Radius* . BASIC maintains a list of memory locations which it creates afresh every time a program is run. As this is the first active line, it won't have a memory location called *Radius*, so one is very nicely made for us. Having done this, the value of 5 is written into the new location.

On the next line, something similar happens with the value for *Area* . The assignment is an expression, however, so it needs to calculate this first. When BASIC gets to the section with *Radius* , it goes to its variable list and finds the value. This is 5 from the previous line, so 5 is substituted into the expression. Once complete, the result is assigned to the newly created variable *Area* . The fourth line has BASIC retrieve the value and print it out after dressing with some text for

clarity.



If you have opened any of the programs that come with BBC BASIC, you will have seen lots of lines like the above, so these things called variables must be quite important. We'll dwell on them for a while. Variables by default are floating point numbers, i.e. ones with a decimal place, but there are others. BBC BASIC supports four different types. These are defined by the last character of the variable's name:

Type	Description	Examples	Range	Character
Floating point	Decimals	3.142, 0.001, 1E10-2	–5.9E-39 to 3.4E38	# or none
Integer	Whole numbers	32000, 4, -66	–2147483648 to +2147483647	%
Byte	Whole numbers	0, 128, 255	0 to 255	&
String	Text	"BBC BASIC", "Hello"	any number of characters from 0 to 65535	$

It is possible to change the resolution of floating point numbers for greater accuracy,

see the help files for more details.

A variable name can start with any letter of the alphabet (upper or lower case), an underscore (_) or a grave accent (`). It can then have any combination of upper and lowercase letters, numbers or underscores. The length of the name is not limited, but usually names are kept shortish to save typing. Finally there is the character that gives the type, from the table above. If no character is given, floating point is assumed.

Names are case sensitive so *AREA* , *area* and *ArEa* are different as far as BBC BASIC is concerned (not necessarily so with other dialects, but once you've used this one, you won't need any other). Also BB4W is quite happy with variables of different types with the same name, so *Area%* , *Area* and *Area$* would not confuse it, but might confuse you. It's your choice: be careful!

A final limitation is that variable names must not start with a BASIC command or keyword like PRINT, TAB or CLS. So, these are valid names:

```
_my_float
AnInteger%
FirstName$
NUM1
```

... and these are not:

`1_man_went_to_mow`	(starts with a number)
`Went to mow a meadow`	(contains spaces)
`PRINT_TOTAL`	(starts with a keyword)

Naming variables is an art in itself. Once everything was restricted to one or two letter names which tended to render a program an unintelligible mess. Not so now. BBC BASIC will recognise variable names of (practically) any length. These two would be considered different:

```
averyveryveryveryveryverylongname1
averyveryveryveryveryverylongname2
```

However you would soon get fed up of typing those in every time you needed them and there is a hit on the performance speed of your program with lots of long names. Ten to fifteen letters is a good compromise. Shorter if possible, but get some meaning in there so readers can tell what the data represents. There are two

conventions, one puts underscores between separate words to make them easier to read as spaces are not allowed. The other capitalizes the first letter of each word, like so:

```
number_of_times
NumberOfTimes
```

I tend to use the second method, but it's up to you.

As we have seen, variables are created the first time BASIC encounters them in an assignment statement. They must be on the left of the equals sign. An undeclared variable on the right will lead to an error. To try it, change the third line to:

```
Area=3.14159*Radius*radius
```

and run. Always be careful of case!

Correct our little bug above and give the circle program a run through. Instead of closing the window, type:

```
PRINT Radius
PRINT Area
```

The variables are still in memory and are only cleared when the output window is closed or the program is run again. Type:

```
CLEAR
PRINT Radius
```

Now we get an error. CLEAR forces BASIC to wipe the slate clean and forget all its variables. It can be used in programs to re-initialize before doing another run.

Tip: Running your program from the output window

Just to go slightly off-topic for a second, whilst we're talking about using the output window, you can get your program to run again without closing the window. At the cursor type:

```
RUN
```

and off it goes. It is slightly quicker than closing the window and waiting for it to open again.

Static Variables

BBC BASIC pre-declares 26 integer variables called *A%* through to *Z%* . Their properties are the same as other integer variables, but they have one special use. They are not cleared when the program is rerun, even if the program calls the CLEAR command mentioned above. They are not even cleared if the program calls another one. When BBC BASIC is closed, of course, all values are lost. Use of this functionality probably falls into the advanced category but it is worth noting.

Exercises

Now we have a good idea of variables and their use, try the following exercises:

1) Modify the area program to give the circumference of a circle 3.14159*diameter.
2) Print the area of a rectangle, when given the width and height.
3) Assign a string variable to contain your name and output:

```
Hello, xxxx
```

where xxxx is, of course, your name.

Chapter 5 - Numeric Data Types

As we have noted above, there are three types of data that are considered numeric, these are:

Bytes

The range of a byte is 0 to 255. Whole numbers only, no decimal places, please. It can be used to represent the ASCII code for single characters or for conserving memory if you are sure the range is large enough for your data.

Integers

An integer, like a byte, can only contain a whole number. Unlike a byte, it can be negative and the range can be much greater, as seen from the table in the previous chapter. It is advisable to use them wherever possible because access and manipulation is fastest.

Floating point

Floating point numbers, or real numbers as some refer to them, are numbers that contain decimal places. If you were representing the square root of 2 (1.4142) or acceleration due to gravity (9.81) you would use a floating point number. Floating point numbers are also used to represent very big numbers and very small numbers. Examples of these would be things like the speed of light (3.0E8) or the distances between atomic particles (1.0E-10). The 'E' must always be uppercase, unless you override the defaults.

With the noted restriction on ranges, you can move freely between numeric types, BASIC will take care of the details for you, for example if you have a floating point number, say, 3.14159 and you assign it to an integer variable, BASIC will chop off the .14159 and you are left with 3. In other languages, you have to supply a specific conversion, or you get an error. BASIC will do this without prompting which can be a blessing or a curse, depending on how you look at it.

We have already dealt with the four basic mathematical operators: + – * /. What else

can we do with numbers? Well, lots, actually. For a start, there are three additional operations that I would like to introduce.

To try most of these examples, you can use immediate mode and type them straight in. To select immediate mode, go to the toolbar and click the button with the picture of a keyboard which is second to the end, next to help. An output window will open and you can type the commands in without having to keep closing the window and edit the code. There are some programs in here too, if it's coloured, you need the editor; if it's black, the editor or immediate mode.

Powers of numbers ^

Using ^, we can raise a number to the power of another number:

```
PRINT 4^2
```

Which is four squared. Or:

```
PRINT 4^0.5
```

Which is the square root of 4: 2. The power can be pretty much any number or numeric variable you like, just be sure that the recipient variable is of the correct type to handle the result.

MOD and DIV

MOD gives us the remainder after a division.

```
PRINT 20 MOD 7
```

The answer is 6. 7 * 2 = 14 and 20 – 14 leaves us with 6. To see if a number is odd or even we could MOD it by 2. If the result is 0, 2 went into it with no remainder: it's even. If the result is 1, it must be odd. Try this several times with different numbers:

```
PRINT 43 MOD 2
PRINT 666 MOD 2
```

DIV gives us the number of times the divisor went into the dividend and throws away the remainder (if any).

```
PRINT 20 DIV 7
```

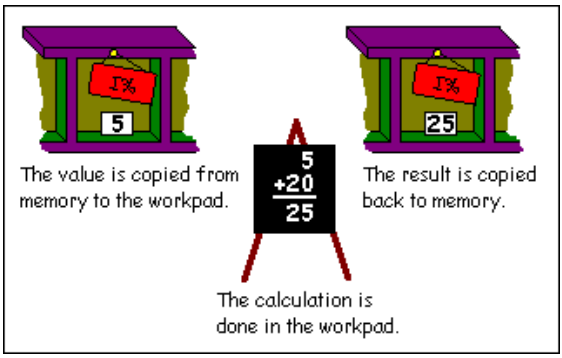
We get 2. These two functions have many uses, for example working out where to print a character on a screen, which should be a whole number.

Variables that change themselves

Look at this line:

```
I%=I%+20
```

Anyone from a strict mathematical background might think this doesn't make sense. How can *I%* be equal to *I%* plus 20. In a computer language, this is perfectly legal. When BASIC does a calculation, it uses a scratchpad area. What it does is get the current value from *I%* , put it in this working area and add 20 to it. Then it takes the result and puts it back in the variable location called *I%* .



You'll see this lots, in fact it's so common that BBC BASIC has a shorthand way of writing it. I could try and explain it but an example is far easier, the above would become:

```
I%+=20
```

It does the same job as the previous line, just saves typing and bytes for the demo version users. If you need to verbalise this when you see it try saying something like: "*I%* is increased by 20". You can do this for all the mathematical operations mentioned above, with the exception of powers. Here is an entirely meaningless program that demonstrates their usage.

```
I%+=1   : REM I% increases by 1
I%-=1   : REM I% decreased by 1
I%*=10  : REM I% multiplied by 10
I%/=2   : REM I% divided by 2
I%MOD =2 : REM I% MODed by 2
I%DIV =2 : REM I% DIVed (!) by 2
END
```

Spaces are not important here. The lines could be written as:

```
I% + = 20
```

However, I prefer to keep the operation and the equals sign together as it's easier to read.

Mathematical Functions

As well as the operations described above, we have an entire armoury of functions that allow us to manipulate numerical values. A function, in this case, is a keyword that can be used in a mathematical expression as it returns a value. Functions usually have a value passed to them which they do something with and return the result. The value passed is called the parameter or argument.

The help files will tell you that brackets are not always necessary when calling functions, but include them as it makes the program easier to understand and it's less to remember in the early days. So, without further ado, here are the mathematical functions that BBC BASIC supports.

Standard Functions

ABS(X)

ABS returns the absolute value of a number. Simply put if X is negative, ABS(X) gives us the positive value.

```
PRINT ABS(-5.5)
PRINT ABS(6.6)
```

Give 5.5 and 6.6 respectively. ABS can be used to find the difference between two numbers without needing to decide which is the higher value:

```
REM ABS demo
A=32.5
B=43.6
PRINT ABS (A-B)
END
```

SGN(X)

SGN stands for sign or signum if you don't want to get confused with sine. If X is negative, it returns –1. If X is zero, it returns 0 and if it's positive, you get +1. Try these:

```
PRINT SGN(23)
PRINT SGN(-5)
PRINT SGN(3-3)
PRINT SGN(25-(2*23))
```

INT(X)

INT rounds a number down to the nearest integer value. For positive numbers, this is pretty obvious:

```
PRINT INT(3.14159)
PRINT INT(99.99)
```

For negative ones, remember that rounding down might give unexpected (though completely explainable) results:

```
PRINT INT(-3.14159)
PRINT INT(-99.99)
```

Tip: Rounding to the nearest integer

If you want to round a number to the nearest whole integer, add 0.5 to it first. This way, if the number is between .0 to .4, it is rounded down and if it is between .5 and .9, it is rounded up.

```
REM Rounding with INT
A=3.14159
B=99.99
PRINT INT (A+0.5)
PRINT INT (B+0.5)
END
```

LN(X) and EXP(X)

LN returns the natural logarithm (base 'e' where e = 2.7183...) of X. EXP returns 'e' raised to the power of X.

```
REM LN and EXP
A=3.4
B=LN (A)
C=EXP (B)
PRINT A;"," ;B;"," ;C
END
```

LOG(X)

LOG returns the logarithm to base 10 of X. There is no opposite antilog function, like LN/EXP. This is quite easy to find however, all we do is raise 10 to the power of the number we wish to antilog:

```
REM LOG and antilog
A=3.4
B=LOG (A)
C=10^B
PRINT A;"," ;B;"," ;C
END
```

SQR(X)

It is possible to find any root of a number by using X^(1/RootRequired). As finding the square root of a number is such a common operation, this keyword is provided for that purpose. Now you can do Pythagoras until the cows come home:

```
REM SQR and right angled triangles
Side1=3
Side2=4
Hyp=SQR (Side1^2 + Side2^2)
PRINT Hyp
END
```

Trigonometric functions

PI

The online help says this is a function. In actual usage, it takes no parameters, so you just use it like a variable that you can't change. It returns a value for good old 22/7 (approximately), used in lots of calculations for circles and angles.

DEG(X) and RAD(X)

All the trigonometric functions work with radians. There are PI radians in 180 degrees (half a circle), so 1 degree = PI/180 radians. To make life a lot more tolerable, DEG(X) will convert a value in radians, X, into degrees and ... wait for it ... RAD(X) converts a value in degrees into radians.

SIN(X), COS(X) and TAN(X)

These supply the sine, cosine and tangent of a given angle X. As noted above, all these work in radians, but conversion is painless thanks to DEG and RAD.

```
PRINT SIN(RAD(60))
PRINT COS(PI/2)
```

ACS(X), ASN(X) and ATN(X)

Given a value X, these will return the corresponding angle in radians for arcsine, arccosine and arctangent.

Special functions

RND(X)

This is the game programmer's favourite function. It returns a random number, great for deciding when aliens should swoop down from the sky, or how much strength you need to wield the Club of Maiming or ... Actually, it's quite a complex beast depending on the value you pass as X. Let's examine the options.

(a) RND with no argument returns a random integer in the range –2147483648 to +2147483647, which is the highest value a 32 bit signed number can hold.

```
PRINT RND
```

(b) RND(X%) where X% is a positive integer value greater than 1. This will return a random number in the range 1 up to and including X%

```
PRINT RND(10)
```

Tip: Generating a random number for different ranges

If you need to generate a random number between, say, 6 and 15, you do this by using RND(10) to get a number and adding an offset to it.

```
PRINT RND(10)+5
```

(c) RND(1) returns a floating point number in the range 0.0 up to but not including 1.0

```
PRINT RND(1)
```

Tip: Converting RND

If converting other dialects of BASIC, this is the usual one that is supplied. To get an integer value you can do this:

```
PRINT INT(RND(1)*10+1)
```

or you can use RND(X%) as described previously.

(d) RND(0) will return the last random number again, but as a floating point number

as described in (c).

(e) RND(X) where X is a negative number. This will seed the random number generator to start from a new number. The numbers generated by RND are not truly random, but there are so many that it would take you a long time to detect when they started to repeat. If you were really bored, you could generate them and find the pattern. Although the random number sequence is set each time BBC BASIC starts so that it doesn't produce the same number, sometimes it is necessary to make the random number generator start from a predefined place in its sequence, this is where the negative number is used.

Operator Precedence

Let's go into immediate mode and try an experiment. Try and predict the result of this line before pressing return.

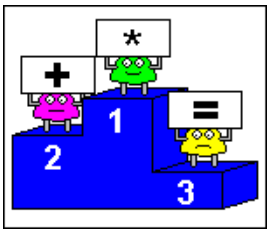
```
PRINT 2+3*4
```

OK, what did you get? Was it the answer you expected? If you predicted 14, you are already familiar with operator precedence. If you expected 20, you might be a bit puzzled, so stick with it and I'll explain. Operator precedence is another phrase for the priority of a mathematical operation. In common maths, multiplication and division have a higher priority than addition and subtraction. Applying this knowledge, we can now break down how our beloved BASIC arrived at the result 14. When presented with the expression, something like this happens:

2+3*4 Multiplication found, do this first

2+12 Now do the addition

= 14



If you want to force BASIC to override the priority, you need to use parentheses:

```
PRINT (2+3)*4
```

This will give 20 as the expression in brackets is forced to be done first. Any calculations within the parenthesis would be done according to the previous rules, including any further expressions with brackets.

Slowly we could build up a list of operation against priority, but to save time, here is the complete list:

Priority	Operation
Highest	literal values, variables, functions, parentheses (), unary+–, NOT
	power ^
	multiplication *, division /, MOD, DIV
	addition +, subtraction –
	comparisons = <> <= >= > < , shifts << >> >>>
	AND
	EOR, OR
Lowest	assignment =

Some of these won't make sense yet, such as the comparison operations, but rather than keep updating the list as we go through, it's all here for you to refer back to. Notice that the lowest priority is the assignment instruction. This means that absolutely everything is done before the value is assigned to the variable lined up to receive the result, which is what you'd expect isn't it?

Exercises

1) Modify the circle program to use PI instead of 3.14159. Save it.
2) Write a program that uses RND to simulate two six-sided dice being thrown. Print the results for each die and the total.
3) Verify that the following formula is true:

```
LOG(X)=LN(X)/LN(10)
```

Chapter 6 - String Data Types

A string is a list of text characters. We tell BASIC that we are dealing with text rather than variable names by enclosing the text in double quotation marks. Applying this, you should be able to see why, if we want to write 'Hello' on the screen we use:

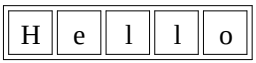
```
PRINT "Hello"
```

and not

```
PRINT Hello
```

The second example would send BASIC scurrying off to its variable list trying to find one called *Hello* . If it just so happened that you had one, it will print its value, most likely you won't so BASIC will complain.

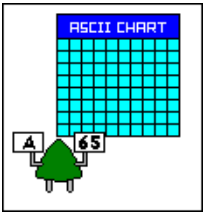
We can think of the way a string variable holds its value as a series of memory locations, each of which holds a character, like this:



The quotes are not kept as part of the text, they are just used as delimiters during programming. Each position is one byte in size, this means it can hold a number in the range of 0-255. So, if a byte can only hold a number, how does it store letters like above? The answer is that the operating system has a lookup table which it uses to translate your text into numeric codes for storing in memory and then translate them back again when we want to print them out. The table is called the ASCII table (American Standard Code for Information Interchange, programmers love acronyms) and it provides a table of corresponding letters, numbers, punctuation marks and other assorted characters. When we store the letters for "Hello", it actually represents them internally like this:

72	101	108	108	111

You can see the full table in the online help under Reference Information. Note that not all the codes have a visible representation and codes less than 32 may cause strange things to happen if you try to print them. These lower codes are often referred to as control characters, they represent things like horizontal tab, form feed etc. Also, the numbers above 127 are a non-standard standard (!) and so will give different characters depending on the font that is being used.



Look at character 32, space. Space is normally filtered out by our brains when we read text, it's there but we ignore it. To a computer, space still needs a representation and so is given a value, just like any of the other punctuation marks such as comma (code 44) or decimal point / full stop (code 46). As BASIC is so picky about spaces, this means that the two strings:

`"Hello"` and **`" Hello "`**

would be considered different, as stated we tend to filter it out, but to the computer it's just another character code.

As numbers have limits, so too do strings. The limits are the code of the character (0 to 255) and the number of characters the string variable can hold, or the length of the string. In BB4W the length can range between 0 and 65535 characters. Zero because you can have a string with nothing in it. In fact when you first declare a string variable, BASIC creates it with zero length, i.e. with nothing in it. This may seem a little odd but is a useful concept. If at any time you wish to set a string to hold nothing, this is how you do it:

```
MyString$ = ""
```

That's double quotes with no gap between them. As a space is a character, this is not the same as:

```
MyString$ = " "
```

If you were to print them out, you would not see any difference, but that, of course, doesn't mean they are the same. A string with nothing in it is variously called an empty string or null string. You'll come across both.

Now we've got to grips with what a string is, what can we do with them? With numeric variables, you can add, subtract, square root etc. Strings are a little more limited. You can only add them:

```
REM Adding strings
S1$ = "Hello"
S2$ = ", "
S3$ = "world"
S4$ = S1$ + S2$
PRINT S4$
S4$ = S4$ + S3$
PRINT S4$
END
```

This is called concatenation, which is a fancy word meaning chain them together. The program copies the contents of *S1$* , splices *S2$* onto the end of it and puts the result into *S4$* . Line 7 adds *S3$* onto the end of *S4$* . There would be nothing to stop you doing all this in one line.

```
REM Adding strings
S1$ = "Hello"
S2$ = ", "
S3$ = "world"
S4$ = S1$ + S2$ + S3$
PRINT S4$
END
```

This is the only mathematical operation that is allowed on strings. None of the others make much sense anyway: how do you find the square root of "Hello"? Don't for one moment think that's it, though, BASIC has a very comprehensive set of functions for manipulating string variables. These are dealt with in the following section.

String Functions

LEN

One of the most useful things we can know about a string is its length. The function LEN tells us exactly this. It must always have one argument, though as is usual, this can be an expression. The result must always be assigned to a numeric value or used in an expression where a numeric value is expected. In immediate mode, try the following:

```
PRINT LEN("Hello")
PRINT LEN("Hello, world")
```

LEN can be used to distinguish between empty strings and strings with no visible characters:

```
REM LEN of an empty string
A$=""
B$=" "
PRINT LEN (A$)
PRINT LEN (B$)
END
```

STRING$

There are times when you want to be able to generate a repeating pattern of text without typing it all in manually. STRING$ does just this. It takes as its parameters a number of repetitions and a base string. It returns a string which is the base string repeated the given number of times:

```
PRINT STRING$(3,"+++===")
```

Here is a little program that will take a string then underline it.

```
REM Underline using LEN and STRING$
Title$ = "BBC BASIC"
L%=LEN (Title$)
PRINT Title$
PRINT STRING$ (L%,"*" )
END
```

Or, just to get carried away, we could put the title in a box:

```
REM Box using LEN and STRING$
Title$ = "BBC BASIC"
L%=LEN(Title$)
PRINT STRING$ (L%+4,"*" )
PRINT "* " ;Title$;" *"
PRINT STRING$ (L%+4,"*" )
END
```

INSTR

Although it's easy to create strings, there are times when we want to inspect their contents. The function INSTR allows us to search a string for a character or pattern of characters. INSTR takes two or three arguments. The first is the string we wish to search. The second is a string containing the characters we wish to search for. The third is optional, we'll get to it in a minute. When supplied with two parameters, INSTR will return the position of the first character in the search string that matches the characters in the list to search for. This example will return the position of the first letter C in the target string:

```
PRINT INSTR("BBC BASIC", "C")
```

The first character in a string is position 1. If INSTR returns 0, it means no match was found.

```
PRINT INSTR("BB4W", "C")
```

The optional third parameter can force INSTR to start at a position other than 1. This means we can search the entire string by remembering the last position returned and starting one character after that.

```
REM INSTR Demo
Posn%=INSTR ("BBC BASIC","C" )
PRINT "C found in position: " ;Posn%
Posn%=Posn%+1
Posn%=INSTR ("BBC BASIC" ,"C" ,Posn%)
PRINT "C found in position: " ;Posn%
END
```

Notice how we have to increment *Posn%* to get it past the first C. If we hadn't, we would have started from position 3 again. As position 3 is a C, the search would have returned the same value again. If the start position is larger than the length of

the string, you get 0 (not found) in return.

INSTR can also search for a sequence of characters in the target string.

```
PRINT INSTR("BBC BASIC","BBC")
```

The thing to be wary of here is how you specify the string to search for

```
PRINT INSTR("BBC BASIC FOR WINDOWS","FOR")
PRINT INSTR("FORTUNE FAVOURS THE BOLD","FOR")
```

Will both tell you that both contain the word "FOR", when clearly the second one doesn't. This again is because BASIC has no concept of language, it just looks for a pattern of characters and when it finds a match, stops. A more correct way would be to search for:

```
PRINT INSTR("BBC BASIC FOR WINDOWS","FOR ")
PRINT INSTR("FORTUNE FAVOURS THE BOLD","FOR ")
```

LEFT$ and RIGHT$

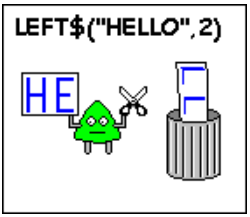
The next two functions return a subsection of a string and are dealt with together as they are functionally similar.

LEFT$ takes two parameters: a target string and a number of characters. It returns a string which is the number of characters in length starting from position 1.

```
PRINT LEFT$("Hello, world", 5)
```

If the number is greater than the total length of the string, you just get the whole string.

```
PRINT LEFT$("Hello, world", 100)
```



LEFT$ will also accept one parameter only:

```
PRINT LEFT$("Hello")
```

This will return all the characters but the last one and is the same as:

```
PRINT LEFT$("Hello", LEN("Hello")-1)
```

It is also possible to use LEFT$ as an assignment. In this mode, LEFT$ will overwrite the characters in the string with the ones being assigned, starting at the first character.

```
REM LEFT$ as an assignment
MyStr$="Hello, world"
LEFT$ (MyStr$,6)="Byebye"
PRINT MyStr$
END
```

If you specify a number less than the length of the replacement, BASIC will only overwrite the number of characters specified. Should you specify more, BASIC will only overwrite up to the maximum characters in the replacement string.

RIGHT$ takes the same arguments as LEFT$ but returns the rightmost number of characters.

```
PRINT RIGHT$("Hello, world", 5)
```

Again, if the number is too big, you just get the whole string back. With only one argument, RIGHT$ will return just the last character.

Predictably, when used in an assignment, RIGHT$ will overwrite the characters at the end of the string.

```
REM RIGHT$ as an assignment
MyStr$="Hello, world"
RIGHT$ (MyStr$,5)="mummy"
PRINT MyStr$
END
```

Exactly what happens if you specify fewer characters than the length of the replacement string is probably best illustrated by example. Change the 5 to 4 in line 3 above and see what happens. It starts 4 characters from the end of the string and copies the first 4 characters from the replacement string. If you tell BASIC to use more characters than are contained in the string, our friendly computer will effectively derive its own number. Substitute 8 in line 3 and see. The replacement doesn't start 8 characters away from the end of the string, it merely works out that the replacement has 5 characters, and starts at that position instead.

Please note that with both LEFT$ and RIGHT$, you cannot lengthen the original string by giving more characters in the replacement than are in the target. BASIC will just truncate the substitute string at the length of the target.

MID$

LEFT$ and RIGHT$ allow us to manipulate the start and end of a string, but what happens if you want to extract from the middle? MID$ will do this for us.

In its more common application, MID$ has three parameters: a string, the start position and a number of characters. As with all strings, the left most character is position 1. Try this:

```
PRINT MID$("Fortune favours the bold", 9, 7)
```

This returns 7 characters starting at position 9 i.e. "favours" in this case.

If the last number is bigger than the length of the string, you just get everything up to the end.

```
PRINT MID$("Fortune favours the bold", 9, 1000)
```

This case is so common that BASIC allows us to omit the final parameter. If you do this, BASIC assumes that you want all the characters from the start position to the end.

```
PRINT MID$("Fortune favours the bold", 9)
```

OK, that was painless enough, but we're not finished. Like RIGHT$ and LEFT$, MID$ can also be used on the other side of the equals sign. This means that you can get BASIC to replace a section of a string:

```
REM MID$ demo
A$ = "Give me patience!!"
MID$ (A$,9,8) = "strength"
PRINT A$
END
```

From the above description, you should be able to guess what it's doing. For completeness: line 3 takes the string "strength" which is 8 characters long and, starting at position 9 in A$, replaces the characters one for one with the characters in "strength".

There are several things to be aware of when dealing with the number of characters. Usually, the number is the same as the length of the replacement string. If the number of characters specified is shorter than the length of the replacement, only that number of characters are copied:

```
MID$ (A$,9,4) = "strength"
```

Also, if the start position in the target string plus the number of characters is greater than the total length of the replacement string, BASIC will only copy characters up to the end of the target string and ignore anything after:

```
MID$ (A$,9,13) = "all your cash"
```

To put it another way, BASIC will not extend the length of the target string.

