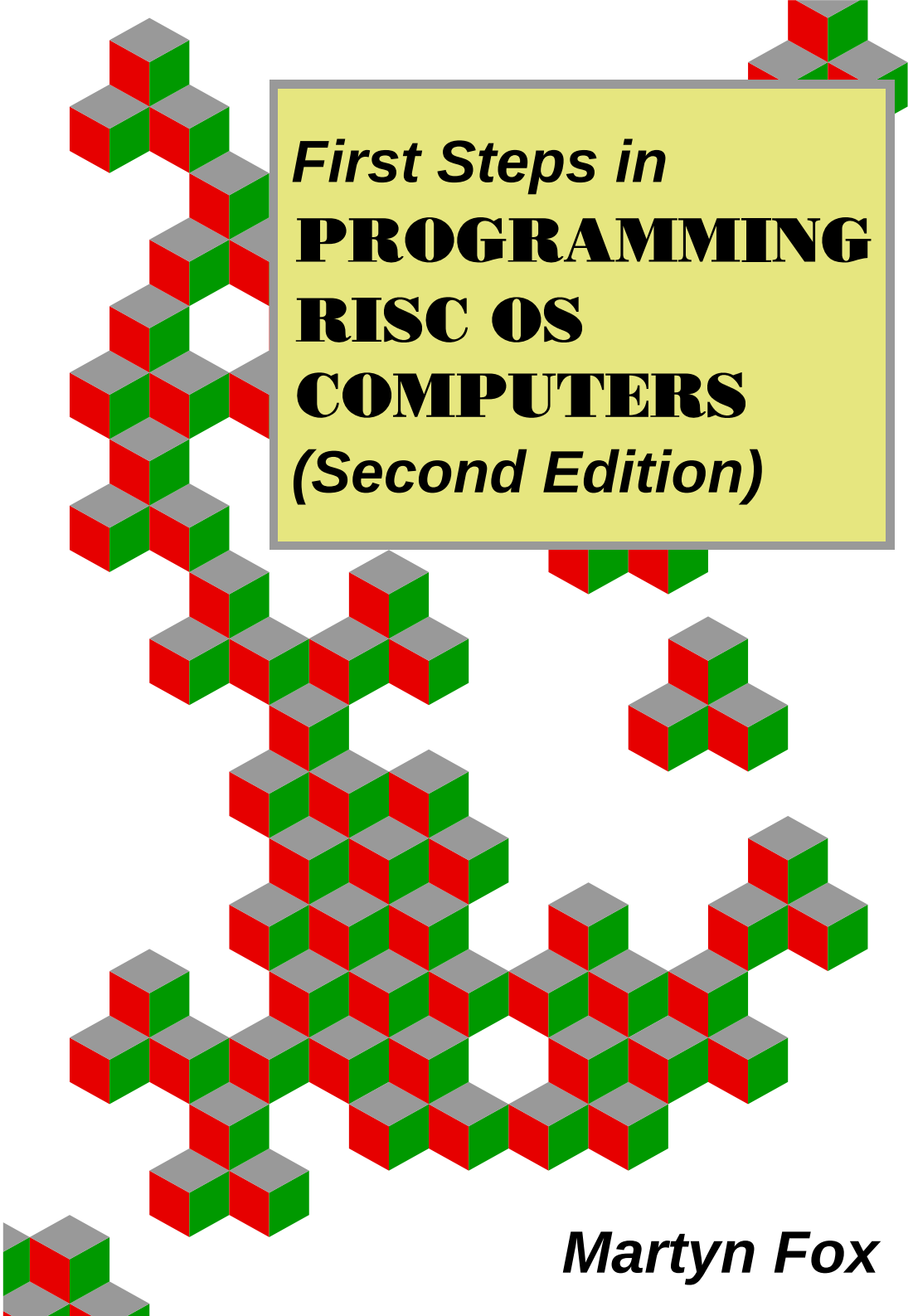


First Steps in Programming RISC OS Computers

Second Edition

Martyn Fox

Preface to Second Edition

Congratulations on buying your RISC OS computer! By the time you get to this guide, you'll probably have used a variety of software applications which came with the machine, you bought yourself or downloaded from the Internet.

Obviously this software didn't write itself! Somebody sat down at a computer very similar to yours and created it. The major packages would have been written by professional programmers, perhaps working for software houses, but a lot has been created by people who simply write programs for pleasure.

Programming for pleasure? Certainly. There is plenty to be derived from creating something in this way. It costs virtually nothing to do, once you have your computer, as all it needs is your time, electricity to run the machine and some space on your hard disc to store the result. If you don't like what you've written, you can always just delete it and try again.

Think of it as a kind of DIY without having to buy any raw materials. When you have finished you will experience the deep satisfaction of possessing something that you created yourself, especially if you can achieve the 'feel' of a professional product. What you have is unique (unless you've given someone a copy!) and you are the one who understands its inner workings and who could modify it if you wanted to.

This guide introduces you to your machine's built-in programming language called Basic, which makes it easy to write programs. We'll start with simple things like putting some text on the screen and work our way up to writing a complete game and a useful database program.

You're probably aware that one thing which sets your RISC OS computer apart from other types is its excellent desktop which uses the Wimp system. Wimp stands for **W**indows, **I**cons, **M**enus and **P**ointers and allows you to run several programs at the

same time − called *multi-tasking* − using windows to share the screen. Writing programs which do this is a trifle complicated but you can find out how to do it in another guide, *A Beginner's Guide to Wimp Programming*.

The first edition of this guide was published under the title *First Steps in Programming Acorn RISC OS Computers* in 1993 in book form by Sigma Press, Wilmslow, Cheshire. If you have a copy, hang onto it! It could be something of a collector's item now as only 1,000 copies were printed. The world of RISC OS computers has moved on a lot since then, though the Basic programming language has remained largely unchanged. This guide has been revised so that it may be followed from the screen and also to take account of the fact that there is now a wide diversity of RISC OS computers in use with differing capabilities where screen modes are concerned. In addition, all the software is available to you electronically. Although the programs are listed in the guide, you'll probably wish to just follow them from the listings and run them, rather than type them in yourself.

Thanks are due to my brother Robert for help with the Munchie game, especially the design of the sprites, and to Richard Hallas for his work on the conversion from *Impression* to *Ovation Pro* format.

Martyn Fox

Contents

Introduction

How do you go about learning to program? Isn't it necessary to learn lots of complicated code numbers and things like that? Not on this machine. It may seem very complex, with pretty icons whizzing about all over the screen, but getting started is actually very simple indeed, thanks to the *language* that is built into the machine. What this means is that you can give it instructions in a form remarkably similar to English. This language is called Basic, and because your machine has an enhanced version of the type of Basic originally produced for the BBC microcomputer which was first manufactured by Acorn Computers Ltd. in the early 1980s, its correct name is BBC Basic.

What is RISC OS?

You will still find frequent references to Acorn computers in magazines and software adverts, and may even occasionally find a reference to the 'Archimedes'. This was the name given to the first generation of Acorn 32-bit computers. You might even be using one now, as they do last a long time. Rest assured that all the software described in this guide should run on it.

Acorn themselves ceased developing and manufacturing computers in 1998 but Acorn-badged machines continued to be produced by Castle Technology Ltd. Computers using the same technology were also developed by other companies such as RiscStation and Microdigital.

All these machines, from the original Archimedes onwards, have the same operating system. This is the software which is built into the machine and makes it go. Because it is stored on a ROM (Read Only Memory) chip, it's there all the time and starts working as soon as you switch on the machine. This operating system is called RISC OS (**R**educed **I**nstruction **S**et **C**omputer **O**perating **S**ystem). Basically, any software that runs on a RISC OS machine should run on yours, unless it has some special

requirement and, likewise, anything you write should run on another RISC OS machine. All the programs in the original printed book version of this guide, and indeed the book itself, were produced on a second-hand Archimedes A310, which was manufactured in 1987.

The most likely special requirement that software might have is memory size. A lot of major packages require a minimum of 4 megabytes (written 4 Mb) of RAM (Random Access Memory). Software for handling graphics frequently requires a lot more.

None of the software that we'll be creating requires anything like even 1Mb however, so this needn't trouble us.

This guide doesn't cover the writing of programs using windows and menus. As we saw in the preface, these are rather complicated and best left until you have mastered the Basic language. When you've mastered everything in this guide, you'll be ready to tackle *A Beginner's Guide to Wimp Programming* which will teach you how to create professional-looking applications. We will, however, create a simple multi-tasking 'front end' for a program which will run in the desktop environment, with its icon on the icon bar.

Plumbing Hidden Depths

The machine's desktop, with its windows, menus and the mouse pointer doesn't seem to offer us much opportunity to give it instructions, so how do we go about it?

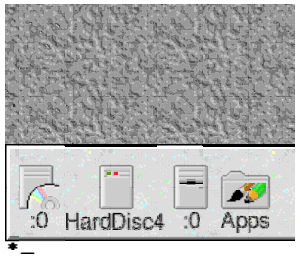
You can write a Basic program using a text editor, such as Edit (which is built into the machine) or the freely-available applications Zap and StrongEd. We'll be doing that shortly, but we'll take our first steps at a lower level.

We're going to start by using Basic's *command level*, also known as *immediate mode*. In this mode, when you type in a command for Basic to do something, it does it immediately, without you having to save a program and run it.

If you're following this guide on the screen, rather than on paper, you'll have to remember the next bit while you try it out:

Press the right-hand function key, labelled F12. The desktop will move up the screen and a white line will appear at the bottom. The desktop will have turned into a frozen image and the mouse pointer will have vanished.

On the left-hand side of the white line will be a black star. That star is the key to plumbing hidden depths in your machine. The flashing black line to the right of it is the *cursor* – it is an invitation to type something in and shows where it will appear on the screen.



... a white line will appear at the bottom

For the moment, though, how do we get back to the nice, friendly desktop? Don't panic, just press the Return key. Provided you didn't type in anything following the star, the desktop will return to normal and the pointer will reappear. If you did type in something, you will probably get a message saying 'File not found', or saying that you made some sort of mistake, followed by another star. In this case, just press Return again. Try it.

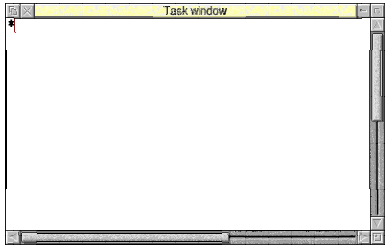
Are you back with us? That line with the star at the beginning of it was the machine's command line. The commands that we type in here, because they are preceded by a star, are known as *Star Commands*.

You can't be expected to memorize everything that this guide tells you to do on the command line, though, so, if you're reading it off the screen, you need another way of using the command line; one which doesn't stop the desktop working.

The Task Window

If you press Ctrl-F12, a window will appear on the screen labelled 'Task Window', with the star prompt in the corner. In place of the flashing black cursor will be the red caret, probably familiar to you from word processing and writable icons.

This is actually a window belonging to the Edit application, which is why the Edit icon will have appeared on the icon bar. It's different, though, because the operating system has started up a new application and this window is linked to it.



'Task Window', with the star prompt in the corner

1

Making the Machine Do Something

You now have two ways of using the command line. Which one you use will depend on your personal preference and whether or not you need to read something on the screen at the same time.

The simplest command of all is one to make the machine tell us which version of the operating system it contains. To do this, type:

```
fx0
```

following the star, so that the whole line reads:

```
*fx0
```

The machine doesn't take any notice of whether the letters in star commands are capitals or lower case. Note, though, that the '0' is the figure nought, not the letter O.

After you have typed this in, press Return (or the 'Enter' key on the right of the numeric keypad). All commands and lines of programming are followed by a press of the Return key, unless we say otherwise. This tells the machine that the line is finished, and that it should *execute* it, that is, do what the instruction in the line tells it to.

A line will appear on the screen saying:

```
RISC OS 4.02 (10 Aug 1999)
```

or something similar, followed by another star on the next line. If you get something different, it means you mis-typed the command.

You have given the machine your first command and it has, hopefully, done what you told it to. If you received some other kind of message, it probably means that you made a typing error, in which case, try again.

Now type:

```
Cat
```

so that the entire line reads:

```
*Cat
```

From now on, we will always show star commands with the star at the beginning. Of course, if there is already a star on the screen, you won't need to type one in, though it doesn't matter if you do.

When you press Return, your hard disc will run and a list of the files and subdirectories in its root directory will appear on the screen. The command you have just given is the one to *Catalogue* the disc.

Getting Into Basic

Now that you've got the hang of typing in simple commands, it's time to investigate Basic, the language we will use for programming.

Type:

```
*Basic
```

The machine will reply with a message saying:

```
ARM BBC BASIC V version 1.20 (C) Acorn 1989
Starting with 651516 bytes free
>
```

This message shows that Basic has now started up. The figure on the second line shows the amount of free memory that Basic can use. This depends on how much memory your machine has and how much of it the desktop was going to allocate to your next application.

You will notice that there is no longer a star inviting you to enter a star command. This is because you are no longer using the command line – you are now in Basic. The '>' character is the *Basic prompt*, inviting you to type in a command that Basic will understand.

Type a number of characters at random, then press Return. Almost certainly the result will be a message saying 'Mistake'. This is an *error message*, which means that the machine couldn't understand what you meant.

Introducing Keywords

Basic understands certain words, known as *Keywords*. There are over 100 of these and you'll find a full list of them and what they do in Appendix 1. They always have to be in capitals, for a reason which we will find out shortly. Turn on the CapsLock key, then type:

```
PRINT "HELLO"
```

Take care to use the double-quote (") rather than the single-quote(') symbol. Because it is not on a letter key, you'll have to press Shift to get it, even though CapsLock is on.

When you press Return, the word HELLO should appear on the next line and the following line will contain the Basic prompt. If you receive an error message instead, it probably means that you made a typing error, possibly leaving out one of the quotes.

... the word HELLO should appear on the next line ...

The command PRINT in Basic doesn't mean 'Send to the printer' but 'Put on the screen'. This is for historical reasons. Basic was invented before the microchip and at that time there were no visual display units for computers – all communication from the computer to its operator had to be via a printer. Today, the command PRINT means 'Display on the screen'.

The command that you typed in tells Basic to print whatever is between the quotes, which is why the word HELLO appeared on the next line. Of course, if you put something else between the quotes, Basic will print that. Try it.

Storing Up a Program

When you have finished this little experiment, type in the following:

```
10PRINT "HELLO"
```

You will notice that this is the same command as before, only this time we put a number in front of it. Now when you press Return, nothing happens, except that the Basic prompt reappears. The machine hasn't ignored your command. Because it started with a number, it has stored it away to use later.

Everything we typed in before our latest line was on Basic's command level, which means that Basic executes it immediately. If the command begins with a number, however, it is stored in the memory for later use.

To prove it, type:

```
LIST
```

This tells Basic to show you what it has stored. You should find that it repeats your command back to you, including the number 10 at the beginning.

This is the whole secret of the way computers can be made to do complicated things – the *stored program*. A program consists of a series of lines of instructions which tell the machine what to do and it follows them in succession.

At the moment, our program only consists of one line, so let's add a second. Type in:

```
20PRINT "GOODBYE"
```

If you now type LIST again, you should see the following:

```
10PRINT "HELLO"
20PRINT "GOODBYE"
```

Basic has LISTed your program to show you what's in it.

Now to make it do your bidding! Type:

```
RUN
```

You should find that the screen displays:

```
HELLO
GOODBYE
>
```


Basic has gone through your program, starting with the first line, which tells it to print HELLO, then the second, telling it to print GOODBYE. After it has done this, it finds there are no more lines to deal with, so it prints the Basic '>' prompt and returns control to you.

Although we entered the lines in the order in which they were to be executed, we didn't necessarily have to do so. When you type in a line, Basic knows where to put it in the program by the number at the beginning, which is, of course, known as the *line number*. To prove it, type:

```
15PRINT "HELLO AGAIN"
```

or put whatever you like between the quotes.

If you type LIST again, you should see:

```
10PRINT "HELLO"
15PRINT "HELLO AGAIN"
20PRINT "GOODBYE"
```

and if you type RUN, you should get:

```
HELLO
HELLO AGAIN
GOODBYE
```

You can see now why program lines are usually numbered in tens; it makes it much easier to insert extra lines into the program!

Each line in this program contains just one instruction, frequently known as a *statement*. You can have more than one statement on one line if you wish, by separating them with a colon (:), for example:

```
10PRINT "HELLO":PRINT "HELLO AGAIN"
```

If you want to change a line, simply type it in again, using the same line number. You can delete a line entirely, simply by typing its number and pressing Return. Try a few experiments and list the program each time to see the result.

If you made a typing error, you will probably get an error message when you run the program. The message will tell you not only what kind of error it was, but on which line it was detected. Note by the way that this is not necessarily the line on which the error occurred, for reasons we will see later.

By the way, PRINT on its own prints a blank line and a single quote (') in a PRINT statement sends the printing operation to the start of the next line. You could have written line 10 in the example above as:

```
10PRINT "HELLO"'"HELLO AGAIN"
```

The Case of The Missing Variable

If you had an error saying ‘Unknown or missing variable’, it means that you missed out the quotes round the word, or words, that followed the PRINT command. We will find out what a variable is in the next section.

If you’re using the command line at the bottom of the screen, you may now be wondering how you can return to the desktop without a star prompt to press Return on. The answer is to type the Basic command:

```
QUIT
```

This will make Basic close down and you will get the star prompt back.

Alternatively, you can type:

```
*Quit
```

from within Basic. This is a star command and you can use any star command when you’re in Basic, provided you type the star first. This tells Basic that what follows is not for it to deal with; instead it is handed over to the main part of the operating system.

Whichever you choose, you should now have the star prompt, with nothing following it. A further press of Return will take you back into the desktop.

If you’re using the task window, just close the window. You will get a prompt asking you to confirm that you wish to discard the task running in the window.

We referred earlier to ‘Read Only Memory’ (ROM) and ‘Random Access Memory’ (RAM) so, before we go any further, let’s just make sure we understand what these terms mean.

Another history lesson here. When computers were in their infancy, there were no microchips. To construct a memory device capable of holding lots of numbers and producing them in any desired order would have required thousands of transistors (and before that even valves!). Such devices were very expensive and could only hold a small amount of data by modern standards. Larger amounts of storage would have been on magnetic discs or tapes, which could only be written to or read from in a fixed order.

Random Access Memory

The small quantity of memory which could store or produce numbers in any order would have been known as *Random Access Memory*, to distinguish it from the other, less flexible, type. Today, all memory chips that can be written to as well as read from are known by this name.

Random access memory has one big disadvantage − it is *volatile*. This means that the numbers stored in it vanish if the power is switched off, even for a moment. If a computer contained only this type of memory, it couldn’t work because its processor would not know what to do when it was switched on. It wouldn’t even be able to load instructions from a disc!

Read Only Memory

A computer needs an operating system to tell it what to do, at least when it’s first switched on. So that the operating system is there all the time, it needs a different kind of chip, one whose contents are permanently fixed into it. Such a device is known as a *Read Only Memory* chip. Once the numbers have been ‘blown’ into it, they can’t normally be changed (though some types can be erased and re-programmed), which is why it can be read from but not written to. On the other hand, it doesn’t lose its contents when the power goes off!

Using the term ‘random access memory’ to mean read/write memory is a bit of a misnomer because read only memory is random access as well. Both types of chip can produce their contents in any order. It’s the convention in computing, though, that ROM means read only memory chips (of course) and RAM means volatile read/write memory chips.

Some other types of computer, notably the IBM-compatible PC, have the bulk of their operation system stored on a disc with just enough of it in a ROM to be able to load the rest into the RAM. Although this makes it easier to upgrade to a newer version of the operating system, it means it has to be loaded every time you start up the machine as well as any time you reset it, possibly because the program has crashed. There is also a risk of the operating system itself being corrupted by a virus.

Your RISC OS machine, on the other hand, has its entire operating system in a ROM. This means not only that it’s there as soon as you turn on or reset the machine, requiring just a few seconds to set itself up, but that you have a large operating system which uses very little RAM. It’s also much less susceptible to viruses!

What is RAM Used For?

Anything that you load into your computer goes into the RAM. This could be a Basic program, any kind of application software, whether written in Basic or machine code, sprites or data of any kind.

The operating system uses part of the RAM to store the numbers that it works with, as does any program which is running. Some software makes use of *relocatable modules*, some of which are contained in the operating system ROM and some of which are loaded from a disc into a part of the RAM known as the *Relocatable Module Area*, or RMA. If you use a wordprocessor or desktop publishing package, you're using outline fonts. These use an area of RAM known as the *font cache*.

Finally, everything on the screen has to be stored in a section of RAM which can also be read by the video controller chip that generates the picture on your monitor. How much RAM this uses depends on the resolution of your screen and the number of colours it can display.

You can see how your RAM is being used by clicking the mouse on the icon in the bottom right-hand corner of the screen and looking at the Task Manager window.

2

Introducing Variables

Before we go any further, let's just sort out one or two things.

You've probably grown a little tired of typing in the word PRINT over and over again and wished there was a shorter way of doing it. Fortunately there is!

Both Basic and the command line interpreter recognise *minimum abbreviations*. This means that you only have to type in enough of a keyword for it to be distinct from any other keyword, then follow it with a full stop (.). If the abbreviation is short for more than one keyword, Basic replaces it with the one which is first on its list, which is usually the most-used one.

The minimum abbreviation for PRINT is P. which certainly saves a lot of typing! Similarly, the minimum abbreviation for MODE is MO.

The command '*Cat' which we used first of all to catalogue the disc has the shortest abbreviation of all:

```
*.
```

All the listings in this guide will be shown with keywords in full, to make their meanings clear. By all means use minimum abbreviations – they speed up entering your program, as well as saving your fingers!

When you're using the command line (but not the task window), another technique that can mean less typing is using the *cursor edit keys*. If you are typing in a line which is similar to one already on the screen, you can use the arrow keys to allow you to copy part of what you already have. Suppose, for example, you were entering our earlier program:

```
10PRINT "HELLO"
20PRINT "GOODBYE"
```

After entering the first line, just type ‘20’ then press the up-arrow key. The cursor will appear to move up one line, leaving behind a solid block. What has happened is that you now have two cursors. The block is in fact now the *write cursor* and the underline character the *read cursor*. Use the arrow keys to position the read cursor underneath the first of the characters you wish to copy, in this case the start of the word PRINT, and press the Copy key. The character above it will be copied to the position of the write cursor and both cursors will move one position to the right, allowing you to copy the next character. In this way, you can easily copy a chunk of your listing without having to type it in again.

you now have two cursors

What are Variables?

The previous section contained a brief reference to variables. A variable is a number which is referred to by a name. We give it a value when we first create it, and we may change it during the program.

Enter Basic and try typing:

```
x=3
```

We have created a variable called simply ‘x’ and given it a value of 3. Now try another one:

```
y=x+2
```

Basic works out the value of the right-hand side of the equation, that is the part following the equals sign, and makes this the value of a new variable, called ‘y’.

Because the value of x is 3, it doesn’t take a mathematical genius to see that the value of y must be 5, i.e. 3+2. You can prove this by typing:

```
PRINT y
```


Basic always works out the value of whatever follows the word PRINT before printing it, unless it is in quotes. For example:

```
PRINT 2+3
```

will print:

```
5
```

but:

```
PRINT "2+3"
```

will print:

```
2+3
```

This is why the words HELLO etc. which we printed in the previous section had to be in quotes. Without them, Basic would have thought that HELLO was the name of a variable and given you an error message saying ‘Unknown or missing variable’.

Choosing Names for Variables

This is because a variable name doesn’t have to be a single letter. In fact it is best if it’s a word, chosen to describe what the variable does. Any group of letters and numbers can be used. Spaces are not allowed, but you can use an underline symbol (_) instead. The name must start with a letter and must not begin with a Basic keyword. You couldn’t, for example, use TOTAL as a variable name, because ‘TO’ is a keyword. You can however, use ‘total’.

This example shows the best way of avoiding the problem. All Basic keywords are in capitals. If you always put variable names in lower case, there will be no chance of one clashing with a keyword. Another advantage is that it makes the program a lot easier to read, as all keywords are in capitals and all variable names are lower case.

Types of Variable

There are three types of variable: *floating point*, *integer* and *string*.

Floating point variables are used to represent numbers that may contain fractions, for example:

```
size=1.25
```

Note by the way that a full stop is used as a decimal point in Basic, as it is elsewhere on the computer.

Integer variables are used to represent whole numbers. An integer variable has a name ending in a '%' sign, for example:

```
size%=20
```

If you try to put a fraction into an integer variable, it will be rounded down to the nearest whole number.

There are three advantages of using integer variables:

- They are handled faster
- They use less memory for storage
- They are completely accurate – a number that should be 10 will not end up as 9.99999999

It is always best to use integer variables if you can. Only use a floating point variable if its value is likely to contain a fraction, or be outside the limits −2147483648 to 2147483647, which integer variables can't handle.

Automatic Line Numbering

We'll look at string variables shortly, but first let's try a very simple program working with numbers. Enter Basic and type:

```
AUTO
```

This will put the line numbers on the screen to save you having to type them at the beginning of each line. If you don't add any figures after AUTO, the line numbers will increase by 10 each time you press Return. Using this facility, type in the following:

```
10PRINT "What is the first number?"
20INPUT first%
30PRINT "What is the second number?"
40INPUT second%
50PRINT first%+second%
60GOTO 10
```

When you have finished, press Esc to get out of the AUTO facility. If you make a mistake and spot it before pressing Return, you can use the backspace key to delete back to it and try again. If you don't notice it until after pressing Return, you'll have to retype the line.

You can restart automatic line numbering at any point you like. If you had stopped, for example, after line 30, just type:

```
AUTO 40
```


to continue. If you want the line numbers to increase in steps other than 10, for example 5, type:

```
AUTO 40,5
```

Using INPUT

The first line of our program prints a message asking you for a number. Line 20 introduces the basic keyword INPUT. This makes the program wait while you type in a number, then makes it the value of the variable called first% when you press Return.

Lines 30 and 40 repeat the procedure so that you can enter a second number, making it the value of second%, then line 50 adds them together and prints the result.

The command GOTO in line 60 is all one word, without a space in the middle. It tells the program to jump back to line 10 and continue from there.

We've used the GOTO command here to avoid introducing too many new concepts in one go. It's not a good command to use, for reasons which we'll find out later, and this is the only place in this guide where it will be used.

When you run the program, it will ask you for a number, wait while you type it in, then ask you for the second one. When you have entered that one, it will print the sum of the two and ask you for a new first number – it has jumped back to the beginning and started again. The best way out of this program is to press Esc.

Errors Involving Variable Names

Lines 20 and 40 in the program we've just looked at create the two variables first% and second% by giving them values. They don't exist until these two lines are executed. Line 50 uses them but they have to exist already for this line to work. If for any reason one of them didn't, you would get an error message saying:

```
Unknown or missing variable at line 50
```

Suppose you had made a mistake typing in line 20, so that it read:

```
20INPUT frist%
```

You would still get the same error message saying that there was a mistake in line 50, although there would be nothing wrong with line 50. The trouble is that a variable has been created in line 20 as intended, but it's been called 'frist%' instead of 'first%' (Basic doesn't know you've made a spelling mistake!). When the program gets to line 50, it looks for a variable called 'first%' and, of course, it can't find it.

The number in an error message tells you the line where the error was *detected*, which is not necessarily where it occurred.

String Variables

The third type of variable is a *string variable*. This type doesn't represent a number, but a string of characters. Its name ends in a dollar ($) sign. An example would be:

```
name$="Fred"
```

String variables can be *concatenated* or added together, which means that the strings of characters themselves are joined together. Try this:

```
first_name$="Fred"
last_name$="Smith"
full_name$=first_name$+last_name$
PRINT full_name$
```

You should get:

```
FredSmith
```

Of course, a space between the first and last names would be nice. You could get one by modifying the third line to read:

```
full_name$=first_name$+" "+last_name$
```

There is a space between the quotes.

Using Parts of Strings

As well as joining strings together, you can obtain parts of them, using the keywords LEFT$, MID$ and RIGHT$.

LEFT$ lets you take some characters from the left-hand end of the string. It is followed by the name of the string variable and the number of characters in brackets, for example:

```
word$="ABCDE"
PRINT LEFT$(word$,3)
ABC
```

If you omit the number of characters, the result is the original string minus its last character:


```
words$="ABCDE"
PRINT LEFT$(word$)
ABCD
```

RIGHT$ does the same thing at the right-hand end of the string:

```
name$="ABCDE"
PRINT RIGHT$(name$,3)
CDE
```

In this case, if you omit the number of characters, you get the last character of the string:

```
name$="ABCDE"
PRINT RIGHT$(name$)
E
```

MID$ allows you to take one or more characters from anywhere in the string. Like LEFT$ and RIGHT$, it is followed by the name of the string in brackets, but there are now two numbers, telling us the position in the string of the first character and the number of characters that we want, for example:

```
name$="ABCDE"
PRINT MID$(name$,2,3)
BCD
```

The first of these two numbers is always needed but if you omit the second one, you get all the characters to the end of the string:

```
name$="ABCDE"
PRINT MID$(name$,2)
BCDE
```

In all cases, you can use a variable in place of a number.

LEFT$, RIGHT$ and MID$ can also be used to replace part of a string, for example:

```
LEFT$(a$,3)=b$
```

The first three characters of a$ are replaced by the first three characters of b$, or all of b$, if it is shorter than this. Similarly:

```
RIGHT$(a$,3)=b$
```

replaces the last three characters with the first three of b$ and you can do a similar operation using MID$ (try it). None of these operations alters the length of a$.

You can find out the length of a string, that is the number of characters in it, by using the keyword LEN, for example:

```
word$="ABCDE"
PRINT LENword$
5
```

3

Loops, Repeats and Conditions

We are now starting to type in longer listings with more lines. To make them easier to read, all the listings from now on will include a space after each line number.

Some lines will have more than one space after the number − they are *indented* to show the start and finish of sections of the program such as *loops* which are explained in this section. Again, this is for clarity and to make the workings of the program easier to understand.

You don't have to type in these spaces, though it doesn't matter if you do. You can see them when you list the program by first typing:

```
LISTO 3
```

The LISTO command is fully explained in Appendix 1. If a line has more than 80 characters, it is shown as though it's being displayed on an 80 column screen.

Deciding IF to Execute

The program in Section 2 which added two numbers together kept repeating the same lines over and over again, doing the same thing every time.

Suppose, though, we wanted the program to do one thing if we typed in one number and a totally different thing if we typed in something else. We could use the IF keyword.

If you already have a program in your machine's memory, get rid of it by typing:

```
NEW
```

If you should change your mind, you can get it back, provided you haven't started to

enter a new program, by typing:

```
OLD
```

Now type in the following:

```
10 INPUT "What is the first number",first%
20 INPUT "What is the second number",second%
30 IF first%>second% THEN PRINT "The first number was higher"
40 IF first%<second% THEN PRINT "The second number was higher"
50 IF first%=second% THEN PRINT "The numbers were equal":x%=1
60 IF first%<>second% THEN PRINT "The numbers were not equal":x%=2
```

Notice the use of the symbols '>', '<', '=' and '<>' between first% and second% in lines 30 to 60.

The first two lines show how INPUT can be used to print a line on the screen and enter a number into a variable at the same time. The program will print the line, then wait for you to type in the number. The comma, incidentally, causes a question mark to be printed at the end of the line. If you don't want a question mark, leave out the comma.

Conditions Attached

The IF keyword causes *conditional execution*. Basic checks the part which follows the keyword to see if it's true. If it is, it carries out the instruction following the THEN. Line 30, for example, means:

> If the value of first% *is greater than* the value of second% then print 'The first number was higher'.

Line 40:

> If the value of first% *is less than* the value of second% then print 'The second number was higher'.

Line 50:

> If the value of first% *is equal to* the value of second% then print 'The numbers were equal' and give x% a value of 1.

Line 60:

> If the value of first% *is not equal to* the value of second% then print 'The numbers were not equal' and give x% a value of 2.

If you want the program to do one thing if the part following the TO is true and another thing if it isn't, you can use the ELSE keyword; for example lines 50 and 60 could be combined into one:

```
  50 IF first%=second% THEN PRINT "The numbers were equal":x%=1 ELSE PRINT "The
numbers were not equal":x%=2
```

This line means:

> If the values of first% and second% are equal then print 'The numbers were equal' and set the value of x% to 1. If they are not equal then print 'The numbers were not equal' and set the value of x% to 2.

IF ... THEN Covering Several Lines

Although the presence of the THEN keyword makes the meaning of the line clearer, in most cases it's possible to leave it out, though you must include it if it's followed by a star command. Its principle use is in enabling you to do things like this:

```
IF x%>y% THEN
  PRINT "You won"
  x%=0
  ELSE
  PRINT "You lost"
  IF lives%=0 PRINT "Game Over"
ENDIF
```

In this case, the section which is only executed if the expression following IF is true can be made to extend over several lines. Because there is nothing following THEN on the first line, Basic only executes the next two lines if x% is greater than y%. If this is not the case, it looks ahead for either an ELSE or ENDIF keyword and continues from there. Of course, if x% is greater than y%, the lines down to ELSE will be executed and those between ELSE and ENDIF skipped.

You only need ELSE if there is some action to be carried out if x% is not greater than y%.

Using a structure of this sort allows you to conditionally execute more instructions than you could conveniently put on one line but there is a further advantage. The line before ENDIF contains another IF keyword. In this line, the program only prints 'Game Over' if x% is not greater than y% *and* lives% equals zero. It is possible to put more than one IF keyword on a line, but there is a risk of ambiguity where you have several IFs and ELSEs mixed up together.

True or False?

Basic uses two numbers called TRUE and FALSE. If you were to type:

```
PRINT TRUE
PRINT FALSE
```

you would discover that the value of TRUE is minus one and the value of FALSE is zero. There is a reason for this which will become apparent later in this guide.

As we have just seen, the IF keyword causes *conditional execution*. What the program does depends on what it finds following IF. Basic works out whether or not this part of the line is true and assigns a value to the whole lot; TRUE if it is true and FALSE if it isn't. Whether it goes on to execute the bit following THEN or the bit following ELSE depends on whether the value is TRUE or FALSE.

Try typing:

```
PRINT 2=2
```

Although this looks like a piece of bad programming which should produce an error message, Basic will simply work out the values of the expressions on either side of the equals sign, decide that they are equal and give you the answer −1 which, as we have seen, is the value of TRUE. Similarly, if you typed:

```
PRINT 2>3
```

Basic would decide that 2 was not greater than 3 and print:

```
0
```

which is the value of FALSE.

In CASE ... OF Trouble

Before we leave the subject of IF ... THEN, look at the following program, though you need not bother to type it in:

```
10 INPUT x%
20 IF x%=1 PRINT "You typed one"
30 IF x%=2 PRINT "You typed two"
40 IF x%=3 PRINT "You typed three"
50 IF x%=4 PRINT "You typed four"
60 PRINT "End of program"
```


In this fairly trivial program, we examine the value of x% and take various steps depending on whether it is 1, 2, 3 or 4, in this case simply printing out a message. This would certainly work, but it can mean several wasted program steps.

Suppose x% was 1. The program detects this in line 20 and goes on to print 'You typed one'. Having done this, it must then check for other values of x% in lines 30, 40 and 50 before reaching the next step in line 60. This is a complete waste of time because x% can only have one value and we've already found it in line 20. We really only want to execute one of the four PRINT statements in lines 20 to 50, then move directly to line 60.

What we need is a command which will check the value of x%, take appropriate action and then move straight on to the next part of the program. For this, we can use a CASE ... OF ... ENDCASE structure:

```
 10 INPUT x%
 20 CASE x% OF
 30   WHEN 1:PRINT "You typed one"
 40   WHEN 2:PRINT "You typed two"
 50   WHEN 3:PRINT "You typed three"
 60   WHEN 4:PRINT "You typed four"
 70   OTHERWISE
 80   PRINT "You didn't type one, two, three or four"
 90 ENDCASE
100 PRINT "End of program"
```

Line 20 looks like a rather strange use of the English language. What it means is that we examine the value of x% and compare it in turn with the various numbers following the WHEN keywords in lines 30 to 60. If the program finds a match on a particular line, it executes the remainder of the line following the colon and proceeds straight to line 100 following the ENDCASE keyword, ignoring the lines in between.

We've added a bit to this program that wasn't in the previous example. If the program doesn't find a match for x% in any of the numbers following the WHEN keywords, it executes the line following the OTHERWISE keyword, line 80. You don't have to have an OTHERWISE keyword in a CASE ... OF structure. You only need it if there is something to be done if none of the conditions following the WHEN keywords is met.

You can use CASE ... OF ... ENDCASE to check various unrelated conditions and take action according to the first which is true like this:

```
CASE TRUE OF
      WHEN x%>y%:PRINT "You won!"
      WHEN a%=b%:PRINT "We drew!"
ENDCASE
```

In this situation, we are comparing TRUE with the various expressions following the WHEN keywords. This means that we will have found a match as soon as we come across an expression which is true. If x% is greater than y%, we print ‘You won!’ and proceed to the next part of the program. If not, we check to see if a% equals b% and print ‘We drew!’ if it does.

You could use this arrangement if you only ever wanted to take *one* of the various courses of action. As soon as the program finds a match, it does whatever is on the rest of the line, then stops looking and moves to the ENDCASE keyword.

Round and Round the Repeat Loop

In Section 2 we used a GOTO command to make a program repeat itself indefinitely.

Experienced programmers don’t like using GOTO, as it can make the program more difficult to follow and also makes it dependent on its line numbers. A better way is like this:

```
10 REPEAT
20   INPUT "What is the first number",first%
30   INPUT "What is the second number?",second%
40   PRINT first%+second%
50 UNTIL FALSE
```

When the program encounters the command REPEAT, it remembers where it was and continues executing the program until it gets to UNTIL. It then works out whether or not the bit that follows UNTIL is TRUE and jumps back to where REPEAT was if it isn’t.

Clearly, FALSE can never be true, so the program always repeats itself until we either shut down the machine or press Esc.

We could easily make the program end when, for example, we enter two numbers which add up to 10 by changing the last line:

```
50 UNTIL first%+second%=10
```

or, if you like:

```
50 UNTIL first%>second%
```

which will end the program if the first number is greater than the second one. Have a go at modifying the program to see what you can produce.

Counting the Loops

Quite often, we want a program to do something a certain number of times. Say, for instance, that you wanted your machine to print ‘Hello’ 10 times. You could do it like this:

```
10 count%=0
20 REPEAT
30   PRINT "Hello"
40   count%+=1
50 UNTIL count%=10
```

Line 40 is a short way of saying:

```
count%=count%+1
```

Its effect is to increase count% by 1. This happens every time our program goes round the loop, so each time we reach UNTIL the value of count% tells us how many times we have been round it. When it gets to 10, the program stops.

This program works perfectly well, but there is a shorter way of doing it, provided we want count% to increase by the same amount each time. Type NEW to get rid of the old program, then try this:

```
10 FOR count%=1 TO 10
20   PRINT "Hello"
30 NEXT
```

For obvious reasons, this is known as a FOR ... NEXT loop. The program first sets count% to 1, then follows the instructions until it comes to NEXT, then checks to see if count% has reached the value following the TO in line 10. If it hasn’t, it increases it by 1 and goes back to FOR to follow the loop again.

Once count% reaches 10, the program stops going round the loop and continues, or in this case ends.

Further Use of the Counting Number

You can either use the variable count% to simply count how many times the program goes round the loop, or you can make use of it in other ways, like this for example:

```
10 INPUT "How many times shall I go round the loop",total%
20 FOR number% = 1 TO total%
30   PRINT "This is loop number ";number%
40 NEXT
```

This example shows us that the thing which governs how many times we go round the loop doesn't have to be a simple number, but could be a variable. In this case, we enter a number ourselves to tell the program how many times to go round the loop.

If you were to leave out the semi-colon (;) in line 30, the program would print a substantial gap between the word 'number' and the number following it, like this:

```
How many times shall I go round the loop?10
This is loop number          1
This is loop number          2
This is loop number          3
This is loop number          4
This is loop number          5
This is loop number          6
This is loop number          7
This is loop number          8
This is loop number          9
This is loop number         10
```

The reason for this is that Basic usually allows room for 10 characters on the screen when printing a number unless we tell it not to. The string inside the quotes ends with a single space between the word 'number' and the quote symbol, and the first nine lines of the loop each contain a further nine spaces before the number.

This is fine if we want the right-hand digits of the numbers to line up – you can see that the '0' of the '10' in the last line is underneath the other numbers. Sometimes, though, we don't want this, so we include the semi-colon, which has the effect of pushing the number as far as it will go to the left. This makes the screen look like this:

```
How many times shall I go round the loop?10
This is loop number 1
This is loop number 2
This is loop number 3
This is loop number 4
This is loop number 5
This is loop number 6
This is loop number 7
This is loop number 8
This is loop number 9
This is loop number 10
```

Your counting number doesn't necessarily have to increase by 1 each time. Try this, for example:

```
10 FOR count%=2 TO 10 STEP 2
20   PRINT count%
30 NEXT
```


The use of STEP followed by 2 makes this program print even numbers from 2 to 10. You can even make it count backwards by using a negative number following STEP:

```
10 FOR count%=10 TO 2 STEP -2
```

Notice the way that 10 comes before 2 in this example − we must always put the initial value of count% first and the final value last.

Waiting Around

Sometimes we may wish to make the program wait a certain length of time before doing something. This is particularly useful in games programs. We could do it easily by making it count up to some large number, for example:

```
FOR count%=1 TO 1000000:NEXT
```

The snag with this technique is that you can never be sure how long it will take. This line will take something like 14 seconds on an older machine fitted with an ARM2 processor; if you have an A3010, A3020 or A4000, its processor will be an ARM250, which is a bit faster, and a machine with an ARM3, such as an A5000, A540 or an upgraded A310 or A3000 will be faster still. A Risc PC with a StrongARM processor would take hardly any time at all.

Fortunately RISC OS provides us with an accurate means of measuring time. Your machine has a counter which is increased once every hundredth of a second and you can read it by using the Basic variable TIME. This should not be confused with the date and time of day, which you can also read with the variable TIME$.

Try this in immediate mode:

```
t%=TIME:REPEAT UNTIL TIME-t%>500
```

This simple program sets t% to the current value of TIME and keeps repeating its empty loop until TIME exceeds this number by at least 500.

When you press Return, you will have to wait five seconds for the Basic prompt to reappear. This is because TIME is in *centi-seconds* or hundredths of a second.

You will notice that we used a 'greater than' symbol rather than an equals sign. This is a safeguard, in case our program didn't happen to read the value of TIME when it was *exactly* 500 greater than t%. This is not very likely in this program, as our simple loop will take much less than one hundredth of a second to run, but it could happen in a more complicated program with a longer loop. If it did, we would find that our loop would go on repeating long after it should have stopped.

WHILE ... ENDWHILE

There's one more type of loop, a bit like the REPEAT ... UNTIL loop but with one major difference. It looks like this:

```
WHILE x%>2
  PRINT x%
  x%-=1
ENDWHILE
```

In this loop, we print the value of x% and reduce it by 1 each time we go round the loop.

We could do something like this with a REPEAT ... UNTIL loop, of course, but there's one big difference. A REPEAT ... UNTIL loop checks the condition that makes the loop repeat at the *end* of the loop; a WHILE ... ENDWHILE loop checks it at the *beginning*. You may think this doesn't make much difference, but consider what would happen if x% was already 2 or less before we reached the loop. In this situation, we may not want our program to print anything at all. A REPEAT ... UNTIL loop must always be executed at least once but a WHILE ... ENDWHILE loop needn't be run at all.

We've seen in this section how listings may be improved by leaving a space after the line number and indenting. This means that two extra spaces are added during multiple line IF ... THEN structures, CASE ... OF ... ENDCASE structures and FOR ... NEXT, REPEAT ... UNTIL and WHILE ... ENDWHILE loops. These spaces and indentations have no effect on the way the program works and you may prefer to leave them out to save typing. As we saw at the beginning of the section, you can list the program with spaces and indentations by using the LISTO command.

4

Saving and Loading

We've now seen how to enter programs and run them, and hopefully you will have tried a few experiments of your own. Please do – it's the best way to explore the possibilities of your machine!

So far, we've not covered saving a program, so that you can run it again, or writing a program using a text editor, such as Edit, Zap or StrongEd.

Using a Text Editor

Writing a program using a text editor is a little different from doing it using Basic's command level. The first major difference is that you won't see any line numbers, unless you choose to have them visible. When you save the program, the editor will number all the lines with a regular interval between them so, if you load a program, add or remove lines, then resave it, the new version will have the lines numbered differently.

This is a good reason not to use a GOTO command. If you were to add an extra line in immediate mode (as we did earlier, when we added a line 15 between lines 10 and 20), then renumbered the program using the RENUMBER command, any references to line numbers, such as a GOTO command, would automatically be updated. A text editor will not do this, though, so the program would not work properly.

Using structures such as loops and conditional execution, which we found out about in the previous section, you should never have to use GOTO again.

We won't deal with the exact techniques for using each type of text editor; see the editor's instructions for that. Where we mention basic principles, though, we'll describe Edit because that is available on every RISC OS computer.

Now let's create a new program, using the editor. First follow its instructions to create a Basic file. If you're using Edit, open its icon bar menu, go to the *Create* submenu and select *Basic*. A window will appear.

Now type in the following:

```
INPUT "What shall I print",a$
FOR n%=1 TO LENa$
  PRINT LEFT$(a$,n%)
NEXT
```

It's up to you whether or not you add an extra couple of spaces to indent the third line.

When you've finished, you can save the program in the usual RISC OS fashion, by either pressing F3 or using the menu to get the 'Save' dialogue box. Assuming that you created a Basic file, rather than a text file, you should see a Basic filetype icon in the dialogue box. If not, it's not too late to change the filetype, using the editor's *Set type* menu option, *provided you do it before you save the file*. Set the filename to 'What' and drag the icon to the directory of your choice.

A Basic filetype icon should appear in the window, representing your file. You can double-click on this icon to run your program. A window will appear in the middle of the screen (called a *Command Window*) and your program will run in it.

When you type in a string and press Return, it will appear one character at a time, as n% increases until it reaches the number of characters in the string, like this:

```
What shall I print?ABCDEFGH
A
AB
ABC
ABCD
ABCDE
ABCDEF
ABCDEFG
ABCDEFGH
```

If you close the window in which you wrote your program, the editor will, of course, forget about it. To load it again, hold down Shift while you double-click on the file icon.

This actually applies to any type of file. If you Shift-double-click on its icon, it will be loaded into your text editor so that you can see its contents.

Trouble-Shooting Trouble

Now try adding an extra line to the program so that it looks like this:

```
INPUT "What shall I print",a$
Z
FOR n%=1 TO LENa$
  PRINT LEFT$(a$,n%)
NEXT
```

The new line just has a letter Z on it which is the easiest way of producing an error!

If you had typed this program in using Basic's command level and run it by typing RUN, you would expect to see an error message:

```
Mistake at line 20
```

If you run it by double-clicking on the file icon, though, all you'll get is an error message saying:

```
Mistake
```

which is not a lot of help in finding the error (except in a program as short as this one!)

Error Handling

You can deal with this problem by incorporating an *error handler* into your programs. This consists of the keywords ON ERROR, followed by instructions for the program to follow. To make it tell you the line number, the error handler line would be:

```
ON ERROR REPORT:PRINT " at line ";ERL:END
```

You would put this line at a very early point in your program, so that the program would encounter it before it gets to any errors (it can't use it if it doesn't know it's there!).

When Basic detects an error, it jumps to the start of this line. The keyword REPORT makes it print the error message. The next command adds the words 'at line', followed by the line number, which it gets from the Basic variable ERL. You will probably want the program to stop at this point, so we add an END which stops the program so that it doesn't plough straight onto the next line (which may well contain the error, sending it straight back to this line again!).

Adding this line to the beginning of the program makes it look like this:

```
ON ERROR REPORT:PRINT " at line ";ERL:END
INPUT "What shall I print",a$
Z
FOR n%=1 TO LENa$
  PRINT LEFT$(a$,n%)
NEXT
```

This time, if you run the program, the error message will read:

```
Mistake at line 30
```

(The error is now on the third line of the program, not the second because you've added a line!)

If you get a message telling you that there is an error at, say, line 3020 of your very long program, you might well say to yourself, "How do I find line 3020 when I can't see any line numbers?" The text editor will have a facility for finding a particular line (if you are using Edit, key F5 to open Edit's 'Go to text line' dialogue box) but only by reference to which line it is, not which line *number* it is.

The secret is to ensure that you know how the editor numbers the lines of your program. If you have it set so that it starts with line 10 and numbers the lines in tens, line 3020 will be the 302nd line in the program.

All the listings in the rest of this guide will show the line numbers to make it easier to refer to them in the text, but you can, of course, enter them in a text editor without line numbers. They will all have error handlers like the one above to make it easier to debug them while editing. The listings will also be shown with spaces and indentations for clarity.

Loading and Saving in Immediate Mode

You may occasionally want to be able to load a program into Basic in its immediate mode and save it again, possibly as an aid to debugging.

You can load a program with the command LOAD, but you have to tell the system where it is, as well as its filename.

Outside the desktop, the filing system makes one directory the *Currently Selected Directory* or CSD for short. Normally this will be the root directory of your hard disc. The root directory always has the name '$' and no other directory may include this symbol in its name.

If you type *Cat as we did right at the beginning of our experiments, the machine lists the contents of the CSD.

If your file is in the CSD, you can load it with the command:

```
LOAD "What"
```

or with whatever the filename is between the quotes. You can then LIST or RUN it.

If the file isn't in the CSD, you could either enter the full pathname of the file between the quotes or change the CSD to the directory containing the file. This is easy if you have RISC OS 4; just click Menu in the directory window and select *Set work directory*. If you do not have RISC OS 4, you can change the CSD with the *Dir command.

To save a file in the same manner, type:

```
SAVE "What"
```

or put the full pathname between the quotes.

There is an easier way, though. Add the following line as the first line of the program:

```
10 REM > What
```

The keyword REM is short for REMark. What follows it is usually a comment for the benefit of anybody reading and trying to understand the program. When Basic comes across a REM, it usually ignores the rest of the line.

