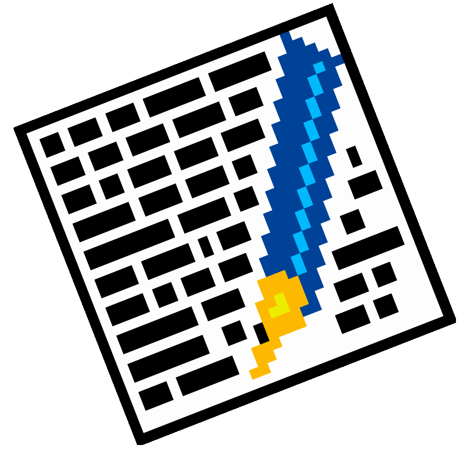
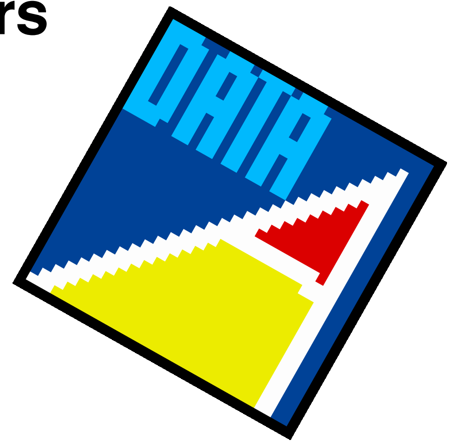
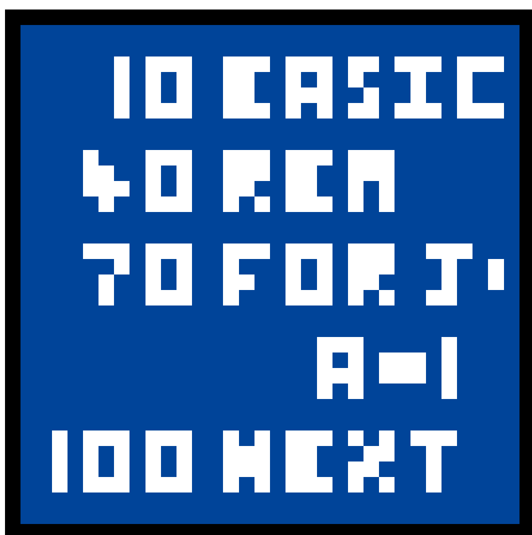


Starting Basic

on Acorn RISC OS computers





Ray Favre

Starting Basic

on Acorn RISC OS computers

Ray Favre

©Ray Favre 1996

This book has been produced for charity. Only costs actually paid to third parties for the reproduction, postage and packing are retained from the selling price. The remainder goes to charity.

I am particularly indebted to John Osborne and his family of ***Temple Printers (at 3 Temple Gardens, Staines, Middlesex, TW18 3NQ Tel: 01784 451898)*** *who, unsolicited, offered to assist in the exercise and undertook reproduction and binding of the initial 50 copies at a price well below commercial quotes. He also provided master sheets to make it easier to run additional copies (like this one) via the local photocopy shop.*

The particular charity is the ***Norfolk and Norwich Children's Fund*** *- a medical assistance charity for children in need. This was chosen by* ***Paul Beverley*** *of* ***Norwich Computer Services*** *- publisher and editor of* ***'Archive'*** *subscription magazine, in whose pages appeared a series of articles by me which prompted the book. Paul also kindly publicised the charity exercise, which brought it to the attention of John Osborne and identified most of the buyers before publication.*

Ray Favre

August 1997

26 West Drayton Park Avenue, West Drayton, Middlesex, UB7 &QA U.K.

E-mail: rayfavre@mail.zynet.co.uk

Contents

	Introduction	**1**
1.	**First steps**	**5**
2.	**The Tutorial project**	**21**
3.	**Variables**	**25**
4.	**Keywords**	**35**
5.	**Procedures and Functions**	**51**
6.	**The REPEAT and WHILE control loops**	**71**
7.	**Making an INPUT and the IF ... THEN construction**	**89**
8.	**Branching with CASE ... OF ... WHEN**	**103**
9.	**The FOR ... NEXT loop**	**109**
10.	**String manipulation**	**119**
11.	**Arithmetic and logical operators and built-in mathematical functions**	**127**
12.	**Introducing graphics**	**133**
13.	**More graphics**	**145**
14.	**Text and graphics viewports and the use of VDU 5**	**157**
15.	**Final steps in Loan project**	**177**
16.	**VDU commands**	**193**
17.	**Data storage/retrieval (READ, DATA and RESTORE)**	**201**
18.	**Data storage/retrieval (cont'd) (Arrays and the keyword DIM)**	**207**
19.	**Data storage/retrieval (cont'd) (Data files)**	**215**
20.	**Data storage/retrieval (cont'd) (Direct memory access/indirection)**	**233**
21.	**Colour**	**239**
22.	**Libraries**	**255**
23.	**Using the Operating System (SYS calls)**	**261**
24.	**The nature of Wimp programming**	**267**

Appendix 1.	'Current Directory'	277
Appendix 2.	Variable names	279
Appendix 3.	TRUE and FALSE	281
Appendix 4.	Table of ASCII codes	285
Appendix 5.	Hex and Binary numbers	287
Appendix 6.	The operators AND, OR, EOR, NOT, <<, >>, and >>>	295
Appendix 7.	Declaring string variables	301
Appendix 8.	Memory and the storage of numbers	303
Appendix 9.	VDU 'control codes'	307
Appendix N.	Operations on whole arrays	309
Appendix 11.	Supplementary notes on Tutorial program formulae	313
	Index	

Foreword

I hadn't anticipated dealing with another of Ray Favre’s books after I completed the re-creation of *Dr Wimp’s Surgery* though, in fairness, back in 2019, Jim Nagel had hinted that there was another book in the wings and I felt I couldn’t just leave it. Gavin Smith was also keen to see the project through to completion. The only stumbling block was the non-availability of a copy of the book.

After an appeal through the mailing lists, Michael Stubbs generously volunteered his own copy. Even more generous was his subsequent agreement to let me strip the book of its spiral binding so that I could feed it through the automatic sheet feeder on my scanner. Without the ability to do that, the process would have been tedious in the extreme. I am most grateful to Michael for his help.

Another problem cropped up with the process of gradually building up the Basic program from chapter to chapter, which was the basis of Ray’s approach to introducing new topics. A misplaced variable had found its way into one of the programs as printed in the book. Not being a programmer, I couldn’t resolve the problem and resorted to a plea on *comp.sys.acorn.misc* for a copy of the program. This time I was rescued by Terry Kelly who found his copy of the disc which came with the book as bought from Ray.

Not only was the program there (in all its stages from chapter to chapter) but all of Ray’s original graphics were there too. That solved another problem for me and so I am most grateful to Terry for his help too.

Happily, the graphics are all in the correct colours which I couldn’t reproduce accurately on RISC OS 5 without much fiddling, especially the default blue screen background.

I’m pleased to say that there isn’t a third book to cope with as time has become more and more limited. Nevertheless I have enjoyed all the time and effort put into these two projects and am grateful to Jim and Gavin for giving me the chance to secure Ray’s legacy.

John McCartney

Oct 2025

Introduction

This book is mainly for those who are new, or fairly new, to programming in BBC Basic on Acorn Risc OS computers.

Acorn's excellent Risc OS computers are designed to operate best in the multi-tasking Wimp environment i.e. the user interacts with the computer using **W**indows, **I**cons, **M**ouse and **P**ointer.

Comprehensive help is available from various sources for those who wish to produce Wimp programs. But all that assistance assumes (necessarily) that the intending Wimp programmer is already competent in BBC Basic or in the C language - which presents a problem to a beginner.

Old books covering BBC Basic (from the early 1980s) are of limited help, because there have been very significant extensions to BBC Basic in the intervening years and they need to be read with some experience in order to translate them into today's context. So, again, the beginner has a problem.

This book therefore tries to fill the gap: it covers today's version of BBC Basic and it starts from scratch in the non-Wimp environment. It is designed to be used both as a hands-on beginner's tutorial and as a subsequent reference manual.

Assumptions

The main assumption is that you have an Acorn Risc OS computer which is armed with the BBC Basic V (or VI) programming language (they all come with this, ready to use) and that you have little or no knowledge about how to program.

It is also assumed that you will probably want to progress to Wimp programming in Basic in due course; so there are frequent comments and pointers to that end and a specific introduction to Wimp programming in the last chapter.

Acorn's Guide books

The only other assumptions are that you have the Acorn **Welcome Guide (WG)** & **User Guide (UG)** for your computer and that you've read them. There are several references to these documents throughout the book, particularly in the early chapters.

These guides cover very well all the commonly-used mouse and button-clicking operations in the Risc OS Desktop mode; such as closing/ opening directories, starting applications, dragging & dropping files, etc. You will need to get used to many of these actions very early in the book, so it is worthwhile spending time on the relevant parts of the WG and UG now!

Some references are also made to Acorn's Basic Reference Manual and Acorn's Programmer's Reference Manual (PRM), but only to tell you where certain advanced information can be found. A.v a beginner, you definitely do not need the PRM - and you can safely leave the Basic Reference Manual until you are happy with the fundamentals: after which it will become essential.

Book sequence

Chapters 1, 2 and 3 start from square one: firstly ensuring you can set-up the computer correctly for Basic programming and then concentrating on how some fundamental programming actions are carried out.

In these chapters, the book deliberately tries to avoid introducing too many new programming instructions - yet, from the start, gives you programs that run properly. This necessarily means that the programs in these chapters are not good examples of how Basic programming ought to be done! Rather they sacrifice 'purity' in the interests of getting you used to the 'mechanics' of entering, listing, amending, saving and running Basic programs - and understanding what is happening.

Chapters 4-15 introduce the more important Basic keywords and show how their associated instructions and constructions are put together in practice. The topics are introduced in an order that is typical of the order in which you would need them in many non-Wimp programming projects. As more keywords/instructions are added to the armoury, more attention is given to the importance of good programming practices. Throughout these chapters, there is an emphasis on 'hands on'.

With the fundamentals now on board, Chapters 16-23 explore several further topics which are encountered in many programs and are therefore on the 'need to know' list for beginners. Several of these chapters are also of particular importance to Wimp programming and points are highlighted accordingly in the text.

Finally, Chapter 24 gives a brief insight into the differences between non-Wimp and Wimp programming and an introduction of the basic Wimp process.

Several Appendices follow the final chapter - containing some important reference material.

Tutorial project

To assist in the 'hands on' process, Chapters 3-15 carry a single, progressing, tutorial program - explained in Chapter 2 - which is developed chapter by chapter as each new topic is covered. This program is used as a vehicle to demonstrate all the fundamental features of BBC Basic and encouragement is given to use it as a 'playground' to explore specific points of interest.

Final point

The only way to progress is to sit and 'play' at the keyboard. You are unlikely to damage the computer - although you will probably 'freeze' it up sometimes and lose some data occasionally - but you'll learn faster through your mistakes.

Conventions used in this book.

`Program listings are in the 'bold' version of this typeface.` The same typeface, but not 'bold', is used within the normal text when reference is made to program items e.g. ".... the procedure `PROCchooseUnknown` is ...."

`<angle brackets>` are used (also with the above program typeface) in two ways: firstly to indicate in example listings that the programmer needs to substitute something appropriate of his/her own choice at that place in the program - and, secondly, to represent a specific keyboard key press or mouse button action e.g. `<select>` to mean a press of the mouse 'select' button.

Bold text is used for **emphasis**.

`Program listing titles are in this typeface.` Program titles are referenced to the chapter in which they appear, using letters to distinguish between more than one in the same chapter e.g. ***`Program Progl9a`*** and ***`Program Progl9b`*** are, respectively, the first and second program listings appearing in Chapter 19.

Programs in the tutorial Loan project series follow a similar pattern, but apart from the first few versions, **only updates of the Loan program are listed**. For example, ***`Loan Update 7a`*** and ***`Loan Update 7b`*** are listings of successive updates to the Loan program, appearing in Chapter 7. When incorporated, these updates will produce ***`Program Loan 7a`*** and ***`Program Loan 7b`*** respectively. *(The disc associated with this book contains all the updates* ***plus*** *the corresponding complete updated programs for each stage.)*

Italic typeface is used mainly for incidental comments within the normal flow of the text.

It is also sometimes used, indented from both margins like this, for an important matter outside the main flow of the text - and also for summary comments at the start and! or finish of chapters.

1. First steps

What a computer program is - The nature of Basic instructions, program lines and line numbering - Getting the computer ready for Basic - Command Line and Task Window - Basic Immediate Mode - Amending, listing and saving programs from command line/Task Window - Using !Edit for programming.

What is a computer program?

A computer program is a sequence of instructions telling the computer to carry out certain tasks in order to produce some desired end result. A program typically starts its life by a programmer typing the required instructions at the computer's keyboard.

The heart of the computer can actually only carry out a fairly small range of very trivial tasks - and then only if the instructions are given to it in a very basic form (called "machine code"). So the programmer somehow has to arrange for the computer to receive these machine code instructions.

Normally, a programmer has a choice: he/she could analyse the job to be done in very, very fine detail, until it is broken down into an enormous sequence of the computer's trivial tasks - then convert this list directly into machine code instructions. Or he/she can use a translation device, which allows the instructions to be written and input to the computer in something more close to normal language - and which also automatically converts these higher-level instructions into the necessary machine code. This choice is depicted in Figure 1.1.

Clearly, most programmers opt for the translation device - which is called a "programming language" - and is, itself, a computer program. *(But note that you do not get something for nothing: any translation device results in some loss of speed and some increase in the memory needed ? compared with using machine code directly. Normally this price is well worth paying.)*

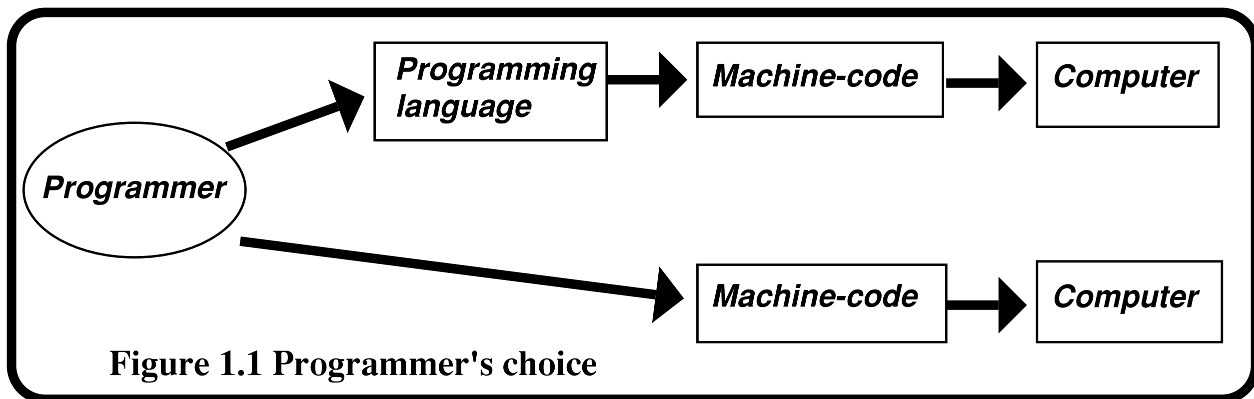


Figure 1.1 Programmer's choice

There are many programming languages, each having its own advantages and disadvantages for the particular job in hand. There are also cases where different programming languages have been developed from the same root and the resulting languages are regarded as different “dialects” of a common language e.g. ‘BBC Basic’ and ‘GWBasic’. This can be a little misleading, because such dialects are invariably incompatible with each other. For instance, you cannot run a program written in ‘GWBasic’ on a computer fitted with ‘BBC Basic’, and vice versa. However, it is usually a practical proposition for an experienced programmer to convert, ‘by hand’, a program from one dialect into another without undue difficulty.

Finally, like most software, it is usual to find different editions or “versions” of a programming language i.e. the language has been upgraded in some way and/or some earlier “bugs” have been eliminated. In these cases, programs written in an older version will often run satisfactorily on a computer fitted with a later version, or any changes needed are likely to be minor.

In this book we will be using the Basic language and, more specifically, the “BBC Basic” dialect.

Further, we will be concentrating on Version V of BBC Basic (with some mentions of Version VI, which is not really a later version). BBC Basic V is an extremely good dialect of Basic: it is both easy to learn and entirely suitable for full commercial use. It also has the distinct advantage of providing an easy way to write and employ machine code routines directly from the Basic program, for circumstances where the programmer would prefer this.

Unless there is a good reason otherwise, this book will refer to BBC Basic V simply as “Basic” from now on.

Programming instructions

In Basic, the available programming instructions are based on a set of words ("keywords") which are, more or less, in plain English. These keywords are always strictly in capital letters ("upper case"). Thus, `PRINT`, `INPUT` and `CIRCLE` are all Basic keywords, each telling the computer to take some action (closely matching what the word normally means in English, which makes Basic programs easy to understand in print.)

There are about 160 Basic keywords, which is really a very small number for learning purposes. Chapter 4 says more in introduction of keywords.

A Basic program is constructed from the keyboard by typing the instructions, line by line. Each line is started with a number which identifies the line uniquely and sets the the order in which its instruction(s) will be carried out when the program is run. The required Basic instruction(s) then follow. A program line is ended by pressing the `<return>` key *(see WG "Keyboard")*.

For example, a single line from a typical Basic program might be:

```
110 PRINT "Press any Key"
```

This program line would be known as "Line 110". It has one Basic instruction in it - which happens to tell the computer to print the words `"Press any Key"`. To end this line the programmer has pressed the `<return>` key - which, in print, is an invisible action, but the computer responds to it nonetheless. *(By the way, a sequence of letters/characters such as "`Press any Key`" is called a "string", however many, or few, characters it contains.)*

It is common to call a single, complete instruction a "Basic statement".

You can put more than one Basic statement on a single Basic line, by separating each statement with a colon. Thus, the single program line:

```
120 MOVE 0,0 : DRAW 100,100 : DRAW 512,512
```

contains three Basic statements - it is a "multi-statement line". The three instructions will be carried out in the conventional order (i.e. `MOVE,DRAW,DRAW`) and it will be treated exactly the same as if the three statements had been put on three separately-numbered lines in the same sequence, such as:

```
120 MOVE 0,0
130 DRAW 100,100
140 DRAW 512,512
```

The only limit to the length of a Basic program line - whether single or multi-statement - is that one line must not exceed about 240 characters. *(Three 80-character lines is a good guide. Explanation of the precise limit is beyond this book, but Basic's error reporting system will tell you when a line is too long.)*

It is not good practice to use multi-statement lines when developing a program, particularly for beginners, so we will normally use single statement lines in this book.

When a Basic program is run, the computer deals with these lines of instructions in line number order (and, as already stated, from left to right on multi-statement lines) and thus the sequence of the program is determined. But, as will be seen, some keywords deliberately tell the computer to leave the normal numerical sequence permanently or temporarily and in this way the programmer can control the structure and flow more comprehensively.

Basic program lines are numbered using whole numbers ("integers") in the range from 0 to 65279 inclusive, which is more than adequate for all normal purposes. In fact, it is conventional to start at 10 and go up in steps of 10 - and even that usually copes. Increasing by steps of 10 makes it easier to add extra lines during development.

Getting something on the screen

To get started (**after** you have read and tried out the things introduced in "Getting started" in your Welcome Guide) switch your computer on and wait for the start-up sequence to be completed - the screen will then almost certainly look something like Figure 1.2 or Figure 1.3 - although the 'prompt' may not be an asterisk in Figure 1.2, or there may also be 'windows' present on Figure 1.3.

If your screen looks like Figure 1.2, your computer is configured to start up in Command Line mode. If it looks like Figure 1.3, it is configured to start up in "Desktop mode" *(see WG)*.

We want to be in Desktop mode; so, if your screen looks like Figure 1.2, simply type Desktop at the "prompt" - followed by a press of the `<return>` key. After a short re-start sequence, your screen should then look something like Figure 1.3. *(The User Guide chapter on "The Command Line" helps considerably with the differences between desktop' and 'command line' and the different ways of moving between them - and how to configure your computer to start in desktop mode.)*

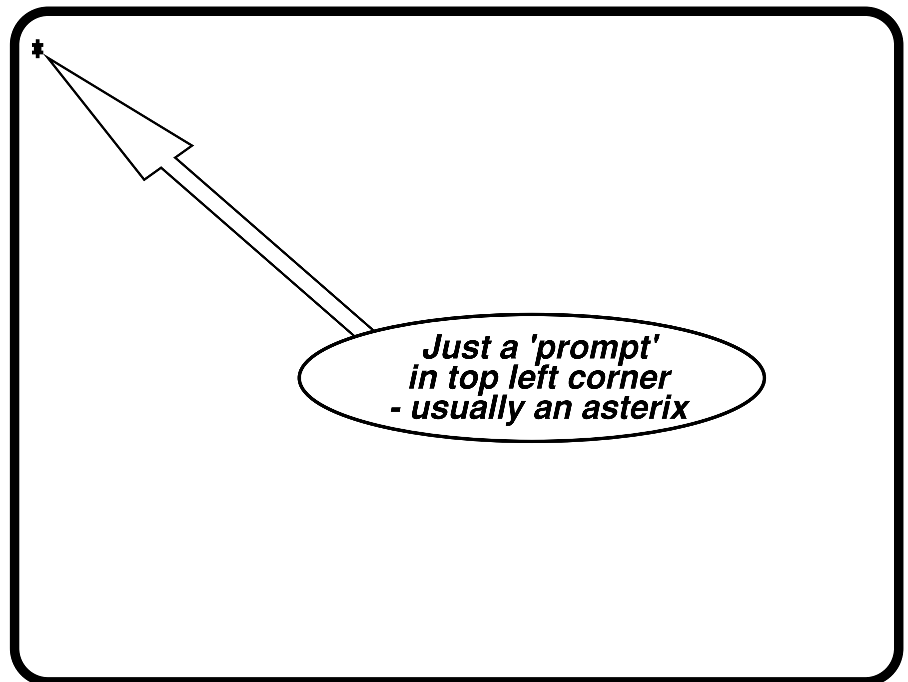


Figure 1.2

Typical 'Command Line' start-up screen

Figure 1.3

Typical Desktop start-up screen

Once in Desktop mode, the bottom right hand corner of the screen will show a green acorn (the Acorn ‘Task Manager’ symbol) or one of its earlier equivalents perhaps, *(see WG “RISC OS Desktop”)*. Click with `<menu>` on this symbol and then click with `<select>` on the menu item “Task Window” - see Figure 1.4.

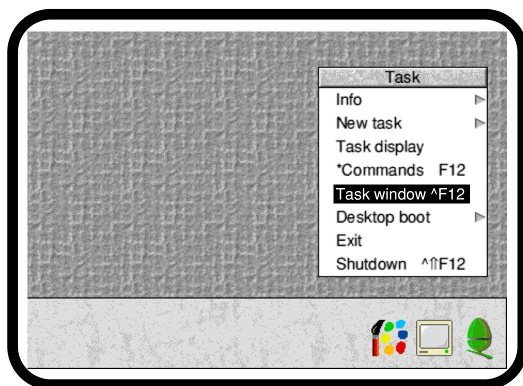


Figure 1.4

A window will then open on the screen - see Figure 1.5 - and it will have the Command Line prompt in its top left-hand corner (normally an asterisk, as in the figure) with a thin vertical "caret" next to it.

The caret is the text cursor which tells you where the next letter will appear when you press a keyboard key *(see WG "Keyboard"). (The Task Window background colour will probably be white and the caret will probably be red (and may be flashing). You may also notice that an icon for the !Edit application - or another similar text editor - has appeared on the icon bar.)*

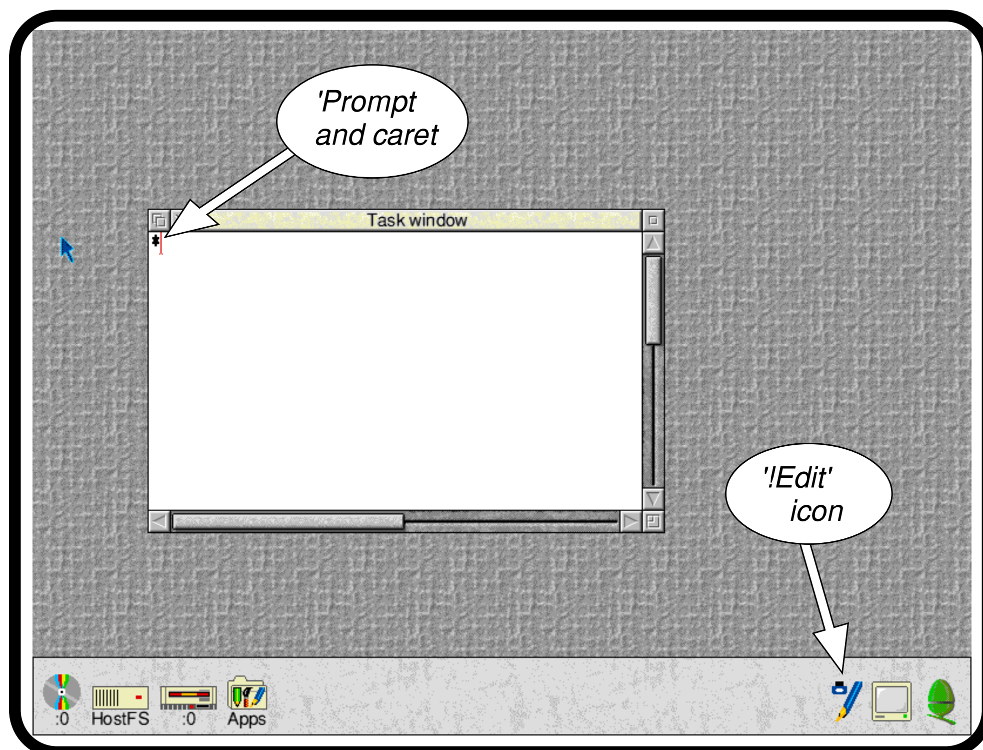


Figure 1.5

'Task Window' opened

You are now in Command Line Mode within a desktop Task Window, usually much more convenient than working in 'true' Command Line mode (i.e. as in Figure 1.2, which is 'outside' the desktop and takes over the whole computer, stopping multi-tasking).

Now try things out: type some nonsense like zzz then press <return>. You’ll get an error message “File ‘zzz’ not found” and the next line will be set up with the “*” again, waiting for you to type something else.

This time, type:

`BASIC` (plus <return>)

You’ll now get a short message confirming you have now entered the Basic environment - i.e. you are “in Basic” - and giving the Version Number (worth noting, by the way) and how many bytes of RAM (see WG Glossary) you have available for Basic operations.

Also, the prompt has changed to a “>” symbol - which is the normal Basic prompt. The fourth and fifth lines printed on the screen in Figure 1.6 show the type of confirmatory message you will get, with the new prompt seen at the start of the sixth printed line.

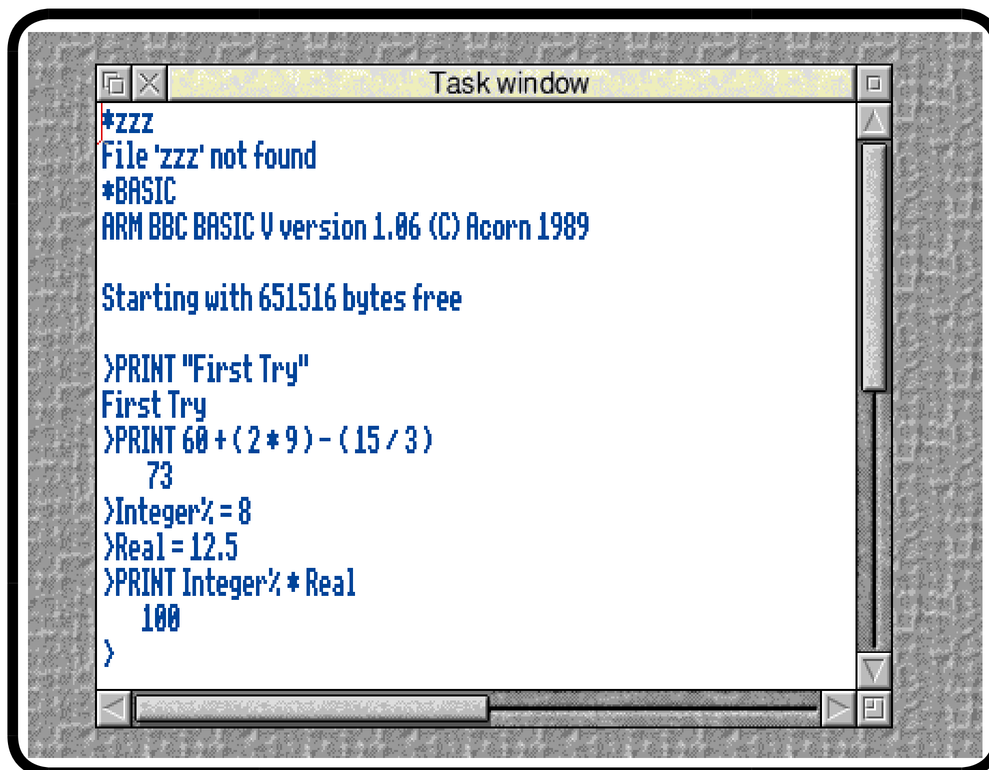


Figure 1.6

The first few steps: starting Basic and using Immediate Mode

Basic Immediate Mode

Now type the Basic instruction:

`PRINT "First Try"` (plus `<return>`)

and on the next line will appear:

```
First Try
```

i.e. the computer has immediately carried out your typed Basic instruction. Note that the Basic prompt reappears on the following line, waiting for your next move. *(In a Task Window or Command Line, if you make a typing mistake on a line before you press the* `<return>` *key, you can use the* `<Delete>` *key - not the usual backspace key - to erase the mistake. Then retype as required.)*

Now type:

`PRINT 60 + ( 2 * 9 ) - ( 15 / 3 )` (plus `<return>`)

and the result:

```
73
```

will be printed on the next line - and the prompt reappearing again.

Once again, the computer has carried out the Basic instruction immediately. *Note that, in Basic instructions, the "*" symbol is used instead of "×" to mean "multiply" and the symbol "/" is used instead of "÷" to mean "divide".*

Now type:

```
Integer% = 8
Real = 12.5
PRINT Integer * Real
```

with a `<return>` after each line. The answer:

```
100
```

will then appear. Figure 1.6 shows the screen up to this point.

What these trivial examples have shown is that, once you have entered the Basic environment - in a Task Window (as above) or in Command Line mode proper - you can use Basic instructions in Immediate Mode if you need to. The Basic processor will respond immediately to your input - as it has done above - rather like an up-market calculator.

This may not seem to be a big deal at the moment, but as you progress in Basic programming you will find numerous occasions where a quick check of something in Immediate Mode is very helpful - and, as we will see, it can also sometimes get you out of a problem when a Basic

program keeps returning an error.

But note that the Task Window cannot action Basic's graphics instructions - it will simply ignore them. Further, the Command Line proper is not very satisfactory for graphics either. Don't worry! By the time you reach the chapters on graphics, you will be competent to display them in other ways.

Basic Programming Mode

Despite needing to know how to use it, Immediate Mode is clearly very limited, because you have to type in the Basic instructions each time you want to use them. What we need is a means of creating, storing and recalling a set of Basic instructions so that they can be used time and time again i.e. a program. In Basic, this is achieved simply by numbering each instruction line - in the following way.

Typing a program

While still in the Task Window, carefully type the following list of Basic instructions - called ***Program Prog 1a*** - line by line, using the line numbering shown and not forgetting to press `<return>` at the end of each line.

Be careful not to add any spaces where none are shown and to use upper and lower case letters exactly as shown, otherwise an error may occur when you run this program a little later. However, where spaces are shown, it does not matter if you add extra ones - or omit them altogether - but they generally make things easier to read. The lines with just a colon in them are deliberate and will be explained shortly. *(Don't type the words "Program Prog1a" - it is just a title for reference in the book.)*

Program Prog1a

```
10 REM> Prog1a
20 :
30 :
40 String$ = "Second Try"
50 PRINT String$
60 :
70 PRINT 60 + ( 2 * 9 ) - ( 15 / 3 )
80 :
90 Integer% = 8
100 Real = 12.5
110 PRINT Integer% * Real
120 :
130 END
```

Figure 1.7 shows how things will look on the screen as you are typing the listing. *(The instructions in this particular program will be explained a little later.)*

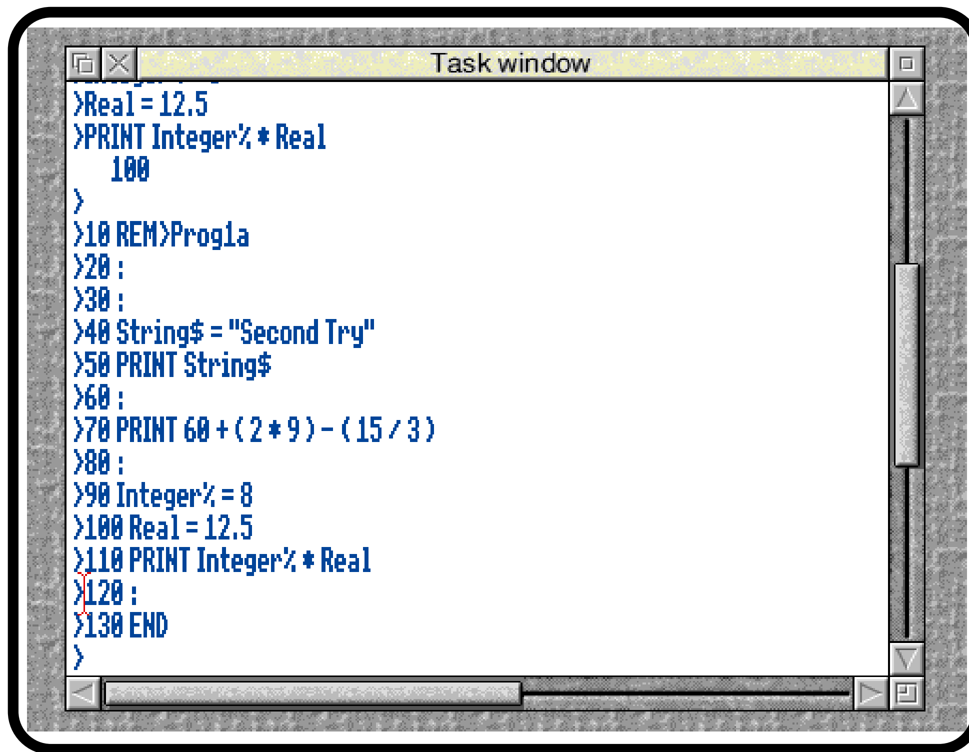


Figure 1.7

Continuing from Figure 1.6: first Basic program, as it appears after typing in

Typing each line number tells the computer that you are entering a Basic program line. So now it does not act straightaway on the instruction when you press `<return>` at the end of each line - as it did in Immediate Mode earlier.