You can leave out the number of characters. In this case BASIC assumes the length of the replacement string, but still obeys the rules given above.

Now for a little demo that uses INSTR, LEFT$ and MID$. Suppose we have

someone's full name and we want to separate it into first name and surname. We know that the two names are separated by a space, so first we use INSTR to locate the space. Then we copy all the characters up to, but not including, the space into the string that keeps the first name. Next we take all the letters starting after the space up to the end and save them in the surname. Have a crack at this yourself first before looking at my result if you want to, it's the only way to learn.

```
REM Separate names
FullName$ = "Joe Soap"
Posn% = INSTR (FullName$, " " )
FirstName$ = LEFT$ (FullName$, Posn%-1)
Surname$ = MID$ (FullName$, Posn%+1)
PRINT "Your first name is: " ;FirstName$
PRINT "Your surname is: " ;Surname$
END
```

How did you do? There are always as many ways to code the solution to a program as there are people trying to code it, so if you got a different solution that's fine. Also don't be upset if you didn't get it completely right first go, I didn't: it's all part of the programming process.

ASC and CHR$

We have already made the acquaintance of the ASCII table. It is very useful to be able to find the codes that correspond to the letters and vice versa. That's the job of ASC and CHR$.

ASC returns an integer which is the ASCII code for the character passed as a parameter:

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

Gives 65, as expected.
Note also:

```
PRINT ASC("1")
```

Gives 49, which is the code for the character "1", NOT the value 1.

If the string is bigger than one character, ASC just returns the code for the first character. To inspect other positions, we need to use MID$:

```
PRINT ASC(MID$("BB4W",2,1))
```

which gives the code for the second character, "B".

As you may expect, CHR$ does the reverse of ASC: give it a number and it will return a single character string containing the corresponding ASCII code.

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

CHR$ is particularly useful for making strings out of the characters you can't get on the standard keyboard:

```
PRINT "The temperature is 21.2"+CHR$(176)+"C"
```

This can be a useful technique for printing cursor control characters or user defined characters, which are described in a later section.

If you give CHR$ a number which is bigger than 256, BASIC divides it by 256 and gives the character corresponding to the remainder.

Tip: Printing quotation marks

If you want to print a double quote in a string, you can do it in two ways, the first one involves building a string using CHR$(34), which is the code for double quote.

```
Greeting$ = CHR$ (34) + "Hello, world" + CHR$ (34)
PRINT Greeting$
```

The other way is a little trick that BB4W allows us. You can actually put the quote in the string, but you use two double quotes together so BASIC knows that we want to print the quote character and not end the string.

```
Greeting$ = """Hello, world"""
PRINT Greeting$
```

As the quotes in this string are at the beginning and end, there are three lots, which definitely looks odd. Take the beginning, the first indicates the start of the string and the next two tell BASIC to store a quote. The end is the same but in reverse.

VAL and STR$

The next two commands allow us to convert between numeric and string data types.

VAL takes as its argument a string representation of a number and returns the numeric equivalent of that number.

```
PRINT VAL("123")
```

If the string contains non-numeric information, it will convert until it fails:

```
PRINT VAL("123xyz")
```

or if the non-numeric stuff comes first, you just get 0 back.

```
PRINT VAL("xyz123")
```

The counterpart of VAL is STR$, which you probably guessed. You might also have guessed that this takes a number or numeric variable and converts it into a string representation. Now we can add a number to a string:

```
REM STR$ demo
A$ = "The temperature outside is " + STR$ (21.6)
PRINT A$
END
```

There are default settings which control the format of the string produced. This is well documented in the online help and is changeable at runtime if you require, but is a little beyond the scope of this tutorial.

EVAL

The last string command that must be mentioned is EVAL. I'll give a flavour of what it can do rather than a full description because it is such a powerful command. In essence, it allows you to evaluate the contents of a string expression. Take the description of VAL, which converts a string to a number. At some point programmers try, inadvertently or otherwise, something like this:

```
PRINT VAL("22/7")
```

VAL returns 22 as described above. Now try:

```
PRINT EVAL("22/7")
```

Not impressed? Try:

```
PRINT EVAL("SIN(PI/2)")
```

Take it from me, that's not something you get with any old BASIC. You can pass any string expression and EVAL will evaluate it and return a numeric or string value, just as if you had entered the code into a line of a program. As demonstrated above, you can use internal BBC BASIC functions (though commands like CLS etc. will not work). You can even use variables within the program:

```
REM EVAL demo
Side1 = 3
Side2 = 4
Hyp = EVAL ("SQR(Side1^2+Side2^2)" )
PRINT "Hypotenuse is: " ;Hyp
END
```

The possibilities that this presents spiral off into infinity, so that's all I'm going to say about it here.

Exercises

1) Set a string to hold the days of the week like this:

```
"Sun Mon TuesWed ThurFri Sat"
```

All names are 4 characters in length including a space if necessary. Given a number for a day, use MID$ to extract the correct abbreviation for the day.
2) Set a string to hold your first name. Use MID$ and ASC to find the ASCII codes of the letters in the name.
3) Set three strings to hold your first name, second name (if you haven't got one, make it up) and surname. Use LEFT$ to find your initials and concatenation to create a new string in the format "R. T. Russell"

Chapter 7 - Coloured Text

The earlier chapters gave quite a lot of information if you've never encountered this sort of thing before. If you're suffering from mental indigestion, this part is a little more light hearted, just to give time for the rest of it to become comfortable. Eventually, it'll become second nature.

In this section I want to extend PRINT a little further and show you how to print in glorious technicolour instead of black and white as we've been doing so far. To change the colour of the text is easy and involves one new keyword: COLOUR (BB4W will also accept COLOR if you want.)

In general, there are 16 colours we can use to liven up our text, but be wary because exactly how many of these colours are available depends on which MODE you are in. Due to restraints in the original micros like the Acorn BBC, different screen modes had different resolutions and, to conserve memory, different numbers of colours. To change modes you use the command MODE followed by the number of the mode you wish to invoke. Each mode has different resolutions in terms of number of columns wide, rows high and amount of colours available. For example, MODE 4 has 40 columns by 32 rows and only two colours.

There are many different screen modes available and you can find out all about them under MODE in the help files. Most of the examples here are in the default mode unless stated otherwise and so all 16 colours are available. The colours are invoked by using a number to represent each colour. The numbers are:

0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta (blue-red)
6	Cyan (blue-green)
7	White
8	Intensified Black (grey)
9	Intensified Red
10	Intensified Green
11	Intensified Yellow
12	Intensified Blue
13	Intensified Magenta
14	Intensified Cyan
15	Intensified White

If you think of ink and paper, the colour of the ink is often referred to as the foreground colour and the colour of the paper is known as the background. To change the colour of the foreground, we merely call COLOUR X, where X is one of the numbers above - 0 to 15. The next time PRINT is called, the text will be printed in that colour. Easy.

```
REM Foreground colour
COLOUR 4
PRINT "I'm blue!"
END
```

When the program finishes, it leaves the foreground and background colours last selected. It's normally good etiquette to put things back as you found them, so perhaps we should modify the above to do this.

```
REM Foreground colour
COLOUR 4
PRINT "I'm blue!"
COLOUR 0
END
```

We can use COLOUR to change the background colour too. To do this, we add 128 to the above table and any text printed after will have the colour background given. Some programmers actually present this as a sum so it's slightly easier to read as it saves mental arithmetic. I like this but then my mental arithmetic was never too hot: it's up to you.

```
REM Background colour
COLOUR 128+6
PRINT "Hello, world"
COLOUR 128+15
END
```

You have to use a different COLOUR statement each time you change the foreground or background.

```
REM Foreground and background colour
COLOUR 4
COLOUR 128+6
PRINT "Hello, world"
COLOUR 0
COLOUR 128+15
END
```

If you are changing screens in your program, you frequently want to wipe the slate clean so information from the previous screen is not interfering with the current one. BASIC has a command to do this: CLS. This stands for CLear Screen, but takes less typing. To use it is quite simple, three letters and the previous screen is consigned to history:

```
REM CLS demo
PRINT TAB (29,10);"BBC BASIC For Windows"
WAIT 200
CLS
PRINT TAB (36,10);"Rules!!"
END
```

Should we want to make the whole of the screen change colour, first set the background colour then call CLS, like this:

```
REM Setting the screen colour
COLOUR 128+7
CLS
COLOUR 2
PRINT TAB (10,10);"Hello, world"
COLOUR 0
END
```

For a mode with limited colours it is possible to use the COLOUR command and substitute the defaults with others from the above table. The default colours for mode 4 are a black background with white text. To change the text colour to magenta call COLOUR 1,5 like this:

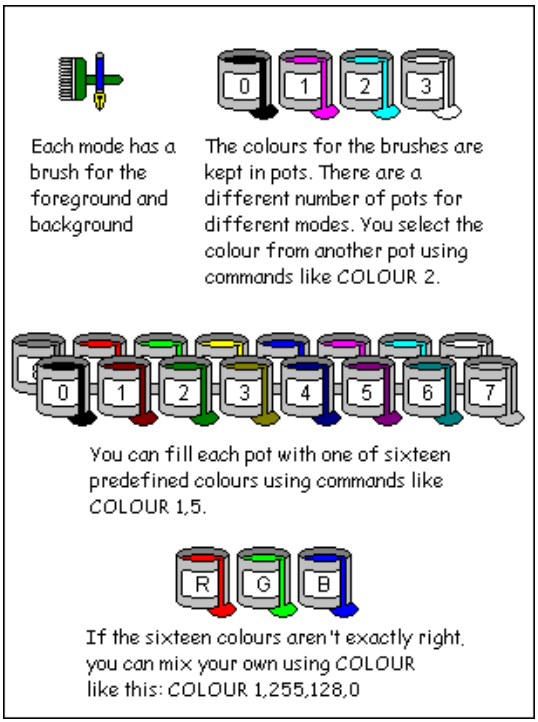
```
REM Setting the screen colour
MODE 4
PRINT "Default colour"
COLOUR 1,5
PRINT "Substituted colour"
END
```

This raises an interesting point. Each mode has a varying number of colours available. Mode 4 has two colours: 0 and 1, mode 5 has four: 0 to 3 and so on. When using COLOUR X%, the actual colour of the text may not directly correspond to the table above. Rather, each mode has a number of boxes into which one of the sixteen colours is slotted. If we don't like the colour BASIC chooses as a default for that box, we can change it for another as in the previous example. The box is referred to as the logical number whilst the colour is referred to as the physical number.

There are times when the default sixteen colours just aren't enough. Like when your

mother just has to have Seagull Sunset Red for the bathroom. As we have just seen it is possible to make BASIC swap the colours we don't want for others from the table of sixteen, but we can also create our own and substitute these into the logical boxes. We can think of each colour for what it is: a mixture of red, green and blue and tell BASIC for colour 3 don't print yellow, print my new colour with this mix of red, green and blue. To do this, we invoke COLOUR with four parameters. The first is the colour number we wish to change, the second, third and fourth are the red, green and blue components respectively each with a range of 0 - 255. Example, to set colour 3 to orange (red = 255, green = 128, blue = 0), we would do the following:

```
REM Changing colours
REM Change colour 3 to orange
COLOUR 3,255,128,0
REM Now we've set the colour change to it
COLOUR 3
PRINT "Hello, world"
COLOUR 0
END
```



Even when the program stops, the colour will retain its new value. You can also change the colour as many times as you want, each time subsequent PRINTs will use the new colour without affecting anything previously printed with that colour index:

```
REM Changing colours
REM Change colour 3 to orange
COLOUR 3,255,128,0
REM Now we've set the colour change to it
COLOUR 3
PRINT "Hello, world"
REM Change colour 3 to brown
COLOUR 3,128,64,0
PRINT "Hello, world"
COLOUR 0
END
```

Once we've redefined a colour, it is quite easy to get back to the original colour for that particular box. If we are in a sixteen colour mode, just substitute the correct colour number from our little table and the default colour is restored again. For other modes, look in the help and it will tell you which are the defaults for that mode.

If you've got really carried away, changed lots of colours and now want to restore the defaults, BB4W has a special command that will do just this. The command belongs to a family of commands that all affect the output to the screen in some way and are hence called VDU commands. To invoke it we type VDU and one or more numbers. The effects of the VDU command depend on the number that follows and are fully described in the help, the one we require is VDU 20. Here's how we do it:

```
REM Restoring colours
MODE 6
REM Change colour 3 to orange
COLOUR 3,255,128,0
REM Now we've set the colour change to it
COLOUR 3
PRINT "Hello, world"
REM Reset the colours back to the default
VDU 20
REM Reselect the foreground colour
COLOUR 3
PRINT "Hello, world"
END
```

Resetting the colours with VDU 20 also restores the default choice for the background and foreground colours.

Tip: Changing colours whilst printing

We can embed colour commands into a string variable. This way we can change colours on the fly whilst printing. To do this, we use CHR$(17) followed by CHR$(Colour_Number) before the string we want to print:

```
REM Changing colours in mid stream
PRINT CHR$ (17)+CHR$ (4)+"Hello" ;
PRINT CHR$ (17)+CHR$ (3)+" World"
COLOUR 0
END
```

CHR$(17) in this instance acts like the COLOUR command. Obviously, these can be hard to read, so if you're using this sort of thing a lot, you can make some strings up with this information pre-declared:

```
REM Changing colours in mid stream
Blue$=CHR$ (17)+CHR$ (4)
Yellow$=CHR$ (17)+CHR$ (3)
PRINT Blue$+"Hello" +Yellow$+" World"
END
```

Exercises

1) Examine the different MODEs in the help file, try printing text in MODEs 1 to 6 to get the feel for what they look like.
2) Modify the Changing colours program to produce new random colours by using three RND(255) statements in the call on lines 2 and 5.

Chapter 8 - Interacting with the User

The earlier examples, like our circle program, have an obvious flaw. In order to run them, users have to manually edit the code and put new values into the initial variables. For some reason, most computer users, even those who think they know quite a lot about computers, turn pale at the thought of actually programming and would rather pay to have someone else do it for them. Strange really, but it's kept me in employment for a long time. What we need is a way of allowing the user to enter the value, for example, of the radius of our circle. This way the poor dears will never have to soil their hands with CODE. The designers of BASIC have already thought of this. Try this:

```
INPUT A$
PRINT "You entered: " ;A$
END
```

When run, the screen sits there with a question mark until you type something and press Enter. After this, execution continues as normal. Great stuff. Now we can modify our circle program to make it interactive:

```
REM Area of a circle
INPUT Radius
Area=PI *Radius^2
PRINT "The area of your circle is " ;Area
END
```

I've updated line 3 to include the things we've already covered. It would be nice though to make the program tell the user what it's expecting. After all, even less than writing code, users don't like reading manuals. (Most programmers don't either, actually.) INPUT is another command like PRINT with lots of variations. For a start, you can put a text string to tell the user what the program is expecting:

```
INPUT "What is the radius" ; Radius
```

If you leave out the semicolon (;) BB4W will not print the question mark, so you

don't have to phrase your prompt as a question:

```
INPUT "Please enter the radius: " Radius
```

Also, it is possible to prompt for more than one variable at a time providing they are separated by commas:

```
INPUT "Enter the width and height " W,H
PRINT "Rectangle has an area of " ;W*H;" m^2"
END
```

You can enter the values separated by commas or Enter. BASIC will keep prompting until it has enough data to fill the variables specified. It is also possible to split the prompt up:

```
INPUT "Enter the width " W " and height " H
```

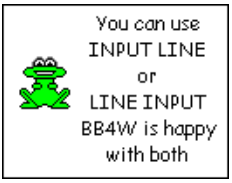
Finally, although there is more in the online help, you can put your prompt anywhere on the screen using TAB:

```
INPUT TAB (5,5);"What is your name" ; N$
PRINT TAB (5,6);"Hello, " ;N$
END
```

INPUT LINE

There is a variation to INPUT that is worth mentioning at this point. By including the keyword LINE, we can make BASIC accept everything the user types and put it into one string variable. Ordinarily, INPUT will remove leading spaces and stop at the first comma after the initial text. By using INPUT LINE, we override this behaviour. To illustrate:

```
REM INPUT vs. INPUT LINE
INPUT "Enter your full address: " Address$
PRINT "Address with INPUT: " ;Address$
INPUT LINE "Enter your full address: " Address$
PRINT "Address with INPUT LINE: " ;Address$
END
```



It's very much horses for courses here, one variation is not better than another, it just depends what you need.

Tip: Entering numbers
When employing INPUT and INPUT LINE, the user has to be intelligent about his choice of input. If you're expecting a number and a string is entered, BASIC will return zero. A good way round this is to use a string to catch the entry then convert it to a number with VAL. Bearing this in mind INPUT will serve us as a good way to write interactive programs.

Exercises

1) The volume of a cylinder is PI*Radius^2*Length. Write a program that will ask for radius and length then give the volume.
2) Have a program prompt to enter your full name. Print the number of characters (including spaces) in the name.

Chapter 9 - Conditional Execution

In the cases we have seen so far all the code is executed when the program is run. Many times this is not what is required. There needs to be a way of running a block of code only if certain conditions are met. This need is so fundamental to programming languages that all have it and many (PASCAL, C/C++, JAVA and, of course, BASIC to name some) use the same word: IF.

Example:

```
INPUT "Enter your score: " Score%
IF Score%>40 THEN PRINT "You passed!"
END
```

Line 2 makes a test on the variable *Score%* and the PRINT statement is only run if the condition is found to be true. A full list of comparisons is given below, but as can be seen common sense and basic maths were used in their implementation.

Expression	Evaluates to true when
A < B	A is less than B
A <= B	A is less than or equal to B
A = B	A is equal to B
A >= B	A is greater than or equal to B
A > B	A is greater than B
A <> B	A is not equal to B

We can expand our age test program:

```
INPUT "Enter your score: " Score%
IF Score%>40 THEN PRINT "You passed!"
IF Score%=40 THEN PRINT "You just passed!"
IF Score%<40 THEN PRINT "You failed!"
END
```

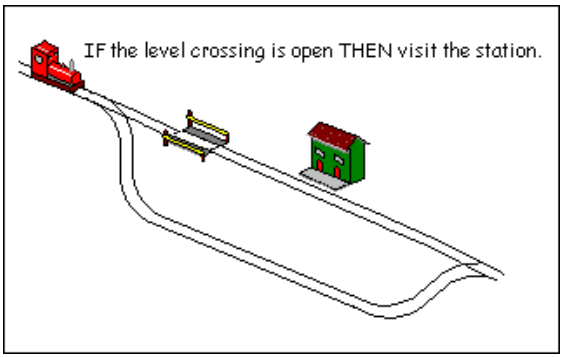
Furthermore, A and B can be expressions in themselves:

```
IF MyAge%=YourAge%+5 THEN PRINT "I'm older"
```

When used in an expression like the above, = acts as a comparison not an assignment, so no values would actually be altered during this test.

It is perfectly possible to test strings as well. In this case, the test is made on the value of the characters from the ASCII table, as mentioned in the chapter on strings. When comparing strings, BASIC will start with the first character of the first string and compare it with the first character of the second string. If they are the same, it will test the second characters of both strings and so on, until it can make a decision. Here are some examples:

```
IF "abc" ="abc" PRINT "both the same"
IF "abc" <"bcde" PRINT "a comes before b"
IF "abc" <"abde" PRINT "c comes before d"
IF "abc" <>"ABC" PRINT "different cases"
IF "ABC" <"abc" PRINT "upper case is first"
END
```



AND and OR

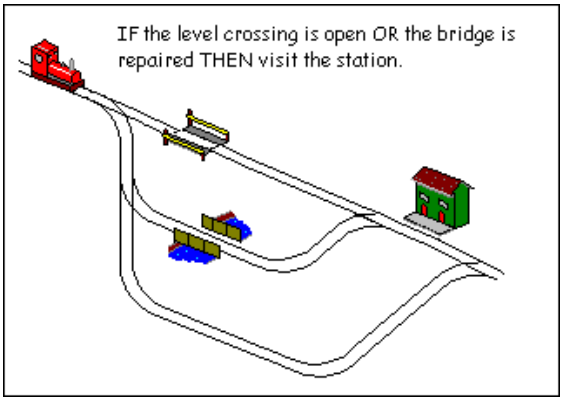
Type in and run this little program:

```
REM Quiz
INPUT "Is London the capital of England" ;A$
IF A$="Y" THEN PRINT "Correct"
IF A$="N" THEN PRINT "Wrong"
END
```

Line 3 above would only work if the user typed an uppercase 'Y'. If they typed 'y', they would never find out how clever they are. Not very friendly. It is possible to combine more than one condition at a time using keywords AND and OR. We can modify lines 3 and 4 to do this:

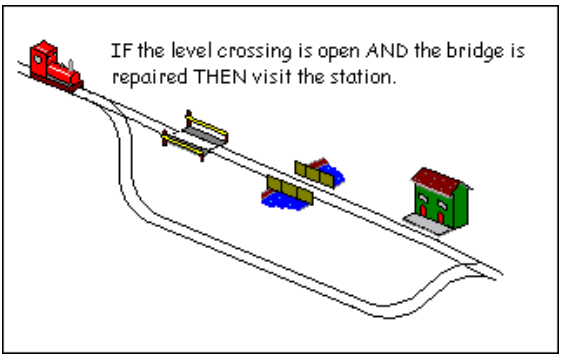
```
REM Quiz
INPUT "Is London the capital of England" ;A$
IF A$="Y" OR A$="y" THEN PRINT "Correct"
IF A$="N" OR A$="n" THEN PRINT "Wrong"
END
```

This is much more usable.



We can also add a trap for an incorrect response:

```
REM Quiz
INPUT "Is London the capital of England" ;A$
IF A$="Y" OR A$="y" THEN PRINT "Correct"
IF A$="N" OR A$="n" THEN PRINT "Wrong"
IF A$<>"Y" AND A$<>"y" \
\ AND A$<>"N" AND A$<>"n" \
\ THEN PRINT "Invalid response"
END
```



It is customary at this point to provide truth tables of the various conditions:

AND		
When A is	AND B is	Result
False	False	False
True	False	False
False	True	False
True	True	True

OR		
When A is	OR B is	Result
False	False	False
True	False	True
False	True	True
True	True	True

Reading the third line of the AND table tells us if A is false and B is true, the overall result will be judged as false. Common sense really.

EOR

EOR (which to the delight of students everywhere is pronounced like the donkey in Winnie the Pooh) stands for Exclusive OR. It will test for a condition like OR but will only be true when one test is true, not both. Perhaps a truth table will explain it better.

EOR		
When A is	EOR B is	Result
False	False	False
True	False	True
False	True	True
True	True	False

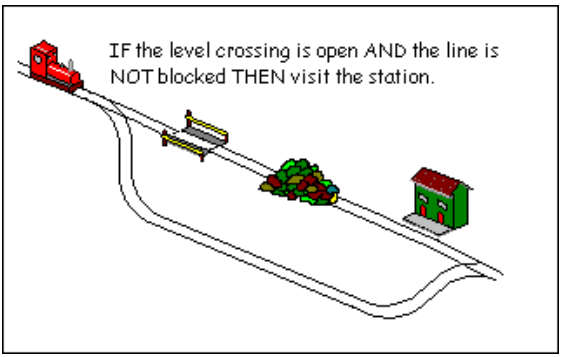
NOT

NOT is another word that can be used in comparisons. Unlike AND, OR and EOR, it is not used to join statements but simply negates a condition. As an example, take the test for a negative response at the end of our quiz program:

```
REM ...
IF A$="Y" OR A$="y" THEN PRINT "Correct"
IF A$="N" OR A$="n" THEN PRINT "Wrong"
IF A$<>"Y" AND A$<>"y" \
\ AND A$<>"N" AND A$<>"n" \
\ THEN PRINT "Invalid response"
END
```

As line 4 is a bit long, we decide that any response which is not "Y" or "y" can produce the same message. Line 3 can be rewritten and line 4 removed:

```
IF NOT (A$="Y" OR A$="y" ) THEN PRINT "Wrong"
END
```



There are a large variety of words that can be used here. Don't let them confuse you, they are there to be used to make your program more intelligible. If you find your logic tests are tying your head in knots (or NOTs, groan), there's probably a better way to express it, try restructuring.

ELSE

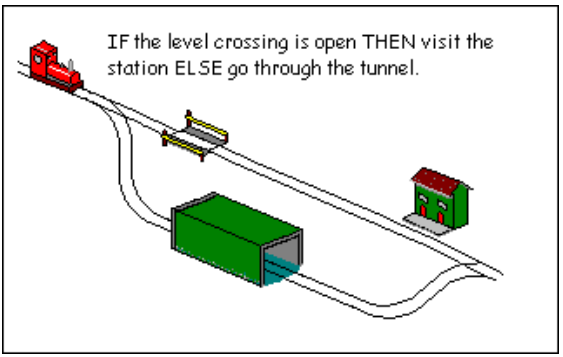
In this example, we want to set a variable to say whether the user passed or failed a test. The variable could be used in IF statements throughout the rest of the program. We could do this:

```
INPUT "Enter your score: " Score%
IF Score%>=40 THEN Pass=1
IF Score%<40 THEN Pass=0
END
```

However, it does seem a bit inefficient using two mutually exclusive lines. In this case, IF can be combined with ELSE.

```
INPUT "Enter your score: " Score%
IF Score%>=40 THEN Pass=1 ELSE Pass=0
END
```

If *Score%* >=40, the first assignment is executed. If it's false, the second assignment is executed. It's not too different from English really and if the pass mark changes, there's only one line to alter.



The word THEN is usually optional on single line IF statements (multi-line ones are coming up next) provided the meaning is clear:

```
IF Score%>=40 Pass=1 ELSE Pass=0
```

Multi-line IF statements

BB4W allows you to split the IF statement across multiple lines. For example, it's possible to do something like this:

```
IF Salary>1E6 THEN BuyYacht=1 : BuyVilla=1
```

If *Salary* is more than 1 million (assuming we're not in Turkey here!) then both the variables will be set to 1. If we were really affluent, we might also want to buy a helicopter as well. Adding to our line puts us in danger of unreadable lines that scroll off the edge of the screen. Isn't it much easier to read it like this:

```
IF Salary>1000000 THEN
  BuyYacht=1
  BuyVilla=1
  BuyHelicopter=1
ENDIF
```

That way we can run lots of code conditionally. When THEN is the last statement on a line (not even a REM is allowed here), BB4W knows that it is about to be presented with a multi-line IF. The end of the block is denoted by the corresponding ENDIF statement. Never put a space in ENDIF. Some other dialects of BASIC will accept END IF as two separate words. BBC BASIC won't. It will take the first END to mean the end of the program and stop dead. The editor will automatically indent blocks like this so it is easier to read. If the indentation goes awry, there's a warning that something is wrong.