In this case, though, it makes an exception. If you type SAVE on its own (or even just SA.) without a filename, Basic looks at the first line of the program, checking for a REM keyword, followed by a '>' character and treats what follows it as a filename or pathname.

From now on, we will normally show programs with embedded filenames in the first line, showing the file being saved in the Currently Selected Directory. It's up to you to make any changes you need.

5

Putting It All Together

Now that we've seen how to handle variables and save and load a program, we can begin to put it all together to produce a workable program using what we've learnt so far, with a few extra bits.

This program will tell you the day of the week of any date between 1900 and 2099. It is the first program in this guide which may also be found as a file in the accompanying directory so that you don't have to type it in. If you feel you could do with more practice, type it in a bit at a time, saving it as you go. That way, if you make a serious mistake, you can reload the last saved version and start again from the point where you saved it.

Always save a program before you run it, even if you're using a text editor which allows you to run without saving. It's always possible that your program may contain an obscure bug which will cause the computer to crash and need resetting. If this happens, you will lose whatever is in the memory, including your precious program, if you didn't save it!

```
   10 REM > Days
   20 REM calculates day of week
   30 ON ERROR REPORT:PRINT " at line ";ERL:END
   40 REPEAT
   50   INPUT "Please enter the day of the month "date%
   60   INPUT "Now enter month number (1-12) "mon%
   70   INPUT "Now enter the year "year%
   80   IF year%<40 year%+=100
   90   IF year%<140 year%+=1900
  100 IF year%<1900 OR year%>2099 THEN PRINT '"Sorry, this program only works wi
th years 1900 to 2099"':UNTIL FALSE
  110 year%-=1900
  120 leaps%=year% DIV 4
  130 IF (year% MOD 4)=0 AND mon%<3 AND year%>0 THEN leaps%-=1
```

```
140 IF mon%=1 monnum%=0
150 IF mon%=2 monnum%=3
160 IF mon%=3 monnum%=3
170 IF mon%=4 monnum%=6
180 IF mon%=5 monnum%=1
190 IF mon%=6 monnum%=4
200 IF mon%=7 monnum%=6
210 IF mon%=8 monnum%=2
220 IF mon%=9 monnum%=5
230 IF mon%=10 monnum%=0
240 IF mon%=11 monnum%=3
250 IF mon%=12 monnum%=5
260 total%=year%+leaps%+monnum%+date%
270 total%=total% MOD 7
280 IF total%=1 day$="Monday"
290 IF total%=2 day$="Tuesday"
300 IF total%=3 day$="Wednesday"
310 IF total%=4 day$="Thursday"
320 IF total%=5 day$="Friday"
330 IF total%=6 day$="Saturday"
340 IF total%=0 day$="Sunday"
350 PRINT '"That day was a "day$
360 INPUT '"Do you want another go (y/n)",char$
370 PRINT
380 UNTIL LEFT$(char$,1)<>"Y" AND LEFT$(char$,1)<>"y"
```

If you're using Basic's command level, you may have trouble listing this program, as it grows too long to all fit on the screen at once. If you want to examine the first few lines after you've typed most of it in, you'll find that they scroll straight up and off the top of the screen when you press Return after typing LIST, and you don't get a chance to read them.

The solution is to put the screen into *page mode*. You do this by pressing Ctrl-N (hold down Ctrl while you press and release N). In page mode, the screen will scroll part of the way, then stop. To make it scroll some more, press either of the two Shift keys. If the program is long, you may find that too much of it scrolls up the screen before you can release the Shift key and you miss some. You can prevent this happening by holding down Ctrl while you press and release Shift, which makes it scroll more slowly.

While the listing is partly scrolled, you can't do anything else with the machine. To get out of a partly-scrolled program, either hold down Shift to scroll to the end, or press Esc. To get out of page mode, type Ctrl-O.

The program starts with two REM lines. The first contains the embedded filename and the second tells you what the program does. It is always a good idea to include REMs

of this sort. You know what a program does when you've just written it but, if you found it in six months time you might have trouble remembering what it's for. Similarly, you can include REMs throughout the program explaining what various parts of it do. Don't overdo it, though − as well as making the file longer and possibly slowing down the execution speed slightly, it means a lot of extra typing!

Watch Out For Idiots!

To keep things as simple as possible, we will enter the date as three separate variables for the day of the month, the month number (1 to 12) and the year. (The indentation of the REPEAT ... UNTIL loop between lines 50 and 380 stops at line 100 because the 'UNTIL FALSE' on the end of this line fools the LISTO function.)

Lines 80, 90 and 100 are the nearest we'll get in this program to 'idiot-proofing'. It is possible to build in lots of safeguards, for example to reject a month number greater than 12 or to stop you entering 31st February, or indeed 31st April! All these extras add to the complexity of the program, however, and make it more difficult to understand. For the moment, just be careful how you use it.

This program deals with dates in the 20th and 21st century. It is quite possible that someone will enter the year without the century number, for example '93' when they mean '1993' or '02' when they mean '2002'. This program makes the assumption that if you enter '40' or higher, you mean '1940' etc. and if you enter '39' or lower, you mean '2039' etc.

Lines 80 and 90 deal with this situation by adding 100 to the year number if it's less than 40, then adding 1900 if it's less than 140.

Line 100 sees our first use of the keyword OR. Instead of checking if one condition is true following the IF keyword, this line checks two conditions. It checks to see if *either* year% is less than 1900 *or* year% is greater than 2099. If either of these is true, we have entered a year number outside the bounds of our program, so the rest of the line is executed, beginning with a warning message telling us that we've gone wrong.

If we wanted to do something only if two conditions are both met we would use the AND keyword, for example:

```
IF x%=2 AND y%=3 PRINT "They are both correct"
```

This line would print its message if *both* x%=2 *and* y%=3.

The AND and OR keywords are examples of *Boolean algebra*, which we will discover more about later in this guide.

We've put single quotes (') before and after the apology message to print blank lines on the screen. This is purely to make the screen output look better. Following this, the UNTIL FALSE sends us back to line 40 for another try.

Doing the Calculations

Assuming that our year number was correct we get to line 110 and proceed to do a few calculations. First, because we are only interested in years from the beginning of the 20th century, we subtract 1900 from the year number.

The year 1900 was not a leap year, but 2000 was, which is why our calculations will be valid over a 200 year period. This method is based on the fact that 1st January 1900 was a Monday. We need to know how many years and leap years there have been since then, add on the day of the month and a special number for each month, divide the whole lot by seven and take the remainder to tell us which day of the week the date was (or will be).

Line 120 uses keyword DIV, which is a way of performing whole number division. The result is a whole number and any remainder is discarded. Examples of this are:

7 DIV 4=1
8 DIV 4=2
9 DIV 4=2

This line tells us the number of leap years since 1900, remembering that 1900 itself wasn't one.

If the year is a leap year and the month is January or February, we mustn't include that year when we count the leap years. Line 130 deals with this by checking three conditions. The first part of the line checks whether or not the year is a leap year, i.e. divisible by 4. We do this with the MOD keyword. This is rather like the opposite of DIV, in that it performs whole number division but throws away the result and keeps the remainder. To repeat our earlier examples:

7 MOD 4=3 i.e. 1 remainder 3
8 MOD 4=0 i.e. 2 remainder 0
9 MOD 4=1 i.e. 2 remainder 1

Note the use of the brackets round (year% MOD 4). Basic always works out what is inside brackets first before removing them. This removes the risk of it starting by checking to see if 4=0 and applying the value FALSE to it!

The second condition to be met is that the month number, mon%, is 1 or 2 for January or February. This is written as 'mon%<3', meaning mon% *is less than* 3, but it could

just as easily have been written 'mon%<=2', meaning mon% *is less than or equal to* 2. Needless to say, '>=' also means *greater than or equal to*.

The final condition is to ensure that we don't apply this process to 1900, which was not a leap year, which we do by making sure that year% is greater than zero. The two AND keywords ensure that we only reduce the leap year count by 1 if all three conditions are met.

Monthly Adjustments

We need to add a special number to our calculations to compensate for the fact that the months don't all start on the same day of the week. We will call this number monnum%, and we set its value with lines 140 to 250. There are more elegant ways of doing this which don't take up 12 lines, but we're already seeing plenty of new techniques in this example and we'll leave them for later.

Line 260 adds all the necessary numbers together and line 270 divides the result by 7, giving us the remainder by using the MOD keyword. This is a number between zero and 6 which represents the day of the week that we're looking for. All that remains is to turn this into a word that we can print. Lines 280 to 340 do this for us, again in a rather long-winded way.

Line 350 provides the answer we're looking for, by printing the text between the quotes followed by the string variable, so that if day$ was, say, Friday, it would print:

```
That day was a Friday
```

Line 360 prints a blank line then invites us to have another go. Our reply to this is examined by line 380. This line literally means 'Go back to the REPEAT keyword unless the first character of char$ isn't 'Y' *and* the first character of char$ isn't 'y'. This means that, as long as we type in something whose first character is either 'Y' or 'y', line 380 sends us back to the REPEAT at line 40. We could get another go by answering 'Yes!!!'.

6

Graphics and Colours

We've just had a good bit of practice at using numbers and text strings, and printing words on the screen.

'Doesn't it all look boring!', you're probably thinking, 'It's just text scrolling up inside a command window in the middle of the screen. What about some pretty pictures?'

We've left pictures, or *graphics* until now because you need to use numbers to produce them, but this is a good time to have a go.

To use graphics, you need to understand *coordinates*. These are numbers which refer to the position of a point on the screen. Each point has two coordinates; an *x coordinate* which specifies how far in from the left-hand edge the point is and a *y coordinate* which tells you how far up it is from the bottom of the screen. It's standard practice to specify the x coordinate first. A point with both coordinates zero, which is frequently written (0,0), would be at the bottom left-hand corner of the screen. This point is known as the *graphics origin*. It is possible to move the graphics origin to another point on the screen, in which case coordinates to the left of it or below it would be negative numbers, but that's another story.

OS Units, Pixels and Screen Modes

Coordinates are expressed in *Operating System units* or *OS units*, also known as *graphics units*. How many of these there are, horizontally across the screen and vertically up it, depends on which *screen mode* your computer is using for its display. You can find details of available modes in your computer's User Guide.

On a modern RISC OS computer (one with RISC OS 3.50 or later) you can set up which screen you want, in terms of size, number of colours etc., using the Display Manager icon near the right-hand end of the icon bar. When it comes to changing mode

within a program, though, we'll restrict ourselves to using mode numbers, as used by earlier machines.

The size of the screen in a particular mode is usually expressed in *pixels*. A pixel is one of the dots − or **pic**ture **el**ements − which make up the display and so is the smallest shape which, in practice, can be displayed.

Some modes, particularly the higher resolution ones, have square pixels. Mode 27, for example, is 640 pixels wide by 480 high. Each pixel measures 2 OS units wide by 2 OS units high so the mode 27 screen is 1280 × 960 units. Other modes, particularly the lower resolution ones such as mode 12, have rectangular pixels which are also 2 OS units wide but 4 OS units high. Mode 12 measures 1280 × 1024 OS units but has 640 pixels horizontally and only 256 pixels vertically.

Note, by the way, that the height of the screen is roughly ¾ of the width. If the number of pixels vertically in a mode is roughly ¾ of the horizontal number, the mode has square pixels but if it's only half this amount, the mode has rectangular pixels.

Different Modes for Different Monitors

If we include a command in one of our programs to change mode, we may encounter a problem if we want the program to run on any computer. Some older RISC OS computers may have a TV-standard monitor (indeed, the Acorn A3010 could actually use a TV set as a monitor). These monitors can only cope with modes 0 to 15. Later computers, such as the Risc Pc, however, are designed to work with higher resolution monitors and usually display these modes with the height reduced (usually known as 'letter-box format').

Some of the text-based programs in this guide will run perfectly adequately with their output in a command window. These programs don't include a command to change mode; you can always put one in, if you would like the program to use the whole screen.

The programs which do change mode are written so that they use a VGA mode, such as mode 27 or 28. These modes, as we have seen, are 1280 OS units wide and 960 units high. If you have a 'standard resolution' monitor, you will have to change the appropriate line in the file so that it changes to a mode such as 12 or 15.

Now to try drawing some graphics! If you're following this guide on the screen, you'll have to read the next bit of it, or possibly write it down on paper, then try what it says. When you're ready to start, press F12 to get the star prompt and enter Basic that way. When you've finished, quit Basic and return to the desktop. The guide should be waiting, ready for you to try the next bit.

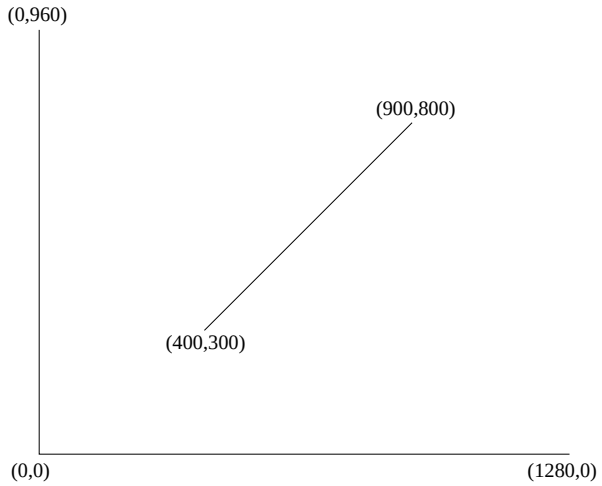
Enter Basic and type:

```
MODE 27
```

or MODE 12 if you have a standard resolution monitor, then type:

```
LINE 400,300,900,800
```

You will see a diagonal white line across the middle of the screen. The command that you gave the machine told it to draw a line from coordinates (400,300) to (900,800), that is from a point 400 units from the left and 300 units from the bottom to a point 900 units from the left and 800 units from the bottom.



This diagram shows the line, together with the left and bottom edges of the screen, and their coordinates.

Now re-enter Basic (if you've left it) and type **`LINE 400,300,900,800`** again, then try typing:

```
DRAW 900,500
```

The result will be a vertical line from the top of the diagonal one to a point below it.

How did we manage to draw a line by typing in only one pair of coordinates and why did it come from the top end of the diagonal line, rather than the other end?

The answer lies in the fact that we're using the *graphics cursor*. This is not a cursor you can see, like the flashing line which shows you where text is going to appear on the screen. When you do any drawing on the screen, the machine remembers the

coordinates of the last point ‘visited’. This is the position of the graphics cursor. You can move the cursor to a new position with the MOVE command and draw a line from it to another point with the DRAW command.

The command you typed using the LINE command was the same as typing:

```
MOVE 400,300
DRAW 900,800
```

Moving the Graphics Cursor

As well as drawing the line, these commands result in the graphics cursor ending up at (900,800). This is why our second line was drawn from the top right-hand end of the first line. We could just as easily have drawn the line with the command:

```
LINE 900,800,400,300
```

This draws exactly the same line, but backwards. It looks the same but the graphics cursor ends up at the left-hand end. The result is that, when you add a second line with the DRAW command, it now comes from the left-hand end.

If you want to experiment with drawing more lines, you will want to clear the screen, so that it doesn’t get too cluttered. You don’t have to change screen mode to do this – just type:

```
CLS
```

which is the Basic command to **CL**ear **S**creen.

Let‘s have a bit of fun drawing lines with the mouse. Take a look at the program file called MouseLines:

```
 10 REM > MouseLines
 20 REM Draws lines using the mouse
 30 ON ERROR REPORT:PRINT " at line ";ERL:END
 40 MODE 27:REM use mode 12 for standard resolution monitor
 50 *Pointer
 60 REPEAT
 70   MOUSE x%,y%,button%
 80   CASE button% OF
 90     WHEN 1:MOVE x%,y%
100     WHEN 2:CLS
110     WHEN 4:DRAW x%,y%
120   ENDCASE
130 UNTIL FALSE
```


When you click the left-hand mouse button, the program draws a line to the pointer. Clicking the right-hand button produces a MOVE command to the pointer position and the middle button clears the screen.

Line 50 shows how we can use a star command within a Basic program. The star tells Basic to hand the rest of the line over to the operating system. In this way, we can use any operating system command within a Basic program.

The command *Pointer turns on the mouse pointer and sets it to the usual blue arrow.

Line 70 gives us information about the state of the mouse and puts it into the three variables following the MOUSE keyword. The x and y coordinates are put into x% and y% respectively and button% indicates the state of the three mouse buttons:

- If the right-hand button is pressed, button%=1
- If the middle button is pressed, button%=2
- If the left-hand button is pressed, button%=4

If two buttons are pressed, the appropriate numbers are added together, but our simple program doesn’t respond to this situation.

It is a simple matter for lines 90 – 110 to either MOVE or DRAW, using the coordinates in xpos% and ypos%, or clear the screen.

Drawing Shapes

You could draw a rectangle by using a MOVE command to position the graphics origin at one corner, followed by four DRAW commands to draw the sides, ending up back at the corner where you started, but it’s a lot quicker to use the RECTANGLE command.

An example is:

```
RECTANGLE 300,200,800,500
```

The four numbers are the x and y coordinates of the bottom left-hand corner, the width and the height. If you want to draw a square, you can miss out the last figure.

If you use RECTANGLE FILL, you can fill it in as well. For example:

```
RECTANGLE FILL 600,500,100
```

will draw a solid square with sides 100 units long, with its bottom left-hand corner at (600,500).

You can also draw a circle or an ellipse using the CIRCLE and ELLIPSE commands. You can find out how to use them in Appendix 1.

It's also possible, though more complicated, to draw other shapes by using the PLOT command. This is described in Appendix 3.

Adding Colour

Although we've tried out the rudiments of making pretty pictures, it still looks rather boring, doesn't it, because everything is in white on a black background. Let's try bringing a little colour into our lives!

We've been experimenting in graphics using mode 27 or mode 12, either of which is capable of displaying 16 colours at once. If you consult the list of screen modes in your machine's User Guide, you'll see that various modes have 2, 4, 16 or 256 colours. Modern RISC OS computers are capable of displaying up to 16 million colours but for drawing purposes, we're going to restrict ourselves to these screen modes.

Colours are referred to by numbers and you can change the colour with the COLOUR command for text and the GCOL command for graphics. Let's experiment with a four-colour mode first, for simplicity.

```
10 REM > Colours4
20 MODE 26:REM four colour mode - use mode 8 for standard resolution monitor
30 ON ERROR REPORT:PRINT " at line ";ERL:END
40 FOR col%=0 TO 3
50   COLOUR col%
60   PRINT "This is colour ";col%
70 NEXT
```

The first line of text is invisible for a very good reason – it's in black on a black background! You can see that the colours in four colour modes are:

0: black
1: red
2: yellow
3: white

Now try the same experiment in a 16 colour mode by changing the first four lines of the program, so the whole program looks like this:

```
10 REM > Colours16
20 MODE 27:REM sixteen colour mode - use mode 12 for standard resolution
monitor
30 ON ERROR REPORT:PRINT " at line ";ERL:END
40 FOR col%=0 TO 15
50   COLOUR col%
60   PRINT "This is colour ";col%
70 NEXT
```


Are You Feeling Dizzy?

Depending on how your machine is set up, you may find that the result of running this program is that you feel a little dizzy, as half the display seems to be jumping up and down and behaving like a chameleon! This is because colours 8 to 15 are defined as flashing colours and, to make matters worse, colour 8 flashes black-and-white as colour 15 flashes white-and-black.

The Basic prompt is also flashing because the last line left the colour defined as flashing black and white. If you want to stop it flashing, type COLOUR 7 to select white. Better still, change mode to get rid of the whole lot!

To save you from having to run it again to see what the colours were, here is a list of the colours in 16 colour modes:

0: black
1: red
2: green
3: yellow
4: blue
5: magenta
6: cyan
7: white
8: flashing black-white
9: flashing red-cyan
10: flashing green-magenta
11: flashing yellow-blue
12: flashing blue-yellow
13: flashing magenta-green
14: flashing cyan-red
15: flashing white-black

If the colours didn't flash, you may find that colours 8 to 15 are one of the two colours shown.

As you will gather, COLOUR changes the colour of any text printed on the screen after the command is given. You can experiment with this by typing COLOUR followed by a number and see the Basic prompt and whatever you type in next change colour.

Changing the Background Colour

When you print text, you actually put two colours on the screen, the *foreground* and *background* colours. These are set to white and black respectively when you change screen mode, but they can both be changed. We've seen how to change the foreground colour; to change the background colour, you use the COLOUR command in exactly the same way, except that you add 128 to the colour number.

You can demonstrate this easily by typing:

```
COLOUR 128+1
```

or COLOUR 129, if you prefer. The Basic prompt and any text you print acquires a red background. If you type CLS to clear the screen, the whole screen turns red. This is because the screen is cleared by filling it with the current background colour.

There is another command, CLG, which clears the screen to the current graphics background colour.

To show all the coloured backgrounds, we need to print some text in a colour that will show up against them. We already know what a black background looks like so let's set the text to black and show backgrounds in all the other colours, starting with red.

```
   10 REM > BackCols
   20 REM Demonstrates background colours
   30 MODE 27:REM sixteen colour mode - use mode 12 for standard resolution mon
itor
   40 ON ERROR REPORT:PRINT " at line ";ERL:END
   50 COLOUR 0:REM set text foreground colour to black
   60 FOR backcol%=1 TO 15
   70   COLOUR 128+backcol%
   80   PRINT "This is background colour 128+";backcol%
   90 NEXT
  100 COLOUR 7:COLOUR 128:REM set things back to normal
```

Graphics Colours

You can set the colour of graphics in the same way as for text, by using the GCOL command, which stands for **G**raphics **COL**our. We can illustrate this by drawing overlapping rectangles in each colour, though we'll stick to the ones that don't flash!

```
   10 REM > Boxes
   20 REM Creates rectangles to show different colours
```



```
   30 MODE 27:REM use mode 12 for standard resolution monitor
   40 ON ERROR REPORT:PRINT " at line ";ERL:END
   50 FOR colour%=7 TO 0 STEP -1
   60   GCOL colour%
   70   RECTANGLE FILL 600-colour%*60,400-colour%*50,120+colour%*120,100+colour%
*100
   80 NEXT
   90 GCOL 7
  100 CIRCLE 660,450,300
```

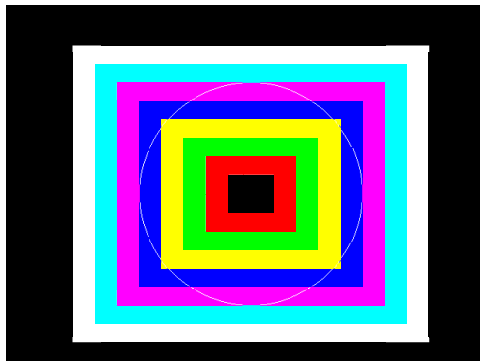
The star (*) symbol in line 70 is the way we show a multiplication sign in Basic. The program follows the usual rules of mathematics, which is to work out anything inside brackets first, then do multiplication and division before addition and subtraction. Thus in the first part of line 70, colour% is multiplied by 60 and the result subtracted from 600.

Line 50 gives us an example of the STEP keyword in a FOR ... NEXT loop. In this case, colour% is *decreased* by 1 each time we go round the loop, because the number following STEP is -1. The program draws progressively smaller rectangles on top of each other, then sets the graphics foreground colour back to white (7) and draws a circle over the whole lot.

Keep this program safe because we make use of it later in the guide.

Redefining Colours

You're probably thinking that the colours we've used so far are a bit gaudy. In fact, they're all produced by turning the red, green and blue guns in your monitor fully on or fully off – a technique that was used on computers such as the BBC Model B in the early 1980s! You may also have realised that modes 12 and 27 are used by the desktop, which has far more subtle colours.



... progressively smaller rectangles ...

We can redefine any colour by using the COLOUR command followed by either two or four figures or variables. At its simplest, this can be used to replace any colour by any other one, using numbers taken from the list of colours used by 16 colour modes.

Mode 0 and 25 are two colour modes, so naturally the colours are numbered 0 and 1. When you start up this mode, colour 0, as always, is black and colour 1 is white.

You can replace the white by, say, yellow. First enter mode 25 (mode 0 if you have a standard resolution monitor), then type:

```
COLOUR 1,3
```

The 1 is the *logical colour*, that is the number by which we refer to the colour and 3 is the *physical colour*, that is the colour which actually appears on the screen.

The Basic prompt and any text already on the screen will turn yellow. The effect is instant because you're not redrawing any characters – you are redefining the actual colour in which they are displayed.

Now type:

```
COLOUR 0,4
```

and the background, which is colour 0, will turn blue. You are still in a two colour mode; your text is still colour 1 and the background colour 0, but you now have yellow text on a blue background.

A Greater Choice of Colours

So far, so good, but this still hasn't got us away from the gaudy colours. Your RISC OS machine is capable of far more than this! We should be able to define any colour in terms of how much red, green and blue it has in it.

We do this with the COLOUR command followed by four numbers. The first is the colour number and the other three are the amounts of red, green and blue respectively.

For technical reasons, these numbers have to lie in the range from 0 to 255. Each colour can be set to one of 16 levels, so these levels are numbered in steps of 16 – level 0 is number 0, level 1 is 16 and so on up to level 15 which is 240. If you use a number between these figures, it will be rounded down. The reason for all this will become apparent in Section 9.

If we set the red, green and blue of colour 0 to half their maximum levels, we will get a grey background. We can do this by typing:

```
COLOUR 0,128,128,128
```


Now try:

```
COLOUR 0,240,144,0
```

This sets red to maximum, green to a little more than half and blue to zero, which will give you an orange background.

You can alter the colour of the text by redefining colour 1 in the same way and, of course, in modes 12 and 27 you have 16 colours to play with in this way!

256 Colour Modes

In a 256 colour mode, such as mode 15 (or mode 28, for VGA monitors), we do not use logical colour numbers like the other modes; instead we define the colour directly with the colour number.

This is a little complicated and may not appear to make much sense at the moment. All will be revealed when we get to the section entitled *Bits, Bytes and Binary Numbers*.

Red, green and blue each have four possible levels, which are given numbers 0 to 3. Having chosen them, you need to do the following:

- Leave the 'red' number as it is
- Multiply the 'green' number by 4
- Multiply the 'blue' number by 16
- Add the three numbers together.

If you want bright red, for example, type COLOUR 3. The number for medium green is $2 \times 4 = 8$, and for bright blue $3 \times 16 = 48$.

Finally, if it's the background colour you want to change, add 128 to the total.

Adding TINT

You may have worked out that the maximum colour number you can have is:

$$3 \times 16 + 3 \times 4 + 3 = 63$$

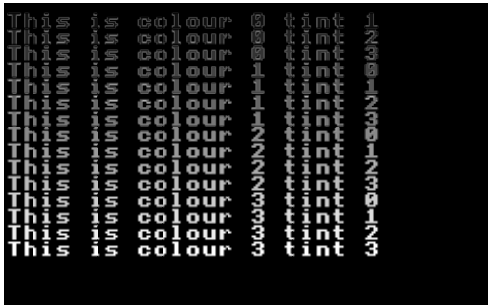
which means 64 possible colours, including zero (black). But isn't this meant to be a 256 colour mode? We only seem to have a quarter of that number.

We can't specify the colour with a number between 0 and 255 because numbers 128 and above are used to set the background colour. Instead, we set four levels each of red, green and blue with the COLOUR command and fill in the gaps between them with another command, TINT.

Again for technical reasons, the four values of TINT have to occupy the range 0 to 255, so they go up in steps of 64. We can illustrate the use of COLOUR and TINT with another short program:

```
 10 REM > shades
 20 REM Shows shades of grey in 256 colour mode
 30 MODE 28:REM use mode 15 for standard resolution monitor
 40 ON ERROR REPORT:PRINT " at line ";ERL:END
 50 FOR col%=0 TO 3
 60   FOR tint%=0 TO 3
 70     COLOUR col%+col%*4+col%*16 TINT tint%*64
 80     PRINT "This is colour ";col%" tint ";tint%
 90   NEXT
100 NEXT
```

Line 70 applies the same value to red, green and blue so that all the colours produced are shades of grey. We have two FOR ... NEXT loops, one inside the other (we say they are *nested loops*). The effect of this is that each time the outer loop (from lines 50 to 100) goes round once, the inner loop (lines 60 to 90) goes round four times – for each value of col% there are four values of tint%. This gives us 16 different shades of grey (including black) and we haven't even varied the amounts of red, green and blue!



... 16 different shades of grey

7

Procedures, Functions and Structured Programming

The programs that we've met so far have been fairly simple. They have only performed each operation once, except where it formed part of a FOR ... NEXT or REPEAT ... UNTIL loop.

It frequently happens, however, that we want to perform the same, or a similar, operation at various points in a program. Consider the following listing, though you needn't bother to type it in or run it:

```
 10 REM > Delay
 20 REM Inputs numbers with a delay between them
 30 ON ERROR REPORT:PRINT " at line ";ERL:END
 40 INPUT "What is the first number",first%
 50 T=TIME
 60 PRINT '"I will now wait 3 seconds"'
 70 REPEAT UNTIL TIME-T>300
 80 INPUT "What is the second number",second%
 90 T=TIME
100 PRINT '"I will now wait 3 seconds"'
110 REPEAT UNTIL TIME-T>300
120 PRINT "The sum of the numbers is ";first%+second%
130 T=TIME
140 PRINT '"I will now wait 3 seconds"'
150 REPEAT UNTIL TIME-T>300
```

This is a rather trivial program but it includes a sequence of instructions which is repeated three times. It asks for a number, announces that it will wait for three seconds, does so, asks for a second number, announces that it will wait for three seconds, does so, prints the sum of the two numbers, announces that it will wait for three seconds, does so, then returns control to you!

As you can see, lines 50 to 70 are identical to lines 90 to 110 and lines 130 to 150. It would be useful if we could just type these lines in once and call them each time we need them.

This is a job for a *procedure*. Doing it this way, the program looks like this:

```
 10 REM > Delay
 20 REM Inputs numbers with a delay between them
 30 ON ERROR REPORT:PRINT " at line ";ERL:END
 40 INPUT "What is the first number",first%
 50 PROCtime_delay
 60 INPUT "What is the second number",second%
 70 PROCtime_delay
 80 PRINT "The sum of the numbers is ";first%+second%
 90 PROCtime_delay
100 END
110 :
120 DEFPROCtime_delay
130 T=TIME
140 PRINT '"I will now wait 3 seconds"'
150 REPEAT UNTIL TIME-T>300
160 ENDPROC
```

The extra lines which we needed to produce our procedure mean that, in this case, we've actually ended up with one more line than we started with, but the program serves its purpose. The three lines which were repeated three times are now in a procedure which we've put at the end of the program. We give it a name which describes what it does. It starts with a DEFPROC keyword so that Basic knows where to find it, and finishes with an ENDPROC keyword.

Incidentally, you can put a space between DEF and PROC if you wish, but there must be no space between PROC and the procedure name.

Each time we want to call the procedure we use a PROC command, followed by the procedure name, as in lines 50, 70 and 90. Again, there mustn't be a space between PROC and the procedure name. When the program gets to one of these lines, it jumps to the procedure, executes it down to the ENDPROC keyword and jumps back to where the PROC command was, ready to continue.

The keyword END in line 100 tells Basic that the end of the program has been reached. If it wasn't there, it would plough straight on and execute the procedure as though it were part of the main program. We haven't needed an END keyword so far, as all our programs have finished when they ran out of lines to execute. In this program, however, the listing doesn't end where the execution of the program ends.

Passing Parameters

In the original version of this program, the three time delays were identical, so the procedure which replaced them had to do exactly the same job each time it was called. What can we do, though, if we want a different delay each time we call the procedure? We can vary its behaviour by passing a number to it, telling it how many seconds delay we want.

The numbers that we pass to procedures are called *parameters* and we put them in brackets after the procedure name:

```
 10 REM > Delay
 20 REM Inputs numbers with a delay between them
 30 ON ERROR REPORT:PRINT " at line ";ERL:END
 40 INPUT "What is the first number",first%
 50 PROCtime_delay(2)
 60 INPUT "What is the second number",second%
 70 PROCtime_delay(3)
 80 PRINT "The sum of the numbers is ";first%+second%
 90 PROCtime_delay(4)
100 END
110 :
120 DEFPROCtime_delay(t%)
130 T=TIME
140 PRINT '"I will now wait ";t%" seconds"'
150 REPEAT UNTIL TIME-T>t%*100
160 ENDPROC
```

Each time we call PROCtime_delay we pass it a number, though it could just as easily be a numeric or string variable. This number becomes the value of variable t% which is in the brackets following the procedure name in line 120. The procedure now uses this value when printing how many seconds it's going to wait and also multiplies it by 100 for counting the centi-seconds in line 150.

Local Variables

Because t% was created in this way, it becomes a *local variable*, which means that it exists only within this procedure. If there is another t% somewhere else in the program, it's an entirely separate variable with a different value. This is useful because it means that you don't have to think up a new set of variable names each time you write a procedure and it reduces the risk of accidentally having two variables in the program with the same name.

A variable which is not local is known as a *global variable.*

Variables like t% which are introduced as parameters are treated as local variables automatically, but it's also possible to create other local variables, using the keyword LOCAL, for example:

```
LOCAL xpos%,ypos%
```

Now let's try a new version of our program that used the mouse to draw lines on the screen, this time to draw crosses and circles:

```
 10 REM > Mouse_X_O
 20 REM Draws crosses and circles using the mouse
 30 MODE 27:REM use mode 12 for standard resolution monitor
 40 ON ERROR REPORT:PRINT " at line ";ERL:END
 50 *Pointer
 60 REPEAT
 70   MOUSE x%,y%,button%
 80   CASE button% OF
 90     WHEN 1:PROCcircle(x%,y%)
100     WHEN 2:CLS
110     WHEN 4:PROCcross(x%,y%)
120   ENDCASE
130 UNTIL FALSE
140 END
150 :
160 DEFPROCcross(xpos%,ypos%)
170 LOCAL size%
180 size%=50
190 LINE xpos%-size%,ypos%-size%,xpos%+size%,ypos%+size%
200 LINE xpos%-size%,ypos%+size%,xpos%+size%,ypos%-size%
210 ENDPROC
220 :
230 DEFPROCcircle(xpos%,ypos%)
240 LOCAL radius%
250 radius%=60
260 CIRCLE xpos%,ypos%,radius%
270 ENDPROC
```

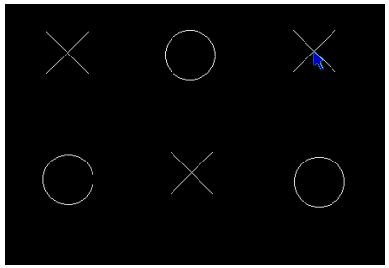
In this program we pass two parameters to our procedures – the x and y coordinates of the mouse. These determine where the centres of the crosses or circles lie. The size of the cross depends on the value of local variable size% in PROCcross and the size of the circle on the value of radius% in PROCcircle.

Structured Programming

This example shows a big advantage of using procedures and one of the great strengths of BBC Basic, namely *structured programming*. This basically means that the program is broken down into smaller sections, each of which does a specific job. These sections may themselves be divided into smaller ones still, as procedures can call other procedures. (In some cases, it's even possible for a procedure to call itself – this is known as *recursion*).

If you look at the main part of the program, in lines 10 to 140, you will see that it's easy to get an overall view of how it works. We read the mouse position and the state of the buttons in line 70. If the left-hand button is pressed, we draw a cross; if the right-hand button is pressed, we draw a circle. We know this because we've given our procedures names which indicate what they do, though we don't have to know exactly *how* they draw a cross or a circle to understand the main loop.

When we get down to understanding the procedures themselves, the situation is reversed. We don't have to know how the x and y coordinates were produced − only that they are passed to the procedure, which draws a cross or circle around them.



... we draw a cross ...

It's worth using procedures to break down your program in this way, even if you only call each of them from one place in the program, as we do in the Mouse_X_O program. The main part of a typical game program, for example, may look like this (without the line numbers):

```
REM > Game
PROCinitialise
REPEAT
  PROCplay_game
UNTIL FNyesno=FALSE
END
:
```

The first procedure, PROCinitialise, deals with everything that has to be set up at the start of the game, and is only called once. We then enter the game's main loop. This consists of a call to PROCplay_game which contains the main body of the game itself. FNyesno is a *function*, about which more shortly, and the game repeats itself until FNyesno returns a value of FALSE.

PROCplay_game could well look like this:

```
DEFPROCplay_game
PROCreinit
PROCdraw_screen
REPEAT
  PROCnext_go
  lives%-=1
UNTIL lives%=0
PROCgame_over
ENDPROC
:
```

PROCreinit would re-initialise any variables which had to be reset at the start of each game, for example the number of lives that you start with, in variable lives%. After PROCdraw_screen has redrawn the screen, we enter another loop in which we play the game once, in PROCnext_go, until we lose a life. We then reduce lives% by 1 and go back for another go until lives% reaches zero, when PROCgame_over prints the words 'Game Over' in fancy lettering on the screen and asks if we want another go. The procedure then finishes, or *exits*, back to the main program.

We are progressively breaking down our program into smaller units which are easier to handle. This also makes it simpler to write the program, as you can do it in sections. If you want to concentrate on writing PROCnext_go first of all, you can write a simpler version of PROCdraw_screen which will draw an elementary screen to begin with. Only when you've got PROCnext_go working well need you concentrate on getting the screen drawing right.

GOSUBs and Subroutines

Some other versions of Basic use the GOSUB command to do a similar job to procedures. This keyword is included in BBC Basic (see Appendix 1) for compatibility with other versions but it's better not to use it for several reasons. GOSUB, like GOTO, jumps to a particular line number instead of calling a piece of programming by name. There is nothing at the beginning of the section (called a subroutine) to indicate what it does or indeed that it is the first line of a subroutine, unless you use a REM statement. In addition, you cannot pass parameters or use local variables.

Structured programming using procedures and functions, on the other hand, is much easier to understand when looking at a listing. We've already seen how we can break each section down into smaller ones. You can see when looking at a listing where each section starts, by the use of DEFPROC or DEFFN and where it ends. A well-chosen name for the routine will give you a good idea of what it does, both at its start and at the point where it is called. The use of parameters and local variables, again with well-chosen names, helps in understanding its inner workings. Imagine trying to understand the workings of the main loop of the Mouse_X_O program if lines 90 and 110 just said 'GOSUB 160' and 'GOSUB 230'!

Functions

In this program we used a function called FNyesno which, you may have noticed, was used just like a variable. This is because a function is similar to a procedure, but it returns a value. There are no prizes for guessing that, in this program, FNyesno returns TRUE if you type 'Y' and FALSE if you type 'N'.

Let's write a program with a simple function to convert temperatures from Celsius to Fahrenheit. To do this we have to multiply by 1.8, then add 32:

```
 10 REM > Degrees
 20 REM converts celsius to Farenheight
 30 ON ERROR REPORT:PRINT " at line ";ERL:END
 40 REPEAT
 50   INPUT "What is the temperature in degrees Celsius",temp%
 60   PRINT "That is ";FNconvert(temp%)" degrees Farenheight"
 70   PRINT
 80 UNTIL FALSE
 90 END
100 :
110 DEFFNconvert(c%)
120 LOCAL f%
130 f%=c%*1.8+32
140 =f%
```

The main program simply inputs the temperature in degrees Celsius and prints a message which includes the function as if it were a variable. You will see that FNconvert has one parameter, which is how we pass the value of temp% to it. Inside the function this becomes local variable c%.

As you can see, a function begins in a similar way to a procedure, with a DEFFN keyword. The ending is very different, though. Line 140 looks as if it has a mistake in it, as it begins with an equals sign. This, in fact, signals the end of the function and also shows the value that's passed back to the main program, in this case the value of variable f%.

This function could actually have been written in two lines, like this:

```
110 DEFFNconvert(c%)
120 =c%*1.8+32
```

RETURNing Values

We've just seen how we can use a function to work out one value and return it to our main program, but suppose we wish to work out two or more values at the same time. You may, for example, need to get the x and y coordinates of an object on the screen. One way would be to use two functions:

```
xcoord%=FNx_position(monster%)
ycoord%=FNy_position(monster%)
```

We call one function, FNx_position, passing it a number which represents the object in question and the function works out the object's x coordinate and returns it to us. Next, we obtain the y coordinate in the same way, calling FNy_position. There is nothing wrong with this method but we can do both jobs in one go by using a procedure.

In our Mouse_X_O program earlier in this section, we called two procedures, PROCcross and PROCcircle, passing them two parameters. In each case, the global variables were called x% and y% and their values were passed to local variables xpos% and ypos%, which existed only within their procedures.

Nothing actually happened to the value of xpos% or ypos% in either of the procedures but, if it had, there would have been no effect on the values of x% and y% when control was returned to the main program. The passing of values through parameters is normally a one-way affair.

We could use a procedure to do the job of several functions if we could make this value passing two-way. To do this, all we have to do is put the keyword RETURN in front of each parameter which we wish to treat in this way, following the DEFPROCcommand. Our earlier example might look like this:

```
PROCposition(monster%,xcoord%,ycoord%)
   .
   .
   .
DEFPROCposition(char%,RETURN xpos%,RETURN ypos%)
```

Here we see the line which calls the procedure and the first line of the procedure definition. The values of xcoord% and ycoord% are passed to xpos% and ypos% in the usual way (though this procedure would probably ignore them). When ENDPROC is

reached at the end of the procedure, the current values of xpos% and ypos% are passed back to xcoord% and ycoord%. By this means we can obtain several values at once from a procedure but we can't, of course, use it like a variable as we did in line 60 of our Celsius to Farenheight program.

Choice of Two-way Parameters

A variable doesn't exist until you create it by giving it a value. We could not call PROCposition in this example if we hadn't already created monster% because it wouldn't have a value to pass to local variable char%. The other two parameters, however, are a different matter. Because xpos% and ypos% have RETURN in front of them in the procedure definition, their corresponding global variables xcoord% and ycoord% don't have to already exist when we call the procedure. In this case, a local variable will initially be given a value of zero and the global variable will be created when a value is passed back to it.

A parameter used in two-way value passing must, of course, be a simple variable. We could not, for example, type:

```
PROCposition(monster%+2,5,ycoord%*3)
```

There is no problem about passing the value of monster%+2 to local variable char%; this may be the way our program is intended to work. If we pass 5 to xpos% and ycoord%*3 to ypos%, however, the procedure will attempt to pass the final values of xpos% and ypos% back to number 5 and expression ycoord%*3, both of which, of course, are impossible.

Function and Procedure Libraries

You can store frequently-used functions and procedures in library files and call them from your program. This is outside the scope of this guide, but you will find a brief account of how to do it in Appendix 1. The keywords which handle library files are INSTALL, LIBRARY and OVERLAY.

8

More on Variables and Errors

We've seen how to add, subtract and multiply variables but there are lots more things you can do with them – the most obvious is division, which you do with a slash (/) symbol.

In mathematics, some operations are done before others. You do multiplication and division *after* working out what is inside brackets, but *before* addition or subtraction.

In Basic, the operations are divided into groups, which are carried out in order:

Group 1:

	Plus or minus sign in front of a variable, e.g. x%=-5
	Logical NOT. If x% is TRUE, NOT x% is FALSE
	Functions
( )	Brackets
?!$|	Indirection operators. We will meet these when we deal with memory addresses in Section 9

Group 2:

^	Raise to the power of. This symbol is on the '6' key

Group 3:

*	Multiplication
/	Division
DIV	Integer division
MOD	Remainder after integer division

Group 4:

+	Addition
-	Subtraction

Group 5:

=	Equal to
<>	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
<<	Shift left
>>	Arithmetic shift right
>>>	Logical shift right

Group 6:

AND	AND operation

Group 7:

OR	OR operation
EOR	Exclusive OR

The AND, OR and exclusive OR operations can be applied to variables or to the *bits* of a number. This is explained in Section 9, along with left and right shifts.

For example, if you want to do a whole number division of x% by y% + 3 and make the result the value of z%, don't put:

```
z%=x% DIV y%+3
```

This will do a whole number division of x% by y%, then add 3 to it. Not quite what we want. Use brackets instead, so that the line reads:

```
y%=x% DIV (y%+3)
```

Arrays and Data

Back in Section 5, we met the program for working out the day of the week of any date, which had a rather cumbersome way of setting the variables monnum% and days$. We can improve on this substantially by using *array variables*.

An array is rather like a set of variables, all with the same name but identified by a number which follows the name. To see how we use them, together with an easy way of getting numbers or strings into them, let's rewrite the program:


```
  10 REM > Days2
  20 REM calculates day of week
  30 ON ERROR REPORT:PRINT " at line ";ERL:END
  40 DIM monnum%(12),day$(6)
  50 FOR m%=1 TO 12:READ monnum%(m%):NEXT
  60 FOR d%=0 TO 6:READ day$(d%):NEXT
  70 REPEAT
  80   INPUT "Please enter the day of the month "date%
  90   INPUT "Now enter month number (1-12) "mon%
 100   INPUT "Now enter the year "year%
 110   IF year%<40 year%+=100
 120   IF year%<140 year%+=1900
 130 IF year%<1900 OR year%>2099 THEN PRINT '"Sorry, this program only works w
ith years 1900 to 2099"':UNTIL FALSE
 140 year%-=1900
 150 leaps%=year% DIV 4
 160 IF (year% MOD 4)=0 AND mon%<3 AND year%>0 THEN leaps%-=1
 170 total%=year%+leaps%+monnum%(mon%)+date%
 180 total%=total% MOD 7
 190 PRINT '"That day was a "day$(total%)
 200 INPUT '"Do you want another go (y/n)",char$
 210 PRINT
 220 UNTIL LEFT$(char$,1)<>"Y" AND LEFT$(char$,1)<>"y"
 230 DATA 0,3,3,6,1,4,6,2,5,0,3,5
 240 DATA Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday
```

Straightaway we've shortened the program from 38 lines to 24.

If we want to use an array, we must tell Basic about it and how many *elements* it has so that it will allocate an appropriate amount of memory to hold it. This is the job of the DIM keyword, short for *dimension*, in line 40.

We're defining two variables, separated by a comma. The number in brackets after each array name is the highest element number we shall use with this array. In fact, the number of elements is 1 more than this, as the elements are numbered from zero. To keep things simple, we won't use monnum%(0) but just use monnum%(1) to monnum%(12), since these numbers correspond to the month number which we type in.

The end result of the calculations is a number between 0 and 6 which represents the day of the week we're looking for, 0 for Sunday and 6 for Saturday. In order to print the name of the day, we will print one of the elements of day$(), whichever one holds the appropriate name.

How, though, do we get all the names into the elements of the array? We could use a number of lines of Basic, such as day$(0)="Sunday":day$(1)="Monday" etc., but that

would be just as long-winded as our original method. This is where the READ command comes in.

Learning to READ

When Basic encounters a READ command, it looks for a DATA keyword. It finds the first of these in line 230, followed by a series of numbers, separated by commas. It sets the value of the variable following the READ command to the first of these numbers and also remembers whereabouts in the program the number was. Each time it encounters a READ command, it takes the next DATA number after the last one it used. If it gets to the end of the line, it looks for another DATA keyword on a later line.

These lines, called *DATA statements*, can be placed anywhere you like in the program. You may like to put them immediately after the part containing the READ command (but outside any loops), to make it clear what they're associated with. In this case, we've put them at the end, so as not to interrupt the flow of the program, which would make it more difficult to understand.

You can use the READ command with either numeric or string variables, but the data must match the type of variable you're READing, otherwise you will get an error message.

All our READing is done in FOR ... NEXT loops. This makes it easy for us to go through all the elements of monnum% and day$, putting a value into each of them. The first time we go round the loop in line 50, m% is 1 so we read a zero into monnum%(1), which is the first number after the DATA keyword in line 240. The next time round, m% is 2 and we read a 3 into monnum%(2) and so on until we get to monnum%(12), which we make 5. At this point, Basic knows that we've used up all the numbers in line 230, so the next READ command, which occurs in line 60, takes data from line 240.

Now we have values in all the elements of monnum%() and day names in all those of day$() and it's a simple matter to use the appropriate element of monnum%() in line 170 and to print the appropriate day of the week in line 190.

You can reset the point where Basic takes its data from using the command:

```
RESTORE xxx
```

where xxx is the line number. To avoid referring to a particular line number, you can type:

```
RESTORE +1
```

This will make the program use the first DATA statement following this line.

More Dimensions

The arrays which we've used so far are *one-dimensional* arrays. You can have an array with two or more dimensions by including the appropriate number of figures in the brackets, for example:

```
DIM matrix%(4,5)
```

This will set up a two-dimensional array called matrix%(), having a total of (4+1) times (5+1), that is 30 elements. You can see that the size of multi-dimensional arrays can easily grow very large and gobble up vast amounts of memory if you DIM them larger than you really need them!

By the way, once you've defined the size of your array with the DIM keyword, you can't alter it. Basic has allocated the appropriate amount of memory to your array and it can't be changed.

Letters and ASCII Codes

Your computer is very good at handling numbers. It stores them in its memory and its ARM or StrongARM processor deals with them very quickly.

How, though, does it handle text? The answer is that it uses a code number to represent each letter or other character that can appear on the screen.

Text in computers has been around for almost as long as computers themselves, as a result of which the code system for dealing with it has been thoroughly standardised. Because of this, it's possible to write a text file on your RISC OS machine, using a text editor, save it on a PC format disc and use it with a word processor on a PC or any other machine that handles text files. This code is called the *ASCII* code, which stands for **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange.

The code consists of numbers up to 255, though 128 and above are not used as often as 127 and below and may vary between different versions of the code. The numbers which mainly interest us are the ones from 32 to 127, which represent characters shown in the table on the following page.

Code 32 is a space and 127 means *backspace and delete*. Codes below 32 are used for other purposes and don't print characters on the screen. We'll meet them later in Appendix 2.

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_	127	delete

ASCII codes

GET and INKEY

You can see ASCII codes in action by using the GET keyword. This is a kind of Basic variable which takes its value from a key which you press. Each time Basic encounters GET in your program it checks to see if you've pressed a key, and waits for you to do so if you haven't. There are two versions of GET; GET$ produces a one-character string containing whichever character you typed, and GET produces a number, which is the ASCII code of the character.

Start up Basic's command level and enter:

```
REPEAT PRINT GET:UNTIL FALSE
```

The Basic prompt will disappear and the cursor (unless you're using a task window) will blink at you, waiting for you to press a key. Try typing a capital 'A'. The number 65 will appear on the screen. If you check the list, you will see that 65 is the ASCII code for A.