Instead, the computer stores temporarily what you have typed, in line number order - and presents you again with the usual Basic prompt on the next line, waiting for your next action (which may or may not be the entry of another line of the program). Provided you put the shown line number with the shown instruction it does not matter in which order you type these lines. The computer will still store (and later, execute) them in number order. You will note that there is no more than one Basic statement per line in this example.

You do not have to enter all the lines in one go. Provided you do not exit from Basic, you can break off temporarily (after completing any line) and use Immediate Mode if you want, then carry on entering the program lines.

Amending and listing

In a Task window, to amend any existing line you only need to retype it again, with its original line number, in the required new form. As soon as you press `<return>` the new line will replace the original.

If you have typed the lines out of order, or want to see the new listing after amendment, there is a command to list the program in the proper order. The command is list.

To show this and amending in action, first type:

`20 PRINT "First Try"` (Don't forget `<return>`)

which will replace the original Line 20.

Then type:

`LIST` (Again, don't forget `<return>`)

and you will see the revised Line 20, in its right place, as soon as `LIST` does its stuff.

(To delete an existing line completely while in Command Line, simply type its line number and press `<return>`*. Try it. Be careful not to put a space between the line number and pressing* `<return>`*. Use* `LIST` *to prove it's happened. Then retype the original line again to restore things to as they were - once again checking with* `LIST`*.)*

Note the instruction `END` at Line 130, which explicitly tells the computer to end the program. It is not essential in this particular case, but it usually is needed and it is therefore always good practice to include it.

You have now entered and amended a Basic program. The program resides in your computer RAM memory awaiting your next action.

Saving the program

If you have read the WG glossary about RAM, you will know that your program is not very safe at the moment - and would be lost if you quit Basic, switched off the computer or had a power failure. So, before going any further, get into the saving habit. In this case simply type:

`SAVE` (Definitely the last reminder about `<return>`)

Figure 1.8 shows the remaining screen sequence up to this point.

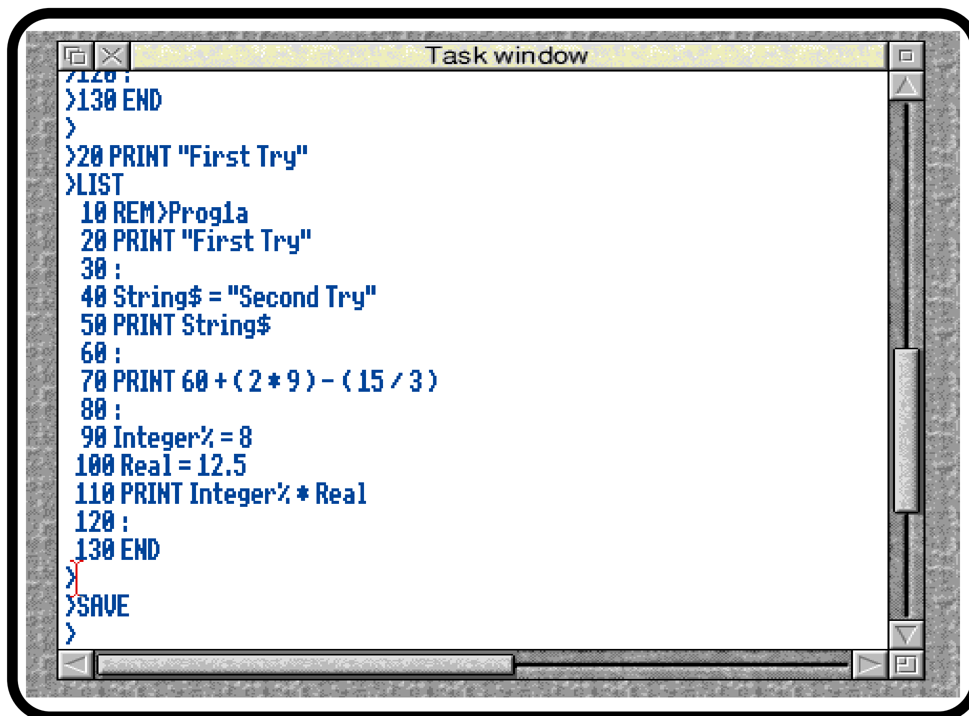


Figure 1.8

Continuing from Figure 1.7: amending Line 20 listing to show amendment, then saving program

You will now find a new file called `"Progla"` in the display of your Current Directory *(see UG "Star Commands" - it is also sometimes called the Currently Selected Directory.)* with a soon-to-be-familiar white-on-blue Basic file icon - as in Figure 1 .9.

Figure 1.9

Basic file icon

The name "Progla" underneath the icon is taken automatically from the first line of the program if you use the rem> format in the first line of the program - as in Line 10 above.

(If you cannot see the new file "`Progla`", even after moving the screen windows around, don't panic! It probably means that the 'Current Directory' is not open on your desktop screen. Appendix 1 anticipates this and tells you how to find your new file.)

You now have ***`Program Progla`*** safely tucked away as "`Progla`".

Always save a program before you run it. *(It would be easy to sermonize about the need to save your work frequently - every five minutes if you're typing a long listing. None of us takes any notice until we each learn the hard way!)*

Also, **never modify or 'play' with the only saved version of a**

program - always copy it first *(see WG and UG)*, with a name of your choice, and then modify the copy. You can then always get back to where you started if you need to - and you will!

Although ***Program Prog 1a*** is trivial and short it contains some important items which we will start exploring soon. It uses some direct values (Lines 20 (as amended) & 70) and introduces "variables" (`String$`, `Integer%` & `Real`) and shows how they can be substituted for direct values in a statement - rather like in algebra. Lastly, and not least, it shows how empty lines can help to make the listing more understandable by splitting it up into logical groupings.

To reassure you that the program does exist, type:

```
RUN
```

Is the result what you expected? As Figure 1.10 shows, you should see "`First Try`","`Second Try`","`73`","`100`" printed on separate lines - then the Basic prompt again.

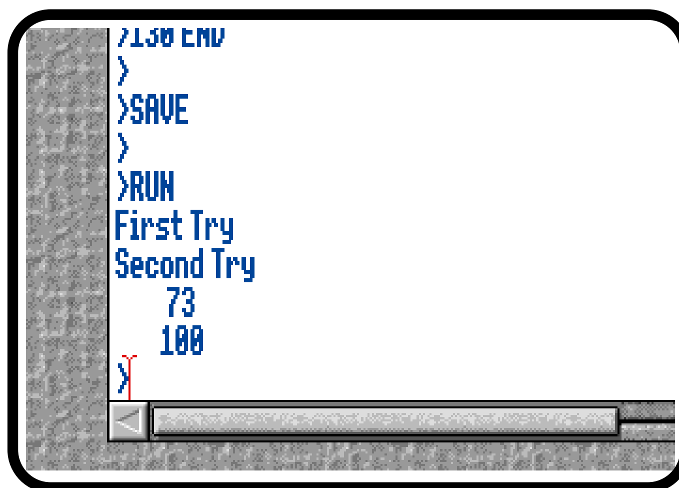


Figure 1.10

Result of running program 'Prog1a'

Did you remember to press `<return>` after `RUN`?

If you get a message telling you that something is wrong, check your typed listing again - in particular, check that the variables are exactly the same each time they occur. (There is a way to get a Basic program itself to help you identify faulty typing etc. - and we will shortly introduce this but it was more important in our first program to keep it simple, without embellishments. So persevere to get Program Progla running OK!)

> ***Colons in deliberately 'empty' lines****. There are two reasons why the deliberately empty lines in* ***Program Progla*** *have had a colon put in them. Firstly, if you take the colons out (and any spaces) and use* `LIST` *again you will find that the empty lines are deleted - simply because, as we are using the Command Line, the computer thinks you*

have taken the line deleting action described a little earlier. Secondly, this book needs a way to distinguish visibly in print between entering a deliberately empty line and deleting an existing line - and the best way is always to use a colon in a deliberately empty line. The colon is used as a separator between statements in multi-statement lines and thus has no harmful effect on its own.

Using !Edit for Basic programs

It is vital to know how to use Basic as described above - and there will certainly be further references to these methods later - but under normal circumstances there is a much better way to enter, edit and run Basic programs.

The !Edit application which came with your computer has specific facilities for Basic programs and we'll normally be using this. *(There are other applications you can use instead - !DeskEdit, !StrongED and !Zap being popular ones. But every Acorn computer has !Edit and therefore it makes sense for this book to use it.)*

So, following on from the above exercise, close the Task Window and select 'Discard' when asked. This will probably leave your iconbar with the !Edit icon still on it. *(However, if another text editor remains instead, then quit that application and open the !Edit application i.e. click* `<select>` *on your "Apps" iconbar icon and double-click on the !Edit icon which appears in the Apps directory window - see UG)*

With the !Edit icon somewhere on the right-hand side of the iconbar, click `<menu>` over it and the menu headed "Edit" - see Figure 1.11 - will appear. Position the mouse pointer over this menu until the word "Create" is highlighted. Now slide the pointer to the right until it is over the short arrow to the right of the highlighted word. This will cause a second menu to open, headed "Create". Move the pointer until "BASIC" is also highlighted - then press `<select>`.

The result will be that a fairly normal-looking desktop window opens - an !Edit window in this case - into which you can type a new Basic program. Again, you have now effectively entered the Basic environment - although it is somewhat different to that entered from the Command Line.

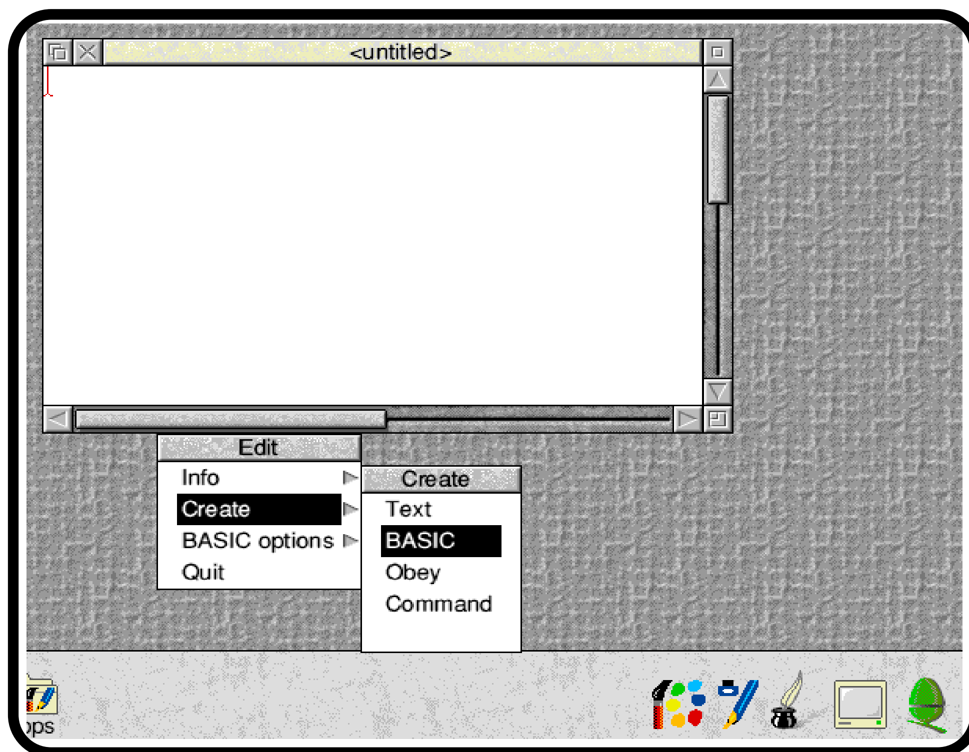


Figure 1.11

Creating a Basic window in '!Edit'

Using !Edit in this way, you do not have to type in line numbers - although you still can, or you can mix lines with and without numbers. !Edit will automatically insert line numbers for you when you save a listing. The price of this convenience is that now you **must** ensure that un-numbered lines are in the correct order before you save/run the program. !Edit is not a mind reader!

You can also ‘drag’ *(see WG/UG)* a saved Basic program file to the !Edit iconbar icon - in which case it will detect it as a Basic program and open a new window with the program listed *(with or without line numbers, according to your choice from the “BASIC options” item on the “Edit” menu)*.

!Edit is far more user-friendly than the Command Line as far as Basic program editing is concerned - mainly because all the advantages of working in desktop mode are available.

The UG has a good description of what !Edit can do and familiarity comes quickly with practice. The ability to save/copy programs easily and to copy and move text quickly is particularly useful for program editing.

To delete an existing line completely, you now simply have to eliminate it with the `<backspace-and-delete>` key or by selecting the line and using `<ctrl-X>` (or its menu equivalent). Conversely, pressing `<return>` on its own will now enter a blank line (unfortunately for this

book's purpose, without a colon in it!).

Purely to get used to using !Edit for Basic, type in ***Program Progla*** (as amended) again, this time in the !Edit window and without the line numbers. Also, change the first line to:

```
REM> Proglb
```

Then press `<menu>` over the !Edit window to reach the Save option, which will produce a typical "Save as" window (see UG) - often just called the "Save box". This will have `BASICfile` written in its writable icon as the default suggested file name. Delete this name and type `Proglb` instead. Now drag the Basic file icon from the Save box to the same directory as ***Program Progla***. This will save the renamed program to that same directory - and the two programs will probably appear side by side in the directory window.

If you then double-click with `<select>` on either file icon, the program will run without further ado - producing its results in a small window somewhat like the previous Task Window and ending with a message for you to "`Press SPACE or click Mouse to continue`". When you comply, the screen will clear and the Desktop will reappear as it was before you double-clicked to run the program.

You have now successfully entered, saved and run a Basic program using !Edit. Practice will soon make the mechanics of doing this second nature, so that you can concentrate on the programming itself.

Read the !Edit section of the User Guide to find how to show or remove line numbers in the !Edit window listings. Try adding line numbers to just some of the lines in the !Edit window and see how user-friendly !Edit is when you save the amended program.

This chapter has covered a lot of fundamental "how to" ground for beginners - and made many references to the User Guide and Welcome Guide. It is very important to get to grips with these points and it is very worthwhile spending time 'playing' at the keyboard with the above material before moving on. In particular, you should feel confident about how to enter the Basic environment in both the Command Line and !Edit - and be familiar with their similarities and differences once you get there.

2. The Tutorial project

To encourage a ‘hands on’ approach, this book is going to carry out a project to build a program step by step over several chapters. Each of these chapters will introduce at least one important programming topic, which will then be used to add another step to the program.

If you follow this project it will introduce you to the practical aspects of Basic programming very quickly and soundly. The various programming topics will be introduced in an order which a beginner is most likely to need - rather than in an order which a straightforward reference book might use.

Just as important perhaps, every time the tutorial program is updated/ amended it will be left in a state that can be run, so that you can see (and try) the effects of the changes at each stage. This feature does bring a minor drawback in that - to ensure even the earliest programs run - a few things need to be introduced into the early versions in advance of explaining them properly. However, there are very few such items and they are probably simple enough to be self-evident anyway. Either way, rest assured that an explanation arrives somewhere in the book before too long.

The Project

The project program works out loan repayments, interest rates etc. This may not sound very exciting, but it has the main advantage that most people will have an idea of what it is about and it will allow us to approach some essential beginners’ topics in a sensible order.

When complete, by the end of Chapter 15, it will include all the major Basic programming instructions and constructions, as well as graphics.

The Loan project will be non-Wimp - deliberately, at this stage - although it could also be a good subject for a later first crack at a Wimp version.

By “non-Wimp” we mean that, when run, the program will take over the computer completely, as far as the user is concerned, until the user exits the program - and user input will be via the keyboard only. Compare this with “Wimp” programs, which run within a desktop window - apparently simultaneously with other Wimp programs i.e. “multi-tasking” - and much of the user input is via the mouse/pointer.

Planning

It will sound like a sermon, but at the start of any programming exercise it does pay to use a pencil (or word processor) for a few moments - to make sure we know what we are trying to achieve.

So, for our Loan project, we need to note that it will concern four main inter-related factors:

loan amount

interest rate

repayment amount

number of repayments

and that, if **any three** of these are known, the fourth can be found by calculation.

Therefore, the aim is to produce a program which, as a bare minimum, allows a user to:

decide which of the four factors is to be the unknown;

input values for the other three (the “knowns”);

calculate and display the fourth value after the above two steps;

repeat the above steps if desired.

With this aim understood, we can now hazard an outline program structure something like:

```
Show a menu of the four main factors.
Let the user choose the unknown one.
Let the user input values for other three.
Do calculations to find value of unknown.
Output result (preferably enhanced with some graphics).
Offer user choice of repeating sequence or ending.
```

Jotting down a broad structure like this in note form before starting to type program lines is highly recommended for all programs. *(It is often referred to as 'pseudo-code' i.e. a shorthand English version of the eventual program.)*

The main advantage is that it forces us to look at the complete program from the start. It will highlight any major sequence problems and start to break the program up into logical segments.

Ideally, we want the program segments to be as self-contained as possible. For instance, we might want to **'Output result'** in several different ways (screen text, screen graphics, printer, file, etc.) and we don't want to have to go back and modify the **'Input values...'** segment each time we have a new output idea.

Our Loan project will use the above broad structure to build up the program sequentially in Chapters 3-15.

3.Variables

What are variables - Demonstration of use in initial Loan program - Declaring variables and assigning values to them ? The three variable types: integer number, real number and string - Null strings - Using them in programs - Incrementing numeric values - Variable names.

Variables play such an important role in programming that it is worth introducing them with no distractions.

So, Program Loan3a below - although it is also the first step in our Loan project - is a very limited and deliberately naive program, which we will rapidly convert in the next chapter. Its sole purpose is to introduce variables in a working program with the bare minimum of other topics.

So, open an ! Edit Basic window (see Chapter 1) and type in and save Program Loan3a. Then run it just to check it's all OK.

Even if you have the disc of the book's programs, it is recommended that you actually type in the listings at this stage. There really is no substitute for this powerful aid to early understanding - you will know when it ceases to have much value for you personally.

> *(Some single-statement program lines e.g. Lines 170 and 530, are shown on more than one printed line in the listing. This is inevitable and no problem. Simply type them as a single line and !Edit will automatically continue onto the next screen line as and if it becomes necessary - note that 'word wrap' does not occur when entering Basic programs. Where the line breaks actually occur will depend on your own configuration and it will probably differ from the listing below.)*

In ***Program Loan3a***, the only topics introduced out of order, so to speak, are an “error trap” at Line 40, and several occurrences of the keywords rem and TAB. All of these are part of the first group of Basic instructions covered in detail in Chapter 4. But, for now:

Bearing in mind that this book is for beginners and ***Program Loan3a*** is more than a few lines long, typing errors might well be a problem. Thus, when the program is run, Line 40 provides a simple ‘error trap’ which will put a helpful message on the screen saying which lines are faulty, if any are detected - and then cause the program to end. This will help you locate typing errors.

It only needs to be said, for the moment, that the computer will ignore anything after `REM`, on the same program line. So the `REMs` do not affect the program in any way - they provide a means of making comments to help with the understanding of the program.

The use of `TAB` in ***Program Loan3a*** is probably fairly self-explanatory - it acts something like the Tab key on a typewriter/wordprocessor.

Program Loan3a

```
10 REM> Loan3a
20 REM** Initial Loan program **
30 :
40 ON ERROR REPORT : PRINT " at Line " ; ERL : END
50 :
60 REM ***** Declaration of initial variables *****
70 :
80 Heading$ = "Loan Calculations" :REM** Declares
     string variable **
90 SubHeading$ = "(Simple Interest)" :REM** Declares
     string variable **
100 :
110 Menu1$ = "There are four factors:" :REM** Declares
     string variable **
120 Param1$ = "Loan Amount" :REM** Declares string
     variable **
130 Param2$ = "No. of Equal Payments" :REM** Declares
     string variable **
```

```
140 Param3$ = "Amount of Each Payment" :REM** Declares
string variable **
150 Param4$ = "Interest Rate" :REM** Declares string
     variable **
160 :
170 Menu2$ = "You need to give values for any 3 of
     these to find the 4th." :REM ** Declares string
     variable **
180 Menu3$ = "Please choose the unknown one:" :REM**
     Declares string variable **
190 :
200 REM *** Print menu on screen, using above string
     variables ***
210 :
220 PRINT Heading$ :REM** Prints variable 'Heading$' on
     screen. **
230 PRINT SubHeading$
240 PRINT :REM** Prints a blank line **
250 PRINT Menu1$
260 PRINT TAB(10) Param1$ :REM** Prints variable
     'Param1$' on screen, starting 10 character spaces
     from LH edge of screen. **
270 PRINT TAB(10) Param2$
280 PRINT TAB(10) Param3$
290 PRINT TAB(10) Param4$
300 PRINT
310 PRINT Menu2$
320 PRINT
330 PRINT Menu3$
340 PRINT : PRINT :REM** Prints 2 blank lines **
350 :
360 REM ***** User choices/input *****
370 :
380 REM** Choices are all built-in for this first
     example. **
400 Temp1$ = "The unknown factor is: " + Param3$ :REM*
     * New string variable created by adding two
     strings together **
420 LoanAmount = 1000 :REM** Declares a Real Numeric
     variable **
```

```
440 RatePercent = 8.5 :REM** Declares a Real Numeric
     variable **
460 NumberOfPayments% = 36 :REM** Declares an Integer
     Numeric variable **
470 :
480 REM ***** Calculation of unknown *****
490 :
500 REM** Simple Interest for first example only **
520 REM** Result of RH side of equation is put into new
     real numeric variable named on LH side of
     equation. The symbols "*" and "/" mean "multiply"
     and "divide" respectively. **
530 AmountPerPayment = (LoanAmount + (LoanAmount *
     RatePercent / 100) ) / NumberOfPayments%
540 :
550 REM ***** Output, prints results on screen *****
560 :
570 PRINT Templ$
580 PRINT
590 PRINT Paraml$ ; " = £" ; LoanAmount :REM** Prints 3
     items on same line **
600  PRINT     Param4$ ; " = " ; RatePercent ; "%"
610  PRINT     Param2$ ; " = " ; NumberOfPayments%
620  PRINT     : PRINT
630  PRINT     Param3$ ; " = £" ; AmountPerPayment
640
650  END
```

Run ADFS::HardDisc4.$.Articles.TheBook.LoanProgs.Loan3a

```
Loan Calculations
(Simple Interest)

There are four factors:
          Loan Amount
          No. of Equal Payments
          Amount of Each Payment
          Interest Rate

You need to give values for any 3 of these to find the 4th.

Please choose the unknown one:

The unknown factor is: Amount of Each Payment

Loan Amount = £1000
Interest Rate = 8.5%
No. of Equal Payments = 36

Amount of Each Payment = £30.1388889
```

ardDisc4 :0 Apps

Figure 3.1

Result of running program 'Loan3a'

Compare the above listing with what you see on the screen when you run the program (see Figure 3.1) - reading it in line number order.

Lines 80-180 create several ‘string variables’, and then Lines 220-340 print them on the screen as an opening menu.

To keep this first version of the Loan program simple, the choice of the unknown factor and the values of the other three knowns are made for us, in Lines 400-460, where examples of the remaining two types of ‘variable’ are introduced: two ‘real numeric variables’ and one ‘integer numeric variable’.

The rest of the program calculates the 4th factor and assigns the result to a third ‘real numeric variable’ - all in Line 530. Lines 570-630 then print a results table which repeats the ‘known’ values and then shows the calculated ‘unknown’ value.

As indicated above, ***Program Loan3a*** is a naive program for one purpose only. Use the listing as a reference when reading the more detailed section which now follows.

Variables

Basic programs consist, almost entirely, of operations on "variables". As far as the programmer and computer are concerned, a variable is a small amount of computer memory given a reference name by the programmer and used to store a 'value' during the run of a program. The variable's value does not necessarily need to be a number - it can be a string of letters, for instance.

To use the stored value for some purpose, the programmer merely refers to the variable's name in a Basic statement and the value currently being stored is substituted for the name when the program arrives at that statement.

'Declaring' variables

In a Basic program, a variable firstly needs to be 'declared' (created) by giving it a name and a value. After that, the variable can be used as much as you like in that program, including changing its value. (Some languages require you to declare variables at the start of a program, without assigning values.)

Fortunately, in BBC Basic, declaring a variable is almost transparent and is done automatically by simply assigning a value to the variable the first time it is mentioned - exactly as in lines 80-180, 400-460 and 530 of ***`Program Loan3a`*** i.e. it can be given a direct value (e.g. Lines 80 & 420), or it can be given the results of a calculation (e.g. Lines 400 & 530), which can include the values of **previously-declared** variables.

Taking Line 80 for example, the variable `Heading$` (a string variable) is declared and given a value `"Loan Calculations"` (a string). The other variables are declared similarly.

The best way to read a line such as Line 80 is "Assign the string `Loan Calculations` to the string variable `Heading$`" - or, more succinctly, "Let `Heading$` take the value `Loan Calculations`". *(See the description of the keyword LET in Chapter 4 also.)*

As indicated above, declaring a variable sets aside a space in memory to store the name you have given it plus an associated space to store the variable's value.

If the value of a declared variable changes through the course of a program, the contents of the memory space set aside for the value will change correspondingly - but the variable's name remains the same, of course. So, by referring to the variable name at any later time in the program, we get access to the particular value of that variable at the time

- most importantly by using the variable name in routines and formulae, very similar to algebra.

So, in Line 530, we have used:

```
AmountPerPayment = (LoanAmount + (LoanAmount *
      RatePercent / 100) ) / NumberOfPayments%
```

which is a Basic instruction saying "calculate the expression to the right of the equals sign, substituting the current values of the variables named, and put the result into the (new) variable `AmountPerPayment`". In the program, Lines 420-460 assign the values 1000, 8.5 and 36 to `LoanAmount`, `RatePercent` and `NumberOfPayments%` respectively. So, when Line 530 is arrived at during the program run, it will effectively become:

```
AmountPerPayment = (1000 + (1000 * 8.5 / 100) ) / 36
```

which, when calculated, results in the variable `AmountPerPayment` being assigned the value 30.1388889.

Or, as another example, in Line 220:

```
PRINT Heading$
```

which says "print out the value stored in the variable `Heading$`" - in this case the string `Loan Calculations` which was assigned to `Heading$` in Line 80 (and has not subsequently been changed).

From these examples you can see that, to carry out calculations, numeric variables can more or less be used as numbers themselves in algebraic-like expressions - linking them by +, -, *, / and many other arithmetic operators which we will come to later.

Programming logic versus algebra

There is one other very common way of handling variables that needs to be introduced early on.

Look at the following sequence:

```
Number% = 5
Number% = Number% + 6
PRINT Number%
```

The first line declares the numeric variable `Number%` and gives it a value of 5. But what happens in the second line?

If this was algebra it would certainly be nonsense - but it is perfectly logical if we remember that a variable is a memory location and we read the line as described earlier.

Thus, the second line says "Add 6 to the current value of the variable `Number%`, and assign the result to the variable `Number%`". Hence, `Number%` will be given the new value of 11 and the third line will print this as the result. *(See the description of the keyword LET in Chapter 4 also.)*

The right-hand side of the second line above could be any valid numeric operation and not just a simple addition.

This method of changing the contents of a variable (including a string variable, as we shall see) is probably the most often used and yet it can be very confusing to the beginner until it is explained.

Counting and incrementing

A particularly frequent need is to increase (or decrease) the value of a numeric variable by a fixed numeric amount - usually 1, e.g. a counting routine.

Because it is so common, Basic has a special arithmetic operator to do it. Instead of the previous second line, we could have used:

```
Number% += 6
```

to increase the contents of the variable Number % by 6, or:

```
Number% -= 6
```

to decrease the contents by 6.

Note that there must not be a space between the + sign (or - sign) and the = sign. This form is actioned more quickly by the Basic processor than the previous method.

Variable Types

In BBC Basic, three different types of variable are allowed, each used to store a particular type of 'value'. The three types are 'integer numeric', 'real numeric' and 'string' - and they are distinguished from each other by using unique symbols for the last character of the variable names of two of the types, as the following explains:

integer - An integer numeric variable can only store whole numbers (positive or negative). It must use '%' as the last character of its name. e.g. `integer%`, `Day%`, `aa%`, `AXE%`, `X%` could all be names of integer variables.

In ***Program Loan3a***, the variable `NumberOfPayments%` is our only integer variable. *(It's the only case in that program where we can be sure that only whole numbers will be needed - all other*

numbers might be either whole or fractional.)

The main advantages of using integer numeric variables (compared with real numeric variables) are they are processed faster, their accuracy is precise and they use less space to store their value. The range of allowable integer numbers is from +2,147,483,647 to -2,147,483,648 *(see Appendix 8 also)*.

(The 26 integer numeric variables formed by a single capital letter are known as 'resident integer variables' e.g. `A%, B%, C%,` *etc. They are stored at a special location and their values are not erased when a new program is run or loaded. They can therefore be used to pass simple data between Basic programs.)*

real - A real numeric variable can store any number, whether a whole number or one containing a decimal fractional part. It is also called 'floating point' because it is not fussy about how the decimal point is shown when you assign a value to it i.e. a real variable will recognise 1.2, or 1234.56, or 0.004567 equally happily.

A real numeric variable is distinguished by **not** having any special symbol in the last character of its name. So, `Real, vat, axe, pp` could all be real variable names. In the program three real variables are used - line 530 contains all three, plus the only integer variable.

Real variables are slower to process, are subject to 'rounding errors' in calculations & use more memory than integer variables to store their values. (For these reasons it is sensible to use integer variables whenever you can, although it is not often that you will notice any differences in speed.) The range of allowable real numbers is from $\pm 1.7\times 10^{38}$ to $\pm 1.5\times 10^{-39}$ in Basic V (Basic VI has a much larger range).

string - A string variable stores strings i.e. a sequence of letters and/or characters. It is distinguished by using the '$' symbol as the last character in its name. e.g. `a$, x$, day$` are all valid names for string variables. Our program uses several string variables.

The number of characters stored in a string variable is called the **length** of the string and the maximum length allowable is 255. A string variable can also store zero characters i.e. have zero length, and this is called a "null string". It is entered from the keyboard by typing a pair of double quotes with nothing in

between them.

For example:

```
String$ = ""
```

assigns a null string to `String$` - and note that this is not the same as:

```
String$ = " "
```

which has the (invisible) space character between the quotes and which therefore assigns a perfectly valid string (of length 1) to `String$`.

As this book progresses you will find that this simple pattern of three types of variable weaves its way into several Basic topics, including arrays, procedures/functions and files - so it is important to become familiar with it as early as possible.

Variable names

You will have seen already that this book favours the use of meaningful names for variables e.g. `Heading$` for the variable that holds the words to be used as a heading on the screen. We could have used something as stark as, say, `h$` instead - but there really is no contest when reading a long listing and trying to keep track of what is happening.

Nonetheless, there are a few penalties to pay for long variable names: more program storage space and processing time is used. However, storage space of the order involved is not usually a problem nowadays - and, anyway, there are several decent program compactors around which will alleviate both problems if need be, once you are happy with the program. So, stick with meaningful names.

You need to be aware that there are some restrictions on what characters can be used in variable names and fuller details are given in Appendix 2.

Variable names in this book

For variable names, this book will use mainly lower case letters - but normally with an upper case letter at the start of each ‘word’ in the variable. For example:

`NumberOfPayments%` (as already used in ***`Program Loan3a`***)

Apart from providing good readability, there is another reason for adopting this policy, which we will come to in the next chapter.

4. Keywords

The 'verbs' of Basic 'sentences' - Getting a list of keywords and their 'syntax' from computer - Keywords and variable names - Keywords LET, REM, PRINT and TAB(introduced and demonstrated in detail - Print 'modifiers' - Parameters and arguments - Keyword ERROR introduced - Use of ON ERROR in simple 'error traps' - Common error messages.

As we have said, the computer needs to translate the Basic programming language instructions into corresponding machine-code to carry them out. To do this there has to be no doubt whatsoever about the meanings of the 'words' of the programming language and how they are put together in 'sentences'.

In Basic, the 'words' are called 'keywords' and every Basic statement ('sentence') must have (or uniquely imply) at least one keyword - rather like every sentence in English must have a verb. The whole point of Basic is that these keywords closely match their English equivalents in meaning, making it easier to learn (for English speakers!).

Each keyword therefore has a rigid description of how it can be used: down to whether spaces, commas, semi-colons, etc. are important and even where it must sit in a 'sentence'. These essential details are called the 'syntax' - another straight copy from English.

To see some examples, enter Basic from a Task Window (see Chapter 1) and type:

`HELP .` (note the full stop)

This will produce a full list of Basic keywords on the screen - and typing:

`HELP <keyword>`

gives the syntax of any one of them. Type:

`*HELP syntax` (note the asterisk)

to get an explanation of the syntax symbols! *(Acorn's BBC Basic Reference Manual contains a full description of all keywords and many examples of how they are used.)*

You will not need to understand these screen lists at the moment, but it is important to know how to how to look up a particular keyword, to jog your memory as you build up competence.

Most keywords have an optional abbreviated form e.g. `P.` for `PRINT` (again, note the full stop) and these can certainly reduce finger wear when typing in programs. *They are listed in Acorn's BBC Basic Reference Manual.*

When you `LIST` the program in a Task Window (or save and reload it in !Edit) the full names will be shown, so legibility is not lost by using these abbreviations. *(At least one of the non-Acorn alternatives to !Edit has an option which picks up the abbreviations as you type them in and converts them immediately to the full keyword on the screen.)*

You'll probably be pleased to hear that this book will not be attempting to explain the 160 or so Basic keywords one by one. Instead it will introduce the main ones as they arise under topic headings. By the end of the book all the important ones for a beginner will have been covered in detail - and the Index will help you use the book for reference.

Keywords and Variable names

It has already been stressed that all Basic keywords are strictly in CAPITAL LETTERS ("Upper case") ONLY. If you refer to Appendix 2 you will see that one of the restrictions on variable names is that they must not start with a Basic keyword - otherwise the computer thinks the variable name is a keyword.