You can also use ELSE in this structure as well:

```
IF Salary>1000000 THEN
  BuyYacht=1
  BuyVilla=1
  BuyHelicopter=1
ELSE
  PRINT "Work, work and more work"
ENDIF
```

Each IF statement can only have one corresponding ENDIF and, optionally, one ELSE. It is possible to nest statements:

```
IF Salary>1000000 THEN
  BuyYacht=1
  IF ReallyExtravagant=1 THEN
    BuyVilla=1
    BuyHelicopter=1
  ENDIF
ELSE
  PRINT "Work, work and more work"
ENDIF
```

Operator precedence

It is possible to combine comparisons with AND and OR statements in any

combinations that logic allows, but be careful because BASIC's interpretation might not be yours.

```
IF Raining=1 OR Snowing=1 AND Boots=0 THEN
  PRINT "You'll need your boots"
ENDIF
```

If we go back to our table of operator precedences in Chapter 5, we can see that AND has priority over OR. The line will be interpreted like this:

```
IF Raining=1 OR (Snowing=1 AND Boots=0) THEN
  PRINT "You'll need your boots"
ENDIF
```

If it's raining, you'll always be told to take your boots, whether you have them or not. Only if it's snowing and you've no boots will you be prompted for them. When in doubt, use parentheses:

```
IF (Raining=1 OR Snowing=1) AND Boots=0 THEN
  PRINT "You'll need your boots"
ENDIF
```

TRUE and FALSE

As an end to this section, I would like to mention that BBC BASIC has two predefined variables: TRUE and FALSE. These actually have values, to see them go into immediate mode and type:

```
PRINT TRUE, FALSE
```

This might not seem like a particularly useful thing to have, but it is great for testing yes / no situations and making a program more readable.

```
REM Using TRUE / FALSE
INPUT "Enter password " Password$
IF Password$="Super" THEN
  Supervisor%=TRUE
ELSE
  Supervisor%=FALSE
ENDIF
IF Supervisor% THEN
  REM Show main configuration screen
  PRINT "Welcome, master"
ELSE
  REM Show ordinary user's screen
  PRINT "What do you want now?"
ENDIF
END
```

The variable used to test true or false situations is often referred to as a flag. If this program was a lot bigger we could use the flag repeatedly instead of making the same comparison. That way, when the password changes, we only have to change one line of code, rather than lots. Look at line 8. This is equivalent to *IF Supervisor%=TRUE ...* but BASIC is clever enough to insert the implied *=TRUE* for us when it runs, again making the code more readable. If you wanted to test the reverse condition, you would still need to put *IF Supervisor%=FALSE ...* or *IF NOT Supervisor% ...*

Tip: Disabling a block of code

Any IF expression is evaluated until it returns TRUE or FALSE. To disable code whilst testing, we can wrap it in block, like this:

```
REM Some code
IF FALSE THEN
  CLS
  PRINT "Introductory screens"
ENDIF
REM Rest of code
```

IF looks for the condition to be true. FALSE, by definition, is never true, therefore the code will never be run.

Exercise

Below is a sample output from two runs of a program that acts as a simple calculator:

```
Enter first number: 5
Enter second number: 6
Enter 1 to add or 2 to subtract: 1
5 + 6 = 11
>RUN
Enter first number: 45
Enter second number: 55
Enter 1 to add or 2 to subtract: 2
45 - 55 = -10
>
```

Can you write the program that produces the above screen? Allow one INPUT for each number, one INPUT for the operations. Use IF to select the correct PRINT statement to display the result. Extra marks if you expand to include multiply and divide.

Chapter 10 - Selection using CASE

There are times when you will need to test the same variable for a number of different cases, of which only one will result in any code being executed. An example that springs to mind is a menu system. Let's build it.

```
REM Simple menu system
CLS
PRINT "Press 1 for option 1"
PRINT "Press 2 for option 2"
PRINT "Press 3 for option 3"
INPUT "Enter choice: " Choice%
IF Choice%=1 THEN PRINT "You chose option 1"
IF Choice%=2 THEN PRINT "You chose option 2"
IF Choice%=3 THEN PRINT "You chose option 3"
END
```

If the user chooses 1, the code associated with choice 1 is run. When finished BASIC goes off and checks if *Choice%* is 2 or 3 even though it clearly cannot be. This is not very efficient and, if each option evolves into several lines of code, it becomes easy to lose track of the IFs. Enter IF's bigger brother: CASE. We'll rewrite the program using CASE:

```
REM Simple menu system
CLS
PRINT "Press 1 for option 1"
PRINT "Press 2 for option 2"
PRINT "Press 3 for option 3"
INPUT "Enter choice: " Choice%
CASE Choice% OF
  WHEN 1: PRINT "You chose option 1"
  WHEN 2: PRINT "You chose option 2"
  WHEN 3: PRINT "You chose option 3"
ENDCASE
END
```

I'm sure you can see what's happening here. The CASE line tells BASIC it's using *Choice%* as a comparison variable. Each WHEN line gives a value to compare with. Upon finding a match, the associated code is run up to the next WHEN. BASIC then jumps to the ENDCASE statement and continues the rest of the program, ignoring all the other WHENs. As with ENDIF, ENDCASE is always one word.

WHEN can have a range of values, modify the CASE statement to add:

```
CASE Choice% OF
  WHEN 1: PRINT "You chose option 1"
  WHEN 2: PRINT "You chose option 2"
  WHEN 3: PRINT "You chose option 3"
  WHEN 4,5,6: PRINT "Option not yet implemented."
ENDCASE
```

CASE is just as capable of testing strings:

```
REM Geography quiz
PRINT "What is the capital of France:"
PRINT "a) Paris"
PRINT "b) London"
PRINT "c) Madrid"
INPUT "Enter a,b or c: " Reply$
CASE Reply$ OF
  WHEN "A" , "a" : PRINT "Correct"
  WHEN "B" , "b" : PRINT "Sorry, that's England"
  WHEN "C" , "c" : PRINT "Sorry, that's Spain"
ENDCASE
END
```

If none of the WHENs are executed, we can catch the fact and do something about it. The keyword OTHERWISE must be the last case before ENDCASE and has no conditions.

```
REM Geography quiz
PRINT "What is the capital of France:"
PRINT "a) Paris"
PRINT "b) London"
PRINT "c) Madrid"
INPUT "Enter a,b or c: " Reply$
CASE Reply$ OF
  WHEN "A" , "a" : PRINT "Correct"
  WHEN "B" , "b" : PRINT "Sorry, that's England"
  WHEN "C" , "c" : PRINT "Sorry, that's Spain"
  OTHERWISE : PRINT "Sorry, invalid response"
ENDCASE
END
```

It is, as stated, optional. Each WHEN could be, and often will be more than one line of code. If the tests in the WHENs overlap or are the same, the first one is executed and the second one ignored.

That's about it for CASE, very useful, especially when dealing with subroutines later on. One final word of warning, however, don't be tempted to put a comment (REM) after the OF or the whole thing won't work and you'll be scratching your head as to why.

Exercise

Recreate an executive decision maker. Generate a random number 1 to 6 and print a message depending on the result:

1 - Hire a yes man
2 - Fire someone
3 - Delegate
4 - Cancel all overtime
5 - Give yourself a rise
6 - Raid the pension fund

Chapter 11 - Looping with FOR

Back in the '80s, I used to work as a programmer for a company that made weighing machines. One project I had was to design a program to load weights onto a series of scales and take readings. The test station was to test four scales in turn. A colleague had tried the task and failed because he developed the code for one machine and then tried to copy it four times. A BBC Model B had 32k of memory: his program wouldn't fit. He was completely unaware of the computer's ability to run the same code repeatedly using a FOR loop. Effectively, I rewrote the program so it did this:

```
REM Scale test
FOR Scale%=1 TO 4
  REM load some weights
  REM get some readings
  REM print the results
NEXT Scale%
END
```

Line 2 is the start of the loop. It tells the computer to use a variable (*Scale%* in this case) to keep count of how many times a loop has run and stop it when it gets past a limit, in this instance 4. When the line is first encountered, *Scale%* is created and set to 1. Lines 3-5 are then run. When line 6 is executed, the program jumps back to line 2, increases *Scale%* by 1, compares it with the final value, sees that it is less and runs lines 3-5 again. On the last repeat, when *Scale%* is 4, the program increases *Scale%* by 1, sees it has exceeded the final value and jumps to line 7 where it carries on with the rest of the program.

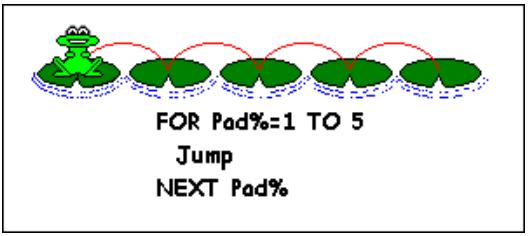
That's quite a lot of functionality for 2 lines of code, but as you've probably guessed, it doesn't stop there. For a start, the counter variable can be used in the loop itself so the code knows where it's up to:

```
REM Scale test
FOR Scale%=1 TO 4
  PRINT "Currently testing scale: " ;Scale%
  REM load some weights
  REM get some readings
  REM print the results
NEXT Scale%
END
```

... or to make decisions:

```
REM Scale test
FOR Scale%=1 TO 4

  CASE Scale% OF
    WHEN 1 : PRINT "Testing first scale"
    WHEN 2 : PRINT "Testing second scale"
    WHEN 3 : PRINT "Testing third scale"
    WHEN 4 : PRINT "Testing fourth scale"
  ENDCASE

  REM load some weights
  REM get some readings
  REM print the results
NEXT Scale%
END
```

... or anything else you may need it for. Obviously, the counter has no validity outside the loop, so don't use it there.



The start and finish values can take any logical value:

```
FOR Count%=-10 TO 10
  PRINT Count%
NEXT Count%
```

or they can be variables or expressions:

```
FOR Count%=MinValue% TO MaxValue%-10
  PRINT Count%
NEXT Count%
```

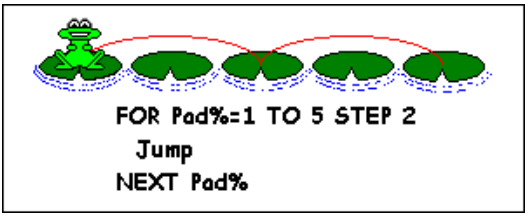
Each counter will start at the first value and increase by one until it passes the second value. You, as a programmer, must be careful here. If the final value is less than the start value, BB4W won't complain but you might be puzzled as to why your loop isn't behaving. Different dialects of BASIC react differently to this situation. BBC BASIC will run the loop at least once whatever values you give it.

There may be times when you want the program to repeat loops in increments other than one. Here, a new keyword can be used to force this. The following code will print all the odd numbers up to 20.

```
REM STEP demo
FOR Count%=1 TO 20 STEP 2
  PRINT Count%
NEXT Count%
END
```

The keyword STEP forces the increment to the value specified. Again, it can be any logical value or expression for the type of variable you are using.

```
FOR I=1 TO 5 STEP 0.5
  REM ...
```



It can also be negative:

```
FOR I%=10 TO 1 STEP -1
  REM ...
```

In this case, BASIC works out that it's going in reverse and instead of stopping when *I%* is greater than 1, which it would do instantly, it stops when *I%* is less than 1 - makes sense really.

As with any other construct, it is possible to nest one inside another:

```
FOR I%=1 TO 5
  FOR J%=1 TO 5
    PRINT I%,J%
  NEXT J%
NEXT I%
```

Here, the inner loop is executed 5 times for each time the outer loop runs. The inner loop has to be fully contained in the outer loop. Try swapping NEXT *J%* and NEXT *I%* and you will produce an error.

After all these variations on FOR, the other end of loop is not very exciting. NEXT is used as a placeholder to denote the end of the loop. The name of the count variable is not really needed as BASIC is clever enough to work out which NEXT ties up with which FOR. I prefer to put them in as it makes the code easier to read, especially if the loop covers more lines than can fit on the screen at one time.

In our nested example above, line 4 could have been replaced with:

```
NEXT J%,I%
```

Try it, remove or REM line 5 first. The program will run as before doing 5 inner loops for each outer loop. BASIC will even accept:

```
NEXT ,
```

Personally, I don't particularly like this method as it is not obvious where loops end and it messes up the indentation that the editor so neatly provides. That's just my preference though and you will see this used in other code.

Tip: Warnings with FOR loops

To end, I would like to mention several common errors that experience has taught:

a) Don't jump out of a loop using GOTO; if it's necessary, force the counter to a number above the final value and let it end naturally.

b) Please, please, please don't use conditional NEXTs. People condemn GOTO for encouraging bad programming practices, but it is possible to do bad things with any command. What I'm talking about is this sort of thing:

```
REM ... program
FOR I%=1 TO MaxNumber
  PRINT I%^2
  IF I% MOD J%=0 THEN NEXT
IF I%-1=J% NEXT ELSE J%=I%+1 : NEXT
REM Rest of program ...
```

There are so many NEXTs here that the editor gets confused, and so will you. Always have only one NEXT for each FOR and give it a line to itself so the indentation allows you to clearly see where the loop starts and ends. You can always use block IFs to run code conditionally if required.

c) It's common amongst programmers of many languages to use *I%* and *J%* as counters. So much so that it is easy to forget and reuse the same counter deep within the code of another loop:

```
FOR I%=1 TO 10
  REM ... several pages of code ...
  FOR J%=1 TO 10
    FOR I%=1 TO 99
      REM Problem here using I% as we've
      REM just reset the first loop
```

You'll see what I mean one day because everyone does it, even though they've been warned!

Exercises

1) Write a program with a FOR loop to print the 5 times table. Each line should be in the format:

```
1 * 5 = 5
2 * 5 = 10
...
12 * 5 = 60
```

2) Use two nested FOR loops to draw a rectangle on the screen by using a line like PRINT TAB(X,Y);"*"

Chapter 12 - Other loops: REPEAT and WHILE

REPEAT .. UNTIL

Our little circle program has had a rest just recently, so let's wake it up again and discuss conditional loops. As we last left it when the user had finished, they had to run it again. It would be nice if the program looped until the user had finished with it. A FOR loop would be no good here as we have no idea how many times our user would want to run the program. This is where we meet our first conditional loop: REPEAT UNTIL. As usual, here's an example which I'll explain afterwards:

```
REM Area of a circle
REPEAT
  INPUT "Enter a radius: " Radius
  Area = PI *Radius^2
  PRINT "Area of your circle is " ;Area
  INPUT "Another go? Y/N " Reply$
UNTIL Reply$="N"
END
```

You can guess here that the program carries on looping until the user presses "N". The loop starts with line 2: REPEAT. There are no conditions or other keywords here, just the one word. BB4W indents all the code following and recognizes UNTIL in line 7 as the end of the loop. UNTIL tests for a condition in the same way as an IF does. If the condition is false, back we go to line 2 for another round. If the condition is true, the loop is ended and execution continues at the next line. All the intelligence happens in the UNTIL line, REPEAT just marks the beginning of the loop.

If you've tried the program, you'll have noticed that we've got our old problem back again: assuming the user enters an uppercase "N". The test that UNTIL does can be as complex as any IF statement and uses exactly the same logic to work things out, so we can change line 7 to:

```
UNTIL Reply$="N" OR Reply$="n"
```

... and all will be well again. Armed with our new knowledge, I'm sure you can see another candidate for a REPEAT. Line 6 asks for *Reply$* , we can test this until we get a valid response before continuing. Try it yourself first then compare results.

```
REM Area of a circle
REPEAT
  INPUT "Enter a radius: " Radius
  Area = PI *Radius^2
  PRINT "Area of your circle is " ;Area
  REPEAT
    INPUT "Another go? Y/N " Reply$
  UNTIL INSTR ("YyNn" ,Reply$)<>0
UNTIL INSTR ("Nn" ,Reply$)<>0
END
```

I used INSTR here, but you could equally have used *Reply$="Y" OR Reply$="y"* etc.

The main problem you will find with conditional loops is the endless loop syndrome: always be sure to set an exit condition in the loop.

```
REM Endless REPEATs
Done = FALSE
REPEAT
  PRINT "Have I missed something here?"
UNTIL Done
END
```

WHILE .. ENDWHILE

REPEAT loops test for a condition at the end of the loop. This means that you always run the code in the body of the loop at least once. Sometimes this is not desirable. A WHILE loop tests for a condition before entering the loop and running the code therein. Somewhat differently, the code is only run when the test condition is true, as opposed to REPEAT which runs while the test condition is false. Watch for this one. The end of the loop is denoted by ENDWHILE, with the usual warning about not splitting into two words.

Here is a program that starts with 1 and keeps halving it until BASIC rounds to zero, to try and find the smallest number BB4W will hold. This doesn't actually give quite

the smallest number, but it's pretty close.

```
REM Smallest floating point using WHILE
Number = 1
WHILE Number > 0
  LastNumber = Number
  Number = Number / 2
ENDWHILE
PRINT "The value of number is " ;Number
PRINT "The previous value was " ;LastNumber
END
```

Again, the logic for the test condition can be as complex as is necessary.

Tip: Exiting loops

The FOR loop is quite forgiving when it comes to testing the exit condition. If you exceed the final value BASIC will not mind and terminate the loop as expected. Not so with REPEAT and WHILE. Look at the following code:

```
REM Endless loops
I%=1
REPEAT
  I%+=2
  PRINT I%
UNTIL I%=10
END
```

The loop will never exit because BASIC looks for an exact match. It is much better with REPEAT and WHILE not to use precise values unless you are 100% sure they can be met. This is particularly true in the case of floating point numbers. To fix the example, replace '=' with '&gt=' in the UNTIL line and all will be well again.

Exercises

1) Produce a program that will ask for a string and print the ASCII code of the first character. Have it continue until the string entered is empty ("").
2) Write a program that will ask for a number. Add this number to a running total. Repeat until the number entered is 0. Print the result.

Chapter 13 - Other Ways of Entering Information

GET and GET$

The problem with INPUT is that you have to enter the keys you want and then press return. If you are waiting for the user to enter information that the program can act on, like the radius of a circle, that's ok but there are times when the keyboard is used for other purposes. GET waits for a key to be pressed and returns the ASCII code for that key, without printing it back to the screen. You can use this to assign the key pressed to a variable, or just to wait for a key press after displaying a screenful of information.

```
REM GET demo
PRINT "Press any key to continue"
Key% = GET
PRINT "Code for key pressed: " ;Key%
END
```

GET takes account of cursor keys, function keys etc. The codes returned are given in the help files under GET / GET$, or you can keep the above program handy and run it when you need to find the code for Alt-F1 or whatever.

GET$ returns a single character string representing the key the user pressed. As with its twin above, a value is returned for each key pressed including function keys etc. These return characters above 127, so if you try to print them, you may get strange results. Try this little program and verify that it does return different values for F1, Shift+F1, Ctrl+F1 and Alt+F1.

```
REM GET$ demo
Key$=GET$
PRINT Key$,ASC (Key$)
END
```

Using GET$ and GET allows us to vet the input that the user is making so we can check that keys being pressed are valid. This gets around a problem with INPUT.

BASIC waits for return to be pressed and then tries to deal with the result, which often it can't. This is a useful little routine that only allows the user to enter numbers 0 to 9 and ignores other keys.

```
REM Number filter
Total$=""
PRINT "Type your number or Enter to finish"
REPEAT
  Key$=GET$
  IF Key$ >="0" AND Key$ <="9" THEN
    Total$=Total$+Key$
    PRINT TAB (0,1);Total$;
  ENDIF
UNTIL ASC (Key$)=13
PRINT '"You entered: " ;VAL (Total$)
END
```

INKEY and INKEY$

INPUT, INPUT LINE, GET and GET$ all wait for a key to be pressed before the program continues. We will now put two commands under the microscope that allow us to specify an amount of time we can wait before BASIC gives up and continues. The functions INKEY and INKEY$ can both take a number which specifies the amount of time to wait in centiseconds (just like our friend the WAIT command). If a key is pressed in this time, the function returns its value just as GET and GET$. INKEY returns the ASCII value and INKEY$ returns a single character string. The codes for the extended keys (function, cursor etc.) are just the same as GET and GET$, which is a good thing, really. If no key was pressed in the time, –1 or an empty string is returned.

```
REM INKEY demo
PRINT "Press a key or wait 5 seconds"
REM 500 = 5 seconds / 0.01
Key%=INKEY (500)
IF Key%<>-1 THEN
  PRINT "You pressed " ;Key%
ELSE
  PRINT "You pressed nothing"
ENDIF
END
```

If the time value is 0, the INKEY twins return immediately without waiting. Most useful in games where you want to have the aliens lurching across the screen without waiting for the player to move. This modest example counts the number of repeats of a loop until a key pressed.

```
REM INKEY$
Count% = 0
REPEAT
  WAIT 50 : REM Delay for 0.5 seconds
  Count% = Count%+1
UNTIL INKEY$ (0)<>""
PRINT "You waited " ;Count%/2;" seconds"
END
```

INKEY (not INKEY$) has an extended functionality in that it can be used to test for individual keys being pressed. To do this, we give it a negative argument. The table of values used to represent the required key is not related to the ASCII code and is given in full in the help files under INKEY / INKEY$. When used in this way, it returns immediately with a value of TRUE (–1) or FALSE (0) depending on whether the required key was pressed. Again, this is really good for games:

```
REM INKEY to detect individual keys
PRINT "Use left and right, space and x"
REPEAT
  pause% = INKEY (1)
  IF INKEY (-26) PRINT "Move left"
  IF INKEY (-122) PRINT "Move right"
  IF INKEY (-99) PRINT "FIRE!!!"
UNTIL INKEY (-67)
END
```

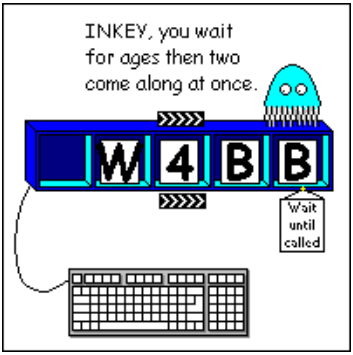
Note that pressing each key can generate more than one message as the program is testing each key many times a second, not just once every time the key is held down. You can also try holding the left and fire keys at the same time to see if this works.

The pause%=INKEY(1) line has two functions: it prevents the program 'hogging' the computer's processor (which can slow down other programs or cause a laptop's battery to run down more quickly) and it stops the keyboard buffer (see below) filling up if you press the keys many times. If the buffer fills, the computer will

complain by beeping.

The Keyboard Buffer

BASIC will accept key presses at any time, not just when you're looking for them with INPUT etc. To ensure that no keystrokes are missed, there is a little queue into which all key presses are inserted. When BASIC uses INKEY / GET$ etc. it doesn't inspect the keyboard directly, but pulls the first entry it finds in this keyboard queue or buffer, as it is generally known. If you press a key when the program is not expecting it, the key pressed is added to the buffer and sits there until the next GET or INKEY comes along and reads it. As a result of this, the key you get might not be the key you were expecting, depending on if there was any junk left over in the keyboard buffer.



It is a very useful thing to be able to clear the buffer at will before making a call to GET or INKEY. All we have to do is loop until INKEY returns nothing:

```
REPEAT UNTIL INKEY (0)=-1
```

As a demonstration, run this little program.

```
REM Keyboard buffer
PRINT "Press a key to continue"
A%=GET
PRINT "Ok, now press another, quickly"
WAIT 500
PRINT "Press a key to continue"
REM REPEAT UNTIL INKEY(0)=-1
A%=GET
PRINT "Program finished"
END
```

If you press a key twice before the 5 second delay has expired (line 5) the key is added to the buffer and sits there. The next call to GET in line 8 then reads this directly without waiting and the program ends. Take the REM out of line 7 and try again. Now the buffer is emptied before the second call, so all is well.

This might seem like a lot of fuss over something so small, but unless you are aware of it, it'll trip you up someday and you'll thank me for it. Honest.

Exercises

1) Modify the number filter program to accept one (and only one) decimal point in the number. If you're feeling ambitious, add a further modification to intercept backspace and delete the last character, if any, from the string. You'll need to print a space after the string to erase the character from the screen.
2) Make a simple drawing program. Use X% and Y% in a TAB statement to plot an 'X' on the screen. When you press the arrow keys, adjust X% or Y% according to the key pressed. For example, if you press left, decrease X%, down - increase Y%. Plot an 'X' at the new position so you can leave a trail on the screen. Restrict the area in which you can draw to a 20 * 20 grid.

Chapter 14 - Grouping Data in Arrays

For reasons that will become apparent, arrays are often used in conjunction with an earlier subject, loops. To illustrate the need for arrays, let's pretend you're working on a space invaders game. Somewhere near the end of the code, you need to see if there are any aliens left alive so you can start a new level. If each alien has a variable to say whether it's still alive, you could do this:

```
REM ...
NewLevel=TRUE : REM Assume new level
IF Alien1Alive THEN NewLevel=FALSE
IF Alien2Alive THEN NewLevel=FALSE
IF Alien3Alive THEN NewLevel=FALSE
IF NewLevel THEN
  REM setup new level
ENDIF
REM ...
```

The first line assumes a new level is required. The program then inspects each *AlienXAlive* flag in turn and if this alien is still alive, resets the *NewLevel* flag to false. If it completes its checks and the flag is still true, all the aliens must be dead so line 'em up and start again. Question: what happens if there are 40 aliens? Or 100? Using this method, that's a lot of lines of code. There must be a better way:

```
DIM AlienAlive(50)
REM ...
NewLevel=TRUE : REM Assume new level
FOR I%=1 TO 50
  IF AlienAlive(I%) THEN NewLevel=FALSE
NEXT I%
IF NewLevel THEN
  REM setup new level
ENDIF
REM ...
```

The first thing is the keyword DIM. DIM stands for DIMension and is an instruction to the computer to set aside an area of memory for an array variable. You tell BASIC how many variables the array holds in the brackets after the name. In this case, we are telling the computer we have 50 aliens, so reserve 50 locations, one for each. Previously, we have let BASIC define our variables when first used, but an array is different. By defining the size of an array, BASIC can perform a check each time it is used in code. If we try to access outside the array, BASIC will tell you by halting the program and producing a friendly message (i.e. it crashes). Although not compulsory, it is good practice to declare all arrays at the top of the program. This puts them all together in one place and gets all the memory allocations out of the way when the program starts up.

Once declared, we can use the array pretty much as we do any other variable, each individual element is accessed by specifying its number: *AlienAlive(1)* , *AlienAlive(39)* or as in the line IF *AlienAlive(I%)* ... using the counter in a FOR loop to decide which element we wish to inspect. By using this method, we can check as many aliens as we wish all in the same number of lines of code.

Here is a complete example which stores grades for a number of pupils and then does something with the data:

```
REM Student grade program
DIM Grade%(5)

REM First, collect the data
FOR I%=1 TO 5
  PRINT "Enter grade for student " ;I%;": " ;
  INPUT Grade%(I%)
NEXT I%

REM Work out the average
Total%=0
FOR I%=1 TO 5
  Total%=Total%+Grade%(I%)
NEXT I%
REM Print the average
PRINT "The average grade was " ;Total%/5

REM Find the minimum grade
```

```
Minimum%=999
FOR I%=1 TO 5
  IF Grade%(I%)<Minimum% THEN
    Minimum%=Grade%(I%)
  ENDIF
NEXT I%
REM Print the minimum
PRINT "The minimum grade was " ;Minimum%

END
```

Make room for 100 pupils if you want by changing the relevant lines, but you might get bored entering all that data!