You can try this with any characters you like – capital or lower case – and press Esc when you get tired of it, but before you do so, try typing a few letters with Ctrl held down. You'll find that you get the ASCII code for the capital letter with 64 subtracted from it. Ctrl+A for example produces an ASCII code of 1.

When you use GET, your program will always wait for you to press a key and will not continue until you do. This may not be satisfactory for certain types of operation − you might, for example, have objects moving around the screen and you won't want them to freeze until you press a key. In this case, you need the INKEY keyword.

Try typing:

```
PRINT INKEY(500)
```

and see what happens.

Just as with GET, the Basic prompt disappears and you're left staring at the cursor. After five seconds, however, the number -1 will appear, followed by the prompt. If you press a key within this time, the prompt will reappear immediately and the number will be the ASCII code of the key.

Unlike GET, INKEY does not wait forever for you to press a key but puts a time limit on your actions, determined by the number in brackets following the keyword. Because this number is in centi-seconds, INKEY(500) waits a maximum of five seconds. If you press a key during this time, the waiting period ends immediately, you get the code for the key pressed and the program continues. If you don't press a key, INKEY returns -1.

Like GET, there is a variation of INKEY called INKEY$. This works in the same way as INKEY except that it returns a one-character string containing the character typed. If you don't press a key within the time limit, it returns a null string (zero characters long).

You can check for a keypress without waiting at all by using INKEY(0). It is most unlikely, of course, that you will press a key at the exact instant when this command is executed but it is not that critical. The machine stores keypresses in the *keyboard buffer*

until a program reads them so x%=INKEY(0) simply checks to see if a key has been pressed since the last time the buffer was read.

You can prevent spurious keypresses interfering with your program by *flushing* the buffers, that is emptying them. This can be done with a star command:

```
*FX15
```

VDU Codes

Now for the other end of things – sending ASCII codes to the screen. We do this with the VDU command. After escaping from your previous loop, type:

```
VDU 65
```

A capital ‘A’ will appear on the screen with the Basic prompt to the right of it. We’ve sent 65 to the screen, which caused it to print the ‘A’, but we didn’t tell it to go to a new line, so the prompt appeared on the same line.

```
>VDU 65
A>_
```

... with the Basic prompt to the right of it ...

You can follow a VDU keyword with a string of numbers, or variables, separated by commas, for example:

```
VDU 65,66,67
```

will print ‘ABC’.

You can try sending codes less than 32, but some of them will produce unpredictable results and you may end up having to reset your machine. Try:

```
VDU 7
```

This will make the machine beep, and typing Ctrl G will produce the same result, because it produced ASCII code 7. You can liven up quite a few programs with VDU 7!

VDU 14 (Ctrl N) and VDU 15 (Ctrl O) will put the screen into and out of page mode, as we saw in Section 5 and VDU 12 (Ctrl L) will clear the screen.

If you enter VDU 13 (Ctrl M), the cursor will return to the beginning of the line and the Basic prompt will appear there. If you enter it on its own, it will look as though nothing has happened. You may have discovered when you were doing your ‘GET’ experiments that 13 is the ASCII code produced by the Return key, which would normally cause the cursor to move down one line, so why does it behave differently here?

The explanation is that, in normal use, a Return character is followed by a *Line Feed*, which is ASCII code 10 (Ctrl J). This causes the cursor to move down one line. When you PRINT something, or type in a line of Basic, pressing Return causes both these codes to be sent to the screen, producing the desired effect.

Why, you may be wondering, do the ASCII codes for figures start at 48, the codes for capital letters at 65 and for lower case letters at 97? Why do ASCII codes go up to 255, the same figure that kept cropping up when we were dealing with colours? All will be revealed in the next section!

Further Use of Error Handling

Our Days2 program earlier in this section included an instruction in line 130 to report that we had made an error. It also interrupted what we were doing, in this case by sending us back to the beginning of the REPEAT ... UNTIL loop. These two operations are usually performed by the machine’s error handling system, so let’s examine how we can make it do the job for us.

Here is a new version of the program, with some alterations:

```
  10 REM > Days3
  20 REM calculates day of week
  30 ON ERROR REPORT:PRINT " at line ";ERL:END
  40 DIM monnum%(12),day$(6)
  50 FOR m%=1 TO 12:READ monnum%(m%):NEXT
  60 FOR d%=0 TO 6:READ day$(d%):NEXT
  70 ON ERROR PROCerror
  80 REPEAT
  90   INPUT "Please enter the day of the month "date%
 100   INPUT "Now enter month number (1-12) "mon%
 110   INPUT "Now enter the year "year%
 120   IF year%<40 year%+=100
 130   IF year%<140 year%+=1900
 140   IF year%<1900 OR year%>2099 THEN ERROR 1<<30,"Sorry, this program only
works with years 1900 to 2099"
 150   year%-=1900
 160   leaps%=year% DIV 4
 170   IF (year% MOD 4)=0 AND mon%<3 AND year%>0 THEN leaps%-=1
 180   total%=year%+leaps%+monnum%(mon%)+date%
```

```
190   total%=total% MOD 7
200   PRINT '"That day was a "day$(total%)
210   PRINT '"Do you want another go (y/n)"
220   char$=GET$
230   PRINT
240 UNTIL char$<>"Y" AND char$<>"y"
250 END
260 DATA 0,3,3,6,1,4,6,2,5,0,3,5
270 DATA Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday
280 :
290 DEFPROCerror
300 CASE ERR OF
310   WHEN 1<<30:PRINT'REPORT$'
320   OTHERWISE
330   REPORT:PRINT " at line ";ERL:END
340 ENDCASE
350 ENDPROC
360 :
```

The first thing we've done is to add a second error handler in line 70. You can have as many error handlers as you like in a program; defining a new one usually makes Basic forget about the previous one. If you want to alter the way in which your program handles errors just temporarily, you can precede the new error handler with:

```
LOCAL ERROR
```

This makes Basic remember about the previous error handler. When you've finished with the new one, the command:

```
RESTORE ERROR
```

will reactivate the earlier one.

In our program, we keep the original error handler in line 30, in case anything goes wrong in lines 40 to 60.

Using Two Error Handlers

The positioning of the new handler in line 70 is very important. Because error handling from this point onwards will be more complex, we'll use a procedure to deal with it, called by the new handler. Under certain circumstances, as we shall see shortly, the program will not stop if there is an error but will continue, picking up its operation following the error handler. If we had simply modified the existing handler in line 30, this would have resulted in lines 40 to 60 being executed again. Line 40 would have given us an error message because we can't DIM an array more than once and lines 50 and 60 would have told us that we had run out of data.

If an error occurs in the main REPEAT ... UNTIL loop, the program jumps to the error handler in line 70. Because this is defined using the ON ERROR keywords, Basic forgets about any procedures, functions, FOR ... NEXT or REPEAT ... UNTIL loops or any other structures which it may have been executing. It just assumes that it is executing the main part of the program and not in any procedures, functions or loops. Because the error handler is located immediately before the main REPEAT ... UNTIL loop, any error within the loop which doesn't actually stop the program will cause it to go back to the beginning of the loop and start again (in the previous version this was done by the UNTIL FALSE instruction following the error message).

You can create an error handler which remembers all the procedures, loops etc. being executed by putting the word LOCAL after ON ERROR. You can find further details of this in Appendix 1.

We've put PROCerror at the end of the program. Because the previous version didn't have any procedures, it had no need of an END statement, so we've added one. It could have been put after the DATA statements, but it seems appropriate to stop execution of the program after line 250, though it makes no practical difference.

Error Lines and Error Numbers

Anything within the machine which generates an error has to produce two things; an error number and an error message. Errors in Basic programs also produce the line number on which the error was detected. Basic makes the error number the value of variable ERR and, as we've already seen, the error line number the value of ERL.

We can generate our own error using the ERROR keyword followed by an error number and a message. Because our error handler is going to identify the nature of the error by its number, we must be careful not to use an error number which may crop up for other reasons, so we shall stick to numbers which won't occur elsewhere.

In RISC OS, a range of numbers from 1,073,741,824 onwards can be used within programs. For reasons which we will discover in Section 9, this number can be written as 1<<30, the next number as (1<<30)+1 and so on. Line 140 generates an error using this number as the error number, followed by the error message.

Making Use of the Error Number

When this error is detected, the program forgets about the FOR ... NEXT loop and jumps straight to the error handler in line 70, which passes it on to PROCerror at the end of the program. The first action of this procedure is to examine the error number, using a CASE ... OF structure. It may seem pointless using this structure with only one

WHEN keyword but this will allow you to expand the program by adding other forms of error detection. You could, for example, generate an error if the month number is greater than 12, giving it error number (1<<30)+1, another one if the date number is too high for the month, giving it (1<<30)+2 and so on. Adding more WHEN lines to the procedure allows each of these to be treated differently. The reason for the brackets can be seen if you look at the order of arithmetic operations at the beginning of this section − '<<' is a 'shift left' operation which is normally performed after plus and minus. If we didn't put brackets round these two expressions, they would be taken as meaning '1<<31' and '1<<32', which is not what we want.

If the error number is 1<<30, the procedure simply prints the error message (preceded and followed by a blank line) and allows the program to continue. For any other error number, it carries out the usual practice of printing the message, reporting the line number and terminating the program.

The only other change in this version is that we are using GET$ instead of INPUT when asking if the user wants another go. This means that you no longer have to press Return after typing 'y' or 'Y' − a single keypress is sufficient.

9

Bits, Bytes and Binary Numbers

Be warned! This section may seem a bit tedious at first but, if you stick with it, you'll understand more about your computer and the workings of ASCII codes and colour numbers.

Our counting system is based on the number ten. We use single-digit numbers up to 9, then put a '1' on the left and set the units back to zero. When the count reaches 99, we add another digit. The number '100' means 'ten times ten', '1000' means 'ten times ten times ten' and so on. We say that ten is the *base* of our numbers and we call them *decimal* numbers.

There is no mathematical reason why we have to use a base of ten, except that everybody does it and we all understand it. It's commonly supposed that early man learnt to count on his fingers and thumbs and, of course, we have ten of those!

Changing the Base

A better system might have been based on twelve, because it's divisible by two, three, four and six while ten is only divisible by two and five. In this case, '10' would mean the number we think of as twelve and we would have to invent two extra squiggles to represent ten and eleven.

How does a computer store and process numbers? If it worked with a base of ten as we do, each element in its memory would have to be capable of being set in ten different states to remember one digit. If we represented a number by a voltage on a wire, we would have to use ten different voltages. The voltage representing 'five' would not be very different from the voltages for four and six and it could very easily make a mistake.

It would be a lot easier to design the machine if it used numbers with a base less than ten – the smaller the better. What is the smallest base it is possible to have for a counting system? Two!

Binary Numbers

Numbers with a base of two have only two digits – 0 and 1 – and are known as *binary numbers*. They can be handled very easily by a computer and all computers, including your RISC OS machine, use them. Each element of your computer's memory can be set in one of two states, rather like a switch that can be on or off.

Let's see what binary numbers look like:

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000

If you examine the numbers in the right-hand column you'll see a definite pattern. Where a number consists of a '1' followed by several noughts, the number in the left-hand column is always a power of 2, that is 2 multiplied by itself a certain number of times. This is to be expected, as all decimal numbers which consist of a '1' plus several noughts are powers of ten, for example 100, 1000, 10,000 etc.

This also means that a number which is all '1's is always one less than a power of 2, for example 3, 7 and 15. These are also the highest numbers that can be represented by a given number of binary digits.

Bits and Bytes

In computing, each 0 or 1 is called a *bit*, which you can think of as being short for **b**inary dig**it**. You may have noticed that your machine is referred to as a *32-bit* computer. This means that its ARM processor can handle a number consisting of 32 bits at the same time. They move round the machine on 32 wires, known as the data bus (they are actually tracks on the printed circuit board). A older 16-bit machine had a data bus consisting of 16 wires, and older machines still, such as the BBC Model B and Master, used 8 bits.

We refer to 8 bits as a *byte*. The highest number that a byte can hold is 11111111, which in decimal is 255. Does that number seem familiar? It's one less than the highest number of colours we can have on the screen at any one time in certain modes, it's the highest number we can use when defining colours and it's also the highest number that we can use for ASCII codes (though we usually only use codes up to 127, which in binary is 01111111).

There are no prizes for guessing that each of these things is represented by one byte. Numbers which are powers of two (or 1 less than them) keep cropping up in computing, so it's handy to be familiar with them. Let's list some of them:

	Decimal	Binary
2^1	2	10
2^2	4	100
2^3	8	1000
2^4	16	10000
2^5	32	100000
2^6	64	1000000
2^7	128	10000000
2^8	256	100000000

You can see one of the snags with binary numbers – they are very long! We've only got up to 256 and already the number has nine digits. A number such as 14,146 in binary would be:

```
11 0111 0100 0010
```

This is a bit of a handful, and it's by no means the largest number your machine will handle. We need a shorthand way of referring to binary numbers, one which will allow us to see where all the '1's and '0's are.

What the Hexadecimal?

The clue to doing this lies in the way we just displayed the number. You may have noticed that the digits are grouped in fours, with spaces between them. Suppose we take each group and represent it by the decimal number that it makes up:

```
11 0111 0100 0010
 3    7    4    2
```

Each of these groups of four bits can represent a number up to 15. You can perhaps see that grouping them in this way is the equivalent of inventing a numbering system with a base of 16, and that is precisely what we've done. The number at the bottom is, in fact:

```
    3 × 16 × 16 × 16
+   7 × 16 × 16
+   4 × 16
+   2
```

which happens to add up to 14,146, the number represented by the binary number that we started with.

Numbers with a base of 16 are known as *hexadecimal numbers*, and can actually have fewer digits than their decimal equivalents. We usually write them with an ampersand or '&' sign in front, to show they are hexadecimal. Thus &3742 is the hexadecimal version of 14,146.

So far so good. Suppose, though, the number we wanted to represent was, say, 14,155. In binary, with the equivalent numbers underneath, this would be:

```
11 0111 0100 1011
 3    7    4   11
```

Now you can see we have a problem. The right-hand group of four bits make a number greater than nine, so we need a two-digit decimal number to represent them. We could hardly write down the hexadecimal version of 14,155 as &37411, as this wouldn't convey the fact that the last two digits stand for one group of four bits.

At the beginning of the section we saw how we could use a counting system with a base of twelve, provided we invented a couple of extra squiggles to represent ten and eleven. Because hexadecimal numbers have a base of 16, we need six extra squiggles, to represent numbers 10 to 15. By convention, we use capital letters A to F, so that the full set of hexadecimal characters up to 15 looks like this:

Decimal	Hexadecimal	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

So the hexadecimal equivalent of 14,155 becomes &374B.

You can show this by entering Basic and typing:

```
PRINT &374B
```

Basic knows that the number you want it to print is hexadecimal because you precede it with an '&' symbol, so you should get the answer 14155. To go the other way and tell it to print a number in hexadecimal form, we use the tilde or '~' character, so that:

```
PRINT ~14155
```

will give you the answer 374B.

Negative Binary Numbers

Hexadecimal numbers are particularly useful when dealing with the 32-bit numbers that your machine is capable of handling. The largest number that can be represented by 32 bits is 32 '1's, which in binary looks like this:

```
1111 1111 1111 1111 1111 1111 1111 1111
```

This would be written in hexadecimal form as &FFFFFFFF which is a lot shorter!

In fact, by convention, any number whose left-most bit is a '1' is usually taken to be a negative number. This is the case if the left-hand hexadecimal digit is 8 or higher.

The number &FFFFFFFF is actually -1. To understand this, think of a mechanical counter like a car mileometer, counting downwards. When the figures pass through zero, they go to a row of nines. Thus, if the counter has four digits, you can think of 9999 as -1, 9998 as -2 and so on.

It's the same with binary and hexadecimal numbers. &FFFFFFFF (all '1's) represents -1, &FFFFFFFE (all '1's except the right-hand bit) is -2 and so on.

You can see this for yourself by entering Basic and typing:

```
PRINT &FFFFFFFF
```

This explains why the value of TRUE is -1. FALSE is a number which is all zeros; TRUE is a number which is all ones.

The largest positive and negative numbers that you can represent using 32-bit hexadecimal numbers are &7FFFFFFF and &80000000, which are 2147483647 and -2147483648 respectively. These also happen to be the largest values of integer variables – this is because an integer variable is held in 32 bits or four bytes.

By the way, half a byte, or four bits, is occasionally referred to as a nibble, or nybble if you like. Four bytes, or 32 bits, are not known as a gobble though, but as a *word*. Your RISC OS machine does a lot of work with words because it handles 32 bits at a time.

ASCII Codes Explained

Now let's take another look at ASCII codes which we first encountered in Section 8. You may have wondered why the codes for numbers started at 48, capital letters at 65 and lower case letters at 97. There was a clue in the fact that the codes were listed in columns of 32 characters.

We've listed the codes again on the opposite page, this time in hexadecimal form. To remind you, &20 is a space and &7F is backspace and delete.

You can now see that, in hexadecimal form, ASCII codes for numbers start at &30.

If we look at the bit pattern for, say, &36, it looks like this:

```
0011 0110
 3    6
```

If we could turn all the 1s in the left half of the number to zeros, we would be left with the binary number 6 which is, of course, the digit which the ASCII code represents.

20	space	40	@	60	`
21	!	41	A	61	a
22	"	42	B	62	b
23	#	43	C	63	c
24	$	44	D	64	d
25	%	45	E	65	e
26	&	46	F	66	f
27	'	47	G	67	g
28	(	48	H	68	h
29	)	49	I	69	i
2A	*	4A	J	6A	j
2B	+	4B	K	6B	k
2C	,	4C	L	6C	l
2D	-	4D	M	6D	m
2E	.	4E	N	6E	n
2F	/	4F	O	6F	o
30	0	50	P	70	p
31	1	51	Q	71	q
32	2	52	R	72	r
33	3	53	S	73	s
34	4	54	T	74	t
35	5	55	U	75	u
36	6	56	V	76	v
37	7	57	W	77	w
38	8	58	X	78	x
39	9	59	Y	79	y
3A	:	5A	Z	7A	z
3B	;	5B	[	7B	{
3C	<	5C	\	7C	|
3D	=	5D	]	7D	}
3E	>	5E	^	7E	~
3F	?	5F	_	7F	delete

Letter Codes Investigated

Now let's look at the codes for an upper case and lower case letter, using G and g as our example:

```
G:    0100 0111
       4    7

g:    0110 0111
       6    7
```

The only difference between the codes is that the sixth bit from the right (usually referred to as bit 5) is 0 for upper case and 1 for lower case. This applies to any letter of the alphabet. It may also be worth noting that the number in the five right-hand bits is the position in the alphabet of the letter; 7 in the case of G.

If we could change the state of bit 5 (that is from 0 to 1 or 1 to 0), we could turn a letter from upper case to lower case or vice versa. This could be useful, particularly if we wanted to check to see if we had typed in a particular string. Suppose, for example, you wanted to find out whether or not the user's name is Fred. You could use a program like this:

```
10 REM > Fred
20 REM Checks to see if 'Fred' typed
30 ON ERROR REPORT:PRINT " at line ";ERL:END
40 INPUT "What is your name? "name$
50 IF name$="Fred" PRINT "That's right" ELSE PRINT "That's wrong"
```

If you type in 'Fred', the program will answer 'That's right', and if you type something else, it will answer 'That's wrong'. Unfortunately, it will also say 'That's wrong' if you type 'FRED' or 'fred', which may not be what you wanted it to do.

We could get round this problem if we had some way of converting any lower case characters that we enter to capitals and then comparing the string with 'FRED'. For this we need some way of manipulating the bits.

Boolean Algebra

A 19th century mathematician named George Boole invented a form of algebra in which all the variables have the value 1 or 0. These variables could be combined together in two ways, known as AND and OR.

The rules are as follows:

- x AND y AND z = 1 only if x, y and z are **all** 1. If any of them is 0, the result is 0.
- x OR y OR z = 1 if **any** of x, y and z is 1. The result is only 0 if they are all 0.

We can also *invert* a variable by using NOT so that, if x is 1, NOT x is 0 and vice versa.

This form of algebra became known as *Boolean Algebra*. It's difficult to see what possible use it could have been in the 19th century, but it now forms the basis of all computers, because it's a very convenient way of manipulating binary numbers!

Modern computers and other digital equipment contain a great many *logic gates*. These are simple electronic circuits which mimic AND and OR functions and which are known as *AND gates* and *OR gates*. A logic gate responds to voltages on its input pins which correspond to 1 or 0 to produce a similar voltage on its output pin.

Logic Operations on Bits

You can also do AND and OR operations on numbers in Basic. Each bit of one number is ANDed or ORed with the corresponding bit of another number to produce the result. To show this, enter Basic and type:

```
PRINT 6 AND 3
```

You should get the answer 2.

To understand this, look at 6 and 3 in binary (we need only concern ourselves with the lowest bits in this case):

6	0110
3	0011
6 AND 3	0010

Each digit in the bottom row is the result of doing an AND operation on the two digits above it. To remind you, the result is only 1 if **both** the top two digits are 1. This only applies in one column (bit 1, the second from the right), so the result is 0010, which is 2.

Now try another example:

```
PRINT 6 OR 3
```

This time the answer is 7. In binary, it works like this:

6	0110
3	0011
6 OR 3	0111

We get a 1 on the bottom line wherever there is a 1 in either of the lines above.

It's clearly much easier to use hexadecimal numbers for this than decimal ones as you can work out the states of the individual bits.

Now try typing:

```
PRINT NOT 0
```

and

```
PRINT NOT 1
```

What's NOT Zero?

You may be surprised to find that NOT 0 is -1 and even more surprised that NOT 1 is -2, until you remember that, in hexadecimal, -1 is written as &FFFFFFFF and -2 as &FFFFFFFE.

The NOT operation has been applied to all 32 bits of the number. In the case of 0, they've all been turned from 0 to 1, producing &FFFFFFFF or -1. In the case of 1, they've all been turned from 0 to 1 except for bit 0, which is already 1, and so is turned from 1 to 0. This produces &FFFFFFFE or -2.

These AND, OR and NOT operations are known as *logic operations*. There is one more, known as *exclusive or*, or EOR, which can be applied to two numbers. The rule here is that, if the bits are different (one is 1 and the other is 0), the result is a 1, and if they are the same, either both 0 or both 1, the result is 0.

It is possible to manipulate the individual bits of a number by doing a logic operation on it, together with a *mask* number, whose bits are in whatever state you need to do the operation.

- To force a bit to 1, set the corresponding bit in the mask to 1 and do an OR operation. The other bits in the mask should all be zeros.
- To force a bit to 0, set the corresponding bit in the mask to 0 and do an AND operation. The other bits in the mask should all be ones.
- To invert a bit, set the corresponding bit in the mask to 1 and do an EOR operation. The other bits in the mask should all be zeros.

We can use these techniques to get a number from its ASCII code:

```
10 REM > Ascnum
20 REM derives a number from its ASCII code
30 ON ERROR REPORT:PRINT " at line ";ERL:END
40 PRINT "Enter a number (0 - 9)"
50 code%=GET
60 num%=code% AND &F
70 PRINT "Twice that number is ";num%*2
```

Line 50 sets the variable code% to the ASCII code of the key you press. If the number is, say, 5, code% will be &35, or 0011 0101. To convert this code to the number it represents, we have to force the left-hand set of four bits to zero, leaving 0000 0101 to represent 5. We do this in line 60 by ANDing code% with &F, which is 0000 1111.

This is our mask number. Because the four right-hand bits of this number are 1, the right-hand bits of &35 are left alone. All the bits of the mask number apart from these four are zero, which has the effect of forcing the corresponding bits of code% to zero. This turns 0011 0101 into 0000 0101, which is 5.

Changing Case of Letters

This is the technique we need to convert all the characters of a string to upper case in our Fred program. All upper case letters have an ASCII code &4x or &5x, where x is any hexadecimal character (this also applies to the six characters with codes &5A to &5F, but this needn't bother us). Similarly, all lower case letters have codes &6x or &7x.

In binary, this appears as follows:

Upper case: 010x xxxx
Lower case: 011x xxxx

where x can be 0 or 1. This covers all letters of the alphabet.

You can see that converting lower case to upper case is done by forcing bit 5 to 0. As we saw earlier, we can do this with an AND operation. We must leave all the other bits alone, so we need a mask number which has a zero in bit 5 and ones in all the other bits, in other words 1101 1111. In hexadecimal form this is &DF.

If we wanted to convert a letter to lower case, we would have to force bit 5 to 1. We would do this with an OR operation, using 0010 0000 as a mask number, or &20.

We can use both these techniques in a new version of our Fred program:

```
 10 REM > Fred2
 20 REM Prints a name in lower case with an initial capital
 30 ON ERROR REPORT:PRINT " at line ";ERL:END
 40 INPUT "What is your name? "name$
 50 char%=ASC(LEFT$(name$,1))
 60 char%=char% AND &DF
 70 name2$=CHR$char%
 80 FOR n%=2 TO LENname$
 90   char%=ASC(MID$(name$,n%,1))
100   char%=char% OR &20
110   name2$+=CHR$char%
120 NEXT
130 PRINT "Hello "name2$", nice to see you."
```

You can type in a name using any combination of upper and lower case letters, and the program will repeat it back to you with a capital for the first letter and the rest in lower case.

ASC and CHR$

In this program, we're using two new Basic keywords, ASC and CHR$. ASC produces the ASCII code for the first character of the string which follows it and CHR$ does the opposite, producing a string containing one character whose ASCII code is the number following it. Try typing:

```
PRINT ASC"A"
```

You should get 65 which, you will remember, is the ASCII code for 'A'. Now try:

```
PRINT CHR$65
```

You should get 'A'.

Line 50 gets the ASCII code for the first character of name$ and puts it into variable char%. This is the character that we wish to convert to upper case, so we AND it with &DF in line 60. In line 70 we begin to create a new string, name2$, giving it a character whose ASCII code is the new value of char%. This is the same letter as the first character of name$, but converted to upper case if necessary.

We don't actually need the brackets and the LEFT$ arrangement in line 50, as the ASC keyword will give us the code for the first character of the string anyway, but their presence makes the meaning of the line a little clearer.

In line 80, we start a FOR ... NEXT loop. We use this to select each letter of name$ in turn, beginning with the second one and continuing until we reach the end of the string, when n%=LENname$. We make char% the ASCII code of the character in the same way as before, but using MID$ this time to produce a one-character string containing the letter that we're dealing with. Line 100 converts this character to lower case and line 110 adds the result to our new string. You may remember that this is a short way of writing:

```
name2$=name2$+CHR$char%
```

This line adds the character whose ASCII code is char% onto the end of the string.

Understanding Colour Codes

In Section 6 we saw how to define a colour in a 256 colour mode. To remind you, the colour number consists of:

- A number between 0 and 3 representing red
- A number between 0 and 3 representing green, multiplied by 4
- A number between 0 and 3 representing blue, multiplied by 16

all added together, with 128 added on if we're defining the background colour.

We could also apply a TINT number, also between 0 and 3, but multiplied by 64.

All this must have seemed very arbitrary at the time! If we examine these numbers in their binary or hexadecimal form, however, it will all make sense.

In a 256-colour mode, the colour of each dot, or *pixel*, on the screen is held, surprise, surprise, in one byte of the machine's memory. It would be very convenient if the number in this byte could refer directly to the amount of red, green and blue in the pixel.

To give equal space in the byte to each colour, though, would need a number of bits which is a multiple of 3 and, of course, there are eight bits in a byte. This means that we can only allocate two bits each to red, green and blue, giving us four levels of each colour – a total of 4 × 4 × 4 or 64 colours in what is supposed to be a 256-colour mode. We also have two bits left over.

Making up the Colour Number

The machine gets round this problem, as we saw, by allowing four TINT, or brightness, levels to each colour. The TINT number uses up the final two bits in each byte.

It would be simple if we could define this number directly with the COLOUR or GCOL command, followed by a number between 0 and 255. Unfortunately, though, we already have the convention of using 128 and above to indicate that we're defining the background colour, which is why we have to specify the tint bits separately.

The eight bits of our colour number look like this:

fxbb ggrr

The bits representing red, green and blue are shown as r, g and b respectively.

Bit 7, shown as 'f', is the foreground/background bit. This is set to 0 if we're defining the foreground colour and 1 for the background colour. Bit 6, shown as 'x' is not used.

You should be able to see that the two 'rr' bits on their own represent a number between 0 and 3, if the other bits are set to 0.

If you move all the digits of a decimal number one place to the left and add a nought on the end, you multiply it by ten. In the same way, if you move all the bits of a binary number one place to the left, you double it. If you move them two places, you multiply it by 4, three places by 8 and so on. The two ‘gg’ bits, therefore, also form a number between 0 and 3, but, as it’s shifted two places to the left, it’s multiplied by 4, to give us 0, 4, 8 or 12.

In the same way, the two ‘bb’ bits make a number between 0 and 3, multiplied by 16.

The TINT number just uses the two highest, or left-hand, bits of the byte, which explains why it’s a number between 0 and 3, multiplied by 64.

Experiments With Colour Numbers

You can try all this out by typing in some binary numbers. To tell the machine that the number is binary, precede it with a ‘%’ sign in the same way that we use ‘&’ for a hexadecimal number.

You will have to try this a bit at a time if you’re following this guide off the screen, as it doesn’t work in a command window. First enter Mode 28 (or mode 15 if you have a standard resolution monitor). If you get an error message saying ‘Bad Mode’, go back to the desktop and use the Task Manager window to increase the size of the screen memory.

Now type:

```
COLOUR 1
```

You should get the Basic prompt in very dark red.

Now try:

```
COLOUR %10
```

This will produce a slightly brighter red and:

```
COLOUR %11
```

will produce a brighter red still. You have now tried all four levels of red, though with only one tint.

Now let’s try the green:

```
COLOUR %0100
```

will produce a very dark green,


```
COLOUR %1000
```

will give you a brighter green and

```
COLOUR %1100
```

will give you bright green.

You can produce the corresponding shades of blue by adding two extra noughts onto these numbers.

... typing in some binary numbers

If you type:

```
COLOUR %1111
```

you will have bright red and bright green together, which will give you yellow. If you have less green than red, by typing COLOUR %0111 or COLOUR %1011, you should be able to produce two shades of orange.

The TINT number uses the top two bits of a byte, which is why it’s a number between 0 and 3, multiplied by 64. Try typing:

```
COLOUR %101 TINT %00000000
COLOUR %101 TINT %01000000
COLOUR %101 TINT %10000000
COLOUR %101 TINT %11000000
```

Changing the tint number doesn’t have as much effect as changing the colour number, as the different tints just fill in the gaps between the colour steps, but you should just be able to see a difference between the lines when you type them in. Remember that each line sets the colour of the *following* line, so the last line just sets the colour of the Basic prompt on the line below.

Now that you know how to use the bits of colour numbers, you may find it easier to define colours as binary numbers when you write your own programs. Suppose, for

example, you wanted to draw a pale green circle in the middle of the screen. You would need plenty of green and just a little red and blue. Using 3 for green and 1 for each of the other two colours, then multiplying by 4 or 16 as appropriate, would give you number 45. It's a lot easier to work out, though, if you can write it as %011101. You can include it in the command:

```
GCOL %011101
CIRCLE FILL 600,500,100
```

Addresses and the Memory Map

We've referred frequently to Basic storing programs and numbers in the machine's memory and we've also mentioned memory *elements*. How, you may be wondering, does the machine know where in the memory anything is that it wants? The answer is that each element of the memory has a number called an *address*.

We saw earlier how numbers move around the machine on the *data bus*, consisting of 32 wires or tracks on the printed circuit board. There's another set of tracks, known as the *address bus*. When the ARM processor wants to read in the number from a particular memory location, it puts the address number of that location on the address bus. This makes the memory chips put the number contained in the location onto the data bus, where it can be read by the processor. The same thing happens when we want to store a number in memory except that this time it's the processor that puts the number on the data bus. It also sends out a 'write' instruction on a separate wire and an address on the address bus to tell the memory chips where to store the number.

The address bus also has 32 wires, which means that it can hold over four thousand million address numbers. The way in which these numbers are used is called the *memory map*.

Our memory map is large enough to cope with far more memory than your machine is ever likely to contain. Most of the addresses are never used but that doesn't matter – address numbers are free! Not all the addresses that are used belong to RAM or ROM. Some are used by input/output devices, usually known as I/O. These include such things as disc controller chips, the memory controller and the chip that generates the screen display.

Each address contains one byte or eight bits. 'But this is a 32-bit machine!' you will be saying. 'Why only eight bits to an address?'

A 32-bit number actually occupies four addresses and the machine usually handles addresses four at a time. This arrangement makes it easier to deal with strings of ASCII characters, which only occupy eight bits each.

When you put a 32-bit number into memory, it is important that the address of its first byte is divisible by four. An address of this sort is said to be *word-aligned*.

Looking at Memory

You can take a look at what's in the memory by using a star command called *Memory. First, though, you need to know where to look. If you just try an address at random, you could well end up with an error message such as 'Abort on data transfer', which means that you tried to read from or write to an address that didn't contain any memory. Even if you were successful, you would probably find that the numbers you found were meaningless.

Let's try a few experiments with memory. (If you're following the guide off the screen, you can do this in a task window.) Enter Basic and load the 'Fred2' program which we looked at earlier, so that we have something to look at, using the technique that we saw in Section 4. First make sure that your currently-selected directory is the one containing the program, then use the command:

```
LOAD "FRED2"
```

This will load the program into memory. To find out where it is, type:

```
PRINT ~PAGE
```

PAGE is the name of the address where Basic stores its programs, and the tilde (~) character tells the machine to print it in hexadecimal form.

Your machine will almost certainly tell you that PAGE is at &8F00. Basic actually uses memory from &8000. The first part of this is used for its own variables and the rest is for the program, starting at &8F00.

Try typing:

```
*Memory 8F00
```

You don't have to type the '&' symbol in star commands because the machine assumes that any numbers you include in them are hexadecimal. You should see something like this:

```
>PRINT ~PAGE
      8F00
>*MEMORY 8F00
```

```
Address  :     3 2 1 0     7 6 5 4     B A 9 8     F E D C :    ASCII Data
00008F00 :    0D0A000D    203E20F4    64657246    14000D32 : ....ô > Fred2...
00008F10 :    5020F439    746E6972    20612073    656D616E : 9ô Prints a name
00008F20 :    206E6920    65776F6C    61632072    77206573 :  in lower case w
00008F30 :    20687469    69206E61    6974696E    63206C61 : ith an initial c
00008F40 :    74697061    000D6C61    20EB081E    000D3231 : apital....ë 12..
00008F50 :    20EE1B28    3AF62085    202220F1    6C207461 : (.î   ö:ñ " at l
00008F60 :    20656E69    3A9E3B22    32000DE0    2220E820 : ine ";fi:à..2 è "
00008F70 :    74616857    20736920    72756F79    6D616E20 : What is your nam
00008F80 :    22203F65    656D616E    3C000D24    61686316 : e? "name$..<.cha
00008F90 :    973D2572    616EC028    2C24656D    0D292931 : r%=-(Àname$,1)).
00008FA0 :    63154600    25726168    6168633D    80202572 : .F.char%=char% €
00008FB0 :    46442620    1150000D    656D616E    BD3D2432 :  &DF..P.name2$=½
00008FC0 :    72616863    5A000D25    6E20E313    20323D25 : char%..Z.ã n%=2
00008FD0 :    6EA920B8    24656D61    1964000D    72616863 : ¸ ©name$..d.char
00008FE0 :    28973D25    6D616EC1    6E2C2465    29312C25 : %=-(Áname$,n%,1)
00008FF0 :    6E000D29    61686315    633D2572    25726168 : )..n.char%=char%
```

Each row contains the contents of 16 addresses, each of which holds one byte. The number on the left-hand side is the address of the first of these bytes and the numbers along the top are the last figure of the address of each individual byte.

You will see that the bytes are split into groups of four, numbered backwards. The reason for this is that 32-bit numbers are stored with the lowest eight bits in the lowest address and the highest eight bits in the highest address. The location starting at &8F10, for example, shows what would have happened if we had stored &5020F439 at this address. The bottom byte, &39, is in location &8F10, the next byte, &F4, is in &8F11, &20 is in &8F12 and the highest byte, &50, is at &0F13. The next location, &0F14, is the start of the next 32-bit number.

The right-hand side of the display shows what happens when all these numbers are converted into ASCII codes. This makes it easy to pick out strings of text. A lot of this program will not make any sense – Basic doesn't store its keywords as text strings, but 'tokenises' them. Each keyword has a number between 128 and 255. The &F4 at &8F04, for example, is the token for the REM keyword in the first line. It's followed at &8F05 by &20 (a space), &3E ('>'), &20 (another space), then a string of ASCII codes making up the word 'Fred2'. Finally, in &8F0D, comes &0D, which is a Return character, which is the way all lines of Basic end. It's not important to be able to follow the program in great detail like this, but it does show how the memory works.

Byte Arrays and Indirection Operators

It is possible to write numbers into a block of memory and read them out again, but it's not a good idea to refer directly to the actual address in your program. This is because

there is always a possibility that a newer version of Basic will use different address numbers and your program won't work.

Look at the following program:

```
  10 REM > Bytes
  20 REM illustrates indirection operators
  30 DIM block% 100
  40 ?block%=&00
  50 block%?1=&11
  60 block%?2=&22
  70 block%?3=&33
  80 block%!4=&87654321
  90 $(block%+8)="Hello There"
 100 PRINT~block%
```

Notice first the DIM command in line 30. We're more used to this command followed by the name of an array with the number of elements in brackets, for example DIM array%(8). When we use the DIM command in this program, without brackets, we're telling Basic to set aside 100 bytes of memory and put the address of the first byte into variable block%. We sometimes say that block% is a *pointer* to this block of memory.

Jumping ahead to the last line of the program, we tell the machine to print the value of block% so that we can use it in a *Memory command later. The actual figure that you get for block% will depend on the length of your program, including spaces. This is because variables, including blocks of memory, are stored after the program listing. If you run the program provided, or type it in exactly as it's shown (but without the spaces after the line numbers), you will probably get a value of block% of &8FD4.

We transfer numbers to and from our memory block using *indirection operators*. There are four types of these and this program shows three of them in use.

Using Indirection Operators

The use of a question mark in line 40 means that we put number &00 into the location whose address is the value of block%, so we put a zero into address &8FD4. The expression 'block%?1' in line 50 means the same as '?(block%+1), so in this case it means 'put &11 into &8FD5. Similarly, lines 60 and 70 put &22 and &33 into &8FD6 and &8FD7 respectively.

The question mark acts on a single byte. As we're using a 32-bit machine, however, we frequently need to handle four bytes, or 32 bits, at the same time. This is where our second indirection operator, an exclamation mark, comes in. Line 80 means 'put 32-bit number &87654321 into four bytes, starting at block%+4, or &8FD8'.

Our final indirection operator is a dollar sign ($) and is used for entering strings. We can't use it in the same way as ? and !, in the form block%$8, for example, so we have to enter it as $(block%+8), which means the same thing.

Line 90 means 'put the string after the equals sign into memory, its first character going into address block%+8 and finishing it with a Return character'.

We can read numbers from memory just as easily as writing them by using the indirection operators on the other side of the equals sign, for example:

```
x%=block%?1
```

You will have to start up Basic, either in a task window or from the command line and load the program into it for this exercise. When you run the program, it will print the value of block%. Use this number in a *Memory command, such as:

```
*Memory 8FD4+20
```

The number following the plus sign tells the machine to show the contents of 32 (&20) bytes. If you leave it out, you will see 256 bytes.

You will probably see something like this:

```
>RUN
      8FD4
>*MEM.8FD4+20

Address  :     7 6 5 4     B A 9 8     F E D C     3 2 1 0 :    ASCII Data
00008FD4 :    33221100    87654321    6C6C6548    6854206F : .."3!Ce Hello Th
00008FE4 :    0D657265    00000000    00000000    00000000 : ere.............
>
```

The first four bytes contain the single-byte numbers which were put in by the '?' indirection operators in lines 40 – 70. The 32-bit number from line 80 has gone in addresses &8FD8 to &8FDB. Notice that, because the bytes are displayed backwards, the number looks the same as it did in the listing.

The string from line 90 has gone into &8FDC onwards and ends with a Return (&0D).

The fourth type of indirection operator is '|'. This is the character obtained by pressing Shift and the backslash (\) key (just above the Return key on older machines and next to the left-hand Shift key on more modern ones). It reserves five bytes and puts a floating point number into them. Like the string indirection operator ($), you can't use an expression such as 'block%|20'. You must type |(block%+20).

Bit Shifting

If we move all the digits of a decimal number one place to the left, adding a nought on the end, we multiply it by ten. Similarly, if we move them one place to the right, we divide by ten.

In the same way, moving the bits of a binary number one place to the left doubles it, moving two places multiplies it by four and so on, and moving them to the right divides it in a similar way.

We frequently need to move the bits of a number one way or the other for various reasons, not just for multiplication and division. We can move them to the left using the 'shift left' symbol '<<'. Try, for example, typing:

```
n%=5
PRINT n%<<1
```

This expression means 'n% with all its bits shifted one place to the left', so you should get the answer ten. Because n% is an integer variable, it is stored as a 32-bit number looking like this:

```
n%=5
                0000 0000 0000 0000 0000 0000 0000 0101          =5

n%<<1
                0000 0000 0000 0000 0000 0000 0000 1010          =10

n%<<2
                0000 0000 0000 0000 0000 0000 0001 0100          =20

n%<<3
                0000 0000 0000 0000 0000 0000 0010 1000          =40
```

Similarly, n%<<2 will give 20, n%<<3 40 and so on. In each case, we've shifted all 32 bits of the number one, two or three places to the left, the highest bits have disappeared off the end of the number and the lowest ones have been replaced by zeros. If you defined n% as any number and told the machine to PRINT n%<<32, the answer would be zero because you would have shifted all the bits off the end of the number and replaced them all with zeros!

Right Shifting and Signed Numbers

Shifting to the right is a little more complicated because there are two ways of doing it. We may be dealing with a signed number, that is one whose left-hand bit or *most significant bit* denotes whether it is positive or negative. If the number is positive, we can simply shift all the bits to the right, replacing the left-hand ones with zeros. If we tried this with negative numbers, though, we would end up with a very large positive number:

```
n%=10
              0000 0000 0000 0000 0000 0000 0000 1010          =10

n%>>>1
              0000 0000 0000 0000 0000 0000 0000 0101          =5

n%=-10
              1111 1111 1111 1111 1111 1111 1111 0110          =-10

n%>>>1
              0111 1111 1111 1111 1111 1111 1111 1011          =2147483643
```

Notice that, in the above example, we used the symbol '>>>' when we moved the bits of n% to the right. This operation is known as a *logical shift right* and it does exactly what shift left does only in the other direction. There are times when we may want to do this when manipulating bits but, when dealing with negative signed numbers, it will produce disastrous results, as we can see.

Arithmetic and Logical Shifting

The correct way to shift a signed number to the right is to leave bit 31 (the left-most bit) as it is and also copy it into the next bit, bit 30, doing this each time we shift the bits. In this way, the bits of positive numbers are replaced by zeros and the bits of negative numbers by ones, which gives the correct result. This is known as an *arithmetic shift right* and has the symbol '>>':

```
n%=-10
              1111 1111 1111 1111 1111 1111 1111 0110          =-10

n%>>1
              1111 1111 1111 1111 1111 1111 1111 1011          =-5
```


We saw in Section 8 how we could use a range of error numbers beginning with 1,073,741,824. The reason for this rather strange large number is that its hexadecimal equivalent is &40000000. In RISC OS, error numbers with bit 30 set to 1 are reserved for use within a program. This means any number between &40000000 and &7FFFFFFF, or indeed negative numbers &C0000000 to &FFFFFFFF, which should be plenty for our use!

Because &40000000 is a number in which bit 30 is a one and all the others are zeros, we can write it as 1<<30 (try typing PRINT ~1<<30). Similarly, &40000001 can be written as (1<<30)+1 and so on. This is why we used 1<<30 as an error number in Section 8.

If you managed to get through the whole of this section in one go, congratulations! Hopefully, you now have a much better understanding of your machine and the way it handles numbers.

10

Improving the Screen Display

This section deals with ways of using the whole screen to improve the appearance of programs. If you type in some of the examples, you won't be able to follow the guide on the screen while you do so, so you may wish to write them down first.

Let's return to our program for working out the day of the week of any date. Although it does what it's supposed to, the screen display is rather boring. It's just white text on a black background, the text just scrolls up a command window and, if we try several dates, our previous attempts remain on the screen, which is a bit untidy!

Here is a new version of the program which improves matters. It makes some use of colour and the text is put nearer the middle of the screen, which is cleared each time you enter a new date.

```
 10 REM > Days4
 20 REM calculates day of week
 30 MODE 27:REM use mode 12 for standard resolution monitor
 40 ON ERROR REPORT:PRINT " at line ";ERL:END
 50 COLOUR 128+4
 60 DIM monnum%(12),day$(6)
 70 FOR m%=1 TO 12:READ monnum%(m%):NEXT
 80 FOR d%=0 TO 6:READ day$(d%):NEXT
 90 ON ERROR PROCerror
100 REPEAT
110   CLS
120   INPUT TAB(15,10) "Please enter the day of the month "date%
130   INPUT TAB(15,11) "Now enter month number (1-12) "mon%
140   INPUT TAB(15,12) "Now enter the year "year%
150   IF year%<40 year%+=100
160   IF year%<140 year%+=1900
170   IF year%<1900 OR year%>2099 THEN ERROR 1<<30,"Sorry, this program only
```

```
works with years 1900 to 2099"
  180   year%-=1900
  190   leaps%=year% DIV 4
  200   IF (year% MOD 4)=0 AND mon%<3 AND year%>0 THEN leaps%-=1
  210   total%=year%+leaps%+monnum%(mon%)+date%
  220   total%=total% MOD 7
  230   COLOUR 3
  240   PRINT TAB(15,14) "That day was a ";:COLOUR 7:PRINT day$(total%)
  250   PRINT TAB(15,16) "Do you want another go (y/n)"
  260   char$=GET$
  270 UNTIL char$<>"Y" AND char$<>"y"
  280 END
  290 DATA 0,3,3,6,1,4,6,2,5,0,3,5
  300 DATA Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday
  310 :
  320 DEFPROCerror
  330 CASE ERR OF
  340   WHEN 1<<30:PRINT TAB(10,14)REPORT$':PRINT TAB(10,16) "Press any key to
continue";:REPEAT UNTIL GET
  350   OTHERWISE
  360   REPORT:PRINT " at line ";ERL:END
  370 ENDCASE
  380 ENDPROC
  390 :
```

Improving the Display Colours

The mode change in line 30 sets the foreground colour to white and the background colour to black. Our first task after this is to change the background colour to make the display look more attractive.

If you refer back to the list of colour numbers in Section 6, you will see that blue is colour 4. To make the background this colour, we use the COLOUR command with 128 added to the colour number. We could show it as 132, but leaving it as ‘128+4’ in line 50 helps us understand the workings of the program when we refer back to it. It takes a tiny amount of time for Basic to work out what ‘128+4’ means, so it’s not a good idea to do this inside a loop which has to repeat itself as fast as possible, but it doesn’t matter here as this part of the program is only run once.

Line 110 clears the screen. The first time we get to this line, of course, the screen is already clear as we’ve just changed mode. The purpose of this operation is to clear the screen again each time our program goes round its main loop.

The text cursor will normally be in the top left-hand corner of the screen after a mode change or CLS command. When you print something on the screen, the line of characters usually starts on the left-hand side and, when you press Return, the cursor returns to the left-hand edge on the line below.

Keeping TABs on Text

We can print text wherever we like on the screen by using the TAB keyword. This is followed by either one or two numbers in brackets and has the effect of moving the cursor to the column and row referred to by the numbers. If, for example, we type:

```
PRINT TAB(40) "Hello"
```

this will print ‘Hello’ on the next line, but 40 character positions in from the left. If you’re using an 80 column mode, such as mode 12 or mode 27, this will place the beginning of the word halfway across the screen.

If you type:

```
PRINT TAB (40,10) "Hello"
```

the word ‘Hello’ will appear 40 columns in from the left and 10 rows down from the top. In mode 27, which has 60 rows, this would place it about one sixth of the way down the screen. Mode 12 has 32 rows, so it will be about one third of the way down.

We can use TAB with the INPUT command, as this also involves printing on the screen, and this is how we first encounter it in lines 120, 130 and 140. The first message is printed starting 15 positions across and 10 rows down (column 0 is the left-hand side of the screen and row 0 is the top row). The date number appears as we type it in at the end of this line, then the next line is printed one line below and the third one line below that, each starting 15 columns in.

Improving the Error Message

Line 340 now contains an improved way of handling the error message that we would print if a year was entered outside the range of the program. Because this line is long, we’ll start it 10 columns in instead of 15.

Because we’re going to blank out the screen when the user has another go at entering the date, we must allow them a chance to read the message first; hence the line ‘Press any key to continue’. This line is printed two lines below the preceding one to improve the appearance of the program.

... 40 columns from the left and 10 rows down ...

The expression ‘REPEAT UNTIL GET’ is a way of getting the program to wait until we press a key. Basic gets the value of GET by waiting for you to press a key and reading its ASCII code. Provided this is not zero (which it will only be if you type Ctrl-@), this mini-loop will only be executed once and the program will continue as soon as you press a key. The program will continue after the error handler, starting the main loop again, and the first thing it will do is clear the screen.

The effect of the semi-colon after ‘Press any key to continue’ is to prevent the cursor going back to the beginning of the next line. It looks a little better if it’s flashing away at the right-hand end of the last line on the screen.

When you sort out the screen layout of your own programs, remember to check what happens when an error message of this sort is printed. It’s very easy to produce a beautiful-looking program which suddenly looks scruffy when somebody presses a wrong key!

Returning to the main program, line 230 introduces a small variation in colour so that the first part of line 240 is printed in yellow. We’ve split the printing in this line so that we can change the colour back to white before we print the name of the day. This will make it stand out more. Again, note the semi-colon after ‘That day was a’, which prevents the cursor going back to the beginning of the next line.

We’ve now put our text in the middle of the screen, but this means the surrounding area is blank. Our program might look more attractive if we could put some sort of pattern round it.