The obvious way to avoid running into this restriction unwittingly is simply to adopt a policy at the start of your programming life of **not** choosing variable names with all upper case letters - and this is the additional reason referred to at the end of the last chapter when the policy for naming variables in this book was stated.

Thus, in this book, keywords and variables will always contrast in print - which will help your understanding of the listings and also help to minimize your typing errors.

It is now time to start describing some specific keywords in more detail - and in this first batch we will include those which we have already used in our introductory program listings in Chapters 1 and 3.

Keyword LET

We are introducing this keyword first for the odd reason that we are never going to use it again!

The sole purpose of `LET` is to assign a value to a variable. Thus, Line 80 of ***Program Loan3a*** could have been written:

```
80 LET Heading$ = "Loan Calculations"
```

and in many Basic dialects you would need to do this - and it has to be said that writing it this way does help a little in emphasizing that a variable assignment is taking place rather than an algebraic equation.

However, in BBC Basic, `LET` is optional in nearly all cases - and must **not** be used in a few special cases of variable assignment. The result is obvious: it is now rarely used in BBC Basic - because that is the safest option!

So, we will not be using `LET`, but it is important to know it exists - and that it is the implied keyword missing from all our assignment statements.

Keyword REM

This ought to be the most often used of all keywords - and it probably is. It does nothing! Everything after it **on the same program line** is ignored by the computer.

It is therefore used to make remarks/comments *(REMarks, OK?)* about the program, and also to mark boundaries between segments of the program. ***Program Loan3a*** gives plenty of examples of both these uses e.g. Lines 60 and 80. It is also helpful to look at other people's listings, to see how they use `REMs`. Some quite eye-catching effects can be achieved, which aids their readability.

Get into the habit of adding `REMs` as you go, rather than leave them until later: otherwise you are sure to forget at least some of them.

`REM` is also often used to 'delete' a program instruction temporarily during development, to avoid having to re-type it if subsequently needed. Just type `REM` in front of the part to be deleted temporarily, and save the program again. Then remove the `REM` (and re-save) if you want to reinstate the instruction again later. A typical use of this during development is if you want to check the values of certain variables at a particular point in the program, by printing them on the screen, but will not need to do this when the program is complete. Just put a `REM` in front of the printing instructions when you no longer want to see the variables.

Unless you are really short of memory space, don't be tempted to delete temporary instructions permanently. If you had need for them once, you'll probably need them again sometime!

Finally, we have also already seen that using:

```
REM> ProgName
```

as the first line of a program will automatically save the program under the file name `'ProgName'` if we use `SAVE` in Basic command mode. It does not matter if there are spaces between the three parts of this statement e.g.

```
REM > ProgName
```

would work just as well.

Keyword 'PRINT' and TAB(

These are very common keywords and best introduced together. You are unlikely to author any program without using them.

`PRINT` is mostly used to write and format text strings and numbers on screen - but it actually sends characters to the currently configured 'output stream'. This means it can also send text to a printer or other output device - with or without also sending it to screen. *(Under Desktop operations, the preference is to use 'drag & drop' or the Wimp Message system to activate printers, rather than direct printing from Basic. However, in non-Wimp programs it is often helpful to use a little direct printing and a few words on this are included in Chapter 16.)*

Unless stated otherwise the following refers to printing on the monitor screen, which is the default situation i.e. the normal configuration in which the Basic processor starts itself.

When you run a program (or change/select a screen display mode) the text cursor is automatically first put at the 'home position' - which, in English, is the top left-hand corner of the screen i.e. where you would normally start to write on a page.

As you cause printing to appear on the screen, the cursor travels with the writing - always showing where the **next** printing action will commence. If you press `<return>` the cursor will jump to the start (left-hand edge) of the next line down - the 'line feed plus carriage return' action from typewriter days. *(When developing a program, get used to noticing where the cursor is on the screen when the program is waiting for some further action/input - or has just stopped unexpectedly! The cursor position often gives valuable clues*

when things do not go as expected.)

Therefore, the keyword `PRINT` instructs the computer to write something on the screen and the printing action will commence at wherever the text cursor happens to be when the instruction occurs.

The programmer can move the cursor and modify the format of what is printed - by means of several format-modifying symbols e.g. comma, semi-colon, single quote, etc. Some of these were used in ***Program Loan3a*** earlier - Line 600, for instance.

These 'print modifiers' generally alter the printed format (from the default format) **for one operation only**. It is therefore important to know the default arrangements and ***Program Prog4a*** demonstrates both the default situation and the effect of the most common modifiers.

Again, this is a naive program, seeking to minimise the use of new Basic instructions - and in this context you should not worry about the meaning of Lines 60 and 100 at the moment. The `REMs` should help to explain what is happening.

Program Prog4a

```
 10 REM> Prog4a
 20 REM** Some 'PRINT' Modifiers **
 30 :
 40 REM****************************************
 50 :
 60 MODE12 : REM** Can be any Mode with 80 characters
      width. **
 70 :
 80 ON ERROR REPORT : PRINT " at Line" ; ERL : END
 90 :
 100 COLOUR 132 : CLS : COLOUR 7
 110 :
 120 REM****************************************
 130 :
 140 REM** First declare some numeric and string
      variables of different lengths and formats. **
 150 :
 160 a% = 1
 170 b% = 22
 180 c% = 333
 190 d% = 4444
```

```
200 e = 555.555
210 f = .666666
220 g = 777777.777777
230 :
240 a$ = "A"
250 b$ = "BB"
260 c$ = "CCCCCCCCCCCC"
270 :
280 REM********* Opening screen ***************
290 :
300 PRINT STRING$(8,"0123456789") : REM** Prints
     numbers across screen to aid TAB counting **
310 PRINT : REM** Prints an empty line. **
320 :
330 REM———————————————
340 :
350 REM** All numbers right-justified on 10th character
     position i.e. on TAB(9) (default 'print field' is
     10) **
360 PRINT a%
370 PRINT b%
380 PRINT c%
390 PRINT e
400 Pause% = GET
410 :
420 REM********* Step 2 ***********************
430 :
440 PRINT
450 PRINT a% , b% , c% , d% , e , f , g : REM** Commas
     give 10-character column effect for numbers
     (default 'print field' is 10). Note changes with
     long numbers **
460 Pause% = GET
470 :
480 REM********* Step 3 ***********************
490 :
500 PRINT
510 PRINT a% ; b% ; c% ; d% ; e ; f ; g : REM** a%
     printed as above, but following semi- colons
     hold, cursor at end of each subsequent number -
```

```
     producing no spaces between them - 'print field'
     ignored. **
520 :
530 PRINT ; a% ; b% ; c% ; d% ; e ; f ; g : REM** As
     above but now semicolon in front of a% holds
     cursor at TAB(0)before printing first number. **
540 Pause% = GET
550 :
560 REM********* Step 4 **********************
570 :
580 PRINT
590 PRINT TAB(4) b%
600 PRINT TAB(14) b% : REM** TAB value effectively
     moves 10-character column ('print field') to
     right by 4 character spaces, keeping right
     justification **
610 Pause% = GET
620 :
630 REM********* Step 5 **********************
640 :
650 PRINT
660 REM** Strings left justified on TAB(0). Note effect
     if preceding string greater than column ('print
     field') width. **
670 PRINT a$
680 PRINT b$
690 PRINT c$
700 PRINT a$ , b$ , c$
710 Pause% = GET
720 :
730 REM********* Step 6 **********************
740 :
750 PRINT
760 PRINT a$ ; b$ ;
770 PRINT c$
780 PRINT c$ , a$
790 PRINT
800 PRINT TAB(14) b$ : REM** String left justified on
     TAB(14). **
810 Pause% = GET
```

```
820 :
830 REM********* Step 7 **********************
840 :
850 PRINT
860 PRINT a$ ' a% ' c$ : REM** single quote mark forces
     new line, but R/L justifications in 'print field'
     unaltered **
870 PRINT
880 Pause% = GET
890 :
900 REM********* Step 8 **********************
910 :
920 PRINT TAB(3,24) b$ : REM** TAB( with 2 numbers
     moves cursor to stated position, overwriting if
     already occupied. In this case overwrites part of
     previous line. **
930 :
940 END
```

This program runs in steps, with the following step waiting until you press any key on the keyboard. Run ***Program Prog4a*** now and examine the screen in conjunction with these notes, pressing any key to move to the next step:

Opening screen Firstly, a line of numbers is drawn on the top line (by Line 300) to help you count the character positions. This is followed by the results of Lines 360-390 i.e. four numbers, each of a different number of digits. You will see that each number is on a separate line and that the **right-most** digit always appears at the same position - in this case always under the first 9 in the top line. This is what is called ‘right justification’ and it can be applied to text as well as numbers.

Note that this result has been achieved by the simple use of `PRINT` without any modifiers i.e. no commas, semi-colons, etc. - so this result is the default output for numbers. Note also that the justification occurs at character position 9 i.e. the **tenth** character position from the left-hand edge. *(Get used to starting to count from zero - most programming counting does so.)* Again, this tells us that the default ‘print field’ is 10 characters wide and we shall see more of its effect below.

Step 2 You will now see the result of Line 450, which uses one `PRINT` command to display seven different numeric variables and uses commas to separate each variable in the instruction. As we are still using the default ‘print field’ (of 10 characters) the effect is to print the numbers in columns across the screen - with each number right justified in a separate ‘print field’. Note that the right-most character of each number is always underneath a 9 in the top line. This confirms that the column/ ‘print field’ width is 10 characters.

Step 3 This shows the effect of Lines 510 and 530. Here, the seven numeric variables are separated by a semi-colon instead of a comma. This holds the cursor at the right-hand end of each number after it has been displayed, so that the next number starts immediately - without any space. The semi-colons therefore override the default right justification of numbers and ignore the ‘print fields’.

You can see that the only difference between Lines 510 and 530 is the extra semi-colon before the first variable - which stops the first variable being right justified at character position 9.

Note what happens to the value of the variable `g` . It was declared as a number with 12 digits, but it is rounded to 9 digits to fit into the ‘print field’ of 10 - the decimal point counting as one character.

Step 4 This shows the effect of Lines 590 and 600, which both introduce the `TAB(` keyword. The single, opening bracket after `"TAB"` is deliberate, because it is actually part of the keyword - and it also serves as a good reminder never to put a space between the `"TAB"` and the `"("`.

The effect of `TAB(` differs according to whether there is one or two numbers in the brackets. If only one number is used then the effect is to move the cursor by that number of character spaces before the printing action commences. It does not change the printing action itself. So, in both Lines 590 and 600, which do not have any commas or semi-colons, the number is still printed right justified in a ‘print field’ 10 characters wide - but the column starts at TAB value 4 and 14 respectively (with the right-most character of the column at 13 and 23 respectively).

`TAB(` with one number therefore controls the horizontal

position of the cursor - counting in character positions across a line, starting at 0 at the left-hand edge. If the cursor is already further to the right of the TAB value you ask for, then a new line will be started first. Try it by changing the TAB value in Line 600 to 5. *(Change it. Save it. Run it. See the difference. Change it back. Re-save it! You'll find it quicker to use the keyboard shortcuts for bringing up the 'Save box'.)*

Step 5 This shows the effect of Lines 670-700, which print strings. Note that using commas in Line 700 again produces a column effect, as for numbers, but in this case the strings are left justified in the 'print field'. Hence, the default arrangement for strings is left justification in the 'print field'.

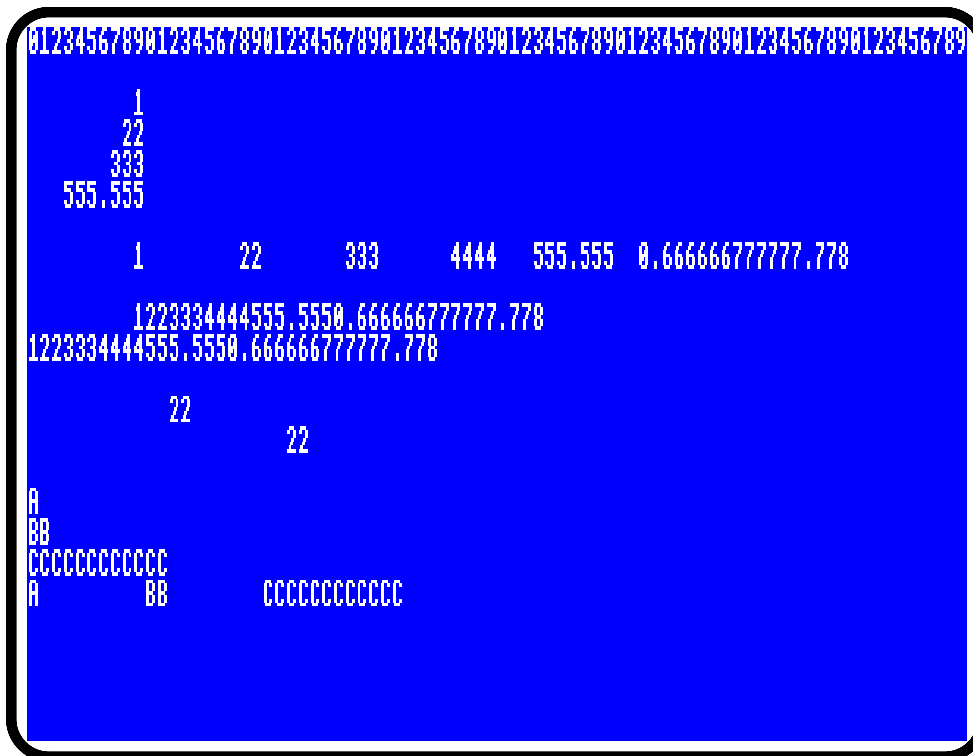


Figure 4.1

Screen shot of program 'Prog4a' after Step 5

Step 6 This shows, firstly, the effect of using a semi-colon, which is basically the same as for numbers. Note the final semi-colon in Line 760 and the consequential result of Lines 760 & 770 together. The result is the same as using the instruction `PRINT a$ ; b$ ; c$` and the same effect would have been obtained with numbers instead of strings i.e. a semi-colon keeps the cursor where the previous writing ended.

Secondly, Line 780 shows what happens if a string longer

than the column width/‘print field’ is printed - in this case `c$`. Note that the string is not curtailed (as it was, by rounding, in the number case) and that the following string, separated by a comma in this case, is left justified at the start of the next available column/‘print field’ - which in this case starts at TAB value 20.

Thirdly, Line 800 shows the effect of using `TAB(` with a single number for printing a string. Note that the string is left justified at TAB value 14 - and compare this with the effect of Line 600 which was right justified in a column 10 characters wide which starts at TAB value 14.

Step 7 This shows the effect of the single quote character in Line 860. It simply forces the printing to start a new line - but does not change the printing action i.e. strings and numbers still follow their default justifications (unless changed by some other modifier).

Step 8 This final step shows the effect of using TAB(with two numbers, in Line 920. The first number acts as before (i.e. horizontal position), but the second number specifies the **vertical** position of the start of the printing action. Counting for this vertical position starts at zero at the top of the screen. Thus `TAB(0,0)` is the top left-hand corner of the screen.

It is important to note that the printing action in this case **deletes and overwrites anything already on the screen at the defined location** - as is demonstrated in Line 920, which overwrites a portion of the printing already on the screen arising from Line 860.

It is worth noting here that `TAB(` instructions using one number will work with either screen or printer outputs - but using TAB(with two numbers will only work properly with screen output.

Finally, when using `TAB(` it is necessary to keep an eye on the screen display mode - some have more text lines and/or characters per line than others, which can lead to unexpected results. The User Guide has a useful list of screen modes (called ‘old style screen modes’ if you have a Risc PC) and we will come to this in more detail in Chapter 12.

Other features of `PRINT` will be covered as and when we come to them - but there is plenty to ‘play’ with in ***Program Prog4a*** to get to grips with the use of `PRINT` and its modifiers.

In passing, where a Basic keyword needs some other factor added to it to make it a complete statement, that other factor is called a "parameter" or an "argument". They are different. For example, `TAB(` *on its own would be meaningless and needs to be completed with, say,* `TAB( 6 , 8 )` *to make it valid - and the numbers 6 and 8, which are used to carry out the task but are not themselves changed, are referred to as the parameters of the* `TAB(` *keyword!statement. In contrast, when we come to Basic's built-in mathematical functions, we have to supply a value for the function to operate on e.g.* `SQR (16)` *to find the square root of 16. In this case, where the supplied value is the target of some change by the Basic function, the supplied value is called an argument. Similar phraseology is used in maths.*

Wrapping and Scrolling

Two related features which are worth noting at this stage can be explored by changing the values in ***`Program Prog4a`*** to see what happens towards the right hand side of the screen - and also the bottom - with both numbers and strings. You will see 'wrapping' and 'scrolling' occur i.e. the text automatically continues onto the 'next line' when it reaches the right-hand edge and, when the bottom of the screen is reached, the whole display seems to move upwards to make way for new text. The downside is that you lose the text at the top! Basic offers ways to modify things here and we will come to them later.

Keyword 'ERROR'

If the computer detects something wrong whilst it is churning away it will usually try to let you know by displaying an 'error message'.

There are many error messages and can they arise from several different levels within the computer. We deliberately induced one from the Command Line in Chapter 1, and (very) occasionally you might get one generated from deep down in the internal workings of the machine. Much more common are those generated by the programming language software or the application program currently running. In Wimp applications in particular, it is very easy for the programmer to incorporate custom-built error messages.

Some text editors - !Edit, for instance - go one better: they can detect some errors as you are typing the program lines and/or when you try to save your program. This all helps, but only certain types of error can be picked up by text editors. *(By the way, they only detect and warn you. It is up to you whether you correct it or not. They will not prevent you from carrying the error*

into the saved program.)

In programming, 'error' has a wider meaning than its normal English usage. It includes helpful warnings that are aimed to stop you doing things you may not want to do e.g. close an amended file before you have saved it.

Some errors are 'fatal' i.e. if they are only picked up during program run they will stop the running and you might even have to restart/reset the computer. 'Fatal errors' will almost certainly cause you to lose any unsaved data/typing and, by the well-known laws, will generally occur just at that one time that you thought you would risk carrying on without saving!

Our main concern in this book will be to harness the help of the Basic error message system to overcome problems in a running program while we are developing it - and the keyword `ERROR` is the means provided.

It is most often met with a second keyword on and the general form is:

```
ON ERROR <do something>
```

Compare this with our actual usage in ***Program Loan3a*** and ***Program Prog4a***, where we have used:

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

The `<do something>` here is:

```
REPORT : PRINT " at Line" ; ERL : END
```

which is very simple and means "report what the error is and at which program line." *(*`ERL` *acts as a special 'variable' which holds the line number where the last error occurred.)*

The `ON ERROR` instruction dictates how the running program will react if/when it detects something wrong **after this line**, i.e. it is "an error trap".

If you want a different error reaction later in the program you can introduce another `ON ERROR` line which will supersede the previous one.

When an error is detected in the running program, the program will jump from wherever it is back to the `ON ERROR` line **last encountered** and carry out the instructions there before trying to carry on.

Generally speaking, while developing a program, we want to stop the program (in an orderly fashion) after we have read the error message - so that we can correct it before trying again. Otherwise we are likely to run into the same error again and again in an endless loop. This is why our error traps so far have used a short multi-statement `ON ERROR` line with the keyword `END` as the last instruction. The meaning of `END` is exactly

the same here as in Line 940 - it is quite acceptable to have more than one possible end point in a program.

Common typing error messages

To gain familiarity with some of the more common error messages it is a good exercise to introduce some deliberate mistakes into (a copy of!) Program Loan3a. *(In the following examples, ignore any warnings that your text editor might give you - we want the errors to occur in the running program.)*

For example:

a) Delete one (then both) quote marks from a string declaration. When run, you will get the error messages "`Unknown or missing variable`" or "`Missing ""` ", depending from which end you delete the quote mark. Without quote marks at the start of a string, the processor thinks it has met another variable - with the name of the first word of the string - but one that hasn't been declared. So it tells you "`Unknown or missing variable`". Had you, coincidentally, already declared a numeric variable with that name, the error message would have been "`Type mismatch, string needed`". If you delete the quotes only at the end of the string the processor easily detects that and accurately identifies the error as "`Missing ""` ".

b) Assign a string to a numeric variable. This is straightforward: the error message is "`Type mismatch, number needed`".

c) Miss out a colon before a REM etc. Depending on where the colon is omitted, the error message will probably be "`Unknown or missing variable`" or "`Syntax error`". From a) above you should be able to see why the former has occurred - and, if it is the latter, it means (somewhat vaguely) that something isn't right in the line (!). *(This is another powerful reason to try to keep one Basic statement per line. Your search area for the cause of an error becomes very much more limited.)*

The "`Unknown or missing variable`" error message occurs with several sorts of typing errors, particularly if you type two items and miss out a colon, semi-colon or comma between them. With familiarity you'll get used to the most likely reasons for the various messages.

Finally, assign a real number to an integer variable and re-run the program. You will not get an error message. The variable is assigned with only the integer part of the real number i.e. the part to the left of the decimal point. This is probably a sensible compromise, but you need to

be aware of it. It can result in a program apparently working OK but giving wrong results, and it can be a real pain to trace. *(It provides a good reason always to check your program with both real and integer inputs, against calculated answers.)*

Although this chapter has not added anything new to our Loan program listing, we have nonetheless come a long way again in covering some fundamental points. We can now use a few of the most common Basic keywords and read and understand Basic listings containing them.

Our program examples so far have been deliberately designed to introduce these fundamentals without too many other distractions. However, we would soon find it distinctly unhelpful if we carried on and tried to develop the program without employing Procedures and Functions, which are the subject of our next chapter.

5. Procedures and Functions

What they are - Two main uses - DEF PROC/FN described with examples - Where DEFs are placed in program listings - Calling PROC/FNs demonstrated - PROC/FN names - Formal parameters and passing parameters demonstrated - PROC/FN as structural aids - Empty PROCs - Upgrading Loan program with PROC structure - Initial brief introduction to Screen modes, colours, menu formatting and pausing a program - First use of upgrade listing for Loan program.

Procedures (`PROCs`) and Functions (`FNs`) are arguably the most powerful constructions provided by BBC Basic. They are very similar to each other but are used for slightly different purposes. They warrant early introduction in this book because a program shouldn't proceed very far without using them.

Procedures (PROCs)

A Procedure (`PROC`) is simply a user-defined programming routine identified by a unique reference name i.e. a sequence of Basic instructions chosen by the programmer for some purpose and given its own reference name so that the routine can be called into use in the program.

Procedures have two main uses: the first is for routines which are going to be used repeatedly during a program (or used in more than one program) - and the second is as a structural aid to sound programming.

The 'mechanics' of using `PROCs` involves two steps: firstly, a `PROC` must be defined (which includes giving it a name) and then the `PROC` is brought into action (as many times as you like) simply by 'calling' the `PROC` by name whenever and wherever it is needed. Let's give some examples.

Firstly, the PROC definition format is:

```
DEF PROCprintDistance
Message1$ = "Overall journey = "
Message2$ = " miles"
Distance = 10
PRINT TAB(5) Message1$ ; Distance ; Message2$
ENDPROC
```

These are normal Basic program lines. Each `PROC` definition must start with a line such as `DEF PROCprintDistance` and end with the line `ENDPROC`. Here, the reference name given to the `PROC` is "`printDistance`". There need not be a space between the two keywords `DEF` and `PROC`, but there must not be a space between `PROC` and `printDistance`.

The lines between `DEF PROCprintDistance` and `ENDPROC` contain the programmer's required sequence of Basic instructions, which will be carried out each time that the `PROC` is 'called'.

To call this `PROC` in a program, we simply use the Basic statement `PROCprintDistance` wherever we need to.

When, during the program run, the statement `PROCprintDistance` is encountered, the instructions in the `PROC` definition will be carried out before continuing on. In other words, calling a `PROC` causes the program to divert from its line number sequence temporarily - jumping to the corresponding `DEF PROC` line and carrying out the instructions found there - then jumping back to the statement **after** the one that called it. (If you use multi-statement lines, the statement following the `PROC` call may still be on the same program line.)

Program Prog5a shows a complete (albeit trivial) program using the above `PROC` - purely to show in detail how the `PROC` is called and where a `DEF PROC` is usually located in a program listing:

Program Prog5a

```
10   REM> Prog5a
20   :
30   PRINT "Trivial Program"
40   PRINT
50   PROCprintDistance
60   PRINT
70   PRINT "END"
80   END
90   :
```

```
100 DEF PROCprintDistance
110 Message1$ = "Overall journey = "
120 Message2$ = " miles"
130 Distance = 10
140 PRINT TAB(5) Message1$ ; Distance ; Message2$
150 ENDPROC
```

The program main structure finishes at Line 80 and the single DEF PROC is listed afterwards.

When run, the program, as usual, starts carrying out the instructions in line number order. However, on reaching Line 50 (which calls the PROC) the program jumps to the DEF PROC at Line 100 and carries out the instructions from there onwards - until it meets the ENDPROC at Line 150. It then jumps back to the instruction following the one that called it i.e. jumps back to Line 60 - and carries on until it ends at Line 80. The screen output will therefore be as in Figure 5.1.

Most programs will, of course, have many PROCs and their

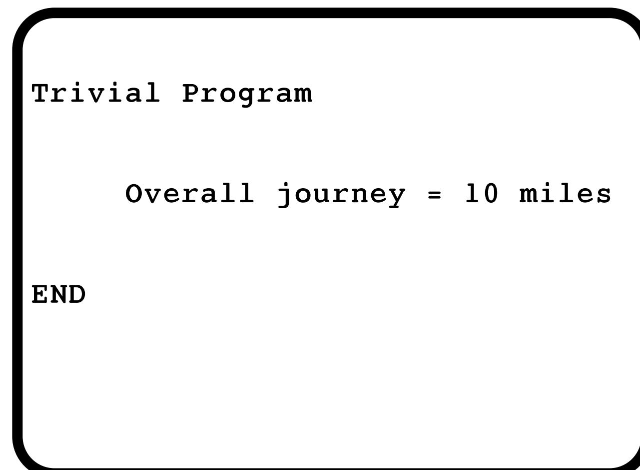


Figure 5.1

Result of running 'Prog5a'

corresponding definitions are simply entered in the same format as Lines 100-150 above - one after the other. From the computer's viewpoint it doesn't matter what order the DEF PROCs are entered - but it helps you, the programmer, to enter them in a logical order.

PROC names

PROC names are similar to variable names, but a little less restricted - but don't add "%" or "$" to them. They can start with a digit or a keyword, for instance. Appendix 2 gives the details. Again, there is no length restriction and it pays to name them meaningfully. We will adopt the naming policy already described for variables, except that we'll always start a PROC name with a lower case letter - to contrast it with the

keyword `PROC` itself - as in `PROCprintDistance` above.

If you need to, you can use the same name for a `PROC` and for a variable e.g. `PROCdate` and `date` - they will be treated as separate entities.

PROCs with formal parameters

The use of `PROCs` is extended considerably by 'parameter passing'. We could change our earlier example to:

```
100 DEF PROCprintDistance( Distance )
110 Message1$ = "Overall journey = "
120 Message2$ = " miles"
130 PRINT TAB(5) Message1$ ; Distance ; Message2$
140 ENDPROC
```

In the `DEF PROC` line, `Distance` is called a "formal parameter" and can be any of the three variable types, complete with `%` or `$` name endings if appropriate. (Here, it is a real numeric variable and therefore doesn't have a special ending symbol.) There must not be a space between the end of the `PROC` name and the opening bracket.

This `PROC` is then called by using, say, `PROCprintDistance( 22.6 )` somewhere in the program - causing the value 22.6 to be substituted for Distance ("passed to the `DEF PROC`") when the `PROC` call is actioned.

So, if we used this new `PROC` definition in ***Program Prog5a***, Line 50 would also need to be changed to, say, `PROCprintDistance ( 22.6 )` and the output message on screen would change correspondingly to that in Figure 5.2.

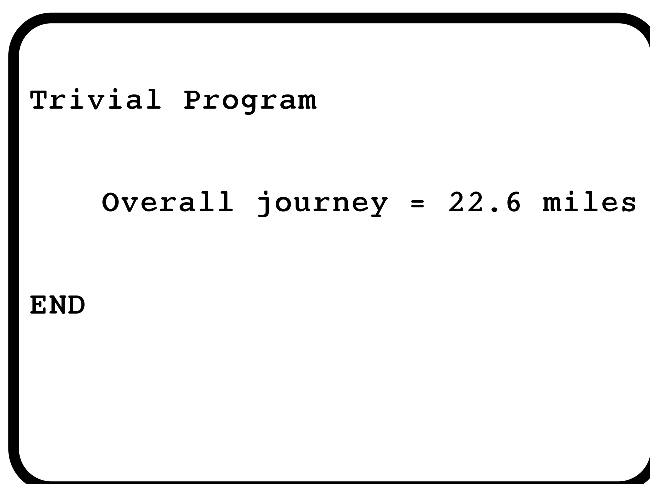


Figure 5.2

Alternatively, we could call the `PROC` by, say, `PROCprintDistance( Number )` and this time the value held in the variable `Number` at the time of the call is passed to the `DEF PROC` and duly inserted in the screen message. (Clearly, `Number` needs to be declared sometime before the `PROC call.`)

The big advantage of using formal parameters is that we can call the same `PROC` any number of times - with different parameter values each time, if we wished.

There is no practical limit to the number of formal parameters you can use (a dozen or more is quite common in Wimp applications). When more than one is used they are separated by commas in the `DEF PROC` line. For example:

```
DEF PROCPROCName(Param1% , Param2 , Param3$ , etc....)
```

As you can see, any mixture of the three variable types can be used, but you must make sure you use the same types of values/variables and the same order when you call the `PROC` - or errors will occur which will not necessarily cause error messages. We will be using formal parameters in our Loan program soon.

PROCs as structural aid

The second use of `PROCs` is as a structural aid to programming. You'll recall that ***Program Loan3a*** occurred before `PROCs` had been introduced. Therefore it was deliberately written without them. Now let's revisit that program and re-write it with `PROCs`. The main structure of the program might then be reduced to a sequence consisting entirely of `PROC` calls, something like:

```
ON ERROR PROCerror
PROCsetUpVariables
PROCmenu
PROCchooseUnknown
PROCinputKnowns
PROCcalc
PROCresults
END
```

and all the program instructions we have in Program Loan3a could be carried within one or other of the corresponding `DEF PROCs`. In this way - and note the use of meaningful `PROC` names - the main program structure can be seen better and we can change the structure or the detail much more simply. We will shortly show the conversion of ***Program Loan3a*** to this form.

(There is a further extension on the use of parameters in PROCs available in Basic V, but we do not need to cover it in this book.)

Functions ("FNs")

Everything we have said above about `PROCs` applies equally to `FNs` - except in the final line of their definition, which reflects their slightly different purpose.

A `FN` 'returns' a value (real, integer or string) to the Basic statement which called it. Thus, the calling statement needs to be correctly constructed to receive this - including matching the type of value being returned.

Again, an example will clear up any confusion. A typical, albeit short, `FN` definition - in this case using one formal parameter - might look like:

```
DEF FNkilometresToMiles( Kilometres )
      = 0.6214 * Kilometres :REM** Remember, the "*"
      symbol means "multiply". **
```

and would be called by a statement such as:

```
Miles = FNkilometresToMiles( 33.7 )
```

When this statement is actioned, the value 33.7 is passed to the `DEF FN` and the resulting converted value of 20.94118 will be returned by the `FN` - and in this case assigned to the variable `Miles`. In other words, this particular `FN` converts kilometre values into their equivalent in miles.

If you examine the above simple calling statement you can see that it is the same as a normal declaration of (or assignment to) the real numeric variable `Miles` - except that the part to the right of the "equals" sign, `FNkilometresToMiles ( 33.7 )`, has directly replaced the real number (or the real numeric variable) being assigned. In fact, it is easiest to remember that `FNs` can be directly substituted anywhere in a Basic statement that you could use a variable (of the corresponding type) - and in nearly every place that you could use a direct value (again of the corresponding type).

The last line of a `DEF FN` must, as shown above, always start with an "equals" sign followed by the result that is being 'returned' to the calling statement. (An "equals" sign used like this doesn't look quite right at first arid may take a bit of getting used to. It may help to remember that the result is usually going to be assigned to a variable.)

`FNs` often contain just as many formal parameters and lines of action as `PROCs`, but they must always end up by returning a value. Don't forget

that a value can be a string.

Further points

The definitions of PROCs and/or FNs can themselves call other PROCs and/or FNs. For example, having defined DEF FNkilometresToMiles(Kilometres), we could now modify our earlier PROC definition to be:

```
 100 DEF PROCprintDistance(Kilometres)
110 Message1$ = "Overall journey = "
120 Message2$ = " miles"
130 PRINT TAB(5) Message1$ ;
    FNkilometresToMiles(Kilometres) ; Message2$
140 ENDPROC
```

As already shown, when writing a program the DEF PROC/FNs are always put after the main program structure, with an END statement usually serving as the separator e.g. Line 80 in ***Program Prog5a*** above. As the Basic processor tries to follow any program line by line in number sequence, most programs would simply continue to step straight into the DEFs (which would then be treated as part of the natural program sequence) unless the main program is stopped by an END statement. For example, without Line 80 ***Program Prog5a*** would continue to step through from Line 70 to Line 100 and beyond.

A DEF (but only that line) is treated as a REM if a program comes across one other than by use of a PROC/FN call.

One of the features of PROCs (not shared by FNs) is that you can define them with nothing between DEF PROC and ENDPROC and they will not interfere with the flow of your program while developing it. Empty PROCs may sound a pointless idea, but it definitely helps to get the main structure in place and it reminds you that you've still got something to do. In large programs it can be a great help, particularly when you have to leave the programming for a while.

We have not finished our description of PROC/FNs yet. There are some further points about their construction and use which are very important - particularly in Wimp programming - but they can safely be left until later in this book (Chapters 15 and 22).

Upgrading Program Loan3a

With the above new topics under our belt we can use them to upgrade ***Program Loan3a***.

First upgrade

The first upgrade is an interim one *(so do not bother to type this listing)*. It puts in place a main structure of PROCs (similar to the example earlier) and transfers all the program material - but not all the REMs - from ***Program Loan3a*** into the corresponding DEF PROCs.

This, and only this, is done in ***Program Loan5a***, solely to concentrate on the way PROCs are listed and called without added distractions. Use it only to compare with ***Program Loan3a***.