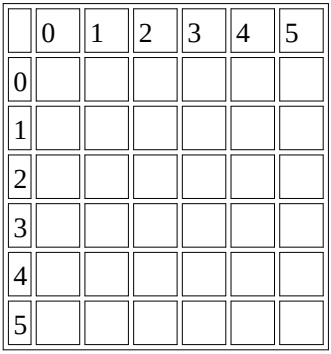
BBC BASIC actually gives you an extra element, so when you declare *Grade%(5)* you get 6 elements: *Grade%(0)* to *Grade%(5)* . In practice, it's often easier to ignore the first element because it's human nature to think in terms of pupil 1, pupil 2 etc, not pupil 0. But it's there if you want it. All the elements of an array are set to 0, or empty in case of strings, on declaration and once declared, you can't resize the array by re-declaring it later on. It may help to think of an array as a collection of boxes, so if we entered values for the above students of 40,80,60,70 and 55, this is how they would be stored:

Element	0	1	2	3	4	5
Value	0	40	80	60	70	55

You can declare arrays with more than one dimension. If you think about a grid (like the imaginary one you use to place characters on the screen) we would have a two dimensional array:

```
DIM Grid%(5,5)
```

This would define a grid 6 cells by 6 cells (remember element 0). This is how it would look in our box diagram:

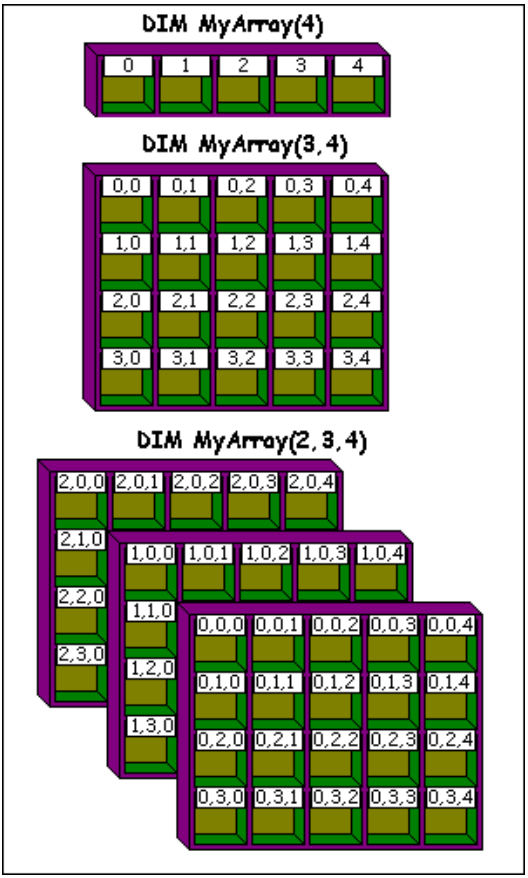


Each row has 6 columns as can be seen from above. These are accessed by *Grid%(0,0),Grid%(0,1),Grid%(0,2) ... Grid%(0,5)* then *Grid%(1,0),Grid%(1,1) ... Grid%(1,5)* etc. up to *Grid%(5,5)* . Into each of these locations, we can put a value, that's 36 locations in all. As you can see, this gives us a powerful way to group information.

You can define an array with 3 dimensions:

```
DIM Grid%(10,10,10)
```

If you think of a two dimensional grid as a page, we've just declared 11 pages, each containing a grid 11 elements square. If you're working with the demo version, arrays are a superb way to run out of memory. Think about it, there are 11*11*11 = 1331 elements in this array, so just be wary before getting too carried way. Realistically, unlike some other BASICs, with BB4W you can declare as many dimensions as you want, but in practice programmers rarely use more than 3.



Initialising Arrays

READ, DATA and RESTORE

Arrays can be used to hold data, but one of the problems is getting the data into the array. In the grade program above, entering is fine because every time you run it, you would probably want to enter different values. Consider this:

```
REM Days in a month
DIM Month%(12)

REM First, collect the data
FOR I%=1 TO 12
  PRINT "Enter days in month " ;I%;": " ;
  INPUT Month%(I%)
NEXT I%

REM Now ask for month
INPUT "Enter month number: " M%
PRINT "Month " ;M%;" has " ;Month%(M%);" days."

END
```

Sort of defeats the object really, doesn't it? BASIC, of course, has got there before us and provides a way to set up variables and arrays that are going to be the same each time.

```
REM Days in a month
DIM Month%(12)

REM First, collect the data
FOR I%=1 TO 12
  READ Month%(I%)
NEXT I%

REM Now ask for month
INPUT "Enter month number: " M%
PRINT "Month " ;M%;" has " ;Month%(M%);" days."

END
DATA 31,28,31,30,31,30,31,31,30,31,30,31
```

That's much better. There are a pair of keywords here, READ and DATA. DATA contains just that, a collection of data, numeric or string that can be used in the program. What the data is and the order you put it is entirely up to you. When the program encounters a READ statement, it goes off and finds the next DATA statement. It then reads the value back into the variable given, a little like INPUT.

BASIC remembers where it got up to, so the next time it encounters READ, it carries on from where it left off.

Obviously, there should be as many pieces of data as there are READ instructions (including the number of times READ is called in a loop) or the program gets upset. You are not constrained to using READ only with arrays, it is perfectly acceptable to set single variables using this method, but with a loop it is possible to initialise an array with very few lines of code.

DATA statements can be split in any way you choose, for example, we could have written:

```
DATA 31,28,31,30,31
DATA  30,31,31,30,31
DATA  30,31
```

or even:

```
DATA 31
DATA  28
DATA  31
DATA  30
DATA  31
DATA  30
DATA  31
DATA  31
DATA  30
DATA  31
DATA  30
DATA  31
```

You're the programmer, it's up to you, just make sure that there are as many items as there are READs.

You can mix and match items as long as you read the correct type into the correct variable.

```
REM Days in a month
DIM Month%(12), Name$(12)

REM First, collect the data
FOR I%=1 TO 12
  READ Month%(I%)
  READ Name$(I%)
NEXT I%

REM Now ask for month
INPUT "Enter month number: " M%
PRINT Name$(M%);" has " ;Month%(M%);" days."

END
DATA 31,January,28,February,31,March
DATA 30,April,31,May,30,June
DATA 31,July,31,August,30,September
DATA 31,October,30,November,31,December
```

DATA statements can be placed anywhere in the program, BASIC will just ignore them until told to use them. They are usually placed at the bottom of the program after END so they don't clutter the code, with the exception given below.

There are occasions when it is necessary to reread a set of data. Suppose you had a default set of values which could be reset from an option in a menu. To force the data pointer back to a specific place, we use the RESTORE command. RESTORE has a few options, the first has no argument and resets the data pointer to the first DATA statement in the program.

Another use is to specify a line number. (Line numbers are discussed in Appendix A . If you're not familiar with them, the general consensus these days is that you are not missing much but might like to zip off there and give them a glance.) Using RESTORE with a line number will reset the data pointer to the first item on the line given. This can be useful for specifying alternate sets of data.

```
 10 REM Months in English and French
 20 DIM Month$(12)
 30 REM First, collect the data
 40 INPUT "Do you want French? (y/n) " Ans$
 50 IF Ans$="N" OR Ans$="n" THEN
 60    RESTORE 170
 70  ELSE
 80    RESTORE 210
 90  ENDIF
100 FOR I%=1 TO 12
110    READ Month$(I%)
120 NEXT I%
130 REM Now ask for month
140 INPUT "Enter month number: " M%
150 PRINT "Month " ;M%;" is " ;Month$(M%)
160 END
170 DATA January,February,March
180 DATA April,May,June
190 DATA July,August,September
200 DATA October,November,December
210 DATA Janvier,Fevrier,Mars
220 DATA Avril,Mai,Juin
230 DATA Juillet,Aout,Septembre
240 DATA Octobre,Novembre,Decembre
```

The line number can be calculated if required.

To remove the line numbers, RESTORE gives us yet another option. In this we specify an offset from the line containing the RESTORE instruction (NOT the line with the first DATA statement). The number given tells BASIC the number of lines to move forward from the current position. To indicate that we are using an offset and not a line number, the number must be preceded by a +. Here is the above program without line numbers using this method. Examine the numbers in the RESTORE lines and count forwards to see where each one points to.

```
REM Months in English and French
DIM Month$(12)
REM First, collect the data
INPUT "Do you want French? (y/n) " Ans$
IF Ans$="N" OR Ans$="n" THEN
  RESTORE +11
ELSE
  RESTORE +13
ENDIF
FOR I%=1 TO 12
   READ Month$(I%)
NEXT I%
REM Now ask for month
INPUT "Enter month number: " M%
PRINT "Month " ;M%;" is " ;Month$(M%)
END
DATA January,February,March
DATA April,May,June
DATA July,August,September
DATA October,November,December
DATA Janvier,Fevrier,Mars
DATA Avril,Mai,Juin
DATA Juillet,Aout,Septembre
DATA Octobre,Novembre,Decembre
```

You can only go forwards here, try specifying a negative offset and BASIC will complain. Lines here are actual physical lines, including blank ones and REMs, not just those containing code. As can be seen, if the program was to be extended, we could easily lose track of the offsets and create chaos. Perhaps this method is best employed when the DATA is close to the RESTORE statements.

```
REM ...
IF Ans$="N" OR Ans$="n" THEN
  RESTORE +4
ELSE
  RESTORE +6
ENDIF
DATA January,February,March
DATA April,May,June
DATA July,August,September
DATA October,November,December
DATA Janvier,Fevrier,Mars
DATA Avril,Mai,Juin
DATA Juillet,Aout,Septembre
DATA Octobre,Novembre,Decembre
```

In any of the above methods, if there is no data at the line given by RESTORE, BASIC finds the next line that contains DATA and uses that instead.

Initializing without READ and DATA

It is possible to initialize an array directly in code. This saves DATA and READ statements and is more in keeping with the way languages like C do this sort of thing. The first example initializes all the elements after the array has been declared. Don't forget that *MyArray%(3)* has 4 elements 0 to 3 so we need 4 values.

```
REM Inline initialization
DIM MyArray%(3)

MyArray%() = 1,2,3,4
FOR I%=0 TO 3
  PRINT MyArray%(I%)
NEXT I%

END
```

This can be done at anytime, not just when the array has been declared.

```
REM Inline initialization
DIM MyArray%(3)

MyArray%() = 1,2,3,4
FOR I%=0 TO 3
  PRINT MyArray%(I%)
NEXT I%

MyArray%() = 5,6,7,8
FOR I%=0 TO 3
  PRINT MyArray%(I%)
NEXT I%

END
```

If you supply less than the number of elements, only the given number are initialized.

```
REM Initialize the first three elements
DIM MyArray%(3)

MyArray%() = 1,2,3
FOR I%=0 TO 3
  PRINT MyArray%(I%)
NEXT I%

END
```

However, and this is really useful, you can preset an entire array if only one value is given. Great for initializing big arrays.

```
REM Set each element to 50
DIM MyArray%(100)

MyArray%() = 55
FOR I%=0 TO 100
  PRINT MyArray%(I%)
NEXT I%

END
```

Multi-dimensional arrays are not a problem, as long as you get the order correct: right to left. In the example, the line has been split. This is not a problem, but you must still include a comma at the end of the split line, just as you would if it was continuous.

```
REM Initializing a multi-dimensional array
DIM MyArray%(2,3)

MyArray%() = 1,2,3,4,10,20,30, \
\ 40,100,200,300,400
FOR I%=0 TO 2
  FOR J%=0 TO 3
    PRINT MyArray%(I%,J%)
  NEXT J%
NEXT I%

END
```

Did somebody say strings? You can do these as well, but then, you'd expect that.

```
REM Initializing a string array
DIM Month$(12)

Month$() = "" ,"January" ,"February" , \
\ "March" ,"April" ,"May" ,"June" , \
\ "July" ,"August" ,"September" , \
\ "October" ,"November" ,"December"

FOR I%=1 TO 12
  PRINT Month$(I%)
NEXT I%

END
```

When do you use inline and not READ/DATA? It's completely up to you. One of the advantages of the inline method is that it gets round the dreaded line number dependency when using RESTORE. On the other hand, if you have large amounts of DATA, then you may prefer to keep it from clogging up the body of the code.

Finding the size of an array

The DIM statement can be used as a function. It can have one or two arguments passed to it. The first parameter is always the name of the array. When used with just the name, DIM will respond with the number of dimensions of the array.

```
REM Finding the size of an array
DIM Array1D(10)
DIM Array2D(10,9)
DIM Array3D(10,9,8)

PRINT "Array" ,"Dimensions"
PRINT "Array1D" ,DIM (Array1D())
PRINT "Array2D" ,DIM (Array2D())
PRINT "Array3D" ,DIM (Array3D())

END
```

As previously stated, it is possible to have an array name which is the same as a variable so we use the array name followed by the empty brackets to distinguish it from an ordinary variable.

Once you know the number of dimensions, it is possible to ask DIM for the size of each by passing the particular dimension number as the second parameter.

```
REM Finding the size of an array
DIM Array3D(10,9,8)
PRINT "Dimension 2 has " ; \
\       DIM (Array3D(),2);" elements."
END
```

To combine the above two, we can now find the number of dimensions and the size of each one.

```
REM Finding the size of an array
DIM Array3D(10,9,8)

N%=DIM (Array3D())
FOR I%=1 TO N%
  PRINT "Dimension:" ;N%;
  PRINT " Elements:" ;DIM (Array3D(),I%)
NEXT I%

END
```

We'll mention this use of DIM again in the section on procedures and functions which is coming up soon, whereupon you will be able to see why we would want to do this. Until then, just bear it in mind.

Exercise

Here are the figures for the first six months' sales of triple fruit chocolate covered syrup and treacle flavour ice lollies from one local newsagent:

```
January     105
February    261
March       482
April       195
May         347
June        626
```

Set an array to hold these values. Then add to the program so it loops through the values to find and print:
a) the month number for the lowest sales;
b) the month for the highest sales;
c) the total sales.
You can use the same FOR loop to achieve all three or do them separately: your choice.

Modify the program to have an array of month names, initialize it and adapt the above program to display real names for the months.

Chapter 15 - Grouping Data in Structures

Arrays are a great way to organize data. In your programs you will come across numerous uses for them. Eventually, you may happen upon a situation like the following. Suppose you are writing a killer game. The player has several variables associated with him: x-coordinate, y-coordinate, number of lives and name. You could easily define several variables like this:

```
PlayerX%=5
PlayerY%=12
PlayerLives%=3
PlayerName$="GR8"
```

Or you could use an array for the first three variables and make a mental note that index 1 refers to x position, 2 to y position and 3 to lives. *PlayerName* is a problem and would have to remain an individual string.

BBC BASIC offers the ability to create your own 'super' variable which will group all these individual variables together in one place and hence make them easier to keep track of. This device is called a structure as you probably guessed from the title of this section. To declare a structure, you use DIM, like an array but with curly brackets (braces) instead of the normal round ones so BASIC knows that we're talking about a structure. Within the braces go the names of the variables that we wish to include in that group. Usual naming rules apply. Let's see an example:

```
DIM Player{X%,Y%,Lives%,Name$}
```

We can access and use any of the variables and do all the normal things with them. The structure only acts as a grouping mechanism. Access is done by using the name of structure (*Player* in our case) followed by a dot and the name of the variable we want. To change the number of lives when a player gets hit, we would do this:

```
Player.Lives%=Player.Lives%-1
IF Player.Lives%<=0 THEN PRINT "GAME OVER"
```

The official jargon here is:

```
StructureName.MemberName
```

So now you can declare nice little units to hold all your variables in logical groups. But there's more. You knew I was going to say that, didn't you? The members of a structure are not limited to single variables. You can have an array as a member of a structure. Suppose your player could have up to 10 items of equipment: underwater flame thrower, anti-gravity boots etc. You need a way to represent whether he's picked up each of these items. Let's have an array called *Item%* . This is how we would declare the structure now:

```
DIM Player{X%,Y%,Lives%,Name$, Item%(10)}
```

Access is just as before:

```
IF Player.Item%(5)=TRUE THEN
  PRINT "You have the 1000 league boots"
ENDIF
```

Similarly, you can have another structure as a member of the structure. For example, it is common practice (particularly in Windows programs) to make a structure to hold the X and Y coordinates. You do it like this:

```
DIM Player{Posn{X%,Y%},Lives%, \
\          Name$, Item%(10)}
```

Posn (for position) is referred to as a substructure and accesses to its components are like this:

```
Player.Posn.X% = Player.Posn.X% + 5
```

Substructures can contain other substructures, arrays and variables in any combination. The idea here, of course, is to divide the data up into logical, easy to manipulate groups, not to show how clever you are by nesting 10 levels of data inside 10 levels of data.

Arrays of Structures

When our player has reached the end of the game, victorious and covered in gore, we would like to add a high score table so he can brag to his friends. The table will consist of the top 10 scores. By now, it should be apparent how this is done:

```
DIM Table{Name$,Score%}
```

Great, but that only takes care of the top position, we want a table. This is how we declare an array of structures:

```
DIM Table{(10) Name$,Score%}
```

The first item in the variable list is the number of elements in round brackets (parentheses). This will give us an array called *Table* with 11 structures 0 to 10. To print out the contents of the high score table, we access the array like this:

```
FOR I% = 1 TO 10
  PRINT I%,Table{(I%)}.Name$;
  PRINT Table{(I%)}.Score%
NEXT I%
```

You need to wrap the array index in braces to tell BASIC you're working with a structure. As with normal arrays, multi-dimensional arrays are allowed and the same rule about numbering from element zero applies.

Prototype Structures

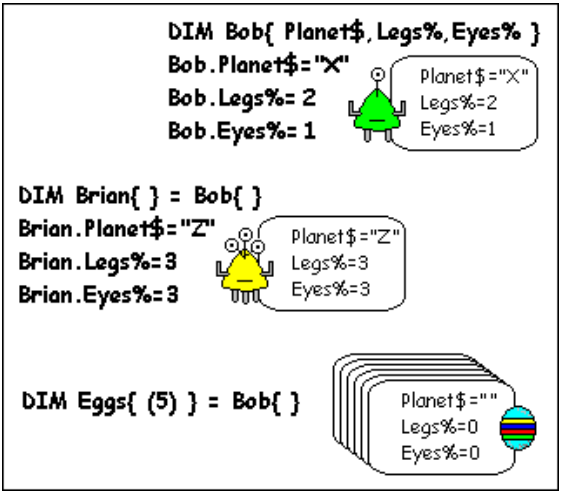
Back in our hypothetical game we have a structure to hold the aliens: current position, health points etc. At the end of the level there is the tough-as-old-boots-end-of-level boss that you have to defeat before winning through to the next level. This chap has the same attributes as an ordinary alien, but it doesn't really make sense to include him with the rank and file. We need to create another structure for him which has the same attributes as his underlings, but is kept separate so we can deal with him as a unique case. BBC BASIC allows us to use an already defined structure as the template or prototype for another one.

```
REM Prototype structures
DIM Alien{XPosn%, YPosn%, Health%}

REM Setup our alien
Alien.XPosn%=50
Alien.YPosn%=627
Alien.Health%=100

DIM BigBadBoss{}=Alien{}
REM Let's play ...
```

BigBadBoss will now have the members:

```
BigBadBoss.XPosn%
BigBadBoss.YPosn%
BigBadBoss.Health%
```

Don't get confused here. At the point when the structure is declared, all the members of *BigBadBoss* will be set to zero, it is only the names of the structure members that are copied over, not their actual values.



Using this technique, it is also possible to declare an array of the specified structure.

```
REM Arrays of prototype structures
DIM Alien{XPosn%, YPosn%, Health%}

REM Declare a full level of aliens
DIM LotsOfAliens{(20)}=Alien{}
LotsOfAliens{(0)}.XPosn%=50
LotsOfAliens{(0)}.YPosn%=627
LotsOfAliens{(0)}.Health%=100

REM ...
```

Line 5 gives us 21 (0 to 20) of the little blighters ripe for decimation.

In one of the previous sections, we had a player structure which had a position structure nested within it. This technique can be applied to prototypes as well. As an example, we can define the position of our alien using *XPosn%* and *YPosn%* , but we could make this a structure too. That way each time we had something that needed a position, the prototype would give it to us ready-made. This is useful, because it means we are always able to access the position in the same manner, wherever it is found. This is how we would do that:

```
REM Nested prototype structures
DIM Position{X%,Y%}
DIM Alien{Posn{}=Position{}, Health%}
DIM Player{Posn{}=Position{}, \
\          Lives%, Name$, Item%(10)}

REM Initialize our alien
Alien.Posn.X%=50
Alien.Posn.Y%=627
Alien.Health%=100

REM Initialize our player
Player.Posn.X%=50
Player.Posn.Y%=627
Player.Lives%=100

REM ...
```

Any changes made to the *Position* structure after the initial coding would instantly be reflected in the changes to the structures that use *Position* as a prototype. This is good and bad. If the game was to be expanded into three dimensions, we would redeclare like this:

```
DIM Position{X%,Y%,Z%}
```

and presto, both Alien and Player now have another dimension to play in. (You still have to write the code to handle that yourself. BB4W isn't THAT clever.) The point to be wary of is going the other way, if you remove an element, you must also remove every reference to it in every structure that contains it. Think about the layout of your data before implementing it, this way you save hours of frustration and rework.

Obviously, these techniques come into their own when using larger structures, but don't ever lose sight of the fact that they are merely ways of collecting similar data together so it's all accessible from the same place. Once you get the hang of structures, you'll wonder how you ever managed without them.

Tip: Structure names and abbreviated keywords

BBC BASIC allows the use of abbreviated names for most keywords and commands. These can be found under the entries for each word in the help files. As an example PRINT can be shortened to 'P.' - see the problem? If you declare a structure called P, when you try and access its members, BASIC will expand a line like 'P.Int1%=2' into 'PRINTInt1%=2' which is definitely not what you wanted. Single letter names don't fall into our book of good programming practices anyway.

The easiest way around this problem is to use meaningful names that have mixed case. It is possible to turn the abbreviated names on and off: from the menu choose *Options | Customize* then untick the box in the Program Editor group. The problem with this is that the option is on by default, so if you give your code to anyone else, they would have a problem and, because they are not as clever as you, wouldn't know what had happened.

If you read the help files, it's possible to do wonders with structures, but this introduction will suffice for many a while to come.

Exercises

1) If you wanted to store a pupil's grades for five subjects along with their first name and surname, can you suggest a structure that would do this?
2) Write a program that declares such a structure and prompts for the information. Calculate the average grade and print the results.
3) If you had 20 pupils in a class, how would you make an array of the above structure?

Chapter 16 - User Defined Routines: PROC

We have already seen there are built-in commands and functions that are the backbone of BASIC. Such examples are CLS, which clears the screen and LOG(X) which returns the log of X. Sooner or later, as programs get bigger, you'll find that you are using the same lines of code several times in a program. This section shows how we can make our own commands out of these common lines and then call them up at will.

BBC BASIC provides two methods of doing this: PROC and FN. Both allow you to call a routine in the same way you use in-built BASIC commands. The difference is that PROC is used as a command (no return value) whereas FN behaves like a function, it returns a value that you can assign or use in an expression. Largely, with these differences aside, the structure and rules are the same so we'll deal with PROC first, but bear in mind that much of what is said will apply to FN as well.

Suppose you want to clear the screen and print a title at the top of the page. That's easy, you say, two lines of code:

```
CLS
PRINT TAB (2);"BBC BASIC Tutorial"
```

Now, if we wish to do this several times, we have the choice of copying the lines each time or we can enclose them in a block of code that can be called up whenever we want. This is how we do it:

```
REM PROC demo
PROC _ScreenSetup
INPUT A$
PROC _ScreenSetup
END

DEF PROC _ScreenSetup
CLS
PRINT TAB (2);"BBC BASIC Tutorial"
ENDPROC
```

PROC stands for procedure, by the way. Before a PROC can be called, it must be defined. The start of the definition is denoted by the keyword DEF (for DEFine). The end is denoted by ENDPROC (no space) and all the code between is referred to as the body of the procedure. When the program runs, it gets to line 2 and makes a note of where it was. It then jumps off and runs the body of the PROC. When it finds ENDPROC, it goes back to where it left off and continues the program as before. A similar thing happens when it reaches line 4.

It is common practice to place PROC definitions after the END of the main program, so the main body of the code is not interspersed with declarations. The name of the PROC follows the conventions for naming variables: start with a letter or an underscore (you can actually use a number as well, unlike variables) then include any alphanumeric characters. The first letter of the name must follow the word PROC, no spaces. It is common practice, though not compulsory to start with an underscore, just to make things a little easier to read.

So now we can print our title on the screen, what happens if we have different titles depending on the screen we wish to display? Well, we could write a different PROC for each screen, but if we have 10 screens, that's a lot of duplicate code. BB4W allows us to pass a value into a PROC for that PROC to use in any way it sees fit, so we can pass a different title as a text string each time we call the PROC.

```
REM Passing a value to a PROC
PROC _ScreenSetup("First screen" )
INPUT A$
PROC _ScreenSetup("Second screen" )
END

DEF PROC _ScreenSetup(Title$)
CLS
PRINT TAB (2);Title$
ENDPROC
```

How useful is that? The text for the title is copied into the string variable *Title$* and this is used throughout the body of the routine, just as it would a normal variable. Note that the contents of *Title$* are only meaningful whilst the PROC is being processed.

A PROC can have as many values passed to it as you wish, the rule is that they must be of the same type and in the same order when called as in the line in which they are declared. For example, here is a variation that allows us to specify the title and where to print it on the top line.

```
REM Passing a value to a PROC
PROC _ScreenSetup(5, "First screen" )
INPUT A$
PROC _ScreenSetup(10, "Second screen" )
END

DEF PROC _ScreenSetup(Col%,Title$)
CLS
PRINT TAB (Col%);Title$
ENDPROC
```

When a variable is passed to a PROC, the procedure is free to manipulate the local value in any way it wants. As it is working with a copy, the original variable is not changed. This method of passing data is termed 'call by value'.

```
REM Call by value demo
MyString$ = "Hello, world"
PRINT "Value before PROC " ;MyString$
PROC _DoSomething(MyString$)
PRINT "Value after PROC " ;MyString$
END

DEF PROC _DoSomething(AString$)
PRINT "Value passed to PROC " ;AString$
AString$="Goodbye, cruel world"
PRINT "Value after changing " ;AString$
ENDPROC
```

There are occasions when we want the routine to do something permanent to the contents of a variable. To do this we use the keyword RETURN in the declaration of the PROC. Change the declaration line and run again to see the effect:

```
DEF PROC _DoSomething(RETURN AString$)
```

This is termed 'call by reference'.