Text and Graphics Windows

Unfortunately, the more complicated the pattern, the longer it would take the machine to redraw it when we clear the screen each time we go round the loop. It would be very convenient if we could just clear the middle bit where we print the text, leaving the surrounding part untouched.

We can actually do this by changing the size of the *text window*. The screen contains both a text and a graphics window. When you change screen mode, these are both set so that they occupy the whole screen, but you can change the size of either of them, using VDU commands.

Text and graphics windows should not be confused with the Wimp windows (the ones with scroll bars etc.) that the desktop uses. Some Acorn documentation at one time referred to text and graphics windows as viewports.

When you print text on the screen, it behaves as though the screen is the size of the text window. The numbers following the TAB keyword refer to the position of the text relative to the top left-hand corner of the window and the CLS command clears just the window, leaving the rest of the screen untouched.

Let’s see how we can make use of the text window with yet another version of our ‘days’ program. In order to make it look good in either mode 27 or mode 12, there are two versions of the program among the files, called Days5_12 and Days5_27. The only differences between them are the mode they select and the vertical TAB values.

```
  10 REM > Days5_27
  20 REM calculates day of week
  30 REM VGA resolution version
  40 MODE 27
  50 ON ERROR REPORT:PRINT " at line ";ERL:END
  60 COLOUR 128+6
  70 CLS
  80 PROCsurround
  90 VDU 28,10,50,70,10
 100 COLOUR 128+4
 110 DIM monnum%(12),day$(6)
 120 FOR m%=1 TO 12:READ monnum%(m%):NEXT
 130 FOR d%=0 TO 6:READ day$(d%):NEXT
 140 ON ERROR PROCerror
 150 REPEAT
 160   CLS
 170   INPUT TAB(12,14) "Please enter the day of the month "date%
 180   INPUT TAB(12,16) "Now enter month number (1-12) "mon%
 190   INPUT TAB(12,18) "Now enter the year "year%
 200   IF year%<40 year%+=100
 210   IF year%<140 year%+=1900
 220   IF year%<1900 OR year%>2099 THEN ERROR 1<<30,"Sorry, this program only
works with years 1900 to 2099"
```

```
  230   year%-=1900
  240   leaps%=year% DIV 4
  250   IF (year% MOD 4)=0 AND mon%<3 AND year%>0 THEN leaps%-=1
  260   total%=year%+leaps%+monnum%(mon%)+date%
  270   total%=total% MOD 7
  280   COLOUR 3
  290   PRINT TAB(12,21) "That day was a ";:COLOUR 7:PRINT day$(total%)
  300   PRINT TAB(12,23) "Do you want another go (y/n)"
  310   char$=GET$
  320 UNTIL char$<>"Y" AND char$<>"y"
  330 END
  340 DATA 0,3,3,6,1,4,6,2,5,0,3,5
  350 DATA Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday
  360 :
  370 DEFPROCsurround
  380 COLOUR 1
  390 PRINT TAB(20,4) "T W O   C E N T U R Y   C A L E N D A R"
  400 COLOUR 7
  410 ENDPROC
  420 :
  430 DEFPROCerror
  440 CASE ERR OF
  450   WHEN 1<<30:PRINT TAB(3,21)REPORT$':PRINT TAB(3,23) "Press any key to co
ntinue";:REPEAT UNTIL GET
  460   OTHERWISE
  470   REPORT:PRINT " at line ";ERL:END
  480 ENDCASE
  490 ENDPROC
  500 :
```

Setting the Text Window

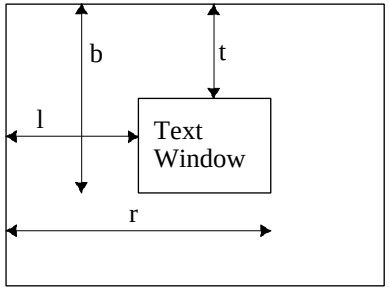
We've changed the beginning of the program somewhat. Line 60 sets the background colour to cyan and line 70 clears the screen to turn it this colour. The business of putting something round the edge of the screen is dealt with by the procedure PROCsurround. You can experiment with various shapes round the screen if you like by rewriting this procedure. As it stands, it simply prints the words 'Two Century Calendar' spaced out in red at the top of the screen, above where our text window is going to be set, then changes the foreground colour back to white ready for printing the text in the main loop.

Having drawn or printed the surroundings, we can now set our text window. This is done by sending ASCII code 28 to the screen, using the VDU command, followed by four other numbers which refer to the positions of the sides of the text window, in terms

of columns and rows. This command takes the form:

```
VDU 28,l,b,r,t
```

The second and third numbers, l and b, refer to the position of the bottom left-hand corner of the window; l is the number of columns in from the left and b is the number of rows down from the top. The other two numbers, r and t, give the position of the top right-hand corner, again r being columns from the left and t rows from the top.



... the text window coordinates

In line 90 then, we send number 28 to the screen to tell it that the four numbers that follow are the text window coordinates. The left-hand side is 10 columns in, which is $^1/_8$ of the way across, as we're in an 80-column mode. The bottom, in mode 27, is 50 rows down, leaving 10 rows underneath, as there are 60 rows on the screen and in mode 12 is 27 rows down, leaving 5 rows underneath, as there are 32 rows on the screen. The right-hand edge is 70 columns from the left and the top edge is either 10 or 5 rows down from the top of the screen.

Having set our window, we can change the background colour to blue, and the text window will turn this colour when we clear the screen at the start of the main loop. This means we will actually have two background colours on the screen at the same time. The window was cleared to cyan when it occupied the whole screen and cleared again, to blue, after it had been reduced in size.

The rest of the program works in the same way as last time, except that we've changed the numbers after the TAB keywords to make the text fit nicely in the window, remembering that they now refer to the number of columns in and rows down from the top left-hand corner of the text window, not the screen.

If you run the program from the Basic command line, rather than the desktop, LIST the program after RUNning it. The listing will fill the text window and scroll upwards. Notice though that only the text window scrolls – the surrounding area, with the words along the top, or any graphics you may have drawn, stays where it is.

This is a very useful feature of text windows. You can have a picture on the screen with a small area of text, within a text window. If you want to let more text appear, it can be kept within its own small area without scrolling the whole screen.

The Graphics Window

Now we've found out about text windows, let's have a look at graphics windows. The text window is the area on the screen where we can put our text, so there are no prizes for guessing that the graphics window is the area where we can draw pictures!

Try this experiment. Enter Basic, select mode 27 or 12 and type:

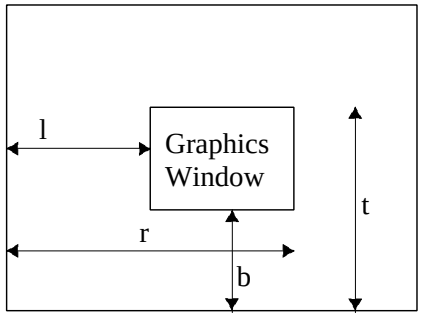
```
CIRCLE FILL 600,500,400
```

You should have a big white circle, more or less in the centre of the screen.

Now we'll set a graphics window. The command for this is similar to the one for the text window and takes this form:

```
VDU 24,l;b;r;t;
```

The meanings of l, b, r and t are similar to those for a text window, except that b and t are measured up from the *bottom* of the screen, not down from the top.



... b and t are measured up from the bottom of the screen ...

Be particularly careful to type semi-colons rather than commas after most of the numbers (we'll find out why later) and don't forget the semi-colon after the last number:

```
VDU 24,250;150;950;850;
GCOL 1
CIRCLE FILL 600,500,400
```

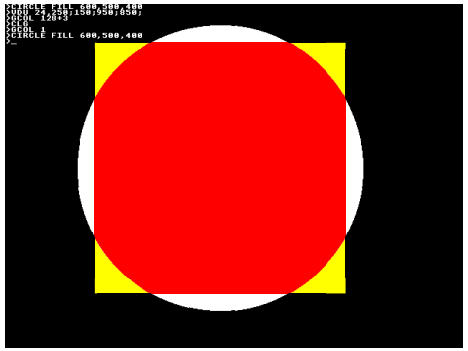

This is exactly the same CIRCLE FILL command as before, so you might think it would draw a red circle to cover up the white one precisely. You'll see, however, that only part of the circle has been redrawn in red − the sides, top and bottom of the original white circle have been left untouched. If you now type:

```
GCOL 128+3
CLG
```

a yellow rectangle will appear. This is the new size of your graphics window, which we've just cleared to a yellow background. CLG, by the way, means **CL**ear **G**raphics.

Clearing the graphics window made the red circle disappear but you can, of course, put it back by repeating the CIRCLE FILL command, to produce an interesting shape.

We set a text window with VDU 28, followed by four numbers indicating the positions of its edges in terms of columns and rows of text. The command to set the graphics window is similar, beginning with VDU 24, but the numbers which follow are graphics coordinates, and the vertical positions are measured up from the graphics origin at the bottom of the screen, rather than down from the top, as in the case of the text window.



... to produce an interesting shape ...

There is a problem here. The VDU system which receives numbers sent to the screen only works with 8-bit numbers, that is numbers between 0 and 255. If you try to send it a number greater than 255, it will ignore all the bits above the first eight, rather as though you had sent the number MOD 256. You can show this by typing VDU 263 and the machine will beep at you, as if you had typed VDU 7. If you subtract 256 from 263, of course, you get 7.

This doesn't matter when you're setting a text window as none of the numbers for rows and columns will be anything like as large as 255. Graphics coordinates, on the other hand, can involve numbers over 1000. For this reason, each coordinate number has to be sent as two separate bytes – the first one is the number MOD 256 and the second the

number DIV 256. It is as if the number has 16 bits and we send the lowest eight first, followed by the highest eight.

Two-byte VDU Numbers

We could split each of our coordinates into two numbers in this way and type VDU 24, followed by eight numbers, but we don't have to because Basic is kind to us and works it out for us. All we have to do is put a semi-colon after the number and it sends it to the VDU as two bytes. We don't put one after the 24, because that is a one-byte number. All the other numbers must have one, even if they're less than 256, because the VDU expects two numbers for each coordinate, making a total of nine for the whole command. This is also why we have to put a semi-colon after the last number.

You can see the effect of the semicolon by typing:

```
VDU 16961
```

This will print a letter 'A' on the screen. If you type:

```
VDU 16961;
```

you should get 'AB'. To find out why this is, type:

```
PRINT ~16961
```

which will print the number in hexadecimal form and show you that it's &4241. The lower byte is the ASCII code for 'A' and the higher byte the code for 'B'. If you send the number to the VDU without a semi-colon on the end of it, only the lower byte is acted on, resulting in an 'A' being printed, but adding the semi-colon means that both bytes are sent, one after the other, and you get both letters.

The Effect of the Graphics Window

Changing the size of the text window affects the position of text on the screen (though it doesn't move any text already there), as the TAB keyword counts columns and rows from the top left-hand corner of the window. The size of the graphics window, however, doesn't affect the positions of graphics, as these are relative to the graphics origin, which is normally in the bottom left-hand corner of the screen. Instead it determines what is drawn and what isn't. Graphics can only be drawn inside the graphics window and anything you try to draw outside will be invisible.

You can show this by typing MODE 27 (or MODE 12) to reset everything, then typing:

```
DRAW 1280,960
```


You will get a white line from the bottom left-hand corner to the top right. The graphics cursor was set to the bottom left when you changed mode and the command drew a line from there to (1280,960).

Now type the MODE command again to both clear the screen and set the graphics cursor back to the bottom left, then alter the graphics window with:

```
VDU 24,300;200;900;800;
```

If you now enter the command to draw the line exactly as before, you'll find that only part of it is drawn, in the middle of the screen. This is because the ends lie outside the graphics window, and so were not drawn.

We'll just do one last experiment before we leave this subject. Don't disturb the setup of the graphics window, but type:

```
MOVE 600,500
VDU 5
```

The first command moves the graphics cursor to the middle of the screen, halfway along the visible part of the line, The result of the second command is that text is no longer printed at the text cursor, but at the graphics cursor instead, so the Basic prompt reappears in the middle of the screen.

Try typing in a line of text. You'll find it will become invisible when it reaches the edge of the graphics window.

You can return text printing to the text cursor either by changing screen mode or by typing VDU 4, and you can reset text and graphics windows so that they cover the whole screen with VDU 26.

Graphics windows are also referred to as *clipping windows* and are the key to how Wimp windows used by the desktop work. When you have mastered the basic techniques in this guide, you may wish to move on to using **W**indows, **I**cons, **M**enus and **P**ointers, as described in *A Beginner's Guide to Wimp Programming*.

11

A Game With Moving Graphics

In Section 6 we found out how to create lines and shapes on the screen, using commands such as DRAW and CIRCLE FILL. This is one of two ways of using graphics; the other is to use pre-prepared pictures and *plot* them on the screen where you want them.

These pictures are called *sprites*. You've already met them when using the desktop as they form the little pictures in all the icons, and you may have tried designing them using Paint. A sprite is actually part of the screen display and it can be any size, from a single pixel to the entire screen, or even larger.

Our next program will grow to be the longest and most complex we've met in this guide so far.

Designing the Sprites

In this program, we'll use virtually everything we've met in this guide so far, so if you skipped something, you'd better go back and read it!

We are going to create a pac-man type character and move him around the screen. Later we'll make him gobble up dots on the screen and find out how to move him over a patterned background.

Like the previous programs in this guide, this program is designed to work in either mode 27 or mode 12. There are a fair number of differences between them, though, especially in the design of the sprites, so there are two complete sets of program and sprite files, in sub-directories Munchie_27 and Munchie_12. Open whichever one is right for you and make it the currently selected directory.

If you're content to use the sprite files already provided, skip the next section. If you'd like to try creating your own, read on.

The first step is to create the character, using Paint.

You'll find full details of how to use Paint in your RISC OS Applications Guide. Because we're going to use simple sprite plotting, you will need to create sprites with the appropriate pixel shape and number of colours for the mode you'll be using. It's easier to do this with the desktop in the same mode. You'll probably have it in that mode already, except possibly for the number of colours so first ensure that the desktop is in a sixteen colour mode.

Start up Paint and click on its icon on the icon bar. A sprite file window will appear, together with a 'Create New Sprite' dialogue box.

The dialogue box will show you the number of colours, which should be 16, and the resolution. For mode 27, this should be 90 pixels per inch horizontally and vertically. For mode 12, it should be 90 horizontally and 45 vertically.

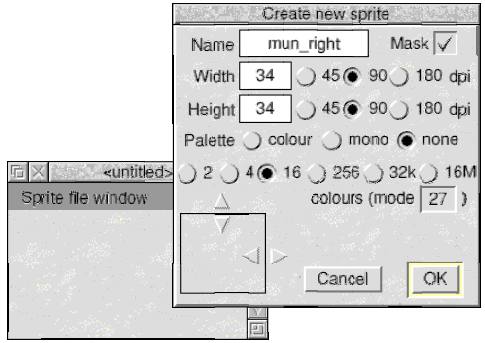
Every sprite must have a name. As we're calling our program 'Munchie' and our first sprite will show our character facing to the right, we'll call it 'mun_right', so type that into the 'name' box. It's important to get the sprite names correct, as we will be calling them by name in the program. Note, by the way, that sprite names are always lower-case.

Choosing the Sprite Size

We now have to decide how big we want our sprite to be. The width and height are given in pixels, not OS units. As we saw in Section 6, mode 27 has square pixels, two OS units wide and two high, and in mode 12, each pixel is two units wide and four high.

You can make the character any size you like, but you will have to set two variables in a later version of the program according to the width and height you choose. In the listings, we're going to assume that the sprites are the same size as the icons in directory windows. These are usually 34 pixels square in mode 27, or 34 pixels wide by 17 high in mode 12.

When you've entered the width and height, turn the mask on and the palette off, then click on the 'OK' box. The dialogue box will vanish and another window will appear, showing the sprite itself (a white square) The sprite window will probably be on top of the sprite file window, unless you've moved it. Click Menu over the sprite window, go through to the 'Paint' submenu and select *Show colours* with Adjust (to keep the menu open) and *Show tools* with Select.



... turn the mask on and the palette off ...

Before we go any further, look at the colours in the 'colours' window. You will see that they have eight shades of grey and the various other colours of the desktop, together with an extra box which represents the sprite's mask. Remember that we won't be using our sprite in a desktop display, but in a normal mode, either 27 or 12, and we saw its colours in Section 6. If we design our sprite using the colours we have here, it will look all wrong when we plot it.

When we set the colour of a pixel, we simply give it a number. All we have to do is change the colours in our display to be the same as the ones in the mode which we'll be using and all will be well. Click Menu on the sprite file window and go to the 'Display' submenu. You'll see an option called *Use desktop colours* which has a tick. Clicking on this will turn off the tick, and close the window if you used Select.

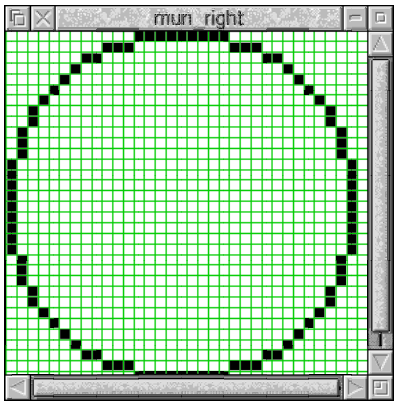
At the same time, the colours will change. They still won't look right if you're in a sixteen colour mode, though, because this mode can't show the colours we want as well as all those of the rest of the desktop at the same time. The solution is to change to a 256 colour screen mode. You should now have two identical sets of colours. The white square in the sprite window will appear to have vanished, leaving just an outline, because it's in colour 0 and colour 0 has just changed from white to black.

Drawing the Outline

Before setting to work on the sprite, you'll need to enlarge the work area so that you can easily select individual pixels. Click Menu over the sprite window, go to the 'Zoom' submenu and enlarge the black sprite in the window to 8 times its normal size. There is a grid over it, so that you can see the individual pixels, but you probably won't be able to see it at first, because it is also black! Go to the **Grid** submenu and turn it green. The small rectangles you can see are the pixels. If you are in mode 12, they will be twice as high as they are wide; if you are in a square pixel mode, such as mode 27, they will be square.

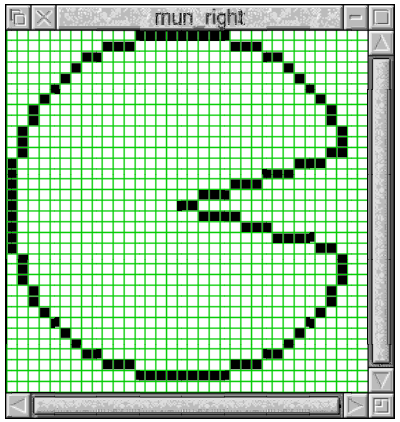
Because we are going to start by drawing black lines, the first thing to do is turn all the pixels white again. Go to the ‘tools’ window and choose **Replace colour** – it doesn’t matter here whether you use **local** or **global** – then choose white in the ‘colours’ window and click on any pixel in the sprite window. All the pixels will turn white.

The next thing we need is a black circle. Choose **Circle outlines** and black. then draw a circle filling the sprite, like this:

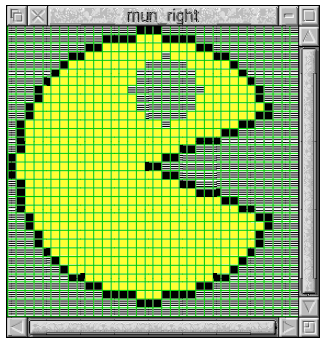


We’re going to need this circle again later, so we need to keep a copy of it in its present form. Click Menu over the sprite in the sprite file window, go to the **Sprite 'mun_right'** submenu and use the **copy** option to make a copy of it called, say, ‘mun_x’.

The next step is to draw an open mouth. Use the **Set/clear pixels** option to remove the straight line down the right-hand side of the circle by turning its pixels white, then draw a black ‘v’ shape, like this:



You can now use the **Replace colour − Local** option to turn the space inside the shape yellow and the surrounding area the mask colour. Finally, give your creation an ‘eye’, which can either be a colour or a section of mask, so that it looks like this:



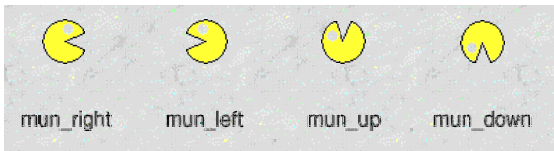
You now have something worth saving, so save the sprite file with the filename ‘MunSprites’.

We now have a sprite showing our character facing right. As we will want to move him in all directions, we’ll need three more sprites showing him facing left, up and down.

You made a copy of the sprite earlier, when it was just a circle. Do the same thing again with the completed sprite, calling this copy ‘mun_left’. To make the copy face in the opposite direction, double-click Select on it to open its sprite window, open a menu on the window, go to **Edit** and choose **Flip horizontally**. This will turn the sprite round and you can close the sprite window again and resave the file.

Completing your First Four Sprites

You could create two more sprites, with the character facing up and down, by rotating the one you already have through 90 degrees, but it may not look very good, especially if you are using mode 12. As the pixels aren’t square, RISC OS has to create one pixel out of two in some places and two pixels from one in others. An alternative method is to copy the ‘mun_x’ sprite that you saved when it was just a plain circle and create a new sprite from that one. You can create a character with a mouth at the top, called ‘mun_up’, and flip it vertically to create ‘mun_down’. You should now have four sprites that look like this:

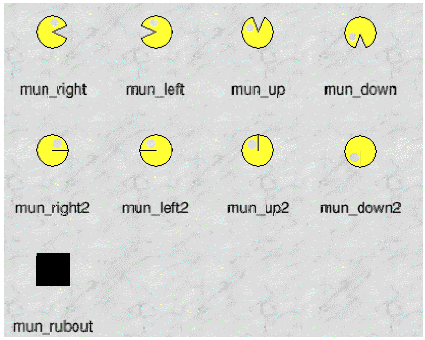


So far, so good, but we don't want our character to wander around the screen with his mouth gawping open all the time. He really needs to open and close it as he goes, so that he appears to be gobbling things up!

This means that we need yet another four sprites with their mouths closed. Fortunately these are easier to draw.

Go back to your circle, make a copy called 'mun_right2' and draw a black line from a little to the right of the centre out to the right-hand side of the circle. Colour the space inside the circle yellow and give it an eye as before. Make a copy, flipped horizontally, called 'mun_left2' and repeat the procedure for 'mun_up2' and 'mun_down2'.

Before you finish this exercise, there is one more job to do. We need a sprite to draw over our character to rub it out when we move it. Open the menu on the sprite file window, go to **New sprite** and create a sprite the same size as your existing ones called 'mun_rubout'. This is just a completely black sprite without a mask or palette. Your sprite file window should now look like this:



Save your sprite file for the last time and close down Paint. We're ready to start looking at the program.

The Munchie Listing

The first program will concentrate on moving our character around the screen. We won't list it in one piece but you can see the whole thing by looking at the file.

```
10 REM > Munchie
20 REM Draws pacman-type figure
30 ON ERROR REPORT:PRINT" at line ";ERL:END
40 PROCinit
50 PROCgame
60 END
70 :
```


This listing shows how we can structure a program by breaking it down into procedures. The main section, from lines 10 to 70, just calls two procedures, PROCinit, which sets up the initial conditions, and PROCgame, which plays the game itself. In later versions we'll call PROCgame repeatedly, using a REPEAT ... UNTIL loop, to give you several tries at the game.

Initialisation and System Sprites

```
 80 DEFPROCinit
 90 *SLoad MunSprites
100 hstep%=10:vstep%=10
110 MODE 27:REM use mode 12 for standard resolution monitor
120 GCOL 8,0
130 ENDPROC
140 :
```

The first action of PROCinit in line 90 is to load the file containing the sprites which we've just created. Line 90 does not quote a full pathname for the file, so the filing system will look for it in the Currently Selected Directory. This would normally be the root directory of your hard disc, so you need to change it to the directory containing the sprite file.

Sprites can be stored in various places in the machine's memory. A properly written application which uses sprites would have its own sprite area within its section of the memory, which other programs wouldn't have access to. The Wimp system, which takes care of the desktop, has an area of this sort.

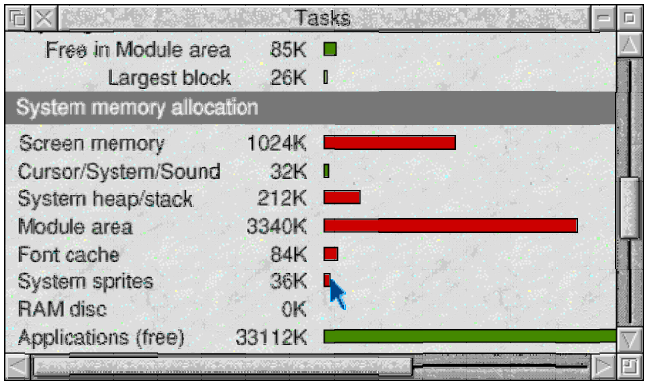
Because we want to keep our program simple, we're using a sprite area called the *System sprites area*. Sprites stored here can be handled using star commands and any program can get to them. It's generally recommend that you don't normally use this area and it's usually configured to be zero bytes in size. Before you can run this program you will have to allocate some space to the system sprite area, which you can do by opening the Task Manager window. Click Select on the icon at the right-hand end of the icon bar and the window will appear. Look under 'System memory allocation' to find 'System sprites' and drag out a red bar to the right of it. If you're using the MunSprites file included in the files, you'll find that you have to drag it out to at least 192k if you're using mode 27, or 96k for mode 12, because the file contains a large background sprite that we'll be using later. Don't drag it out too far because you'll be taking memory away from the rest of the machine.

You can configure your machine to start up with memory already allocated to system sprites **with the command:**

```
*Configure SpriteSize 192k
```

or whatever size you want. This will take effect when you press Ctrl-Reset. To return to normal, type:

```
*Configure SpriteSize 0k
```



drag out a red bar to the right ...

Incidentally, after you've run the program, you can see what is in the system sprites area by typing *SList, and how much spare space you have by typing *SInfo. The *SLoad command removes any sprites already in the area before loading in your new file. If you already have some sprites loaded and you want to add some more to them, use *SMerge instead.

The variables hstep% and vstep% in line 100 are the distance that our character will move horizontally and vertically every time you press a key, and are in OS units. If you think the game would be improved by moving in smaller or larger steps, you just have to change these variables.

This shows the value of setting up such variables at the beginning of the program. It means you can alter the way it works without having to hunt through it.

You will notice that the GCOL command in line 120 has two numbers after it. When this happens, the graphics colour is set by the second number and the first number determines how the colour is plotted. You can find a full list of GCOL numbers in Appendix 1.

Getting the Sprite Onto the Screen

In this case, we're not going to draw lines and shapes but plot sprites, so it doesn't matter what we put as the second number. Making the first number 8 means that when sprites are plotted, the machine will take account of their masks. This won't make any difference in this first version of the program, because we're putting our character on a black background, but it will make a difference later on.


```
150 DEFPROCgame
160 xpos%=200:ypos%=200
170 *SChoose mun_right
180 PLOT 237,xpos%,ypos%
190 REPEAT
200   PROCmove
210 UNTIL FALSE
220 ENDPROC
230 :
```

Having finished setting up the initial conditions, we now get onto PROCgame. The first action of this procedure is to set up two variables, xpos% and vpos%, which determine where the sprite will appear on the screen. These are the horizontal and vertical coordinates of the bottom left-hand corner of the sprite in OS units.

There are two steps to putting a sprite on the screen. We first have to select which sprite we want to draw, then we give a command to *plot* it. Because our sprites are in the system sprite area, we can use a star command to select one and we do this in line 170 with *SChoose followed by the name of the sprite.

The keyword PLOT is a general purpose command for putting graphics on the screen. MOVE, DRAW, CIRCLE FILL etc. are all special types of PLOT command. The first number following PLOT is the type of *PLOT action*. You can find a full list of PLOT actions in Appendix 3. As an example, PLOT 4,100,100 means MOVE 100,100 and PLOT 5,500,600 means DRAW 500,600. Line 180 with Plot code 237 means 'draw the selected sprite with its bottom left-hand corner at (hpos%,vpos%)'.

Lines 160 to 180 between them put our character on the lower left-hand side of the screen, facing to the right. Having done this, we can set about reading the keys and moving him.

Reading Keys and Moving Sprites

We can now see the advantages of structured programming coming into play. The loop in lines 200 to 220 simply calls PROCmove repeatedly until we press Esc to end the program. All we need to know at this stage is that PROCmove reads the keyboard once for each time round the loop and moves the character if we pressed an appropriate key.

```
240 DEFPROCmove
250 char%=INKEY(0)
260 IF char%=ASC("Z") OR char%=ASC("z") PROCleft
270 IF char%=ASC("X") OR char%=ASC("x") PROCright
280 IF char%=ASC("'") PROCup
290 IF char%=ASC("/") PROCdown
300 ENDPROC
310 :
```

When we examine PROCmove, we see that its first action is to get the ASCII code for the key pressed, not using GET but using the INKEY keyword which we met in Section 8. If we used GET, the program would freeze on line 250 until we pressed a key. It would then check the code of the key which we'd pressed, move the character if it was an appropriate key and repeat itself, freezing on line 250 again. It would go round the loop once for every keypress. Because we're using INKEY, the program keeps going round and round the loop regardless of whether we press a key or not.

In the case of this simple program, it doesn't make a lot of difference whether we use GET or INKEY as nothing happens between keypresses. The advantage of using INKEY is that you can make other things happen in the meantime. You might, for example, have a second character on the screen which moves around all the time. If you used GET, you wouldn't be able to make it move continuously between keypresses as the program would spend most of its time frozen in the GET instruction. This will make a difference when we get to the next section and add music to our game.

This program goes round the PROCmove loop continuously, waiting no time at all for you to press a key. It actually checks to see if you've pressed one since it last checked – anything you keyed in would be in the *keyboard buffer*.

Lines 260 to 290 check for one of four keys – Z, X, ' or /, which move our character left, right, up and down respectively. We check for the ASCII codes of both shifted and unshifted Z and X so that the program works whether the CapsLock key is turned on or off. This isn't necessary with the up and down characters, unless you want to play the game with the Shift key held down! It also gets round the problem that if you're running the program on an older machine, the shifted '' is '"', but on a more modern machine it is '@'. If none of these keys has been pressed, the program simply goes back round the loop.

Moving Around

We now look at four procedures for moving the character, one in each direction, and one for rubbing him out before we redraw him:

```
320 DEFPROCrubout(x%,y%)
330 *SChoose mun_rubout
340 PLOT 237,x%,y%
350 ENDPROC
360 :
370 DEFPROCleft
380 PROCrubout(xpos%,ypos%)
390 xpos%-=hstep%
400 IFxpos% MOD (hstep%*2)=0 THEN *SChoose mun_left
410 IFxpos% MOD (hstep%*2)<>0 THEN *SChoose mun_left2
```



```
420 PLOT 237,xpos%,ypos%
430 ENDPROC
440 :
450 DEFPROCright
460 PROCrubout(xpos%,ypos%)
470 xpos%+=hstep%
480 IFxpos% MOD (hstep%*2)=0 THEN *SChoose mun_right
490 IFxpos% MOD (hstep%*2)<>0 THEN *SChoose mun_right2
500 PLOT 237,xpos%,ypos%
510 ENDPROC
520 :
530 DEFPROCup
540 PROCrubout(xpos%,ypos%)
550 ypos%+=vstep%
560 IFypos% MOD (vstep%*2)=0 THEN *SChoose mun_up
570 IFypos% MOD (vstep%*2)<>0 THEN *SChoose mun_up2
580 PLOT 237,xpos%,ypos%
590 ENDPROC
600 :
610 DEFPROCdown
620 PROCrubout(xpos%,ypos%)
630 ypos%-=vstep%
640 IFypos% MOD (vstep%*2)=0 THEN *SChoose mun_down
650 IFypos% MOD (vstep%*2)<>0 THEN *SChoose mun_down2
660 PLOT 237,xpos%,ypos%
670 ENDPROC
```

The four procedures for moving our character are very similar to each other. In each one we call PROCrubout, which deletes the sprite from its current position, change the value of xpos% or ypos% as appropriate, select the correct sprite to make the character face the way it is going and plot it in its new position.

Each procedure has two lines to select a sprite, ensuring that we get an open-mouth and a closed-mouth sprite on alternate keypresses. The technique relies on the fact that the starting values of xpos% and ypos% are multiples of hstep% and vstep%. Every time we move the character, we add or subtract either hstep% or vstep% to or from xpos% or ypos%. This means that, if we divide, say, xpos% by *twice* hstep%, the remainder (which we obtain with MOD) switches between zero and non-zero every time we press a key. Lines 400 and 410 in PROCleft, and the corresponding lines in the other moving procedures, make use of this to select the open-mouth and closed-mouth sprite alternately.

PROCrubout simply plots our black sprite, mun_rubout, over the top of the character to obscure it.

You should now be able to move your character around the screen, using the Z, X, ' and / keys. If you move him off the screen, you will have to bring him back again, as we haven't yet included any checks to limit the values of xpos% and ypos%.

You're probably thinking 'He's got a mouth, he should be able to eat something'. That is our next step.

Collision Detection

Rather than have our character wandering aimlessly around the screen, waiting only for you to press Esc, we'll draw some small white circles for him to gobble up and, when he's eaten the last one, we can print a message asking if you want another go.

Drawing the circles is quite easy. We just have to invent some coordinates, using Basic's *random number* generator, and draw circles in the appropriate places. We'll deal with that part shortly. The matter that requires rather more thought is *collision detection* – knowing when our character has found one of the circles.

There are two ways of doing this. One way is by examining the colour of each pixel in front of the character and checking to see if one of them isn't black. We could do this with Basic's POINT keyword, like this:

```
col%=POINT(x%,y%)
```

The value of col% will be the logical colour number of the pixel at (x%,y%).

This would certainly work with our present program, where we have a plain background. If we were moving our character, say, right, we would simply examine the colours of each pixel to the right of the sprite and check to see if any of them weren't the background colour.

Things would get more complicated, though, when we moved on to using a patterned background, as we will do later. We could ensure that the background didn't include the colour of the circles and check for that one, but that would restrict our design. Things would also get even more difficult if, instead of a plain circle, we plotted another multi-coloured sprite for our character to gobble up.

Storing the Positions of Circles

The other technique for detecting a collision is to keep a note of where everything is on the screen and simply work it out. We can use an array variable with 10 elements to store the coordinates of 10 circles and we know where the bottom left-hand corner of the character is, as its coordinates are stored in xpos% and ypos%.

In this program we will give each circle a radius of 10 OS units. We'll also introduce two new variables, width% and height%, to store the width and height of the character. Remembering, then, that xpos% and ypos% are the coordinates of the character's bottom left-hand corner, we have found a circle if:

x coordinate of circle > xpos%-10 (circle is not wholly to the left of the sprite)

AND:

x coordinate of circle < xpos%+width%+10 (circle is not wholly to the right of the sprite)

AND:

y coordinate of circle > ypos%-10 (circle is not wholly below the sprite)

AND:

y coordinate of circle < ypos%+height%+10 (circle is not wholly above the sprite)

When all four of these conditions are met, a collision has occurred between the character and the circle.

We don't need to look at the entire program again. PROCmove and PROCrubout are unchanged and the four procedures for moving the sprite have only had one line added to each of them. We'll be adding a number of lines to the beginning of the program, however.

For the same reason, we won't list the whole game again, but there are some changes to the first part:

```
10 REM > Munchie2
20 REM Draws pacman-type figure
30 ON ERROR REPORT:PRINT" at line ";ERL:END
40 PROCinit
50 REPEAT
60   PROCgame
70 UNTIL (char%AND &DF)<>ASC("Y")
80 END
90 :
```

There are only two lines to add to the main part of the program, which are now lines 50 and 70. Instead of PROCgame running indefinitely, it will now finish when the last circle has been gobbled up and asks the question 'Another go?'. The ASCII code for the key which you then press is put into variable char%. Line 70 forces this to upper

case, as we saw in Section 9, so that it doesn't matter whether the Caps Lock key is on or off. If you pressed 'Y', the loop repeats and you get another game. Any other keypress will end the program.

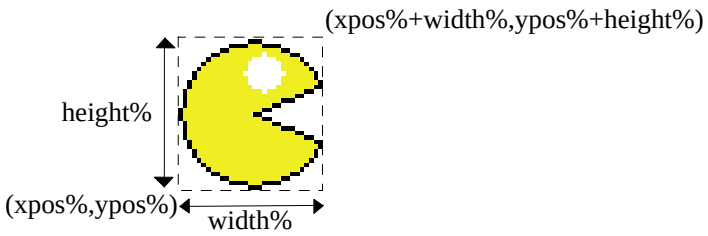
```
100 DEFPROCinit
110 *SLoad MunSprites
120 dotnum%=9
130 width%=68:height%=68
140 hstep%=10:vstep%=10
150 DIM dot%(dotnum%,1)
160 MODE 27:REM use mode 12 for standard resolution monitor
170 ENDPROC
180 :
```

PROCinit has had three lines added and one removed. The old line 120 contained an instruction GCOL 8,0 which told the machine to plot sprites with a mask. This has now been moved into PROCgame.

The new line 120 sets up a variable dotnum%, which contains one less than the number of circles which we want to plot. It's one less because the 10 circles will be numbered 0 to 9.

We need to know the size of the character sprite for collision detection, so we introduce width% and height%, which are in OS units. Whether you're using mode 27 or mode 12, the sprite measures 68 units square (unless you created your own and made it a different size), so width% and height% will both be this value.

Finally, in line 150 we define a two-dimensional array variable to hold the coordinates of all the circles. The elements of an array are numbered from zero so, if dotnum% is 9, they range from dot%(0,0) to dot%(9,1), enough to hold two coordinates for 10 circles. We'll use the elements whose second number is 0 for the x coordinate and the one whose second number is 1 for the y coordinate. The x coordinate of circle 5, for example, would be in dot%(5,0) and its y coordinate in dot%(5,1).



xpos% and ypos% refer to the bottom left-hand corner of the sprite


```
190 DEFPROCgame
200 dotleft%=dotnum%+1
210 CLS
220 GCOL 7
230 FOR num%=0 TO dotnum%
240   PROCsetdot(num%)
250 NEXT
260 GCOL 8,0
270 xpos%=200:ypos%=200
280 *SChoose mun_right
290 PLOT 237,xpos%,ypos%
300 start%=TIME
310 REPEAT
320   PROCmove
330 UNTIL dotleft%=0
340 PRINT TAB(10,25)"You took ";(TIME-start%)/100" seconds"
350 PRINT TAB(10,26)"Another go? (y/n)"
360 REPEAT
370   char%=GET
380 UNTIL (char% AND &DF)=ASC("Y") OR (char% AND &DF)=ASC("N")
390 ENDPROC
400 :
```

The new version of PROCgame begins with the definition of yet another variable. This one, dotleft%, holds the number of circles remaining on the screen at any point in the game. It starts life as one more than dotnum% – 10 for 10 circles – and is decreased by 1 each time we eat up a circle. The procedure's main loop, which calls PROCmove, stops repeating when dotleft% is reduced to zero and all the circles are gone.

Drawing Circles

At the start of the game, we clear the screen and set the graphics foreground colour to white, ready to draw our circles. The drawing is done by PROCsetdot which we call once for each circle, passing to it the number of the circle in its parameter num%. That procedure, as we shall see shortly, both draws the circle on the screen and puts its coordinates into two elements of the array.

Having drawn the circles, we get to line 260 which contains the GCOL instruction telling the machine to plot sprites with masks.

The next few lines are the same as in the old listing of PROCgame, setting up the starting position of the character with xpos% and ypos%, choosing the right-facing sprite and plotting it. There is one extra line before we get to the loop that calls PROCmove. It would be rather nice to tell the player how long he took to eat up 10 circles so that he could try to beat his time on the next go. We saw in Section 3 how the

Basic variable TIME increases every one hundredth of a second. We store its value at the start of the game in variable start%.

In the earlier version, we called PROCmove with a REPEAT ... UNTIL FALSE loop which kept on repeating until we pressed Esc or reset the machine. There was no other way of terminating our character's aimless wanderings around the screen! We now have a definite point at which to end the game – when the last circle has been eaten. Line 330 takes care of this, by allowing the program to continue when dotleft% has been reduced to zero.

The remainder of the procedure reads TIME again and prints how many seconds have elapsed since the start of the game, then prompts you to press 'Y' or 'N'. The final loop in the procedure waits for the correct keypress.

Plotting at Random

In PROCsetdot we invent the x and y coordinates for a circle and plot it on the screen. We wouldn't want the circles to appear in the same places every time or the game would soon become very boring. What we need for our coordinates are *random numbers* and we generate these with Basic's RND keyword.

An expression such as RND(6) will produce a whole number between 1 and 6. The number is not chosen completely at random, but the machine uses such a complicated procedure for producing it that it will seem like it. If the number in brackets is (1), the expression produces a fraction between 0 and 1.

If we want to produce a number between, say, 4 and 8, we could use:

```
x%=RND(5)+3
```

Because the random number is between 1 and 5, adding 3 to it will give us a result in the range that we want.

We use this technique in PROCsetdot.

```
410 DEFPROCsetdot(n%)
420 dot%(n%,0)=RND(1200)+40
430 dot%(n%,1)=RND(600)+300
440 CIRCLE FILL dot%(n%,0),dot%(n%,1),10
450 ENDPROC
460 :
```

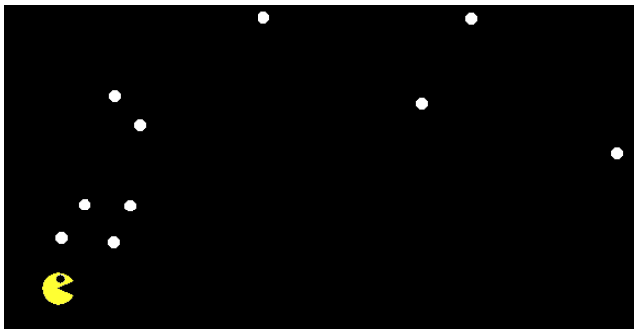
Line 420 generates an x coordinate between 40 and 1240 which will keep the circles away from the sides of the screen. Line 430 not only keeps them away from the top, but

ensures that none of them will be put in the bottom third of the screen. This means we can be certain that we will not begin a game with a circle underneath the starting position of the character, which would spoil it.

You will recall that PROCgame calls this procedure once for each circle, passing the circle number to it in n%. We use this number to select which elements of dot%() are used to store the coordinates.

Spotting Collisions

If you make these changes and run the game, you will find that it behaves just the same as before, except that 10 circles appear on the screen in random places. Nothing happens, however, when you find one of them. A circle may get rubbed out when you move the character over it, but the program doesn't know it's gone and the game won't end when you've rubbed out the last one. This is because we haven't yet built in any collision detection.



... ten circles appear on the screen in random places

Add an extra line to each of the four 'moving' procedures, PROCleft, PROCright, PROCup and PROCdown, e.g.:

```
600 DEFPROCleft
610 PROCrubout(xpos%,ypos%)
620 xpos%-=hstep%
630 PROCcollision
640 IFxpos% MOD (hstep%*2)=0 THEN *SChoose mun_left
650 IFxpos% MOD (hstep%*2)<>0 THEN *SChoose mun_left2
660 PLOT 237,xpos%,ypos%
670 ENDPROC
680 :
```

The extra line calls the collision detection procedure. It doesn't affect the way PROCleft etc. works but just arranges for a collision to be checked for each time the character is moved.

Now we just add one more procedure and a function:

```
  960 DEFPROCcollision
  970 FOR num%=0 TO dotnum%
  980   IF FNfound(num%) THEN
  990     GCOL 0
 1000     CIRCLE FILL dot%(num%,0),dot%(num%,1),10
 1010     GCOL 8,0
 1020     VDU 7
 1030     dot%(num%,0)=-1
 1040     dotleft%-=1
 1050   ENDIF
 1060 NEXT
 1070 ENDPROC
 1080 :
 1090 DEFFNfound(num%)
 1100 IF dot%(num%,0)>=xpos%-10 AND dot%(num%,0)<=xpos%+width%+10 AND dot%(num%
,1)<=ypos%+height%+10 AND dot%(num%,1)>=ypos%-10 AND dot%(num%,0)>0 THEN =TRUE
ELSE =FALSE
```

The procedure PROCcollision is called each time we recalculate the position of our character. This procedure delegates detection of whether or not a collision has occurred to a function, FNfound, and takes the appropriate action if it has.

We need to check the position of each circle in turn to see if our character has collided with it, so we use a FOR ... NEXT loop to go through all the circle numbers. We pass each number to FNfound, which returns a value, TRUE if there is a collision or FALSE if there isn't.

Lines 980 to 1050 are an example of conditional execution over several lines, which we first met in Section 3. Lines 990 to 1040 are called if a collision is detected. Our first task is to rub out the circle. Redrawing the character might do this, by being plotted over the top of it, but we could easily have the situation where only part of a circle gets rubbed out, so we must do it properly.

In this version of the program, we can simply draw a black circle over the top of the white one. Line 990 sets the colour, line 1000 draws the circle and line 1010 returns the GCOL setting to the one required for plotting a sprite with a mask.

Our program would be greatly improved if the machine made some sort of sound every time we gobbled up a circle. The simplest way of doing this is with a VDU 7 command, which is provided by line 1020.

Getting Rid of the Circle

Although we've rubbed out the circle, we haven't done anything about its coordinates. Because FNfound works purely by checking coordinates, it could think it had found another collision if we were to steer the character back to where the circle was, even though we'd made it invisible. We need to do something.

The answer is to 'move' the coordinates of the circle out of the way. We'll do this simply by making the x coordinate –1, which would put the circle just off the left-hand side of the screen. In practice, there's nothing to stop the character moving off the left-hand edge and thinking it has found a collision, so we'll arrange FNfound so that it refuses to recognise a collision with a circle whose x coordinate is less than zero.

The only thing left in this version of the program is to get down to the nitty-gritty of FNfound and how it detects a collision, We listed all the criteria which must be met a few pages back. Line 1100 combines all of them into one line, using AND keywords so that they must all be correct, and also adds an extra one. The x coordinate of the circle must be greater than zero, for reasons that we've just discussed. If all five of these conditions are true, the function ends by returning TRUE, otherwise it returns FALSE.

Your program should now work as intended, deleting circles when you touch them and beeping as it does so. When you've eaten all the circles it should tell you how long you took and invite you to try again.

You could, of course, replace the circles with anything you like, provided you can draw it, by modifying PROCsetdot. If the object is larger than the circles, though, you'll have to modify line 1100 in FNfound.

What we really need to do something about, though, is that plain black background.

Adding a Patterned Background

First produce your background. You can put anything you like on the screen, though it had better be in the same mode as your game to avoid complications. You then have to save the screen.

Back in Section 6 we met a program called 'boxes' which drew multi-coloured rectangles on the screen. You may consider this a suitably striking background for moving sprites around in front of, in which case, you will find that the MunSprites file already contains a sprite which holds a copy of the entire screen produced by this program (known as a *screendump*), called 'mun_bg'.

Alternatively, you may wish to create your own picture. If you write a program to do this, put this line on the end:

```
*ScreenSave Background
```

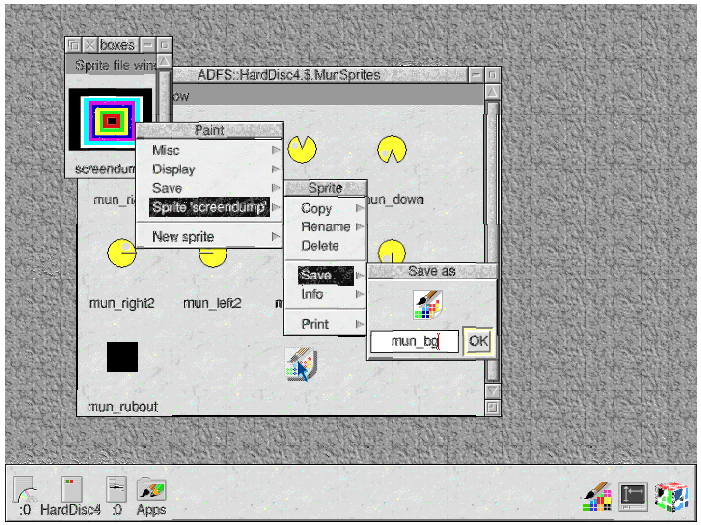
'Background' is a filename. you can, of course, use any other filename you like.

This command saves the screen as a sprite in the Currently Selected Directory (unless you give it a full pathname). It actually saves the current graphics window, so if you use it in a program where this is less than the full screen size, reset it to full size first by using a VDU 26 command.

You'll find that your new file has a 'sprite' filetype and, when you load it into Paint, you'll discover that it contains a single sprite called 'screendump'. You can, of course, use Paint to modify it if you think you can improve it.

It would be more convenient if all the sprites in our game were in the same file. You can add the background sprite to the MunSprites file with a Save operation.

Load your 'MunSprites' file into Paint so that its sprite file window is open. Click Menu over the 'screendump' sprite in its sprite file window and go to the **Sprite** '**screendump**' submenu and through to **Save**. When you have the Save box open, change the sprite name to 'mun_bg' and drag it into your MunSprites file window. This will copy the background sprite across to your MunSprites file, which you should then resave. We've now finished with the 'Background' file and you can delete it.



... drag it into your MunSprites file window

This new version is included in the files as Munchie3. The first changes to the program are at the beginning of PROCgame, which now looks like this:

```
190 DEFPROCgame
200 dotleft%=dotnum%+1
210 PROCbackground
220 GCOL 3,7
230 FOR num%=0 TO dotnum%
240   PROCsetdot(num%)
250 NEXT
260 GCOL 8,0
270 xpos%=200:ypos%=200
280 PROCget_bg
290 *SChoose mun_right
300 PLOT 237,xpos%,ypos%
310 start%=TIME
320 REPEAT
330   PROCmove
340 UNTIL dotleft%=0
350 PRINT TAB(10,25)"You took ";(TIME-start%)/100" seconds"
360 PRINT TAB(10,26)"Another go? (y/n)"
370 REPEAT
380   char%=GET
390 UNTIL (char% AND &DF)=ASC("Y") OR (char% AND &DF)=ASC("N")
400 ENDPROC
410 :
```

Lines 210 and 220 have been altered and there is a new line which is now line 280.