Program Loan5a

```
10 REM> Loan5a
20 REM** Loan3a program converted into PROC structure.
     No other changes **
30 :
40 ON ERROR PROCerror : END
50 :
60 PROCsetUpMenuVariables
70 :
80 PROCmenu
90 :
100 PROCchooseUnknown
110 :
120 PROCinputKnowns
130 :
140 PROCcalculations
150 :
160 PROCdisplayResuits
170 :
180 END
190 :
200 REM******************************************
```

```
210 REM****************************************
220 :
230 DEF PROCerror
240 REPORT
250 PRINT" at Line " ; ERL
260 ENDPROC
270 :
280 REM****************************************
290 :
300 DEF PROCsetUpMenuVariables
310 Heading$ = "Loan Calculations"
320 SubHeading$ = "(Simple Interest)"
330 :
340 menul$ = "There are four factors:-"
350 Paraml$ = "Loan Amount"
360 Param2$ = "No. of Equal Payments"
370 Param3$ = "Amount of Each Payment"
380 Param4$ = "Interest Rate"
390 :
400 menu2$ = "You need to give values for any 3 of
     these to find the 4th."
410 menu3$ = "Please choose the unknown one:"
420 :
430 ENDPROC
440 :
450 REM* ***************************************
460 :
470 DEF PROCmenu
480 PRINT Heading$
490 PRINT SubHeading$
500 PRINT
510 PRINT menul$
520 PRINT TAB(l0) Paraml$
530 PRINT TAB(l0) Param2$
540 PRINT TAB(l0) Param3$
550 PRINT TAB(l0) Param4$
560 PRINT
570 PRINT menu2$
580 PRINT
590 PRINT menu3$
```

```
600 PRINT:PRINT
610 :
620 ENDPROC
630 :
640 REM****************************************
650 :
660 DEF PROCchooseUnknown
670 REM** Choices made for first example. **
680 :
690 Temp1$ = "The unknown factor is: " + Param3$
700 :
710 ENDPROC
720 :
730 REM****************************************
740 :
750 DEF PROCinputKnowns
760 REM** Choices made for first example. **
770 :
780 LoanAmount = 1000
790 :
800 RatePerCent = 8.5
810 :
820 NumOfPayments% = 36
830 :
840 ENDPROC
850 :
860 REM****************************************
870 :
880 DEF PROCcalculations
890 REM** Simple Interest for first example only **
900 :
910 AmountPerPayment =
     (LoanAmount+(LoanAmount*RatePerCent/100))/
     NumOfPayments%
920 :
930 ENDPROC
940 :
950 REM****************************************
960 :
970 DEF PROCdisplayResults
```

```
980 :
990 PRINT Templ$
1000 PRINT
1010 PRINT Param1$;' = £";LoanAmount
1020 PRINT Param4$;" = ";RatePerCent;"%"
1030 PRINT Param2$;" = ";NumOfPayments %
1040 PRINT:PRINT
1050 PRINT Param3$;" = £";AmountPerPayment
1060 :
1070 ENDPROC
```

Perhaps the main thing you will notice in comparing ***Program Loan3a*** with ***Program Loan5a*** is that it has become a lot longer - for no difference in output - and much of that increase in length comes from 'decorative' `REMs` and empty lines.

There is no way round this and you should not worry unduly about it. It is much more important that you can follow a program clearly, particularly when you leave it for a while and come back to it. It is also worth pointing out that the proportion of these extra lines does tend to fall as the program gets longer, so this type of 'overhead' always seems worse at the start.

Second upgrade

The second upgrade is Program Loan5b, listed below - which is the first 'proper' version of the Loan program.

This makes a substantive change in the program by putting a better looking menu on the screen, which involves four activities:

screen mode choice

colours

better formatting of menu items

pausing the program

Some of these require the use of a few Basic keywords and/or instructions which we have not yet introduced. So, after the listing, some brief preliminary comments are made about them which will be sufficient until they are covered in more detail in later chapters.

Note that although ***Program Loan5b*** *has been listed in full, its line numbers do not run consecutively - so check your typing of the line numbers very carefully - and do not use any automatic re-numbering facility!*

This non-consecutive line numbering occurs because ***Program Loan5b*** *is the real starting point for the Loan program - and the line numbers used in it are those which will normally be carried through to the final version several chapters ahead. As we develop the program we will gradually fill in the missing lines,* ***so do not renumber any lines in the Loan programs from now on!*** *This method suits the tutorial nature of this part of the book because:*

a) it allows us to make references to lines and/or their contents uniquely - maybe several chapters later than they first appear;

b) additions/changes can be introduced using ***update*** *listings without the need to renumber the complete new program after each change.*

So here is the effective starting version of our Loan program:

Program Loan5b

```
10 REM> Loan5b
20 REM** Developed from Loan5a. Effective starting
     version of Loan program. **
30 :
40 MODE 12
50 :
60 ON ERROR PROCerror : END
70 :
80 PROCinit
90 :
100 PROCsetUpMenuVariables
110 :
130 PROCmenu
140 pause% = GET
150 PROCchooseUnknown
160 :
180 PROCinputKnowns
190 :
200 PROCcalculations
210 :
220 PROCdisplayResults
260 :
270 END
280 :
290 REM****************************************
```

```
300 REM***************************************
310 :
320 DEF PROCerror
330 :
350 REPORT
360 PRINT" at Line " ; ERL
370 :
380 ENDPROC
390 :
400 REM***************************************
410 REM***************************************
420 :
430 DEF PROCinit
440 :
450 PROCcolourDefs
460 COLOUR TextScreenBackgroundCol% : CLS :REM** Sets
     text screen background to blue and clears it to
     that colour **
470 COLOUR NormalTextCol% :REM** Sets text to white **
500 :
510 Offset1% = 10 : Offset2% = 15
520 :
530 ENDPROC
540 :
550 REM***************************************
560 :
570 DEF PROCcolourDefs
580 :
590 TextScreenBackgroundedCol% = 132 : REM** Blue **
600 NormalTextCol% = 7 :REM** White **
610 EmphasisTextCol% = 3 :REM** Yellow **
620 ActionTextCol% = 2 :REM** Green **
710 :
720 ENDPROC
730 :
740 REM***************************************
750 REM***************************************
760 :
770 DEF PROCsetUpMenuVariables
780 :
```

```
790 Heading$ = "Loan Calculations" :REM** String
     Variable **
800 :
810 Menu1$ = "There are four factors:-" :REM** String
     Variable **
820 Param1$ = "Loan Amount               (L)" :REM**
     String Variable **
830 Param2$ = "No. of Equal Payments  (N)" :REM**
     String Variable **
840 Param3$ = "Amount of Each Payment (P)" :REM**
     String Variable **
850 Param4$ = "Interest Rate             (R)" :REM**
     String Variable **
860 :
870 Menu2$ = "You need to give values for any 3 of
     these to find the 4th."
880 Menu3$ = "Please choose the unknown one (L/N/P/R) "
900 :
910 ENDPROC
920 :
930 REM****************************************
940 REM****************************************
950 :
960 DEF PROCmenu
990 :
1000 PRINT:PRINT:PRINT
1010 PROCcentrePrint(Heading$)
1020 :
1030 PRINT:PRINT
1040 PRINT TAB(Offset1%) Menu1$
1050 PRINT
1060 COLOUR EmphasisTextCol%
1070 PRINT TAB(Offset1%+Offset2%) Param1$
1080 PRINT TAB(Offset1%+Offset2%) Param2$
1090 PRINT TAB(Offset1%+Offset2%) Param3$
1100 PRINT TAB(Offset1%+Offset2%) Param4$
1110 COLOUR NormalTextCol%
1120 PRINT
1130 PRINT TAB(Offset1%) Menu2$
1140 PRINT:PRINT
```

```
1150 COLOUR ActionTextCol%
1160 PRINT TAB(Offset1%) Menu3$
1170 :
1180 COLOUR NormalTextCol%
1190 PRINT:PRINT
1200 ENDPROC
1210 :
1220 REM****************************************
1230 :
1240 DEF PROCcentrePrint(String$)
1270 :
1280 ScreenWidth% = 80
1290 Tab% = (ScreenWidth%-LEN(String$)) DIV 2
1300 PRINT TAB(Tab%) String$
1310 :
1320 ENDPROC
1330 :
1340 REM****************************************
1350 REM****************************************
1360 :
1370 DEF PROCchooseUnknown
1380 REM** Choices made for first example. **
1390 :
1400 Temp1$ = "The unknown factor is: " + Param3$
1430 :
1440 ENDPROC
1590 :
1600 REM****************************************
1610 REM****************************************
1620 :
1630 DEF PROCinputKnowns
1640 REM** Choices made for first example. **
1650 :
1660 LoanAmount = 1000 :REM** Real numeric Variable **
1670 :
1680 RatePerCent = 8.5 :REM** Real numeric Variable **
1690 :
1700 NumOfPayments% = 36 :REM** Integer numeric
     Variable **
2160 :
```

```
2170 ENDPROC
2540 :
2550 REM****************************************
2560 REM****************************************
2570 :
2580 DEF PROCcalculations
2590 REM** Simple Interest for first example only **
2600 :
2610 REM** Result is put into new real numeric variable
     **
2620 AmountPerPayment = (LoanAmount + (LoanAmount *
     RatePerCent / 100)) / NumOfPayments%
2630 :
2680 ENDPROC
2690 :
2700 REM* * **************************************
2710 REM* * **************************************
6240 :
6250 DEF PROCdisplayResults
6260 PRINT Templ$
6270 PRINT
6280 PRINT Param1$ ; " = £" ; LoanAmount
6290 PRINT Param4$ ; " = ' ; RatePerCent ; "%"
6300 PRINT Param2$ ; " = " ; NumOfPayments%
6310 PRINT:PRINT
6320 PRINT Param3$ ; " = £" ; AmountPerPayment
6330 ENDPROC
```

As was said earlier, changes have been made under four headings to produce ***Program Loan5b*** from ***Program Loan5a*** (apart from the line numbering aspect) and these only require a brief explanation at this stage. Taking each heading in turn:

Screen mode

The User Guide describes the basic features of different screen modes sufficiently for our immediate purposes, although we will be returning to the subject in later chapters. Screen modes allow the programmer to choose the best combination of screen resolution, number of colours, text

lines and characters per line etc. In ***Program Loan5b*** `MODE 12` has been chosen (Line 40) because it is the lowest number mode giving 80 characters per line and 16 colours - which most Risc OS computers will be able to use. Any other screen mode with these features will do equally well.

Changing screen mode actually does quite a number of things and can affect how we program. So it is normal practice in non-Wimp programs to set the screen mode very early in the main program structure. Hence, `MODE 12` is the new first instruction of the program.

Colour

We will say a lot more on colour later (Chapter 21), but for now it only needs to be noted that text and graphic colours are handled separately, and the keyword `COLOUR` is the one used to set text colours.

It is helpful to try to get some uniformity in the use of colour over different programs. *(In Wimp there is pressure - and much help - to follow the Acorn 'Style Guide', so it is easier there.)* Therefore, it normally makes sense to define a set of colours at the start of a non-Wimp program, using descriptive variable names for the colours. We then simply use these variable names in the rest of the program to pick the 'pre-set' colours when we want them.

A sensible place to define the colours is in a separate `PROC` definition and, therefore, in `DEF PROCcolourDefs` at Line 570 we have started by defining four colours: a screen background colour plus three different text colours - for the three different purposes indicated by their variable names. Don't worry about the particular numbers assigned to these variables - just accept for now that they mean the colours shown in their associated `REM`s. (Using a `PROC` also means that we can more easily use the same colour definitions in other programs.)

Having defined the colours in `DEF PROCcolourDefs`, we need to call this `PROC` to set up those colour variables. The best place to do this is from inside a general 'initiation' `PROC`, named `PROCinit`.

You'll find a `PROCinit` in many programs. It's a good habit. You call it at, or very near, the start of a program (here Line 80) to set up anything which applies to the program as a whole - a set of colours is a good example. It is also frequently used to reserve memory space for special purposes, as we will come to later. So, a `DEF PROCinit` has been introduced into ***Program Loan5b*** at Line 430 and `PROCcolourDefs` is called from Line 450 within it.

You will also see that two 'offset' variables have been included in `PROCinit`. These variables will be used as the horizontal TAB values when printing the initial menu on the screen (see `PROCmenu`). As the program will need more than one menu (or similar) it will make sense to try to make them look consistent on the screen - hence the 'offset' variables are likely to be useful throughout the program - and therefore conveniently placed in `PROCinit`.

Menu formatting

For neater menus, there is usually a need to centre some text on the screen and/or to be able to line it up at desired TAB positions. This sort of process is ideally suited to a `PROC` or `FN`.

Therefore, within `PROCmenu`, `PROCcentrePrint(String$)` is called, at Line 1010.

It is defined at Line 1240 and it is probably reasonably clear what happens. The string to be centred on the screen is passed as the only formal parameter and its length is found using the keyword `LEN` (later!). Half this length is subtracted from half the screenwidth (80 characters per line in our chosen mode) and the string is printed starting at that calculated TAB position. Don't worry for the moment about the keyword `DIV` - reading it as "divided by" will be good enough for now - *but see Chapter 11 if you cannot wait!*

Pausing

After we have put our nice new menu on the screen it would be best if we paused the program to check that the menu looks OK - particularly as we know we are going to have to make many changes to what comes after the menu. The keywords `GET` and `GET$` are the simplest way to make a non-Wimp

program pause until you want it to continue - but they should not be used in Wimp programs for this purpose. We'll cover these keywords in more detail in the next chapter, but for the moment we will use the simple instruction `pause% = get` at Line 140, which will hold things on the screen until you press any key.

When you do press a key you will see more or less the same display as occurred in the last half of ***Program Loan3a*** - see Figure 5.3. We will not be keeping this much longer, but it seemed best to leave it for this upgrade, to provide some reassuring continuity.

Figure 5.3

The menu from program 'Loan5b'

We have again come a fair way in this chapter - indeed, you may want (need!) to go through it more than once. Although our Loan program still doesn't go very far, it now begins to travel in acceptable style. ***Program Loan5b*** *- our base-line version for all further updates - has a good* `PROC` *structure in place and we can now afford to concentrate more on fleshing them out. You will therefore find we will seem to move a little faster from now on.*

6. The REPEAT and WHILE control loops

User selection from screen menu - Control loops introduced - REPEAT ... UNTIL and WHILE ... ENDWHILE loops explained in detail and compared - Exit and entry conditions - First use of TRUE/FALSE - Nested loops - Keywords INSTR(, CHR$, ASC, GET and GET$ introduced and incorporated into Loan program - Forcing a valid keypress - Masking to upper/lower case - INKEY and INKEY$ briefly introduced as alternative to GET/GET$.

In our Loan project we have put an opening menu on the screen and now need to make arrangements to let the user select from it.

Selection from a menu of choices is a very common programming requirement. In non-Wimp programs, one of the simplest ways to do this uses the following new keywords in combination:

```
REPEAT ... UNTIL
INSTR(
CHR$
GET (or GET$)
```

and this chapter is going to introduce these in detail before using them to upgrade our Loan program.

In two of the cases it is logical also to introduce companion keywords at the same time - and thus we start our description of `REPEAT ... UNTIL` under a wider heading.

Control loops

'Control loops' are programming constructions which perform a sequence of Basic statements repeatedly until (or unless) some predetermined condition is encountered. BBC Basic has three different types available, which are:

```
REPEAT ... UNTIL
WHILE ... ENDWHILE
FOR ... NEXT
```

In this chapter we will cover only the first two of these, which are conveniently regarded as a complementary pair. (We cover the `FOR ... NEXT` loop in Chapter 9)

REPEAT... UNTIL loop

The general nature of a `REPEAT ... UNTIL` loop is a 'sandwich' of Basic statements between a `REPEAT` and an `UNTIL` statement, which is actioned thus:

```
Enter loop at the REPEAT statement.
Carry out any instructions sandwiched between the
     REPEAT and UNTIL statements.
At the UNTIL statement, check predetermined end
     condition. If it is true, then exit loop,
     otherwise go back to REPEAT statement and enter
     loop again.
```

Converting this to real Basic statements, a typical `REPEAT ... UNTIL` routine is:

```
100 Count% = 0
110 REPEAT
120    Count% += 1 :REM** Increments Count% by 1 **
130    PRINT Count%
140 UNTIL ( Count% = 6 )
```

which will print the digits 1 to 6 and then exit.

The sequence in more detail is as follows. A loop counter variable `Count%` is set to zero before the loop is entered and the first statement within the loop (Line 120) increments this by 1. The value of `Count%` (1 at this point) is then printed by the second statement (Line 130). The condition in the `UNTIL` line is then checked and is found to be false on this first time through the loop instructions. Hence the program jumps back to the start of the loop and starts again - with `Count%` now at the value 1, of course.

The program continues to go round and round this loop - with `Count%` increasing by 1 each time - until counts reaches the value of 6. After printing out this value, the `UNTIL` line again examines the exit condition and this time finds it is true - so the program now exits the loop (and continues with the rest of the program).

> *To avoid complication, the above introduction has been deliberately written using the normal meanings of the words "true" and "false". However, it is* **vital** *to know that the concepts of "true" and "false" have*

very specific meanings in Basic programming - and there are two keywords TRUE *and* FALSE *which are covered in depth in Appendix 3. You will need to take this Appendix on board before very long - and certainly before the next chapter. Strictly speaking therefore, the loop exits when the exit condition in the* UNTIL *line "evaluates to* TRUE*". However, if you do not want to stop the flow of this chapter, the normal meaning of the words will be quite satisfactory for this initial description of the* REPEAT ... UNTIL *loop.*

The exit condition in Line 140 above has been put in brackets purely for emphasis - it is not necessary, but this book will continue with this practice because it can be helpful at times.

There can be multiple conditions for exit, but - taken as a whole - they must yield a single true or false answer ("must evaluate to TRUE or FALSE"). Therefore, multiple conditions need to be linked by keywords such as AND/OR - *which are also quite understandable using their normal English meanings for the moment, but are covered in more detail in Appendix 6.* For example:

```
UNTIL ( Moon% = Blue% ) OR ( Pig$ = Fly$ )
```

which will exit if either Moon% = Blue% or Pig$ = Fly$ (or both), because as long as one of them is true, the answer to the multiple condition is also true.

Or:

```
UNTIL ( Moon% = Blue% ) AND ( Pig$ = Fly$ )
```

which will exit only when both Moon% = Blue% and Pig$ = Fly$, because both parts need to be true to produce a single answer of true for the multiple condition.

REPEAT and UNTIL do not need to start on a new line and they can be part of multi-statement lines. However, it is strongly recommended that you use one line each for REPEAT and UNTIL - and that you use indenting for the lines in between. The PRINT Count% routine above shows both of these features, which make the beginning and end of each loop immediately clear in a listing, and therefore much reduces the chances of problems (particularly when using 'nested loops' - see below).

On exiting a REPEAT ... UNTIL loop, the program continues at the line immediately after the UNTIL line.

Note the key feature that, as the exit condition is in the last line of the loop, the program has to carry out the loop instructions at least once. This is not always convenient and the `WHILE ... ENDWHILE` loop construction avoids this problem.

The WHILE ... ENDWHILE loop

This control loop complements the `REPEAT ... UNTIL` loop. As indicated above, the crucial difference is that a `WHILE ... ENDWHILE` loop has an entry condition at the start of its loop, in contrast with an exit condition at the end of the `REPEAT ... UNTIL` loop. The general sequence is:

```
Check if Entry Condition in WHILE statement is true. If
     it isn't, don't enter loop (i.e. go to line after
     ENDWHILE line).
If loop is entered, carry out any instructions
     sandwiched between the WHILE and ENDWHILE
     statements.
On reaching ENDWHILE, go back to WHILE statement.
```

The entry condition must be able to be evaluated to `TRUE/FALSE`. If, and only if, it is `TRUE`, the instructions within the loop (i.e. up to the `ENDWHILE` statement) will be carried out. The program then returns to the while statement again and the entry condition is re-tested. If and when the entry condition evaluates to `FALSE`, the program skips directly to the statement after the endwhile statement.

Therefore, if the entry condition is false when the program first arrives at the while statement, the loop is never entered.

Here is an example, again using only one statement per program line:

```
100 Count% = 10
110 WHILE Counts > 3
120 PRINT Count%
130 Counts -= 1 :REM** Decreases Counts by 1 **
140 ENDWHILE
```

Run as shown, this will print the numbers from 10 down to 4. But if you change Line 100 to `Count% = 2`, the loop will never be entered.

As with `REPEAT ... UNTIL` loops, the different elements of `WHILE ... ENDWHILE` loops need not be on separate program lines, but it is better if they are.

The great merit of `REPEAT ... UNTIL` and `WHILE ... ENDWHILE` loops is their simplicity, which is why they are used very frequently - and is also why we need say no more about them in explanation. Figure 6.1 compares them in flowchart form.

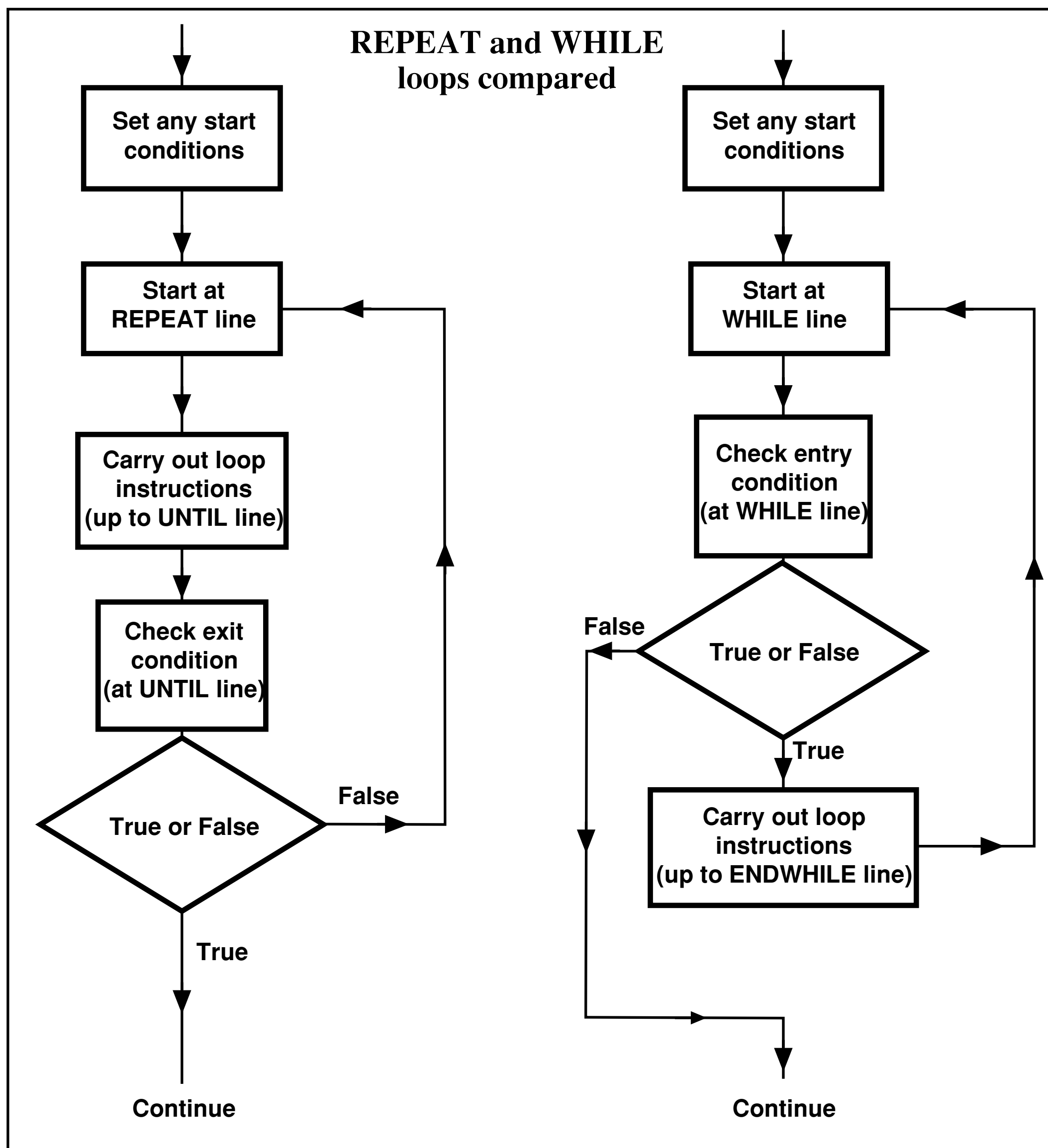


Figure 6.1

Nested loops

All loops can be ‘nested’ - that is, loops (of any of the three types) can be placed within other loops (of any of the three types) - but they must be totally nested, not ‘crossed’. For example, the sequence:

```
REPEAT :REM** entry main loop **
     <optional basic instructions>
     REPEAT :REM** entry first nested loop **
        <optional basic instructions>
        REPEAT :REM** entry second nested loop **
           <optional basic instructions>
        UNTIL :REM** exit conditions for second nested
     loop **
        <optional basic instructions>
     UNTIL :REM** exit conditions for first nested
     loop **
     <optional basic instructions)
UNTIL :REM** exit conditions for main loop **
```

would be OK, because the second ‘nest’ is completely surrounded by the first ‘nest’ and the latter by the main loop. Note how indenting helps to identify the three loops clearly.

However, the following sequence:

```
REPEAT :REM** entry main loop **
      <optional basic instructions>
      REPEAT :REM** entry first nested loop **
         <optional basic instructions)
UNTIL :REM** exit conditions for main loop **
     <optional basic instructions)
     UNTIL :REM** exit conditions for first nested
     loop **
```

would be a disaster, because the processor would assume the first `UNTIL` applied to the nested loop and the second to the main loop.

There is a limit to the number of ‘nests’ but you’ll be hard-pushed to reach it. It is more likely to be met as a result of meeting an error inside the loops.

Keyword INSTR(

Continuing on with our descriptions of the new keywords to be used for menu selection, `INSTR(` is the next on the list at the start of this chapter and it is one of the keywords provided for 'string manipulation'.

`INSTR(` acts as a function and generally takes the form:

```
Position% = INSTR( SearchString$ , SubString$ )
```

where `SubString$` needs to be of shorter length than `SearchString$`. You will note immediately that, as with `TAB(` in Chapter 4, the opening bracket is actually part of the keyword and therefore there must be no space between "`INSTR`" and "`(`".

`INSTR` looks for the presence of `SubString$` within `SearchString$` and, if found, it 'returns' the position within `SearchString$` where it found it. (`INSTR` is usually spoken as "In string", which also adequately describes what it does.) It's easier to show some examples:

```
Position% = INSTR("archimedes" , "arc")
Position% = INSTR("Archimedes" , "arc")
Position% = INSTR("Archimedes" , "e")
```

would result in `Position%` being assigned values of 1, 0 and 7 respectively.

In the first case, "`arc`" matches the first three characters of "`archimedes`" - so it returns a value 1, signifying the match starts at the first character (from the left) of the search string. In the second case, no match is found - because we've changed the first letter to a capital - and the value returned for 'no match found' is 0. In the third case, the value returned is 7 because, although there are two occurrences of "`e`", the function only considers the first occurrence of a match.

Note that, with `INSTR(` the numbering of the character positions starts at 1, rather than zero - and you will find that other string manipulation keywords (see Chapter 10) use the same counting.

If `SubString$` is longer than `SearchString$` then, again, 0 is returned.

You can force the search to start from a position other than the first character by adding a third number to the brackets, giving the start position. So:

```
Position%* = INSTR("Archimedes" , "e" , 8)
```

will produce a value of 9 - i.e. a match with the second "`e`" at position 9. Again, this is a simple, but powerful, keyword.

Keywords CHR$ and ASC

These two keywords are best introduced together, although we will only be using CHR$ in our Loan project at the moment. *Do not be misled by the brevity and simplicity of this section. It is of fundamental importance to programming - whether in Basic, other languages or machine code.*

Because computers, at heart, only recognise numbers, every character and symbol shown on the screen or printed needs to be represented within the computer by an integer number - and the range 0-255 suffices.

There is an international standard covering most of these and, for historical reasons, most people use the phrase 'the ASCII Code' when referring to this standard. (ASCII is an acronym for American Standard Code for Information Interchange.)

This leads directly to the keyword ASC, which is a function returning the ASCII code of a letter/character. For example, the statement:

```
Ascii% = ASC("A")
```

will result in the variable Ascii% being assigned with the value 65 - which is the ASCII code for "A".

In fact, ASC actually returns the ASCII code of the first character of the string in the brackets, so we would have got the same result from:

```
Ascii% = ASC("Archimedes")
```

The keyword CHR$ does the reverse translation. So:

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

will put the letter "A" on the screen.

These two keywords really are that simple - but it is obvious that they are not much use without a table of at least the ASCII codes 0-127 and Appendix 4 gives this - with the ASCII codes in both decimal and hexadecimal forms.

There are often advantages in using the hexadecimal numbering system for ASCII codes (and also for other programming topics, as will be seen later). If you are not familiar with it Appendix 5 will help.

Note from Appendix 4 that ASCII codes 0-31 are non-printing codes - usually called 'control codes'. They are used for various printing control operations e.g. TABs, carriage return, new page etc. and also for automatic data control purposes e.g. acknowledge, ready to send, etc. You need to be a little careful in using these control codes, as will be seen

later. So avoid using them for now.

The ASCII codes from 128-255 are not part of the standard and vary among different computers. They are also often used by the programmer for defining special characters.

Keywords GET$ and GET

These are the last of the new keywords to be introduced in this chapter - and they have close links with the previous section on ASCII codes.

`GET` and `GET$` are very commonly used in non-Wimp programs as a means of allowing the user to make a rather specialised type of input from the keyboard in a running program - but they should not be used for the same purpose in Wimp programs. *(There are uses for these keywords in Wimp programs but they are not often seen.)*

Firstly, `GET$` is a function returning a **one-character** string from the keyboard. Thus, if a running program arrives at the statement:

```
Char$ = GET$
```

it will assign to the string variable `Char$` the single character **of the key next pressed** by the user on the keyboard - **and will wait until one is pressed**. *(Therefore, in a normal program run, the program will pause until the user presses a key.)*

For example, if you press the keyboard key marked "H" - and you do **not** have `<Caps Lock>` engaged - `Char$` will be assigned with "h" and the program will then carry on.

`GET` works the same way but returns the ASCII code value of the key pressed e.g. it would return the integer numeric value of 104 in the above circumstances - *see the chart at Appendix 4*. So you would need to use a numeric variable to assign the return from `GET`.

The relation between the two keywords is often best regarded as "`GET$` returns `Chr$(GET)`". However, note that if the ASCII code is 0-31 (or 127) then `GET$` will return a null string, whereas `GET` will return the correct ASCII code.

When using `GET` or `GET$` the response to the keypress is instant. The character corresponding to the keypress is not automatically displayed on the screen and it does not wait for a confirmatory press of the `<return>` key. Indeed, `<return>` is a valid keypress in its own right - it will return 13 with `GET` (i.e. 'carriage return', which is what you might expect, from Appendix 4) but returns a null string with `GET$`.

Thus, the use of `GET` or `GET$` is quick but rather unforgiving and does

not give you a chance to change your mind - which means that some post-entry checking of the input is often needed.

Practical points

Possibly the most important practical point about using `GET` or `GET$` is to remember that they are functions, not variables and the program will stop and wait for a new keypress every time the keyword is encountered. Because of the way they are written, it very easy to forget this and mistakenly repeat the use of `GET/GET$` in a routine when you really want to use the character/code returned from the first keypress. For example, it is all too easy to write:

```
Choice% = INSTR( "ABCDE", GET$ )
PRINT GET$
```

and wonder why the program comes to a second halt, without printing, immediately after you have pressed a keyboard letter. If you are still puzzled, the second line is waiting for a second keypress - which, incidentally, will bear no relation to the purpose of the first line!

It is therefore best, but by no means foolproof, for beginners to adopt a habit of always assigning the result of `GET/GET$` immediately to an appropriate variable - rather than use `GET/GET$` as direct substitution for a string in a compound expression. For example, this is safer:

```
KeyPress$ = GET$
Choice% = INSTR( "ABCDE", KeyPress$ )
PRINT KeyPress$
```

Because of their characteristics, `GET` and `GET$` tend to be used for user-input cases where the user is required to make a single choice from at least two presented options. For example, `Y/N` for “`Yes/No`” or “`Choose one letter from ABCDE`” etc.

(In the next chapter we will introduce another means of user input, which suits other purposes.)

Menu selection

We now have nearly all we need to take the next step with our Loan project. If you re-run ***Program Loan5b*** you will recall that the initial menu asks the user to choose from four items which are labelled with the letters “`LNPR`”.

So, if we were to include the routine:

```
KeyPress$ = CHR$(GET)
Position% = INSTR("LNPR" , KeyPress$)
```

after showing the menu on the screen, the program would wait for us to press a key. If we then press L or N or P or R the string variable `Keypress$` will be assigned with the letter we have pressed - and the integer variable `Position%` will end up with a value of 1, 2, 3 or 4 correspondingly. *(Re-read the above descriptions of the keywords if this sequence is not clear.)* We can then use the result to direct our program further. (We could, of course, use `GET$` directly in the above, but wait a few lines to see the 'method in the madness'!)

There are a couple of problems with the above routine: what happens if we press a different letter, or if we press the lower case versions 1, n, p or r? We need a way to stop the program continuing until we've pressed one of the required letters and we need to consider it's 'user friendliness' if a lower case version is chosen.

Waiting for a valid keypress

Firstly, have a look at the following routine, which brings in the remaining keywords introduced in this chapter:

```
REPEAT
     KeyPress$ = CHR$(GET)
     Position% = INSTR("LNPR" , KeyPress$)
UNTIL ( Position% > 0 )
```

This will enter the loop and wait (in the second line) for a keypress, which will result in `Position%` being assigned a value of 0 (if the keypress doesn't match one of the search string letters) or otherwise a value in the range 1-4.

No other result is possible with a search string four characters long. So, we tell the program, in the `UNTIL` line, that we are not interested unless a number in the range 1 -4 is produced - we can use "greater than zero" in this case because we know a negative number cannot be produced. This loop will simply keep repeating until the user presses one of those four letters. So, that takes care of the first problem.