You can pass entire arrays and structures into a PROC, not just single value variables. To do this, just use the name followed by empty brackets to indicate that it is not a plain old variable that we are dealing with. If you remember the use of DIM to find the size of an array in the section on arrays, you can now see an immediate use for it. If an array is passed, we can use it to check that it has the correct number of dimensions of the right size. This helps eliminate out of bounds errors.

```
REM Passing an array
DIM IntArray%(2,3)
PROC _ArraySize(IntArray%())
END

DEF PROC _ArraySize(Array%())
PRINT "This array has " ;
PRINT DIM (Array%());" dimensions."
ENDPROC
```

Structures have no corresponding function, you just have to know the members associated with that structure.

```
REM Passing a structure
DIM Struct{A%,B$}
Struct.A%=23
Struct.B$="Twenty-three"
PROC _PrintStruct(Struct{})
END

DEF PROC _PrintStruct(St{})
PRINT St.A%
PRINT St.B$
ENDPROC
```

Arrays and structures are always passed by reference. The reason is that an array or structure can be (and frequently is) big, that's why we use them. To make a complete copy would be expensive, both in terms of memory and the processing time it takes to physically copy each value. This means that if you change a value in a passed array or structure, the change is permanent.

```
REM Passing a structure
DIM Struct{A%,B$}
Struct.A%=23
Struct.B$="Twenty-three"
PROC _PrintStruct(Struct{})
PROC _IncStruct(Struct{})
PROC _PrintStruct(Struct{})
END

DEF PROC _PrintStruct(St{})
PRINT St.A%
PRINT St.B$
ENDPROC

DEF PROC _IncStruct(St{})
St.A%=24
St.B$="Twenty-four"
ENDPROC
```

LOCAL Variables

The idea that we can pass values into a PROC is an important one, it means that we

can write routines which are portable between programs, so once you've tested a routine, you can paste it into your next program, put it in a library or give it to all your BBC BASIC friends without redeveloping it. In order to do this most effectively, we need to ensure that every variable used in the PROC is only used in the PROC and nowhere else. Values passed from outside the procedure have already been dealt with. What we need is a method of making a variable belong solely to a routine. It's about now that we run into the idea of local variables.

A local variable is a variable that exists whilst the body of the procedure is being executed and is only accessible to that procedure. There is a keyword necessary to let BASIC know that the variable is to be treated as local and it is called LOCAL. A variable defined as local comes into existence when the routine is called, can be used anywhere in the body of the code and is discarded when the routine exits. Let's see an example, suppose we want to draw a line of characters on our PROC:

```
REM LOCAL demo 1
PROC _DrawLine("*" )
END

DEF PROC _DrawLine(Char$)
LOCAL Count%
FOR Count% = 0 TO 79
  PRINT Char$;
NEXT Count%
ENDPROC
```

Local variables follow conventions of normal (global) variables as regards naming and defining types.

You may also declare arrays and structures as LOCALs but to do this requires the use of an extra DIM statement:

```
REM LOCAL array
PROC _A
END

DEF PROC _A
LOCAL MyArray%()
DIM MyArray%(10)
REM Do something ...
ENDPROC
```

The LOCAL statement here tells BASIC it will need to reserve some space for an array, the next line with the DIM tells it how much. LOCAL structures are achieved in the same way:

```
REM LOCAL structure
PROC _A
END

DEF PROC _A
LOCAL MyStruct{}
DIM MyStruct{a%,b,c$}
REM Do something ...
ENDPROC
```

PRIVATE Variables

Once you are familiar with LOCAL variables, PRIVATE ones are easy. A PRIVATE variable is declared in the same way as a LOCAL variable, but uses the keyword PRIVATE to define it. It is valid during the execution of the routine but not outside it, just like a LOCAL. The difference is a PRIVATE variable is not forgotten about when a procedure is finished, its value is remembered and re-used the next time the routine is called.

```
REM PRIVATE variable
PRINT "Calling first time"
PROC _A
PRINT "Calling second time"
PROC _A
END

DEF PROC _A
PRIVATE MyInt%
PRINT "Value of MyInt% = " ;MyInt%
MyInt%=MyInt% + 100
ENDPROC
```

First time round, *MyInt%* is created and set to zero, the value is printed out for all to see. *MyInt%* is then increased by 100. The second time PROC_A is called, *MyInt%* has a value from the previous call, this is shown by the PROC printing the value again.

Scope

Up to encountering the use of local and private variables, all variables have been accessible anywhere in the program, they are known as 'global' variables. Global variables can be accessed anywhere in the program, even after we have finished chopping it up into manageable sections with PROC and FN. If you declare a variable in a routine without first telling BASIC it will be local or private, this variable will be global just like all the rest. Programmers (of which you are one) refer to this 'visibility of a variable' as scope, probably because it's easier to say. When a variable is not visible, it is said to be 'out of scope'.

Exercises

1) Modify the PROC_ScreenSetup to accept the background and foreground colour.
2) Write and test PROC_Greater(A%,B%) which compares *A%* and *B%* . If the *A% > B%* , do nothing, if *B% > A%* , exchange the two values using a local variable. You'll need to pass by reference so the calling program can print the results.

Chapter 17 - User Defined Functions: FN

You will be pleased to know that all the information about PROCs covered in the previous section also applies to user defined functions - FN. The only difference is that FNs return a value to the calling line, hence, like the inbuilt functions such as SQR, they can be used in expressions.

```
REM Squaring a number using FN
A=4
B=FN _Square(A)
PRINT A;" squared is " ;B
END

DEF FN _Square(Num)
=Num^2
```

We'll look at the definition first. As can be seen, we tell BASIC that we're declaring a function by using FN instead of PROC. There's no ENDFN. The end of the function is indicated by the line starting with '='. The expression is calculated and this value is returned to the caller, line 3 in our case.

In some other dialects of BASIC, you have to include the return type in the function's name, like when you declare a variable. BBC BASIC allows you to do this, but it doesn't check the return type against the type specified in name. Usually, you don't bother.

Like PROCs, FNs can be as long and complex as you choose to make them. They can call other PROCs and FNs. The one rule I do suggest you stick to, with both types of routine, is only have one exit point. Having multiple = lines, depending on conditions might seem like a good idea, but does lead to trouble. It is much cleaner to use a local variable to hold the result and then return this at the end:

```
REM Vowel test
INPUT "Enter a letter: " Char$
IF FN _IsAVowel(Char$) THEN
  PRINT Char$;" is a vowel"
ELSE
  PRINT Char$;" is not a vowel"
ENDIF
END

DEF FN _IsAVowel(Ch$)
LOCAL Result%
IF INSTR ("AEIOUaeiou" , LEFT$ (Ch$,1)) THEN
  Result%=TRUE
ELSE
  Result%=FALSE
ENDIF
=Result%
```

Rather than this, where if more code is added each return point can get lost in the rest of the code:

```
REM ...
DEF FN _IsAVowel(Ch$)
IF INSTR ("AEIOUaeiou" , LEFT$ (Ch$,1)) THEN
  =TRUE
ELSE
  =FALSE
ENDIF
```

You can only pass single values back, not arrays or structures. (There is a method of doing this, but that definitely falls into the domain of advanced.) If you need to pass a value back, it's better to pass the whole thing as a parameter and manipulate it in the function.

Tip: Find that routine

In any but the most trivial program you will have several PROCs and FNs. To locate the definition of the routine you are trying to find, right click anywhere in the editor and as if by magic, all the PROCs and FNs contained in the currently open program are listed at the bottom of the popup menu. To jump to one, point and click.

Tip: Routines with the same name

It should be pretty obvious that you don't have two PROCs or FNs with the same name, however BB4W won't object if you do. If BASIC finds two routines with the same name, it will use the last one and ignore all others. With a little program, that's easy to spot. When the programs grow to more than one page, it becomes difficult to detect and if you are using libraries, which allow you to split your code over several files, a major headache. If the code you're editing doesn't seem to be having any effect, look for duplicate names.

That's all there is. As stated, all the things about LOCAL, PRIVATE, variable scope etc. are just the same as with PROCs.

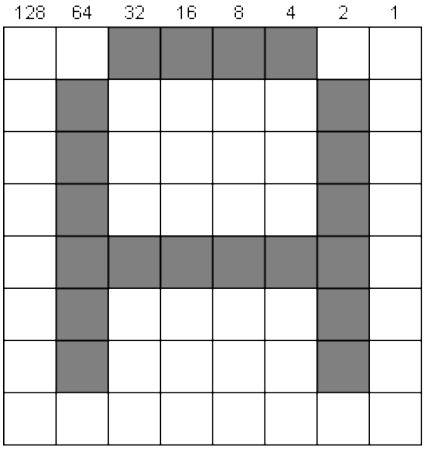
Exercise

1) It's a very useful function that waits for the user to press y (yes) or n (no) in response to a prompt and returns either TRUE or FALSE. It should, of course, check for case.
2) Write a function FN_Lower that accepts a string. It goes through each character in the string and converts all uppercase letters to lowercase. Other characters are left as they are. Return the converted string.

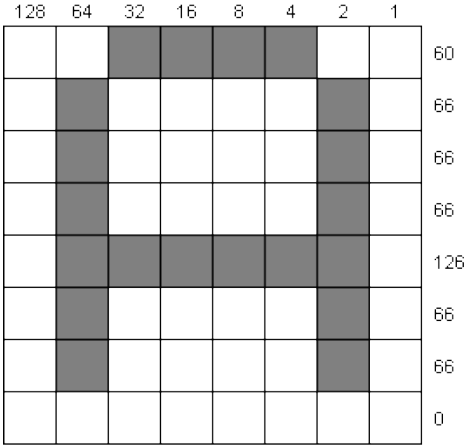
Chapter 18 - User Defined Characters

BBC BASIC allows you to make your own graphic characters and this section shows you how. It assumes you know a little about binary numbers. If you don't, there's an appendix you can go and read. Return here when you're comfortable with base 2.

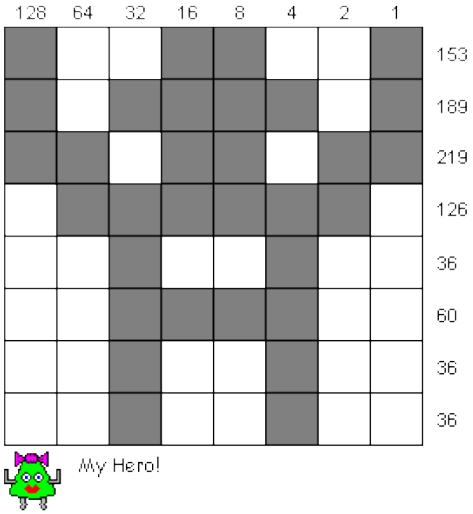
Every character is represented by a grid of 8 by 8 bits. This is how we would represent a letter 'A'.



This does depend on which character set or font you are using, but I'm sure you get the idea. By treating each column as a bit in an 8-bit byte, we get the following representation by adding the value of each column that is set. Row 0 is 32+16+8+4=60.



By redefining these numbers, we can alter which bits are set in which rows and hence draw our own characters. We could make our very own alien like this:



BBC BASIC allows us to redefine the characters in the range 32 to 255. Usually, it's better to stick to ones that don't interfere with the ordinary characters below 128 and most programs start around 200. To redefine a character, we use a VDU command. There are dozens of these commands, have a look in the help to see what they can do for you. The one we are concerned with here is number 23. This allows us to redefine a character (amongst other things). We do it like this:

```
VDU 23,CharNumber,Row0Total, ... ,Row7Total
```

Applying to our alien, we get:

```
VDU 23,240,153,189,219,126,36,60,36,36
```

Let's try this in a program:

```
REM Making aliens
MODE 6
PRINT "Before adjustment " ;CHR$ (240)
VDU 23,240,153,189,219,126,36,60,36,36
PRINT "After adjustment " ;CHR$ (240)
END
```

In the default mode, the new character comes out very small, so I've used MODE 6 which makes the characters larger so we can see them better.

Obviously, you can apply different colours to our alien, because it is just another text character as far as BASIC is concerned. Now we've got our character, we can make a simple animation sequence. By printing the character, erasing it by printing space and reprinting in a different position, we can make our alien walk across the screen.

```
REM Walking alien
MODE 6
VDU 23,240,153,189,219,126,36,60,36,36
PRINT TAB (0,10);CHR$ (240)
FOR I%=1 TO 19
  PRINT TAB (I%-1,10);" "
  PRINT TAB (I%,10);CHR$ (240)
  WAIT 25
NEXT I%
END
```

The WAIT instruction is a small delay, adjust to taste, to prevent the whole thing being rendered in Flic-o-vision.

User defined characters have endless uses, not just in games, but for lots of general purpose programs where you want a character that is not available in the standard character set.

The cursor can be a distraction at times, so it's nice to be able to turn it off and on at will. Inspection of the help for VDU 23 reveals that it can do this for us if we call it like this:

```
VDU 23,1,0;0;0;0; : REM Disable cursor
VDU 23,1,1;0;0;0; : REM Enable cursor
```

However, because this such a common requirement, BBC BASIC allows using OFF to switch off the cursor and ON to put it back again.

We can apply this to our walking alien program.

```
REM Walking alien
MODE 6
OFF
VDU 23,240,153,189,219,126,36,60,36,36
PRINT TAB (0,10);CHR$ (240)
FOR I%=1 TO 19
  PRINT TAB (I%-1,10);" "
  PRINT TAB (I%,10);CHR$ (240)
  WAIT 25
NEXT I%
ON
END
```

Wasn't that better? As with the text colours, it's usually considered good manners to restore the cursor before we leave the program.

Tip: Larger user defined characters

8 * 8 not big enough for you? There's nothing to stop you defining several characters, each containing a segment of the overall picture them building a string with the larger picture in it like this:

```
REM Composite Characters
MODE 6
OFF
VDU 23,240,58,69,130,137,149,73,34,28
VDU 23,241,92,162,65,145,169,146,68,56
Butterfly$=CHR$ (240)+CHR$ (241)
REM Print 20 butterflies at random positions
FOR I%= 1 TO 20
  PRINT TAB (RND (30+5),RND (15+5));Butterfly$
  WAIT 25
NEXT I%
ON
END
```

Exercises

1) Make the alien walk back across the screen, right to left when it has reached the right-hand side.
2) Create another alien with its arms pointing down using character 241. Modify the animation from 1) to alternate aliens as it moves across the screen.

```
I% MOD 2
```

will tell you if *I%* is an odd or even number.

Chapter 19 - Games and SOUND

Now we've covered user-defined graphics, the next logical step would be to combine them and make a little game. After all, what's the point in aliens if you can't blast them to bits? Actually, my effort is a little more friendly: you are in charge of an ore collection vessel flying through an asteroid belt in deepest space (i.e. a black background!). You have to pick up the meteors but avoid the aliens. We'll develop this in steps and introduce some new techniques as we go. First we need some characters to play with. Here are mine. We can define them then wrap them in a string with the chosen colour.

```
REM Meteor run
VDU  23,220,129,219,255,255,126,60,60,24
VDU  23,221,24,60,118,221,251,110,60,24
VDU  23,222,153,189,219,126,36,60,36,36
REM Build characters
Ship$=CHR$ (17)+CHR$ (3)+CHR$ (220)
Rock$=CHR$ (17)+CHR$ (4)+CHR$ (221)
Alien$=CHR$ (17)+CHR$ (2)+CHR$ (222)

REM Print to see how they look
MODE  6
PRINT  Ship$
PRINT  Rock$
PRINT  Alien$

END
```

Once we've printed the characters, we need a way to make them move up the screen. Fortunately this is easy. By printing on the bottom line, we can cause the whole screen to scroll up, which is a cheap and cheerful way of achieving animation. On each loop we print a meteor and an alien in a random column, by printing a semicolon after the alien we suppress the line feed. Press ESC to finish.

```
REM Meteor run
VDU  23,220,129,219,255,255,126,60,60,24
VDU  23,221,24,60,118,221,251,110,60,24
VDU  23,222,153,189,219,126,36,60,36,36
REM Build characters
Ship$=CHR$ (17)+CHR$ (3)+CHR$ (220)
Rock$=CHR$ (17)+CHR$ (4)+CHR$ (221)
Alien$=CHR$ (17)+CHR$ (2)+CHR$ (222)

MODE  6

REPEAT
  WAIT  15
  REM Print at bottom of screen
  PRINT  TAB (RND (40),24);Alien$;
  PRINT  TAB (RND (40),24);Rock$
UNTIL  FALSE

END
```

We need to add the ship. As the screen is scrolling up it must be reprinted each time on the top line. We can use the cursor keys to move it left and right but need a variable to record the position. Let's switch the cursor off too so it doesn't get in the way.

```
REM Meteor run
VDU  23,220,129,219,255,255,126,60,60,24
VDU  23,221,24,60,118,221,251,110,60,24
VDU  23,222,153,189,219,126,36,60,36,36
REM Build characters
Ship$=CHR$ (17)+CHR$ (3)+CHR$ (220)
Rock$=CHR$ (17)+CHR$ (4)+CHR$ (221)
Alien$=CHR$ (17)+CHR$ (2)+CHR$ (222)

MODE  6
X%=20
OFF
REPEAT
  WAIT  15
```

```
    REM Print at bottom of screen
    PRINT  TAB (RND (40),24);Alien$;
    PRINT  TAB (RND (40),24);Rock$
    REM Reprint ship
    PRINT  TAB (X%,0);Ship$
    REM Detect ship movement
    IF  INKEY (-26) AND  X%>0 X%-=1
    IF  INKEY (-122) AND  X%<39 X%+=1
  UNTIL  FALSE
  ON
  END
```

Getting there, but there's no indication of when we hit a rock or a baddie. We need to be able to detect the character directly in front of our ship so we can award points or subtract a life as the case may be. When a new object is created, we could remember the co-ordinates and update them on each loop, but that seems like a lot of work just to test that one little square in front of the spaceship. Fortunately, BBC BASIC is capable of reading the character back from a given screen location. Even better, you already know the keyword involved: GET. When supplied with a pair of co-ordinates, GET will return the ASCII code of the character at the given location. This enables us to make our game truly interactive. First we read the screen then test the character returned to see if it is an alien or a meteor. As well as making the game pause for a short while so the player knows something has happened, we can award points and remove lives as the occasion demands. Our program now looks like this:

```
REM Meteor run
VDU  23,220,129,219,255,255,126,60,60,24
VDU  23,221,24,60,118,221,251,110,60,24
VDU  23,222,153,189,219,126,36,60,36,36
REM Build characters
Ship$=CHR$ (17)+CHR$ (3)+CHR$ (220)
Rock$=CHR$ (17)+CHR$ (4)+CHR$ (221)
Alien$=CHR$ (17)+CHR$ (2)+CHR$ (222)

MODE  6
X%=20
Score%=0
Lives%=3
OFF
```

```
REPEAT
  WAIT  15
  REM Print at bottom of screen
  PRINT  TAB (RND (40),24);Alien$;
  PRINT  TAB (RND (40),24);Rock$
  REM Reprint ship
  PRINT  TAB (X%,0);Ship$
  REM Read character in front of player
  Ch%=GET (X%,1)
  IF  Ch%=222 THEN
    Lives%-=1
    WAIT  20
  ENDIF
  IF  Ch%=221 THEN
    Score%+=10
    WAIT  20
  ENDIF
  REM Detect ship movement
  IF  INKEY (-26) AND  X%>0 X%-=1
  IF  INKEY (-122) AND  X%<39 X%+=1
UNTIL  Lives%=0
CLS
COLOUR  7
PRINT  TAB (13,12);"You scored " ;Score%
ON
END
```

There's not much more to say about GET when used like this. Keep the co-ordinates in the range of the MODE you are using, remembering that the top left corner is 0, 0 and all will be well.

REFRESHing the screen

Depending on the speed of your computer, you may notice that the scroll / update cycle in our game causes an irritating flicking of the characters, especially the player's ship. This flicker can be viewed as a minor distraction or a major annoyance, depending on how generous you feel. BB4W has a rather neat way of eliminating it. When we write to the screen normally, BASIC and Windows take care of actually displaying the pixels we want. This is well and good most of the

time but there are occasions when we require a little more control. BB4W has a set of commands that relate to the operating system in the same way that the VDU commands affect the display on the screen. All these commands start with an asterisk (*) and if you look under 'Operating System Interface' in help, you'll see them listed there. They are generally referred to as star commands for reasons that I'll let you guess. We'll concentrate on *REFRESH. Using *REFRESH we can actually tell BASIC when we want to update the screen. First we can draw all our characters and, when finished, say 'Right here's the complete screen, display it now'. To do this we call *REFRESH OFF which suppresses the automatic update and then call *REFRESH when we want to show our finished screen. Here's a simple example:

```
REM Using *REFRESH
*REFRESH OFF
REM Write some text
PRINT  "Hello, world"
REM Wait for key press
Dummy$=GET$
REM Display screen
*REFRESH

*REFRESH ON
END
```

The star commands are not recognised as keywords so the syntax colouring doesn't pick them up. Although the text is printed to the screen, it is not actually displayed until the *REFRESH command is reached. The *REFRESH ON just restores the default behaviour when we no longer require the facility. You only need to call *REFRESH OFF once. After that call *REFRESH each time you need to update the screen.

```
REM Using *REFRESH
*REFRESH OFF
REM Write some text
PRINT  "Hello, world"
REM Wait for key press
Dummy$=GET$
REM Display screen
*REFRESH

CLS
Dummy$=GET$
*REFRESH

PRINT  "Back again"
Dummy$=GET$
*REFRESH

*REFRESH ON
END
```

Applying all this to our little game, I'm sure you can see that we need to do the scroll and redraw of the ship before the showing the screen again. Add three lines so the result looks like this:

```
REM ...
OFF
*REFRESH OFF
REPEAT
  WAIT  15
  REM Print at bottom of screen
  PRINT  TAB (RND (40),24);Alien$;
  PRINT  TAB (RND (40),24);Rock$
  REM Reprint ship
  PRINT  TAB (X%,0);Ship$
  *REFRESH
  REM rest of code ...
UNTIL  Lives%=0
*REFRESH ON
CLS
```

```
COLOUR  7
REM ...
```

Try these lines in the game and notice that the whole affair is much easier on the eye. Don't forget the final *REFRESH ON after the main loop or you won't get to see your score. When refresh is disabled, the display sometimes leaves ghost cursors scattered around the screen. If this is important to your application, disable the cursor using OFF.

Introducing SOUND

I wasn't aiming for Half-Life here and what we've got isn't bad for a few lines of code but our little game still lacks something. One way we can instantly upgrade our epic is by adding some sound. The obvious places would be a small explosion when we hit an alien and perhaps a little tune when we pick up a meteor. BBC BASIC is a pretty unique programming language in that it has a command that will allow us to make simple noises without having to distribute WAV files that are 50 times larger than our simple program. Usually languages allow you to make the speaker go bleep. BB4W allows you to access the soundcard straight from native commands. Entire books have been written on SOUND and its best friend ENVELOPE, but we'll just give the aerial view here. If you want more, the best way is to experiment and read the help files - well worth it.

Right, let's make a simple explosion - just a burst of white noise will do. Go into immediate mode and type:

```
SOUND 0,-15,4,20
```

SOUND takes four parameters. In order they are:

Channel	values 0 to 3. 0 is for noises, as above, 1 to 3 allow musical notes.
Volume	values -15 (loudest) to 0 (silent)
Pitch	on channel 0 this is the type of noise - values are 0 to 7 on channels 1 to 3 this is the pitch of the note - value 0 to 255
Duration	values -1 to 254, the length of time to play the sound in twentieths of a second. -1 plays forever.

When using channel 0, the third parameter alters the noise produced. 0 to 3 give buzzing noises and 4 to 7 give white noise or hiss. Here are some examples to start

you off:

```
SOUND 0,-5,4,20
SOUND 0,-15,0,100
SOUND 0,-10,3,-1
```

There's no actual difference between the white noise generated with pitch set from 4 to 7.

To play a tune, we use channels 1 to 3. The channels are the same in terms of capacity to play notes, they all have the same range. There are three so you can play chords and counter-melodies if you wish. As the second parameter (volume) and the fourth (duration) retain the same function, we only need to concentrate on the pitch. Values range from 0 to 255, but with 0 you don't get a sound, so realistically the lowest pitch is 1 up to the highest of 255. Try:

```
SOUND 1,-15,1,20
SOUND 1,-15,255,20
```

Increasing the pitch by one raises the pitch by one quarter of a semitone, if you understand such things. Put another way, the difference between two consecutive frets on a guitar or two consecutive notes on a keyboard is equal to four increments of pitch in the SOUND statement. There is a complete list of the values for all the available notes in the help files under SOUND and if you're writing tunes, you'll need to get a feel for these values. For our part, we just want a little three-note sequence like this:

```
SOUND 1,-15,200,2
SOUND 1,-15,208,2
SOUND 1,-15,212,2
```

Put the noise in the alien collision, the tune in the meteor section and here, at last, is

our complete game.

```
REM Meteor run
VDU  23,220,129,219,255,255,126,60,60,24
VDU  23,221,24,60,118,221,251,110,60,24
VDU  23,222,153,189,219,126,36,60,36,36
REM Build characters
Ship$=CHR$ (17)+CHR$ (3)+CHR$ (220)
Rock$=CHR$ (17)+CHR$ (4)+CHR$ (221)
Alien$=CHR$ (17)+CHR$ (2)+CHR$ (222)

MODE  6
X%=20
Score%=0
Lives%=3
OFF
*REFRESH OFF
REPEAT
  WAIT  15
  REM Print at bottom of screen
  PRINT  TAB (RND (40),24);Alien$;
  PRINT  TAB (RND (40),24);Rock$
  REM Reprint ship
  PRINT  TAB (X%,0);Ship$
  *REFRESH
  REM Read character in front of player
  Ch%=GET (X%,1)
  IF  Ch%=222 THEN
    SOUND  0,-15,4,10
    Lives%-=1
    WAIT  20
  ENDIF
  IF  Ch%=221 THEN
    SOUND  1,-15,200,2
    SOUND  1,-15,208,2
    SOUND  1,-15,212,2
    Score%+=10
    WAIT  20
  ENDIF
```

```
  REM Detect ship movement
  IF  INKEY (-26) AND  X%>0 X%-=1
  IF  INKEY (-122) AND  X%<39 X%+=1
UNTIL  Lives%=0
*REFRESH ON
CLS
COLOUR  1
PRINT  TAB (13,12);"You scored " ;Score%
ON
END
```

Notice that with the SOUND statements we still have to put a WAIT in there. BASIC doesn't halt the execution of the code until the duration of the sound has expired. Instead it places the instructions in a queue and continues on its way. The sound queue is then processed at its own rate.

Most of what you can do with SOUND involves experimentation and imagination. If you're interested investigate the entries in help. They tell you how to synchronise the sounds on different channels so you can play chords and how to alter the 'shape' of the sounds using ENVELOPE. Here are a couple of effects to whet your appetite:

```
REM Siren

Inc%=1
Pitch%=100
FOR  I%=1 TO  20
  IF  I%=10 Inc%=-1
  Pitch%+=Inc%
  SOUND  1,-15,Pitch%,2
NEXT  I%

END

REM Modem

FOR  I%=1 TO  10
  SOUND  1,-15,RND (4)*4+150,1
  SOUND  1,-15,0,2
  WAIT  3
```

```
NEXT  I%

SOUND  0,-15,2,15
SOUND  0,-15,0,15
SOUND  0,-15,4,25

END
```

I would like to emphasize here that this is not the limits of BBC BASIC in terms of graphics and if you are interested in games, there is even an entire sprite library. Using these simple techniques, however, you can liven up your programs quite considerably. Once upon a time, the computer world was awash with little programs like our game, basically because the micros of the day didn't have enough memory to do anything else! I believe there is still a place for fun programs like this. Just because you have gigabytes of memory doesn't mean your program is better for using as much as possible. As you progress as a programmer, your programs will get bigger. How to plan and develop these is the subject of the next chapter. In the meantime, you can have a lot of fun practising BASIC by concocting little programs such as this.