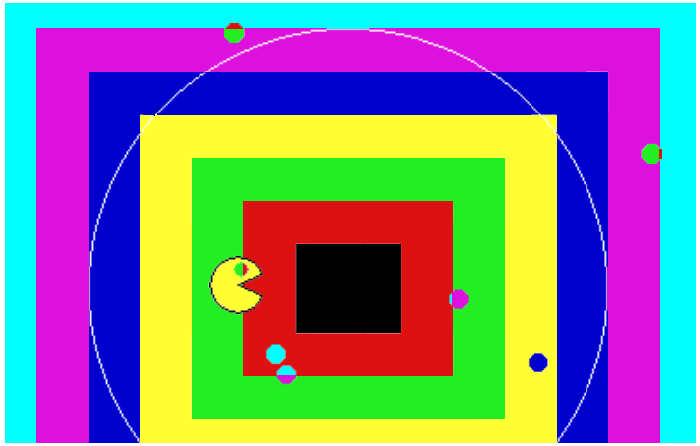
In the previous version, PROCgame started by clearing the screen. We now call PROCbackground, which simply plots the background sprite to redraw the background.

Making the Circles Stand Out

After drawing the background, the next job is to draw the circles, but we now have a snag. If we draw a white circle in a place where the background is white it will be invisible. This is where GCOL followed by two numbers comes to the rescue. You will recall that the second number refers to the colour and the first number tells the machine how to plot it. The first number in line 220 is a 3, which means the colour number is *exclusive OR*ed with what is already on the screen.

We met the exclusive OR function in Section 9. It compares each bit of one number with the corresponding bit of the other and produces a bit which is zero if they are the same and 1 if they're opposite. Because the second number after the GCOL keyword is 7 (binary 111), the lowest three bits of what is already on the screen are inverted. If the

background is black (colour 0), it becomes white (colour 7). Conversely, a white background turns black. If the background is, say, red (colour 1), it turns cyan (colour 6). This means that our circles will always stand out from the background, whatever colour it is. If a circle covers the boundary between two colours, the circle itself will have two colours!



… our circles will always stand out from the background ...

Replacing the Background

Because we now have a background with some detail in it, we must have a way of putting it back in place of our character when we delete him to move him. There is more than one way of doing this and we'll take a look at two ways.

The first technique is to save the piece of background where you're going to plot a sprite, using a special sprite for the purpose, and put it back again when your character moves on. This is the purpose of PROCget_bg, which we call in line 280 before we plot the character sprite in its starting position.

We'll look at PROCget_bg later. All we need to know for now is that it saves the area of the screen that our character is going to cover, putting it in the 'mun_rubout' sprite. This is very convenient, as it means that we don't need to modify PROCrubout. That procedure removes the character sprite from the screen by plotting the mun_rubout sprite over it. This sprite was originally black, but in this version of the program it contains whatever background was present on that part of the screen before the character sprite was plotted over it.

We need to add one more line to each of our four 'moving' procedures, so that they all look like this:


```
710 DEFPROCright
720 PROCrubout(xpos%,ypos%)
730 xpos%+=hstep%
740 PROCcollision
750 PROCget_bg
760 IFxpos% MOD (hstep%*2)=0 THEN *SChoose mun_right
770 IFxpos% MOD (hstep%*2)<>0 THEN *SChoose mun_right2
780 PLOT 237,xpos%,ypos%
790 ENDPROC
800 :
```

Once the sprite in its old position has been rubbed out and the new position defined, the portion of the background in the new position is saved by PROCget_bg before the sprite is plotted over it.

Before we come to our two new procedures, we must make a change to PROCcollision. The old version deleted a circle by drawing a black circle over it – not a very good idea when we're using a patterned background! We need to put the background back where the circle was.

We could save a small sprite to hold the background behind each circle, but we'll use a simpler idea. We will set the graphics window so that it just covers the circle and plot the background sprite.

The beginning of PROCcollision now looks like this:

```
1010 DEFPROCcollision
1020 FOR num%=0 TO dotnum%
1030   IF FNfound(num%) THEN
1040     VDU 24,dot%(num%,0)-10;dot%(num%,1)-10;dot%(num%,0)+10;dot%(num%,1)+10
;
1050     PROCbackground
1060     VDU 26
1070     VDU 7
```

The rest of the procedure is unchanged.

We've changed the content of three lines which are now numbered 1050 to 1070.

Plotting the Background Over a Circle

In line 1040, we set the graphics window to the size of the circle. As we saw in Section 10, we do this by sending a sequence of numbers to the VDU circuitry. The first is 24 to tell it that we're setting the graphics window. This is followed by eight numbers in pairs – don't forget the semi-colons which tell Basic to send two 8-bit numbers for each

coordinate. The coordinates are in the order minimum x, minimum y, maximum x and maximum y, and are each 10 OS units greater or less than the x or y coordinate of the centre of the circle. You will have to change this line if you replace the circles with something larger.

Having set the graphics window, we can give the instruction to plot the background sprite, via PROCbackground, but only a tiny part of it will actually be plotted. Not only is this quicker; it also means we will not rub out the other circles and the character sprite. The VDU 26 command in line 1070 restores the graphics window after we've finished so that it covers the whole screen.

All that remains now are our two new procedures:

```
1170 DEFPROCbackground
1180 *SChoose mun_bg
1190 PLOT 237,0,0
1200 ENDPROC
1210 :
1220 DEFPROCget_bg
1230 MOVE xpos%,ypos%
1240 MOVE xpos%+width%,ypos%+height%
1250 *SGet mun_rubout
1260 ENDPROC
```

PROCbackground plots the background simply by selecting the background sprite and plotting it, taking the graphics origin as the position of its bottom left-hand corner, because it contains the entire screen.

Saving the Background as a Sprite

PROCget_bg introduces us to the command *SGet. This saves part of the screen as a sprite, taking the two most recent positions of the graphics cursor as opposite corners. Line 1230 moves the graphics cursor to the bottom left-hand corner of the area we wish to save and line 1240 moves it to the top right-hand corner.

The *SGet command in line 1250 stores the area where we're going to plot our character as sprite 'mun_rubout'. As this sprite already exists, its previous contents are overwritten by the new sprite. If the sprite didn't already exist, it would be created.

This is a very thorough way of saving and replacing the background if you have other objects on the screen and you don't know where they are. Whatever is behind the character will reappear when he moves on, whether or not it's part of the original background sprite. Unfortunately it's also rather slow. Every time you move the character, you have to plot a sprite, grab a sprite and plot the character sprite, and that's

ignoring the small amount of time it takes to calculate its new position and deal with anything else which may be happening! The character may not show up clearly, as it spends only a small part of its time actually on the screen and the rest of it invisible, while the background is being redrawn and resaved.

Another problem manifests itself if you're using an older machine with the slower ARM 2 processor. You may find that the time taken for each move is actually longer than the repeat key time delay on your keyboard. This means that, if you hold a key down to send your character across the screen, he will continue moving after you've taken your finger off the key because you've stored up keypresses faster than the program could handle them.

Clearly we could do with a faster way of doing things.

Faster Background Replacement

When we rubbed out the circles in this version, we set the graphics window to just cover a circle and simply plotted the background sprite. There is no reason why we shouldn't use this method to rub out our character as well.

Instead of adding to the program and further complicating it, we're actually simplifying it this time and it will end up shorter than before. We'll call this version Munchie4.

This is what you would get if you were to go through the program and delete all the lines which call PROCget_bg. There should be five of these; one in PROCgame and one in each of the four 'moving' procedures. You could then delete PROCget_bg itself from the end of the program.

We're no longer going to save the background before plotting the sprite, and as we don't rub it out by drawing the 'mun_rubout' sprite over it, you could, if you wished, delete the sprite from the sprite file.

All we need to do now is a new version of PROCrubout which looks like this:

```
550 DEFPROCrubout(x%,y%)
560 VDU 24,xpos%;ypos%;xpos%+width%;ypos%+height%;
570 PROCbackground
580 VDU 26
590 ENDPROC
600 :
```

We've already seen the principle of this procedure when we were rubbing out circles in the previous version. Line 560 sets the graphics window so that it just contains the character sprite. Line 570 plots the background sprite but the plotting action is confined to the graphics window.

If you had another object moving around the screen which could go behind your character, you would have to redraw both of them. Start off by replacing the background over both of them and then redraw first the back object, then the front one.

Dealing With a Bug

If you've tried to produce Munchie4 by modifying Munchie3 as we've just seen, you would find a problem with this version as it stands. If you allow the character to wander off the screen, all the circles disappear!

To understand why, look at PROCrubout. Its action consists of setting the graphics window to cover the character in its present position, calling PROCbackground to plot the background sprite in the window, then resetting the window to cover the whole screen, by means of the VDU 26 command. This last action is necessary so that we can plot the sprite in its new position.

If the new position of the sprite is partially off the screen, our next call to PROCrubout will attempt to set a graphics window which is also partially off the screen. RISC OS does not allow this and ignores the command completely, leaving the window at its previous setting, covering the entire screen. The result is that the call to PROCbackground plots the background sprite over everything, obliterating the circles.

The best way out of this problem is to prevent the character going off the screen! We can do this by adding some checks to PROCmove:

```
  470 DEFPROCmove
  480 char%=INKEY(0)
  490 IF (char%=ASC("Z") OR char%=ASC("z")) AND xpos%>=hstep% PROCleft
  500 IF (char%=ASC("X") OR char%=ASC("x")) AND POINT(xpos%+width%+hstep%,0)>=0
PROCright
  510 IF char%=ASC("'") AND POINT(0,ypos%+height%+vstep%)>=0 PROCup
  520 IF char%=ASC("/") AND ypos%>=vstep% PROCdown
  530 ENDPROC
```

Lines 490 and 520 are concerned with moving the sprite left and down. We simply have to check whether or not the new position of the bottom left-hand corner (xpos%,ypos%) has a coordinate below zero and avoid calling the moving PROCleft or PROCdown if it has. If the other two lines worked in the same way, they would need to take the screen size into account and you would have to remember to alter them if you rewrote the game to run in another screen mode.

We can avoid having to do this by using the POINT expression which we looked at when we were considering ways of detecting collisions. We follow the POINT keyword with the x and y coordinates of the point we're interested in and the expression produces a number which tells us the colour of the pixel at that position. The colour doesn't interest us in the slightest but if the point is off the screen, the number is −1. This gives us a way of checking if our character would be drawn partially off the screen without having to know the screen size.

Note the extra pair of brackets round the first two expressions in each line. These are necessary to ensure that the OR comparison is made before the AND one.

Using a Separate Sprite Area

As we saw earlier, the sprites used by this program were loaded into the system sprite area so that they could be handled by star commands. This is not a particularly efficient way of doing things because some memory has to be specially allocated to this area, which will be wasted after you've quit the program.

A properly written program which use sprites has its own user sprite area within the program memory. This has to be specially set up and the sprites loaded and manipulated using an operating system command called a *software interrupt* or *SWI*. This type of command is introduced in section 14. The SWI which handles sprites is called OS_SpriteOp. Because this SWI does everything associated with sprites, it is very comprehensive and a full description is outside the scope of this guide but section 14 will show you how to use it in this program. A full account of all software interrupt routines can be found in the *RISC OS Programmer's Reference Manual*, available on CD-ROM from RISCOS Ltd.

We also still have the problem of the whereabouts of the MunSprites file. We either have to keep this file in the currently selected directory or write the program to include its full pathname. This makes it more difficult to copy the program or move it about on the hard disc. We'll see how to solve this problem in section 15.

12

Adding Sound

The game that we produced in section 11 may have had some good graphics but it definitely lacked something which any decent game has. It was remarkably silent. To remedy this we need to look, not surprisingly, at the SOUND command.

Your RISC OS machine is capable of producing eight sounds at the same time, but only one of them is usually working. There is a very good reason for this. Acorn machines pride themselves on being extremely fast computers, thanks to their ARM or StrongARM processors. A sound channel takes a lot of the machine's processing power and slows it down a little bit. Eight sound channels would slow it down a lot, even if they weren't actually making sounds, so seven of them are left turned off unless they're needed. One is always active so that it can produce the 'beep'.

Raising Your VOICE

Basic refers to a sound channel as a VOICE. The first thing to understand is that the rest of RISC OS, including the part that processes star commands, uses the word 'voice' to refer to the waveform which produces a particular sound, and refers to a sound channel as a channel!

Try typing;

```
*Voices
```

and you should see something like this:

```
          Voice      Name
1           1   WaveSynth-Beep
            2   StringLib-Soft
            3   StringLib-Pluck
            4   StringLib-Steel
```

```
                5   StringLib-Hard
                6   Percussion-Soft
                7   Percussion-Medium
                8   Percussion-Snare
                9   Percussion-Noise
^^^^^^^^ Channel allocation map
```

The names on the right refer to the sound waveforms that the machine knows about. They are built into the machine's ROM and are always there. If you've been playing a game with its own sounds, it's possible that some of the waveforms are still loaded and will show up in the list. The game's directory may well contain some module files for producing these sounds and you can add them to the list by loading them, either by double-clicking on them or using the *RMLoad command.

Waveform Names and Numbers

To the left of the waveform names are the numbers by which they can be called. The extra number 1 to the left of 'WaveSynth-Beep' shows that channel 1 has this waveform assigned to it. You can change this by typing, say:

```
*ChannelVoice 1 5
```

or alternatively:

```
*ChannelVoice 1 StringLib-Hard
```

If you now type *Voices again, you will see that the '1' has moved down so that it's opposite waveform 5, 'StringLib-Hard'.

Now press Ctrl-G to produce a beep and you'll hear a great difference! Channel 1 is the beep channel and anything you do to this channel affects the beep. Any program you run now will produce a beep with the new sound, unless it deliberately reprograms channel 1 itself.

Don't worry, you haven't broken your machine. You can restore the beep to its nice old soft self (unless of course you prefer the new one!) by typing:

```
*ChannelVoice 1 1
```

to put things back as they were.

Basic has two very similar keywords, VOICES and VOICE, which should not be confused. VOICES sets the number of active sound channels, which must be 1, 2, 4 or 8.

VOICE is the Basic equivalent of the star command *ChannelVoice, but can only be followed by the *name* of the waveform in quotes, not its number, for example:

```
VOICE 1,"StringLib-Hard"
```

The SOUND Command

Now that we've investigated voices and channels, we're ready to make a sound. We'll stick with sound channel 1 for now and you can assign whichever waveform you like to it for the purpose of this experiment.

There are four numbers which follow a sound command − the channel number, amplitude, pitch and duration. If you don't want the sound to be made instantly, you can add a fifth one to represent a delay.

Try typing:

```
SOUND 1,-15,53,20
```

You should get a note lasting one second which is, in fact, middle C.

The first number is the channel number and indicates that the command is for channel 1, which we have to use because it's the only one active at the moment. The number for the amplitude, or volume, is -15. Amplitude can be represented by a negative number, from zero for silence to -15 for maximum loudness. It is also possible to use a positive number from 256 to 383. In this case the number is logarithmic, with each increase of 16 doubling the sound amplitude. We'll stick with negative numbers in this guide.

The pitch number 53 is the number for middle C. Pitch is represented by a number from 1 to 255, as follows:

Octave number

Note	**1**	**2**	**3**	**4**	**5**	**6**
A		41	89	137	185	233
A#		45	93	141	189	237
B	1	49	97	145	193	241
C	5	53	101	149	197	245
C#	9	57	105	153	201	249
D	13	61	109	157	205	253
D#	17	65	113	161	209	
E	21	69	117	165	213	
F	25	73	121	169	217	
F#	29	77	125	173	221	
G	33	81	129	177	225	
G#	37	85	133	181	229	

Octave 2 is the one containing middle C.

It is also possible to represent pitch by a number from &100 (256) to 32767 (&7FFF), in which case middle C is &4000. You may have to use pitch numbers in this range if you want to experiment with sound from module files.

Durations and Timings

The final number represents the duration of the note and is in twentieths of a second. As this number is 20, our note lasted for one second. Naturally, if you change the number to 10, the note will last for half a second.

You can also change the pitch to, say, D above middle C by changing the pitch number to 61. You may like to try playing the first two notes of a tune by putting two sound commands one after the other, like this:

```
SOUND 1,-15,53,10:SOUND 1,-15,61,10
```

Unfortunately you will find that the second sound is made immediately and cancels out the first so you only get one note. What we need is a way of delaying the second sound.

BEAT, BEATS and TEMPO

The sound system takes its timings from the *beat counter*. This counter starts from zero and counts up to a number determined by the Basic keyword BEATS, at which point it's reset to zero and starts again.

If you type:

```
PRINT BEATS
```

you will almost certainly get the answer zero, suggesting that the beat counter doesn't count at all. This is because BEATS is set to zero when the machine is reset or Esc pressed.

Before we set a value for BEATS, we need to know how fast the beat counter counts. This is determined by another Basic variable, TEMPO. If you type:

```
PRINT TEMPO
```

you will probably get 4096, which in hexadecimal form is &1000. TEMPO is measured in beats per centi-second but the three lowest hexadecimal digits are a fractional part of the number. In other words, if TEMPO is &1000, the beat counter increases at one beat per centi-second (or 100 beats per second). If you double TEMPO to &2000 (or 8192), the beat counter counts at twice the rate and if you halve it to &800 (or 2048), it counts at half a beat per centi-second, 50 beats per second. For our experiments, we may as well leave it as it is.

Now that we know how fast the counter runs, we can decide how often we want it to be reset. Let's reset it once every second when it's reached 100, which we do by typing:

```
BEATS 100
```

We can now delay the second note by adding a fifth number to its command, the *after* parameter. This tells the sound system not to produce the note until the beat counter has reached a certain value. If we sound the first note for half a second, we want to delay the second one by this amount, that is 50 beats.

After setting BEATS to 100, you could try typing the following:

```
SOUND 1,-15,53,10:SOUND 1,-15,61,10,50
```

Notice the extra number on the end of the second command, delaying the sound by 50 beats.

Unfortunately, this won't work properly. If you try it you will probably either get a shortened first note or no first note at all. The problem is that the beat counter is counting all the time. If it's somewhere between zero and 50 when the commands are given, the second note will be sounded as soon as it gets to 50 which will shorten the first note. If it has already passed 50, the second note will be sounded immediately, so there will be no first note at all.

The way out of this problem is to wait until the counter is reset to zero and then give the commands. We can read the counter with the Basic keyword BEAT (not to be confused with BEATS), and wait for it to return to zero using a REPEAT ... UNTIL loop like this:

```
REPEAT UNTIL BEAT=0:SOUND 1,-15,53,10:SOUND 1,-15,61,10,50
```

There may be a slight pause before the first note, while the loop waits for the counter to come back round to zero but you should now have both notes sounding correctly.

An Arpeggio

By using four half-second notes and with the beat counter repeating every two seconds, we can play an arpeggio:

```
10 REM Arpeggio
20 REM plays an arpeggio
30 BEATS 200
40 REPEAT UNTIL BEAT=0
50 SOUND 1,-15,53,10
60 SOUND 1,-15,69,10,50
70 SOUND 1,-15,81,10,100
80 SOUND 1,-15,101,10,150
```

Line 40 waits until the beat counter has been reset to zero, then the remaining lines give their sound commands. The first note is sounded without a delay, the second with a delay of 50 beats (50 centi-seconds as we haven't altered the value of TEMPO), the third with a delay of 100 beats and the fourth a delay of 150 beats.

We can think of the time for 200 counts as one bar of music. If we wanted to add a second bar, we could wait for BEAT to come round to zero again and add some more notes.

A Simple Tune

We are now ready to think in terms of playing a tune. Frère Jaques is a good one as it's very simple, and to make it even easier, we'll play it in C major (no sharps or flats!).

```
 10 REM > F_Jaques
 20 REM Plays a simple tune
 30 ON ERROR REPORT:PRINT" at line ";ERL:END
 40 BEATS 200
 50 REPEAT
 60   REPEAT UNTIL BEAT=0
 70   PROCbar
 80   REPEAT UNTIL BEAT<>0
 90 UNTIL FALSE
100 :
110 DEFPROCbar
120 beat%=0
130 REPEAT
140   READ pitch%
150   IF pitch%=-1 THEN
160   RESTORE +1:PROCbar:ENDIF:ENDPROC
170   ELSE
180   READ dur%
190   IF pitch%<>0 SOUND 1,-15,pitch%,dur%,beat%
200   beat%+=dur%*5
210 UNTIL beat%>=200
220 ENDIF
230 DATA 53,10,61,10,69,10,53,10
240 DATA 53,10,61,10,69,10,53,10
250 DATA 69,10,73,10,81,20
260 DATA 69,10,73,10,81,20
270 DATA 81,5,89,5,81,5,73,5,69,10,53,10
280 DATA 81,5,89,5,81,5,73,5,69,10,53,10
290 DATA 53,10,33,10,53,10,0,10
300 DATA 53,10,33,10,53,10,0,10
310 DATA -1
320 ENDPROC
```


The notes are all read in from DATA statements. You could replace the numbers with your own to make the program play any tune you like. Each note has two numbers referring to the pitch (which you can get from the table near the beginning of this section) and the duration, in twentieths of a second.

To simplify the DATA statements, one line is used for each bar. The eight lines in fact consist of four pairs of identical lines, simply because every bar in this tune repeats itself.

The list of data is finished off with a -1. When the program reads this, it starts reading the data from the beginning again so that the program repeats itself *ad infinitum* (or *ad nauseam*!).

Line 40 sets the limit of the beat counter to 200. This allows us to play four half-second notes in one bar.

The REPEAT ... UNTIL loop between lines 50 and 90 calls PROCbar each time the beat counter is reset. Line 60 waits for the reset, line 70 calls PROCbar and line 80 ensures that the beat counter has moved on from zero before returning to the beginning of the loop. If we didn't do this the program may try to play several bars at the same time.

Setting the Timing for Each Note

PROCbar uses variable beat% to work out how much delay should be given to each note, by adding up the durations of the preceding notes in the bar. For each note, line 140 reads the pitch data. If it reads -1, it's reached the end of the data. Line 160 then restores the DATA pointer to the beginning of the DATA statements and calls PROCbar again (this is called *recursion*) to restart the piece.

Assuming that the pitch data wasn't -1, line 180 then reads the next number, which is the duration of the note. Armed with these two pieces of information, line 190 is able to construct the SOUND command for the note. Each note is given a delay, taken from the value of variable beat%.

After the command, beat% is increased to take account of the length of the note. The variable dur%, which sets the duration of the note in the SOUND command, is in twentieths of a second so it has to be multiplied by 5 before being added onto the value of beat%, which is in centi-seconds.

A convention which we've adopted here is to use a pitch number of zero to indicate a rest. If pitch% is zero, line 190 doesn't produce a SOUND command at all but beat% is still increased to take account of the length of the rest.

When beat% reaches (or exceeds, to be on the safe side) 200, the bar is full and PROCbar ends, returning control to the main program to wait for the next reset of the beat counter.

If you think the tune is played too slowly, you can always increase TEMPO. To double the speed for example, type:

```
TEMPO &2000
```

before running the program. Although this shortens the delay between the notes it doesn't make the notes themselves any shorter. Each one is, in fact, shortened by being cut off when the following one is produced, but the rests disappear.

Adding Music to the Game

Now we can consider how to add a procedure to our Munchie game to make it play a tune.

This next listing only shows the parts of the program which have been changed.

```
  10 REM > Munchie5
  20 REM Draws pacman-type figure
  30 ON ERROR VOICES 1:VOICE 1,"WaveSynth-Beep":REPORT:PRINT" at line ";ERL:END
  40 PROCinit
  50 REPEAT
  60   PROCgame
  70 UNTIL (char% AND &DF)<>ASC("Y")
  80 VOICES 1
  90 VOICE 1,"WaveSynth-Beep"
 100 END
 110 :
 120 DEFPROCinit
 130 *SLoad MunSprites
 140 dotnum%=9
 150 width%=68:height%=68
 160 hstep%=10:vstep%=10
 170 DIM dot%(dotnum%,1)
 180 MODE 12
 190 VOICES 2
 200 VOICE 2,"WaveSynth-Beep"
 210 VOICE 1,"StringLib-Hard"
 220 BEATS 200
 230 bar_sent%=FALSE
 240 ENDPROC
 250 :
 260 DEFPROCgame
 270 dotleft%=dotnum%+1
```



```
 280 PROCbackground
 290 GCOL 3,7
 300 FOR num%=0 TO dotnum%
 310   PROCsetdot(num%)
 320 NEXT
 330 GCOL 8,0
 340 xpos%=200:ypos%=200
 350 *SChoose mun_right
 360 PLOT 237,xpos%,ypos%
 370 start=TIME
 380 RESTORE +1
 390 REPEAT
 400   IF BEAT<100 THEN
 410     IF NOT bar_sent% THEN PROCbar:bar_sent%=TRUE
 420     ELSE
 430     bar_sent%=FALSE
 440   ENDIF
 450   PROCmove
 460 UNTIL dotleft%=0
 470 PRINT TAB(10,25)"You took ";(TIME-start)/100" seconds"
 480 PRINT TAB(10,26)"Another go? (y/n)"
 490 REPEAT
 500   char%=GET
 510 UNTIL (char% AND &DF)=ASC("Y") OR (char% AND &DF)=ASC("N")
 520 ENDPROC
 530 :
1310 DEFPROCbar
1320 beat%=0
1330 REPEAT
1340   READ pitch%
1350   IF pitch%=-1 THEN
1360   RESTORE +1:PROCbar:ENDIF:ENDPROC
1370   ELSE
1380   READ dur%
1390   IF pitch%<>0 SOUND 2,-15,pitch%,dur%,beat%
1400   beat%+=dur%*5
1410 UNTIL beat%>=200
1420 ENDIF
1430 DATA 53,10,61,10,69,10,53,10
1440 DATA 53,10,61,10,69,10,53,10
1450 DATA 69,10,73,10,81,20
1460 DATA 69,10,73,10,81,20
1470 DATA 81,5,89,5,81,5,73,5,69,10,53,10
1480 DATA 81,5,89,5,81,5,73,5,69,10,53,10
1490 DATA 53,10,33,10,53,10,0,10
1500 DATA 53,10,33,10,53,10,0,10
1510 DATA -1
1520 ENDPROC
```

We've added some lines to PROCinit. Line 190 sets the number of active channels to two. This is so that we can play the tune on channel 2 while still producing beeps on channel 1 whenever our character swallows up a circle.

Line 200 sets the waveform for the music, while line 210 redefines the one for the beep. It has to compete with the music now so it needs to be louder. Line 220 sets up the beat counter and line 230 creates a variable which we will use to tell us whether or not PROCbar has been called.

In the main program, the error handler on line 30 and the two extra lines 80 and 90 reset the sound channels to their original state, so that a beep reverts to its usual self again when the program has finished, whether by coming to an end or through an error.

Keeping the Loop Going

PROCgame contains the loop which the game runs round while it's being played. We can use this loop to keep calling PROCbar but we mustn't hold it up or the game would freeze. For this reason we can't use our REPEAT ... UNTIL loops to wait for the beat counter to come round to zero so we have to think of something else. This is where variable bar_sent%, which we created in PROCinit, comes in.

If line 400 detects that the beat counter is in the first half of its count, the following line checks to see if PROCbar has already been called that time round the counting cycle. If it hasn't, line 410 calls it, then sets bar_sent% to TRUE, indicating that it has been dealt with. On the next few times round the loop, bar_sent% will be seen to be TRUE, preventing PROCbar from being called again.

Once the beat counter has passed the halfway stage in its counting, bar_sent% can be reset to FALSE again by line 430. This procedure allows plenty of flexibility in the exact time when BEAT is read. It isn't necessary to detect *exactly* when BEAT is zero, which is important because it could happen while the program is busy redrawing a sprite.

PROCbar is identical to the procedure in our Frère Jaques program except that the SOUND command now calls channel 2.

When a game is completed, the tune stops because the loop in PROCgame is no longer being run. When you start a new game, PROCgame is called again and line 380 resets the DATA pointer to the beginning of the data, so the tune starts from the beginning again.

Stereo Sounds

Some models of RISC OS computer have only one internal speaker but others have two and can reproduce stereo sounds. All machines, though, have a stereo audio socket which can feed a Hi-Fi amplifier or stereo headphones so any RISC OS computer can produce a stereo output.

You can position any sound channel wherever you like in the stereo image by using the STEREO command, for example:

```
STEREO 1,-127
```

The STEREO command is followed by two numbers. The first is the channel number and the second is the stereo position, −127 for extreme left, 0 for centre and 127 for extreme right. The example positions channel 1 on the left-hand side of the stereo image.

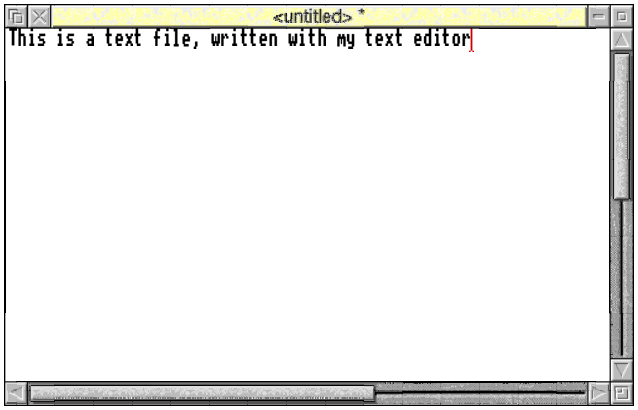
All channels are automatically set to the centre unless you change them.

13

File Handling and Databases

We've only used files for storing Basic programs so far, but we can use files for other purposes besides. A wordprocessor uses files to store its text, Draw and Paint use them to store drawfiles and sprites and Maestro uses them for music. You can save anything in a file and retrieve it again, as long as it consists of numbers which the machine can handle.

Start up your text editor, create a text file and type in:



Don't press Return at the end but save the file in the currently selected directory, using the filename 'MyText', then start up Basic, either using the command line or in a task window

We're going to load this short text file into memory and look at it. To do this, we need to know where we can put it, so type:

```
PRINT ~PAGE
```

As we saw in Section 9, PAGE is the address where Basic stores the start of the program and the tilde (~) character makes sure it is shown as a hexadecimal number.

We also saw that you'll probably get a value for PAGE of &8F00. It's best not to interfere with the first few bytes starting at this address so, to keep clear of it, we'll load our text file 256 bytes further on, at &9000.

Loading in a Text File

As we're not loading a Basic program, we don't use a Basic command but a star command to let the operating system do the job for us. The command is (would you believe it?) *Load, and it usually has to be followed by the address where the first byte of the file is to go, so type:

```
*Load MyText 9000
```

Your text file will now be loaded at this address. Star commands usually assume that addresses are hexadecimal, unless you tell them otherwise.

We can look at the file using the *Memory command. Try typing:

```
*Memory 9000
```

and you should see something like this:

```
Address  :     3 2 1 0     7 6 5 4     B A 9 8     F E D C :    ASCII Data
00009000 :    73696854    20736920    65742061    66207478 : This is a text f
00009010 :    2C656C69    69727720    6E657474    74697720 : ile, written wit
00009020 :    796D2068    78657420    64652074    726F7469 : h my text editor
00009030 :    00000000    00000000    00000000    00000000 : ................
00009040 :    00000000    00000000    00000000    00000000 : ................
00009050 :    00000000    00000000    00000000    00000000 : ................
00009060 :    00000000    00000000    00000000    00000000 : ................
00009070 :    00000000    00000000    00000000    00000000 : ................
00009080 :    00000000    00000000    00000000    00000000 : ................
```

We saw the use of this command in Section 9. Each line shows the contents of 16 bytes, with the address of the first one on the left-hand side, and each group of four bytes is shown with the low byte on the right and the high byte on the left. Being a text file, each byte contains the ASCII code for a character. The section on the right-hand side of the screen shows the bytes turned back into characters which is, hopefully, the text you started with.

Cleaning Up the Memory Block

You will quite likely find that the bytes following your text don't contain zeros as shown, but various other characters. This part of the memory wasn't over-written when you loaded your file. If you want to, you can clean it up *before* you load the file by typing:

```
FOR N%=0 TO 252 STEP 4:N%!&9000=0:NEXT
```

This short program puts zero into four bytes each time round the loop.

The display shows 256 bytes which is what you get if you don't tell the machine otherwise. You can show, say, just the first three lines (48 bytes) by typing:

```
*Memory 9000+30
```

The number after the plus sign is the amount of memory to be displayed, in hexadecimal form. If you want to see more than 256 bytes (&100), it's best to put the screen into page mode by typing Ctrl-N, to stop the display scrolling up the screen too fast to be seen.

If you wanted to save this section of memory as a file, you would use the *Save command. Besides the start address, the operating system also needs to know how big the block of memory is that you want to save so you have to tell it that as well. There are two ways of doing this:

```
*Save MyFile 9000 9100
*Save MyFile 9000+100
```

Both these commands will save 256 (&100) bytes starting at &9000. The first command works by telling the machine the end address of the section to be saved and the second by telling it the length of the section. Note that the second number in the first example is actually the address of the first byte *not* to be saved − the machine saves 256 bytes from &9000 to &90FF.

Saving a Block of Memory

We saw in Section 9 how we could define a block of memory and use *indirection operators* to put single bytes, four-byte words and strings into it. To remind you, we used a program like this:

```
10 REM > Bytes2
20 REM illustrates indirection operators
30 DIM block% 100
```

```
 40 ?block%=&00
 50 block%?1=&11
 60 block%?2=&22
 70 block%?3=&33
 80 block%!4=&87654321
 90 $(block%+8)="Hello There"
100 OSCLI ("Save MyFile "+STR$~block%+"+20")
```

We've changed the filename slightly in the first line and rewritten the last line. Instead of showing us the address of the block, it now saves the block as a file.

If you run this program in Basic's command mode and then type:

```
PRINT ~block%
```

The machine will tell you the address of the start of the block and you'll probably get a figure something like &8FF0. Armed with this information, we can see what's in the block by typing:

```
*Memory 8FF0+20
```

which will show us:

```
Address  :     3 2 1 0     7 6 5 4     B A 9 8     F E D C :    ASCII Data
00008FF0 :    33221100    87654321    6C6C6548    6854206F : .."3!Ce Hello Th
00009000 :    0D657265    00000000    00000000    00000000 : ere.............
```

To save this section, we would have to give the machine the command:

```
*Save MyFile 8FF0+20
```

Unfortunately, we can't put this command into the program as it stands because we won't know the actual address of block% while we're writing the program. It depends on the length of the program itself and the amount of memory that's allocated to other variables before the program gets round to defining block%. It's also possible that a future version of RISC OS could set PAGE to a different value, which would change the starting point of the program in memory.

It is generally not good practice to use addresses directly in this way; it's better to quote the names of the Basic variables which contain them. Unfortunately we can't put block% into a star command. This is because star commands aren't handled by Basic, but by the operating system's *Command Line Interpreter*, which doesn't understand Basic variables.

Using Basic Variables in a Star Command

What we need to do is to get the program to construct a string for us containing the command and pass that on to the CLI. This is what the OSCLI command in line 100 does. The first part of the string is spelt out in quotes, consisting of 'Save MyFile ', with a space on the end before the closing quote. The next part is the address, which is provided by STR$. This keyword works out the value of whatever follows it (block%, in this case) and turns it into a string. The tilde (~) sign means the string is in hexadecimal form. Finally, the last part of the string is again in quotes, giving us '+20'. If the value of block% is &8FF0, the entire string will be:

```
Save MyFile 8FF0+20
```

which is precisely what we want, and OSCLI passes it to the operating system. Notice that an OSCLI command doesn't need a star at the beginning because Basic already knows that it is to be handled by the Command Line Interpreter.

When you run the program it will save the block as a file called 'MyFile' or whatever you chose as a filename. You can load the file back into any suitable address and look at it with the *Memory command to prove that it worked.

Handles and Random Access Files

Everything we've done with files so far has consisted of saving and loading entire files in one go. It is also possible to read or change individual bytes within a file, or add something onto the end of an existing file. Using this technique, you could work on a file which is actually too long to load into your machine's memory in one go.

To use a file in this way we first have to *open* it and when we've finished we have to *close* it again. There are three commands for opening files:

- OPENIN opens a file for read access only. The file can be read but not altered.
- OPENUP opens a file for read or write access.
- OPENOUT creates a new file, initially zero bytes long. If a file of the same name already exists, it's deleted and a new one created. The file has read or write access but you obviously can't read anything from it until you've written something to it.

If you still have your 'MyText' file, try opening it with this command:

```
h%=OPENUP("MyText")
```

The filename is in brackets and can be the actual filename (or complete pathname) in quotes or a string variable containing it.

The variable h% is the *file handle*. This is a number given to the file by the operating system when it's opened. From now on we refer to the file not by its filename, but by its file handle.

There are several keywords connected with the file which use the file handle. The first of these is EXT# which tells us the length of the file in bytes:

```
PRINT EXT#h%
```

The text file shown earlier contains 48 characters, so this command will produce 48.

The keyword PTR# refers to the *file pointer* which selects which byte in the file we're going to read or write to. We can read a byte with BGET# and write one or more with BPUT#.

Try this:

```
PTR#h%=10
a%=BGET#h%
PRINT CHR$(a%)
```

This sets the file pointer to 10 bytes in from the start of the file. In this file, this byte contains the first letter of the word 'text' so the machine will print a 't'. After reading the byte, PTR#h% will automatically be increased by 1, ready to read the next byte, so if you repeat the last two lines, the result will be the next letter, an 'e'.

Writing to the File

As we used OPENUP to open the file, we can change a byte in it, using the BPUT# command which will write either a byte or a string to the file.

Let's replace the 't' of 'text' with a capital 'T'. We could put:

```
PTR#h%=10
BPUT#h%,84
```

quoting 84, which is the ASCII code for 'T', or we could quote a one-character string, like this:

```
PTR#h%=10
BPUT#h%,"T";
```

The semi-colon (;) on the end of the line is important. When the BPUT command writes a string to a file, it follows it with a Line Feed (ASCII code 10) to mark the end

of the string, unless you tell it not to by including a semi-colon, rather like the PRINT command. If you left out the semi-colon, the next character, 'e', would be over-written by the Line Feed.

You can tell whether or not the pointer has reached the end of the file by the state of the variable EOF#, which stands for **E**nd **O**f **F**ile. If the pointer is somewhere within the file, EOF#h% is FALSE. After reading or writing the last byte, the pointer will be set to one byte *after* the end of the file and EOF#h% becomes TRUE.

We can use this to display the whole file:

```
PTR#h%=0:REPEAT:a%=BGET#h%:PRINT CHR$a%;:UNTIL EOF#h%:VDU 10
```

We start by setting the pointer back to the beginning of the file, then read and display each character in turn. The semi-colon after PRINT CHR$a% prevents the PRINT statement from sending a Line Feed to the VDU, otherwise each letter would be on a new line. After the last one has been read, PTR#h% is beyond the end of the file, EOF#h% becomes TRUE and the loop stops repeating. Finally, VDU 10 send a Line Feed to the VDU so that the Basic prompt is on a new line.

Finally, when you've finished with a file, it's important to close it with the CLOSE# command:

```
CLOSE#h%
```

This is particularly true if you've written to the file. The file is not changed on the disc every time you write a byte to it − that might well result in too much disc drive activity and worn-out discs. The filing system stores a section of the file in memory and makes the changes there. These changes are only transferred to the disc when it has to move onto a new section of the file, or when the file is closed.

This means that if you alter a file, then reset your machine or switch it off without closing the file first, the file on the disc may not be up-to-date.

Creating a Database Program

We're going to use what we've learned so far in a simple database manager program. It doesn't have many of the features of the sophisticated packages which you can pay a lot of money for, but you can use it for simple storage of information, it will show you how file handling works and it will cost you nothing!

Despite the simple nature of the database manager, it's still the longest program in this guide. We've left out a lot of features, otherwise it would have been even longer as well as being more difficult to understand! We won't build it up a bit at a time as we did

with the Munchie program because a lot of it needs to be present for it to do anything. Instead, we'll go through its workings a bit at a time, assisted by its structured nature − the fact that it's split up into functions and procedures.

This program has been enhanced slightly since it was published in the first edition of this guide. There are two versions in the files directory, for mode 12 and mode 27, called Database12 and Database27 respectively. Everything in the mode 27 version is displayed 10 text lines lower on the screen.

Before we start, we need to understand a few things about database programs. The usual starting point is to compare a database with a card index file. Each card contains several pieces of information about somebody or something. In the database, the pieces that would be on one card are grouped together and called a *record*. Each piece of information within a record, for example a name or phone number, is called a *field*. The whole thing makes up a *database file*.

To simplify this program, all the fields in a record have to be the same length. This means that they all contain the same number of characters, even if a lot of them are spaces.

The program has been specially written so that you can 'customise' it to suit your own requirements. The field length and the number of fields in each record are set up in variables just after the start of the program, clearly marked with REM statements. Each field can be displayed on the screen with a *label* to tell you what it is, such as 'Name :' or 'Address :'. These are set up in a procedure called PROCset_labels which has been deliberately placed right at the end of the program.

Not all the fields will need labels; in the program as shown here, a record consists of a name, four lines for the address and a phone number. Each line of the address is a separate field; obviously only the first one needs a label.

Using the Program

When you run the program, you're first presented with three options − create a new database file, load an existing file or quit the program completely. Every time you finish with a file you come back to this point.

If you choose one of the first two options, the program either creates a new file, zero bytes long, or loads an existing one. If you choose the first option, the program checks first to see if a file with the name you typed already exists. This is very important because without it, if you chose this option by mistake, you would delete your existing file and lose all your data!

After creating or loading your file, the program checks to see if it's zero bytes in length. If it is, you'll be put straight into the editor screen and you won't be allowed out until you've saved a record in the file. This situation happens if you create a new file or if the one you load is zero bytes long.

In the edit screen, the fields are represented by stars to indicate their length. You can move the cursor around the fields by using the arrow keys and type information into them. If you want to start again, pressing Ctrl-B clears all the fields. When you've finished, Ctrl-A saves the record to the file and puts you into the normal display mode or Ctrl-C abandons the edit session.

In the normal display mode, you can browse through the records by pressing 1 to go backwards and 2 to go forwards. Pressing 3 will let you edit the record you are on, 4 lets you create a new record and 5 closes the database file when you've finished with it. This returns you to the opening screen, ready to open another file or quit the program.

Errors With Open Files

Let's look first at the initial part of the main program, before it gets to its main loop:

```
 10 REM > Database
 20 REM Simple database manager
 30 ON ERROR REPORT:PRINT" at line ";ERL:END
 40 REM *** Set up database parameters here ***
 50 field_no%=6:REM number of fields
 60 field_len%=30:REM length of fields
 70 REM *** field labels set in PROCset_labels at end of program
 80 REM *** End of parameter setup ***
 90 PROCinit
100 ON ERROR PROCerror:END
```

You can see that, in addition to our usual error handler in line 30, there is another in line 100 which calls a procedure, PROCerror:

```
510 DEFPROCerror
520 REM error handling after initialisation
530 IF file%<>0 CLOSE#file%:REM close database file if open
540 REPORT:PRINT" at line ";ERL
550 ENDPROC
```

If an error occurs, or we press Esc, with a file open, we need to close it before leaving the program. If we don't do this, any changes we've made to the database may not be saved and if we try to do anything with this file, such as opening it again, we will get a 'File open' error message.

We saw earlier in this section how when we open a file we get a number called the file handle which we use to refer to it. In this program, the file handle is the value of variable file%. If we do not have an open file, we set file% to zero. In this way, we can use file% to tell us whether or not a file is open. If our error handler is called, which means that we're going to have to shut down the program, we need to know if we have an open file and close it if we do. PROCerror checks file% for this purpose.

To be safe, we need to create variable file% before there is any chance of PROCerror being called. If we didn't do this and an error occurred before we'd opened a file, line 530 in PROCerror would try to check the value of file% and find it didn't exist. This would result in another call to the error handler, with an 'Unknown or missing variable' error message. When the error handler reached line 530, the same thing would happen again and the program would go round and round in an infinite loop, freezing the machine. We create file%, with its initial value of zero, in PROCinit. Just in case there is an error before this happens, we start off with the error handler in line 30, then replace it with the one in line 100 *after* we've created file% but *before* we've opened any files.

Customising the Program

Lines 50 and 60 are concerned with altering the program to suit your own requirements. You can vary the number of fields by changing field_no%, or their length by changing field_len%. These lines are put close to the beginning of the program to make them easier to find and are marked by REM statements containing a number of asterisks.

The labels which identify each field are stored in the elements of an array variable called label$(). You can't set up these elements until the array has been DIMmed, which happens in PROCinit, so this job is done by a procedure, PROCset_labels, located at the very end of the program:

```
  410 DEFPROCinit
  420 REM program initialisation
  430 DIM label$(field_no%),field$(field_no%)
  440 PROCset_labels
  450 MODE 27
  460 quit%=FALSE
  470 file%=0
  480 rec_len%=field_no%*field_len%
  490 ENDPROC
  500 :

 2410 DEFPROCset_labels
 2420 REM *** Define field label names here ***
 2430 label$(1)="Name:"
 2440 label$(2)="Address:"
```



```
 2450 label$(6)="Phone:"
 2460 ENDPROC
 2470 :
```

As well as setting up the label array, PROCinit also DIMs an array called field$(). This array is used as a temporary store for the contents of each field in the record which is being displayed or edited. The procedure also sets up variable quit%, which tells the program when it's time to close down, and rec_len%. This is the total length of each record in the file, which is the field length multiplied by the number of fields.

After setting up these parameters, which only has to be done once, the program enters its main loop:

```
  110 REM *** Main program loop ***
  120 REPEAT
  130   PROCoptions
  140   CASE char% OF
  150     WHEN ASC("1"):PROCcreate
  160     WHEN ASC("2"):PROCload
  170     WHEN ASC("3"):quit%=TRUE
  180   ENDCASE
  190   IF NOT quit% THEN
  200     record%=0
  210     REM *** Secondary program loop ***
  220     REPEAT
  230       IF EXT#file%=0 THEN
  240         PROCedit(0)
  250         ELSE
  260         PROCdisplay
  270         char%=FNmove_opts
  280         CASE char% OF
  290           WHEN 1:IF record%>0 record%-=1 ELSE VDU 7
  300           WHEN 2:IF (record%+1)*rec_len%<EXT#file% record%+=1 ELSE VDU 7
  310           WHEN 3:PROCedit(1)
  320           WHEN 4:record%=EXT#file% DIV rec_len%:PROCedit(0)
  330           WHEN 5:CLOSE#file%:file%=0
  340         ENDCASE
  350       ENDIF
  360     UNTIL char%=5:REM *** end of secondary program loop ***
  370   ENDIF
  380 UNTIL quit%:REM *** end of main program loop ***
  390 END
  400 :
```

We'll take a look at how the loop as a whole works first, before examining its procedures.

The loop starts with a call to PROCoptions. This procedure prints an instruction telling you which keys to press and waits for you to press 1, 2 or 3 before returning to the main loop. If you press 3, the program simply sets quit% to TRUE, skips the rest of the main loop and exits. If you press 1 or 2, it calls PROCcreate or PROCload as appropriate. PROCcreate creates a new file zero bytes long using OPENOUT. PROCload loads an existing file using OPENUP.

Whichever option you choose, variable record%, which holds the number of the record you're working on, is set to zero and the program enters its secondary loop, inside the main loop.

The first action of the secondary loop is to check the length of the file. If it's zero bytes (which probably means you've just created it with PROCcreate), you've no option but to add the first record to it so the program puts you into edit mode by calling PROCedit.

By the time you come out of PROCedit, you will have added a record to the file as it won't let you leave until the file has something in it. The next time round the secondary loop, the program calls PROCdisplay, which shows you the contents of the record and waits for you to press a suitable key. If you press 1 or 2, the program checks to make sure you're not on the first or last record of the file, increases or decreases record% by 1 and returns to the start of the secondary loop, ready to display a new record, selected by the new value of record%.

If you press 3, the program calls PROCedit to alter the record that you're on. When you press 4, the program creates a new record. It does this by calculating the number of records already in the file by dividing the file length (using EXT#) by the record length, rec_len%, and setting record% to this number. Because records are numbered from zero, this new value of record% represents a new one and PROCedit is called to produce it.

When you press 5, the file is closed. The secondary loop repeats until you've done this, then puts you back to the start of the main loop, giving you the choice of loading or creating a new file or of quitting the program.

Getting Into Finer Detail

We'll look in detail first at PROCedit since you start in edit mode. You may have noticed that when the secondary loop calls this procedure, it passes it a number, 0 or 1, as a parameter. This is to tell PROCedit whether it's creating a new record or editing an existing one.

```
1070 DEFPROCedit(x%)
1080 REM edit record
1090 REM x% = 0 if creating new record, x% = 1 if editing existing record
```



```
1100 LOCAL n%,char%
1110 PROClabels
1120 PROCstars
1130 PRINT TAB(21,30)"Key Ctrl-A to store record"
1140 PRINT TAB(21,31)"Key Ctrl-B to clear record"
1150 PRINT TAB(21,32)"Key Ctrl-C to leave edit mode"
1160 CASE x% OF
1170   WHEN 0:PROCedit_clear
1180   WHEN 1:
1190   FOR n%=1 TO field_no%:PRINT TAB(25,n%+19)FNtrunc(n%):NEXT:REM display f
ields
1200   VDU 31,25,20:REM move cursor to start of first field
1210 ENDCASE
1220 *FX4,1
1230 REPEAT
1240   char%=GET:REM get keypress
1250   CASE char% OF
1260     WHEN 8,136:PROCedit_left
1270     WHEN 137:PROCedit_right
1280     WHEN 13,138:PROCedit_down
1290     WHEN 139:PROCedit_up
1300     WHEN 1:VDU 7:PROCstore
1310     WHEN 2:PROCedit_clear
1320     WHEN 3:IF record%*rec_len%>=EXT#file% AND record%>0 record%-=1
1330     OTHERWISE:PROCupdate(char%)
1340   ENDCASE
1350 UNTIL (char%=1 OR char%=3) AND EXT#file%>0
1360 *FX4
1370 ENDPROC
1380 :
```

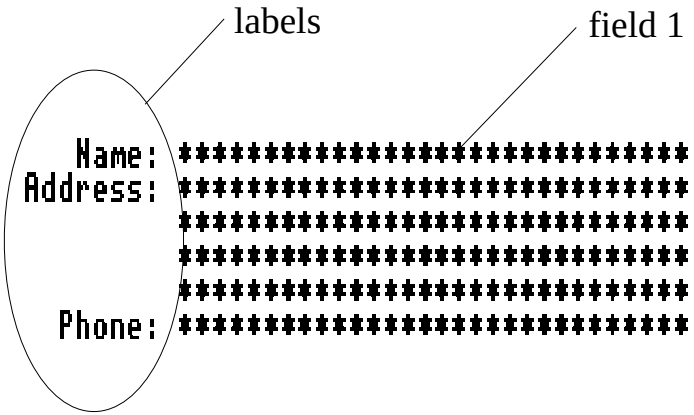
The first action of the procedure is to call PROClabels:

```
1990 DEFPROClabels
2000 REM Print record number and names of labels
2010 LOCAL n%
2020 CLS
2030 PRINT"Record number ";record%+1
2040 FOR n%=1 TO field_no%
2050   PRINT TAB(24-LEN(label$(n%)),n%+19)label$(n%):REM print label, right-ju
stified
2060 NEXT
2070 ENDPROC
2080 :
```

This procedure prints the record number in the top left-hand corner of the screen, adding 1 to it so that the records appear to be numbered from 1, not 0. It then prints the

name of each label in turn. The rather complicated-looking TAB statement has the effect of *right-justifying* the label. This means that its horizontal position is adjusted so that the right-hand character is 23 positions in from the left-hand side of the screen, however long the label is. This is purely to improve the appearance of the screen.