'Masking' to upper case

There is more than one way of overcoming the lower-case problem. The obvious is to change the routine to:

```
REPEAT
     Keypress$ = CHR$(GET)
     Position% = INSTR("LlNnPpRr" , KeyPress$)
UNTIL ( Position% > 0 )
```

which allows either case of letter and returns a `Position%` value in the range 1-8. However, this solution needs extra lines of programming subsequently to use the result.

A neater way for our current needs is to take advantage of the logic of the ASCII codes *(see chart at Appendix 4)*. This allows us to run the ASCII code of a keypress through a 'sieve' or 'mask' which passes upper case letters unaltered and converts lower case letters to upper case.

It sounds complex but, in fact, we only need to change the second line above to:

```
KeyPress$ = CHR$(GET AND 223)
```

You need to be familiar with binary arithmetic to understand this and Appendix 5 gives a suitable introduction to binary and hexadecimal counting *(both of which are important to understand at an introductory level and neither of which is as daunting as they may sound)*.

If you look at the binary equivalent of 223 (it is 11011111) you'll see what is happening.

All lower case letters of the alphabet are in the ASCII range 97-122 *(see ASCII chart at Appendix 4)* and all upper case are in the range 65-90 - with the lower case version of a letter having an ASCII value 32 higher than its upper case version e.g. `ASC ("A")` is 65 and `ASC ("a")` is 97.

The binary equivalents of numbers in the range 97-122 all have a binary 1 in the same bit position that 223 has it's only zero (i.e. 'bit 5'). Now:

```
1 AND 0 = 0
1 AND 1 = 1
0 AND 1 = 0
```

So, '`ANDing`' any of the numbers 97-122 with 223 changes 'bit 5' to zero and leaves all the other bits unaltered - effectively reducing the value of the number by 32. Other numbers in the range 0-127 stay the same. Reducing any of these ASCII numbers by 32 simply converts them into their upper case versions - which is just what we need.

(To practice your binary manipulation, check to see if you can similarly understand why `GET OR 32` returns the lower case of a letter keypress.)

Back to the Loan program

We can now upgrade our Loan program to permit the user to choose, from the opening menu, which of the four main factors is to be the 'unknown'.

From now on tee will be listing only the updates rather than the completely new version. Thus, the program lines listed in **`Loan Update6a`** *have to be* ***added*** *to the listing of the previous version (here,* **`Program Loan5b`***) to produce the next version (here,* **`Program Loan6a`***). The first two lines of all update listings will show the name of the next version plus the name of the previous version to which the update must be applied.*

Note particularly that there are two types of blank lines in the **`Loan Update`** *listings. If the update listing shows a Line Number with a colon (e.g. Line 140 in* **`Loan Update6a`***) then the line is intended to be entered, with its colon, as a deliberately 'blank' line. However, if the update listing shows a Line Number but nothing else (e.g. Line 1640 in* **`Loan Update6a`***) then the whole line (including its line number) is intended to be deleted. In this way - as was explained earlier - deliberately 'blank' lines will always contain a colon.*

Note also that in the final version (arrived at in Chapter 15) line numbers will start at 10 and increment by 10. However, prior to the final version, a few lines are given line numbers which do not conform to this pattern - and all this signifies is that the line contains items which are purely temporary and the whole line will be deleted by a later update. (The pausing instruction in Line 1427 of **`Loan Update6a`** *is a typical example. This instruction is moved more than once as the program builds up.)*

*If you are typing the updates, the procedure is simple but needs to be followed precisely. Firstly, copy the program to be converted (***`Program Loan5b`** *here) calling the copy, say, '***`NewProg`***'. Load* **`NewProg`** *into !Edit - or into whichever text editor you are using. Then type in all the additions/changes shown in the appropriate* **`Loan Update`** *listing (***`Loan Update6a`** *here). When completed, save* **`NewProg`** *and check that it runs OK. When you are completely sure that all is well, rename* **`NewProg`** *as the new version (*`Loan6a` *here).*

Naturally, the disc makes everything easier! It contains, separately, all the updates and all the updated versions.

Loan Update6a

```
  10 REM> Loan6a
  20 REM** Updated from Loan5b **
 140 :
1190 :
1380 REM** Permits user choice from menu items, puts
     choice in upper-case letter form and assigns it
     to a string variable. Also constructs a second
     string for later printing. **
1400 Unknown$ = FNmenuChoice :REM** Assigns 'Unknown$'
     with upper-case chosen letter
1415 PRINTrPRINT
1420 ChoiceResult$ = "The unknown factor is: " +
     Unknown$
1425 PRINT TAB(Offset1%) ChoiceResult$
1427 pause% = GET
1450 :
1460 REM****************************************
1470 :
1480 DEF FNmenuChoice
1490 REM** Returns menu choice as upper case letter **
1520 :
1530 REPEAT
1540           KeyPress$ = CHR$(GET AND 223) :REM**
     'Masks' to upper case **
1550           Position% = INSTR("LNPR",KeyPress$)
1560 UNTIL ( Position% > 0 )
1570 :
1580 = KeyPress$
1640
1650
1660
1670
1680
1690
1700
2590
2600
2610
2620
```

```
6260
6270
6280 :
6290 ENDPROC
6300 :
6310 REM****************************************
6320 REM****************************************
6330 :
```

After doing the 'housekeeping' (in Lines 10 and 20) and removing some earlier temporary instructions. ***Loan Update6a*** alters `DEF PROCchooseUnknown` in Lines 1380-1427. This primarily calls the new `FNmenuChoice` and returns the result to the variable `Unknown$`, which is just printed on the screen for confirmation and is effectively the end of the programme at this stage of its development (see Figure 6.2 where the user has chosen "N" as the 'unknown').

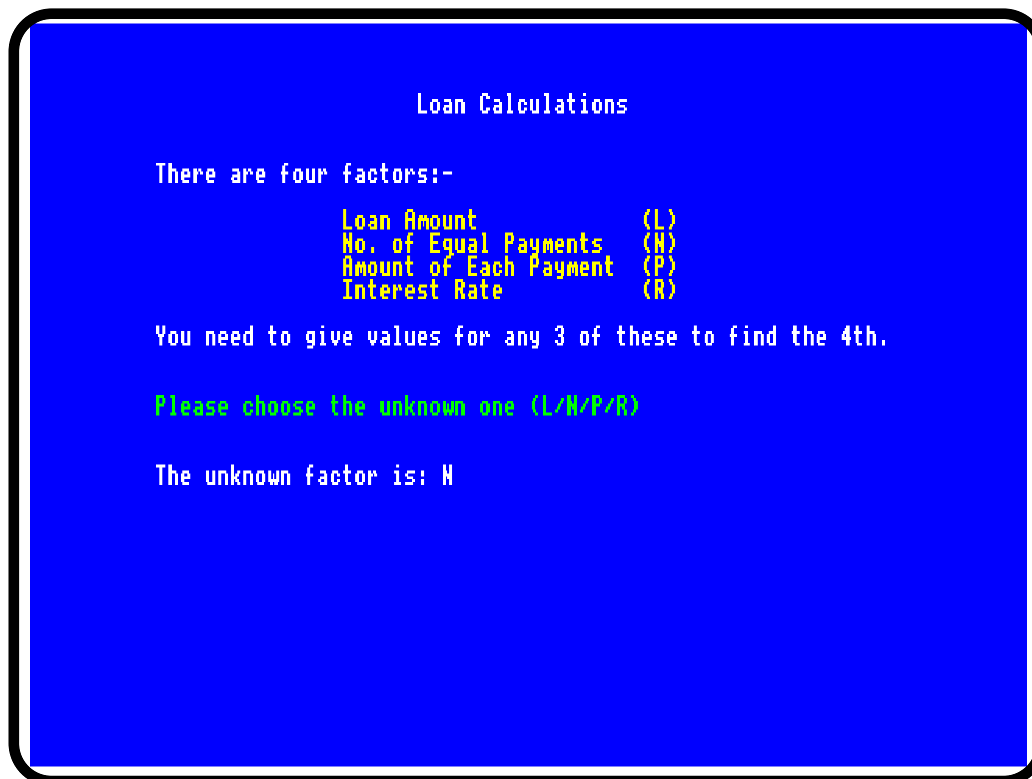


Figure 6.2

The result of running 'Loan6a'

`FNmenuChoice` incorporates the routine introduced a little earlier to return the chosen 'unknown' letter in upper case - from the selection L/N/P/R, each of which represents one of the four main factors we are concerned with.

Some minor printing instructions have also been moved and - to leave our Loan project in a better state for the next chapter - the initial simple interest example calculations have been deleted, leaving three `PROCS` deliberately empty for the time being. As we said in Chapter 5, empty `PROCS` do no harm and act as a useful reminder of things still to be done.

Keyword INKEY (and INKEY$)

Although we will not be using it in our Loan program, BBC Basic offers another means of 'capturing' a user's keyboard keypress within a running program. Like `GET`, it is not recommended for use within Wimp programs.

The keyword involved is `INKEY` and it can be used in two distinctly different ways - depending on whether the integer parameter added to the keyword is positive or negative.

The 'negative INKEY' case is easiest to cover, so we will start with that.

'negative INKEY'

This usage lends itself particularly to cases where the programmer wants the user to press a particular key/letter - rather than to make a single selection from a choice of more than one key/letter. A typical format is:

```
REPEAT
UNTIL ( INKEY (-99) = TRUE )
```

quite often seen just as the multi-statement line:

```
REPEAT UNTIL INKEY -99
```

This routine simply 'waits' (cycles repeatedly) until the space-bar is pressed. More strictly, the function `INKEY -99` only returns true if the space bar is being pressed **at the time the function is called** - otherwise `FALSE` is returned.

If the negative number had been, say, -67 then the "X" key would have to be pressed instead of the space bar. This is because **every** key on the keyboard (and the three mouse buttons) is allocated a unique negative number for this purpose - in the range -255 to -1. A complete list is given in Acorn's Basic Reference Manual.

The brackets surrounding the parameter are optional - and the same goes for spaces.

As the 'negative INKEY' function tests the keyboard at the time it is called, it is usually seen within a `REPEAT ... UNTIL` loop, as above.

Note that (unlike GET) 'negative INKEY' provides no discrimination between upper and lower case characters from a keypress - it is strictly related to the physical key only. This can be an advantage or a disadvantage, according to your needs at the time.

'positive INKEY'

When INKEY is used with a positive integer as a parameter the purpose is very different (and not too far removed from GET). The parameter can be a positive integer in the range 0 to 32767 (&7FFF) - and this is interpreted as **a time limit in centi-seconds** i.e. 100 means 1 second.

The function then returns the ASCII code of the next keypress - **provided it occurs within the time limit**. If no keypress occurs in this time the function returns -1. A typical sequence is:

```
REPEAT
      Keypress% = INKEY (0)
      IF Keypress% = FireButton% THEN PROCfire
      ...
      <other actions>
      ...
UNTIL End% =TRUE
```

As implied by the above example, this usage is often found in games, where the user is expected to make particular keyboard presses without unduly holding up the program e.g. to jump the hero over an oncoming boulder.

Finally, INKEY with a positive parameter has its string counterpart INKEY$, which acts in the same way but returns the character of the keypress - or a null string if no keypress occurs within the time limit.

This chapter has started a new pattern which will occur in all those following which change our Loan project i.e. only the upgrade to the Loan program is listed, - using the line numbering of the final version.

7. Making an INPUT and the IF ... THEN construction

User entry of values with keyword INPUT - INPUT introduced in detail - IF ... THEN ... ELSE ... ENDIF construction introduced in detail (with and without use of ENDIF) - Incorporated into Loan program update - Input validation, with first example used in Loan program.

Having given the user the means to select one unknown from the four key factors, this chapter will upgrade the program so as to present the remaining three ("known") factors to the user and ask him/her to enter values for them.

In a Wimp program, making user input would almost certainly be done using 'writable icons', but in our non-Wimp exercise we will use the keyword `INPUT`. In our particular example, we'll also make use of the `IF ... THEN ... ELSE ... ENDIF` construction.

Keyword INPUT

This keyword allows keyboard entry of text or numbers within a running non-Wimp program. When the Basic processor encounters the word `INPUT`, it pauses and waits for the user to enter something. The user duly enters a number/text (which is automatically repeated on the screen) and signifies that the input is complete by pressing `<return>`.

Compare this with `GET` or `GET$` which permits the user to input a single character - and 'grabs' that character as soon as a keyboard key is pressed. The immediate differences with `INPUT` are that, up to the time that `<return>` is pressed, the intended input is visible and can be amended using the `<backspace-and-delete>` key and retyping. Thus `INPUT` tends to be used for totally different input purposes than `GET/GET$`.

The syntax of `INPUT` is very similar to that of `PRINT` *(see Chapter 4)* and it

is most easily explored in Basic Immediate Mode. So, enter Basic from a Task Window *(see Chapter 1)* and type:

```
INPUT RealNumber
```

This will cause a "?" character to appear on the next line, with the caret immediately after it. The "?" is the default INPUT prompt telling you that your typed input is awaited. So, respond by typing, say:

67.9 (plus <return>)

The Basic prompt (">") will appear on the next line again and nothing else appears to have happened. But now type:

```
PRINT RealNumber
```

and 67.9 will be printed - proving that the INPUT statement plus your typed response did two things: firstly, the variable RealNumber was declared and, secondly, the value you typed in was assigned to it.

The remaining aspects of INPUT concern the use of 'modifiers' (remember them in Chapter 4, discussing PRINT?) which allow us to make things more user-friendly by specifying and formatting meaningful prompts for the user. This time, still in the Task Window, type:

```
INPUT "Integer = " Integer%
```

and, instead of a "?" you will see:

```
Integer =
```

with the caret after the equals sign - and no "?".

In effect, INPUT allows you to customize the prompt by putting your chosen text in quotes immediately after the keyword. If you choose to do this, a further option is available. Make some keyboard input to get back to the Basic prompt, then type:

```
INPUT "Integer = ", Integer%
```

Note the comma. This time the result is:

```
Integer = ?
```

with the caret waiting after the "?" this time. (A semi-colon could have been used instead of the comma.) So, the comma (or semi-colon) 'switches on' the "?" - provided that text-in-quotes is present after the keyword INPUT.

If you have more than one item to be input, they can be included in one INPUT statement, thus:

```
INPUT One, Two, Three
```

will provide a "?" prompt at the start and after you have made the first 'input-plus-<return>' - and again after the second input. Different customised prompts can be given to each one, thus:

```
INPUT "One = ", One "Two = ", Two "Three = ", Three
```

As before, leave out the commas if you don't want the "?" to appear.

Figure 7.1 shows how the screen will appear after all the above actions in the Task Window.

```
>INPUT RealNumber
?67.9
>PRINT RealNumber
  67.9
>
>INPUT "Integer = " Integer%
Integer = 66
>
>INPUT "Integer = ", Integer%
Integer = ?66
>
>INPUT One, Two, Three
?1.1
?2.2
?3.3
>
>INPUT "One = ", One "Two = ", Two "Three = ", Three
One = ?1.1
Two = ?2.2
Three = ?3.3
>
```

Figure 7.1

Demonstrating 'INPUT' in a Task Window

TAB and ' (single quote) can also be used, exactly as with PRINT. Try them for 'homework'.

The only thing you cannot do with INPUT is to use a string variable instead of (or in addition to) the text-in-quotes - because a variable would not be able to be distinguished from the multiple input example above. For example:

```
InputString$ = "Integer = " :REM** Intended as a prompt
      message for the next line. **
  INPUT InputString$, Integer%
```

would not work as intended. The INPUT statement is exactly the same as would be needed for two input items - the first a string (which would replace what has just been assigned to InputString$) and the second an integer.

Another short piece of useful 'homework' is to explore what happens if the keyboard input doesn't match the variable type e.g. input text when a number is needed and check what is assigned. You will find that INPUT is not very good in providing error messages.

Finally, just hitting <return> when input is awaited is a valid response. It is interpreted as zero or a null string, depending on the input variable type. A program may need to guard against this input.

Keywords IF ... THEN ... ELSE ... ENDIF

(As warned in Chapter 6, you need to have read Appendix 3 on TRUE/FALSE *before going further.)*

This is one of two main 'branching' constructions in BBC Basic - the other being CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE which we will come to in Chapter 8.

Single-line form

IF ... THEN ... ELSE ... ENDIF can be used in two distinctly separate ways: firstly by just using the keyword combination IF ... THEN (with or without ELSE) within a one-line statement. For example:

```
IF ( Tuesday% = TRUE ) THEN Itinerary$="It must be
      Rome"
```

or:

```
IF ( Weekday% = TRUE ) THEN Diary$="Got up"  ELSE
      Diary$ = "Stayed in bed"
```

In this one-line form, the THEN is actually optional, but at this stage you are strongly advised to include it.

The expression between IF and THEN is called the 'condition', for fairly obvious reasons, and it must to be able to be evaluated to TRUE or FALSE - exactly as with the conditions we met earlier in discussing the REPEAT ... UNTIL and WHILE ... ENDWHILE control loops.

Also as before, the condition has been put in brackets in the above two examples. This time it also helps to emphasize that **everything** between IF and THEN counts as the condition.

The program action is then as follows:

if the condition evaluates to FALSE, the statements after then up to an ELSE (if there is one) are ignored. If there is an ELSE present, the statements after it are actioned and the program then continues at the next line.

if the condition evaluates to TRUE, the statements after THEN up to an ELSE (if there is one), or (if not) up to the end of the program line, are actioned. If there is an ELSE present, the instructions after it are ignored and the program steps to the next line.

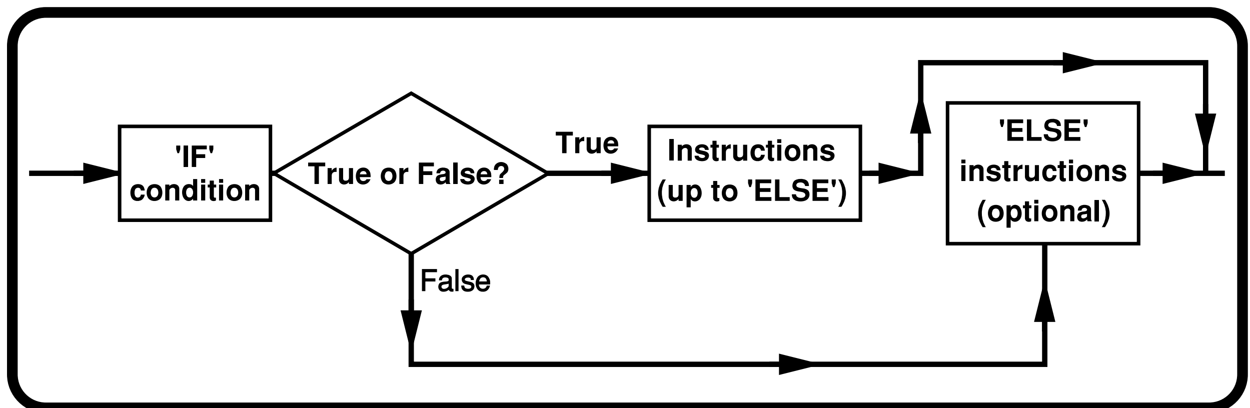


Figure 7.2 'IF ... THEN ... ELSE' construction

Thus, as seen in Figure7.2, this construction gives the programmer a means to direct the program to flow along one of two different paths according to the state of the IF condition at the time it is encountered.

As this usage must be on one program line, it is not uncommon for it to be multi-statement and quite long. Nonetheless, it is straightforward to use in this simple form - and for many years prior to Basic V it was the only option available. However, things can get a little hairy if you attempt to ‘nest’ further IF ... THENs within the one line. It is ‘legal’, but sorting out what should then happen is not always easy - and it is no longer necessary, because Basic V adds ENDIF to the story

Multi-line form

ENDIF allows us to put our IF ... THEN constructions on more than one line - simply ending the construction with the keyword ENDIF to let the computer know that the construction is finished. The general form is now:

```
IF <condition> THEN
      <Basic instructions>
      ELSE
      <Basic instructions>
ENDIF
```

Again, the ELSE (and its following instructions) is optional. As before,

<condition> is evaluated to TRUE/FALSE and the sequences are as before - and Figure 7.2 is still valid (just mentally add an ENDIF box to the right-hand side if you wish).

The vital difference in this variation is that the THEN on the first line **must be present and be the last item on that line** (even a space after it can cause problems).

By using the ENDIF form, particularly with indented listings, all the mind-twisting problems of 'nesting' in the one-line form disappear and quite complex 'nests', become easy to sort out.

Back to the Loan project

In this chapter we are going to upgrade our Loan program twice - firstly to incorporate the above new keywords without complication and then to iron out a problem that this simplicity leaves! Taking it in two bites will ensure that the two aspects are each more clearly introduced.

So, firstly, ***Loan Update7a*** updates ***Program Loan6a*** to ***Program Loan7a***. The change is to fill out PROCinputKnowns, which has also needed a third formatting offset variable to be added to PROCinit.

Loan Update7a

```
10 REM> Loan7a
20 REM** Upgraded from Loan6a **
180 PROCinputKnowns(Unknown$)
510 Offset1% = 10 : Offset2% = 15 : Offset3% = 40
1410 :
1415
1425
1427
1630 DEF PROCinputKnowns(NotThisLetter$)
1660 :
1670 CLS : REM** Clears Text screen **
1680 VDU31,0,4 : REM** Moves text cursor to position
     0,4 **
1690 :
1750 COLOUR EmphasisTextCol%
1760 PROCcentrePrint(  ChoiceResult$ )
1770 PRINT
```

```
1780 COLOUR NormalTextCol%
1790 :
1800 PROCcentrePrint( "Please enter the known values,
     each followed by <return>" )
1810 PRINT:PRINT
1840 :
1850 IF ( NotThisLetter$ <> "L" ) THEN
1860     L = FNinputLoop( "(L) Loan Amount (£)" )
1870 ENDIF
1980 :
1990 IF ( NotThisLetter? <> "N" ) THEN
2000     N = FNinputLoop( "(N) Number of Equal Monthly
     Payments" )
2010 ENDIF
2060 :
2070 IF ( NotThisLetter$ <> "P" ) THEN
2080     P = FNinputLoop( "(P) Monthly Premium (£)" )
2090 ENDIF
2100 :
2110 IF ( NotThisLetter$ <> "R" ) THEN
2120     R = FNinputLoop( "(R) Monthly Interest Rate
     (%)" )
2130 ENDIF
2160 :
2170 ENDPROC
2300 :
2310 REM****************************************
2320 :
2330 DEF FNinputLoop( Input$ )
2430 :
2440 COLOUR EmphasisTextCol%
2450 PRINT TAB(Offset3% - LEN(Input$)) Input$ ;
2470 COLOUR NormalTextCol%
2480 INPUT " " Item
2500 :
2510 PRINT
2520 :
2530 = Item
```

Firstly, `PROCinputKnowns` has been changed to pass a string as a formal parameter - the string being the letter chosen by the user representing the unknown item. The `PROC` is now defined as `DEF PROCinputKnowns(NotThisLetter$)` and is called at Line 180 by `PROCinputKnowns(Unknown$)`. Thus, the upper case letter label of the unknown item previously chosen by the user is now passed to this `PROC`.

In `DEF PROCinputKnowns(NotThisLetter$)` starting at Line 1630 there are four similar `IF ... THEN ... ENDIF` sequences, each with a condition statement for a different one of the letters we are concerned with. Therefore, each time the `PROC` is called, one of these four condition statements will evaluate to `FALSE` - because it's letter will match `Unknown$` passed to this procedure by Line 180 and we have used "`<>`" ("not equal to") as the condition. Therefore, the sequence with this match will be by-passed and the remaining three blocks - which will evaluate to `TRUE` - will be actioned.

Within each `IF ... THEN .. . ENDIF` block, there is a need to handle the input of the items almost identically, and therefore it makes sense to use a `PROC/FN` within it to do this. We have used `FNinputLoop()`, passing the individual text string (different for each item) as the single formal parameter, and returning the value input.

The listing should now be straightforward to follow. The net effect is that the user will always be presented, in a uniform format, with three items for input - but which three depends on which item was previously chosen by the user as the 'unknown'.

Input validation

As it stands in ***Program Loan7a***, the input section works well enough mechanically but leaves a lot to be desired, because it does nothing to protect the program against invalid or inappropriate input entries. We need to do some "input validation" - which is an important general aspect of programming.

Rather than try to solve all the input validation problems in one go, our second upgrade in this chapter will remove the main problems only - to prevent you running into too many difficulties when 'playing' with the program over its main development. At our final steps, in a later chapter, we will re-visit the topic to improve things further.

At this stage it will suffice to set some overall input limits for each of the four input items, show them to the user and prevent input values which are outside these limits.

The overall limits we will use are, arbitrarily:

(L) Loan Amount

Not less than £500 and not more than £15000

(N) Number of Equal Monthly Payments

Not less than 12 and not more than 180

(P) Amount of Monthly Payment

Not less than £20 and not more than £1000

(N) Monthly Interest Rate

Not less than 0.5% and not more than 3.0%

To incorporate these limits into the input process we need to modify `FNinputLoop()`: firstly, by defining the above set of limits and then by adding the right limits to each input message.

Also, the actual `INPUT` routine now needs to be put inside a `REPEAT ... UNTIL` loop, with exit conditions corresponding to the limits. Thus, the user will not be allowed to exit the loop until input values within the pre-set overall limits have been entered - similar to our approach in `FNmenuChoice` (see Chapter 6).

An easy way to achieve these needs is to add the limits as formal parameters to the definition of `FNinputLoop()` and modify the calling statements in `DEF PROCinputKnowns()` correspondingly.

However, it is not all quite straightforward because we need to do a little extra work to show our new overall limits to the user - and to overcome the screen formatting problems that would occur if invalid inputs are made.

Loan Update 7b contains all the necessary changes and we will go through some of the detailed points after the listing. *(A few of the lines in this listing are un-modified repeats of existing lines - purely for clarity when reading the upgrade.)*

Loan Update7b

```
10 REM> Loan7b
20 REM** Upgraded from Loan7a **
1800 PRINT TAB(Offset1%)"Please enter the values of the
     known factors,"''TAB(40)"...... each followed by
     <return>"
1820 :
1830 PROCsetLimits
```

```
1860     L = FNinputLoop( "(L) Loan Amount (£)" ,
     LoanLower , LoanUpper )
2000     N = FNinputLoop( "(N) Number of Equal Monthly
     Payments" , NumberLower , NumberUpper )
2080     P = FNinputLoop( "(P) Monthly Premium (£)" ,
     PaymentLower , PaymentUpper )
2120     R = FNinputLoop( "(R) Monthly Interest Rate
     (%)" , RateLower , RateUpper )
2140 :
2150 COLOUR NormalTextCol%
2180 :
2190 REM****************************************
2200 :
2210 DEF PROCsetLimits
2220 REM** Use to set overall input limits for the four
     factors. **
2230 :
2240 LoanLower = 500 : LoanUpper = 15000
2250 NumberLower = 12 : NumberUpper = 180
2260 PaymentLower = 20 : PaymentUpper = 1000
2270 RateLower =0.5 : RateUpper = 3.0
2280 :
2290 ENDPROC
2300 :
2310 REM****************************************
2320 :
2330 DEF FNinputLoop( Input$ , LowerLimit , UpperLimit
     )
2340 REM** Returns input within given limits **
2370 :
2380 Limit$ = "(" + STR$(LowerLimit) + " - " +
     STR$(UpperLimit) + ")"
2390 :
2400 Loop % = 0
2410 REPEAT
2420     Loop% = Loop% + 1
2430     IF ( Loop% > 1 ) THEN VDU11 : REM** Moves
     text cursor back up to original line if an
     invalid entry has been made **
2440     COLOUR EmphasisTextCol%
```

```
2450      PRINT TAB(Offset3% - LEN(Input$)) Input$ ;
     SPC(15 ) ; Limit$;
2460      VDU31 , Offset3% , VPOS : REM** Moves text
     cursor into gap after 'Input$' **
2470      COLOUR ActionTextCol%
2480      INPUT " " Item
2490 UNTIL ( Item >= LowerLimit ) AND ( Item <=
     UpperLimit )
2500 :
2510 PRINT
2520 :
2530 = Item
```

Firstly the easy parts: Line 1800 amends the input screen opening message. Lines 2210-2290 define the new overall input limits in `DEF PROCsetLimits` and the `PROC` is called at Line 1830. *(You may feel that the call would be equally appropriate in* `DEF PROCinit` *instead.)* Lines 1860, 2000, 2080 and 2120 merely change the calls to `FNinputLoop()` to include the appropriate new overall limits as extra parameters.

It is the new `DEF PROCinputLoop()` that needs a closer explanation. Let's gloss over Line 2380 for now: it uses string handling methods which we haven't yet covered - although you can probably follow this example easily enough. The `REPEAT ... UNTIL` loop starting at Line 2410 holds the user until a valid input is made. On first entering this loop, the counter `Loop%` is zero and immediately is incremented to 1. Thus, Line 2430 is bypassed and we quickly arrive at Line 2450. This writes the input prompt message and chosen limits to the screen on the same line - but with the limits separated by a 15 space gap beyond the message, using the new keyword `SPC` which simply 'returns' the number of space characters specified by its single parameter number.

Line 2460 then moves the text cursor into that gap, to a position on the left-hand side just after the input message. The user's keyboard input (Line 2480) appears there - and don't forget that the final action of any response to the keyword `INPUT` is the press of the `<return>` key.

If a valid input value is made, the loop simply exits. However, if an invalid input is made, the loop repeats and counter `Loop%` becomes greater than 1 - bringing Line 2430 into play. This shifts the cursor up one line before the printing starts again. Why? Because the `<return>` after the keyboard input (at line 2480) will have moved the cursor down one line. Therefore, the net effect is that Line 2450 reprints the input

prompt message on the same line as before. The added bonus is that reprinting the gap erases the invalid entry - so the result is neat and tidy and reasonably user-friendly (see Figure 7.3).

(If you cannot follow what is happening, try putting a temporary new line containing `Pause% = GET` *between* ***each*** *line of the loop. You can then step through line-by-line by pressing any key and watching the screen action and cursor position in steps. This is a very useful general 'trick' to remember in non-Wimp programming - it often explains things and/or identifies problems.)*

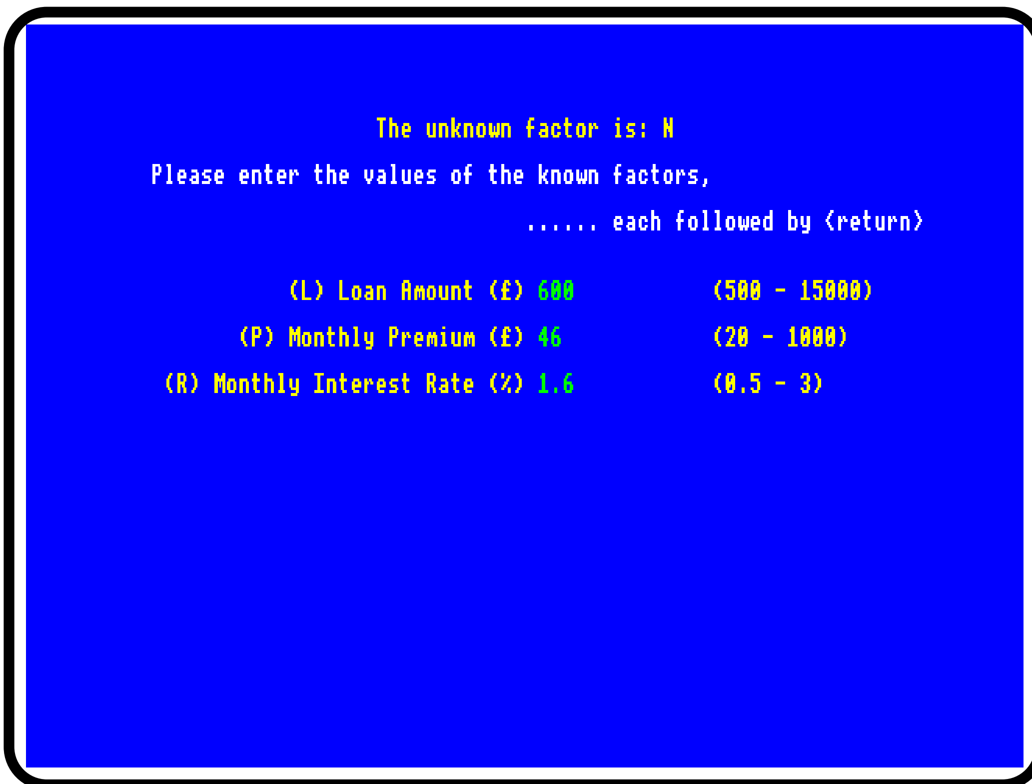


Figure 7.3

The result of running 'Loan7b'

The new keyword `VPOS` (in Line 2460) acts as a function returning the current vertical position of the text cursor, measured in lines from the top of the screen (and remembering that the top line is vertical position 0). Its horizontal position counterpart is `POS`.

Also new to you will be the keyword `VDU`, which appears more than once in this routine. The use of `VDU` is somewhat vast (and still increasing) but some commands - such as `VDU 11` and `VDU 31` above - are simple and provide useful facilities. The `REMs` give brief explanations and Appendix 9 gives a summary of all `VDU` calls, with Chapter 16 covering the keyword in more detail.

A few other minor changes have been made between ***Program***

Loan7a and ***Program Loan 7b***, to tidy presentation.

Finally, you may have noticed that real numeric variables have been used throughout in `DEF PROCsetLimits`, although `NumberLower` and `NumberUpper` are probably always going to be integer values. This is mainly for uniformity but it also enables you to use your own values without restriction.

Our program has now collected all the input data, so the following steps will concentrate on calculating the answers. As there can be any one of four choices for the ‘unknown’ factor, there need to be four corresponding calculation paths to get the answer i.e. the program now needs to be able to branch along one of four paths.

8. Branching with CASE ... OF ... WHEN

CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE construction introduced in detail with several examples - Incorporated into Loan program upgrade.

In our Loan project we've reached a point where we want the program to carry out some calculations to find the value of the fourth ('unknown') factor, having input the values of the other three factors. The trouble is, the precise calculation depends on which of the four factors is the unknown one.

So we need a means of branching the program under our control along one of a number of paths - and the `CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE` construction fits the bill.

Keywords CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE

The general form of this construction is:

```
CASE <expression> OF
     WHEN <expressionA>
          <Basic instructions A>
     WHEN <expressionB>
          <Basic instructions B>
     .............
     .............
     WHEN <expressionN>
          <Basic instructions N>
     OTHERWISE
          <other Basic instructions)
ENDCASE
```

The words used in this construction are not as close to normal English and therefore it probably needs more concentration to grasp what is happening.