Exercises

1) Create a sound effect that uses short bursts of hiss to generate a noise like a machine gun.

2) Here is an incomplete game:

```
REM Chicane
VDU  23,220,255,213,171,213,171,213,171,255
VDU  23,221,90,126,90,24,90,126,90,24
Road$=CHR$ (17)+CHR$ (3)+STRING$ (4,CHR$ (220))
Car$=CHR$ (17)+CHR$ (4)+CHR$ (221)

MODE  6
CarX%=20
RoadX%=18
Score%=0
Lives%=5
Count%=0
```

```
OFF
REM Fill screen before we start
FOR  I%=1 TO  24
  WAIT  20
  PRINT  TAB (RoadX%,24);Road$
NEXT  I%
*REFRESH OFF
REM Main loop
REPEAT
  WAIT  20
  REM Score
  Score%+=1

  REM Create new road
  RoadX%=RoadX%+RND (3)-2
  REM Keep on screen
  IF  RoadX%<5 THEN  RoadX%=5
  IF  RoadX%>30 THEN  RoadX%=30
  PRINT  TAB (RoadX%,24);Road$;

  PRINT  TAB (0,24)
  PRINT  TAB (CarX%,0);Car$
  *REFRESH

  REM Add car control code here ...

UNTIL  Lives%=0

FOR  I%=200 TO  150 STEP  -1
  SOUND  1,-15,I%,1
  WAIT  1
NEXT  I%
*REFRESH ON
CLS
COLOUR  7
PRINT  TAB (6,12);"You scored " ;Score%
ON
END
```

Try it. You will see a road snake its way up the screen. Your task is to add the lines of code that control the car. This should be added where indicated. Inspect the left and right keys and adjust the position, *CarX%* , accordingly. Keep the car in the range of 5 to 34. Examine the character in front of the car. If it is not 220 (the road character), make a noise and lose a life.

Chapter 20 - Developing Real Programs

This is the final part of our tutorial. This section deals with how we write programs. It doesn't introduce any new BASIC commands but does try to put across one of the most popular methods of writing a program. Up to now, all our programs have been little 10 line wonders designed to demonstrate individual aspects of the language. There is a tendency amongst programmers to start small like this and build a bit at a time: add a couple of lines here, a subroutine there ... Eventually the whole thing becomes a tangled mess, hard to follow and harder to debug or expand.

Traditionally programming was taught by teaching people to break things down into logical, progressive steps. Sometimes this is so easy you don't consciously do it. For example in the section on user defined characters, we have to set the character for the alien before making him walk across the screen. It wouldn't be any use to do this the other way round. Designing a 'proper' program follows a similar line but on a much larger scale. So much so that it might be days before we actually get round to typing in any code. The general advice is to concentrate on the generalities first, the big picture, and then break these tasks into progressively smaller ones until they get so small the code just drops out of its own accord.

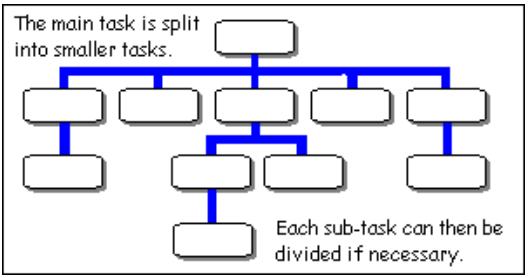
It's very easy to wave your hands in the air trying to make generalized gestures in the hope that everyone will see what you mean, like playing Charades. To avoid this, I intend to walk through the design of an actual program, explaining the steps as we go so you can apply them to your own creations. The process is often referred to as developing from the 'top down'. This is a common technique used by many programmers, not something I dreamed up after three nights of reading William Gibson novels. The reason many programmers use it is it's easily adapted to different languages and it gives good results, i.e. it works.

If we take user defined characters, we can see that these are very useful. What can be a pain is actually setting up the grid and doing the mental arithmetic to derive the necessary row totals. This looks like a good candidate for a program to me.

How do we start? There is a very loose language called pseudo-code whose exact definition is somewhat vague as everyone tends to develop their own. What it

usually looks like is a mixture of standard English with the odd word of BASIC thrown in. What is represents is the logical sequence of events necessary to accomplish the task in hand.

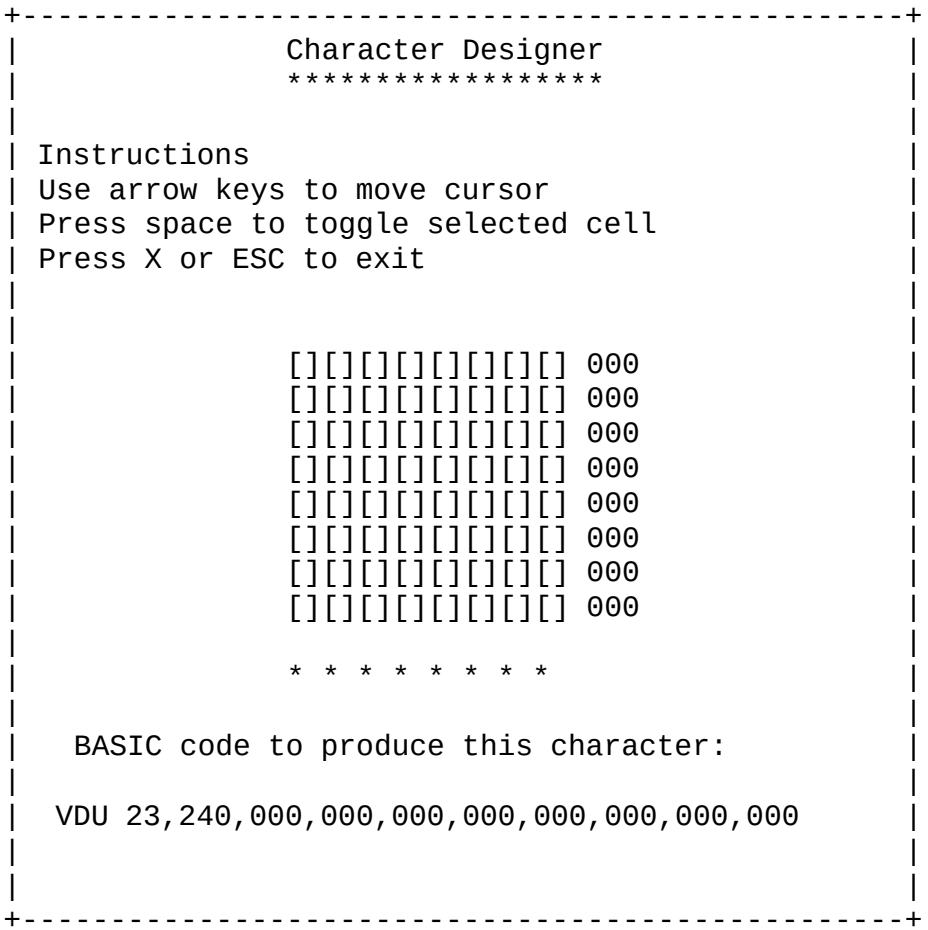
We start by getting the big picture. By this I mean we break down the tasks our program has to deal with into large chunks each of which can be summed up in a single line of several words.



Something like this:

```
Declare global arrays and structures
Initialization
Draw main screen
Draw character
Draw cursor
REPEAT
  Get user action
  Process user action
UNTIL User chooses exit
Shutdown
END
```

That's our basic program design. As you see, no specifics, everything is delegated to sub tasks. The next step is to take each of these sub tasks and refine them in a similar fashion. We'll leave the first two lines until later because at this early stage we have no idea what variables we need or how to initialize them. We start with Draw main screen. For this we need a rough idea of what our user will be presented with when the program is running. Sketch this on paper first, use squared paper if you want so you get a better idea of size. Here's what we're aiming for:



The top half of the screen is what I would term the main screen, it won't change much during the course of the program. The lower part is the grid, VDU codes and life-sized characters. This will change a lot, so the design must deal with this separately. The line of stars underneath the grid represents a line of the actual characters. One thing we can instantly see is that the VDU line is bigger than the 40 characters that MODE 6 gives up, so we'll need to select another mode to accommodate this. MODE 21 gives us 50 columns which should be fine. We now know enough to work out what *Draw main screen* should look like.

```
Draw main screen

Set background colour
CLS
PRINT Title
PRINT help instructions

END
```

CLS was used instead of 'clear the screen' because it's obvious this is the required command.

Not much to do there, the next one promises to be a little more challenging: *Draw character* . This can be seen to consist of three separate parts: the grid, the actual character and the VDU codes. Keeping it general we get the following.

```
Draw character

Draw grid
Draw actual character
Print VDU codes

END
```

It's pretty obvious there's more work to be done here, but this is a first pass so we'll leave the detail until later. During the design, things often change, so there is no point getting too involved at this stage.

Next up: *Draw cursor* . This will be used to move the cursor around the grid. To make the cursor appear to move, we need to first erase it from its old position and redraw it in the new one. We can use a different colour to highlight the cursor as it travels round the grid, so we need to know whether the cell it's on is set or not.

```
Draw cursor

Set to normal colour
Erase cursor at old position
Set to highlight colour
Highlight cell at current position

END
```

Now we dive into the REPEAT loop and find *Get user action* . This will be responsible for intercepting key presses from the keyboard, filtering out the unwanted and translating needed ones into a code for the rest of the program to use. From our sketched screen, we can deduce that the keys we are interested in are the arrow keys, the space bar and X or x. Using ESC to exit is a built-in feature, so we don't need to deal with it as such. The program can sit and wait for a key to be pressed as all other actions depend on this. Once pressed we can decide if the key is useful or not and translate it to a code if it is.

```
Get user action

REPEAT
  Wait for key press
  Translate key press into code if valid
UNTIL valid key found

END
```

Lastly, for the moment, we have *Process action* . This will take the action from the previous routine and do something with it. The options are quite straightforward so we can write it without further ado.

```
Process action

CASE Code OF
WHEN Up:    Move cursor up
            Draw cursor
WHEN Right: Move cursor right
            Draw cursor
WHEN Down:  Move cursor down
            Draw cursor
WHEN Left:  Move cursor left
            Draw cursor
WHEN Exit:  Set global Exit flag
ENDCASE

END
```

Okay, that's our first pass, let's bring it all together so we can see it in one place.

```
Main program

Declare global arrays and structures
Initialization
Draw main screen
Draw character
Draw cursor
REPEAT
  Get user action
  Process user action
UNTIL User chooses exit
Shutdown
END

Draw main screen

Set background colour
CLS
PRINT Title
PRINT help instructions
```

```
END

Draw character

Draw grid
Draw actual character
Print VDU codes

END

Draw cursor

Set to normal colour
Erase cursor at old position
Set to highlight colour
Highlight cell at current position

END

Get user action

REPEAT
  Wait for key press
  Translate key press into code if valid
UNTIL valid key found

END

Process action

CASE Code OF
  WHEN Up:    Move cursor up
              Draw cursor
  WHEN Right: Move cursor right
              Draw cursor
  WHEN Down:  Move cursor down
              Draw cursor
  WHEN Left:  Move cursor left
```

```
                Draw cursor
  WHEN Toggle:Toggle grid postion
  WHEN Exit:  Set global Exit flag
ENDCASE

END
```

If you were at all intimidated by the thought of writing a reasonable size program like this, I hope you can now see how we are beginning to split it up into the bite-sized pieces you've been used to working with so far.

Several of the routines look like they could use more refinement. These are *Draw character* and *Process action* .

Draw character has three lines that need thinking about. The first says 'Draw grid'. Drawing the grid includes the row totals at the end of each row too. What we need to do is decide if drawing the grid and calculating the row totals will need sufficient code to merit its own routine or will it go into this one without overloading it? By now it's becoming obvious that we're going to need an array to hold the grid. To draw the grid, we'll use one screen position for each cell, so all we need to do is go through each row and column printing the correct character as we go. As we are already embroiled in the grid, it makes sense to work out the row totals here as well. For each row, we need to set the total to zero, then, when we find a grid position that's 'on', add the value of that column to the total. If we have a value set to 128 and halve it for each column, we can work out the value for that column.

```
Draw grid and calculate row totals for each row
FOR each row
  Set column value to 128
  FOR each column
    Set row total to 0
    IF Grid at row column position = 0 THEN
      PRINT at position empty cell character
    ELSE
      PRINT at position filled cell character
      Set row total to row total + column value
    ENDIF
    Set column value to column value / 2
NEXT column
PRINT at end of row row total
```

```
NEXT row
```

Personally, I think that fits comfortably in the existing routine, no need for a new one. Let's dissect the next line: Draw actual character. We have already found the values for each row in the previous lump of code so this is now quite easy.

```
Call VDU 23 with chr 240 and row totals
PRINT in position character 240 eight times
```

Finally we need to print the totals out in a nice line so users can copy them into their programs.

```
PRINT in position "BASIC code to produce this charac
PRINT in position "VDU 23, 240";
FOR each row
  PRINT ",";total for row;
NEXT row
```

So our entire routine looks like this:

```
Draw character

Draw grid and calculate row totals for each row
FOR each row
  Set column value to 128
  FOR each column
    Set row total to 0
    IF Grid at row column position = 0 THEN
      PRINT at position empty cell character
    ELSE
      PRINT at position filled cell character
      Set row total to row total + column value
    ENDIF
    Set column value to column value / 2
NEXT column
PRINT at end of row row total
NEXT row

Call VDU 23 with chr 240 and row totals
PRINT in position character 240 eight times
```

```
PRINT in position "BASIC code to produce this charac
PRINT in position "VDU 23, 240";
FOR each row
  PRINT ",";total for row;
NEXT row

END
```

See how we still kept the original task descriptions? These usually end up as REMs in the completed program as they tell the reader what each section is trying to achieve.

Onto *Process action* . Two things here. The first one is a slight restructure. Each action involves redrawing the cursor, so maybe we should pull this out of each statement and move it to the end, saves space and typing. That's easy enough because we haven't written any code yet. The other is the use of move cursor. We have a choice again: make a separate routine or write code in each place. I chose the first option because then we can write a generic routine to handle all cursor movement. We can make a note of this and design the routine in a minute. Meanwhile, is there anything else? We haven't dealt with what happens when the space bar is pressed. In reality, this is not too much of a challenge:

```
WHEN Toggle:  Toggle grid position
              IF Grid at cursor position = 0 THEN
                Set Grid at cursor position to 1
              ELSE
                Set Grid at cursor position to 0
              ENDIF
```

All we need to do is update the grid, *Draw character* in the main program will sort the rest out. This is our revised *Process action* routine:

```
Process action

CASE Code OF
WHEN Up:    Move cursor up
WHEN Right: Move cursor right
WHEN Down:  Move cursor down
WHEN Left:  Move cursor left
WHEN Toggle:Toggle grid postion
            IF Grid at cursor position = 0 THEN
              Set Grid at cursor position to 1
            ELSE
              Set Grid at cursor position to 0
            ENDIF
WHEN Exit:  Set global Exit flag
ENDCASE
Draw cursor

END
```

After all this, we can return to our cursor movement routine. All that is needed here is to take the direction the cursor wants to move in and check that it is a valid position i.e. not off the end of the grid. If the postion checks out move the cursor there. Before moving the cursor, we need to remember the previous position so *Draw cursor* can erase it before highlighting the new position.

```
Move cursor

Save current position in old position
CASE Direction OF
  WHEN Up:    IF Cursor row > 1 THEN
                Decrease Cursor row by 1
  WHEN Right: IF Cursor column < 8 THEN
                Increase Cursor column by 1
  WHEN Down:  IF Cursor row < 8 THEN
                Increase Cursor row by 1
  WHEN Left:  IF Cursor column > 1 THEN
                Decrease Cursor column by 1
ENDCASE
```

```
END
```

Version 2 of our program design now looks like this:

```
Main program

Declare global arrays and structures
Initialization
Draw main screen
Draw character
Draw cursor
REPEAT
  Get user action
  Process user action
UNTIL User chooses exit
Shutdown
END

Draw main screen

Set background colour
CLS
PRINT Title
PRINT help instructions

END

Draw character

Draw grid and calculate row totals for each row
FOR each row
  Set column value to 128
  FOR each column
    Set row total to 0
    IF Grid at row column position = 0 THEN
      PRINT at position empty cell character
    ELSE
      PRINT at position filled cell character
      Set row total to row total + column value
```

```
    ENDIF
    Set column value to column value / 2
NEXT column
PRINT at end of row row total
NEXT row


Call VDU 23 with chr 240 and row totals
Call PRINT in position character 240 eight times

PRINT in position "BASIC code to produce this charac
PRINT in position "VDU 23, 240";
FOR each row
  PRINT ",";total for row;
NEXT row
END

Draw cursor

Set to normal colour
Erase cursor at old position
Set to highlight colour
Highlight cell at current position

END

Get user action

REPEAT
  Wait for key press
  Translate key press into code if valid
UNTIL valid key found

END

Process action

CASE Code OF
WHEN Up:    Move cursor up
```

```
    WHEN Right: Move cursor right
    WHEN Down:  Move cursor down
    WHEN Left:  Move cursor left
    WHEN Toggle:Toggle grid postion
                IF Grid at cursor position = 0 THEN
                  Set Grid at cursor position to 1
                ELSE
                  Set Grid at cursor position to 0
                ENDIF
    WHEN Exit:  Set global Exit flag
    ENDCASE
    Draw cursor
    END

    Move cursor

    Save current position in old position
    CASE Direction OF
      WHEN Up:    IF Cursor row > 1 THEN
                    Decrease Cursor row by 1
      WHEN Right: IF Cursor column < 8 THEN
                    Increase Cursor column by 1
      WHEN Down:  IF Cursor row < 8 THEN
                    Increase Cursor row by 1
      WHEN Left:  IF Cursor column > 1 THEN
                    Decrease Cursor column by 1
    ENDCASE

    END
```

Isn't cut and paste wonderful?

Finding the variables

The design has reached a point where you can start to 'see' the underlying code i.e. we've refined it enough. It's now time to invite the friends and family round so we can play 'hunt the variable'. At this point, I usually get a red pen and scribble on the pseudo-code. This is a bit difficult to do in Notepad or Internet Explorer, so we'll do a blow by blow analysis of each routine. Fortunately this will not take as long as the

previous section. For each routine, we need to know the name, type and scope of each variable we find.

Looking through the main program, there are two variables that spring out. One is the flag that is used to Exit, the other is the choice of action from *Get user action* routine. Both are integers and by virtue of the fact that they are in the main program, global. Exit will need to be set up to false when the program is started, so this is a job for the initialization routine.

Draw main screen doesn't have any variables, it's all print statements. Easy!

Draw character is a bit more involved. We will need two local integers for the FOR loops, call them *Row%* and *Col%* . Another integer is needed for the value of each bit, call this *ColValue%* , again, no-one else needs to know about this, so it's local. The main character data itself is crying out to be held in a two dimensional array. We'll call this *Grid%* and it is global as other routines need access to it. Another array is needed to hold the totals for the rows, *RowTotal%* . This will be a single dimension and be integers. Again, we'll make this global.

Draw cursor seems to have four integers associated with the current and previous position of the cursor. It would be nice to keep all these together, so let's put them in a structure. The structure needs to be global as we know that the *Move cursor* routine uses it as well. All are integers and the initial position will be set in the initialization routine.

Get user action has one integer variable to hold the key press, *Key%* , and one integer to hold the return code, *Code%* . Both are local.

Process action can have the value for the action code passed to it. The *Exit* flag is global as previously described.

Move cursor manipulates the cursor structure as described in draw cursor.

There are several other details that need to be considered. For example, the position of each PRINT statement needs to be decided. We can take a rough guess at this based on the screen layout, but be warned, you never get it right first time round!

One thing we have not considered is how to represent the grid. Each position can be depicted by one character position. If it's filled we can use a solid block. If it's empty a space would seem to be the obvious choice. However, if we use a space, how do we know where the cursor is? We need a character for an empty cell as well. One

with a single bit outline will suffice. So we need two user defined characters:

```
XXXXXXXX  255          XXXXXXXX  255
XXXXXXXX  255          X......X  129
XXXXXXXX  255          X......X  129
XXXXXXXX  255          X......X  129
XXXXXXXX  255          X......X  129
XXXXXXXX  255          X......X  129
XXXXXXXX  255          X......X  129
XXXXXXXX  255          XXXXXXXX  255
```

Where 'X' = filled and '.' = empty. Characters 241 and 242 will do fine for these. Setting them up is a job for the initialization routine.

Using this information, we can now write the initialization routine.

```
Initialize

Set graphics mode
Setup user characters 240, 241 and 242
Setup cursor location
Turn off text cursor

END
```

... and lastly, it's complement, *Shutdown* , whose job it is to do any tidying up before the program stops. We don't want to clear the screen here as the user might want to make a note of his new creation after the program has finished. We'll just turn the text cursor back on and move it to the bottom of the screen so it's not in the way.

```
Shutdown

Re-enable text cursor
Send cursor to bottom of screen

END
```

The Program

At last we are in a position to write the code. As you have probably guessed by now,

each task such as *Draw character* should be kept separate from the main body of the code. To achieve this, we use PROCs and FNs. I'm not going to give a line by line account of this as if you have followed all the above, there's nothing that should shock or astound you. Click the link for the complete program. I like to leave blank lines to make things easier to read. Also each task is separated by a line of stars and a little description, this makes it easier to find when scanning through the code. I know I said at the beginning that you should type all the code in by hand, please feel free to do this, but just in case you are itching to see what the finished creation looks like, here's a cut and paste friendly version. Click the link below then, if you're in Internet Explorer, click the Edit menu and Select All, Edit again and Copy. You can then paste the whole thing into the editor. If you have another browser, there will probably be a similar method lurking in there somewhere. Use Back on your browser to return here when you've looked at it.

Main program listing

Conclusion

And there is our program. Not too bad, was it? The problem with writing this down is it makes it appear like a completely linear process. It's not. Often the pseudo-code will go through a number of revisions before settling on the final version. Even then when allocating variables or writing code you will encounter situations that necessitate going back and rehashing something. Far from being a waste of time, it's the best way to develop. Why? Simply because the more you sit down and think about something, chewing over different methods of doing it, the more chance you stand of getting it right. If you had an important interview or appointment, you wouldn't trust to luck that you could find the way to your destination without planning a route. So how would you expect to write a decent program if you just roll up your sleeves and start typing? Should you do this, on having typed in your program it's all too easy to try and bodge round the bits that don't behave rather than retrace your steps and admit that you made a wrong decision early on. If you didn't plan, there are no steps to retrace and everywhere you turn is quicksand. If you did plan, chances are with a bit of experience you would have recognised the problems with your approach and thrown it out before it ever reached the coding stage. As I previously stated, none of the ideas here are new or mine. I have no vested interest in pushing this method except that I was taught it early in my career and have used it in PASCAL,C/C++, and VB/VBA. Oh, and BBC BASIC too. So have hundreds of other programmers. Use it. It works. (End of rant.)

Exercises

1) I'm sure you can see lots of improvements here, try adding two more commands, one to completely clear the grid and one to fill it. You could use C and F to do this. Notice how it is easy to code modifications like this because the structure makes everything easy to find.
2) It's nice to be able to reverse engineer characters too, given the row totals. Modify the project to allow the user to enter a total for the row he's currently on. Then redisplay the character.

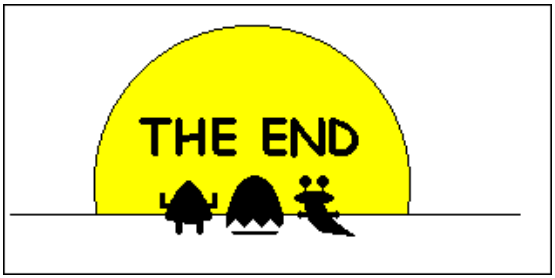
Chapter 21 - Over to You

If you have followed everything so far, you should have a thorough understanding of the principles involved in writing programs. You've actually covered a lot of ground and I hope the explanations and examples here will serve you for a while to come yet. Don't try to memorize everything here before you start writing your own programs, no one does that. Just keep in mind where to find the information that you need for a particular command, either here or in the help files.

As stated in the introduction, this only covers the basics, but enough basics to enable you to tackle a wide variety of applications. Look at other people's code, try and figure out how it works. Read the help for the keywords and techniques you don't know. You now know enough to make your mark in the world.

When looking for challenges start small. Every programmer has an unfinished project that they started but never quite got round to completing. The usual reason is lack of experience and / or time. Beware of tackling something that is too big too early. This way you won't get discouraged when your attempt to rewrite Windows or produce that killer version of Quake before next Saturday fails.

Thank you for your interest in this and I hope it has been a valuable insight into the wonderful world of programming.



Appendix A - Line Numbers and the Dreaded GOTO

Most seasoned programmers will throw up their hands at the mention of line numbers and GOTO. They are not considered polite conversation and ladies should be asked to leave the room before being discussed. There are an equal number of programmers who swear by them, rather than at them, insisting that with proper use they are valid. The truth is probably somewhere between the two. Don't e-mail me on this as it is a point over which many kilobytes have been exchanged without convincing anyone to change sides. At some point you will come across them in other people's programs so it would be remiss not to mention them.

Line Numbers

Traditional BASICs, of which this is one (and that's something to be proud of), always had line numbers. Though their use has fallen somewhat by the wayside, in BB4W they are still useful. If you make a mistake on a line, when the program is run Basic will present you with an error message specifying the line on which it found the mistake. BB4W allows lines in the range 0 - 65535. It is traditional to start with 10 and go up in increments of 10.

Lines must be in numeric order and no duplicates are allowed. If you need to insert another line, make sure it falls between the two numbers before and after or BB4W will complain. If you need to insert lots of lines, BB4W allows you to enter the lines without numbers and will renumber them afterwards if you ask it nicely. To demonstrate this, enter the following revolutionary piece of code:

```
REM Line number demo
PRINT "Hello, world"
WAIT 100
PRINT "That's all folks!"
END
```

Now, from the toolbar select Renumber. A dialogue appears: accept the default settings which start at 10 and go up in 10s and press OK. The listing is updated. For

a bigger program, this is a lot easier than renumbering by hand. Run the program to prove it works. It is also possible to remove line numbers. To do this, invoke the renumber box again and tick 'Remove unused line numbers'. 'Unused ' will be qualified in the next section on GOTO, but it is possible to get a program to leap around within itself during execution. The target of these jumps can be a line number, so if a line jumps to a line number, this number is classed as used and will not be removed.