When the array which holds the labels, label$(), is created in PROCinit, each element in it is set to the null string, that is zero characters in length. If you don't define a particular label in PROCset_labels, it stays like this and PROClabels doesn't print anything for that line.




```
Key Ctrl-A to store record
Key Ctrl-B to clear record
Key Ctrl-C to leave edit mode
```

The standard layout used by this program has each field appearing on the screen 25 characters in from the left, and the first field, number 1, 20 lines down from the top (10 in mode 12). This means that any particular field is on row 19 (or 9) plus the field number. This is important, because PROCupdate, which we'll get to later, uses the position of the cursor to work out which character in a field to alter.

Seeing Stars

After printing the labels, PROCedit calls PROCstars which prints a rectangle of stars, one for each character of each field. This is to show you the maximum number of characters you can type into each field. This is important because the fields all have to be exactly the correct length so the program knows where to find them in the file. It doesn't matter how short the text in a field is as the rest of the field is filled with spaces.

Returning to PROCedit, after printing a message on the bottom of the screen telling you which keys to press, we look at the number which was passed as a parameter when PROCedit was called, which is now local variable x%. If this is zero, indicating that we're editing a newly-created record, we clear all the fields. PROCedit_clear does this by setting each element of the array field$() to a string of spaces, one for each character in the field.

If x% is 1, PROCedit is being used to edit an existing record so we want to display the data in its fields. We don't want to dispense with the stars completely, though, because they show us how many characters we could add to a field, so we print each field over the top of the stars, leaving the ones at the end of each line untouched. This is done by using FNtrunc which removes the excess spaces from the end of each field string.

Suppose, for example, you had a name consisting of 25 characters. Unless you've modified the program, each field has a length of 30 characters, which means that PROCstars has put 30 stars on each line. It also means that field$(1) is 30 characters long. The first 25 contain the name and the last five are spaces. FNtrunc removes these spaces and produces a string 25 characters long containing just the name. When you print this string, the first 25 stars are over-written and the remaining five left untouched.

It may seem unnecessary to use a CASE ... OF structure to handle the various values of x% as there are only two options and we could easily use IF ... THEN instead. Doing it this way makes it easier for you to add other options to use PROCedit in different ways.

```
     Name: Arnold Alexander Thompson*****
  Address: 25 Archimedes Terrace*********
           Floppy Drive*******************
           Databridge*********************
           Middleshire XX1 2XX************
    Phone: 01111 999999********************


      Key Ctrl-A to store record
      Key Ctrl-B to clear record
      Key Ctrl-C to leave edit mode
```

... the first 25 stars are over-written ...

Having displayed everything on the screen, the next thing to do is wait for a keypress. Before we do that, though, we need to modify the actions of the arrow keys so that we can use them to move the cursor around the fields.

Normally, these keys give us cursor editing. If we want to change this, we use the command *FX4:

*FX4 or *FX4,0 − cursor editing

*FX4,1 − keys produce ASCII codes:

Copy: 135
Left: 136
Right: 137
Down: 138
Up: 139

*FX4,2 − arrow keys behave like function keys:

Copy: Key 11
Left: Key 12
Right: Key 13
Down: Key 14
Up: Key 15

In this case, we use *FX4,1 and check for codes 136 to 139. When PROCedit ends, we use *FX4 to put the arrow keys back to normal.

We use a CASE ... OF structure between lines 1250 and 1340 to examine the code produced by the GET keyword. The four numbers produced by the arrow keys call procedures for moving the cursor, using VDU calls. These calls are all listed in Appendix 2. As well as using the arrow keys, the cursor can be moved to the left by pressing Backspace (code 8) and down by pressing Return (code 13).

The ASCII codes 1, 2 and 3 are produced by Ctrl-A, Ctrl-B and Ctrl-C respectively (you can, of course, change these keys, if you wish). Ctrl-A beeps and stores the edited record in the file, Ctrl-B clears all fields of the record, replacing them with stars, so that you can start again and Ctrl-C simply allows you to abandon the edit. You can only do this if there is something in the file. If PROCedit had been called to add a new record onto the end of the file, the file pointer, PTR#file% would be past the end of the file and abandoning the edit in this way would cause problems. This is avoided by reducing record% by 1 so that it contains the number of the last record already in the file.

Updating Fields

If none of these options is chosen, it means we've typed a character for insertion into one of the fields. This is handled by PROCupdate:


```
1590 DEFPROCupdate(char%)
1600 REM update character in field
1610 IF char%>31 AND char% <127 THEN
1620   MID$(field$(VPOS-19),POS-24,1)=CHR$(char%)
1630   VDU char%
1640   IF POS>field_len%+25 VDU 8
1650 ENDIF
1660 ENDPROC
1670 :
```

This procedure uses the position of the text cursor to work out which character has to be replaced in which field. It does this by using the keywords POS and VPOS. POS is the horizontal position of the cursor on the line and VPOS the vertical position (these are the numbers you put in a TAB command to move the cursor to that position).

We saw earlier that each field is displayed 25 spaces in from the left and also that field 1 is 20 rows down from the top (10 in mode 12). For this reason, we can work out which character of which field the cursor is on by subtracting 24 from POS to get the character, and 19 (or 9) from VPOS to get the field. The procedure does all this and changes the appropriate character in the field string, using the MID$ keyword.

Each time we update a character, the action of the VDU moves the cursor one space to the right, ready to update the next one. If we type in the last permitted character in a field, the cursor would move to a space beyond the end of the field. If this happens, line 1640 sends a code 8 to the screen which moves it back one space again.

Back in PROCedit, at line 1300, when Ctrl-A is chosen, the file is updated by PROCstore:

```
1680 DEFPROCstore
1690 REM store record in file
1700 LOCAL n%
1710 PTR#file%=record%*rec_len%
1720 FOR n%=1 TO field_no%
1730   BPUT#file%,field$(n%);
1740 NEXT
1750 ENDPROC
1760 :
```

The file pointer is first moved to the start of the record by line 1710. This is a position in the file which is worked out by multiplying the record number by the record length. Each field string is then written to the file, using the BPUT# command. Notice that the command is followed by a semi-colon (;) to prevent the string from being followed by a Line Feed character.

When Ctrl-A or Ctrl-C is pressed, PROCedit is terminated, provided there is something in the file (this has to be the case if you pressed Ctrl-A).

Displaying Records

Apart from PROCedit, the most interesting procedure in the program is probably PROCdisplay, which gets all the data for one record from the file and puts it on the screen.

```
1870 DEFPROCdisplay
1880 REM display contents of record and wait for keypress
1890 LOCAL n%
1900 PROClabels
1910 PROCdisplay_bottom
1920 PTR#file%=record%*rec_len%
1930 FOR n%=1 TO field_no%
1940   field$(n%)="":FOR I%=1 TO field_len%:field$(n%)+=CHR$(BGET#file%):NEXT
1950   PRINT TAB(25,n%+19)field$(n%)
1960 NEXT
1970 ENDPROC
1980 :
```

Like PROCedit, the first action of this procedure is to display the field labels by calling PROClabels. The instructions at the bottom of the screen telling you which key to press are contained in a separate procedure called PROCdisplay_bottom.

Getting the information from the file consists of setting the file pointer to the start of the record and reading the appropriate number of bytes. Because the records are numbered from zero, each one starts at a position worked out by multiplying the length of one record by the record number. Record 2, for example, has two records before it in the file and starts at 2 times rec_len%.

After setting the pointer, the procedure goes through each element of field$() in turn, setting it to the null string (zero bytes in length), then adding the appropriate number of characters from the file and finally displaying it in the correct place on the screen.

Further Steps

You'll probably agree that what you have is a very simple database program. Nevertheless, it is over 200 lines long and may have taken a bit of assimilating. When you understand how it works, you may like to add to it.

How about, for instance, a search facility? You could type in a string and tell the program which field or fields to check. Try using the INSTR keyword, described in Appendix 1.

If you're feeling more adventurous, you could try a sorting procedure. Tell the program which field to sort on and rearrange the records in order.

To make your database program really useful, though, it would be very helpful if you could vary the size and number of fields, and in particular the field labels of individual database files so that your program could be used to store data of all kinds. You could include this information in a 'header' to go at the beginning of the file. You would, of course, have to take the length of the header into account when setting PTR# to the start of a record.

14

SWIs and Assembly Language

This section is intended to whet your appetite only. Entire books can be written about both these subjects and they can get very complicated.

The ARM or StrongARM processor in your machine doesn't actually know anything about Basic or even what to do with star commands. It works with very simple instructions called machine code. Basically, these instructions tell it to move numbers to and from the memory and shuffle them around within itself. RISC OS is written in machine code and when you're running a Basic program, you're actually running a part of RISC OS called the *Basic interpreter*. This is a large machine code program which looks at each line of your Basic program and works out what to do with it.

'Why learn machine code?', you may be wondering. For most things, you won't need to, as the version of Basic in your machine is very flexible. It's possible, though, that you'll come up with a requirement that Basic can't handle fast enough, especially when you're plotting graphics on the screen. A machine code program could probably do this faster for you. When you get more advanced, you may wish to write your own relocatable module for some purpose or other. These *have* to be written in machine code.

Software Interrupts

We'll concern ourselves with machine code in general later. Right now, we'll look at one particular instruction called a *software interrupt* or *SWI*. RISC OS uses this as a way of calling the many hundreds of routines contained within its ROM chip.

It's worth learning about SWIs because they allow you to use parts of the operating system which do things that Basic doesn't have commands for.

The SWI instruction contains a number which can be up to 24 bits long, though only the bottom 20 bits, that is five hexadecimal digits, are used. The number determines which RISC OS routine is being called. Each SWI number also has a name which describes what it does.

An SWI is rather like a procedure in Basic. When the processor encounters one, it jumps to a particular address in RISC OS. The software there examines the number in the instruction, then executes one of the operating system routines. Afterwards, the processor returns to the instruction which followed the SWI.

The ARM processor contains a number of *registers* which are referred to as R0, R1, R2 and so on. Each of these can hold one 32-bit number. They have a great many uses, including passing parameters to and from the SWIs.

The SYS Command

We can use these software interrupts quite easily through the Basic SYS command. A typical command might look like this:

```
SYS "OS_WriteC",65
```

In this command, the keyword is followed by the name of the SWI in quotes. The number after the comma is put into register R0 before the SWI is called. There might be more numbers, separated by commas, in which case they are put into registers R1, R2 and so on.

If you type in this command, you must take care to get the name exactly right or you'll get an error message saying 'SWI name not known'. It's important to make the correct use of capital and lower case letters.

Each SWI name is in two parts, separated by an underline (_) symbol. The first part refers to the section of software which handles the SWI. This might be the central part or kernel which uses the prefix OS, as in this case, or it could be a relocatable module, either in the operating system ROM or loaded from disc. The second part of the name describes what the SWI does. In both cases, words are strung together without any spaces between them and each word begins with a capital letter.

When you press Return after typing in this command, the letter 'A' will appear on the screen − its ASCII code is, of course, 65. This command is the equivalent of the Basic VDU command; indeed, VDU uses this SWI. 'OS_WriteC' is short for **'O**perating **S**ystem **Write C**haracter' and its exact function is to send the character in register R0 to the screen.

Now let's try another one. Type:

```
SYS "OS_ReadC" TO a%
```


Notice this time the word TO before a%. Any numbers or values of variables after the SWI name but before TO are put into registers R0, R1, R2 and so on *before* the SWI call and any variables listed after TO are set to the numbers in the registers *after* the call. This means that a% will be set to whatever is in R0 after the SWI call.

When you type in this command and press Return, the Basic prompt will not reappear until you press another key. If you now type:

```
PRINT a%
```

you'll get the ASCII code for the character you typed. 'OS_ReadC' is short for **'O**perating **S**ystem **Read C**haracter' and there are no prizes for guessing that this SWI is the equivalent of the Basic keyword GET. It doesn't matter what is in the processor's registers when you call this SWI, but it returns with the code for the keypress in R0.

Status Flags

A SYS command might have the form:

```
SYS "Xxx_XxxXxx",a%,b%,c% TO x%,y%,z%;f%
```

In this case, 'Xxx_XxxXxx' is the name of the SWI and the values of a%, b% and c% are loaded into registers R0, R1 and R2 respectively. The SWI is then called and the contents of R0, R1 and R2 are put into x%, y% and z% respectively.

The processor also has a number of *flags*. These are single bits which indicate something about the last number that was processed. The ones of most interest to us at the moment are the **N**egative, **Z**ero, **C**arry and o**V**erflow flags, known as N, Z, C and V respectively. These are put together to form a binary number. If the SYS command has a variable following a semi-colon (;), this variable is set to the number formed from the flags:

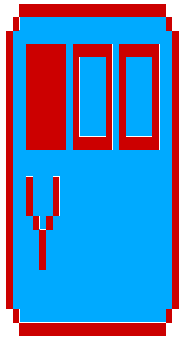
Bit	Flag
0	Overflow
1	Carry
2	Zero
3	Negative

If you are writing a program which asks a question requiring a yes or no answer, you can use this SWI call:

```
SYS "OS_Confirm" TO key%;flags%
```

There are no parameters passed to the SWI so there are no numbers before the TO keyword.

This call waits for you to press a key. If the mouse pointer is turned on (with MOUSE ON), it changes shape to look like a mouse with the left-hand button highlighted to indicate that you should press this button to answer ‘yes’ and either of the other two to answer ‘no’.



... it changes shape to look like a mouse ...

When you press a key or click the mouse, control is returned from the SWI to the program. The ASCII code for the key which you pressed, converted to lower case, is in R0 and is passed to key%. If you pressed ‘Y’ or the left-hand mouse button, the Zero flag is set which means that flags% will be 4. If you pressed Esc, the Carry flag will be set, making flags% 2. If you pressed another key or clicked one of the other mouse buttons, flags% will be zero.

Improving the Munchie Program

As we saw at the end of Section 11, the Munchie program would work better if it stored its sprites in a user sprite area within its own memory. This will not make any difference to how it looks and sounds but you will no longer have to allocate memory to the System sprite area as we won’t be using it.

Handling sprites in a user sprite area requires the use of SWIs, which is why we’ve left it until now. All sprite operations are dealt with by one SWI called OS_SpriteOp which can carry out over 40 different operations, selected by the number in R0. Complete details of this SWI are outside the scope of this guide; if you wish to use sprites extensively, you should consult the *RISC OS Programmer’s Reference Manual*, published on CD-ROM by RISCOS Ltd., which has a complete list of SWIs and full details of how they work.

The listings in this section show modifications to the version of the program from Section 12, which has had sound added. Some of the line numbering is different as we change the number of lines in some procedures.

The first alteration is to the beginning of PROCinit:

```
120 DEFPROCinit
130 DIM sblock% 165620
140 !sblock%=165620:sblock%!8=16
150 SYS "OS_SpriteOp",9+256,sblock%
160 SYS "OS_SpriteOp",10+256,sblock%,"MunSprites"
170 dotnum%=9
```

In each of these listings, we’ll include one or two lines before and after the ones which we modify to make it clear which part of the procedure we’re looking at.

The first thing we need is a block of memory to hold our sprites. It needs to be at least as large as the MunSprites file, so we define a suitable block using the DIM keyword in line 130. As we saw in Section 9, this allocates a block of memory and sets the value of sblock% to the address of the first byte. The block needs to be just large enough to hold the sprites in the MunSprites file, plus a few extra bytes. In the mode 27 version, this is 165,620 bytes, but in the mode 12 version, we restrict it to 90,000 bytes in case you’re short of memory.

Having defined our block, we need to initialise it. Before we do this, however, we must set the first word to contain the size of the block and the word beginning eight bytes in to contain the position of the start of the first sprite (16 in the case of an empty block).

All calls to OS_SpriteOp have a number called a reason code in R0 to indicate which operation we wish to carry out; the first two we shall use are reason code 9 in line 150 to initialise the block, and code 10 in line 160 to load the sprite file. The number in R0, however, indicates more than this. In order to tell the SWI that we’re using a user sprite area, rather than the system sprite area, we have to set bit 8 of R0 to 1, which corresponds to adding 256 to the number. This will apply to all the calls to OS_SpriteOp in this program. To make it plain that we’re calling operation number 9, we will write this as ‘9+256’ rather than ‘265’.

OS_SpriteOp 9 sets up our user sprite area so that we can load a sprite file into it and the following line loads the file. In both cases, R1 contains the address of the sprite area, sblock%. In the case of OS_SpriteOp 10, R2 contains the filename or complete pathname. It’s not possible, of course, to get a filename into a 32-bit register, but Basic’s SYS command comes to our rescue here. What the SWI actually wants is the address of a string containing the filename and this is what the SYS command puts into R2. It can do this either with a string in quotes, as in this example, or a string variable.

The next modification is to PROCgame:

```
  330 FOR num%=0 TO dotnum%
  340   PROCsetdot(num%)
  350 NEXT
  360 xpos%=200:ypos%=200
  370 MOVE xpos%,ypos%
  380 SYS "OS_SpriteOp",256+28,sblock%,"mun_right",,,8
  390 start=TIME
```

This shows how we plot a sprite using OS_SpriteOp 28. The operation is a little different from the way we did it using star commands. Previously, we selected the sprite first, then plotted it with a PLOT instruction, quoting the coordinates of the bottom left-hand corner. In this case, we first use a MOVE command to put the graphics cursor where we want this corner to be, then call OS_SpriteOp. Once again, the address of the sprite area is in R1 and this time R2 contains the name of the sprite. The number 8 in R5 is the plot action, meaning that the sprite is plotted with a mask, if it has one.

The remaining modifications all make the same use of OS_SpriteOp 28:

```
  760 DEFPROCleft
  770 PROCrubout(xpos%,ypos%)
  780 xpos%-=hstep%
  790 PROCcollision
  800 MOVE xpos%,ypos%
  810 IFxpos% MOD (hstep%*2)=0 THEN SYS "OS_SpriteOp",28+256,sblock%,"mun_left"
,,,8 ELSE SYS "OS_SpriteOp",28+256,sblock%,"mun_left2",,,8
  820 ENDPROC
  830 :
```

PROCright, PROCup and PROCdown are modified in the same manner as PROCleft, with the appropriate sprite names and xpos% and hstep% replaced with ypos% and vstep% in PROCup and PROCdown.

The only other procedure to modify is PROCbackground:

```
 1240 DEFPROCbackground
 1250 MOVE 0,0
 1260 SYS "OS_SpriteOp",28+256,sblock%,"mun_bg"
 1270 ENDPROC
 1280 :
```

You will notice that the three commas and number 8 are missing from the end of line 1260. This is because there is no point in plotting the mun_bg sprite with a mask as it hasn't got one − it fills the whole screen. The number for simply plotting the sprite is zero. If we don't tell the SYS command which number to put in a particular register, it automatically gives it a zero, so there is no point in telling it to put zero in R5.

Assembly Language and Machine Code

As we saw at the beginning of this section, the ARM or StrongARM processor in your machine only understands a simple set of instructions called machine code. To understand what this is all about, let's look at some.

Just suppose you had loaded a file into memory, beginning at address &9000. To see the first few bytes, you would type:

```
*Memory 9000
```

You would see something which starts like this:

```
Address  :     3 2 1 0     7 6 5 4     B A 9 8     F E D C :    ASCII Data
00009000 :    E28F00F0    E28C2000    E04D3002    EF020049 : ð.•â. …â.0MàI..ï
00009010 :    61A0F00E    E4926004    E4D67001    E4D63001 : .ð a.`‹ä.pÖä.0Öä
00009020 :    E0877403    E4924004    E4D45001    E4D43001 : .t à.@‹ä.PÔä.0Ôä
00009030 :    E0855403    E3550004    3A000024    E5D43000 : .T à..Uã$..:.0Ôå
00009040 :    E333006E    1333004E    1A000020    E5D43001 : n.3ãN.3. ....0Ôå
```

and so on.

You'll no doubt agree that this is pretty well incomprehensible. You can certainly see the contents of each four-byte word but the numbers don't convey anything to us; neither do the equivalent characters shown on the right-hand side of the display.

Each word, however, contains an individual instruction telling the processor to do something. This is machine code but it's obvious that we need a much simpler way of reading, writing and understanding it.

Disassembling the Code

Now suppose you looked at the memory again, but this time typed:

```
*MemoryI 9000
```

Notice the letter I on the end of the command. This time, you would see:

```
00009000 : ð.•â : E28F00F0 : ADR    R0,&000090F8
00009004 : . …â : E28C2000 : ADD    R2,R12,#0
00009008 : .0Mà : E04D3002 : SUB    R3,R13,R2
```

```
0000900C : I..ï : EF020049 : SWI    XOS_ReadArgs
00009010 : .ð a : 61A0F00E : MOVVS  PC,R14
00009014 : .`‹ä : E4926004 : LDR    R6,[R2],#4
00009018 : .pÖä : E4D67001 : LDRB   R7,[R6],#1
0000901C : .0Öä : E4D63001 : LDRB   R3,[R6],#1
00009020 : .t à : E0877403 : ADD    R7,R7,R3,LSL #8
00009024 : .@‹ä : E4924004 : LDR    R4,[R2],#4
00009028 : .PÔä : E4D45001 : LDRB   R5,[R4],#1
0000902C : .0Ôä : E4D43001 : LDRB   R3,[R4],#1
00009030 : .T à : E0855403 : ADD    R5,R5,R3,LSL #8
00009034 : ..Uã : E3550004 : CMP    R5,#4
00009038 : $..: : 3A000024 : BCC    &000090D0
0000903C : .0Ôå : E5D43000 : LDRB   R3,[R4,#0]
00009040 : n.3ã : E333006E : TEQ    R3,#&6E          ; ="n"
00009044 : N.3. : 1333004E : TEQNE  R3,#&4E          ; ="N"
00009048 :  ... : 1A000020 : BNE    &000090D0
0000904C : .0Ôå : E5D43001 : LDRB   R3,[R4,#1]
00009050 : e.3ã : E3330065 : TEQ    R3,#&65          ; ="e"
00009054 : E.3. : 13330045 : TEQNE  R3,#&45          ; ="E"
00009058 : .... : 1A00001C : BNE    &000090D0
0000905C : .0Ôå : E5D43002 : LDRB   R3,[R4,#2]
```

We have one line for each word this time, as you can see by the way the numbers increase down the left-hand side, and that means one line for each instruction. The second column shows the characters that the bytes would stand for if they were ASCII codes and the third column the contents of the word.

Mnemonics

It is the next two columns which are of most interest to us. The first of these contains some letters which make up a mnemonic − a description of what the instruction does − and the column next to it tells us which registers or numbers it works with. This is a big step towards understanding what all those numbers mean!

As an example, look at the line at address &9008, which says:

```
SUB R3,R13,R2
```

This means ‘subtract the number in register 2 from that in register 13 and store the result in register 3’. Not much in itself, but a vital small component of a much larger program.

It’s possible to follow the workings of a program of this sort without troubling ourselves with the hexadecimal digits in the machine code instructions at all. We simply need to look at the mnemonics and the list of registers. Indeed, this is the way to create a machine code program − not by typing in all those incomprehensible numbers, but with the language of the mnemonics. This is known as *Assembly Language* and the software that you write it with is called an *assembler*. The software that we just showed being used to go the other way and produce assembly language from machine code is called a *disassembler*.

Writing Assembly Language

Fortunately for us, the Basic language in your machine includes a built-in assembler. We’ll see how to use it shortly. First, though, we need to understand something about the internal structure of the ARM or StrongARM processor.

As we saw at the beginning of this section, the processor contains a number of registers. In fact, there are 16 in action at any one time. The processor has several modes of operation and when it changes mode, some of the higher numbered registers are replaced by others. We’ll only ever write code for it to run in its User mode, so we are just concerned with 16 registers, numbered R0 to R15.

Register R15 is a very special one, known as the *program counter*. This stores the address that the next instruction is to be taken from and is increased by 4 each time an instruction is read, ready for the next one. Some instructions, known as *branches*, make the program jump to a different address by changing the number in the program counter, rather like a GOTO instruction in Basic.

Some other instructions, known as *branch linking* instructions, make the program jump to a new address but keep a note of the old one, so that it may return. This is rather like using a GOSUB command or calling a procedure in Basic. Remembering the return address is where register R14 comes in. This is known as the *link register* and branch linking instructions automatically store their return addresses in it.

Now let’s try writing some assembly language. The first thing we need is some memory in which to assembly the machine code. This is where our old friend the DIM command comes in. If you want to try it yourself, start up Basic and type in the following program, though you’ll also find it in the listings. We’ll look at how it works when you’ve done it:

```
 10 REM > M_Code1
 20 REM first experiment in assembly language
 30 ON ERROR REPORT:PRINT " at line ";ERL:END
 40 DIM code% 1000
 50 P%=code%
 60 [OPT 3
 70 MOV R0,#65
 80 SWI "OS_WriteC"
 90 MOV PC,R14
100 ]
```

Line 40 sets aside a block of 1000 bytes for our machine code which is enough for 250 instructions. We're not going to write anything like this number but it gives us plenty of space to expand our program without having to remember to increase it.

Line 50 sets variable P% to the start of this block. When the assembler is at work, it treats P% as the program counter, putting each instruction word in the address which it contains, increasing it by 4 after each instruction and using it to work out the numbers in branch instructions. The result of line 50 is that the first instruction will go into the first word of the block.

Controlling Assembly With OPT

The square bracket in line 60 turns on the assembler. The number after the OPT keyword controls how the assembler behaves. If bit 0 is clear (a zero), the assembled program isn't shown on the screen. If it's set (a 1), you can see what the assembler has produced. If bit 1 is clear, the assembler ignores references to Basic variables which it doesn't know about, which usually refer to addresses further down the program. If it's set, unknown variables produce an error. The number after OPT in this case is 3, which has both bits set, so the assembly will be shown on the screen and any errors reported.

Lines 70 to 90 contain the assembly language instructions which we'll look at in a moment. The closing square bracket in line 100 turns off the assembler.

When you RUN the program, you should see something like this:

```
00008FC4                OPT 3
00008FC4 E3A00041       MOV R0,#65
00008FC8 EF000000       SWI "OS_WriteC"
00008FCC E1A0F00E       MOV PC,R14
>
```

In this case Basic created the memory block at &8FC4. Your numbers may be different if you had more or fewer spaces in your program or some other difference.

The first instruction is a MOV command. This is one of the simplest commands and tells the processor to move a number into one of the registers. This might be the contents of another register, but in this case it's an *immediate* instruction, meaning 'move the number contained in this instruction into R0' − in other words, load R0 with 65. The thing that tells the assembler that it's an immediate instruction is the hash (#) symbol before the number.

The next instruction calls a software interrupt routine. You'll remember this one from the first part of this section. It's OS_WriteC which sends the number in R0 to the screen and you'll probably be aware that this short program is going to print an 'A' on the screen.

Getting Back to Basic

There's one last thing for the program to do, though. After printing the letter, it must return control to the Basic interpreter. When Basic makes the processor jump to your code, it does so with a branch linking instruction which means that the return address is stored in R14. To get back to Basic, all we have to do is copy this address into the program counter and that is what the MOV command in line 90 does. We can refer to R15 as 'PC' to remind us of what it does. Indeed, we can give any register the name of a Basic variable by setting up the variables before we start up the assembler.

When you typed RUN, the program only assembled the machine code, it didn't actually run the code itself. That's why there is no letter 'A' on the screen. To run the code, you have to CALL it. Before you do this, though, there is one golden rule about writing assembly language which you should follow; **always save it before you call the code**. It's always a good idea to save programs before you run them anyway of course, but it's particularly important when dealing with machine code. The Basic interpreter is full of routines to check for mistakes, which is why it can produce so many error messages. Machine code has none of these checks (unless you write them into your program) and any mistake can easily cause the entire machine to crash. The only way out of this situation is often to press the reset button or switch the machine off and on again, which would cause the loss of your program, if you hadn't saved it.

Having ensured that your program is safely stored on disc, you can run the machine code by typing:

```
CALL code%
```

The file M_Code1a in the listings has this command added, preceded by a 'REPEAT UNTIL GET' loop which waits for you to press a key. Provided you've run the program to assemble the code, Basic will know the value of code% and will load this number into the processor's program counter to make it jump to your code, You should now have your 'A' on the screen, followed by the Basic prompt or a 'Press space or click mouse' message.

Using Basic Variables

Because the assembler is part of the Basic interpreter, you can use Basic variables to represent numbers and addresses. This can make writing your program a lot easier. In our first example program, for instance, we told the machine to load R0 with 65, which we knew was the ASCII code for 'A'. To save looking it up, we could have written the line as:

```
MOV R0,#ASC("A")
```

ASC, you will remember, gives the ASCII code of the character in brackets, so this line produces exactly the same machine code instruction as the previous version.

It's also possible to use a Basic variable as a *label*. This is a way of marking a particular address so that you can make the program branch to it or move numbers in and out of it.

Let's try a slightly more complicated program:

```
 10 REM > M_Code2
 20 REM Demonstrates use of labels in assembly language
 30 ON ERROR REPORT:PRINT " at line ";ERL:END
 40 DIM code% 1000
 50 P%=code%
 60 [OPT 3
 70 .data
 80 EQUD 65
 90 .start
100 ADR R1,data
110 LDR R0,[R1]
120 SWI "OS_WriteC"
130 MOV PC,R14
140 ]
150 REPEAT UNTIL GET
160 CALL start
```

The word 'data' with a full stop in front of it in line 70 is a Basic variable used as a *label*. When the program gets to this line, data is given the current value of P% which, you will remember, is keeping track of the address of each instruction. In this example, in fact, data will have the same value as code%, the start of the block, but that doesn't matter.

The keyword EQUD is a way of putting a particular number into four bytes of memory when using the assembler. You could also use EQUW, which inserts two bytes, EQUB, which inserts one or EQUS, which inserts a string, as we shall see shortly.

To make it a little easier to remember which is which, EQUW means 'insert a word' and EQUD 'insert a double word'. This may seem a little strange as a word is four bytes.

The reason for this little anomaly is that BBC Basic was originally devised for the BBC Microcomputer, which was an 8-bit machine. It was decided at the time that 16 bits, or two bytes, would form a word. With the coming of 32-bit machines, 'word' has been redefined to mean 32 bits or four bytes. The original meanings of EQUW and EQUD have been kept for reasons of compatibility with the earlier machines.

Keeping Things Word-Aligned

Line 80, then, inserts number 65 into the first of four bytes at the beginning of the memory block. We're using EQUD rather than EQUB because it's important to use up four bytes so that the machine code instruction in the next line starts on a *word-aligned* address, that is one which is divisible by four. An alternative would be to use EQUB and add an extra line:

```
85 ALIGN
```

This increases P% if necessary, so that it's divisible by four.

Because the first word at the beginning of the block is no longer where the machine code instructions start, we need to tell the machine to jump to somewhere different when we call the code, so we'll use another label to mark the start of the code itself. What better name than 'start'! By the way, you can put a label on the same line as the instruction that follows it if you wish, provided you leave a space between the two. It's up to you. It's mainly a matter of making the program as readable and easy to understand as possible.

Instead of loading R0 directly with the number to be printed, we're going to load it from memory, so the first instruction has been altered. We can't directly tell the machine which address to load from because there are not enough spare bits available in the instruction word to hold an entire address, so we have to load the address into another register, R1. The simplest way to do this is with the ADR command. This isn't actually a machine code instruction at all. The assembler uses it to construct an instruction to add or subtract an appropriate number from the current value of the program counter.

Line 110 contains an instruction to load R0 from the address in R1, which is what we want.

To call this program, we mustn't CALL code% this time, or the processor will attempt to execute the 'instruction' in the first word which, of course, isn't an instruction at all. The result would be very unpredictable! To call the start of the code, we use the instruction:

```
CALL start
```

which will be executed when you press a key.

Because the number in the first byte is put there when the code is assembled, you can overwrite it between running the program and calling the machine code. If, for example, you were to type:

```
!data=66
```

the program would print 'B' instead of 'A'.

Two-Pass Assembly

You may be thinking ‘Why not put the data *after* the code so that the code starts at the beginning of the memory block?’. We can certainly do this, but there’s a snag. Line 100 refers to the label ‘data’, which is a Basic variable. As our program stands, Basic knows the value of data when it gets to this line, because it had already been set up in line 70. If data came at the end though, line 100 would have to set up an instruction referring to the value of a variable which it hadn’t yet met.

This is a job for OPT, which controls how the assembler behaves. You will remember that if bit 1 of the OPT number is zero, the assembler ignores errors of the ‘Unknown or missing variable’ type. It can’t assemble all the instructions properly because it doesn’t know the values to put in them but it can put something in their places.

The solution is to assemble the code *twice*. The first time, we go through it with an OPT value of zero (we don’t need to display it both times). The code is assembled but not all of it is correct. At the end of this first *pass*, however, Basic knows the values of all the variables which are used as labels. On the second pass, we use an OPT number of 3 and it does the job properly, displaying the result.

What better way of doing a job twice in Basic than with a FOR ... NEXT loop:

```
 10 REM > M_Code2a
 20 REM two pass assembly
 30 ON ERROR REPORT:PRINT " at line ";ERL:END
 40 DIM code% 1000
 50 FOR pass%=0 TO 3 STEP 3
 60   P%=code%
 70   [OPT pass%
 80   .start
 90   ADR R1,data
100   LDR R0,[R1]
110   SWI "OS_WriteC"
120   MOV PC,R14
130   .data
140   EQUD 65
150   ]
160 NEXT
170 REPEAT UNTIL GET
180 CALL start
```


Two Different Types of Pass

We need an OPT number of 0 for the first pass and 3 for the second. Line 50 provides this. Notice that the line setting up P% (line 60) is *inside* the FOR ... NEXT loop. This is important because both passes of the assembler have to begin with the same value of P% in order to assemble the code at the same place each time. At the end of the first pass, P% would have been increased to the address where the assembled code ends and that is where the second pass would start if we didn’t reset it.

When line 90 is reached on the first pass, Basic doesn’t know anything about the variable called data because it hasn’t reached line 130 in which it’s created, so it can’t create the instruction properly. Instead, it allocates one word to it and continues. By the time it gets to the NEXT command in line 160, it knows all about data and any other labels we used in the program. On the second time round the loop, it’s able to generate the ADR instruction properly.

Using Loops in Machine Code

Let’s try just one more example before we leave machine code. You’ll be very familiar now with things like REPEAT ... UNTIL loops and IF ... THEN structures. These devices are the key to any type of computer programming, whether in Basic, assembly language or anything else. We’re going to use a couple of them to print a string backwards.

Here is our final assembly language program in this guide:

```
 10 REM > M_Code3
 20 REM reverses a string
 30 ON ERROR REPORT:PRINT " at line ";ERL:END
 40 DIM code% 1000
 50 FOR pass%=0 TO 3 STEP 3
 60   P%=code%
 70   [OPT pass%
 80   .start
 90   ADR R1,data
100   MOV R2,#0
110   .loop1
120   SWI "OS_ReadC"
130   SWI "OS_WriteC"
140   STRB R0,[R1,R2]
150   ADD R2,R2,#1
160   CMP R0,#13
170   BNE loop1
180   MOV R0,#10
```

```
190   SWI "OS_WriteC"
200   .loop2
210   SUB R2,R2,#1
220   LDRB R0,[R1,R2]
230   SWI "OS_WriteC"
240   CMP R2,#0
250   BNE loop2
260   MOV R0,#13
270   SWI "OS_WriteC"
280   MOV R0,#10
290   SWI "OS_WriteC"
300   MOV PC,R14
310   .data
320   EQUS STRING$(100," ")
330   ]
340 NEXT
350 CALL start
```

We've dispensed with the loop that waits for you to press a key this time. When you run the program, the code will be assembled and CALLed immediately, and the cursor will appear on the next line, waiting for you to enter a string. Type in anything you like, then press Return. Your string will appear backwards on the next line.

```
Run SCSI::HardDisc.$.Martyn.First.Graphics.M_Code3
00009138             .start
00009138 E28F1048            ADR R1,data
0000913C E3A02000            MOV R2,#0
00009140             .loop1
00009140 EF000004            SWI "OS_ReadC"
00009144 EF000000            SWI "OS_WriteC"
00009148 E7C10002            STRB R0,[R1,R2]
0000914C E2822001            ADD R2,R2,#1
00009150 E350000D            CMP R0,#13
00009154 1AFFFFF9            BNE loop1
00009158 E3A0000A            MOV R0,#10
0000915C EF000000            SWI "OS_WriteC"
00009160             .loop2
00009160 E2422001            SUB R2,R2,#1
00009164 E7D10002            LDRB R0,[R1,R2]
00009168 EF000000            SWI "OS_WriteC"
0000916C E3520000            CMP R2,#0
00009170 1AFFFFFA            BNE loop2
00009174 E3A0000D            MOV R0,#13
00009178 EF000000            SWI "OS_WriteC"
0000917C E3A0000A            MOV R0,#10
00009180 EF000000            SWI "OS_WriteC"
00009184 E1A0F00E            MOV PC,R14
00009188             .data
00009188                     EQUS STRING$(100," ")
This string has been reversed by a machine code program.
.margorp edoc enihcam a yb desrever neeb sah gnirts sihT
```

Your string will appear backwards on the next line

The beginning and end of the program are the same as before. Virtually all assembly language programs start and finish like this, defining a block of memory to hold the assembled code, setting up a FOR ... NEXT loop to do two-pass assembly and setting P% to the start of the block.

The string that we type in will be put into a block of memory called a buffer. This is set up by defining a string at the address where our 'data' label is located, consisting of 100 spaces (you can increase this up to 255, if you think you'll want to type in a longer string than this). As the block which holds the assembled program is 1000 bytes long and the rest of it isn't used for anything else, it probably doesn't matter whether we set up this string or not, but it would matter if we were to assemble some more code following this buffer.

The machine code starts in line 90 in the same way as in our previous program, by loading R1 with the address of the buffer. We also load zero into R2, using a MOV instruction in immediate mode.

Line 110 is the start of our first loop. There isn't a special instruction at the start of a loop − just a label so that we can set up a branch instruction to jump back to it at the end.

The first action of the loop is to read a character from the keyboard. You will remember we did this in the first part of this section, dealing with SWIs. We're doing it the same way here, using OS_ReadC which returns with the ASCII code for the character in R0. The following line prints the character on the screen so that we can see it.

Storing Bytes in Memory

Line 140 stores the character in the buffer. We saw an LDR instruction, which loads a word into a register, in our earlier examples. There is a similar instruction, STR, which stores a word. The instruction in line 140, though, is STRB, which means 'store a single byte'. It stores the bottom eight bits of R0 at an address determined by whatever is between the rectangular brackets.

In this case, the instruction means 'store the byte that's in the bottom eight bits of R0 at the address found by adding the contents of R1 and R2 together'. Because R2 contains zero the first time round the loop, this is the same as the address in R1, which is the first byte of the buffer at the 'data' label. After doing this, though, we increase R2 by 1 which we do in line 150 by adding 1 to the number in R2 and storing the result back in R2. On the next time round the loop, the character will go into the next byte of the buffer. This process continues until we press Return.

The instruction in line 160 compares two numbers, in this case the contents of R0 and the immediate number 13. Doing this has an effect on the processor's flags which were referred to in the section of this section on SWIs. If the two numbers are equal, the zero flag will be set to one, otherwise it will be cleared to zero. We make use of this fact in the following line.

What the CMP instruction is doing is checking to see if we've pressed the Return key, which produced ASCII code 13. If we have, the instruction will set the zero flag.

The BNE instruction in line 170 means '**B**ranch if **N**ot **E**qual'. This really means 'branch to the loop1 label if the zero flag isn't set'. As we've seen, this will be the case if the character we typed isn't a Return.

Once we type a Return, then, the program doesn't branch back to the beginning of the loop, but carries straight on. Imagine the situation if we had typed 'ABCDEFGH' and pressed Return. We would have gone round the loop a total of nine times and the buffer would contain nine characters − eight ASCII codes for letters A to H and one code 13 for the Return, which would have been stored in address data+8. The number in R2 would be 9 because it was increased by 1 after storing the Return code.

Reading the String Out Again

The effect of sending code 13 to the screen would have been to send the cursor back to the beginning of the line. To avoid overwriting the string we've just entered with the one we're about to produce, we need to move it down one line, which requires a Line Feed character (code 10). Lines 180 and 190 send this to the screen. Having done that, we can start reading the string backwards out of the buffer and printing it.

Because of the way things were after we left the first loop, R1+R2 together contain the address of the byte *following* the Return character so we'll start our second loop by subtracting 1 from R2. We then read a byte from the buffer, send it to the screen and go back to subtract another 1 from R2. By this means, we read the string backwards. When we've dealt with the first character that we'd typed in, R2 will have zero in it. After printing each character, we check for this in line 240 and branch back for another character if we haven't reached the beginning of the string.

If we returned to Basic at this point, the Basic prompt would reappear on the end of the reversed string. It would be nice to tidy up the program by moving it to the beginning of the following line, which again needs a Return and a Line Feed. Lines 260 to 290 send these two characters to the screen and we finally return to Basic in line 300.

Graphics in Assembly Language

The previous example will have shown you how to use VDU calls in an assembly language program. If you look at Appendix 2 you will see that VDU calls can do a great many things, from making a beep to changing screen mode. You can do all these things using OS_WriteC.

You will also discover from Appendixes 2 and 3 that virtually any graphics drawing operation can be done using a PLOT operation and that PLOT may be reduced to VDU

25 plus a further five bytes. You could certainly do this in assembly language by calling OS_WriteC six times but there is a quicker way. Basic's PLOT command uses a special SWI called OS_Plot, which cuts the number of operating system calls from six to one.

The PLOT code is in register R0, R1 contains the x coordinate and R2 the y coordinate. The Basic command:

```
PLOT 85,300,200
```

for example, draws a filled triangle between (300,200) and the previous two points visited by the graphics cursor. Using the SWI, this becomes:

```
SYS "OS_Plot",85,300,200
```

The Meaning of RISC

You may have noticed from these examples that the command for loading an immediate number into a register is the same as that for copying the contents of one register into another and it's even the same as the one which returns from a subroutine or hands back to Basic at the end of our machine code. In each case, it's done with a MOV command.

This is an example of the way in which a few instructions have a great many uses in the ARM or StrongARM processor. Because it uses fewer instructions than some other processors, it's known as a **R**educed **I**nstruction **S**et **C**omputer, or RISC for short.

The great advantage of a RISC processor is its speed − because there are fewer types of instruction, it takes less time to work out which one it's dealing with. That's why your RISC OS machine is so fast compared with some other computers.

Another reason for its high speed is that every instruction is contained within one four-byte word, including all the numbers that it has to work with. Because your data bus is 32 bits wide, these four bytes are loaded into the processor in one fell swoop.

Some other processors have to load an instruction, check what it is, then load some more bytes containing the required numbers. This takes more time. The ARM processor, in fact, loads one instruction, decodes it while loading the next one and executes it while loading the third one and decoding the second!

As you will appreciate, there is a great deal more to software interrupts and machine code than we've dealt with in this section. You should, however, have a good idea of how the SYS command allows you to use SWIs in Basic and how the assembler works.

We relied in this last program on the fact that both OS_ReadC and OS_WriteC had no effect on the contents of R0, R1 and R2, except for the action of OS_ReadC, which puts

the code for the key pressed in R0. This isn't always the case; some SWIs change the numbers in some of the registers they use.

To use SWIs, you need complete details of what they do and which registers they use. All these details are to be found in the *RISC OS Programmer's Reference Manual*.

You'll also appreciate that we only scratched the surface of machine code and assembly language, just introducing the instructions that we needed for our little experiments.

15

A Wimp Front End

Although by now you can doubtless produce some weird and wonderful programs of your own, you may be envious of commercially-produced packages which operate in the desktop environment with their windows, menus and icons on the icon bar.

The programs we've met so far have not exploited one of the most powerful features of RISC OS and the Acorn machine − its ability to *multi-task*, that is have several programs running at the same time, using its Wimp environment. Wimp stands for **W**indows, **I**cons, **M**enus and **P**ointers.

Moving into the multi-tasking environment is where this guide gives way to *A Beginner's Guide to Wimp Programming* but we will allow ourselves a little dabble at it in this section; enough to produce a multi-tasking 'front end' containing our Munchie program. We will use the version that we produced in the previous section which introduced a user sprite area. You could probably use this technique to run any program you like, with a little modification. It won't multi-task while the game is running but it will allow it to sit on the icon bar so that you can call it in the middle of word-processing or whatever else you're doing. It will also get round the problem of having to keep the sprites file in the currently selected directory.

From Munchie to !Munchie

You may already have discovered that an application consists of a directory whose name begins with an exclamation mark (!). Within this directory are the files used by the application. To open an ordinary directory, we double-click on it but, of course, doing this on an application directory runs the application. To open such a directory, hold down Shift while double-clicking.

If you do this on any application, you will find a variety of files inside, depending on how the application works. Almost certainly there will be three files called !Run,

!RunImage and !Sprites. The !Sprites file contains the sprite used to represent the application directory in its parent directory window. The !RunImage file contains the body of the program and will probably be either a Basic or machine code file.

You will find versions of the !Munchie application in the mode 12 and mode 27 subdirectories. The mode 12 version contains, in addition to the MunSprites file, another sprite file called !Sprites which itself contains one sprite; a copy of the 'mun_right' sprite but with the name '!munchie'. When RISC OS opens the parent directory containing the application, it looks for this file, then looks inside it for a sprite with the same name as the application. If it finds one, it loads it into a special sprite area called the Wimp sprite area and uses it to represent the application directory in the parent directory window.

... there will be three files ...

The mode 27 version has the MunSprites and !Sprites files, and also another one called !Sprites22. This file also contains a '!munchie' sprite but there is a difference between the two. The one in the !Sprites file has rectangular pixels, as used in mode 12, and the one in the !Sprites22 file has square ones, as used in mode 27. If the computer is operating in a rectangular pixel mode when you open the directory window, it loads the !Sprites file; if it's in a square pixel mode, it loads the !Sprites22 file. In this way, it uses whichever sprite would look best on the screen.

If you wish to create your own application directory, you start in the same way as for an ordinary one, by clicking Menu over a filing system window and choosing the appropriate option. If you call it '!Munchie', it will be represented by an application icon. Clearly, the first thing we need is a sprite file so that our application will be properly represented in its parent directory window.

If you created your own Munchie sprites, measuring 34 pixels by 17 in mode 12 or 34 by 34 in mode 27, you can use the one called 'mun_right' that you already have. It's exactly the right size and will look good in any desktop mode. If you have the mode 27 version, you can use this sprite in the !Sprites22 file and also make a copy of it with rectangular pixels (using Paint's 'Use sprite as brush' option) for the !Sprites file. Open your !Munchie directory and copy the munsprites file into it, then load the file into Paint, keeping the directory window open. Click Menu over the Mun_right sprite, go to the **Sprite 'mun_right'** submenu and through to **Save**. This will let you save a file containing just this sprite. Before saving, change its filename to '!Sprites' (or '!Sprites22' − don't forget the '!'), then drag it into the !Munchie directory window.

Load this new file into Paint, change the sprite name to '!munchie' and resave it. The sprite should have exactly the same name as the application except that, like all sprite names, it should all be lower case.

The colours for the game sprites were chosen to look correct using non-desktop colours because that is how they are used in the game. The new !munchie sprite will be used in an icon on the desktop so you may have to change its colours using Paint so that it looks correct.

Provided you have the sprite name correct and its file is called !Sprites, the application icon will now be replaced in the parent directory window by your Munchie sprite, though you may have to reset the machine first for this to happen.

Your application directory will now contain more than one sprite files. We're keeping the sprites used by the game separate from the one in the !Sprites file for a very good reason. The game sprites will be loaded into a user sprite area using the technique which we learnt in the previous section and will not continue to occupy any memory once we've quit the game. The sprite in the !Sprites file will go into the Wimp sprite pool, as we shall see shortly, and will remain there until we switch off or reset the machine.

The !Run File

If you were creating an application directory from scratch and tried clicking on it as though you were trying to run the application, you would get an error message saying something like 'File 'ADFS::Games.$.!Munchie.!Run' not found'. The operating system would have tried to run a file called '!Run' in the !Munchie directory and, of course, it couldn't find it!

A !Run file is almost always an Obey file. This is a special type of file containing commands which are executed by the command line interpreter, just like the star commands which we used at the beginning of this guide. To create such a file, start up your text editor and choose the appropriate option. If you're using Edit, click Menu on its icon and go to the **Create** submenu. This presents you with a choice of filetypes. Choose **Obey** and a window will appear, ready for you to type in the following listing:

```
|Run file for Munchie
Set Munchie$Dir <Obey$Dir>
IconSprites <Munchie$Dir>.!Sprites
WimpSlot -min 192k -max 192k
Run <Munchie$Dir>.!RunImage
```

The first line is a comment, like a REM line in Basic. The '|' character (a shifted backslash) makes the CLI ignore the rest of the line.