The key to it is the 'expressions'. Firstly, `<expression>` (in the `CASE` line) and `<expressionA>`, `<expressionB>`, etc. (in the `WHEN` lines) can be any numeric or string expressions (e.g. variables, direct values, `FN`s) - which must exist prior to their use in the construction.

When the case line is arrived at, the current value and type of the chosen expression in the first line is compared with the values and types of `<expressionA>`, `<expressionB>`, etc. in turn, to try to find an exact value match:

if a match is found, the `<Basic instructions>` corresponding to that match is carried out - and the program then goes straight to the `ENDCASE` line and 'drops out of the bottom' of the construction to continue.

if no match is found, the program carries out the `<Basic instructions>` attached to the `OTHERWISE` statement - if there is one (it's optional) - or 'drops out the bottom' at the `ENDCASE` line.

There are several important features to note:

-The `CASE` statement must be the last statement on its program line - and the `OF` must be the last item on that line. *(Unlike after* `THEN` *in the* `IF ... THEN ... ENDIF` *case, it does not seem to matter if you add spaces after the* `OF`*.)*

-Each `WHEN` must be the first statement on a line. *(Any line put between the* `CASE` *statement and the first* `WHEN` *statement appears to be ignored - there must be a practical use for this feature!)*

-There can be as many `WHEN` statements as you like.

-A `WHEN` statement can have as many `<expressionX>` items following it as you like, each one separated by a comma. This permits identical actions for different `<expression>` values.

-The `<Basic instructions>` can be on the same line as their `CASE` or `WHEN` statement, separated by a colon - or can be multi-line statements. Everything up to the next `WHEN` (or `OTHERWISE`, or `ENDCASE`) is taken as part of the `<Basic instructions>` for the previous `WHEN`.

-If there is more than one match, only the first one (lowest line number) will be actioned. i.e. after a match is found and its `<Basic instructions>` have been executed, the program goes straight to the `ENDCASE` line - it does not continue checking for further matches. This is a very important point to take account of in some programs.

It is worth concentrating on the `<expression>` and `<expressionX>` items. In any one construction, they must each be able to be evaluated to the same type of value i.e. real or integer numeric or string. They cannot be mixed.

Here are some examples, each with the `CASE <expression>` in brackets for emphasis only:

```
CASE ( Colour% ) OF
     WHEN 0: PRINT "Black"
     WHEN 1: PRINT "Red"
     WHEN 2: PRINT "Green"
     OTHERWISE PRINT "Not interested"
ENDCASE
```

`Colour%` is an integer numeric variable, so putting a selection of integers (0, l, 2 here) with the `WHEN` statements will work `OK`. If Colour% has the value 0 when the routine is encountered in the program, the word Black will be printed - and the values 1 and 2 would produce Red and Green respectively. Any other value stored in `Colour%` will give rise to the result `Not interested`.

```
CASE Celebration$ OF
     WHEN Birthday$, Retirement$
        PRINT"Happy "; Celebration$ : PROCgift
     WHEN Wedding$
        PRINT "Good Luck!"
        PROCgift
ENDCASE
```

This example shows how two (or more) “expressions” can be associated with one `WHEN` case. `Celebration$` will need to have a certain string assigned to it by the time the `CASE` construction is entered. If that string is completely identical to the string assigned at that time to `Birthday$` or `Retirement$`, then the message `Happy Birthday` or `Happy Retirement`, respectively, will be printed and `PROCgift` carried out. Or,

if Celebration$ matches the contents of Wedding$, then Good Luck! will be printed and then PROCgift carried out.

```
CASE (TRUE) OF
WHEN ( Ascii% > 64 AND Ascii% < 91 ) : PRINT "Capital
     Letter"
WHEN ( Ascii% >= 0 AND Ascii% < 32 ) : PRINT "Not
     printable"
ENDCASE
```

This is a little different to the normal pattern. Using TRUE as the <expression> in the CASE statement can offer wide flexibility, but you then must ensure that the WHEN statements carry <expressionX> items which will similarly evaluate to the TRUE or FALSE conditions you want - rather than straightforward variables. Therefore, in the WHEN statements above, if the integer variable Ascii% (declared/assigned beforehand) has a value between 65 and 90 inclusive then the words Capital Letter will be printed. If Ascii% has a value between 0 and 31 inclusive then the words Not printable will be printed. Any other Ascii% values will produce nothing.

This example also shows that you do not have to cover all eventualities with the WHEN cases - as long as you realise that (without an OTHERWISE line) the result will simply be that the program continues with no action taken by this construction. Compare this with our first example, which included an OTHERWISE line.

```
CASE (GET$) OF
WHEN "A": PRINT "Acorn"
WHEN "A", "E", "I", "O", "U" :PRINT "Vowel"
ENDCASE
```

This is a straightforward but an often useful use of GET$. Note the deliberate ‘mistake’ here - if you press A you will get Acorn and not Vowel because it meets that comparison first. If you reversed the line order Acorn would never be printed.

Back to the Loan project ...

Loan Update8a introduces a straightforward `CASE ... OF ... ENDCASE` construction into `PROCcalculations`. It branches the program into one of four calculation routes, corresponding to the ‘unknown’ factor to be found.

Four new calculation `DEF PROCS` are created and called - and at this stage are each temporarily filled with a single message to be printed out. This allows the program to run and to report on screen that the right branch has been entered - leaving the substantive completion of the `DEF PROCS` to later.

Again, a few lines are repeated to make the update listing easier to follow.

Loan Update8a

```
10 REM> Loan8a
20 REM** Upgraded from Loan7b **
2590 :
2600 CASE Unknown$ OF
2610     WHEN "L" : PROCfindLoanAmount
2620     WHEN "N" : PROCfindNumberOfPayments
2630     WHEN "P" : PROCfindPaymentsAmount
2640     WHEN "R" : PROCfindInterestRate
2650 :
2660 ENDCASE
2670 :
2680 ENDPROC
2690 :
2700 REM****************************************
2710 REM****************************************
2720 :
2730 DEF PROCfindLoanAmount
2800 PRINT "PROCfindLoanAmount"
2810 ENDPROC
2980 :
2990 REM****************************************
3000 REM****************************************
3010 :
3020 DEF PROCfindNumberOfPayments
3470 PRINT "PROCfindNumberOfPayments"
3480 ENDPROC
```

```
3490 :
3500 REM****************************************
3510 REM****************************************
3520 :
3530 DEF PROCfindPaymentsAmount
3600 PRINT "PROCfindPaymentsAmount"
3610 ENDPROC
3620 :
3630 REM****************************************
3640 REM****************************************
3650 :
3660 DEF PROCfindInterestRate
3730 PRINT "PROCfindInterestRate"
3740 ENDPROC
6210 :
6220 REM****************************************
6230 REM****************************************
6260 :
6270 pause% = GET
```

The string variable `Unknown$` - the output from calling `PROCchooseUnknown` at Line 150 - is made the `CASE` expression in Line 2600 and four `WHEN` statements are created in the following four lines - one for each of the four possible letters with which `Unknown$` can be assigned.

In each branch, a new corresponding calculation `PROC` is then called. The `DEF PROCS` for these four different calculation `PROCs` are at Line 2730 onwards. As indicated above, when any one of these `PROCs` is called, the corresponding temporary confirmatory message will appear on the screen - to prove that the branching operation is functioning OK.

This has been a short chapter covering only one topic - albeit an important new construction. We will now need to put an appropriate calculation into each of the new DEF PROCs - but that needs to await the introduction of the third and final control loop construction in the next Chapter.

9. The FOR ... NEXT loop

FOR ... TO ... NEXT ... STEP construction introduced in detail and incorporated into Loan program upgrade.

Keywords FOR ... TO ... NEXT ... STEP

The two Basic loop constructions already covered in Chapter 6 were not complicated, but this third one is even simpler - and it is the fastest. It is used whenever you want a loop process to occur a known number of times. The general form is:

```
FOR Counter% = Start% TO Finish% STEP Step%
     <Basic instructions>
NEXT Counter%
```

but more commonly it is seen like this:

```
FOR Counter% = Start% TO Finish%
     <Basic instructions>
NEXT
```

The loop counter `Counter%` is initially automatically set to the value in the variable `Start%` and let's assume, for the moment, that `Step%` is positive. The program then carries out the `<Basic instructions>`.

When `NEXT` is reached, `Counter%` is automatically incremented by the value held in the variable `Step%` and, if the resulting new value of `Counter%` is less than or equal to the value of `Finish%`, the loop is re-entered and the `<Basic instructions>` repeated.

This continues until the automatic incrementing of `Counter%` at the `NEXT` line makes it **greater** than `Finish%`. When that happens, the loop exits immediately.

If `Step%` is negative, `Counter%` is reduced each time round the loop - and the loop then exits when `Counter%` becomes less than `Finish%`.

As already hinted, the keyword `STEP` is optional. Without it, the step is automatically +1.

Also, there is no need to include the loop counter name `Counter%` in the `NEXT` line - Basic keeps track without this and it is more of an aid to the programmer. (Indenting helps the programmer sufficiently usually, but if you prefer to show the counter name in the `NEXT` line, then do it!)

The construction is free of most constraints and can be made wholly or partially part of multi-statement lines.

There are only two points to note: firstly, like the `REPEAT ... UNTIL` loop, there is no way to avoid going through the `<Basic instructions>` at least once - whatever the values of `Start%`, `Finish%` and `Step%`.

Secondly, read the above description of what happens to `Counter%` very carefully - when the loop exits, it leaves `Counter%` with a value different to `Finish%`. Most frequently, i.e. when `Step%` is not used, `Counter%` will end up being equal to `Finish% + 1`. This often doesn't matter, but if you are subsequently going to use `Counter%` for something else - be warned!

To ensure you are thoroughly familiar with what happens, it's worth playing with the following short program. Enter Basic from a Task Window (see Chapter 1) and type in:

```
10 FOR Counter% = 1 TO 9 STEP 3
20   PRINT Counter%
30 NEXT
40 PRINT "On exit, Counter% = "; STR$(Counter%)
50 END
```

As it stands, if you run this the result is a list of the numbers 1, 4, 7 only, with `Counter%` ending up with the value 10. Did you expect that? Explore further for 'homework' by changing the values in Line 10 and noting the results.

Back to the Loan project....

As indicated at the end of the last chapter, we are now going to start programming the calculations necessary to find the value of the unknown item from the user input values of the three known items.

To do so will need a slightly (but only slightly) better understanding of

simple algebra than this book has needed so far. Appendix 11 gives a little more detail about the derivation of the formulae below - but, if maths leaves you cold, you do not need it to understand the main programming points.

To program the calculations we need to start with a formula which links our four main factors L, N, P and R. As this book is about programming, rather than algebra or finance, we do not need to spend too much time on justifying the use of a particular formula. In fact, specifically to introduce a new programming topic a little later, we will be using two formulae derived from the same starting point - again, see Appendix 11.

Firstly and mainly, we will be using the compound interest formula:

$$L \div P = B + B^2 + B^3 + B^4 + ... + B^{N-1} + B^N$$

..........(Formula 9.1)

where $B = 100 \div (100 + R)$, and:

L=Loan Amount (£)

P= Amount of each monthly payment (£)

N=Number of equal monthly payments

R=Monthly interest rate (%)

As this one formula links all four of our main factors, it should be possible to calculate the value of any one of the factors if we have values for the other three.

You can probably already see that the calculation process is fairly straightforward in the three cases where R, the interest rate, is not the unknown item. So, we are going to look at these three cases first and leave the more difficult case to a later chapter.

Further, as both L and P appear only once in the equation - and both on the left-hand side - the calculations for finding L and P are going to be very similar. So it makes sense to start with these two cases as a pair.

Finding L or P

If either L or P is the unknown item, our only programming concern is how to calculate the right-hand side of the above equation i.e. the summation of the B terms. As we will know the value of N, the exercise suits a `FOR ... NEXT` loop, as follows:

```
CuSum = 0 :REM** Variable name is short for "Cumulative
     Sum" **
BTermValue = 1
FOR Term% = 1 TO N
BTermValue = BTermValue * B
CuSum = CuSum + BTermValue
NEXT
```

When this loop is entered for the first time, the value of the variable `BTermValue` is 1 - and the fourth line of the above routine will multiply that by the value of B, producing a result equal to the value of B in this first journey round the loop - and assigning this to the variable `BTermValue`. The result of the fifth line will now be that the variable `CuSum` - which started with a value of 0 - will also contain the value B.

The second time round the loop will multiply the new value of `BTermValue` by B again - this time making it equal to B^2 - and CuSum will therefore end the second loop with a value equal to $B + B^2$.

Continuing in this fashion round the loop N times will therefore cause `BTermValue` to end up with the value of B^N - and `CuSum` to end up with a value equal to $B + B^2 + B^3 + B^4$ + etc. all the way to B^N.

Compare this final `CuSum` value with the right-hand side of ***Formula 9.1*** and you will see they are identical. So, we can now find L or P by a further single division or multiplication. Note that, for given values of R and N, we will end up with a unique answer for L or P.

Finding N

If we now have a look at the case where N is the unknown, we could use the same formula as above. We would firstly calculate the value of the left-hand side of the equation from, in this case, the known values of L and P - and then rearrange the above summation routine to become a `REPEAT ... UNTIL` loop instead of a `FOR ... NEXT` loop - and go round the loop with increasing N until a `CuSum` answer equal to **(or greater than)** L / P is achieved.

The “or greater than” arises because it is most unlikely that we will arrive at an exact answer - assuming equal payments are to be made - because N is also an integer and the result of the summation term will increase in finite steps. This is quite a common occurrence in calculations and this case will well demonstrate the caution that has to be built into the programming to cope.

However, we can do a little better than this by using the second formula from Appendix 11 - which also lends itself better to demonstrating a further programming topic a little later. So, to find N, we are going to use the formula which tells us how much of the original loan remains unpaid after a given number of equal monthly payments, that is:

Loan Remaining after N equal payments

$$= (L \times B^N) - P \times (1 + B + B^2 + ... + B^{N-1})$$

.........(Formula 9.2)

where, now, $B = (100 + R) \div 100$ i.e. the **inverse** of the previous relationship; and the other factors are the same as before.

We can still use very similar routines as before - in a `REPEAT ... UNTIL` loop with increasing N, until the right-hand side results in a suitable value. Again, there is a decision to be made here: do we continue until we get a negative result (i.e. the Nth payment has paid back too much)? Or do we aim to stop a little short?

The answer is that in a book on programming it doesn't matter - but we have to make a decision! The criterion chosen here is that we will continue until the amount of loan remaining is less than "one monthly payment plus one month's interest on it".

As the above three calculations have some similarities, this chapter's upgrade deals with all of them in one bite. Once again, the following listing repeats a few earlier lines to make things easier to follow.

Loan Update9a

```
10 REM> Loan9a
20 REM** Upgraded from Loan8a **
2730 DEF PROCfindLoanAmount
2740 :
2750 RateFactor = 100 / (100 + R) :REM** This factor
     appears repeatedly in calculations, so best
     calculated only once. **
2760 :
2770 L = P * FNsummation(N) :REM** Straightforward
compound interest, no frills **
2780 :
```

```
2790 PRINT:PRINT TAB( Offset2% ) "Loan Amount (L) = £"
     ; L
2800 :
2810 ENDPROC
2820 :
2830 REM*****************************************
2840 :
2850 DEF FNsummation(NumberOfTerms%)
2860 REM** Returns 'Sum of all terms (RateFactor raised
     to power N)'. NOTE INTEGER PARAMETER, which
     effectively copes with non-integer inputs for N
     **
2890 :
2900 TermValue = 1 : CuSum = 0
2910 :
2920 FOR Term% = 1 TO NumberOfTerms%
2930 TermValue = TermValue * RateFactor :REM**
     Provides (ratefactor raised to power 'Term%') **
2940 CuSum = CuSum + TermValue :REM** Sums
     (RateFactor powers) cumulatively **
2950 NEXT
2960 :
2970 = CuSum
2980 :
2990 REM*****************************************
3000 REM*****************************************
3010 :
3020 DEF PROCfindNumberOfPayments
3050 :
3060 InverseFactor = (100 + R) / 100 :REM** This factor
     appears repeatedly in calculations, so best
     calculated only once. **
3210 :
3220 TermValue = 1 : CuSum = 1
3230 NumOfPayments% = 0
3240 :
3250 REPEAT
3260 NumOfPayments% = NumOfPayments% + 1
3270 IF NumOfPayments% > 1 THEN CuSum = CuSum +
     TermValue
```

```
3280 TermValue = TermValue * InverseFactor
3290 :
3300 RemainingLoan = (L * TermValue) - (P * CuSum)
3360 :
3370 UNTIL ( (RemainingLoan * InverseFactor) <= P ) OR
     ( NumOfPayments% > NumberUpper ) :REM** 1st
     condition means 'until loan remaining plus one
     month's interest on it is less than or equal to
     one month's payment'.
3380 :
3400 IF ( NumOfPayments% > NumberUpper ) THEN
3410           PRINT:PRINT TAB( Offset2% ) * No. of
     payments (N) > ";STR$(NumberUpper)
3420 ELSE
3430 PRINT:PRINT TAB( Offset2% ) " No. of equal
     payments (N) = ";STR$(NumOfPayments%)
3440 ENDIF
3470 :
3480 ENDPROC
3490 :
3500 REM****************************************
3510 REM****************************************
3520 :
3530 DEF PROCfindPaymentsAmount
3540 :
3550 RateFactor = 100 / (100+R) :REM** This factor
     appears repeatedly in calculations, so best
     calculated only once. **
3560 :
3570 P = L / FNsummation(N) :REM** Straightforward
     compound interest, no frills **
3580 :
3590 PRINT:PRINT TAB( Offset2% ) "Amount of each equal
     payment (P) = £" ; P
3600 :
3610 ENDPROC
```

Firstly, we can easily see the similarity between `DEF PROCfindLoanAmount` (Line 2730) and `DEF PROCfindPaymentsAmount` (Line 3530) - both of which use `FNsummation()` (which incorporates the summation routine previously

described) and both of which are easy to follow. Note once again how the `FN` is used exactly as a variable in Lines 2770 and 3570.

The slightly longer `DEF PROCfindNumberOfPayments`, starting at Line 3020, covers the calculations to find N and comparison with the above two `DEF PROCs` should help with its understanding.

The previous B term (called `RateFactor` in the listing) has needed to be turned upside down - so it has been called `InverseFactor` to highlight this.

The `REPEAT ... UNTIL` loop and its starting conditions (from Line 3220 onwards) contains a very similar set of instructions to the `FOR ... NEXT` loop inside `DEF FNsummation()`. The main differences are the starting condition for `CuSum` and Line 3270 which does not start the summing until after the first time round the loop - to ensure that 1 is the first term of the summation in this case. Note also that the order of the `CuSum` and `TermValue` incrementing operations are reversed, compared with the order in `DEF FNsummation()`.

The two loop exit condition statements in Line 3370 need more specific explanation and look a little complicated - but it is straightforward. The first condition statement is a direct conversion of ***Formula 9.2***. If you cannot follow the maths, it means - believe it or not - “until the loan remaining plus one month’s interest on it, is less than or equal to one monthly payment”. The second condition statement is straightforward - having set an upper limit we ought to keep to it.

The main programming points to bring out from these calculations are:

Once you know the relationship (i.e. the formula, or equation) linking the items involved, producing the programming instructions to implement it is, firstly, a matter of manipulating the equation algebraically to get the particular item to be calculated on its own on the left-hand side of the equation.

For example, from the earlier starting equation:

P = L ÷ (Summation of B terms) to find P

L = P × (Summation of B terms) to find L.

The second step is to decide whether the right-hand side of the equation has any factors which warrant separate calculation. In our kilometres-to-miles `FN` in Chapter 5, for example, we just had a simple multiplication to carry out on the right-hand side - but with our compound interest formula (***Formula***

`9.1`) it is fairly clear that the summation of all the B terms is best done separately.

When all the necessary algebra has been completed, the third step is to create variables to represent all those items in the resulting equations which can have variable values - and then simply link the variables with the necessary arithmetic operators.

It is, of course, not always as straightforward as that. Often - as in `FNsummation()` above - you have to devise a way of arriving at the required answer reasonably efficiently. There is always more than one way to do it and frequently it is a matter of personal choice - and here there was also the influence of wanting to demonstrate certain programming points.

Where a factor occurs repeatedly (like B in the formula, which becomes `RateFactor` in the program above) it is always best to calculate that factor once at the start and store the result in a variable - then refer to that variable subsequently.

This is particularly important if you are using the various trigonometric and/or logarithmic functions provided in Basic (which we will be saying some words about in due course). They consume a lot of processing power and it just doesn't make sense to repeat such calculations unnecessarily. So, in our case, we start each calculation `PROC` with one such factor calculation.

A specific point arises in `DEF FNsummation()` at Line 2850. Note that the formal parameter in the brackets of the `DEF FN` is an integer - yet in lines 2770 and 3570 we call the `FN` passing a real number. Here, we are deliberately using our knowledge (see Chapter 4) that if we try to assign a non-integer number to an integer variable, it will be assigned with only the integer part of that number - and without declaring an error.

By using this feature here, we effectively correct any non-integer values entered for 'N' at the previous user-input stage i.e. we have used the feature to help with 'input validation'. Note the prominent `REM` at line 2860 to alert us to what we've done.

The above update produces ***`Program Loan9a`***. If you run this and choose either 'L' or 'P' as the unknown factor and feed in some values for the known items, a result will now be printed - as in Figure 9.1.

```
                The unknown factor is: L
      Please enter the values of the known factors,
                          ...... each followed by <return>

(N) Number of Equal Monthly Payments 36          (12 - 180)
        (P) Monthly Premium (£) 37.89            (20 - 1000)
    (R) Monthly Interest Rate (%) .8             (0.5 - 3)

       Loan Amount (L) = £1181.12019
```

Figure 9.1

The result of running 'Loan9a'

You will notice in Figure 9.1 that the answer is given to several decimal places - which is not very realistic when it represents a sum of money.

We need to convert it to just 2 decimal places i.e. pounds and pence, whatever the result. To do this requires the introduction of another programming topic, in the next Chapter.

10. String manipulation

Keyword STR$ - 'Adding' strings - Keywords LEFT$(, RIGHT$(, MID$(and LEN introduced and demonstrated - Keyword STRING$(- Loan program upgraded.

BBC Basic provides a number of keywords and other facilities specifically for handling strings. They are used in virtually all programs and they are of particular importance for Wimp programming, which tends to use strings for screen printing of numbers as well as words.

We've already introduced string variables (Chapter 3) and how to print them (Chapter 4) - as well as the keywords `CHR$`, `GET$` and `INSTR` (Chapter 6). We've also used a couple of string manipulation facilities without explanation: `STR$` and the '+' sign to join two strings. So we'll start this chapter with these last two.

Keyword STR$

This keyword is a function which converts a number into a string. The conversion can be into either a decimal or hexadecimal number. *(Again, if you are not familiar with 'hex' numbers. Appendix 5 introduces them - along with binary numbering and some associated topics.)*

Examples of the format of the keyword `STR$` are:

```
String$ = STR$ ( 23.45 )
String$ = STR$ ( RealNumber )
String$ = STR$ ( Length * Depth )
String$ = STR$ ~ 75
```

As these examples show, it can operate on anything that evaluates to a number. The result, when printed on the screen, looks exactly like the original number - but, of course, it now follows the print formatting rules

of a string rather than a number, which is often the main reason for making the conversion.

The last example shows how to produce a hex result - use of the "~" character here will assign the string "4B" (the hex equivalent of decimal 75) to `String$`. The character '~' is called a tilde. If you put the tilde in front of a real number in the above format, only the integer part of the real number will be converted to hex. *Incidentally, just in case you are ever asked in Trivial Pursuits, the tilde has the highest value of printed character in the standard ASCII set - 126 (see Appendix 4) - and is therefore sometimes also seen in programs which need to set a variable to the highest ASCII value for some reason e.g. alphabetical sorting.*

The brackets around the number/variable (the "argument") of `STR$` are not needed - indeed the latest Acorn Basic Reference Manual doesn't show them in the formal syntax - but they help to ensure you include precisely what you want in the argument. If you do use brackets then the tilde, if used, must be before the opening bracket. Formatting is not strict and therefore you can use spaces between the different elements, as in the above examples.

One of the main advantages of using `STR$` is that it is much easier to format and position a number on the screen if it is, in fact, a string i.e. "123.45" instead of 123.45. `FNnumberToString` in our next Loan program update will demonstrate this very clearly.

Finally, the complement to STR$ is VAL, which converts the string form of a decimal number into a number - and we will meet this in the next chapter.

Adding strings

The Basic statement:

```
NewString$ = "The cat sat " + "on the mat"
```

will assign to `NewString$` the value "`The cat sat on the mat`". The '+' sign simply joins the two strings together, in the order written, to produce one string. Any number of strings and/or string variables - anything that evaluates to a string, in fact - can be (wait for it!) 'concatenated' in this way - but don't forget that the limit of 255 characters for the result still applies.

A very common example is:

```
PRINT "&" + STR$ ~ (Number%)
```

which prints the hex sign in front of the number-in-hex just to ensure there is no confusion.

By the way, you cannot use the minus sign to 'subtract' strings.

Keywords LEFT$(, RIGHT$(and MID$(

These are probably the main string manipulators and they act as functions, in similar ways.

The normal format of `LEFT$()` is:

```
LeftPortion$ = LEFT*( "ABCDEFGHIJKL" , 6 )
```

which assigns "ABCDEF" to LeftPortion$ i.e. it returns the first 6 characters from the left of the argument string. The number in the bracket is, in fact, optional: if it is omitted all but the last character is returned (i.e. "ABCDEFGHIJK" here):

Similarly,

```
RightPortion$ = RIGHT$( "ABCDEFGHIJKL" , 6 )
```

would assign "GHIJKL" to RightPortion$. In this case, if the number is omitted only the last character ("L" here) is returned.

`MID$()` is slightly more complicated. The general form is:

```
MiddlePortion$ = MID$( "ABCDEFGHIJKL" , 4, 3)
```

which assigns "`DEF`" to `MiddlePortion$` i.e. it returns the three characters which start at the 4th character from the left. A number in the '4' position is essential, but the second number is optional. If a second number is omitted, the function returns the starting point character **plus the whole of the string to its right**.

Note that the opening bracket is part of the keyword in each case, so there must not be a space between, for example, `LEFT$` and `(`.

All of these keywords are more normally seen with string variables (rather than direct strings as used in the examples). As usual, the best way to get familiar with them is to 'play' in a Task Window.

There is a second way in which these three keywords can be used, to substitute part of one string into another, but we have no need to look at that.

Keyword LEN

This is almost self-explanatory, it is a function which returns the number of characters in the subject string. So,

```
Length% = LEN ( "ABCDEFGHIJKL" )
```

would return the integer number 12 to the variable `Length%`.

It is widely used to help with print formatting when we are dealing with strings of different length. You'll remember that we used `LEN` in `PROCcentrePrint()` very early on in our Loan program (Line 1290 of ***Program Loan5b***) for this purpose. The brackets are again optional, and the addition of spaces between `LEN` and its argument is not important.

Short demonstration program

As a demonstration, the following short program uses several of the above keywords to take an entered string and print it vertically on screen. After you are sure it is entered and running correctly, use it as a 'playground'.

In particular, because it can cause confusion if you don't know, deliberately put a space before the opening bracket of a `TAB(` or `MID$(` statement - and re-run it.

Do you understand why you get the error message "`Unknown or missing variable`"? It is because the computer does not recognise `TAB` or `MID$` as a keyword. It thinks it is a variable name - and, without the bracket, they are both perfectly valid (if unwise) variable names. But it is the first time it has seen that variable name - hence the message.

Program Prog10a

```
10 REM> Prog10a "Horizontal to Vertical"
20 ON ERROR REPORT : PRINT " at Line " ; ERL : END
30 REPEAT
40     CLS
50     Prompt$ = "Type in a string .... "
60     PRINT TAB( (30 - LEN(Prompt$)) , 10 ) Prompt$
70     INPUT TAB( 30 ) HorString$
80     FOR Letter% = 2 TO LEN (HorString$)
90         PRINT TAB( 30 ) MID$( HorString$ , Letter% ,
   1 )
100  NEXT
110  PRINT : PRINT "More? (Y/N)"
```

```
120      REPEAT
130          YesNo$ = GET$
140      UNTIL INSTR( "YyNn" , YesNo$) > 0
150 UNTIL INSTR( "YyNn" , YesNo$) > 2
160 END
```

Note also the typical use of GET$ in the final few lines of the above listing. This is a commonly used means of asking the user to make a single choice from several letters/characters.

Keyword STRING$(

This string manipulation keyword is a little different from those described above. It saves a lot of typing and/or listing space. The typical format is:

```
NewString$ = STRING$( 6 , "ABC" )
```

which will assign to `NewString$` a single string comprising 6 ‘copies’ of the string “`ABC`” joined together i.e. “`ABCABCABCABCABCABC`”. Note also that the opening bracket is again part of the keyword and must not be separated by a space.

You may recall that we used this keyword in ***Program Prog4a*** (at Line 300) to write a repeated set of numbers across the screen.

A common use for it is to draw patterns on the screen e.g.

```
PRINT STRING$( 25 , "<—>" )
```

will print a ‘pretty’ separator line across an 80-character screen.

You do need to watch the total length of the result - the error message “`string too long`” will occur if the result exceeds 255 characters *(see Chapter 3)*.

Back to the Loan project...

Our upgrade this time only amends the program to make the money results look like ‘pounds and pence’ prior to printing them on the screen. To do this `Loan Update10a` mainly adds `FNnumberToString()` to the program.

This FN uses string manipulators heavily. It takes any real number (or integer, for that matter), positive or negative, and converts it into a string - in this case with the number rounded to two decimal places i.e. particularly useful to show £/p in the normal way. It has the extra feature that allows the programmer to choose how the number 0 is handled - either producing a null string or the string “0.00” - either of which might

reasonably be wanted in different circumstances.

Existing Lines 2790 and 3590 are also modified to use the new fn and thus cause the results to be printed as realistic money values - correcting the point mentioned at the end of the previous chapter.

Loan Update10a

```
10 REM> Loan10a
20 REM** Upgraded from Loan9a **
2790 PRINT:PRINT TAB( Offset2% ) "Loan Amount (L) = £"
     ; FNnumberToString(L,FALSE)
3590 PRINT:PRINT TAB( Offset2% ) "Amount of each equal
     payment (P) = £" ; FNnumberToString(P,FALSE)
6330 :
6340 DEF FNnumberToString(Number,ZeroBlank%)
6350 REM** Converts a decimal number into a string with
     2 decimal places. Rounds up/down and adds
     trailing zeros as necessary. Copes with negative
     amounts. Useful for printing sums of money.**
6360 REM** Call gives option of printing zeros or blank
     for 'Number = 0' **
6390 :
6400 Negative% = FALSE
6410 :
6420 CASE TRUE OF
6430    WHEN ( Number = 0 ) AND ( ZeroBlank% = TRUE )
6440    Result$ = "" :REM This gives a blank space for
     '0' on printing. **
6450    :
6460    WHEN ( Number = 0 ) AND ( ZeroBlank% = FALSE )
6470    Result$ = "0.00" :REM This gives "0.00" for
     '0' on printing. **
6480    :
6490    OTHERWISE
6500    IF ( Number < 0 ) THEN Negative% = TRUE
6510    :
6520    REM** Next  two lines effectively round number
     up/down to 2 dec. places and multiply result by
     100 i.e. leaving result as whole number. This is
     then converted to a string. **
6530    Rounding = Number + (5 / 1000)
6540    String$ = STR$( INT(100 * Rounding) )
```

```
6550    :
6560    IF ( Negative* = TRUE ) THEN String$ = RIGHT?(
        String$ , LEN(String$) - 1 ) :REM** Eliminates
        negative sign temporarily, if it exists. **
6570    :
6580    IF ( LEN(String$) <= 2 ) THEN String$ =
        STRING$( 3 - LEN(String$) ,"0" ) + String$
        :REM** Adds leading zeros if needed. **
6590    :
6600    REM** Next two lines split string into left and
        right strings, ready to be placed either side of
        decimal point
6610    Left$ = LEFT$( String$ , LEN(String$) - 2 )
6620    Right$ = RIGHT$(String$ , 2)
6630    :
6640    REM** Final string constructed, then minus
        sign put in front if needed. **
6650    Result$ = Left$ + *." + Right$
6660    IF ( Negative% =TRUE ) AND ( VAL(Result$) <> 0
        ) THEN Result$ = "." + Result$
6670 ENDCASE
6680 :
6690 = Result$
6700 :
6710 REM***************************************
6720 REM***************************************
```

With the `REM` comments, `FNnumberToString()` should be fairly easy to follow. Note that the bulk of the action is within a `CASE ... ENDCASE` construction and, in fact, the main stream is within the `OTHERWISE` condition.

Lines 6530-6540 are worth noting: it is a standard way of correctly rounding numbers up/down - in this case specifically limited to rounding numbers to two decimal places.

In effect, 0.005 (i.e. 5 / 1000) is added to `Number`, before it is multiplied by 100 and the integer part of the result is extracted. If the value of the digit in the **3rd decimal place** of the original Number equals or exceeds 5, then adding 0.005 will increase by 1 the value of the digit in the **2nd decimal place** - otherwise the 2nd digit stays unaltered. Thus the integer number extracted will be correctly rounded.

Checks are then made to ensure that the format is correct for negative numbers and for numbers less than 1. The resulting string is then split at the point where the decimal point needs to be inserted - and then reconstructed around a decimal point (Line 6650). The final result is returned from the `FN` as a string.

If you run ***Program Loan10a*** and input the appropriate choices, the screen will look the same as in Figure 9.1 - except that the final result will now, more realistically, look like 'pounds and pence'.

Had we been interested solely in showing the number to 2 decimal places, there are other ways to achieve that - which do not necessarily involve conversion to a string (and, of course, we could still use `STR$(` afterwards if we wanted a string). But they are not as comprehensive as the above in dealing with 0.

`FNnumberToString()` could easily be converted to add a further formal parameter to allow the number of decimal places in the result to be varied.

Note once again that the opening bracket is part of the keyword and must not be separated by a space.

Converting numbers to a string is a very common practice in Wimp programs, where screen printing tends to use strings for better formatting control.

Before upgrading our Loan program further, we need to introduce some topics which justify separate chapters. So the next two chapters will not carry program upgrades and we pick up the development of the program again in Chapter 13.