GOTO

GOTO is a command that is used in conjunction with a line number. When the program executes, a GOTO statement will cause an immediate jump to the line number specified and execution will continue from there. For example, in our well worn circle program, we can make it loop endlessly by inserting line 50.

```
10 REM Area of a circle
20 INPUT "Enter the radius " Radius
30 Area=PI *Radius^2
40 PRINT "Area of the circle is " ;Area
50 GOTO 20
60 END
```

GOTO can be used as a quick and dirty way to skip a block of code when testing:

```
400 REM Stuff
410 GOTO 500
420 REM Help screens you've already tested
430 REM ...
440 REM ...
450 REM ...
460 REM ...
470 REM ...
480 REM ...
490 REM ...
500 REM Interesting stuff starts here
```

GOTOs can be made conditional:

```
10 REM Area of a circle
20 INPUT "Enter the radius " Radius
30 Area=PI *Radius^2
40 PRINT "Area of the circle is " ;Area
50 INPUT "Another go (Y/N)" ; Reply$
60 IF Reply$="Y" OR Reply$="y" THEN GOTO 20
70 END
```

And in single line IFs (don't ever jump out of or into a multi-line block with GOTO) you can miss the THEN or GOTO out completely:

```
50 IF TestResult>50 THEN 1000
```

Or:

```
50 IF TestResult>50 GOTO 1000
```

But never:

```
50 IF TestResult>50 1000
```

Overindulgence can lead to what is commonly known as spaghetti code, which is why people hate it so much. Don't bother to type this in but see if you can follow it:

```
10 GOTO 60
20 PRINT "Help!"
30 PRINT " lines 20 and 50 " ;
40 GOTO 80
50 PRINT "I'm stranded!"
60 PRINT "Why are" ;
70 GOTO 30
80 PRINT "never executed?"
90 END
```

Tip: Not every line needs a number

You don't have to have a line number on each line, only the ones that are a target for a GOTO or similar instruction. The following is perfectly legal:

```
      REM Area of a circle
   20 INPUT "Enter the radius " Radius
      Area=PI *Radius^2
      PRINT "Area of the circle is " ;Area
      GOTO 20
      END
```

This techinque can be used when converting old programs until you have the structure just right and can eliminate the GOTOs completely. When you select 'Remove used line numbers' from the renumber dialog, this would be the result.

The very, very early (we're talking late 60s or early 70s) BASIC programmers had to use GOTO because there wasn't a choice. BB4W has many options to help with conditional execution and repeating blocks of code. It's largely not needed, but, as previously stated, you will see code with it in. If you do use GOTO:

a) Keep jumps short, preferably on one page so you can see the destination.
b) Don't jump in and out of blocks of code like PROCs, loops or IFs (can't emphasize this enough), you completely mess BB4W's internal stacks up when you do this.
c) Never admit to it in public, you'll cause a riot.

Appendix B - Words We Mustn't Say

There are several common BASIC commands that are not mentioned in the main text, but you may have come across them elsewhere e.g. in the help files or other, lesser dialects of BASIC. You are advised not to use these in your programs as small children will mock you in the street as you walk past if you do. The words are GOSUB, ON .. GOTO, ON .. GOSUB and LET. Unlike GOTO, which still has its adherents, these keywords have definitely outlived their usefulness and BB4W offers much better alternatives. We will now make them stand at the front of the class so we can openly scorn them.

GOSUB

GOSUB is a distant relative of PROC. The idea was that you could split your program into sections and call them like a PROC. Unlike a PROC, you could only GOSUB to a line number rather than give a meaningful name. Once found, BASIC would jump off to the given line number, run the code until a RETURN statement was found (similar to an ENDPROC). Then it would jump back to the next statement after the GOSUB. Regrettably, you could not pass or return values and there was no provision for LOCAL or PRIVATE variables, everything had to be global. Awful - don't bother.

ON .. GOTO and ON .. GOSUB

Here a variable is used to decide which line number to jump to.

```
 10 REM Simple menu system
 20 CLS
 30 PRINT "Press 1 for option 1"
 40 PRINT "Press 2 for option 2"
 50 PRINT "Press 3 for option 3"
 60 INPUT "Enter choice: " C%
 70 ON C% GOTO 80 ,90 ,100
 80 PRINT "You chose option 1" : GOTO 110
 90 PRINT "You chose option 2" : GOTO 110
100 PRINT "You chose option 3" : GOTO 110
110 END
```

When *X%* is 1, the first line number (80) is selected, when *X%* is 2, the second (90) and so on. Again this very dependent on line numbers. ON .. GOSUB does the same job but uses subroutines as described above. In both options, use CASE, it's cleaner and easier to read and maintain.

ON .. PROC

BB4W has a variant to the above two called ON .. PROC which allows you to call a PROC dependent on the value of a variable. If the call to the procedures involves passing several arguments, the line quickly becomes unwieldy. The line could be split up using \ , but if you're spreading it over several lines, why not use CASE anyway?

LET

LET is used is to tell the computer that an assignment is about to take place. Using LET, our area calculator would become:

```
10 REM Area of a circle
20  LET Radius=1.5
30 LET Area=PI *Radius^2
40 PRINT "The area of the circle is " ;Area
50 GOTO 20
60 END
```

Not that destructive in terms of structure, just a waste of bytes really. You don't need it.

Appendix C - Binary and Hexadecimal

Numbers in Base 2 - Binary

Up to now, we have been content to think about computers putting numbers in little boxes called memory locations without being too concerned about how this actually happens. Largely, that's okay, like a car, you don't need to know how it works to get in it and drive to your destination. There comes a time however, when a little knowledge comes in useful and allows us to progress.

So, how does a computer store numbers? A computer's memory location is basically made up of switches. Millions of them. A switch can have two states: on or off. One switch on its own can therefore represent two numbers: 0 (off) and 1 (on). We combine groups of switches to allow us to store higher numbers. If we take two switches (call them 0 and 1) together, we can use this to represent 4 different numbers as there are four unique states. Here they are:

Switch		**Number**
1	**0**	
0	0	0
0	1	1
1	0	2
1	1	3

We go from right to left here, just as if we were reading 123 as being one hundred and twenty three. The lowest (right most) switch represents 1 when it's on. The next switch takes a value of 2 when switched on. To get the number represented, we add the values together. Let's add another switch:

Switch			Number
2	**1**	**0**	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Now we can represent numbers from 0 to 7. The first column assumes a value of 4 when on. We've got a pattern here, starting with switch 0:

```
Switch 0 when on is worth 1
Switch 1 when on is worth 2
Switch 2 when on is worth 4
```

Each additional switch is worth twice the previous one. If we add another, its value will be 8 and allow us to represent numbers 0 to 15. This method of counting is termed binary (where 'bi' signifies two because we are dealing with two states). Each switch, mathematically, is 2 to the power of its position. Hence:

```
2^0 = 1
2^1 = 2
2^2 = 4
2^3 = 8
etc.
```

When we talk about 8-bit numbers or 32-bit numbers, we're referring to the number of switches. Programmers call each switch a bit. Hence you can see that an 8-bit number is capable of representing 2^8 = 256 states. Here is what each bit is worth:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Once you grasp this, 16 and 32 bit numbers are just the same, merely more bits. Sometimes the topmost bit is used to represent whether the number is negative, rather than giving it a numeric weight. Hence our 8-bit number above could be used to hold 0 to 255 or –128 to +127 depending on the convention used. 8-bits is such a common grouping that it has its own name: a byte. We have already seen that BB4W has a data type called a byte, so in this instance the top bit takes a numerical value rather than a sign and we can hold numbers 0 to 255.

Numbers in Base 16 - Hexadecimal

It is important to realize that representing something in binary is the same as representing something in decimal. The physical quantity doesn't change, just the means of representation. The method is similar to speaking a foreign language. We can say we have 'two cars' in English or 'deux voitures' in French. We can say 2 in decimal or 10 in binary, it's all the same quantity. Quantities can be represented in any number system that suits us. These systems are said to be in different bases. The base of a number is taken from the value of the second column of the representation.

```
10 in decimal = ten hence we are in base 10
10 in binary = two hence we are in base 2
```

As humans, we can't work in binary for too long, all those 1's and 0's are just too confusing. What is needed is a way to represent a number which will quickly enable us to get back to binary should we need to. If we look at the pattern for the values of bits in binary (1,2,4,8,16 ...) we know that any of these numbers will use all the bits and so would be a good number base to use. If we take the next number after 2, we get 4. This is slightly better, but not much as our sequence of numbers now goes 0,1,2,3,10,11 etc. Try the next power of two, this is 8. This allows us to represent 0,1,2,3,4,5,6,7,10,11. That's getting there and base 8 numbers are used in some circles.

At the next number, 16, computer designers and engineers decided that this was the most convenient. Base 16 allows representation of numbers 0 to 15 (decimal) 0000 to 1111 (binary). Numbers in base 16 are called hexadecimal or hex for short. 10 in hex, 16 in decimal and 10000 in binary. As decimal runs out of characters at 9, the

letters A to F are used to represent the missing numbers 10 to 15. BBC BASIC allows us to use hex numbers in our code. To let BASIC know the number is base 16, we put & in front of it. For the letters A to F use uppercase, it can be selected lowercase, but let's go with the flow here and use defaults. We can use binary numbers by prefixing them with % like this: %111, %1000. A complete list of all the numbers in 8 bits would be pretty boring to read in full so here are the edited highlights:

Decimal	Hexadecimal	Binary
0	&0	%00000000
1	&1	%00000001
2	&2	%00000010
3	&3	%00000011
4	&4	%00000100
5	&5	%00000101
6	&6	%00000110
7	&7	%00000111
8	&8	%00001000
9	&9	%00001001
10	&A	%00001010
11	&B	%00001011
12	&C	%00001100
13	&D	%00001101
14	&E	%00001110
15	&F	%00001111
16	&10	%00010000
17	&11	%00010001
18	&12	%00010010
19	&13	%00010011
...	...	...
127	&7F	%01111111
128	&80	%10000000
129	&81	%10000001
...	...	...
253	&FD	%11111101
254	&FE	%11111110
255	&FF	%11111111

Hex is so convenient because of this direct correlation to binary. One hex digit

covers all possible combinations of four binary bits. The next digit just repeats the same pattern again.

Hex is so embedded in computers that BBC BASIC will even print it for us. In order to convert a number in decimal to a string representation in hex use STR$ with tilde:

```
PRINT STR$~(255)
```

To convert a number back from hex to decimal use:

```
PRINT EVAL ("&FFE")
```

Appendix D - Debugging

If you write programs you write bugs. Don't feel bad about this, look how many service packs and hot-fixes each version of Windows ends up with. Professional programmers write professional bugs! If you use a decent design method, you eliminate a lot of the stress of finding bugs. By splitting your program into chunks, each with a definable task, you can check each PROC and FN separately to ensure it does the job in hand before incorporating it into the main program.

Ideally, you should test each routine with every combination of variables possible, this is especially true when you are asking users to enter information. It's little use turning to the person that just crashed your beloved masterpiece to frustratedly exclaim, "Well, what did you press that key for anyway?". Your program should handle both invalid and valid inputs.

One of the things that separates good programmers from indifferent ones is the ability to seek and destroy bugs in code. (An ability to admit that you're wrong sometimes doesn't go amiss either!) Largely this skill is won through experience and downright pig-headedness: it's only a machine, I won't let it beat me. There are several techniques that can help here. In this section, we'll cover some. Please keep in mind that the list is not exhaustive. Every bug will have a slightly different solution and sometimes you just have to invent your own methods.

This discussion will completely ignore the syntax errors that occur when you misspell a variable name or keyword and BASIC squeals 'Mistake at line 1230'. What we're concentrating on is when the program runs smoothly enough, but just won't behave quite as expected.

The first tool in our armoury is our old friend PRINT. Not sure why your WHILE loop never exits? Put a PRINT in the loop with the variables that dictate the exit condition and watch the results scroll up the screen. There are times when too many PRINTs can spoil the display that should be on the screen, or the data changes too fast to see. In either of these cases you can use PRINT TAB to put the results somewhere out of the way and then have a dummy input line which stalls the program giving you control of when the data changes.

```
REM Pausing a loop
FOR I%=1 TO 10
REM spaces at end overwrite previous data
  PRINT TAB (0,24);I%;"    "
  Dummy$=GET$
NEXT I%
END
```

Line 5 just waits for any keypress and throws it away, thus stalling the loop. You can make this conditional in order to save key presses and time.

```
REM Pausing a loop
FOR I%=1 TO 10
  IF I%>5 AND I%<8 THEN
    PRINT TAB (0,24);I%;"    "
    Dummy$=GET$
  ENDIF
NEXT I%
END
```

The other command that can be embedded in our code is TRACE. TRACE is used with ON and OFF. When BASIC comes across TRACE ON, it will print every line number in square brackets until it encounters TRACE OFF or the program ends. Obviously, you need a program with line numbers to use this. Don't depend on seeing anything that the program is supposed to print out here because it'll zip past in a blur of line numbers. This command is more useful when you want to trace the path that the program takes for a given set of variables.

```
10 REM TRACING a program
20 TRACE ON
30 FOR I%=1 TO 10
40    PRINT TAB (0,24);I%;"    "
50 NEXT I%
60 TRACE OFF
70 END
```

TRACE will only display the line number when the line is first executed. If you have a loop in a line like this:

```
120 REPEAT : UNTIL INKEY$ (0)
```

The line will only show once in the trace despite the fact that BASIC is pounding round the loop looking for a keypress.

TRACE will also accept a line number as a parameter. The idea being that it will only print line numbers less than the one specified. If you write programs as is described in the final section of this tutor, all the PROCs and FNs come after the END, so by using the line number of the END instruction, you will get a view of the overall structure of the program.

I'm not giving too many examples here as the BBC BASIC editor has an excellent debugging tool that eclipses the above methods. Sometimes, even if it's just for inspiration, it's useful to have some other tricks up your sleeve and the above two have been around since BBC BASIC was first written.

As you may be aware, it is possible to see the editor when a program is running. Go to the Windows task bar at the foot of the screen and there it is - back in a click. We can't edit the code when it's running, but the menu still has options that are available to us whilst the program is in progress. From the Utilities menu, select List Variables. This opens a new window which contains a list of all the variables in use at the present time, along with their current values. Knowing this, you can watch a program run without inserting PRINT statements. Very useful.

The window can be resized or scrolled as you'd expect. LOCAL variables are also added to the list when a PROC or FN is entered, along with the name of the PROC or FN, so you can see where you are. If you have duplicate names, e.g. two PROCs that have local variables with the same name, the name only appears once in the list even though there are actually two variables in memory. The list just shows the one that's currently in scope.

There are several buttons in the same block as the immediate button. You have probably used the run button (large black arrow) and possibly the stop (black square). We'll now cover the other two, but first shall we have a program to test with?

```
REM Debug demo
FOR I%=1 TO 10
  PRINT I%
  PROC _Oops
NEXT I%
END

DEF PROC _Oops
LOCAL I
I%=10
ENDPROC
```

Here is a trivial bug. When PROC_Oops runs, it declares a local *I* , but sets the global *I%* to 10, hence the loop ends after just one iteration. We will now use the debugger to track this fault.

First we enable the Trace option from the Utilities menu by clicking on it. This is different to the TRACE command described earlier. When our program is run, the line currently being executed is highlighted in the editor. Try it. The immediate window opens and in the editor the END line is highlighted. As the program runs faster then we can see, that's not very helpful. The show's over before you can blink.

There is another option that allows us to go through the program one line at a time. To demonstrate this, put the editor's cursor on line 3, with the PRINT statement; it doesn't matter where. This time, instead of pressing Run, select the Run menu and click 'Run to cursor'. The Pause button on the toolbar is shown pressed and the Step button comes to life. Every time we press the Step button the program advances to the next line and waits. By stepping through, we can verify that the loop is executed just once. Do this and notice that we lose the Pause when the program ends.

Put the cursor back on line 3. Select 'Run to cursor' and when the pause becomes active, open the variable list. Step through again. When the program gets to PROC_Oops it creates a local called *I* , you can see it appear in the list. Stepping through into PROC_Oops, we see *I%* change value, not *I* as was expected. Round about now the cause of the bug hits us so we don't need to single step anymore. By clicking Pause, step is released allowing the program to end at normal speed. We can now correct our little bug, feel really pleased with ourselves and then mentally chastise ourselves for putting it there in the first place.

Appendix E - Answers to the Exercises

In all of these answers please bear in mind that programming is a very subjective art - if what you got worked, that's right, even if it's different from the answers given here. Rather than use these answers as a right / wrong comparison, see how they differ from yours and why. If you're floundering with a problem or stuck for ideas, have a peep, it's not an exam but rather a learning process.

Chapter 3

1) Modify the first program to read "BB4W" instead of "BBC".

```
REM My first program
PRINT "BBB   BBB     44   W   W"
PRINT "B  B  B  B   4 4   W   W"
PRINT "BBB   BBB   4  4   W W W"
PRINT "B  B  B  B  44444  W W W"
PRINT "BBB   BBB     4   WW WW"
END
```

2) Insert TABs into the print statements to print BB4W starting at row 10, column 10 in the output window.

```
REM My first program
PRINT TAB (10,10);"BBB   BBB     44   W   W"
PRINT TAB (10,11);"B  B  B  B   4 4   W   W"
PRINT TAB (10,12);"BBB   BBB   4  4   W W W"
PRINT TAB (10,13);"B  B  B  B  44444  W W W"
PRINT TAB (10,14);"BBB   BBB     4   WW WW"
END
```

Chapter 4

*1) Modify the area program to give the circumference of a circle 3.14159*diameter.*

```
REM Circumference of a circle
Diameter=5
Circumference=3.14159*Diameter
PRINT "The circumference is " ;Circumference
END
```

2) Print the area of a rectangle, when given the width and height.

```
REM Area of a rectangle
Width=5
Height=10
Area=Width*Height
PRINT "The area is " ;Area
END
```

3) Assign a string variable to contain your name and output:

```
Hello, xxxx
```

where xxxx is, of course, your name.

```
REM Print my name
Name$="Justin Time"
PRINT "Hello, " ;Name$
END
```

Chapter 5

1) Modify the circle program to use PI instead of 3.14159. Save it.

```
REM Area of a circle
Radius=5
Area=PI *Radius^2
PRINT "The area of your circle is " ;Area
END
```

2) Write a program that uses RND to simulate two six-sided dice being thrown. Print the results for each die and the total.

Two dice have a range of 2 to 12, but the probabilities of the different totals are not equal. For example there are five ways of getting a total of 6 (1+5, 2+4, 3+3, 4+2,

5+1) but only one way of getting a total of 12 (6+6). For this reason you must treat each die as a separate entity:

```
REM Tumblin' Dice
Die1%=RND (6)
Die2%=RND (6)
PRINT "Total " ;Die1%+Die2%
END
```

3) Verify that the following formula is true:

```
LOG(X)=LN(X)/LN(10)
```

The easiest way at this stage is to print the results and compare them visually.

```
REM Verify LOG(X)=LN(X)/LN(10)
X=1.234
PRINT LOG (X)
PRINT LN (X)/LN (10)
END
```

Chapter 6

1) Set a string to hold the days of the week like this:

```
"Sun Mon TuesWed ThurFri Sat"
```

All names are 4 characters in length including a space if necessary. Given a number for a day, use MID$ to extract the correct abbreviation for the day.

Each string is 4 characters long, so each name will start at position 1, 5, 9 etc. If we take the day number Sunday = 1 etc. and subtract 1 from it then multiply by 4, we get a sequence that goes 0, 4, 8 ... To complete the formula, add an offset of 1. The total calculation needs brackets to override the operator precedence.

```
(DayNumber-1)*4+1
```

```
REM Day name
Day$="Sun Mon TuesWed ThurFri Sat "
DayNum=3
DayName$=MID$ (Day$,(DayNum-1)*4+1,4)
PRINT "Day number " ;DayNum;" is " ;DayName$
END
```

2) Set a string to hold your first name. Use MID$ and ASC to find the ASCII codes of the letters in the name.

```
REM ASCII Codes in My Name
Name$="Peter"
PRINT MID$ (Name$,1,1);" has code " ;
PRINT ASC (MID$ (Name$,1,1))
PRINT MID$ (Name$,2,1);" has code " ;
PRINT ASC (MID$ (Name$,2,1))
PRINT MID$ (Name$,3,1);" has code " ;
PRINT ASC (MID$ (Name$,3,1))
PRINT MID$ (Name$,4,1);" has code " ;
PRINT ASC (MID$ (Name$,4,1))
PRINT MID$ (Name$,5,1);" has code " ;
PRINT ASC (MID$ (Name$,5,1))
END
```

I hope you used copy and paste for this one. You could also pull each letter out to a separate string like this:

```
Letter$=MID$ (Name$,1,1)
PRINT Letter$;" has code " ;ASC (Letter$)
```

3) Set three strings to hold your first name, second name (if you haven't got one, make it up) and surname. Use LEFT$ to find your initials and concatenation to create a new string in the format "R. T. Russell".

```
REM Initials
First$="Johann"
Second$="Sebastian"
Surname$="Bach"
Name$=LEFT$ (First$,1)+". "
Name$=Name$+LEFT$ (Second$,1)+". "
Name$=Name$+Surname$
PRINT Name$
END
```

Chapter 7

1) Examine the different MODEs in the help file, try printing text in MODEs 1 to 6 to get the feel for what they look like.

A bit open ended this one, the easiest way is to cut and paste three lines several times:

```
REM Different MODES
MODE 1
PRINT "Hello, World"
WAIT 200
MODE 2
PRINT "Hello, World"
WAIT 200
MODE 3
PRINT "Hello, World"
WAIT 200
MODE 4
PRINT "Hello, World"
WAIT 200
MODE 5
PRINT "Hello, World"
WAIT 200
MODE 6
PRINT "Hello, World"
WAIT 200
END
```

2) Modify the Changing colours program to produce new random colours by using three RND(255) statements in the call on lines 2 and 5.

```
REM Changing colours
COLOUR 3,RND (255),RND (255),RND (255)
REM Now we've set the colour change to it
COLOUR 3
PRINT "Hello, World"
COLOUR 3,RND (255),RND (255),RND (255)
PRINT "Hello, World"
COLOUR 0
END
```

With the window still open, RUN this several times to get different colours.

Chapter 8

*1) The volume of a cylinder is PI*Radius^2*Length. Write a program that will ask for radius and length then give the volume.*

```
REM Volume of a cylinder
INPUT "Enter the radius " Radius
INPUT "Enter the length " Length
Volume=PI *Radius^2*Length
PRINT "The volume is " ;Volume
END
```

2) Have a program prompt to enter your full name. Print the number of characters (including spaces) in the name.

```
REM Length of name
INPUT LINE "Enter your name, please " Name$
PRINT "Thank you " ;Name$
PRINT "Your name has " ;LEN (Name$);" characters."
END
```

Chapter 9

Below is a sample output from a two runs of a program that acts as a simple calculator:

```
Enter first number: 5
Enter second number: 6
Enter 1 to add or 2 to subtract: 1
5 + 6 = 11
>RUN
Enter first number: 45
Enter second number: 55
Enter 1 to add or 2 to subtract: 2
45 - 55 = -10
```

Can you write the program that produces the above screen? Allow one INPUT for each number, one INPUT for the operations. Use IF to select the correct PRINT statement and result, then another INPUT to ask for another go. Extra marks if you expand to include multiply and divide.

This is the first solution.

```
REM Basic calculator
INPUT "Enter first number " Num1
INPUT "Enter second number " Num2
INPUT "Enter 1 to add or 2 to subtract: " Op%
IF Op%=1 THEN
  PRINT Num1;" + " ;Num2;" = " ;Num1 + Num2
ENDIF
IF Op%=2 THEN
  PRINT Num1;" - " ;Num2;" = " ;Num1 - Num2
ENDIF
END
```

After this the second one is easy.

```
REM Basic calculator
INPUT "Enter first number " Num1
INPUT "Enter second number " Num2
PRINT "Enter 1 to add, 2 to subtract"
INPUT " 3 to multiply, 4 to divide: " Op%
IF Op%=1 THEN
  PRINT Num1;" + " ;Num2;" = " ;Num1 + Num2
ENDIF
IF Op%=2 THEN
  PRINT Num1;" - " ;Num2;" = " ;Num1 - Num2
ENDIF
IF Op%=3 THEN
  PRINT Num1;" * " ;Num2;" = " ;Num1 * Num2
ENDIF
IF Op%=4 THEN
  PRINT Num1;" / " ;Num2;" = " ;Num1 / Num2
ENDIF
IF Op%<1 OR Op%>4 THEN
  PRINT "Sorry, invalid operation."
ENDIF
END
```

Chapter 10

Recreate an executive decision maker. Generate a random number 1 to 6 and print a message depending on the result:

1 - Hire a yes man
2 - Fire someone
3 - Delegate
4 - Cancel all overtime
5 - Give yourself a rise
6 - Raid the pension fund

```
REM Decision maker
PRINT "Here is your decision for the day"
Dec%=RND (6)
CASE Dec% OF
  WHEN 1 : PRINT "Hire a yes man"
  WHEN 2 : PRINT "Fire someone"
  WHEN 3 : PRINT "Delegate"
  WHEN 4 : PRINT "Cancel all overtime"
  WHEN 5 : PRINT "Give yourself a rise"
  WHEN 6 : PRINT "Raid the pension fund"
ENDCASE
END
```

Chapter 11

1) Write a program with a FOR loop to print the 5 times table. Each line should be in the format:

```
1 * 5 = 5
2 * 5 = 10
...
12 * 5 = 60
```

```
REM 5 times table
FOR I%=1 TO 12
  PRINT I%;" * 5 = " ;I%*5
NEXT I%
END
```

2) Use two nested FOR loops to draw a rectangle on the screen by using a line like PRINT TAB(X,Y);"*"

```
REM Print a rectangle
INPUT "Enter Width " Width%
INPUT "Enter Height " Height%
CLS
FOR Y%=1 TO Height%
  FOR X%=1 TO Width%
    PRINT TAB (X%,Y%);"*"
  NEXT X%
NEXT Y%
END
```

Or, instead of clearing the screen, you could just offset the rectangle:

```
REM Print a rectangle
INPUT "Enter Width " Width%
INPUT "Enter Height " Height%
FOR Y%=1 TO Height%
  FOR X%=1 TO Width%
    PRINT TAB (X%+5,Y%+5);"*"
  NEXT X%
NEXT Y%
END
```

Chapter 12

1) Produce a program that will ask for a string and print the ASCII code of the first character. Have the program stop if the string entered is empty ("").

```
REM Find the code
REPEAT
  INPUT "Enter a string " MyString$
  IF LEN (MyString$) > 0 THEN
    A$=LEFT$ (MyString$,1)
    PRINT A$;" has ASCII code " ;ASC (A$)
  ENDIF
UNTIL LEN (MyString$)=0
END
```

2) Write a program that will ask for a number. Add this number to a running total. Repeat until the number entered is 0. Print the result.

```
REM Totalizer
Total=0
REPEAT
  INPUT "Enter a number " Num
  Total+=Num
UNTIL Num=0
PRINT "Total entered = " ;Total
END
```

Chapter 13

1) Modify the number filter program to accept one (and only one) decimal point in the number. If you're feeling ambitious, add a further modification to intercept backspace and delete the last character, if any, from the string. You'll need to print a space after the string to erase the character from the screen.