The second line sets a system variable. You can see all the system variables by typing;

```
*Show
```

Once a system variable has been set up, it stays there until either it is removed with a *Unset command or the machine is reset. You may notice that any application which the machine 'knows' about will probably have a system variable named after it. The convention in RISC OS appears to be to put a dollar ($) sign in the middle of the name, so we will call our system variable Munchie$Dir.

Learning to Obey$Dir

One of the existing system variables is called Obey$Dir. If you can find it in the list, you'll see that its contents (after the colon) are the pathname of a directory. In fact, it contains the pathname of the last Obey file that was executed. Every time the filing system runs an Obey file, it sets Obey$Dir to the pathname of the directory containing the file.

When the CLI executes a line containing angular brackets (<>), it looks for the name of a system variable between then and replaces it with its contents. This means that our new variable Munchie$Dir will have the same contents as Obey$Dir. Because the line creating it is part of an Obey file, this is the pathname of the directory containing the file being executed, which is also our application directory. This is important because it gives our application a way of finding its files wherever it's located, whether in the root directory of a floppy disc or deep in the directory structure of a hard disc.

The next line loads the !Sprites file into the Wimp sprite pool. It is probably not necessary to do this as it will have been done by the filer when it opened the parent directory but it's best to make sure. The !munchie sprite is the one we're going to put on the icon bar, so we need it in the Wimp pool and this line ensures that it will be there if you should happen to run the application from the command line.

If we were to call the !RunImage file at this point, the program would run and occupy all the memory which the task manager has allocated to the 'next' application, even though it may not need that much. This would be very wasteful and tie up memory which other applications could use so we must first set the size of the 'slot' which our program will occupy. This is the job of the WimpSlot command in the next line of the !Run file. The two suffixes '-min 192k' and '-max 192k' together ensure that this is the amount allocated. It may seem a lot for what is not a very long program but most of it is taken up by the sprite user area. In the mode 12 version, these figures are changed to 128k because the sprite area is smaller.

Incidentally, if your machine has 1Mbyte of RAM, a Wimp slot is allocated in blocks of 8k. For a 2Mbyte machine, this becomes 16k and, for 4Mbytes, 32k. The Risc PC and later machines use blocks of 4k, whatever the size of their memory.

If you try to run the application again, the error message will change to something like 'File 'ADFS::Games.$.!Munchie.!RunImage' not found' because, of course, the last line of the !Run file calls a file which we haven't yet written. You will find, though, that your system variable has been set up.

The !RunImage File

Most of this file will be copied from the last version of the Munchie program but we need to add a bit to the beginning to make it multi-task.

If you're working from scratch, create a new Basic file and type in the following which is the minimum that will allow the program to run without crashing. Lines 160 to 240 can be copied from PROCinit in the Munchie program but line 170 has a bit added to it:

```
 10 REM > !RunImage
 20 ON ERROR REPORT:PRINT "at Line ";ERL:END
 30 REM Munchie game with Wimp front-end
 40 PROCinit
 50 PROCicon
 60 REPEAT
 70   SYS "Wimp_Poll",,b% TO r%
 80   CASE r% OF
 90     WHEN 6:PROCmouseclick
100     WHEN 17,18:PROCmessage
110   ENDCASE
120 UNTIL quit%
130 SYS "Wimp_CloseDown",task%,&4B534154
140 END
150 :
160 DEFPROCinit
170 DIM sblock% 165620,b% 100
180 !sblock%=165620:sblock%!8=16
190 SYS "OS_SpriteOp",9+256,sblock%
200 SYS "OS_SpriteOp",10+256,sblock%,"<Munchie$Dir>.MunSprites"
210 dotnum%=9
220 width%=68:height%=68
230 hstep%=10:vstep%=10
240 DIM dot%(dotnum%,1)
250 quit%=FALSE
260 SYS "Wimp_Initialise",200,&4B534154,"Munchie" TO ,task%
270 ENDPROC
280 :
290 DEFPROCicon
300 !b%=-1:b%!4=0:b%!8=0:b%!12=68:b%!16=68:b%!20=&3002
310 $(b%+24)="!munchie"
320 SYS "Wimp_CreateIcon",,b%
```

```
330 ENDPROC
340 :
350 DEFPROCmessage
360 IF b%!16=0 quit%=TRUE
370 ENDPROC
380 :
```

The beginning of PROCinit does the work of part of Munchie's PROCinit. This consists of actions such as loading the Mun_sprites file, which should only be done once, before you start to play the game. We've left out the commands which modify the sound system for now − we don't want to change the beep yet!

The addition to line 170 creates a data block of 100 bytes. We'll use this several times with the SYS commands that control the Wimp. Line 250 creates variable quit%. The application continues to run as long as it remains FALSE.

At line 260, we meet the first of these SYS commands. The first thing to do is initialise our program in the multi-tasking environment which we do with the Wimp_Initialise call. The number in R0 is the version number of the operating system that we're writing for, multiplied by 100. Unless you're using special new features that only RISC OS 3 and later versions can handle, always make this number 200 so that your program could run under RISC OS 2.

The hexadecimal number &4B534154 in R1 is, in fact, the ASCII code for 'TASK' (the number is stored in memory low byte first, &54 being the code for 'T' and &4B the code for 'K'. This special number tells the Wimp that the program was written to run under RISC OS and not the older Arthur operating system. The string in R2 is the program name for showing in the Task manager window.

The number returned in R1 is the *task handle* for this program. We store it in variable task% in case we need it later.

Following initialisation, the next job is to create an icon on the icon bar. PROCicon does this by setting up the data block and calling Wimp_CreateIcon with the address of the start of the block in R1. Line 300 sets up the various four-byte words of the block. The first of these at b% contains the *window handle* or number of the window in which we want to put the icon. The 'window' in this case is the icon bar which has two handles when it comes to creating icons; −1 for an icon on the right-hand side and −2 for one on the left.

The next four words, from b%+4 to b%+16, contain the minimum and maximum x and y coordinates of the icon. We don't actually know where on the bar our icon will be but in this situation we can make the minimum x and y coordinates zero and the maximum ones the width and height of the sprite − both 68 OS units.

The next word at b%+20 contains 32 bits called *flags* which describe the nature of the sprite. We won't go into them in this guide except to say that bit 1 is set, meaning that the icon contains a sprite, and the number 3 in bits 12 − 15 means that our application is called whenever the mouse is clicked over the icon. Finally, the bytes from b%+24 onwards contain the sprite name, entered in line 310.

Having set up all this information, we put the address of the data block in R1 and call Wimp_CreateIcon in line 320. Although the information in the data block will still be there after the icon has been created, we don't need it any more and the data block can be used for other things.

Polling the Wimp

Returning to the main part of the program, we now enter a loop between lines 60 and 120 which we keep going round as long as quit% remains FALSE. Line 70 calls Wimp_Poll, passing it the address of the data block in R1.

When the call returns, the number in R0 is put into variable r%. This number is a *reason code* which tells us if anything has happened that our program needs to know about. We check for various values of reason code using the CASE ... OF structure between lines 80 and 110, then check that quit% is still FALSE and go back to the top of the loop to call Wimp_Poll again.

That, at least, is what appears to happen. In practice, between our call to Wimp_Poll and its return, the machine may have handed control to every multi-tasking application that is running. Each one in turn checks its reason code, does whatever it has to do and calls Wimp_Poll again, allowing the machine to pas on control to the next application. Provided all the programs behave themselves, the system works. This is the heart of the multi-tasking system and the secret of how several programs can run at the same time, but our program is blissfully unaware of all this as it goes round and round its polling loop.

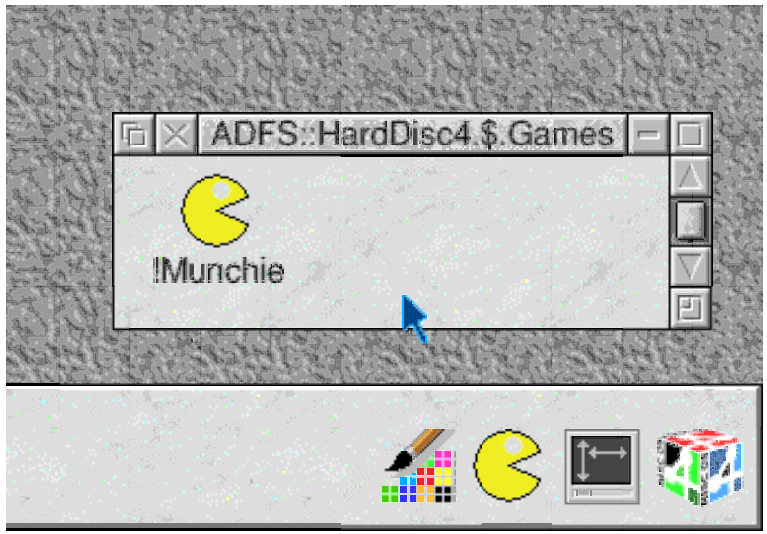
The CASE ... OF structure only checks for the reason codes that we're interested in and does nothing if any others arrive. Codes 17 and 18 both mean that the operating system has sent us a message, the details of which are in the data block. If either of these reason codes is received, line 100 calls PROCmessage to deal with the situation.

The nature of the message is indicated by the *message action code* which is the number in the word at b%+16. The only message we're interested in is one with action code zero which is an instruction to our application to close down. If this number is found in b%+16, line 360 simply sets quit% to TRUE. The next time the program gets to the end of the polling loop at line 120, it carries straight on. Line 130 calls Wimp_CloseDown which does the opposite of Wimp_Initialise. We pass it our task handle in R0 and the

ASCII code for ‘TASK’ in R1. The program then ends and our icon disappears from the icon bar.

If you now double-click on the !Munchie icon, the program should load and its icon will appear on the icon bar. If it doesn’t appear, it could be because the sprite name in line 310 doesn’t match the name of the sprite in the !Sprites file or because the program isn’t going round the Wimp_Poll loop properly.

Avoid the temptation for now to click the mouse on your new icon. Instead, open the Task manager window. You can do this, if you have RISC OS 3 onwards, by clicking on the icon at the right-hand end of the icon bar. If you have RISC OS 2, the icon will be an ‘A’ character. You need to click Menu on this and choose the **Task display** option.



... its icon will appear on the icon bar

The top section of the window is headed ‘Application tasks’ and shows the names of any applications which are running including, hopefully, Munchie with 128k or 192k allocated to it (this is where the operating system makes use of the name in R2 when we called Wimp_Initialise). If you click Menu anywhere along this line, you will get a menu including an option called **Task 'Munchie'**. This leads to a submenu with one item, **Quit**. If you select this, you should find that the Munchie line will disappear from the Task manager window and your icon from the icon bar. Your application has closed down.

What has happened is that the operating system has sent the application a message through Wimp_Poll and PROCmessage has dealt with it.

Dealing with Mouse Clicks

If you’re typing in your own version. there is another way of shutting down the application at this point. If you start it up again and click the mouse over the icon, you will get an error message saying ‘No such function/procedure at line 90’. When you click the mouse, Wimp_Poll returns with a reason code of 6, meaning that the mouse has been clicked over a window or icon belonging to us. Line 90 tries to send the program to a procedure which we haven’t yet written. That is the next part to add:

```
390 DEFPROCmouseclick
400 CASE b%!8 OF
410   WHEN 1:quit%=TRUE
420   WHEN 4:PROCplay
430 ENDCASE
440 ENDPROC
450 :
```

The data block contains information such as which button was clicked, the window and icon the pointer was over when the mouse was clicked and its x and y coordinates. Because we’re keeping things simple in this program, we only need to know which button was clicked. This information is contained in the word at b%+8, which is examined by the CASE ... OF structure.

This number is the same as the third number returned by the MOUSE command which we met in the MouseLines program in section 6. The three least significant, or right-hand, bits reflect the state of the mouse buttons, each one being a 1 if the corresponding button was pressed. Thus, if you click Adjust, bit zero will be set, putting 1 in b%+8. If Menu is clicked, b%+8 will be 2 and if Select is clicked, it will be 4. Lines 410 and 420 check for Adjust and Select respectively.

This arrangement allows us to either play the game by clicking Select over our icon or quit the application by clicking Adjust. If we do the latter, line 410 sets quit% to TRUE, resulting in the program closing down, as we saw earlier. Clicking Select leads us to PROCplay and the start of the game itself:

```
460 DEFPROCplay
470 mode%=MODE
480 MODE 27:REM use mode 12 for standard resolution monitor
490 VOICES 2
500 VOICE 2,"WaveSynth-Beep"
510 VOICE 1,"StringLib-Hard"
520 BEATS 200
530 bar_sent%=FALSE
```

```
540 REPEAT
550   PROCgame
560 UNTIL (char% AND &DF)<>ASC("Y")
570 VOICES 1
580 VOICE 1,"WaveSynth-Beep"
590 SYS "Wimp_SetMode",mode%
600 ENDPROC
610 :
```

This may look familiar; most of it consists of the remainder of the PROCinit and main loop of Munchie. We leave it until now to redefine the voices because, of course, we don't want to modify the beep while our icon is just sitting on the icon bar!

We have added two new lines to the procedure; line 470 to keep a note of the screen mode before we change it and line 590 to restore the desktop. When we get tired of playing the game, the loop between lines 540 and 560 stops repeating, the sound system is restored to normal and the screen mode returned to what it was. It is important to do this with the Wimp_SetMode command rather than using the Basic MODE keyword because this command restores the desktop palette (the colour definitions) and redraws everything in the desktop.

The Remainder of the Game

So far, we've accounted for the main part of the old Munchie program, together with its PROCinit. The rest of the procedures can be copied into the new version of the program without alteration, except for a few changes to the end of PROCgame:

```
800   PROCmove
810 UNTIL dotleft%=0 OR char%=27
820 IF char%<>27 THEN
830   PRINT TAB(10,25)"You took ";(TIME-start%)/100" seconds"
840   PRINT TAB(10,26)"Another go? (y/n)"
850   REPEAT
860     char%=GET
870   UNTIL (char% AND &DF)=ASC("Y") OR (char% AND &DF)=ASC("N")
880 ENDIF
890 ENDPROC
900 :
```

In our previous, non-multitasking version, if we wanted to stop playing the game halfway through, we could simply press Esc. In the Wimp environment, the escape key is usually disabled and just acts as a normal key, producing ASCII code 27.

In order to be able to stop the game with Esc, we need to detect code 27. When we get to line 810, the variable char% contains the code for the last key pressed, produced by

PROCmove. We've modified this line to stop the loop repeating when we press Esc and added lines 820 and 880 to skip the last part of PROCgame and return us straight to PROCplay. Line 560 in PROCplay stops its own loop repeating under these circumstances (27 AND &DF is not equal to the ASCII code for 'Y') and we return to the desktop.

Conclusion

This is the beginning of Wimp programming. We haven't used any windows, menus or icons other than the one on the icon bar, or made any use of the mouse pointer, apart from detecting a button click over the icon, but your program can co-exist in the memory with others and be available for you to run at any time − a multi-tasking application.

And that's it. You've taken your first steps in programming your RISC OS computer. We've covered quite a lot of ground and seen what a great many of the Basic keywords do and how to use them. For a fuller description of all the keywords, see Appendix 1.

Hopefully, while working through this guide, you experimented with writing your own programs. If you did, you'll realise that your RISC OS computer is not only a very powerful tool capable of running sophisticated software; it's also very easy to bend it to your will and make it do something special which only you want it to.

And it's also great fun!

Appendix 1

Complete List of Basic Keywords

This appendix is for reference. It isn't meant to be read through from beginning to end − you would find it rather repetitive if you did. It lists all the keywords in the version of BBC Basic used in RISC OS 3 onwards. You are unlikely to use some of them, but they are all listed here for completeness.

ABS

Gives the magnitude of its argument, i.e. turns negative numbers positive.
Example:

```
PRINT ABS(-5)
5
PRINT ABS(7)
7
```

ACS

Gives the arc-cosine of the number following it in brackets.
Example:

```
ang=ACS(x)
```

In this example, ang is the angle in radians whose cosine is x. See DEG and RAD for conversion between radians and degrees.

See also COS

ADVAL

Reads data from an analogue port, if fitted, or gives data on one of the machine's buffers.
Example:

```
num%=ADVAL(-1)
```

The keyboard buffer stores keypresses which have not been read by software. The variable num% in the above example will show the number of characters waiting in the buffer.

The following program will make the machine beep once every seven keypresses:

```
10 REPEAT
20   num%=ADVAL(-1)
30   IF num%>6 THEN
40     VDU 7
50     *FX15
60   ENDIF
70 UNTIL FALSE
```

The command *FX15 flushes all buffers, i.e. empties them.

AND

Applies a logical AND operation (see section 9), either to multiple statements following an IF keyword or to the individual bits of two numbers.
Examples:

```
num%=char% AND &DF
IF x%=2 AND y%=3 AND z%=4 THEN PROCdo_this
```

In the first example, &DF is used as a mask. Any bit in num% whose corresponding bit in &DF is a 1 has the same state as the equivalent bit in char%. Any bit corresponding to a 0 in &DF is forced to be a zero. The effect of this example is that bit 5 of num% is always zero. When applied to ASCII codes, this has the effect of turning lower case letters into capitals.

In the second example, PROCdo_this is only executed if x% equals 2 *and* y% equals 3 *and* z% equals 4.

See also EOR, OR.

APPEND

Loads a file and adds it onto the end of the Basic program in memory.
Example:

```
APPEND "0.$.Procs"
```

If the file contains a Basic program, its lines are automatically renumbered so that they are added onto the end of the first program.

ASC

Gives the ASCII code of the first character in a string
Example:

```
char%=ASC("HELLO")
```

char% will be 72, which is the ASCII code for letter H.
See also CHR$.

ASN

Gives the arc-sine of the number following it in brackets.
Example:

```
ang=ASN(x)
```

In this example, ang is the angle in radians whose sine is x. See DEG and RAD for conversion between radians and degrees.

See also SIN.

ATN

Gives the arc-tangent of the number following it in brackets.
Example:

```
ang=ATN(x)
```

In this example, ang is the angle in radians whose tangent is x. See DEG and RAD for conversion between radians and degrees.

See also TAN.

AUTO

Puts line numbers on the screen automatically when entering a program, to save you having to type them in.
Examples:

```
AUTO
AUTO 100
AUTO 100,5
```

The first example will start a program with line 10, increasing the numbering in tens. The second will start numbering at line 100. This is useful if you had stopped entering a program after, say, line 90 (perhaps because you'd made a mistake) and you wanted to restart.

The third example will also start at line 100, but will number the lines in fives instead of tens.

To get out of AUTO mode, press Esc.

BEAT

Gives the current beat value, used when dealing with sound.
Example:

```
count%=BEAT
```

The beat counter counts at a rate set by TEMPO until it reaches a number set by BEATS, when it is reset to zero. BEAT gives its current value.
See section 12 for an description of the sound system.

BEATS

Either gives or sets the number that the beat counter counts up to, at a rate determined by TEMPO, when dealing with sound.
Examples:

```
BEATS 3000
PRINT BEATS
```

The first example sets the value of BEATS, the second reports it.

See section 12 for an description of the sound system.

BGET#

Reads a single byte from an open file. An open file is one which has been opened with the command OPENIN, OPENOUT or OPENUP.
Example:

```
char%=BGET#handle%
```

The variable handle% is the *file handle*, which is a number given to the file when it is opened, using OPENIN, OPENOUT or OPENUP.

BPUT#

Writes a byte or a string to an open file. An open file is one which has been opened with the command OPENUP or OPENOUT.
Examples:

```
BPUT#handle%,char%
BPUT#handle%,name$
```

The variable handle% is the *file handle*, which is a number given to the file when it is opened, using either OPENUP or OPENOUT.

The first example writes a single byte to the file. Its position in the file is determined by the file pointer PTR#. The second example writes the ASCII codes of all the characters in the string to the file. This is followed by a Line Feed character (ASCII code 10) unless the name of the string is followed by a semi-colon(;). The value of PTR# is increased by one for each character written.

BY

Can be added to MOVE, DRAW, POINT and FILL to define coordinates relative to the last position.
Example:

```
DRAW BY 100,0
```

will draw a horizontal line from the current graphics cursor position to a point 100 OS units to the right.

CALL

Enters a machine code program from Basic.
Example:

```
CALL code%
```

The variable code% will have been set up to contain the address of the start of the machine code program, possibly by using the DIM command.

When the program is entered, the ARM or StrongARM processor's registers R0 to R7 are loaded from Basic variables A% to H% respectively.

CASE

Used as part of CASE ... OF ... ENDCASE.
Example:

```
CASE a% OF
  WHEN 1:PROCdo_this
  WHEN 2:PROCdo_that
  WHEN 3:PROCdo_also
  OTHERWISE
  PROCdo_something_else
ENDCASE
```

The value of the variable following CASE is checked against each of the numbers or variables following the WHEN keywords in turn. If a match is found, the program executes the rest of the line following the colon and then proceeds straight to ENDCASE, ignoring the lines in between (it will never execute more than one procedure). If a match has not been found by the time OTHERWISE is reached (which is optional), PROCdo_something_else is executed.

CHAIN

LOAD and RUN a Basic program.
Example:

```
CHAIN "MyFile"
```

CHR$

Gives a one-character string determined by an ASCII code.
Example:

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

See also ASC.

CIRCLE

Draws a circle.
Examples:

```
CIRCLE 600,500,200
CIRCLE FILL 400,300,200
```

The first example will draw a circle centred on x coordinate 600, y coordinate 500, with radius 200 OS units.

The second example will draw a solid circle with the same radius, centred on (400,300).

CLEAR

Removes all variables. This is particularly useful if you want to redefine an array, whose dimensions cannot usually be altered once created.

This command dos not affect the Resident Integer Variables A% to Z%.

CLG

Clears the graphics window to the graphics background colour.

This command also resets the graphics cursor to the graphics origin (0,0).

See also CLS.

CLOSE#

Closes a file which had been opened with OPENIN, OPENUP or OPENOUT.
Example:

```
CLOSE#handle%
```

The variable handle% is the file handle, which was assigned to the file when it was opened.

CLOSE#0 will close all files on the current filing system.

CLS

Clears the text window to the text background colour.

This command also resets the text cursor to the top left-hand corner of the screen and resets COUNT to zero.

See also CLG.

COLOUR (also spelt COLOR)

Sets the text colour or alters the palette settings.
Examples:

```
COLOUR 1
```

sets the colour to logical colour 1 (usually red in 16-colour modes).

```
COLOUR 1,3
```

redefines logical colour 1 to be physical colour 3 (yellow). A subsequent COLOUR 1 command will produce yellow.

```
COLOUR 1,240,128,0
```

sets the proportions of red, green and blue in colour 1 to 240, 128 and zero respectively, to give orange.

See also TINT.

COS

Gives the cosine of the angle following it in brackets.
Example:

```
width%=length%*COS(angle)
```

where angle is in radians.

See DEG and RAD for conversion between radians and degrees.

See also ACS.

COUNT

Gives the number of characters printed since the last line feed was sent to the screen.

This function counts the number of characters sent to the screen by PRINT, INPUT or REPORT, but not VDU.

CRUNCH

Used to remove REM statements and unwanted spaces from the current program. The action is determined by which bits are set in the number following CRUNCH:

Bit	Deleted
0	spaces before statements
1	spaces within statements
2	REMs (except in the first line, which may contain an embedded filename)
3	empty statements
4	empty lines

Examples:

```
CRUNCH 4
```

This has bit 2 set and will just delete REMs

```
CRUNCH 31
```

In hexadecimal notation, this would be CRUNCH &1F. This has all bits 0 − 4 set and so will delete everything shown above.

DATA

Used in conjunction with READ to read values into variables.
Example:

```
READ x%,y%,z%
FOR n%=1 TO 3
  READ day%(n%)
NEXT
DATA 5,6,7
DATA Monday,Tuesday,Wednesday
```

The first line of this mini-program reads numbers 5, 6 and 7 into the values of x%, y% and z% respectively. The FOR ... NEXT loop in the following three lines sets day%(1), day%(2) and day%(3) to Monday, Tuesday and Wednesday respectively.

The data may be read as a number or a string, depending on the variable following the READ command.

DEF

Used with PROC or FN to begin the definition of a procedure or function.
Examples:

```
DEFPROCdelay(time%)
DEFFNsize(x%)
```

You may put a space after DEF if you wish, but there must be no space between PROC or FN and the procedure or function name.

DEG

Converts an angular measurement in radians into degrees.
Example:

```
twist=DEG(angle)
```

If angle is the size of an angle in radians, twist is its size in degrees.

One radian is approximately 57 degrees. There are 2*PI radians in a circle.

See also RAD.

DELETE

Used to delete lines of a program.
Example:

```
DELETE 100,150
```

This will delete all lines between and including lines 100 and 150.

It is not necessary to use DELETE when deleting one line. Just type the line number and press Return.

DIM

Creates an array variable or a block of memory and defines its size.
Examples:

```
DIM month%(12)
DIM a%(5),b%(7,2),name$(6)
DIM code% 100
```

The first example defines an array with 13 elements, called month%(0) to month%(12). The second example defines three arrays, the second of which is two-dimensional, having 24 elements, numbered b%(0,0) to b%(7,2).

The last example defines a block of memory 100 bytes in size, which may be used to store data or for the assembly of machine code. The value of the variable code% is the address of the first byte in the block.

DIV

Performs whole number division, ignoring any remainder.
Example:

```
leap%=year% DIV 4
```

The variable year% is divided by 4, any fraction being rounded downwards, thus 15 DIV 4 gives 3 and 16 DIV 4 gives 4.

See also MOD.

DRAW

A command to draw a line from the last position of the graphics cursor to the coordinates given, which then become the new position of the graphics cursor.
Examples:

```
DRAW 800,400
DRAW BY 150,100
```

The first example draws a line from the previous graphics cursor position to (800,400). The second example includes the keyword BY, which takes the coordinates as being

relative to the graphics cursor, not the graphics origin. If the two lines were together in a program, the second one would draw a line to (950,500).

See also MOVE.

EDIT

Enters the Basic screen editor. This was available with RISC OS 2. It is now usual to write and edit Basic programs using a text editor such as !Edit.

ELLIPSE

Draws an ellipse.
Examples:

```
ELLIPSE xpos%,ypos%,maj%,min%
ELLIPSE FILL xpos%,ypos%,maj%,min%,ang
```

The first example draws an ellipse centred on (xpos%,ypos%), with semi-major axis length maj% and semi-minor axis length min%.

The second example has the keyword FILL, so it draws a filled ellipse. The extra variable on the end of the command means that the ellipse is tilted with its semi-major axis at ang radians to the horizontal.

ELSE

Used with IF ... THEN.
Example:

```
IF x%=2 PROCdo_this:PROCdo_this_too ELSE PROCdo_that
```

If x% equals 2 the program executes the remainder of the line up to the ELSE keyword. If x% does not equal 2, the program jumps to the ELSE keyword and executes the remainder of the line.

ELSE may also be used in multiple-line IF ... THEN structures:

```
IF x%=2 THEN
  PROCdo_this
  PROCdo_this_too
  ELSE
  PROCdo_that
ENDIF
```

ELSE is optional and is only needed if there is something to be executed when the expression following IF is not true.

END

(1) Marks the end of the program. END is not necessary when the program finishes on the last line, but is required when the main part is followed by procedures or functions. These will otherwise be executed as part of the main program, usually resulting in a ‘not in a procedure’ error message when the first ENDPROC is encountered.

(2) Gives the highest address used by the program.
Example:

```
PRINT END
```

ENDCASE

Marks the end of a CASE ... OF structure.
Example:

```
CASE a% OF
  WHEN 1:PROCdo_this
  WHEN 2:PROCdo_that
  WHEN 3:PROCdo_something_else
ENDCASE
```

The value of a% is compared with each of the numbers following the WHEN keywords in turn. When a match is found, the remainder of the line following the colon is executed. The rest of the structure is then ignored and the program jumps to whatever follows ENDCASE.

ENDIF

Marks the end of a multiple-line IF ... THEN structure.
Example:

```
IF x%=y% THEN
  PROCdo_this
  IF a%=2 PROCdo_that
ENDIF
```

Multiple-line IF ... THEN structures are useful as they make it easy to use several IF statements at once. In this example, PROCdo_this is executed if x% equals y%, but PROCdo_that is only executed if x% equals y% and a% also equals 2.

ENDPROC

Marks the end of a procedure and hands control back to the point where the procedure was called.

If the definition of the procedure included any parameters preceded by RETURN, their values are passed back to the corresponding global variables.

See also PROC.

ENDWHILE

Used with WHILE to construct a loop.
Example:

```
WHILE x%<5
  PROCenter(x%)
  x%+=1
ENDWHILE
```

EOF#

Indicates whether or not the last byte in an open file has been reached.
Example:

```
handle%=OPENIN("filename")
REPEAT
  char%=BGET#handle%
  VDU char%
UNTIL EOF#handle%
CLOSE#handle%
```

This program opens a file and sends its contents to the screen, one byte at a time.

The variable handle% is the file handle, assigned to the file by the filing system when it is opened by OPENIN. Each time the REPEAT ... UNTIL loop is executed, one byte is read from the file using the BGET# keyword. EOF#handle% has the value FALSE until the final byte is read, when it changes to TRUE.

The file is closed at the end of the operation by CLOSE#handle%.

EOR

Applies a logical exclusive-OR operation (see Section 9), either to two statements following an IF keyword or to the individual bits of two numbers.
Examples:

```
um%=char% EOR &20
IF x%=2 EOR y%=3 THEN PROCsend
```

The first example inverts the state of bit 5 of char%. This has the effect, with ASCII codes, of making capital letters lower case and vice versa.

In the second example, PROCsend is called if either x% equals 2 or y% equals 3, but not if both cases are true.

See also AND, OR.

ERL

Gives the line number where the last error was detected.
Example:

```
ON ERROR REPORT:PRINT " at line ";ERL
```

This line is useful if you run a program from the desktop while it is under development. Errors in this situation usually result in an error message without a line number, making it more difficult to debug the program.

ERR

Error number of the last error.
Example:

```
ON ERROR IF ERR=17 PRINT "You pressed Escape, so goodbye!":END ELSE REPORT:PRINT
" at line ";ERL:END
```

Pressing Esc is treated as an error − its error number is 17. This line prints a special message if you press Esc, but the usual message for any other errors.

ERROR

(1) Used as part of ON ERROR to handle errors.

(2) Generates an error.
Example:

```
IF number%>5 THEN ERROR 1,"You must enter 5 or less"
```

If number% is greater than 5, the error handler is called. This might be a specially written line beginning with ON ERROR. The ERROR keyword in this example is followed by the error number, then the error message. The error handler can be written to detect this particular error number and act accordingly.

See also LOCAL.

EVAL

Gives the numeric equivalent of the string which follows it in brackets.
Example:

```
INPUT "What do you want me to work out",expr$
PRINT EVAL(expr$)
```

This program invites you to type in any expression that Basic could work out the value of, for example 3+2 or (4+5)/3. If you had already defined, say, a% to be 25, you could enter a%+2. The second line works out the answer and prints it.

See also VAL.

EXP

Gives the exponential of the number following it in brackets.
Example:

```
a=EXP(x)
```

a is e raised to the power of x, where e=2.718281828

EXT#

Gives or alters the length of an open file.
Examples:

```
PRINT EXT#handle%
EXT#handle%=EXT#handle%+8
```

The first example prints the length of the file which had the file handle handle% assigned to it when it was opened by OPENIN, OPENOUT OR OPENUP.

The second example increases the length of the file by eight bytes. The extra bytes are filled with zeros.

FALSE

Gives the value zero.
Example:

```
done%=FALSE
```

A later line in the program may begin:

```
IF done% THEN
```

What follows will not be executed unless done% has in the meantime been changed from FALSE to TRUE.

FILL

(1) Fills a shape when it is drawn.
Example:

```
RECTANGLE FILL 300,200,500,400
```

(2) Flood fills from a given point. Provided the point has the graphics background colour, the area around it is turned to the graphics foreground colour. The effect spreads out in all directions as far as any point which is not in the background colour, the edge of the graphics window or the edge of the screen.
Example:

```
FILL 300,200
```

See also CIRCLE, ELLIPSE.

FN

Calls a function, or used with DEF to define a function.
Examples:

```
a%=FNsize(b%)
DEFFNsize(length%)
```

A function is similar to a procedure but it is used like a variable and ends by producing a value. In the first example, FNsize is called, passing a parameter from variable b%. On return, the number created in the function becomes the value of variable a%.

FOR

Part of a FOR ... NEXT loop.
Example:

```
FOR n%=0 TO 5
  PRINT n%
NEXT
```

This loop will print the numbers 0 to 5 in turn. When the loop is first executed, n% is given the value 0 and the program followed up to the NEXT command. Because n% is less than 5, the program returns to the FOR keyword, adds 1 to n% and executes the loop again, adding another 1 to n% each time it reaches NEXT. After the final run, when n% equals 5, the program carries on past the NEXT keyword.

The amount by which n% is increased each time round the loop can be altered by using the keyword STEP.
Examples:

```
FOR n%=0 TO 10 STEP 2
FOR i%=10 TO 1 STEP -1
```

In the first example, n% is increased by 2 each time the loop is executed and in the second, i% is *decreased* by 1 each time.

FOR ... NEXT loops may be *nested*, that is, put inside each other:

```
FOR n%=1 TO 10
  FOR i%=0 TO 5
    PRINT n%,i%
  NEXT i%
NEXT n%
```

It is not necessary to put i% and n% after the two NEXT keywords as the program knows which FOR to go back to, but their inclusion does make the program easier to understand.

GCOL

Sets graphics colours and PLOT actions.
Examples:

```
GCOL 3
GCOL plot%,col%
```

The first example selects colour 3 as the current graphics foreground colour. In a 16-colour mode this would normally be yellow unless colour 3 had been redefined by a COLOUR command. To change the graphics background colour, add 128 to the number. For a yellow background, for example, use GCOL 131.

The effect of a change of graphics background colour is seen when you clear the graphics window with CLG.

The second example has two numbers following GCOL. In this case, the colour is defined by the second number and the first one defines the *PLOT action*, that is how the colour should affect the screen:

plot%	**action**
0	set the screen to colour col%
1	OR the colour on the screen with col%
2	AND the colour on the screen with col%
3	EOR the colour on the screen with col%
4	Invert the colour on the screen
5	Don't change the screen at all
6	AND the colour on the screen with the inverse of col%
7	OR the colour on the screen with the inverse of col%

Add 8 to these colours to plot a sprite with a mask.

See also PLOT.

GET

Gives the ASCII code of a character whose key is pressed.
Examples:

```
char%=GET
REPEAT UNTIL GET=13
```

In each case, the program will wait until it discovers that a key has been pressed. If a key is pressed before the program gets to this point, its ASCII code will be stored in the keyboard buffer and GET will read it immediately.

The second example makes the program wait until the Return key (ASCII code 13) has been pressed. If you want it to wait until you press *any* key, just use:

```
REPEAT UNTIL GET
```

To prevent the risk of a key being pressed in advance and stored in the buffer, you could precede this line with:

```
*FX15
```

which flushes (i.e. empties) all buffers.

See also INKEY, INKEY$.

GET$

Identical to GET, except that it gives a one-character string containing the character typed.

GET$#

Gets a string from an open file.
Example:

```
name$=GET$#handle%
```

The variable handle% is the file handle, which is a number given to the file when it is opened, using either OPENIN or OPENUP.

Characters are added to the string from the file, starting at the file pointer PTR# until a Line Feed, Return or zero is reached, the end of the file is encountered or the maximum string length of 255 characters is reached. The value of PTR# is increased so that it indicates the next byte in the file following the end of the string.

GOSUB

Calls a subroutine by its line number.
Example:

```
GOSUB 2000
```

This example causes the program to jump to line 2000 and resume execution there until the command RETURN is reached. Control is then returned to the point immediately after the GOSUB command.

This command is provided mainly for compatibility with other versions of Basic. It is much better to use a procedure than a subroutine. A procedure is called by a name which, if well chosen, will give some indication of what it does. This makes it easier to follow the program, both at the point where the procedure is called and where its beginning is found in the listing.

Another important feature of procedures is the fact that you can pass variables to them through their *parameters*, which you cannot do with subroutines.

A program may easily be broken down into smaller units through the use of procedures, which can make it easier to understand. This is the essence of *structured programming*. A program broken down into subroutines, identified only by their line numbers, would be harder to follow.

GOTO

Instructs the program to jump to another line.
Example:

```
GOTO 1000
```

Like GOSUB, this command is provided mainly for compatibility with other forms of Basic and its use is not generally recommended. It might be used to produce a continuous loop, jumping back to an earlier line, or to miss out some lines under certain conditions. The first case is better dealt with by using a REPEAT ... UNTIL loop and the second by an IF ... THEN ... ENDIF structure, as these show more clearly what is happening.

HELP

Shows information about Basic or the current program.
Examples:

```
HELP
HELP A
HELP .
HELP PRINT
```

The first example will tell you when your copy of Basic was assembled, how much memory the program uses and how much is left over.

The second example shows a list of all the keywords beginning with ‘A’, or any other letter, while the third lists all the Basic keywords. Note that these lists are not strictly in alphabetical order. They have been rearranged slightly to improve minimum abbreviations, as these are checked against the list in order from the beginning. HELP A, for example, shows:

AND ABS ACS ADVAL ASC ASN ATN AUTO APPEND

Because AND comes at the beginning of the list, it is the keyword whose minimum abbreviation is ‘A.’. This is convenient because you are much more likely to type AND than ABS, ASC etc.

The fourth example gives you information about a particular keyword, in this case PRINT.

HIMEM

The address of the highest byte in memory available to Basic.
Examples:

```
PRINT HIMEM
HIMEM=&A0000
```

IF

Enables conditional execution of statements or lines.
Examples:

```
IF a%=b% THEN x%=2:PROCdo_this ELSE PROCdo_that
```

In most cases, THEN is not absolutely necessary, though its presence makes the line easier to understand.

The expression after the IF keyword is tested to see if it is true, i.e. if a% does equal b%. If so, the remainder of the line up to ELSE is executed, otherwise the part of the line following ELSE is executed. In other words, if a% equals b%, then x% is given the value 2 and PROCdo_this is called. If a% doesn’t equal b%, then PROCdo_that is called.

If there is nothing to be done if a% doesn’t equal b%, the ELSE keyword is omitted.

Conditional execution may be applied over several lines:

```
IF a%=b% THEN
  IF y%=4 x%=2
  PROCdo_this
  ELSE
  PROCdo_that
ENDIF
```

The line beginning with IF has nothing following the THEN keyword, which indicates that conditional execution applies to the following lines, down to the ENDIF keyword.

Multiple line IF ... THEN structures are useful, partly because they allow a large chunk of programming to be conditionally executed and also because extra conditions can be applied to some lines. In this example, x% is only set equal to 2 if a% equals b% and y% also equals 4, but PROCdo_this is always called if a% equals b%.

Again, ELSE is only used if there is something to be done when the expression following the IF keyword is not true.

INKEY

Gets an ASCII code for a character if a key is pressed.
Example:

```
char%=INKEY(100)
```

This keyword differs from GET in that it doesn't hold up the program until you press a key, but waits only up to a maximum time limit depending on the number in brackets, which is in centi-seconds. This example waits up to 1 second (100 centi-seconds). If a key is pressed during this time (or if a keypress has been stored in the keyboard buffer), the value of char% becomes the ASCII code of the key, otherwise char% becomes −1.

This is a useful way of building a time delay into a program, particularly if it can be cut short by pressing a key. You could, for example, display a short message on the screen with an instruction to 'press any key' when it has been read, but decide that 10 seconds is ample time to read it and the program should proceed after that time anyway. Use the following commands:

```
*FX15
x%=INKEY(1000)
```

The first line flushes all the buffers to ensure that there isn't already a keypress in the keyboard buffer and the second line waits for up to 10 seconds. If a key is pressed in this time, the program proceeds immediately. The variable x% is given the ASCII code for the key pressed, or −1 if no key was pressed. If it doesn't matter which key was pressed, x% will be a dummy variable which is not used again.

If INKEY is followed by a negative number, the operating system checks to see if a particular key is pressed and returns TRUE or FALSE, depending on whether or not it's pressed. All keys have negative INKEY numbers, including the Shift, Ctrl and Alt keys and the mouse buttons, but a list of them is outside the scope of this guide.

INKEY$

Behaves in the same way as INKEY, except that it gives a one-character string containing the character whose key was pressed.

Where INKEY would give −1, INKEY$ gives a null string, i.e. a string with no characters.

INPUT

Reads in a number or string from the keyboard and assigns it to a variable.
Examples:

```
INPUT length%
INPUT name$
INPUT "What is your name",name$
INPUT a%,b%,c%,d%
```


The first two examples cause a question mark to be displayed with the cursor to the right of it. The third one displays the text in quotes. If the text is followed by a comma, a question mark follows it on the screen, otherwise it is left out.

The fourth example gives four question marks, prompting for four variables.

If you type in a string containing leading spaces or commas, INPUT will ignore the leading spaces and first comma and everything after it. If you want to be able to enter a string which may contain these, use INPUT LINE instead of INPUT.

See also TAB.

INPUT#

Gets information from an open file. An open file is one which has been opened with the command OPENIN, OPENUP or OPENOUT.
Example:

```
INPUT#handle%,title$,length%
```

The variable handle% is the *file handle*, which is a number given to the file when it is opened, using OPENIN, OPENUP or OPENOUT.

The command in this example expects to receive a string from the file, followed by a number.

See also PRINT#

INSTALL

Loads a Basic file containing a library of functions or procedures into memory.
Example:

```
INSTALL "0.$.Procedures"
```

The file is loaded into the top of available memory and HIMEM is reduced to below the point where it starts. If a procedure is called which isn't in the main program, Basic looks for it in the library.

The installed library remains in memory until you QUIT from Basic.

Library files must not contain references to line numbers such as GOTO or GOSUB. Basic's DATA pointer may be reset by a command such as RESTORE+1. It is safest to use only local variables, used within the function or procedure in question.

A library file should include a description of what it contains in the first line, as this will be listed by the LVAR command.
Example:

```
10 REM > Lib_file1 - Contains graphics routines
```

See also LIBRARY, OVERLAY.

INSTR

Finds the position of one string within another.
Example:

```
a$="Hello mum"
b$="lo"
PRINT INSTR(a$,b$)
4
```

The program looks for the string ‘lo’ within the longer string ‘Hello mum’ and finds it, starting with the fourth letter, so it gives the answer 4. If the second string isn’t present within the first one, INSTR gives a value of zero.

INT

Gives the integer, or whole number, part of a number.
Example:

```
PRINT INT(4.5)
4
```

The number is rounded down to the nearest whole number equal to or below it.

LEFT$

Gives or alters the left-hand part of a string.
Examples:

```
a$=LEFT$(name$,3)
b$=LEFT$(name$)
LEFT$(name$,3)="Abc"
LEFT$(name$)=c$
```

In the first example, a$ is set to the first three characters of name$. The second example doesn’t have a second number or variable in brackets and b$ is set to all of name$ except for the final character.

In the third example, the first three characters of name$ are replaced by "Abc", while in the fourth example, however many characters there are in c$ replace the same number of characters at the beginning of name$.

See also MID$, RIGHT$.

LEN

Gives the length of a string, i.e. the number of characters it contains.
Example:

```
a$="ABCDE"
PRINT LENa$
5
```


LET

Optional keyword used when assigning a value to a variable, included mainly for compatibility with other versions of Basic.
Example:

```
LET x%=a%+b%
```

This has exactly the same meaning as:

```
x%=a%+b%
```

LIBRARY

Loads a Basic file containing functions or procedures into memory, allowing its contents to be called from the main program.
Example:

```
LIBRARY "0.$.Procs"
```

There are two differences between LIBRARY and INSTALL:

- INSTALL cannot be included in a program but must be called from immediate mode. LIBRARY can be included in a program listing.
- INSTALL loads a file into a part of the memory not used by the current program and it stays there until you exit from Basic with QUIT. LIBRARY loads the file into a part of the memory used by the program. If you type NEW or load a new program, the library file will disappear, along with the old program.

See also OVERLAY

LINE

(1) Draws a line between two points.
Example:

```
LINE 300,200,600,800
```

This example draws a line between coordinates (300,200) and (600,800). The graphics cursor position ends up at (600,800)

(2) Used with INPUT to form LINE INPUT and INPUT LINE, both of which have the same meaning.

This combination of keywords allows you to type in a string containing leading spaces and commas. If you use INPUT on its own, leading spaces, the first comma and everything after it will be ignored.

LIST

Displays a listing of the program.
Examples:

```
LIST
LIST 100,600
LIST IFname$
```

The first example lists the entire program. If the program is long, it will fly up the screen too fast for you to read. You can slow it down by holding down Ctrl. Better still, you can put the screen into ‘page’ mode by pressing Ctrl-N before you type LIST. This will stop the screen scrolling by more than the number of lines that can be displayed at once.

To see some more of the program, press Shift (it’s easiest if you hold down Ctrl as you do so, otherwise you may allow some lines to scroll off the top of the screen before you release Shift).

To stop a listing before you reach the end, press Esc. To cancel page mode, press Ctrl-O.

The second example lists part of the program, in this case from lines 100 to 600 inclusive.

The third example is a conditional listing and shows all lines containing the string following IF, in this case all those containing a reference to the variable name$. Note that there isn’t a space between IF and name$. If you include a space, it will be counted as part of the string and a line will only be listed if it contains ‘name$’, preceded by a space.

LISTO

Controls the way the listing appears on the screen.
Example:

```
LISTO 3
```

The appearance of the listing depends on the setting of the various bits of the number following LISTO:

Bit	Purpose
0	Puts a space after the line number
1	Indents FOR ... NEXT, IF ... THEN ... ENDIF and other structures
2	Splits lines at colons
3	Does not list line numbers
4	Prints keywords in lower case

The number 3 in the example has bits 0 and 1 set. This means that a space is inserted after the line number and structures are indented.

If you type in a program when LISTO is not zero, any leading spaces on the lines will be stripped off.

LN

Gives the natural logarithm of the number or variable following it in brackets, that is, the logarithm to the base ‘e’ (2.718281828).
Example:

```
PRINT LN(size)
```

See also LOG.

LOAD

Loads a program into memory at the current setting of PAGE.
Example:

```
LOAD "0.$.MyProg"
```

Any existing program in memory is discarded, along with its variables. The Resident Integer Variables (A% − Z%) are not affected.

See also CHAIN, RUN.

LOCAL

(1) Creates local variables in a procedure or function.
Example:

```
DEFPROCsize
LOCAL length%,width%,height%
```

The variables length%, width% and height% are newly created and exist only within the procedure or function. The values of any other variables with the same names outside the procedure or function (known as *global variables*) are not affected by what happens inside it.

When local variables are created, they are given a value of zero in the case of numeric variables, and the null string in the case of strings.

(2) Used with ERROR to produce LOCAL ERROR.

When you define an error handler, using ON ERROR ... , any previous error handler is normally forgotten by Basic. You can prevent this happening by preceding the new error handler definition with LOCAL ERROR. Basic remembers the previous error handler and will revert to it if you use the command RESTORE ERROR.

You need LOCAL ERROR if you define a local error handler within a function or procedure using ON ERROR LOCAL (see ON).
Example:

```
DEFFNopen_file(filename$)
LOCAL ERROR
ON ERROR LOCAL REPORT:=0
handle%=OPENIN(filename$)
=handle%
```

This function tries to open a file whose filename is passed to it as a parameter and ends by returning the file handle to the main program. If it fails to open the file, the error doesn't stop the program; it simply prints the error message (the REPORT command) and returns to the main program, giving a value of zero. Because LOCAL ERROR was used within a function or procedure, it isn't necessary to use RESTORE ERROR. This happens automatically at the end of the function.

LOG

Gives the logarithm to the base 10 of the number or variable following it in brackets.
Example:

```
PRINT LOG(size)
```

See also LN.

LOMEM

The address in memory where storage of the program's variables starts.
Examples:

```
PRINT LOMEM
LOMEM=TOP+&800
```

The second example sets LOMEM to be &800 (2048) bytes above the top of the program, reserving the memory in between for some special purpose. You can only do this in a program **before** you create any variables. LOMEM is reset to TOP when you type RUN.

LVAR

Lists all the variables used in a program, procedures and functions used and the first line of any libraries loaded.

Note: Apart from the Resident Integer Variables (A% − Z%), which are always present, this information only shows what has actually been encountered while running the program.

MID$

Gives or alters the middle portion of a string.

Examples:

```
a$=MID$(name$,2,3)
b$=MID$(name$,2)
MID$(name$,2,3)="Abc"
MID$(name$,2)=c$
```

In the first example, a$ is set to a string consisting of three characters within name$, beginning with the second one. The second example doesn't have a third number or variable in brackets and b$ is set to all of name$ from the second character onwards..

In the third example, the three characters of name$ beginning with the second one are replaced by "Abc", while in the fourth example, however many characters there are in c$ replace the same number of characters in name$, beginning with the second one.

See also LEFT$, RIGHT$.

MOD

Gives a remainder, following whole number division.
Example:

```
PRINT 17 MOD 7
3
```

Whole number division of 17 by 7 gives 2, with a remainder of 3.

See also DIV

MODE

Sets the screen mode or gives the current mode.
Examples:

```
MODE 12
PRINT MODE
```

The first example changes screen mode to mode 12. This resets everything connected with the VDU to its default settings. The screen is cleared to black, the text cursor is set to the top left-hand corner and the graphics cursor to the bottom left-hand corner (0,0) The text and graphics windows are both set to cover the full screen.

The second example prints the current screen mode.

MOUSE

Gives the current mouse position and the state of its buttons, turns the pointer on and off and sets various other parameters connected with the mouse.
Examples:

```
MOUSE xpos%,ypos%,buttons%
MOUSE ON
MOUSE OFF
```

```
MOUSE COLOUR 1,255,0,0
MOUSE STEP 4,1
MOUSE RECTANGLE 200,200,1000,800
```

The first example assigns the x and y coordinates of the mouse pointer to variables xpos% and ypos%. The bits of buttons% are determined by the state of the buttons:

Bit	Purpose
0	Right-hand button pressed
1	Middle button pressed
2	Left-hand button pressed

If no button is pressed, buttons% will be zero. If the left-hand (Select) button is pressed, buttons% will have a value of 4.