11. Arithmetic and logical operators and built-in mathematical functions

Operator priority - Use of brackets to control calculation order - DIV and MOD Most popular built-in math/trig functions - Use of RND.

We have recently introduced several calculations into our Loan program and we will need to introduce even more in the further upgrades. It is therefore timely to comment in more detail on Basic's arithmetic/logical operators i.e. the arithmetic and logical signs/symbols used in calculations. Believe it or not, Acorn's BBC Basic Reference Manual lists 28 different operators, but we do not need to concern ourselves with all of them.

The key thing to note is that these operators are acted on by the Basic processor in accordance with a set of priority rules.

For instance, what would you say is the answer to the following calculation:

```
12 + 3 * 4 - 6 = ?
```

BBC Basic says it is 18 - but if you did not know the rules you might have got -30, or 6 for instance, depending on which order you worked things out. What this tells us is that BBC Basic evaluates the calculation

as:

```
12 + (3 * 4) - 6
```

i.e. it has a set of rules which effectively inserts brackets before starting the calculation and then does the calculation in a certain order - a priority or precedence order. In this case, it works out the multiplication item in brackets before doing the adding/subtracting.

Much of this is identical to the order in which we were/are taught

arithmetic in schools, so there should not be too many nasty surprises. Table 11.1 is a list of most of the operators and their priorities.

Table 11.1

<table>
<tr><th>Priority</th><th>Operator Symbol</th><th>Meaning</th><th></th></tr>
<tr><td>1</td><td>NOT</td><td>logical NOT</td><td></td></tr>
<tr><td></td><td>FN</td><td>functions</td><td></td></tr>
<tr><td></td><td>( )</td><td>brackets</td><td></td></tr>
<tr><td></td><td>?</td><td>byte indirection</td><td rowspan="4">See Chapter 21</td></tr>
<tr><td></td><td>!</td><td>'word' indirection</td></tr>
<tr><td></td><td>$</td><td>string indirection</td></tr>
<tr><td></td><td>|</td><td>'real' indirection</td></tr>
<tr><td>2</td><td>^</td><td>raise to the power</td><td></td></tr>
<tr><td>3</td><td>*</td><td>multiply</td><td></td></tr>
<tr><td></td><td>/</td><td>divide</td><td></td></tr>
<tr><td></td><td>DIV</td><td>integer divide</td><td></td></tr>
<tr><td></td><td>MOD</td><td>integer remainder</td><td></td></tr>
<tr><td>4</td><td>+</td><td>addition</td><td></td></tr>
<tr><td></td><td>–</td><td>subtraction</td><td></td></tr>
<tr><td>5</td><td>=</td><td>equal to</td><td></td></tr>
<tr><td></td><td><></td><td>not equal to</td><td></td></tr>
<tr><td></td><td><</td><td>less than</td><td></td></tr>
<tr><td></td><td>></td><td>greater than</td><td></td></tr>
<tr><td></td><td><=</td><td>less than or equal to</td><td></td></tr>
<tr><td></td><td>>=</td><td>greater than or equal to</td><td></td></tr>
<tr><td></td><td><<</td><td>bit shift left</td><td rowspan="3">See Appendix Y</td></tr>
<tr><td></td><td>>></td><td>bit shift right (arithmetic)</td></tr>
<tr><td></td><td>>>></td><td>bit shift right (logical)</td></tr>
<tr><td>6</td><td>AND</td><td>logical and bitwise AND</td><td></td></tr>
<tr><td>7</td><td>OR</td><td>logical and bitwise OR</td><td></td></tr>
<tr><td></td><td>EOR</td><td>logical and bitwise Exclusive OR</td><td></td></tr>
</table>

Some of the items in Table 11.1 need further comment:

Brackets

You can deduce from this list that one certain way to prevent problems is to be liberal in your use of brackets. You can freely insert and 'nest' more pairs of brackets than you are ever likely to need - and, because they take Priority 1 you effectively dictate the calculation order you want. This is particularly useful and important when you have a fairly long equation, but it is sound practice at any time. (e.g. Line 6530 of ***Loan Update10a***.)

So, if you really wanted the answer of -30 in the earlier example, you would have written:

```
(12 + 3) * (4 - 6)
```

and Basic would have obliged.

Therefore, the rule is: "If you are not sure which way Basic is going to work it out - insert pairs of brackets to force your wishes." It can't do any harm - but do remember to insert in pairs.

DIV and MOD

These may be new to you, but are both simple to understand and important.

DIV carries out 'integer division' and is best described by some examples:

`6 DIV 2`	produces the result 3
`7 DIV 2`	also produces the result 3
`10 DIV 6`	produces the result 1
`13 DIV 6`	produces the result 2

In other words, `DIV` acts like ordinary division except that it returns only the integer part of the result. Note that it does not 'round up' - rather, it 'rounds to zero'.

It is best to stick to integer numbers (because that is what these operators are intended for), but both numbers are automatically reduced to their integer parts **before** actioning anyway.

As usual, any expression evaluating to a number can be used instead of direct numbers - but, as with ordinary division, the right-hand number must not be zero.

Negative integers can be used but it is best not to at this stage.

It is safest always to put a space between the first number and DIV, because there must be a space here if you use a numeric variable name rather than a number i.e. you must use:

```
FirstNumber DIV SecondNumber
```

rather than:

```
FirstNumberDIV SecondNumber
```

(The space after DIV is not essential.)

Very often, DIV is used with hex numbers *(see Appendix 5)* to extract the upper part of a hex number. For example:

&E7F3 DIV &100	produces the result &E7 (decimal 231)
&E7F3 DIV &1000	produces the result &E (decimal 14)

MOD complements DIV by returning the 'integer remainder' of the integer division. That is:

7 MOD 2	produces the result 1
6 MOD 2	produces the result 0
10 MOD 6	produces the result 4
&E7F3 MOD &100	produces the result&F3 (decimal 243)
&E7F3 MOD &1000	produces the result &7F3 (decimal 2035)

MOD is most often used to ensure that numbers are confined to a certain range of values. For example:

```
AnyNumber% MOD &100
```

will ensure that the answer is in the range 0-255.

All the points made earlier about how to use DIV are equally applicable to MOD.

Because of their characteristics, DIV and MOD are often seen in routines working as a complementary pair to split a number into two or more parts e.g. to convert a time duration in seconds into a "hours:minutes:seconds" equivalent:

```
Seconds% = Duration% MOD 60
Minutes% = Duration% DIV 60
Hours% = Minutes% DIV 60
Minutes% = Minutes% MOD 60
```

Built-in Mathematical Functions

BBC Basic has over 20 built-in Mathematical Functions and the most-used ones are listed in Table 11.2.

Table 11.2

Function	*Returns ...*
ABS	magnitude of argument i.e. changes negative numbers into positive
COS	cosine of argument
DEG	converts radians to degrees
INT	whole number portion of argument
LOG	logarithm (base 10) of argument
PI	the value of pi (no argument)
RAD	converts degrees to radians
RND	a random number in range determined by argument
SIN	sine of argument
SQR	square root of argument
TAN	tangent of argument
VAL	numeric value of a string of a decimal number

Most of these functions need an argument added when called (i.e they need a value to operate on). For example:

```
Value = SQR (16.3)
```

will result in `Value` being assigned with the square root of the argument 16.3 i.e. 4.03732585.

They are used exactly as `FNs` - that is, you substitute the keyword (with its argument, if needed) into a program statement structure exactly as if it were a `FN`. Thus:

```
Circumference = Diameter * PI
Base = Hypotenuse * COS(IncludedAngle)
Integer% = INT(Real)
```

The main point to watch is that **all the trigonometric functions use radians**, rather than degrees, as the units for the angle in their argument. You may not have come across radians - but don't panic - they are just an alternative way of measuring angles and, as you can see in Table 11.2, there are functions to convert to/from degrees/radians. Thus, it is quite common to see expressions such as:

```
Base = Hypotenuse * COS( RAD( IncludedAngle ) )
```

in order to supply `COS` with its argument in radians, but allowing you to express the angle `IncludedAngle` in degrees.

The function RND

The only unusual function in Table 11.2 is RND. This returns a random number in the range determined by the argument, which ought to be an integer.

If the argument is greater than 1, the random number is an integer within the range 1 to `<argument number >`.

If the argument equals 1, the random number is a real number between 0 and 1.

If no argument is given, the random number is an integer within the minimum and maximum range of Basic i.e. -2147483648 to +2147483647.

Thus:

```
Dice% = RND( 6 )
```

would result in `Dice%` being assigned with either 1, 2, 3, 4, 5 or 6 - chosen at random.

It is important to note that if an argument is given it **must** be inside brackets and there must not be a space between `RND` and the opening bracket.

`RND` can also take negative arguments, but this has a special purpose that we do not need to introduce.

We need say no more on these subjects - they are largely material for reference tables etc. and they do assume some familiarity with fundamental trigonometry/algebra.

12. Introducing graphics

Graphics 'play area' - OS units - standard non-Wimp screen - Pixels - Screen resolution - Preparing screen for graphics plotting - Keywords MOVE, DRAW and BY introduced and demonstrated - Effect of screen resolution on pixel/dot size - RECTANGLE, CIRCLE, ELLIPSE and FILL demonstrated - POINT - Graphics grid.

Graphics is a very large subject and in this Beginners' book we have to limit the coverage. Mainly, we will show how to get simple graphics onto a standard non-Wimp screen, introducing the several keywords available and subsequently incorporating some of them into our Loan project. We will also say a little about variations from the 'standard' screen.

The graphics 'play area' and the screen

In BBC Basic, graphics can be drawn using conventional x-y coordinates over a 'play area' which extends from -32768 to +32767 'Operating System units' (OS units) in both x and y directions. *(These limits are the range of values possible using only 2 bytes - see Appendix 8.)*

All graphical plotting instructions in Basic are given in OS units which, as is probably obvious, allows us to ignore the actual monitor screen size.

The screen is best regarded as a viewing window looking at a small part of this play area - and you, the programmer, can move the window and, to some extent, alter its size and shape. The maximum size of the play area that can be viewed at any one time depends on which display mode you choose, but all of them view only a very small part of the total play area.

For non-Wimp programs, the 'standard screen' in BBC Basic is usually regarded as being 1280 OS units wide by 1024 OS units high i.e. about 1/250th of the play area. By default, the bottom left-hand corner of the screen is placed at the origin of the play area coordinate system i.e. the

point (0,0). The top right-hand corner of the standard screen is therefore the point (1279, 1023).

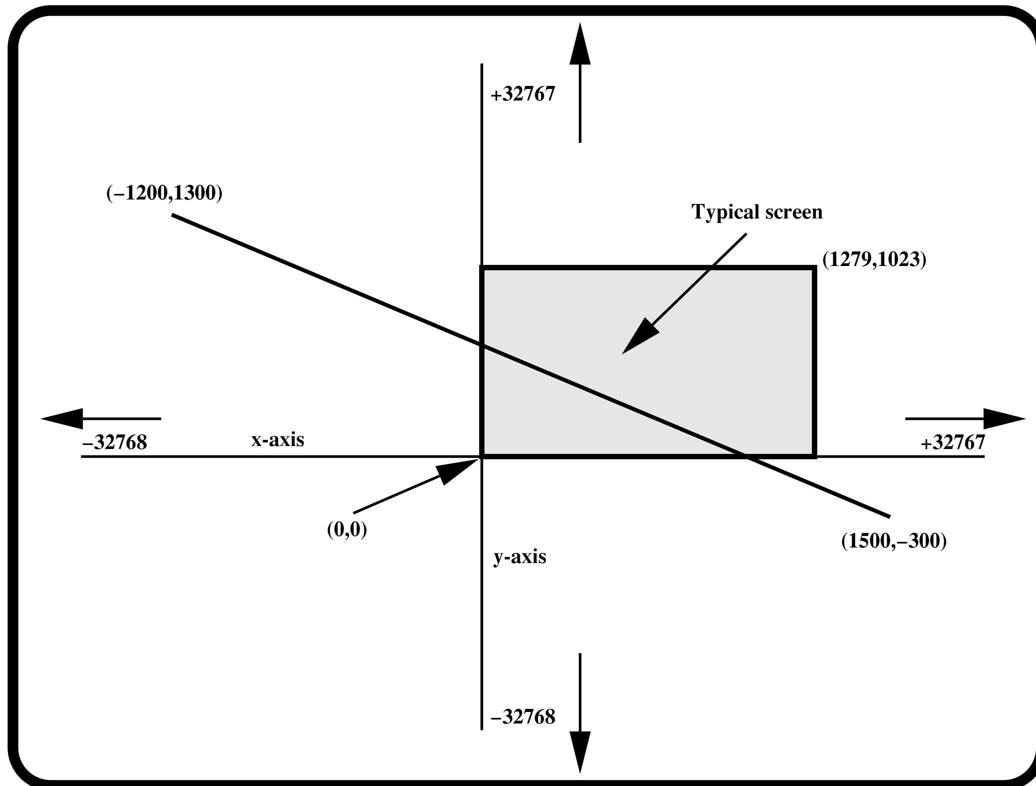


Figure 12.1

The graphics play area

Figure 12.1 shows the situation and you'll see that the default standard screen reflects the usual way we are taught graphs and (x,y) coordinates at school - with x increasing to the right and y increasing upwards. (Compare this with normal BBC Basic text coordinates, which have (0,0) at the top left-hand corner, with y increasing downwards.)

If our program plots a straight line between, say, the coordinates (-1200, 1300) and (1500, -300), as in Figure 12.1, we will actually see only that section of the line which passes through the screen viewing window.

Picture elements ("Pixels")

The plotting of the line (or of anything else) on the screen is carried out by changing the colour of those individual picture elements ("pixels") which go to make up the shape required. If we stick, for the moment, with drawing black lines on a white background, Figure 12.2 shows what is happening.

This shows a short thin vertical line, a diagonal line and a thicker horizontal line - drawn on a very much magnified small part of an imaginary screen (in this case, a screen area only 16 × 8 pixels). All other shapes and colours - including text - are merely extensions of this principle.

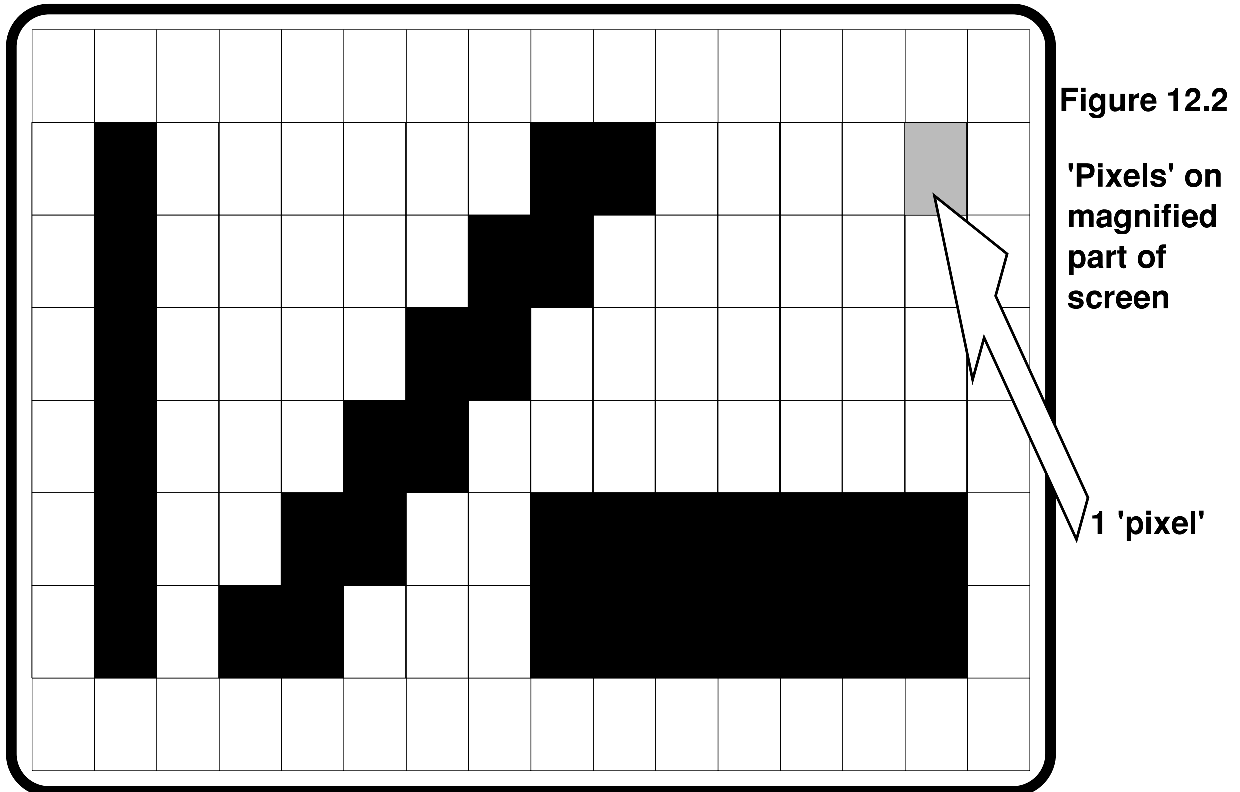


Figure 12.2

'Pixels' on magnified part of screen

You'll note that lines (and therefore shape edges) which are not vertical or horizontal cannot be produced without a certain jaggedness. This is inherent in the pixel grid display process. Whether it is noticeable, however, depends mainly on the particular display resolution. The higher (better) the resolution, the greater the memory and processing speed needed - and, within the range offered by each computer type, the programmer sets the resolution (among other things) when the display mode is chosen.

You may be thinking that pixels always equate to 1 OS unit in each direction, but unfortunately it is not as simple as that. In practice, pixels are normally larger than 1 OS unit wide and/or high - and pixels are often taller than they are wide (as they are in Figure 12.2). It all depends on the display mode. This means that, in many display modes, the smallest 'dot' you can draw on the screen is, in fact, more of a small rectangle - higher than it is wide.

The result is that some display modes will produce chunkier looking graphics or different colours - and some will not use the same shape (aspect ratio) of active screen, or might squash or cut off the shapes somewhat - and the effective width and/or height of these screens may be more/less than 1280 × 1024 OS units - see demonstration program below.

However, by sticking to OS units, we don't have to worry too much about these relationships in our programming. As we said above, the 1280 × 1024 OS unit screen can be thought of as (almost) the standard graphic screen - and if we produce a graphic program to that standard, then it will usually produce a recognisable output in most modes and adjustments for specific modes can be made if need be.

The User Guide lists the available screen display modes ('old-type' screen modes with Risc PC User Guide) with their pixel (and text) resolutions. If the pixel resolution has two numbers which are both sub-multiples of '1280 × 1024', then that mode uses the standard screen e.g. mode 12 (640 × 256) is standard, but mode 27 (640 × 480) is not - the latter has a 'standard' width but has a height of only 960 OS units.

To kick us off, let's write a small demonstration program, this time step by step, just to introduce the fundamental graphics instructions.

Preparing a screen for graphics

In Chapter 5 it was mentioned briefly that text and graphics colours are handled separately in Basic, so our first steps for graphics are to choose a screen display mode and the initial background and foreground colours.

We used the keyword `COLOUR` for text colours - and its equivalent for graphics (in non-Wimp) is `GCOL` ("graphics colour"). `GCOL` can be used for more than just setting the colours, but using it with just one number, as below, simply sets the foreground or background graphics colour. Numbers from 0 to 127 set the foreground colour and numbers from 128 to 255 set the background colour - just as with `COLOUR`.

So, ***part 1*** of ***Program Prog12a*** is a typical opening sequence of a graphics program (with the `END` line and `DEF PROCerror` included from the start, to keep things 'clean'):

Program Prog12a (part 1)

```
10 REM> Prog12a
20 REM** Graphics Introduction **
30 :
40 REM**************************************
50 :
```

```
60 MODE 12 :REM 16-colour mode
70 :
80 ON ERROR PROCerror : END
90 :
100 REM****************************************
110 :
120 GCOL 132 :REM Sets graphic background colour to
     (132-128) = 4, blue.
130 CLG :REM Clears graphic screen to background colour
140 GCOL 7 :REM Sets graphic foreground colour to 7,
     white.
150 :
160 REM****************************************
170 :
750 END
760 :
770 REM****************************************
780 REM****************************************
790 :
800 DEF PROCerror
810 REPORT
820 PRINT " at Line " ; ERL
830 ENDPROC
```

Lines 60-140 are all we need before starting to put graphics on the screen - *but we still need to say a lot more about colours in a later chapter*.

Keywords MOVE, DRAW and BY

Graphics plotting/drawing always starts from the current position of the graphics cursor - which is invisible(!). Don't confuse it with the text cursor, which is often still on the screen, flashing away. By default and after a screen mode change, the graphics cursor starts at (0,0) - at the bottom left-hand corner - but after some plotting action it remains where the end of the plotting action happens to leave it.

So it is often necessary (and always good practice) to move the graphics cursor positively to where you want it to be before you start each plot. This is where the keyword MOVE comes in. It takes a pair of x,y coordinates as its parameters and simply moves the graphic cursor to that absolute position, without putting anything on the screen. So:

```
MOVE 640,512
```

would move the graphics cursor to the centre of the default standard

screen (refer back to Figure 12.1).

Similarly:

```
DRAW x,y
```

draws a straight line from wherever the cursor happens to be to the point (x,y) - and the cursor will end up at (x,y) as a result. (Don't forget that x and y need not be on the screen.) So, ***part 2*** of ***Program Prog12a*** adds the next few lines to the demonstration program, as follows:

Program Prog12a (part 2)

```
180 LeftEdge% = 0
190 RightEdge% = 1279
200 TopEdge% = 1023
210 BottomEdge% = 0
220 :
230 MOVE RightEdge% , TopEdge% :REM Moves cursor to
     point (1279,1023), top right corner of 'standard'
     screen.
240 :
250 REM Next four lines draw line around edge of
     'standard' screen.
260 DRAW LeftEdge% , TopEdge%
270 DRAW LeftEdge% , BottomEdge%
280 DRAW RightEdge% , BottomEdge%
290 DRAW RightEdge% , TopEdge%
300 :
310 Pause% = GET
320 :
330 REM****************************************
340 :
```

Now run the program as it stands - Line 310 introduces a pause which requires any keypress to continue. It will produce a blue, mode 12 (squashed) screen with a white line around its edges. Check what happens if you change `RightEdge%` to 1280 and/or `TopEdge%` to 1024. The right and/or top white borders disappear - they have been plotted just outside the screen area.

Both `MOVE` and `DRAW` can be modified by the addition of the keyword `BY`, which changes the movement to relative rather than absolute. So, as `LeftEdge%` and `BottomEdge%` are both 0, the same borders could have

been produced by changing the program to:

```
260 DRAW BY -RightEdge%,0 :REM Note the negative sign.
270 DRAW BY 0, -TopEdge% :REM Note the negative sign.
280 DRAW BY RightEdge%, 0
290 DRAW BY 0, TopEdge%
```

As you can see, there is nothing very difficult about this type of graphic plotting as long as you keep track of the graphics cursor.

Keywords RECTANGLE, CIRCLE, ELLIPSE and FILL

BBC Basic makes our life easier by providing a series of keywords to draw the most common shapes e.g. RECTANGLE, CIRCLE, ELLIPSE. ***Part3*** of ***Program Prog12a*** adds some of these to the listing.

Program Prog12a (part 3)

```
350 REM Next 3 lines draw box 16 OS units inside screen
     border.
360 Width% = 1279 - 32
370 Height% = 1023 - 32
380 RECTANGLE LeftEdge% + 16 , BottomEdge% + 16 ,
     Width% , Height%
390 :
400 Pause% = GET
410 :
420 REM****************************************
430 :
440 CIRCLE FILL 400 , 700 , 100
450 DRAW BY 100 , 100 :REM A short diagonal line solely
     to show where cursor was left when circle
     finished.
460 :
470 ELLIPSE 1000 , 400 , 200 , -100
480 DRAW BY 100 , 100 :REM A short diagonal line solely
     to show where cursor was left when ellipse
     finished.
490 :
500 Pause% = GET
510 :
520 REM****************************************
530 :
```

The REMs should help you with the syntax.

Run the program again, pressing any key to step through. Check the effect of making the fourth number positive in line 470 - the ellipse is drawn from a start/finish point at its top instead of at its bottom, but nothing else changes.

In Mode 12 you will again find that the shapes are somewhat flattened and that the spacing between the rectangle and the screen edge is different at top and bottom compared with at the sides.

The three shapes produced by RECTANGLE, CIRCLE, ELLIPSE can also be drawn as filled shapes - filled with the current foreground colour. All that is needed is to include the extra keyword FILL before the numbers - as in Line 440.

Keyword POINT and the effect of pixel size and shape

The final part of Program Prog12a is as follows:

Program Prog12a (part 4)

```
540 REM the remaining lines show effect of pixel size,
     see text.
550 MOVE 0 , 200
560 :
570 HorInterval% = 8
580 FOR N% = 0 TO 1279 STEP HorInterval%
590     POINT N% , 200
600 NEXT
610 :
620 MOVE 0 , 160
630 DRAW RightEdge% , 160
640 :
650 MOVE 200 , 0
660 :
670 VertInterval% = 8
680 FOR N% = 0 TO 1023 STEP VertInterval%
690     POINT 200 , N%
700 NEXT
710 :
720 MOVE 240 , 0
```

```
730 DRAW 240 , TopEdge%
740 :
```

The keyword `POINT` draws a single point/dot i.e. one pixel only, at the coordinate given. Therefore, Lines 570-600 plot a series of points equally spaced across the screen. Then Lines 620-630 draw a horizontal solid line next to them for comparison. The same thing is then done in the vertical direction also. *(Note that there are two, different keywords* `POINT` *- the other one is actually called* `POINT(` *- the bracket being essential. We will come to it later when we have a look at Colour as a subject.)*

If you progressively decrease the values in lines 570 and 670, the plotted points eventually get so close together that they look the same as the solid lines next to them. But - and it's important - the **highest** value of interval that causes the lines look the same (in mode 12) is different in the x and y directions - 2 and 4 respectively i.e. the pixel size is 2 × 4 OS units in mode 12.

Now try different modes. Change Line 60 to `MODE 27`, for instance. The shapes will now look normal rather than squashed and, consequently, it uses a square pixel - 2 × 2 OS units, in fact. But note the top border is now off the screen - because `MODE 27` has a 640 × 480 resolution, which gives a screen height of only 960 OS units. Note that `POINT` automatically adjusts the pixel size and shape for the particular display mode - try `MODE 2` to get an extreme example.

Figure 12.3 shows the screen after stepping through the completed program.

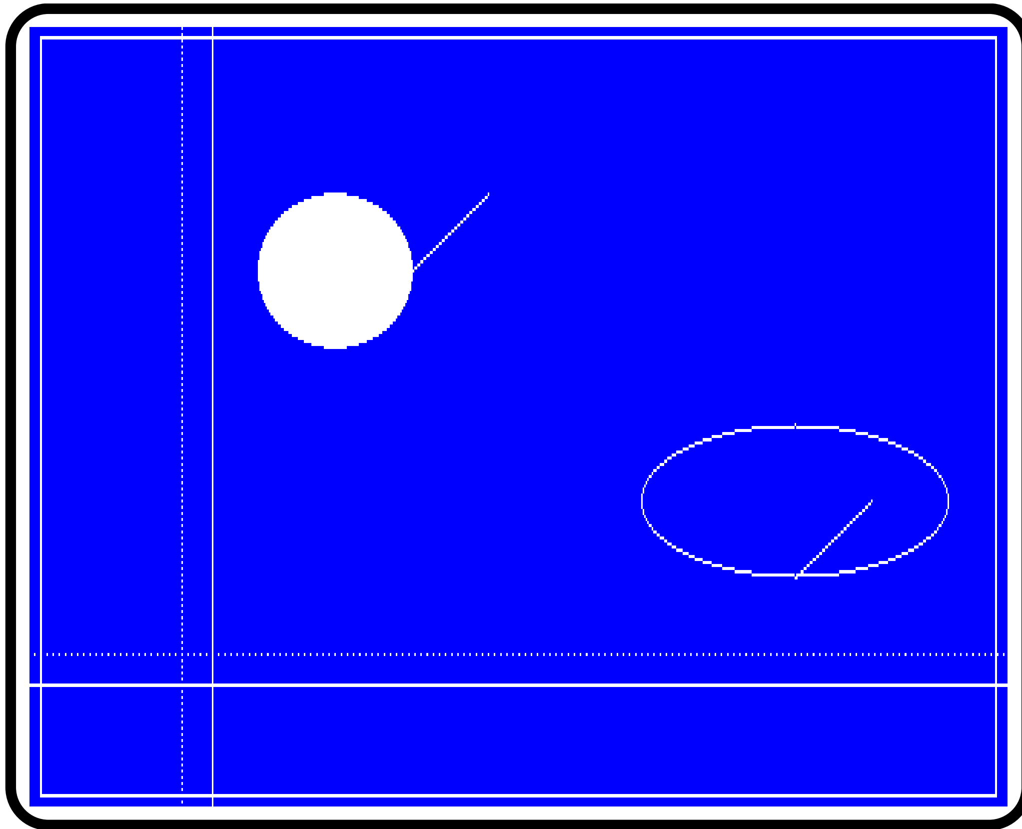


Figure 12.3

The result of running 'Prog12a'

Screen resolution

It is important to appreciate that you cannot move/locate graphics any more precisely than the particular screen resolution will allow. In other words, in any one display mode, the available pixels (of whatever size they happen to be in the chosen mode) are also effectively locked to an invisible grid.

So, in our above `MODE 12` program where the pixel size is 2 × 4 OS units, the use of the instruction `MOVE 641,515` (or any combination of the possible eight positions with an x-value from 640-641 and a y-value from 512-515, see Figure 12.4) will effectively move the cursor to position (640 , 512) - and the instruction `POINT 641,515` will draw a 'dot' the size of the pixel - with its bottom left-hand corner at position (640,512).

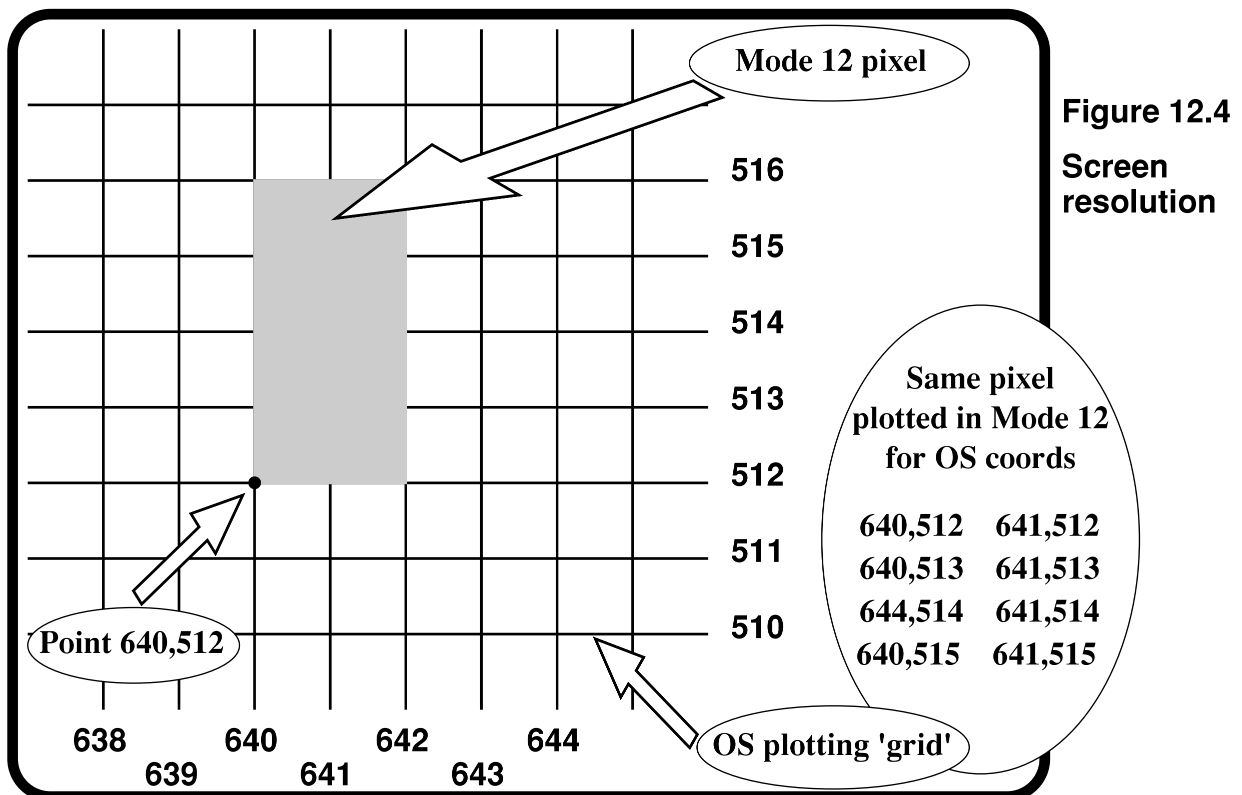


Figure 12.4

Screen resolution

Finally, by the same token, this is why point (1280, 1024) is just off the `MODE 12` screen. If you want to draw something right up against the RH (or top) edge of the standard screen, the safest way is still to use OS value 1279 (or 1023) in the drawing statement (rather than 1278 or 1020), to ensure it stays at the edge (or top) in all other standard modes - or when resolution upgrades come along.

For screen modes which have other aspect ratios, you will need to adjust the edge-of-screen OS values, of course - but they will always be an odd number.

Conversely, to draw a line against the LH (or bottom) edge of the screen, the OS value 0 will always show on the default screen - as you can deduce from Figure12.4 which shows that the pixel is plotted to the right and upwards from point 640,512.

The introduction program provides a good playground for getting to grips with the basics of non-Wimp graphics and mode resolutions. The next chapter introduces some more graphics items and starts to incorporate graphics into our Loan project.

13. More graphics

Keyword PLOT introduced - Plot code block/offset principle, with examples Introduce graphics to Loan program - graph drawing.

Keyword PLOT

So far we have introduced graphics using the `MOVE` and `DRAW` commands, plus the series of self-explanatory commands producing special shapes - like `CIRCLE`, `ELLIPSE` etc. These are fine - but limited.