Picking up the keypresses is no problem, we need a flag to indicate that the string has had a decimal point entered. When deleting the final character, we have to check if it's a point and reset the flag accordingly.

```
REM Number filter
Total$=""
DP%=FALSE
PRINT "Type your number or Enter to finish"
REPEAT
  Key%=GET
  REM Check for numeric characters
  IF CHR$ (Key%)>="0" AND CHR$ (Key%)<="9" THEN
    Total$=Total$+CHR$ (Key%)
  PRINT TAB (0,1);Total$;
  ENDIF
  REM Check for decimal point
  IF CHR$ (Key%)="." AND NOT DP% THEN
    Total$=Total$+CHR$ (Key%)
    DP%=TRUE
    PRINT TAB (0,1);Total$;
  ENDIF
  REM Action backspace only if there's
  REM something to delete
```

```
  IF Key%=8 AND LEN (Total$)> 0 THEN
    REM If last character is a point,
    REM reset flag the DP flag
    IF RIGHT$ (Total$,1)="." DP%=FALSE
    Total$=LEFT$ (Total$)
    REM Print string with trailing space
    REM to erase deleted character
    PRINT TAB (0,1);Total$;" " ;
    PRINT TAB (0,1);Total$;
  ENDIF
  UNTIL Key%=13
PRINT '"You entered: " ;VAL (Total$)
END
```

*2) Make a simple drawing program. Use X% and Y% in a TAB statement to plot an 'X' on the screen. When you press the arrow keys, adjust X% or Y% according to the key pressed. For example, if you press left, decrease X%, down - increase Y%. Plot an 'X' at the new position so you can leave a trail on the screen. Restrict the area in which you can draw to a 20 * 20 grid.*

We restrict the cursor to a grid of 20*20 so there's no problem going off the edge of the screen. The program loops forever, press ESC to get out.

There are two possible commands here GET and INKEY, the first program is the solution using GET.

```
REM Sketch
X%=10
Y%=10
REM Print X in starting place
PRINT TAB (X%,Y%);"X"
REPEAT
  Key%=GET
  REM Move cursor in direction after checking
  REM we're still in limits
  CASE Key% OF
    WHEN 139: IF Y% > 0 THEN Y%-=1
    WHEN 137: IF X% < 19 THEN X%+=1
    WHEN 138: IF Y% < 19 THEN Y%+=1
```

```
    WHEN 136: IF X% > 0 THEN X%-=1
  ENDCASE
  REM Print X in new position
  PRINT TAB (X%,Y%);"X"
UNTIL FALSE
END
```

Here is the solution using INKEY. If you coded it this way, you probably discovered the importance of a WAIT command to stop the 'X' zipping from one side of the screen to the other.

```
REM X-A-Sketch
X%=10
Y%=10
REM Print X in starting place
PRINT TAB (X%,Y%);"X"
REPEAT
  REM Move cursor in direction after checking
  REM we're still in limits
  IF INKEY (-58) AND Y% > 0 THEN Y%-=1
  IF INKEY (-122) AND X% < 19 THEN X%+=1
  IF INKEY (-42) AND Y% < 19 THEN Y%+=1
  IF INKEY (-26) AND X% > 0 THEN X%-=1
  REM Small delay to stop too many prints
  WAIT 20
  REM Print X in new position
  PRINT TAB (X%,Y%);"X"
UNTIL FALSE
END
```

Chapter 14

Here are the figures for the first six months' sales of triple fruit chocolate covered syrup and treacle flavour ice lollies from one local newsagent:

```
January   105
February  261
March     482
April     195
May       347
June      626
```

Set an array to hold these values. Then add to the program so it loops through the values to find and print:

a) the month number for the lowest sales;
b) the month for the highest sales;
c) the total sales.

You can use the same FOR loop to achieve all three or do them separately: your choice.

Modify the program to have an array of month names, initialize it and adapt the above program to display real names for the months.

This is looks a lot but it's not too bad if you break it down and follow the examples.

```
REM Lolly sales
DIM Sales%(6), Month$(12)
REM Initialise
Sales%() = 0,105,261,482,195,347,626
Month$() = "" ,"January" ,"February" , \
\ "March" ,"April" ,"May" ,"June" , \
\ "July" ,"August" ,"September" , \
\ "October" ,"November" ,"December"
REM Set result variables
LowestSale% = 9999
LowestMonth% = 0
HighestSale% = -9999
HighestMonth% = 0
Total% = 0
REM Now find the data
FOR I%=1 TO 6
  IF Sales%(I%) < LowestSale% THEN
```

```
    LowestSale%=Sales%(I%)
    LowestMonth%=I%
  ENDIF
  IF Sales%(I%) > HighestSale% THEN
    HighestSale%=Sales%(I%)
    HighestMonth%=I%
  ENDIF
  Total%+=Sales%(I%)
NEXT I%
REM Print the results
LoMon$=Month$(LowestMonth%)
HiMon$=Month$(HighestMonth%)
PRINT "The lowest sales were in " ;LoMon$
PRINT "The highest sales were in " ;HiMon$
PRINT "The total sales were " ;Total%
END
```

Chapter 15

1) If you wanted to store the grades for five subjects for a pupil along with their first name and surname, can you suggest a structure that would do this?

```
DIM Pupil{FirstName$, Surname$, Grades%(5)}
```

2) Write a program that declares such a structure and prompts for the information. Calculate the average grade and print the results.

```
REM Pupil Report
DIM Pupil{FirstName$, Surname$, Grades%(5)}
REM Collect information
INPUT "Enter first name " Pupil.FirstName$
INPUT "Enter surname " Pupil.Surname$
FOR I%=1 TO 5
  PRINT "Enter grade for subject " ;I%;
  INPUT " " Pupil.Grades%(I%)
NEXT I%
Total=0
FOR I%=1 TO 5
  Total+=Pupil.Grades%(I%)
NEXT I%
```

```
REM Print a report card
PRINT
PRINT "Name: " ;Pupil.Surname$;", " ;
PRINT Pupil.FirstName$
PRINT "Average grade: " ;Total/5
PRINT
END
```

3) If you had 20 pupils in a class, how would you make an array of the above structure?

There are two methods:

```
DIM Pupil{(20) FirstName$, Surname$, \
\              Grades%(5)}
```

or:

```
DIM Pupil{FirstName$, Surname$, Grades%(5)}
DIM Class{(20)}=Pupil{}
```

Chapter 16

1) Modify the PROC_ScreenSetup to accept the background and foreground colour.

```
REM Passing a value to a PROC
PROC _ScreenSetup(5, "First screen" , 0, 130)
INPUT A$
PROC _ScreenSetup(10, "Second screen" , 1, 135)
INPUT A$
REM Restore original colours
PROC _ScreenSetup(0, "" , 0, 128+15)
END
DEF PROC _ScreenSetup(Col%,Title$,Fore%,Back%)
COLOUR Back%
COLOUR Fore%
CLS
PRINT TAB (Col%);Title$
ENDPROC
```

2) *Write and test PROC_Greater(A%,B%) which compares A% and B%. If the A%*

> B%, do nothing, if B% > A%, exchange the two values using a local variable. You'll need to pass by reference so the calling program can print the results.

```
REM Exchange two variables
INPUT "Enter value 1 " Num1%
INPUT "Enter value 2 " Num2%
PROC _Greater(Num1%, Num2%)
PRINT Num1%;" is greater than " ;Num2%
END
DEF PROC _Greater(RETURN A%, RETURN B%)
LOCAL Temp%
REM If greater number is last, swop them
IF B% > A% THEN
  Temp% = B%
  B% = A%
  A% = Temp%
ENDIF
ENDPROC
```

Chapter 17

1) It's a very useful function that waits for the user to press y (yes) or n (no) in response to a prompt and returns either TRUE or FALSE. It should, of course, check for case.

```
REM FN_YesNo
PRINT "Are you sure you want to exit (Y/N)?"
IF FN _YesNo THEN
  PRINT "Fine by me, bye."
ELSE
  PRINT "Sorry, show's over anyway."
ENDIF
END
DEF FN _YesNo
LOCAL Reply$, Return%
REPEAT
  Reply$=GET$
UNTIL INSTR ("YyNn" ,Reply$)<>0
IF Reply$="Y" OR Reply$="y" THEN
```

```
  Return%=TRUE
ELSE
  Return%=FALSE
ENDIF
=Return%
```

2) Write a function FN_Lower that accepts a string. It goes through each character in the string and converts all uppercase letters to lowercase. Other characters are left as they are. Return the converted string.

```
REM FN_Lower
INPUT "Enter a string to convert " MyString$
LoString$=FN _Lower(MyString$)
PRINT "Converted string: " ;LoString$
END
DEF FN _Lower(Convert$)
LOCAL C$, Code%, Return$, I%
Return$=""
FOR I%=1 TO LEN (Convert$)
  REM Get character to work with into
  REM temporary variable, saves code
  C$ = MID$ (Convert$,I%,1)
  REM Test character to see if uppercase
  IF C$>="A" AND C$ <= "Z" THEN
    REM It is, so convert it
    REM Get code for character - offset 'A'
    Code%=ASC (C$)-ASC ("A" )
    REM Add value to offset for 'a'
    C$=CHR$ (ASC ("a" )+Code%)
  ENDIF
  REM Now add character to return string
  Return$+=C$
NEXT I%
=Return$
```

You could also use MID$ to replace the characters in the passed string, like this:

```
DEF FN _Lower(Convert$)
LOCAL C$, Code%, I%
FOR I%=1 TO LEN (Convert$)
  REM Get character to work with into
  REM temporary variable, saves code
  C$ = MID$ (Convert$,I%,1)
  REM Test character to see if uppercase
  IF C$ >= "A" AND C$ <= "Z" THEN
    REM It is, so convert it
    REM Get code for character - offset 'A'
    Code%=ASC (C$)-ASC ("A" )
    REM Add value to offset for 'a'
    MID$ (Convert$,I%,1)=CHR$ (ASC ("a" )+Code%)
  ENDIF
NEXT I%
=Convert$
```

Chapter 18

1) Make the alien walk back across the screen, right to left when it has reached the right-hand side.

```
REM Walking alien
MODE 6
OFF
VDU 23,240,153,189,219,126,36,60,36,36
PRINT TAB (0,10);CHR$ (240)
FOR I%=1 TO 19
  PRINT TAB (I%-1,10);" "
  PRINT TAB (I%,10);CHR$ (240)
  WAIT 25
NEXT I%
FOR I%=19 TO 0 STEP -1
  PRINT TAB (I%+1,10);" "
  PRINT TAB (I%,10);CHR$ (240)
  WAIT 25
NEXT I%
ON
END
```

2) Create another alien with its arms pointing down. Use character 241. Modify the animation from 1) to alternate aliens as it moves across the screen.

```
I% MOD 2
```

will tell you if I% is an odd or even number.

```
REM Walking alien
MODE 6
OFF
VDU 23,240,153,189,219,126,36,60,36,36
VDU 23,241,24,60,219,255,165,189,36,36
PRINT TAB (0,10);CHR$ (240)
FOR I%=1 TO 19
  PRINT TAB (I%-1,10);" "
  IF I% MOD 2 = 0 THEN
    PRINT TAB (I%,10);CHR$ (240)
  ELSE
    PRINT TAB (I%,10);CHR$ (241)
  ENDIF
  WAIT 25
NEXT I%
FOR I%=19 TO 0 STEP -1
  PRINT TAB (I%+1,10);" "
  IF I% MOD 2 = 0 THEN
    PRINT TAB (I%,10);CHR$ (240)
  ELSE
    PRINT TAB (I%,10);CHR$ (241)
  ENDIF
  WAIT 25
NEXT I%
ON
END
```

Chapter 19

1) Create a sound effect that uses short bursts of hiss to generate a noise like a machine gun.

... or a helicopter depending on how you feel.

```
REM Machine gun

FOR  I%=1 TO  5
  SOUND  0,-15,4,1
  SOUND  0,0,0,1
  SOUND  0,-15,4,1
  SOUND  0,0,0,1
NEXT  I%
END
```

2) Here is an incomplete game ...

The missing lines should look something like this:

```
REM Read character in front of player
Ch%=GET (CarX%,1)
REM Off road, lose a life
IF  Ch%<>220 THEN
  SOUND  0,-15,4,5
  Lives%-=1
  WAIT  5
ENDIF
REM Detect player movement
IF  INKEY (-26) AND  CarX%>5 CarX%-=1
IF  INKEY (-122) AND  CarX%<34 CarX%+=1
```

If you think it's too easy, change the WAIT 20 line at the top of the main loop or make the road three characters wide.
My best score is 561 ...

Chapter 20

1) I'm sure you can see lots of improvements here, try adding two more commands, one to completely clear the grid and one to fill it. You could use C and F to do this.

The main problem to be aware of here is how to integrate the new options with the existing program. Looking at the pseudo-code, we can see that there are three routines to change:

PROC_DrawMainScreen, we need to display the new options so the user knows

they're there.

FN_GetUserAction, allow the user to enter two codes, taking care of upper and lower cases and assign an action code.

PROC_ProcessAction, deal with the new actions. All that is required here is to loop through each row and column in the grid, setting or clearing each cell as the choice dictates. Once finished, we need to call PROC_DrawCharacter which will refresh the screen.

Rather than reproduce the whole program, here are the updated routines.

```
REM ****************************************
REM PROC_DrawMainScreen - prints title & help
DEF PROC _DrawMainScreen
REM Set background colour
COLOUR 128+7
CLS
COLOUR 0
REM Print title
PRINT TAB (15,0);"Character Generator"
PRINT TAB (15,1);STRING$ (19,"=" )
REM Print help instructions
PRINT TAB (1,3);"Instructions:"
PRINT TAB (1,4);"Use arrow keys to move cursor."
PRINT TAB (1,5);"Space to toggles selected cell."
PRINT TAB (1,6);"C clears grid, F fills it"
PRINT TAB (1,7);"X or ESC to exits."
ENDPROC
REM ****************************************
REM FN_GetUserAction - returns an action code
DEF FN _GetUserAction
LOCAL Key%,Code%
REPEAT
  REM Wait for keypress
  Key%=GET
  REM Translate key press to action code
  CASE Key% OF
    WHEN 139: Code%=1
```

```
    WHEN 137: Code%=2
    WHEN 138: Code%=3
    WHEN 136: Code%=4
    WHEN 32: Code%=5
    WHEN 67 OR 99: Code%=6
    WHEN 70 OR 102: Code%=7
    WHEN 88: Code%=999
    WHEN 120: Code%=999
    OTHERWISE Code%=0
  ENDCASE
UNTIL Code%<>0
=Code%
REM ****************************************
REM PROC_ProcessAction - actions a valid code
DEF PROC _ProcessAction(Code%)
LOCAL Row%,Col%
CASE Code% OF
  WHEN 1: PROC _MoveCursor(1)
  WHEN 2: PROC _MoveCursor(2)
  WHEN 3: PROC _MoveCursor(3)
  WHEN 4: PROC _MoveCursor(4)
  WHEN 5:
    IF Grid%(Cursor.Col%,Cursor.Row%)=1 THEN
      Grid%(Cursor.Col%,Cursor.Row%)=0
    ELSE
      Grid%(Cursor.Col%,Cursor.Row%)=1
    ENDIF
    PROC _DrawCharacter
  WHEN 6:
    FOR Row%=1 TO 8
      FOR Col%=1 TO 8
        Grid%(Row%,Col%)=0
      NEXT Col%
    NEXT Row%
    PROC _DrawCharacter
  WHEN 7:
    FOR Row%=1 TO 8
      FOR Col%=1 TO 8
```

```
          Grid%(Row%,Col%)=1
        NEXT Col%
      NEXT Row%
      PROC _DrawCharacter
    WHEN 999:
      REM Set exit flag
      Exit=TRUE
  ENDCASE
  PROC _DrawCursor
  ENDPROC
```

2) It's nice to be able to reverse engineer characters too, given the row totals. Modify the project to allow the user to enter a total for the row he's currently on. Then redisplay the character.

This threatens to be a little more complicated. Obviously, there will be modifications to the above three routines again, but how do we achieve the translation of a number into cells of the grid? Let's write the pseudo-code. First we need to get a value to work with:

```
REPEAT
  Prompt for Row Value
  Get a value for the current row
UNTIL Row Value is valid
```

Now we have to convert it into binary. Start with a value of 128, the value of the highest bit in an eight bit number, and compare the row value with it. If the row value is greater than or equal to 128, the top bit must be set so we subtract 128 from the row value and set the correct cell in the grid. After this take the value of the next bit down (64) and try again. We keep going until we have done all eight bits. This looks like a job for a FOR loop to me. Here's what we are trying to achieve:

```
Set Column Value to 128
FOR each column
  IF Row Value >= Column Value THEN
    Set Grid cell at Row, Column
    Subtract Column Value from Row Value
  ELSE
    Reset Grid cell at Row, Column
  ENDIF
  Half Column Value
NEXT column
Draw Character
```

If we write all this into the Process Action routine, it starts getting a little ungainly. We'll give the action a new routine, GetRowValue. We're going to need local variables for the Row Value, Column Value, and a loop counter, Column. This is the complete routine, I added it to the bottom of the program:

```
REM ****************************************
REM PROC_GetRowValue - gets a value for row
DEF PROC _GetRowValue
LOCAL RowValue%,ColValue%, Col%
REM Get a valid value
REPEAT
  PRINT TAB (1,25);SPC (30);TAB (1,25);
  INPUT "Enter value for row: " RowValue%
UNTIL RowValue%>=0 AND RowValue%&lt=255
PRINT TAB (1,25);SPC (30)
REM Convert the number into binary
ColValue%=128
FOR Col%=1 TO 8
  IF RowValue%>=ColValue% THEN
    Grid%(Col%,Cursor.Row%)=1
    RowValue%-=ColValue%
  ELSE
    Grid%(Col%,Cursor.Row%)=0
  ENDIF
  ColValue%=ColValue%/2
NEXT Col%
PROC _DrawCharacter
```

```
ENDPROC
```

Of course there are also the other three routines. Here they are:

```
REM ****************************************
REM PROC_DrawMainScreen - prints title & help
DEF PROC _DrawMainScreen
REM Set background colour
COLOUR 128+7
CLS
COLOUR 0
REM Print title
PRINT TAB (15,0);"Character Generator"
PRINT TAB (15,1);STRING$ (19,"=" )
REM Print help instructions
PRINT TAB (1,3);"Instructions:"
PRINT TAB (1,4);"Use arrow keys to move cursor."
PRINT TAB (1,5);"Space toggles selected cell."
PRINT TAB (1,6);"C clears grid, F fills it"
PRINT TAB (1,7);"V enters a row value"
PRINT TAB (1,8);"Press X or ESC to exit."
ENDPROC
REM ****************************************
REM FN_GetUserAction - returns an action code
DEF FN _GetUserAction
LOCAL Key%,Code%
REPEAT
  REM Wait for keypress
  Key%=GET
  REM Translate key press to action code
  CASE Key% OF
    WHEN 139: Code%=1
    WHEN 137: Code%=2
    WHEN 138: Code%=3
    WHEN 136: Code%=4
    WHEN 32: Code%=5
    WHEN 67 OR 99: Code%=6
    WHEN 70 OR 102: Code%=7
    WHEN 86 OR 118: Code%=8
```

```
    WHEN 88: Code%=999
    WHEN 120: Code%=999
    OTHERWISE Code%=0
  ENDCASE
UNTIL Code%<>0
=Code%
REM ****************************************
REM PROC_ProcessAction - actions a valid code
DEF PROC _ProcessAction(Code%)
LOCAL Row%,Col%
CASE Code% OF
  WHEN 1: PROC _MoveCursor(1)
  WHEN 2: PROC _MoveCursor(2)
  WHEN 3: PROC _MoveCursor(3)
  WHEN 4: PROC _MoveCursor(4)
  WHEN 5:
    IF Grid%(Cursor.Col%,Cursor.Row%)=1 THEN
      Grid%(Cursor.Col%,Cursor.Row%)=0
    ELSE
      Grid%(Cursor.Col%,Cursor.Row%)=1
    ENDIF
    PROC _DrawCharacter
  WHEN 6:
    FOR Row%=1 TO 8
      FOR Col%=1 TO 8
        Grid%(Row%,Col%)=0
      NEXT Col%
    NEXT Row%
    PROC _DrawCharacter
  WHEN 7:
    FOR Row%=1 TO 8
      FOR Col%=1 TO 8
        Grid%(Row%,Col%)=1
      NEXT Col%
    NEXT Row%
    PROC _DrawCharacter
  WHEN 8: PROC _GetRowValue
  WHEN 999:
```

```
    REM Set exit flag
    Exit=TRUE
ENDCASE
PROC _DrawCursor
ENDPROC
```

Appendix F - Character Designer Listing

```
REM  *********************************
REM * User defined character designer *
REM * Version 1.0            2/1/2006 *
REM * Peter Nairn                     *
 REM *********************************

REM Declare arrays and structure
 DIM Grid%(8,8)
DIM RowTotal%(8)
DIM Cursor{Row%,Col%,LastRow%,LastCol%}

PROC _Init
PROC _DrawMainScreen
PROC _DrawCharacter
PROC _DrawCursor

REPEAT
   Action%=FN _GetUserAction
  PROC _ProcessAction(Action%)
UNTIL Exit

PROC _Shutdown

END

 REM  ************************************************
REM PROC_Init - initial setup when program is started
 DEF PROC _Init

REM Setup graphics mode
 MODE 21

REM Setup user characters 240, 241 and 242
 VDU 23,240,0,0,0,0,0,0,0,0
VDU 23,241,255,129,129,129,129,129,129,255
VDU 23,242,255,255,255,255,255,255,255,255

REM Setup cursor location
 Cursor.Row%=1
Cursor.LastRow%=1
Cursor.Col%=1
```

```
Cursor.LastCol%=1

REM Set Exit flag
 Exit=FALSE

 REM Turn text cursor off
 VDU 23,1,0;0;0;0;

ENDPROC

 REM  ************************************************
REM PROC_DrawMainScreen - prints static text,
REM                       title, help etc.
 DEF PROC _DrawMainScreen

REM Set background colour
 COLOUR 128+7
CLS
COLOUR 0

REM Print title
 PRINT TAB( 15,0);"Character Designer"
PRINT TAB( 15,1);STRING$( 19,"=" )

REM Print help instructions
 PRINT TAB( 1,3);"Instructions:"
PRINT TAB( 1,4);"Use arrow keys to move cursor."
PRINT TAB( 1,5);"Press space to toggle selected cell."
PRINT TAB( 1,6);"Press X or ESC to exit."

ENDPROC

 REM  ************************************************
REM PROC_DrawCharacter - called when character changes
REM                      to redisplay it
 DEF PROC _DrawCharacter
LOCAL ColValue%,Row%,Col%

REM Draw grid and calculate totals for each row
 FOR Row%=1 TO 8
  ColValue%=128
  RowTotal%(Row%)=0
  FOR Col%=1 TO 8
    IF Grid%(Col%,Row%)=0 THEN
      PRINT TAB( 18+Col%,10+Row%);CHR$ (241)
    ELSE
      PRINT TAB( 18+Col%,10+Row%);CHR$ (242)
      RowTotal%(Row%)=RowTotal%(Row%)+ColValue%
```

```
    ENDIF
     ColValue%=ColValue%/2
  NEXT Col%

  PRINT TAB( 28,10+Row%);"    "
  PRINT TAB( 28,10+Row%);RowTotal%(Row%)
NEXT Row%

REM Draw actual character
 VDU 23,240,RowTotal%(1),RowTotal%(2),RowTotal%(3), \
\         RowTotal%(4),RowTotal%(5),RowTotal%(6), \
\         RowTotal%(7),RowTotal%(8)
FOR Col%=1 TO 8
  PRINT TAB( 18+Col%,20);CHR$ (240)
NEXT Col%

REM Print VDU codes
 PRINT TAB( 7,22);"BASIC code to produce this character: "
PRINT TAB( 1,24);STRING$( 50," " )
PRINT TAB( 4,24);"VDU 23,240" ;
FOR Row%=1 TO 8
  PRINT "," ;STR$ (RowTotal%(Row%));
NEXT Row%

ENDPROC

 REM  ************************************************
REM PROC_DrawCursor - delete cursor from old position
REM                   and redraw in new
 DEF PROC _DrawCursor

REM Set to normal colour and erase old cursor
 COLOUR 0
PRINT TAB( 18+Cursor.LastCol%,10+Cursor.LastRow%);
IF Grid%(Cursor.LastCol%,Cursor.LastRow%)=0 THEN
  PRINT CHR$ (241)
ELSE
  PRINT CHR$ (242)
ENDIF
 REM Set to highlight colour and draw cursor
 COLOUR 1
PRINT TAB( 18+Cursor.Col%,10+Cursor.Row%);
IF Grid%(Cursor.Col%,Cursor.Row%)=0 THEN
  PRINT CHR$ (241)
ELSE
  PRINT CHR$ (242)
ENDIF
 REM Set back to normal colour
```

```
 COLOUR 0

ENDPROC

 REM  ************************************************
REM FN_GetUserAction - returns a valid action code
REM                    when selected by user
 DEF FN _GetUserAction
LOCAL Key%,Code%

REPEAT
   REM Wait for keypress
   Key%=GET
   REM Translate key press to action code
   CASE Key% OF
    WHEN 139: Code%=1   :REM Up
     WHEN 137: Code%=2   :REM Right
     WHEN 138: Code%=3   :REM Down
     WHEN 136: Code%=4   :REM Left
     WHEN 32: Code%=5    :REM Toggle - space
     WHEN 88: Code%=999  :REM Exit
     WHEN 120: Code%=999
    OTHERWISE Code%=0
  ENDCASE
UNTIL Code%<>0

=Code%

REM  ************************************************
REM PROC_ProcessAction - actions a valid code
 DEF PROC _ProcessAction(Code%)

CASE Code% OF
  WHEN 1: PROC _MoveCursor(1)
  WHEN 2: PROC _MoveCursor(2)
  WHEN 3: PROC _MoveCursor(3)
  WHEN 4: PROC _MoveCursor(4)
  WHEN 5:
    IF Grid%(Cursor.Col%,Cursor.Row%)=1 THEN
       Grid%(Cursor.Col%,Cursor.Row%)=0
    ELSE
       Grid%(Cursor.Col%,Cursor.Row%)=1
    ENDIF
    PROC _DrawCharacter
  WHEN 999:
    REM Set exit flag
     Exit=TRUE
ENDCASE
```

```
PROC _DrawCursor

ENDPROC

 REM  ************************************************
REM PROC_MoveCursor - move cursor in direction given:
REM                   1-Up, 2-Right, 3-Down, 4-Left
 DEF PROC _MoveCursor(Dirn%)

REM Save current position in old position
 Cursor.LastRow%=Cursor.Row%
Cursor.LastCol%=Cursor.Col%

REM Check limits and move
 CASE Dirn% OF
  WHEN 1: IF Cursor.Row%>1 Cursor.Row% -=1
  WHEN 2: IF Cursor.Col%<8 Cursor.Col% +=1
  WHEN 3: IF Cursor.Row%<8 Cursor.Row% +=1
  WHEN 4: IF Cursor.Col%>1 Cursor.Col% -=1
ENDCASE

ENDPROC

 REM  ************************************************
REM PROC_Shutdown - tidies up before exiting
 DEF PROC _Shutdown

REM Re-enable text cursor
 VDU 23,1,1;0;0;0;
REM Set cursor to bottom of screen
 PRINT TAB( 0,26);

ENDPROC
```