The second and third examples turn the pointer on and off and the fourth sets its colour. There are three colours available to the mouse pointer − the default pointer uses two of them, 1 for the outer part of the arrow and 2 for the inner section.

This example sets the outer part of the pointer to bright red.

The fifth example sets the speed at which the pointer will move across the screen when the mouse is moved. In this example, the pointer will move rapidly (speed 4) horizontally, but slowly (speed 1) vertically. If MOUSE STEP is followed by one number, it is applied to both directions.

The last example defines a rectangle which the pointer may not go outside.

MOVE

Moves the graphics cursor to a new position without drawing a line on the way.
Example:

```
MOVE 600,500
```

This example moves the graphics cursor to a point roughly in the middle of the screen. If it is followed by a command such as DRAW 800,700, the result will be a line from (600,500) to (800,700).

If MOVE is followed by BY, the coordinates given are taken to be relative to the previous graphics cursor position.

NEW

Effectively removes the current Basic program from memory.

This command enables you to start entering a new program without having to delete the lines of the old one, but it doesn't actually remove it from memory. If you type NEW, then change your mind **before you start entering a new program**, you can recover what was there by typing OLD.

NEXT

Part of a FOR ... NEXT loop.
Examples:

```
FOR n%=0 TO 3
  PRINT name$(n%)
NEXT n%
FOR n%=0 TO 4

  FOR i%=0 TO 3
    READ slot%(n%,i%)
NEXT i%,n%
```

The first example is a simple loop in which n% is increased by 1 each time the loop is executed. The variable name after NEXT is usually omitted, but its presence makes the program a little clearer to follow and will also produce an error message if you have several loops inside each other (they are said to be *nested*) and their NEXT commands come up in the wrong order.

The second example shows how it's possible to have two or more nested FOR ... NEXT loops. The inner loop is executed four times for each time round the outer loop. These two loops could end with two NEXT commands, but the last line in this example ends both loops at the same time.

See also STEP.

NOT

Changes the state of a variable from TRUE to FALSE or vice versa, or inverts all the bits of a number.
Examples:

```
IF NOT done% THEN PRINT "Another go?"
x%=NOT y%
```

The first example is the same as typing:

```
IF done%=FALSE THEN PRINT "Another go?"
```

The program will only execute the PRINT command if the expression that follows the IF keyword is TRUE. If done% is FALSE, 'NOT done%' is TRUE. Similarly, if done% is FALSE, the expression 'done%=FALSE' is true, because the parts of the expression on each side of the equals sign are the same.

In the second example, each bit of x% is given the opposite state to the corresponding bit of y%. If y% is zero, x% is −1 and if y% equals 1, x% is −2.

The reason for this apparently strange state of affairs is that zero in binary consists of 32 noughts:

0000 0000 0000 0000 0000 0000 0000 0000

NOT zero, therefore, is:

1111 1111 1111 1111 1111 1111 1111 1111

or &FFFFFFFF in hexadecimal, which is usually taken to be −1 (see Section 9).

Similarly, 1 is:

0000 0000 0000 0000 0000 0000 0000 0001

so NOT 1 is:

1111 1111 1111 1111 1111 1111 1111 1110

or &FFFFFFFE, which is −2.

OF

Used as part of CASE ... OF ... ENDCASE.

See CASE.

OFF

Turns off the text cursor, or used as part of other commands.
Examples:

```
OFF
TRACE OFF
SOUND OFF
MOUSE OFF
ON ERROR OFF
```

The first example turns off the text cursor. You can turn it on again by typing ON. It will also reappear if you change screen mode of perform cursor editing.

The following three examples turn off tracing of the current program (see TRACE), all sound output (use SOUND ON to turn it on again) and the mouse pointer.

The final example disables the program's own error handler and reverts to Basic's own way of dealing with errors. If you write a program with a long and complicated error handler routine, it might be advisable to insert ON ERROR OFF while testing it, in case the error handler itself contains an error, which could put you in an endless loop with the error handler calling itself!

OLD

Recovers a program if you type NEW, then change your mind.

This command will only work provided you have not entered any new program lines or created any variables in immediate mode, as these will start to overwrite your old program.

ON

Turns on various functions, provides conditional branching or sets up an error handler.
Examples:

```
ON
SOUND ON
MOUSE ON
ON num% GOTO 1000,2000,3000 ELSE 4000
ON num% PROCthis,PROCthat,PROCother ELSE PROCsomething_else
ON ERROR CLOSE#0:REPORT:PRINT " at line ";ERL:END
ON ERROR LOCAL PRINT "You can't do that";ENDPROC
```

The first three examples turn on the text cursor, sound output and mouse pointer respectively.

The third example examines the value of the variable num% following ON. In this case, if num% equals 1, the program jumps to line 1000; if it is 2, it jumps to line 2000 and so on. If num% is less than 1 or greater than the number of line numbers following ON, the program jumps to line 4000.

If you replace the GOTO with a GOSUB, the program will return to the line following the ON command when it encounters a RETURN.

The fourth example works in the same way as the third one, only by calling procedures. This is preferable to jumping to line numbers as it makes the program more structured and easier to follow.

You may find it more convenient to use a CASE ... OF ... ENDCASE structure to do this job.

Example 6 sets up an error handler routine. This one is similar to Basic's own error handler, except that it closes any files that may be open. If a program opens a file with OPENIN, OPENUP OR OPENOUT, it is advisable to include a CLOSE command in the error handler, otherwise a file may be left open if an error occurs.

If an error occurs within a loop, or a function or procedure, Basic jumps to the error handler forgetting about the structure where the error occurred. To remain within the procedure or loop after the error handler has done its work, use ON ERROR LOCAL, as shown in the final example.

Any error that follows this command makes the program jump to this error handler without forgetting that it's in a procedure. In order that the main error handler can take over again after the program leaves the procedure, ON ERROR LOCAL should be preceded by LOCAL ERROR (see LOCAL).

OPENIN

Opens an existing file for read-only access.
Example:

```
handle%=OPENIN("0.$.MyFile")
```

The filename, or the pathname as shown in this example, is only used when opening the file. Once the file is open, it is always referred to by its file handle, which is the number given to variable handle%. Keywords such as BGET#, EXT# and EOF# which operate on files refer to the file by its handle.

See also CLOSE#.

OPENOUT

Opens a new file which is initially zero bytes long. If a file with the same filename already exists, it is deleted and a new one created. Once the file has something in it, it can be read as well as written to.
Example:

```
handle%=OPENOUT("0.$.MyFile")
```

The filename, or the pathname as shown in this example, is only used when opening the file. Once the file is open, it is always referred to by its *file handle*, which is the number given to variable handle%. Keywords such as BGET#, EXT# and EOF# which operate on files refer to the file by its handle.

See also CLOSE#.

OPENUP

Opens an existing file for read or write access.
Example:

```
handle%=OPENUP("0.$.MyFile")
```

The filename, or the pathname as shown in this example, is only used when opening the file. Once the file is open, it is always referred to by its *file handle*, which is the number given to variable handle%. Keywords such as BGET#, EXT# and EOF# which operate on files refer to the file by its handle.

See also CLOSE#.

OR

Applies a logical OR operation (see Section 9), either to multiple statements following an IF keyword or to the individual bits of two numbers.
Examples:

```
char%=num% OR &30
IF x%=2 OR y%=3 OR z%=4 THEN PROCdo_this
```

In the first example, any bit in either num% or &30 which is a 1 forces the corresponding bit in char% to be a 1. If num% is 5, char% will be &35.

This example derives an ASCII code from a number. If num% is between 0 and 9, char% will be its ASCII code. See Sections 8 and 9 for a full description of ASCII codes.

In the second example, PROCdo_this is only executed if x% equals 2 *and* y% equals 3 *and* z% equals 4.

See also AND, OR.

ORIGIN

Moves the graphics origin.
Example:

```
ORIGIN 600,500
```

This example moves the graphics origin to a point close to the middle of the screen. All MOVE, DRAW, RECTANGLE etc. commands work with coordinates relative to this point.

Note, though, that the graphics cursor position is not moved by the ORIGIN command. If, for instance, you type:

```
MOVE 50,50
ORIGIN 600,500
DRAW 100,100
```

a line will be drawn from the *old* (50,50), close to the bottom left-hand corner of the screen, to the *new* (100,100), a little above and to the right of the centre.

OSCLI

Generates a string and passes it to the Command Line Interpreter (CLI).

This is the part of the operating system that deals with star commands. Because a star command is passed directly to the CLI without being handled by Basic, you can't include references to any Basic variables in it as the CLI will not understand them.
Example:

```
OSCLI ("Load "+fname$+" "+STR$~buf%)
```

It is possible to load a file into memory with a command such as:

```
*Load MyFile A000
```

where A000 is the address in hexadecimal notation of the first byte of the buffer, i.e. the part of the memory where you want the file to go. Well-written programs are independent of absolute memory addresses of this sort and make no reference to them. Instead, a block of memory to be used as a file buffer can be defined using the DIM statement and the program will only know the address as the name of a variable.

In this example, we're also using a string variable to contain the filename. We can't type the filename directly into a star command when we write the program because we may not know it. The program might ask you to enter it, perhaps using the INPUT command, and then load the appropriate file.

This example creates a string within the brackets following the OSCLI keyword. The first part consists of the command 'Load' in quotes with a space on the end, followed by the filename, then another space. The next part creates the address. The expression STR$~buf% creates a string containing the value of num% and the tilde (~) character puts it in hexadecimal form. If the filename in name$ is '0.$.MyFile' and the value of buf% is, say, &A456, the entire string sent to the Command Line Interpreter will be:

```
Load 0.$.MyFile A456
```

OTHERWISE

Used as part of a CASE ... OF structure.
Example:

```
CASE a% OF
  WHEN 1:PROCdo_this
  WHEN 2:PROCdo_that
  WHEN 3:PROCdo_also
  OTHERWISE
  PROCdo_something_else
ENDCASE
```

If a% is 1, 2 or 3, the appropriate procedure is executed. In all other cases, the program executes PROCdo_something_else.

The use of OTHERWISE is optional. You only need it when there is something to be done if none of the 'WHEN' conditions are met.

OVERLAY

Sets an array of filenames for overlay function and procedure libraries.
Example:

```
DIM lib_file$(3)
lib_file$()="filename1","filename2","filename3","filename4"
OVERLAY lib_file$()
```


The second line of the example puts one filename into each element of array lib_file$(). Each file contains a directory of functions and procedures, as used by the INSTALL and LIBRARY commands.

When Basic encounters the OVERLAY keyword, it checks the sizes of all the files in the array and reserves enough memory to hold the largest. If a call is encountered to a function or procedure not in the program, Basic first checks any libraries loaded with INSTALL or LIBRARY. If it cannot find what it is looking for, it goes through each file in the array in turn, loading the appropriate file when it finds it.

If there is a subsequent call to a function or procedure in another file, that file is loaded in place of the original.

PAGE

The address in memory where a stored Basic program starts.
Examples:

```
PRINT ~PAGE
PAGE=&C000
```

By using the second example, it is possible to have more than one program stored in memory at the same time. You can change the value of PAGE and load a new program from disc. To switch between programs, set PAGE to the appropriate value and type OLD.

PAGE must be *word-aligned*, that is, divisible by 4.

PI

Basic's way of writing the Greek letter π, used to describe the ratio between the diameter and circumference of a circle.
Example:

```
PRINT PI
3.141592653
```

PLOT

Moves the graphics cursor or draw on the screen in a particular way, determined by the PLOT code.

See Appendix 3 for a full list of PLOT codes.
Example:

```
PLOT 85,400,300
```

This example draws a filled triangle between (400,300) and the previous two positions of the graphics cursor.

POINT

(1) Plots a single point on the screen.
Example:

```
POINT 900,800
```

This will plot a point at (900,800) and move the graphics cursor to that position.

If POINT is followed by BY, the coordinates are taken to be relative to the last position of the graphics cursor.

(2) Gets the logical colour of a pixel.
Example:

```
col%=POINT(500,400)
```

The value of col% will be the logical colour of the pixel at (500,400). If the screen is in a 256-colour mode, this will be a number between 0 and 63, describing the colour but not the tint. The tint number can be checked by typing:

```
tint%=TINT(500,400)
```

POS

Gives the horizontal position of the text cursor.
Example:

```
x%=POS
```

The value of x% will be the number of columns in from the left-hand side of the screen (or text window, if set) the text cursor is located.

See also VPOS.

PRINT

Displays on the screen.
Example:

```
PRINT "The answer is ";ans%
```

The first part of the remainder of the line following PRINT is in quotes and appears on the screen exactly as shown. The value of variable ans% is then added. If the semi-colon (;) were not present, this number would be *right-justified*, meaning that spaces would be added before it, making a total of 10 spaces and digits (excluding the space at the end of the string in brackets). The presence of the semi-colon means that this does not happen.

If ans% were preceded by a tilde (~), its value would be shown in hexadecimal notation.

To print several right-justified variables, separate them with commas.

An apostrophe (') will cause a new line to be started.

See also TAB.

PRINT#

Sends information to an open file. An open file is one which has been opened with the command OPENUP or OPENOUT.
Example:

```
PRINT#handle%,title$,page_no%
```

The variable handle% is the *file handle*, which is a number given to the file when it is opened, using either OPENUP or OPENOUT.

The command in this example sends a string to the file, followed by a number. Both are stored in a particular way and may be read back in order using INPUT#.

PROC

Used in calling and defining a procedure.
Examples:

```
IF x%=2 PROCsize(length%)
DEFPROCsize(l%)
```

In the first example, PROCsize is called if x% equals 2. The variable length% is passed to it as a parameter.

The second example is located at the start of the definition of the procedure. In this case, the procedure is defined as having one parameter, referred to as l%. Within the procedure, l% is treated as a *local variable*, which makes it totally separate from any other variable with the same name in the main program (a *global variable*).

There must not be a space between PROC and the procedure name in either of these uses, but you can put a space between DEF and PROC.

See also ENDPROC.

PTR#

Gives the position of the pointer of an open file. An open file is one which has been opened with the command OPENIN, OPENOUT or OPENUP.

The pointer is used by commands such as BGET# and BPUT# which read from or write to a file and determines the position in the file where the operation is performed. In this way, random access to a file is possible.
Example:

```
PTR#handle%=10
```

The variable handle% is the *file handle*, which is a number given to the file when it is opened, using either OPENIN, OPENOUT or OPENUP.

This example moves the pointer to a position 10 bytes in from the start of the file.

QUIT

Shuts down Basic.

When this happens, you will be left with the star prompt, which simply allows you to enter star commands. If you had originally started Basic by running a file from the desktop, pressing Return will take you back to the desktop in the state it was when you left it, otherwise type:

```
*Desktop
```

to start up the desktop, or:

```
*Basic
```

to restart Basic.

RAD

Converts an angular measurement in degrees into radians.
Example:

```
twist=RAD(angle)
```

If angle is the size of an angle in degrees, twist is its size in radians.

One radian is approximately 57 degrees. There are 2*PI radians in a circle. Trigonometric functions, such as SIN, COS and TAN work with angles expressed in radians.

See also DEG.

READ

Reads information from a DATA statement.
Example:

```
READ length%,width%
.......
DATA 5,7
```

When the first READ command is encountered, Basic looks for the first DATA keyword in the program (unless it has been told to look elsewhere by RESTORE). Each variable following READ is given a value taken from the number(s) or string(s) following DATA. At the end of the READ operation, Basic remembers where it took the last DATA from (using its *Data Pointer*) and takes more DATA from the point following it, if it encounters any more READ commands.

RECTANGLE

Draws a rectangle on the screen, copies or moves a rectangular portion of the screen to another position or limits the movement of the mouse pointer.
Examples:

```
RECTANGLE 300,200,500,400
RECTANGLE 100,100,400,300 TO 600,500
RECTANGLE FILL 100,100,400,300 TO 600,500
MOUSE RECTANGLE 200,200,1000,800
```

The first example draws a rectangle whose bottom left-hand corner is at (300,200) and which is 500 OS units wide and 400 units high. If RECTANGLE is followed by FILL, the rectangle is filled with the graphics foreground colour.

RECTANGLE leaves the graphics cursor in the bottom left-hand corner but RECTANGLE FILL leaves it in the top right-hand corner.

The second example copies a rectangular section of the screen. The coordinates are defined as in the first example, with the bottom left-hand corner at (100,100), the width 400 units and the height 300 units. The bottom left-hand corner of the copy is at 600,500.

The third example is the same as the second except that the original rectangle is cleared to the graphics background colour, in other words, RECTANGLE ... TO copies a rectangle, RECTANGLE FILL ... TO moves it.

The final example limits the movement of the mouse pointer to a rectangle whose bottom left-hand corner is at (200,200) and whose top right-hand corner is at (1000,800).

REM

Short for REMark. The remainder of the line is ignored by Basic.
Example:

```
REM This is a comment to explain the program
```

This enables you to put comments in the program to explain its workings.

The exception is a REM in the first line of the program which contains an embedded filename.
Example:

```
10 REM > MyFile
```

The REM keyword must be in the first line and be followed by a ‘greater than’ symbol (>), then by the filename. It is then only necessary to type SAVE to save the program.

RENUMBER

Renumbers the program lines.
Examples:

```
RENUMBER
RENUMBER 500
RENUMBER 600,5
```

The first example simply renumbers the program with line numbers starting at 10 and increasing in tens. The second example begins at 500, also increasing in tens, while the final one starts at 600 and increases in fives.

Basic checks for GOTOs, GOSUBs and any other references to line numbers and changes them accordingly.

REPEAT

Used to begin a REPEAT ... UNTIL loop.
Example:

```
REPEAT
char%=GET
UNTIL char%=13
```

This loop waits until you press Return by checking to see if the ASCII code produced by GET equals 13. If what follows UNTIL is not TRUE, the program jumps back to REPEAT.

By using UNTIL FALSE, a loop can be made to repeat indefinitely until an error occurs or Esc is pressed.

Several REPEAT ... UNTIL loops can be *nested*, that is, put inside each other.

REPORT

Displays the error message connected with the last error encountered.
Example:

```
ON ERROR REPORT:PRINT "at line ";ERL:END
```

REPORT starts on a new line. If you want more flexibility than this, you can use REPORT$ instead. This is a string variable containing the error message which can be used in the same way as any other string variable, for example:

```
ON ERROR PRINT "You had a " REPORT$ " at line ";ERL:END
```

RESTORE

Sets the DATA pointer or retrieves the previous error handler.
Examples:

```
RESTORE
RESTORE 500
RESTORE +1
RESTORE ERROR
```


The DATA pointer marks the point in the program that Basic takes data from in READ operations.

The first example resets the pointer to the first DATA statement in the program and the second sets it to the DATA statement on line 500. This operation can be made independent of line numbering by using the third example which sets the pointer to the first DATA statement following the line on which the RESTORE command is given.

The final example is used in conjunction with a LOCAL ERROR command. This command remembers the current error handler so that it isn't forgotten when a new error handler is set up, using ON ERROR ... .

RESTORE ERROR cancels the new error handler and restores the previous one.

If you use LOCAL ERROR within a procedure and set up an error handler with ON ERROR LOCAL, you will not need RESTORE ERROR at the end of the procedure, as this happens automatically.

RETURN

(1) Marks the end of a subroutine

(2) Indicates two-way value passing in a procedure parameter.
Examples:

```
RETURN
DEFPROCget_coords(RETURN xpos%,RETURN ypos%)
```

A subroutine is called using GOSUB and referred to by the number of its first line (see GOSUB). RETURN, as in the first example, makes the program jump back to the point after the GOSUB command.

The second example shows the first line of a procedure which might be used to get the x and y coordinates of a point. Because it has to return two numbers, we can't use a function for this job. Because the procedure definition has two parameters, xpos% and ypos%, the command that calls the procedure must include two global variables as parameters, for example:

```
PROCget_coords(left_side%, top_side%)
```

Normally, left_side% would pass its value to xpos% and top_side% its value to ypos%, but nothing would be passed back the other way when the procedure hands back to the main program. Because both xpos% and ypos% are preceded by RETURN in the procedure definition, however, their final values when the procedure finishes are passed back to become the new values of left_side% and top_side%.

A variable must usually already exist when you use it as a parameter to call a procedure, otherwise it has no value to pass to the procedure and you will get an 'Unknown or missing variable' error. Because xpos% and ypos% have RETURN in front of them, however, it's not necessary for left_side% and top_side% to already exist when you use them to call the procedure.

RIGHT$

Gives or alters the right-hand part of a string.
Examples:

```
a$=RIGHT$(name$,3)
b$=RIGHT$(name$)
RIGHT$(name$,3)="Abc"
RIGHT$(name$)=c$
```

In the first example, a$ is set to the last three characters of name$. The second example doesn't have a second number or variable in brackets and b$ is set to just the final character of name$.

In the third example, the last three characters of name$ are replaced by 'Abc', while in the fourth example, the characters of c$ replace the same number of characters at the end of name$.

See also LEFT$, MID$.

RND

Generates a pseudo-random number.
Examples:

```
x%=RND
x=RND(1)
xpos%=RND(10)
x%=RND(-2)
```

A pseudo-random number generator is one which actually follows a pattern when generating numbers, but one so complicated that the numbers may be regarded as being random.

The first example produces a number between -2147483648 and 2147483647, or, in hexadecimal notation, between 0 and &FFFFFFFF.

The second example, with 1 in the brackets, generates a number between 0 and 0.999999999.

The third one, with a number greater than 1, generates a whole number, in this case between 1 and 10. If you wanted a random number between, say, 4 and 10, you could get it with:

```
x%=RND(7)+3
```


The random number will lie between 1 and 7 so the total will be between 4 and 10.

The final example, with −2 in brackets, will give −2. This may not seem very random, but it has also reset the random number generator. If you do this more than once, using the same negative number each time, then produce numbers between the same limits each time, you will get the same sequence of 'random' numbers. This can be useful while developing a program.

RUN

Executes the program in memory.

Any existing variables other than the Resident Integer Variables (A% to Z%) are cleared before the program starts.

See also CHAIN, LOAD.

SAVE

Saves the program.
Examples:

```
SAVE
SAVE "MyProg"
SAVE "ADFS::HardDisc4.$.Progs.MyProg"
```

The first example can only be used if the first line of the program contains an embedded filename, for example:

```
10 REM > MyProg
```

The second example will save the program, using the filename 'MyProg' in the Currently Selected Directory. This will normally be the root directory of the disc in your floppy drive unless you have changed it.

The third example specifies the pathname of the file. In this case, the program is saved in a directory called 'Progs' which itself is situated in the root directory of the hard disc.

SGN

Gives the sign (positive or negative) of the value following it in brackets.
Example:

```
sign%=SGN(num%)
```

If num% is positive, sign% will be 1.
If num% is zero, sign% will be 0.
If num% is negative, sign% will be −1.

SIN

Gives the sine of the angle following it in brackets.

Example:

```
height%=length%*SIN(angle)
```

where angle is in radians.

See DEG and RAD for conversion between radians and degrees.

See also ASN.

SOUND

Generates a sound or turns sound output on or off.
Examples:

```
SOUND chan%,vol%,pitch%,dur%
SOUND chan%,vol%,pitch%,dur%,delay%
SOUND ON
SOUND OFF
```

The first two examples generate a sound. Variable chan% is the channel number, vol% the amplitude, pitch% the pitch and dur% the duration. The second example allows for a delay, determined by delay%.

See section 12 for a full description of the SOUND command.

The third and fourth examples turn all sound output on or off.

SPC

Used within a PRINT command to print spaces.
Example:

```
PRINT "Hello";SPC(10);"Mum"
Hello          Mum
```

There are 10 spaces between the two words in the second line.

SQR

Gives the square root of the number following it in brackets.
Example:

```
PRINT SQR(16)
4
```

STEP

Used as part of a FOR ... NEXT structure, or to set the mouse speed.
Examples:

```
FOR n%=0 TO 10 STEP 2
FOR n=1 TO 2 STEP 0.1
FOR n%=10 TO 1 STEP -1
MOUSE STEP 4,1
```


In the first example, n% is increased by 2 each time the FOR ... NEXT loop is executed. Similarly, in the second example, n is increased by 0.1 each time (obviously an integer variable can't be used in this situation).

The third example shows how n% can be made to *decrease* each time round the loop. Notice that the number before TO is higher than the number after.

If STEP is not present, the variable following FOR is increased by 1 each time round the loop.

The last example sets the mouse speed to 4 (fast) for horizontal movement and 1 (slow) for vertical movement. If only one number is present, the same speed is set for both directions.

See also TRACE.

STEREO

Sets the stereo position of a sound channel.
Example:

```
STEREO 1,127
```

The first number is the channel number and the second refers to its position. Extreme left is −127, centre is zero and extreme right is +127. The example, therefore, positions channel 1 on the right-hand side of the stereo image.

STOP

Produces a fatal error which stops the program.

A fatal error is one with error number zero and cannot be dealt with by a program's error handler. It always stops the program, giving the error message 'Stopped at line ... '. It can be written into a program to stop it and permit scanning the state of the variables, for example:

```
IF x%>5 THEN STOP
```

following which, you can check the value of x% or any other variable.

STR$

Produces a string containing the number or value following it.
Example:

```
OSCLI ("Load 0.$.MyFile "+STR$~buf%)
```

This example shows how the value of a Basic variable may be included in a call to the operating system, usually performed by a star command. The command is an

instruction to load a file into a section of memory called a buffer. The address of the first byte is the value of the variable buf%.

The STR$ part of the command produces a string containing the characters that would appear on the screen if you told the machine to print buf%. The tilde (~) character after STR$ means that the string contains the hexadecimal form of the number.

If the value of buf% were, say, &A680, the example would send a command to the Command Line Interpreter saying:

```
Load 0.$.MyFile A680
```

which the CLI would understand as though it had a star on the front. The CLI normally works with hexadecimal numbers.

STRING$

Makes a string by repeating a shorter string.
Examples:

```
PRINT STRING$(10,"AB")
PRINT a$:PRINT STRING$(LENa$,"_")
```

The first example will display:

```
ABABABABABABABABAB
```

The second example will print a string and underline it. After printing a$, the program will start a new line and print underlines (_), the number being determined by LENa$, which is the number of characters in the string (see LEN).

SUM

Adds together all the elements of an array.
Examples:

```
total%=SUM cost%()
PRINT SUM address$()
```

The first example adds up the values of the elements of array cost%(). If, for example, this array had been defined with:

```
DIM cost%(9)
```

it would be able to hold 10 numbers, referred to as cost%(0) to cost%(9). The command would add all 10 numbers together.

The second example involves a string array. In this case, the strings in each element are *concatenated*, that is added end to end to form one long string.

SWAP

Exchange the values of two variables or contents of two arrays.
Examples:

```
SWAP old%,new%
SWAP a%(5),b%(6)
SWAP x%(),y%()
```

In the first example, the values of old% and new% are interchanged. In the second example, the value of element 5 of array a%() is swapped with the value of element 6 of array b%().

In the final example, the entire arrays x% and y% are interchanged. If they had been DIMmed differently, each array would acquire the other's dimensions.

If two numerical values are swapped, one may be integer and the other floating point, though the floating point number, if it includes a fraction, will be rounded down to the nearest whole number. Floating point and integer arrays can't be swapped, and string arrays or variables cannot be swapped with numeric ones.

SYS

Calls a routine in RISC OS through a Software Interrupt (SWI).
Examples:

```
SYS "OS_WriteC",65
SYS "OS_ReadC" TO char%
```

The operating system has a great many system calls, that is routines which can be called through a machine code command called a software interrupt. Parameters are passed to and from these routines by means of some of the ARM processor's registers. There are 16 of these, each of which can store a 32-bit number, though only the first seven, numbered R0 to R6 are used for this purpose.

A SYS command may have the form:

```
SYS "Xxx_XxxXxx",x%,y% z% TO a%,b%,c%;f%
```

The first part, "Xxx_XxxXxx" is the name of the SWI. It is very important to get this exactly right or RISC OS will reject it. It is case sensitive.

The next part, x%,y%,z% consists of three variables, or numbers, whose values are put into registers R0, R1 and R2 before the call to the SWI. When the routine has finished, the contents of R0, R1 and R2 become the values of a%, b% and c% respectively.

If there is another variable preceded by a semicolon (;), in this case f%, its value becomes the state of the processor's flags.

The first example above calls the routine 'OS_WriteC'. This call is the machine code equivalent of VDU − the number in R0 is sent to the screen. In this case, 65 is put into R0 before the call, making this example the equivalent of VDU 65. Because 65 is the ASCII code for 'A', this call will print an 'A' on the screen.

The second example calls the routine 'OS_ReadC', which is the equivalent of GET. When you call this routine it waits for you to press a key, then returns with the ASCII code for the key pressed in register R0. In this example, the number in R0 after the call becomes the value of variable char%. This example, therefore, is the equivalent of char%=GET.

For a full description of all SWIs, refer to the *RISC OS Programmer's Reference Manual*, published on CD-ROM by RISCOS Ltd.

TAB

Moves the text cursor as part of a PRINT or INPUT command.
Examples:

```
PRINT TAB(3) name$
PRINT TAB (20,10) "Do you want another go?"
```

The first example prints the string name$ with its first character three spaces in from the edge of the screen, or the text window if it has been set. The vertical position of the cursor is not altered.

The second example prints the string in quotes 20 spaces in from the left and 10 lines down from the top of the screen or text window. It does this wherever the text cursor may have been positioned beforehand.

TAN

Gives the tangent of the angle following it in brackets.
Example:

```
height%=width%*COS(angle)
```

where angle is in radians.

See DEG and RAD for conversion between radians and degrees.

See also ATN.

TEMPO

Gives or alters the rate of the BEAT counter, used in sound generation.
Examples:

```
rate%=TEMPO
TEMPO &800
```


The first example reads the current value of TEMPO and sets the value of variable rate% to it. The second example sets TEMPO to &800.

The value of TEMPO is in beats per centi-second, with the lowest three hexadecimal digits of TEMPO being a fraction. If TEMPO is &1000, this corresponds to one beat per centi-second, Half this number, &800, corresponds to half this speed and doubling it to &2000 corresponds to two beats per centi-second.

TEXTLOAD

Loads a Basic program, which may be in the form of a text file.
Example:

```
TEXTLOAD "MyProg"
```

When a Basic program is stored in memory or saved as a file, its keywords are *tokenised*. This means that each keyword is represented by a *token*, that is, a single byte containing a number between 128 and 255. If you were to load a Basic file into memory and look at it, using the *Memory command, you would not see the keywords, but strange characters in their places. These are the tokens.

If you had a Basic program that had been saved as a text file, perhaps using TEXTSAVE, you could load it using TEXTLOAD. If the text file does not contain line numbers, it will automatically be renumbered.

TEXTSAVE

Saves a Basic program as a text file.
Examples:

```
TEXTSAVE "MyProg"
TEXTSAVEO 3,"MyProg"
```

In a normal Basic file, each keyword is represented by a *token* (see TEXTLOAD). This means you can't load a Basic program into a word processor to edit it (Edit and other text editors such as Zap and StrongED are different − if you load a Basic filetype into one, it automatically de-tokenises it).

If you wish to use a Basic listing in this way, you can save it as a text file using this command.

The first example simply saves the program in the same way as SAVE (though you can't use an embedded filename). The second example, with an 'O' on the end of the keyword, allows you to apply a LISTO option to the layout of the program (see LISTO). This example saves a text file with a space after the line number and all the structures indented.

THEN

Used as part of IF ... THEN ... ELSE
Examples:

```
IF x%=2 THEN PROCdate
IF size%>6 THEN
```

If the expression following IF is TRUE, whatever follows THEN on the line is executed, up to ELSE, if it is present. If THEN is the last thing on the line, as in the second example, the following lines, down to ENDIF are treated as a multi-line IF ... THEN structure.

THEN is frequently optional. The first example could be written:

```
IF x%=2 PROCdate
```

There are some circumstances where THEN must be used. If it is followed by a *pseudo-variable*, such as TIME, you should use it. It's also needed to indicate a multi-line IF ... THEN structure.

TIME

Reads or alters the time counter.
Examples:

```
TIME=0
t%=TIME:REPEAT UNTIL TIME-t%=200
```

TIME should not be confused with the real time and date clock, which can be accessed through variable TIME$. It is a straight count of centi-seconds and is set to zero when the machine is switched on or reset.

The first example sets TIME to zero. The second one produces a delay of two seconds. Variable t% is set to the current value of TIME and the loop repeats until TIME exceeds this figure by 200. As it increases 100 times per second, this takes two seconds.

TIME$

Sets or reads the real-time clock.
Examples:

```
year$=MID$(TIME$,12,4)
TIME$="xxx, 15 Jan 1993"
```

It is possible to alter the date and time or just the date. The machine automatically works out the day of the week so it's not necessary to enter it as part of the string.

TINT

Used with COLOUR or GCOL in 256 colour modes to set the tint, or to detect the tint of a point on the screen.

Examples:

```
COLOUR 3 TINT 255
GCOL 12 TINT 128
tint%=TINT(xpos%,ypos%)
```

TINT is an 8-bit number, but only the top two bits are used. This means that tint numbers 0 − 63 count as 0, 64 − 127 count as 64, 128 − 191 as 128 and 192 − 255 as 192. The first example sets the text foreground colour to bright red and the second sets the graphics foreground colour to medium green.

In the third example, tint% is set to the tint of the pixel at coordinates (xpos%,ypos%).

TO

Part of the FOR ... NEXT structure. Also sets the mouse position.
Examples:

```
FOR count%=1 TO 10
MOUSE TO 500,400
```

The second example moves the mouse pointer to coordinates (500,400)

TRACE

Prints the line numbers of a program as they are executed.
Examples:

```
TRACE ON
TRACE STEP ON
TRACE PROC
```

The first example turns on tracing. When the program is run, the number of each line will be displayed as it is executed. You can limit this to lines below, say, line 100 by typing:

```
TRACE 100
```

The second example stops the program after each line and waits for you to press a key. Again, you can limit this to lines below a certain number, as above.

Try running 'Boxes' from section 6 with TRACE STEP ON set.

The third example prints the name of each procedure or function as it is called. You can also use TRACE STEP PROC.

To turn off TRACE type:

```
TRACE OFF
```

TRUE

Gives the value −1.

Example:

```
active%=TRUE
IF active% THEN PROCgame
```

If you print out the value of active%, it will be −1. In Basic, this number means TRUE. Try typing:

```
PRINT 3=3
```

The result will be −1. In the second line of the example, variable active% is tested to see if it is TRUE and PROCgame will be executed if it is.

UNTIL

Marks the end of a REPEAT ... UNTIL loop.
Examples:

```
UNTIL n%=10
UNTIL FALSE
```

The first example might come at the end of a loop in which n% is progressively increased (though, if n% were automatically increased by the same amount each time round the loop, it would be better to use a FOR ... NEXT loop). The loop is repeated until the value of n% equals 10.

The program tests the expression which follows UNTIL and jumps back to the REPEAT command unless it is TRUE. The second example repeats indefinitely until an error occurs or Esc is pressed, as FALSE can never be TRUE.

USR

Calls a machine code program and gives the number in register R0 on return.
Example:

```
char%=USR(code%)
```

Variable code% contains the address of the machine code to be called. On return, char% is set to the number in R0.

VAL

Gives the numerical value of a number in the string following it in brackets.
Example:

```
PRINT VAL("45.7")+3
     48.7
```

The number may consist of figures 0 − 9, a dot (.) for a decimal point and ‘E’ (as used in scientific notation). The number is calculated from the string up to its end or the first character other than those above.

See also EVAL.

VDU

Sends one or more characters to the screen.
Examples:

```
VDU 7
VDU 24,300;200;1000;900;
```

A VDU code between 32 and 126 will print a character on the screen. VDU 127 is the backspace and delete character. Codes below 32 perform special functions and are frequently followed by a series of numbers to pass data. See Appendix 2 for a full list.

The first example sends a single number 7 to the screen, which produces a beep. In the second example, VDU 24 sets the size of the graphics window and is followed by eight other numbers. Because VDU codes are 8-bit numbers (0 − 255) and screen coordinates can go a lot higher than this, each coordinate is sent as two numbers, the low byte (number MOD 256) and the high byte (number DIV 256). To avoid having to work out these numbers, Basic allows a number to be sent as two bytes by following it with a semi-colon (;). The second example, therefore, consists of nine numbers. It is important to include the semi-colon even if a number is less than 256, and to include a semi-colon after the last number.

VOICE

Sets a sound channel to reproduce a particular sound waveform.
Example:

```
VOICE 1,"StringLib-Hard"
```

This example sets sound channel 1 to produce the ‘**StringLib-Hard**’ waveform. As this is the channel which produces the beep, this command will alter its sound. To get back to normal, type:

```
VOICE 1,"WaveSynth-Beep"
```

or alternatively:

```
*ChannelVoice 1 1
```

The Basic command VOICE does the same job as the operating system command *ChannelVoice, except that the latter can refer to a waveform by its number and VOICE cannot.

VOICES

Sets the number of sound channels to be used.
Example:

```
VOICES 2
```

The number of channels in use has to be a power of two − either 1, 2, 4 or 8 channels. Each channel requires a lot of processing time, so setting more channels than are needed will slow the machine down unnecessarily.

VPOS

Gives the vertical position of the text cursor.
Example:

```
vert%=VPOS
```

If vert% is, say, 3, the text cursor is three lines down from the top of the screen or text window, if set.

See also POS.

WAIT

Waits for the start of the next vertical scan of the monitor.
Example:

```
WAIT
```

This can reduce flicker when redrawing graphics.

WHEN

Part of a CASE ... OF ... ENDCASE structure.
Example:

```
CASE num% OF
  WHEN 1:PROCthis
  WHEN 2,3:PROCthat
ENDCASE
```

If num% equals 1, PROCthis is executed and the program then proceeds straight to ENDCASE. If num% equals either 2 or 3, PROCthat is executed.

If num% is not 1, 2 or 3, nothing happens. To execute an instruction when the value of num% is not covered by any of the WHEN lines, see OTHERWISE.

WHILE

Start of a WHILE ... ENDWHILE loop.
Example:

```
WHILE done%=FALSE
  PROCdo_this
ENDWHILE
```

This loop is similar to a FOR ... NEXT loop except that the condition which determines whether or not the loop is executed is tested at the beginning of the loop, not the end. This allows for the situation where the loop may not be executed at all.

WIDTH

Sets the number of characters displayed or printed on a line when listing a Basic program.
Example:

```
WIDTH=80
```

The listing will start a new line after 80 characters.

The listing on the screen will automatically go to a new line when it reaches the right-hand side of the screen. This instruction is particularly useful if you are printing out a listing and you need to limit the number of characters on each line on the paper.

Appendix 2

VDU Codes

The VDU command sends the numbers or values of variables which follow it to the screen. Numbers between 32 and 126 are ASCII codes for characters which are printed on the screen. Code 127 is backspace and delete. Numbers above 127 also produce characters on the screen.

VDU codes below 32 have various control functions and are explained in this appendix.

VDU	Purpose
0	Does nothing
1	Sends the next character to the printer only
2	Turns on printer
3	Turns off printer
4	Selects text cursor for text printing
5	Selects graphics cursor for text printing
6	Turns on the VDU drivers
7	Produces a beep
8	Moves cursor back one space
9	Moves cursor forwards one space
10	Moves cursor down one line
11	Moves cursor up one line
12	Clears the screen or text window
13	Moves the cursor to the start of the line
14	Selects paged mode
15	Turns off paged mode
16	Clears the graphics window
17	Sets the text colour
18	Sets the graphics colour
19	Redefines logical colour 1

20	Resets default colours
21	Turns off VDU drivers
22	Changes screen mode
23	Miscellaneous VDU commands
24	Sets the graphics window
25	PLOT command
26	Resets text and graphics window to cover the entire screen
27	Does nothing
28	Sets the text window
29	Moves the graphics origin
30	Moves the text cursor to its home position
31	Moves the text cursor

VDU 0

Does nothing.

VDU 1

The next VDU character is sent to the printer only and not to the screen. The printer must have already been turned on with VDU 2.

Printers frequently use sequences of codes beginning with an escape character, which is sent as number 27. If these numbers were also sent to the screen, VDU 27 would do nothing but the following characters may cause havoc.

A printer may, for example, use a sequence consisting of Esc E, meaning ‘Set emphasised mode’. This would be sent as 27, followed by 69 which, if sent to the screen, would display an unwanted ‘E’. This may be prevented by using:

```
VDU 1,27,1,69
```

VDU 2

Turns on the printer. Anything sent to the screen will also be sent to the printer.

VDU 3

Turns off the printer, cancelling the effect of VDU 2.

VDU 4

Text is printed at the text cursor position with the text foreground colour. This is the usual situation. This call cancels the effect of VDU 5.

VDU 5

Text is printed at the graphics cursor position using the graphics foreground colour which is set by GCOL. The text position is set with a MOVE command, not a TAB command.

VDU 6

Turns on output to the screen. This cancels the effect of VDU 21.

VDU 7

Produces a beep.

VDU 8

Moves the cursor back one space. This does not delete the previous character, but positions the cursor under it.

VDU 9

Moves the cursor forwards one space.

VDU 10

Line Feed. Moves the cursor down one line.

VDU 11

Moves the cursor up one line.

VDU 12

If VDU 5 mode is not set, this call clears the text window (or whole screen, if it isn’t set) to the text background colour (CLS).

If VDU 5 mode is set, the call clears the graphics window (or whole screen, if not set) to the graphics background window (CLG).

VDU 13

Carriage Return. The cursor is moved back to the beginning of the line which it is on. This is usually accompanied by a line feed (VDU 10) to move the cursor down one line.

VDU 14

Puts the screen into page mode, which limits scrolling to less than one screen height. The scrolling action then waits for Shift to be pressed.

VDU 15

Turns off page mode, cancelling VDU 14.

VDU 16

Clears the graphics window (CLG).

VDU 17+1 character

Sets the text colour.

VDU 17,2 is the equivalent of COLOUR 2.

VDU 18+2 characters

Sets the graphics colour and the way graphics appear on the screen.

VDU 18,0,3 is the equivalent of GCOL 0,3.

VDU 19+5 characters

Redefines the colour palette.

This command takes the form VDU 19,col%,action%,red%,green%,blue%.

For action% values between 0 and 15, red%, green% and blue% are not used. Colour col% is redefined as colour action%. This is the equivalent of COLOUR col%,action%.

If action% is 16, colour col% is redefined in terms of red%, green% and blue%. This is the equivalent of COLOUR col%,red%,green%,blue%.

VDU 20

Puts all colours back to normal. This cancels out the effect of VDU 19.

VDU 21

Prevents character output from reaching the screen until VDU 6 is sent.

Characters will still be sent to the printer if it has already been turned on with VDU 2.

VDU 22+1 character

Changes screen mode.

VDU 22,12 is equivalent to MODE 12.

VDU 23+9 characters

Miscellaneous commands. The first byte following VDU 23 determines which action is carried out − the remaining eight bytes carry the necessary data.

Many of these commands are outside the scope of this guide. The following, though, are of interest:

VDU 23,1,action%,0,0,0,0,0,0,0

Controls the appearance of the cursor:

action%	Cursor appearance
0	Cursor turned off
1	Cursor turned on
2	Cursor does not flash
3	Cursor flashes

VDU 23,17,5,0,0,0,0,0,0,0

Exchanges text foreground and background colours. A second call changes them back again.

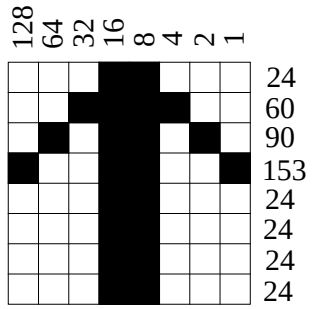
VDU 23,32 to 255, + 8 characters

Redefines the shapes of characters.

Each text character consists of a rectangle measuring eight pixels in either direction. Any character can be redefined by this call, by changing which pixels have the foreground colour and which have the background.

The number immediately following VDU 23 is the ASCII code for the character to be redefined. Each of the eight numbers following contains the pattern for one row of pixels, from top to bottom. In each number, the least significant bit sets the state of the right-hand pixel and the most significant bit the state of the left-hand pixel.

For example, VDU,23,128,24,60,90,153,24,24,24,24 will redefine the character with ASCII code 128 to be an arrow pointing upwards. VDU 128 will plot it on the screen.



This technique was used by earlier Acorn computers such as the BBC Model B to produce user-defined graphics. It has been superseded in 32-bit machines by the use of sprites.

VDU 24+8 characters

Defines the graphics window.

Each coordinate is specified by two bytes − the coordinate MOD 256 followed by the coordinate DIV 256. These may be sent as one number by following it with a semi-colon (;).

For example, VDU 24,300;200;800;600; consists of a total of nine bytes. This call defines a graphics window whose bottom left-hand corner is at (300,200) and whose top right-hand corner is at (800,600).

VDU 25+5 characters

PLOT command.

The first byte after VDU 25 contains the PLOT code. The remaining four specify the x and y coordinates in pairs. Each pair may be given as one number followed by a semi-colon, as in VDU 24.

For example, VDU 25,85,400;500; will plot a filled triangle between (400,500) and the previous two locations of the graphics cursor.

See Appendix 3 for a full list of PLOT codes.

Although VDU 25 may be used for a PLOT command in this way, it is actually much quicker to use Basic's PLOT instruction. The VDU method requires six calls to the operating system − PLOT does it in one call, using a special software interrupt or SWI (see section 14).

VDU 26

Reset both text and graphics windows to cover the entire screen.

This cancels the effect of VDU 24 and VDU 28.

VDU 27

Does nothing.

VDU 28+4 bytes

Redefines the text window.

For example, VDU 28,5,25,45,10 defines the text window as having its bottom left-hand corner five spaces in from the left and 25 rows down from the top, and its top right-hand corner 45 spaces in from the left and 10 rows down from the top.

VDU 29+4 bytes

Changes the position of the graphics origin. The four bytes following VDU 29 contain the x and y coordinates in pairs. Each pair may be given by one number followed by a semi-colon (;) as in VDU 24.

For example, VDU 29,500;400; will redefine the graphics origin (0,0) to be the point on the screen that was originally (500,400).

VDU 30

Moves the text cursor to its 'home' position in the top left-hand corner of the screen. If VDU 5 operation is in force, the graphics cursor is moved to this position instead.

VDU 31+2 bytes

Position the text cursor.

For example, VDU 31,20,10 is the equivalent of TAB(20,10) in a PRINT command and will cause the next text character to be displayed 20 spaces in from the left and 10 rows down from the top.

Appendix 3

Complete List of PLOT Codes

Graphics commands such as MOVE, DRAW, RECTANGLE etc. are actually special types of PLOT command. The PLOT keyword is followed by three numbers. The first is the action code and the other two are the x and y coordinates.

The action code is formed by adding two numbers together. The first of these refers to the type of object being produced, e.g. drawing a line, filling a triangle etc. and the second one refers to the way in which the object is put on the screen:

Code	Action
0	Solid line with both end points
8	Solid line without final end point
16	Dotted line with both end points
24	Dotted line without final end point
32	Solid line without starting point
40	Solid line without either end point
48	Dotted line without starting point, continuing pattern
56	Dotted line without either end point, continuing pattern
64	Plot a single point
72	Draw a line to left and right of point until a non-background colour is reached
80	Fill a triangle, using given and previous two sets of coordinates
88	Draw a line to the right of the point until the background colour is reached
96	Draw a filled rectangle
104	Draw a line to left and right of point until the foreground colour is reached
112	Draw and fill a parallelogram
120	Draw a line to the right of the point until a non-foreground colour is reached
128	Flood fill around the point until a non-background colour is reached
136	Flood fill around the point until the foreground colour is reached
144	Draw a circle

152	Draw and fill a circle
160	Draw a circular arc
168	Draw a segment
176	Draw a sector
184	Move or copy a rectangular portion of the screen
192	Draw an ellipse
200	Draw and fill an ellipse
208	Used with fonts
216	Reserved for future use
224	Reserved for future use
232	Plot a sprite
240	Used by user programs
248	Used by user programs

The second number is added to the first one and describes how the object is plotted on the screen. These numbers are not used for moving or copying rectangles (action codes 184 − 191):

Code	Effect
0	Move graphics cursor relative to the last point
1	Plot relative to the last point using the graphics foreground colour
2	Plot relative to the last point using the inverse of the graphics foreground colour
3	Plot relative to the last point using the graphics background colour
4	Move graphics cursor to the coordinates given
5	Plot at the coordinates given using the graphics foreground colour
6	Plot at the coordinates given using the inverse of the graphics foreground colour
7	Plot at the coordinates given using the graphics background colour

PLOT action codes 184 − 191, which move and copy rectangular portions of the screen, are slightly different:

Code	Action
184	Move graphics cursor relative to the last point
185	Move a rectangle relative to the last point
186	Copy a rectangle relative to the last point
187	Copy a rectangle relative to the last point
188	Move graphics cursor to the coordinates given
189	Move a rectangle to the coordinates given
190	Copy a rectangle to the coordinates given
191	Copy a rectangle to the coordinates given

The PLOT command only includes one set of coordinates but a shape has several points which all need to be specified. This is done by ‘visiting’ one or two points with the graphics cursor and specifying a further one in the PLOT command:

To draw **a line**, MOVE to one end and PLOT 5, specifying the other end.

A **triangle** is filled by a MOVE to two points and PLOT 85, specifying the third.

A **rectangle** is filled by a MOVE to one corner and PLOT 101, specifying the opposite corner.

A **parallelogram** is filled by a MOVE to two adjacent corners and PLOT 117, specifying the corner opposite the first one visited.

A **circle** is drawn or filled by a MOVE to its centre and PLOT 149 or 157, specifying a point on the circle.

An **arc, segment** or **sector** is drawn by a MOVE to the centre of its circle, another MOVE to the clockwise end and PLOT 165, 173 or 181, specifying the anticlockwise end.

An **ellipse** is drawn or filled by a MOVE to its centre followed by a MOVE to where the ellipse passes above or below the centre and PLOT 197 or 205, specifying the highest or lowest point of the ellipse.

Index

Basic keywords are referred to where they are introduced in the main text. In addition, all Basic keywords are listed in Appendix 1.

Program filenames are in italics.