To take us further, Basic provides a more general, all-purpose, graphic command called `PLOT`, which takes the general form:

```
PLOT k, x, y
```

where k is known as the plotting mode (not to be confused with the display mode) and `x`, y are the usual OS coordinate values to be used in the plotting action.

As k can take any value from 0 to 255, there are a large number of graphic effects available, including a `PLOT` equivalent for all the graphic commands we have so far introduced - although some of the latter are a little awkward to use.

Therefore, if we wanted, we could use entirely `PLOT` commands. However, it is easier to recognise `MOVE`, `DRAW` etc. in a listing, compared with their `PLOT` equivalents, so we will normally use `PLOT` in this book only for those cases where a ready-made keyword does not exist.

The 256 available values for k are split into 32 **blocks** of 8 numbers: 0-7, 8-15, 16-23, 24-31, etc. (or, in hex, &00-&07, &08-&0F, &10-&17, &18-&1F, etc. up to &F8-&FF - which shows the pattern more easily, each block starting with either &x0 or &x8).

Each block represents one graphic effect/pattern and the eight numbers within each block can best be regarded as offset values which vary the block plot action in the same way whatever the block. Table 13.1 should help - even though a couple of the meanings include a subject we have yet to cover.

Offset	Meaning
0	Move cursor relative (i.e. use x,y values as relartive to last cursor position)
1	Plot relative, using current foreground colour
2	Plot relative, using logical inverse colour (of colour(s) already on screen)
3	Plot relative, using current background colour
4	Move cursor absolute (i.e. to actual x,y coordinate position)
5	Plot absolute using current foreground colour
6	Plot absolute using logical inverse colour
7	Plot absolute using current background colour

Table 13.1 'PLOT' offset meanings

Acorn's BBC Basic Reference Manual gives a complete list of the block plot actions and a few of them are:

block	plot action
`0-7 (&00-&07)`	`draw solid line.`
`16-23 (&10-&17)`	`draw dotted line including both end points.`
`64-71 (&40-&47)`	`plot point.`
`144-151 (&90-&97)`	`draw circle outline.`

So, for example, both:

```
PLOT 5,x,y      (or PLOT &05,x(y)
```

and

```
PLOT 21,x,y     (or PLOT &15,x,y)
```

which have the same **offset** value of &5 - would mean "draw using the current foreground colour from the current cursor position to the point (x,y)" - but the first would draw a solid line and the second a dotted line, because they are in different **blocks**.

Let's give some more examples of how `PLOT` is used. Up to now we would have used a short sequence like this to draw two lines on the screen:

```
MOVE 100 , 200 :REM Move graphics cursor to required
     start point.
DRAW 300, 400 :REM Draw a solid line from start point
     to (300 , 400)
DRAW BY 500 , 600 :REM Draw a solid line 500 x-distance
     and 600 y-distance away from point (300 , 400).
```

Using plot, this could be replaced by:

```
PLOT 4 , 100 , 200 :REM 'move absolute'
PLOT 5 , 300 , 400 :REM 'draw solid line absolute, in
     foreground colour'
PLOT 1 , 500 , 600 :REM 'draw solid line relative, in
     foreground colour'
```

to give the same result.

Changing the last line to `PLOT 17,500, 600` (or `PLOT &11,50,600` to make it clearer that we are using the same offset in a different block) would make the final line dotted instead of solid.

Play with the `PLOT` mode blocks and offsets to get used to the idea - modifying, say, a copy of ***Program Prog 12a*** as your vehicle. (Don't forget that the Task Window cannot action graphics instructions.)

Note that all the `PLOT &x0` and `PLOT &x4` values are equivalent to a `MOVE BY` or `MOVE` command respectively.

At this stage we need say no more about `PLOT` as an introduction.

Back to the Loan program ...

Before tackling something more substantial, we are going to add a few lines to ***Program Loan10a*** to show how some very simple graphics can enhance a program:

Loan Update13a

```
10 REM> Loan13a
20 REM** Upgraded from Loan10a **
640 :
660 Graphic1Col% = 1 :REM** Red **
3070 :
3080 GraphHeight% = 400 :REM** A convenient height, in
     OS units. **
```

```
3090 Xorig% = 128: Yorig% = 32
3100 Scale = GraphHeight% / L
3110 Gap% = 4
3120 :
3180 PRINT TAB(6 , 30) "0"
3200 PRINT TAB(7 - LEN(STR$(L)) , 18) STR$(L)
3310 :
3320    GCOL Graphic1Col%
3330 :
3340    MOVE Xorig% + ( Gap% * NumOfPayments% ) ,
     Yorig%
3350    DRAW BY 0 , Scale * RemainingLoan
3410    PRINT TAB(45 , 24) " No. of payments (N) >
     ";STR$(NumberUpper);
3430    PRINT TAB(45 , 24) " No. of equal payments (N)
     = " ;STR$(NumOfPayments %);
```

When you have made this update, run the new program (***Program Loan13a***) - choose N as the unknown factor, enter some values for the knowns - and the result is the screenshot shown in Figure 13.1.

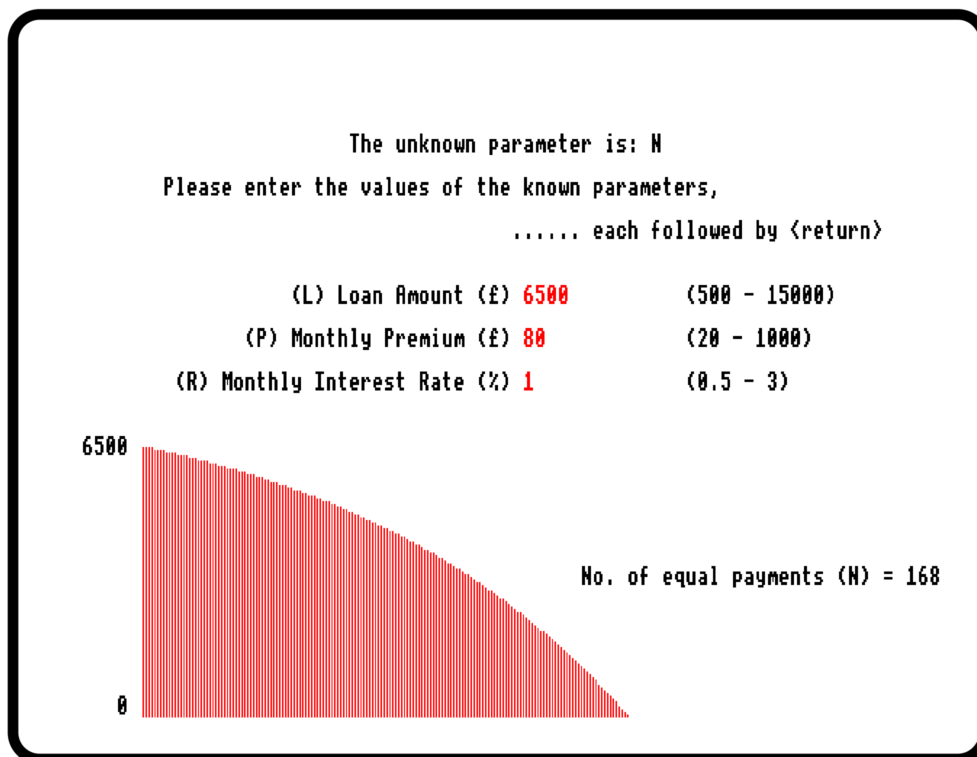


Figure 13.1

A simple graphic enhancement

A small histogram is plotted beneath the input text, showing how the loan amount is gradually reduced to zero as the equal monthly payments are made. The number of payments needed is also printed on the graph. (If the number of payments needed exceeds the top limit set then the graph ends at that limit and you can get a picture of how much of the loan is still outstanding.)

What we've done is mainly to modify `PROCfindNumberOfPayments` to draw a vertical line each time round the `REPEAT ... UNTIL` loop. The length of the line represents the amount of the loan still outstanding at that pass. By plotting each line slightly to the right of the previous one, a histogram is built up. Let's go through the action in detail.

Line 3080 picks a convenient height for the graph, in OS units; Line 3090 sets the origin and Line 3100 calculates the vertical scale to be applied. We want the graph to be the same vertical height whatever value of loan is chosen by the user, so the scale simply does this. Line 3110 sets a variable for the small horizontal gap between each vertical line.

Lines 3180 and 3200 write text to label a crude scale to the left of the first vertical line of the graph. Note how we've coped with the differing number of digits in the loan value L. The value is converted to a string, and the length of the string is used in the `TAB` statement to offset the text printing start point to the left by the number of digits to be printed - so the number is effectively 'right justified' to a chosen spot. This is a very common routine.

Line 3340 moves the cursor to the chosen y-origin, and also moves it to the right along the x-axis by the value of `Gap%` each time - then Line 3350 draws a vertical line of the correctly-scaled height representing the remaining loan.

Finally, after exit from the loop, a message giving the total number of payments made is written.

Even though this small addition could do with some tidying up, you can see that something simple can be quite effective in presentation. Another topic is needed for some of the tidying up, so we'll pick it up again later.

Main graphics upgrade

With this simple example under our belt, let's try something more substantial. ***`Loan Update13b`*** adds a fairly large upgrade which starts to tackle the problem of finding the interest rate when the user chooses R as the unknown main factor. A graphical solution has been chosen deliberately, to fit our needs at this point.

Firstly, we need to refer again to the formula introduced in Chapter 9:

$$L \div P = B + B^2 + B^3 + B^4 + ... + B^{N-1} + B^N$$

.........(Formula 9.1)

where $B = 100 \div (100 + R)$, and:

L=Loan Amount (£)

P= Amount of each monthly payment (£)

N=Number of equal monthly payments

R=Monthly interest rate (%)

As L, P and N are known in this case, one means of finding R (which appears in each B term) is to pick an arbitrary trial value for R and use this to work out the right-hand side of the equation. The resulting answer is then compared with the known, left-hand side value - and it will either be greater or smaller.

The trial value for R is then adjusted by some rationale (designed to pick a better trial value) and the process is repeated - as many times as is necessary - until the right-hand side answer is very close to the left-hand side value (i.e. the target value). This gives a solution for R, to whatever degree of closeness you care to specify.

Further, if we also plot a graph (of the cumulative build-up of the right-hand side) for every trial value of R chosen, we will see these trial values producing end results which progressively converge on the left-hand side (target) value.

Once again, as this is a Beginners' Basic book rather than a Finance book, let's concentrate on the programming and not the maths!

Our process, in pseudo code, will be:

```
Decide accuracy required of final result and set other
     initial values.
Draw and label graph axes, using value of left-hand
     side of equation as our 'target' value.
Pick 1st trial value for R
REPEAT
     Calculate (and draw graph) of result using trial
     R value
     Compare result with left-hand side value (and
     with any other trial results so far)
     Adjust trial value of R if need be
```

```
UNTIL right-hand result is within chosen accuracy
     limits of left-hand side.
Display result.
```

To keep the action clear, ***Loan Update13b*** starts this exercise without the pseudo-code `REPEAT ... UNTIL` loop. It draws the graph axes, shows the ‘target’ value and plots a single graph using only the first trial interest rate. The result is shown in Figure 13.2. Once again, the update listing repeats a few lines for clarity:

Loan Update13b

```
10 REM> Loan13b
20 REM** Upgraded from Loan13a **
480 :
490 GCOL GraphScreenBackgroundCol% :REM** Sets graphic
     screen background to blue, but doesn't clear it
     to that colour yet **
630 MiscTextCol% = 6 :REM** Cyan **
650 GraphScreenBackgroundCol% = 132 :REM** Blue **
680 Graphic3Col% = 6 :REM** Cyan **
700 AxesCol% = 7 :REM** White **
3650 :
3660 DEF PROCfindInterestRate
3670 :
3680 CLG :REM** Clear graphic screen to already defined
     background colour **
3690 Target = L / P :REM** 'Aiming point' for graph -
     see text **
3700 :
3710 PROCaxes(Target)
3720 PROCplots(Target)
3730 :
3740 ENDPROC
3750 :
3760 REM****************************************
3770 :
3780 DEF PROCaxes(TargetLine)
3790 :
3800 REM** Define position and length of graph axes, in
     OS units **
3810 Xorigin% = 100 : Yorigin% = 100
3820 Xend% = 1100 : Yend% = 1000
```

```
3830 :
3840 REM——————————————
3850 :
3860 REM** Calculate horizontal & vertical scales. Make
vertical scale 20% longer than 'target' **
3870 HorScale = (Xend% - Xorigin%) / N
3880 VertScale = (Yend% - Yorigin%) / (1.2 *
TargetLine)
3890 :
3900 REM——————————————
3910 :
3920 REM** Draw axes and 'target' line, with labels. **
3930 GCOL AxesCol%
3940 MOVE Xend% , Yorigin%
3950 DRAW Xorigin% , Yorigin%
3960 DRAW Xorigin% , Yend%
3970 MOVE (Xorigin% + 500) , (Yorigin% + (VertScale *
TargetLine))
3980 DRAW BY (Xend% - Xorigin% - 500 ) , 0
3990 :
4000 PRINT TAB(70 , 5)" Target";
4230 :
4240 Xprint% = 60 : Yprint% = 21
4350 :
4360 COLOUR NormalTextCol%
4370 Trial$ = "Trial rate = "
4380 PRINT TAB(Xprint% - (LEN(Trial$) , Yprint% - 5)
     Trial$
4680 :
4690 GCOL NormalTextCol%
4700 :
4710 ENDPROC
4720 :
4730 REM****************************************
4740 :
4750 DEF PROCplots(TargetLine)
4760 :
4780 LineColour% = Graphic3Col%
4790 :
4830 TrialRate = (100 * P / L) - (100 / N)
```

```
4930 :
4940 COLOUR MiscTextCol%
4950 :
4960 PRINT TAB(Xprint% , Yprint% - 5) "
4970 PRINT TAB(Xprint% , Yprint% - 5) ;
     FNnumberToString(TrialRate , FALSE)
4980 :
5060 :
5070 B = 1 / (1 + (TrialRate / 100))
5080 :
5090 PROClinePlot(LineColour%)
5100 :
5270 COLOUR NormalTextCol%
5280 :
5290 ENDPROC
5300 :
5 310 REM* ***************************************
5320 :
5330 DEF PROClinePlot(Colour%)
5340 REM** Calculates 'N' cumulative summations of 'B'
     terms and plots them, see text. **
5350 :
5360 GCOL Colour%
5370 MOVE Xorigin% , Yorigin%
5380 CumVert = 0 :BTerm% = 0
5390 :
5400 REPEAT
5410 BTerm% = BTerm% + 1
5420 CumVert = CumVert + (B ^ BTerm%) :REM**
     Accumulates sum of B terms **
5430 :
5440     PLOT &15 , (Xorigin% + BTerm% * HorScale) ,
     (Yorigin% + CumVert * VertScale) :REM** Draws
     dotted line from previous level to new cumulative
     level.
5450 :
5530 :
5540 UNTIL ( BTerm% = N ) OR ( CumVert > (1.2 *
     TargetLine) ) :REM** Last term ends graph
     plotting if 'CumVert' goes off top of graph
```

```
      before 'N' terms.
 5550 :
 5560 ENDPROC
```

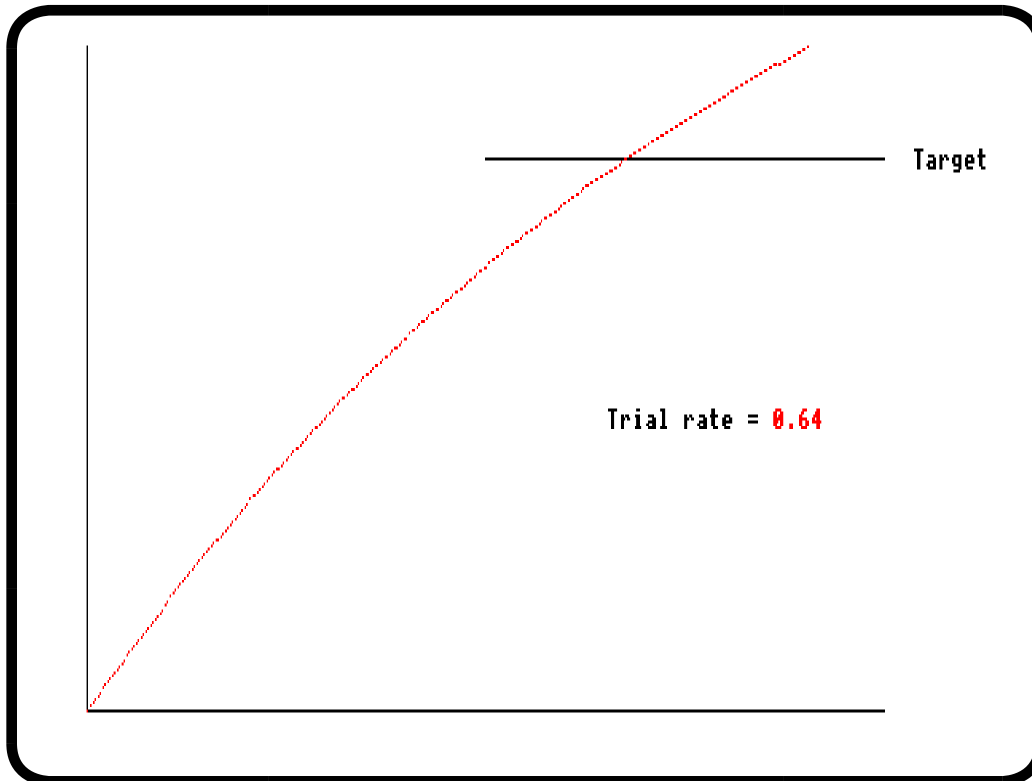


Figure 13.2

Graph of first trial interest rate

`DEF PROCfindInterestRate` at Line 3660 kicks things off by assigning the value of L/P to a variable `Target`. It then calls `PROCaxes()` followed by `PROCplots()` - passing `Target` as a formal parameter in both cases.

`DEF PROCaxes()` is straightforward and the `REMs` in the listing should remove any difficulties. The x-axis of the graph will represent the number of B terms being summed on the right-hand side of the equation and the y-axis will represent the cumulative sum of the value of those terms. So, after, say, six B terms have been summed, a graph showing the rise of the first six cumulative sums will have been drawn - and after all B terms (i.e. N of them) have been summed, the graph will have arrived at its highest and right-most position.

After drawing the main graph axes (Lines 3920-3960) using the OS limits set at Lines 3810-3820, a horizontal line is drawn (Lines 3970-3980) starting 500 OS units to the right of the x-axis and continuing to the left-hand end of the graph - at a y position corresponding to the value of `Target`.

Due to the effects of `VertScale` and `HorScale` (Lines 3870-3880), the target line will always be in the same position - and the horizontal extent of the graph will always be the same, whatever the value of N i.e. the graph is normalised in both axes. The `1.2` in the y-axis scale expression at Line 3880 has the effect of putting the top of the graph a small distance above the target line, so that we will be able to see the effect when our trial rate causes the graph to exceed this level. Some text labels are also added.

`DEF PROCplots()` does the calculation and draws the graph. Line 4830 sets the initial trial interest rate. *(Once again, don't get hung up on the maths of this line. It doesn't matter too much from a programming viewpoint!)* We then print this rate, work out B and call `PROClinePlot` to do the summation of the B terms and plot the corresponding graph line while it is doing so.

You will notice that `DEF PROClinePlot` (Line 5330 onwards) contains some instructions similar to `DEF FNsummation()`. It zeros the accumulation variables and puts the graphics cursor at our axes origin before entering a `REPEAT ... UNTIL` loop which repeatedly accumulates the sum of the B terms.

Each time round the loop `PLOT &15` is used to draw straight, dotted lines between successive (absolute) values of the cumulative `CumVert` value, corresponding (in the `x` direction) to the number of terms so far accumulated by the loop. As the straight lines are short, the effect looks like a curve - and the higher the value of N, the shorter the lines and the more of them.

The loop ends (at Line 5540) when all N terms have been accumulated, or when the graph 'goes off the top' if this occurs first. Without this latter refinement, a `FOR ... NEXT` loop would have been possible.

> *The one graph line drawn with the first trial rate will normally end up (i.e. at its right-hand end) higher than the target line - as in Figure 13.2. The aim of our next steps will he to adjust the trial rate successively until the end of the plotted graph line gets so close to the target line that we are prepared to accept that value as 'equal to the target'.*
>
> *Before upgrading the Loan program further it will be helpful to have a look at the* `VDU5` *command, which allows us to place text on a graphic screen in a better way than we have used so far.*

14. Text and graphics viewports and the use of VDU 5

Separate text and graphics viewports - Use of VDU 5 for text in graphic viewport - Detailed comparison of VDU 4 and VDU 5 text operations with demonstration - Defining text/graphics viewports with VDU 24 and VDU 28 - Incorporation into Loan program upgrade - Solving calculations by successive approximation - More graph plotting.

This chapter starts to introduce the keyword VDU - which covers a vast range of (mainly) visual effects. It is always in the form `VDU xx` (where `xx` is a number in the range 0-255, and the space between `VDU` and `xx` is optional) and often needs further numbers attached to it as parameters. We will be devoting a later chapter to this keyword alone, but for now only five VDU commands are introduced and are also demonstrated by ***Program Prog14a*** which is listed a little later.

Separate viewports

So far, we've used text in the conventional non-Wimp way - positioning it on the screen using line and character spacings counting from the top left-hand corner - and helped by the `TAB` keyword and other 'modifiers' when necessary. There's nothing wrong with this method, but it has its limitations.

It is important to realise that, so far, when we have put text or graphics on the screen, we have in fact been putting the text on the 'text screen' and the graphics on the 'graphics screen'. There are, in effect, two separate 'screens' which can be regarded as separate entities by the programmer - each with its own viewing window onto the screen. *Acorn prefers to use the term "viewport". no doubt to avoid clashing with Wimp "windows".*

Under default conditions, both text and graphics viewports are identical in size and position (both filling the screen). Therefore, under default

conditions, text and graphics appear together, although they are in fact overlaid. A little later we shall see how to change these viewport sizes and the separate text and graphic viewports will then become self-evident.

VDU 4 and 5

Having separate text and graphics viewports raises the question of how to get both text and graphics onto one viewport. Fortunately, BBC Basic is very flexible.

If we issue the Basic instruction `VDU 5`, the text cursor is, effectively, made to move to the graphics viewport and to coincide with the graphics cursor. The text cursor is then inseparable from the graphics cursor - and just as invisible. Text can now be written on the graphics screen and positioned (more precisely than in the text viewport) with graphics statements e.g. move.

When we've finished, `VDU 4` separates the cursors again, putting the text cursor back to the text viewport - and to the exact position it was when the `VDU 5` command was issued.

Once the `VDU 5` instruction has been issued, the text colour control/plotting actions *(see Chapter 12)* are both determined by the graphics colour/plotting actions in force - so, for instance, to set the text colour you now need to use the `GCOL` keyword.

Despite this change of text controlling actions, many of the usual text cursor control commands e.g. line feed, carriage return, `TAB`, etc. all still move the combined text/graphics cursor by the correct amounts for the display mode being used - but the text no longer scrolls when the side or bottom of the viewport is reached.

For instance, let's assume you have already issued the `VDU 5` instruction and the combined text/graphics cursor happens to be somewhere near the centre of the screen. If you now issue the instruction:

```
PRINT "abc"
```

(i.e. without a semi-colon or other cursor-controlling commands) the combined text/graphics cursor will print `"abc"` at the current cursor position and then jump to the left-hand edge of the viewport one line down - just as it would do in the text viewport.

However, when you are using `VDU 5`, this feature will possibly not be what you want - so you will find that you will be using the semi-colon at the end of a print statement very much more, to hold the combined cursor at the end of the printed text.

Better still, you will get into the habit of moving the combined text/ graphics cursor exactly to where you want it before each text/graphic printing/plotting action. This is a far safer practice, because it then doesn't normally matter where the cursor ends up afterwards.

Similarly, when using `VDU 5`, you need to be a little careful about using `TAB` (with a single number). As with the normal text screen, if the cursor is already past your `TAB` number, it will jump to the next line before printing starts.

For these reasons it is less common to see text cursor control commands, apart from the semi-colon perhaps, being used after a `VDU 5` has been issued.

Text character position

Using `VDU 5`, there is an immediate difference seen in the way the text position is defined - compared to what we are used to in the text viewport - because the graphics origin (point 0,0) is at the bottom left-hand corner of the default screen and positive y-axis values move the cursor upwards.

A second, consequential, difference is that text characters are now printed with the **top** left-hand corner of their 'character grid' at the (invisible) graphics cursor position. *(The 'character grid' can best be thought of as a standard rectangular grid of pixels used to define the letter shapes.)* In the text viewport, we are used to the (visible) cursor being positioned in line with the **bottom** of the text characters.

Printing text at graphics point (0,0) will demonstrate these differences - and this is included in the demonstration program ***Program Prog14a*** a little later. You will only just see the tops of some letters. (If you are puzzled why you can see the tops of some letters, rather than nothing at all, remember that plotting point (0,0) actually plots the the bottom left-hand pixel on the screen. Refer back to Chapter 12 about screen resolution, pixel sizes and the graphics plotting grid.)

Overwriting

With `VDU 5`, if you overwrite one piece of text by another, the original will not be deleted - it will be, literally, overwritten. This enables some good text effects to be achieved e.g. repeating some words in a different colour and offsetting them slightly produces shadow effects. Again, this is demonstrated in ***Program Prog14a***.

Defining viewports (Instructions VDU 24, 26 and 28)

We can very easily redefine the graphic and text viewports - using `VDU 24` and `VDU 28` respectively. They both work in a similar manner, but there are differences.

VDU 28

`VDU 28` takes the form:

```
VDU 28 , LeftmostTab% , BottommostTab% , RightmostTab%
     , TopmostTab%
```

Note that there are four numbers after `VDU 28` and they are separated by commas. These numbers, as suggested by the variable names used above, define the position and size of the rectangular text viewport. The values are in text coordinates i.e. character `TAB` and line positions on the default viewport - and the resulting rectangle will include the positions given. So:

```
VDU 28 , 5 , 10 , 15 , 0
```

will change the text viewport to a rectangle 11 text characters wide (from tab position 5 to tab position 15, both inclusive, on the default screen) and 11 lines high (from line 0 to line 10 inclusive).

This command only allows you to redefine a **smaller** text viewport somewhere within the default text viewport. If you use a number larger than the maximum characters-per-line or lines-per-page available in the screen mode in use, then the command will revert to the default viewport. *(As this happens without an error message the cause can be less than obvious sometimes. Be particularly careful if you develop a program in one mode and change to a different mode later.)*

Having redefined the text viewport, all subsequent text printing operations will occur in the new viewport and its top left-hand corner will be the new tab position (0,0). Normal scrolling will occur at the new viewport boundaries. Anything already on the screen before the `VDU 28` instruction and outside the new viewport will remain on screen. *This is particularly useful if you wish to keep the column headings of a scrolling screen in view.*

There is nothing to stop you redefining the text viewport in this way as often as you like and many programs do this.

It really is as simple as that but don’t forget that having issued such an instruction the new viewport will not be self-evident on the screen unless, for instance, you then redefine its background colour.

VDU 24

`VDU 24` takes the form:

```
VDU 24 , LeftmostPoint% ;BottommostPoint%
      ;RightmostPoint% ;TopmostPoint% ;
```

Note the semi-colons after the first comma - and the extra semi-colon at the end. *(Here, the semi-colon indicates that the preceding number needs to be handled by the computer as 2 bytes rather than the usual 1 byte - because OS values can exceed 255. See Appendices 5 and 8.)* This time the values are in normal graphics OS units and again you can only redefine a graphics viewport smaller than and within the default screen size for the screen mode in use.

With `VDU 24`, the graphics origin is not automatically changed. Graphics commands will act exactly as before but only those actions which cause plotting within the redefined window will actually become visible - so the effect is very much more like a smaller window on a larger screen.

VDU 26

`VDU 26` simply returns everything to the default viewports (and returns graphics origin to bottom left-hand corner if it has been changed).

Final point

If you issue a `VDU 5` command in a non-Wimp program, then ensure that your error traps issue a `VDU 4` before they report an error to the screen. Otherwise an error occurring while the `VDU 5` is in force might well print its error message outside the viewport and you will have no idea that an error has occurred, nor what's gone wrong. Similarly, set/reset the text colour to ensure the error message will be visible!

Demonstration program

The points covered above are demonstrated in ***Program Progl4a*** - in rather lurid colours. The program allows you to step through each section in sequence, by pressing any key - and the REMs suggest some changes to make to explore the effects. It is best to refer to the listing and its REMs while stepping through the running program. Figure 14.1, after the listing, shows the screen as it is just before the program ends.

Program Progl4a

```
10 REM> Progl4a
20 REM** Demonstrates use of VDU5 to place text on
graphic screen. **
30 REM** Program steps through by pressing any key
     after each pause. **
```

```
40 REM** Read REMs for fuller details of effects
     demonstrated. **
50 :
60 MODE 12
70 ON ERROR REPORT : PRINT" at Line " ; ERL : END
80 :
90 COLOUR 131 : CLS :REM** Clears text screen to Yellow
     background. **
100 COLOUR 4 :REM** Text foreground colour, Blue. **
110 PRINT "Normal text printing, note flashing cursor."
120 :
130 pause% = GET
140 :
150 REM****************************************
160 REM** Next section shows effects of TAB() etc. when
     using VDU5. **
170 REM****************************************
180 :
190 VDU5 :REM** Engage 'Text at graphics cursor' **
200 :
210 GCOL 0 :REM** Set text/graphic plotting colour to
     0, i.e. black. **
220 :
230 PRINT TAB(0,2)"1st line of VDU5 printing, at
     TAB(0,2) - note flashing cursor gone." : REM**
     TAB() positions text as normal. **
240 PRINT "2nd line starts where combined text/graphic
     cursor was left after first line." : REM** Again
     as normal, this starts at LH edge, one line down.
     **
250 :
260 pause% = GET
270 :
280 PRINT
290 PRINT TAB(75)"3rd";CHR$(10); : REM** These two
     lines act as normal, with the two ";" holding the
     cursor at the end of the text - and CHR$(10)
     shifting cursor down one line from there. Note
     text does not scroll at edge. **
300 PRINT "4th"
```

```
310 PRINT TAB(55)"Note no edge scrolling."
320 PRINT TAB(43)"Missing text written off visible
     screen."
330 PRINT
340 :
350 pause% = GET
360 :
370 PRINT TAB(65)"5th at TAB(65)";
380 PRINT TAB(66)"6th at TAB(66)" : REM** Cursor is
     already past 66 (held at 78 by above ";" ) so
     printing is forced to next line, as normal. **
390 PRINT TAB(50) "'6th' forced to next line"
400 PRINT TAB(50) "as TAB(66) already passed."
410 :
420 pause% = GET
430 :
440 MOVE 0 , 200
450 PRINT "Note tops of letters 'A B C V', but not
     'c',"'"just showing in LH corner below"'"after
     printing at graphics point (0,0)."''"(Note
     flashing text cursor back at its last position
     after VDU4 at Line 490)"
460 MOVE 0,0
470 PRINT"ABcCV" :REM* * Only tops of A B C and V will
     show - proves top LH corner of text characters
     placed at cursor. **
480 :
490 VDU4 :REM** Put text cursor back to text viewport.
     **
500 :
510 pause% = GET
520 :
530 REM****************************************
540 REM** Next section shows overprinting effect when
     using VDU5. **
550 REM****************************************
560 :
570 VDU23,17,7,2,16;32;0;0; : REM** Sets text
     characters to size 16 by 32 pixels, instead of
     their normal 8 by 8. **
```

```
580 :
590 Shadowl$ = "l s t "
600 Shadow2$ = "Basic"
610 Offset% = 160
620 :
630 VDU 5
640 :
650 GCOL 5 : REM** Sets graphics colour to magenta. **
660 MOVE Offset% , 680 : REM** Moves graphics cursor to
     required position before printing text. **
670 PRINT Shadowl$ + Shadow2$ : REM** Prints both text
     in magenta. **
680 :
690 pause% = GET
700 :
710 GCOL 3 :REM** Sets graphics colour to yellow, same
     as background. **
720 MOVE Offset% , 680 :REM** Put cursor at same
     starting point. **
730 PRINT Shadowl$ ; :REM** Overprints lst string in
     background colour, effectively making it
     disappear. Cursor held ready at start of 2nd
     string. **
740 :
750 MOVE BY - 4 , - 4 : REM** Move held cursor a little
     to left and below. **
760 GCOL 4 : REM** Change colour to Blue. **
770 PRINT Shadow2$ : REM** Print 2nd string again. **
780 :
790 VDU 4
800 :
810 VDU23,17,7,2,8;8;0;0; : REM** Restore text
     characters to normal(8by8). **
820 :
830 pause% = GET
840 :
850 REM*****************************************
860 REM** Next section compares scrolling effects of
     VDU 4 and VDU5. **
870 REM*****************************************
```

```
880 :
890 REM
900 REM** First, VDU 4 (i.e. normal) text printing. **
910 REM
920 :
930 VDU28,0,31,39,16 : REM** Defines and creates text
     viewport in bottom LH quarter of screen.
     (Effectively overlaying TAB positions 0-39 and
     lines 16-31 inc. of full screen). **
940 COLOUR 134 : CLS : REM** Clears text viewport to
     Cyan background. **
950 COLOUR 4 : REM** Sets text colour to Blue. **
960 :
970 FOR Line% = 0 TO 14 : REM** Note only 15 lines as
     cursor will end at start of next (16th) line and
     would cause scroll if 0-15 used. **
980     IF Line% = 7 THEN
990       PRINT STRING$( 4 , "0123456789" ) : REM**
     Prints 40 characters. **
1000      REMPRINT STRING*( 5 , "0123456789" ) : REM**
     Prints 50 characters. Used instead of previous
     line to force edge and bottom scroll. **
1010    ELSE
1020       PRINT "Text Line " + STR$( Line% ) : REM**
     Shows line numbers. Note that Line0 is at top of
     viewport. Note that printing scrolls at RH edge
     and bottom when Line 1000 above is used instead
     of Line 990. **
1030    ENDIF
1040 NEXT
1050 :
1060 pause% = GET :REM** Note location of flashing text
     cursor. **
1070 :
1080 REM———————————————
1090 REM** Second, VDU 5 (i.e text printing at graphics
     cursor). **
1100 REM———————————————
1110 :
1120 VDU24,640;0; 1279;511; : REM** Defines and creates
```

```
     graphic viewport in bottom RH quarter of screen.
     (Effectively overlaying TAB positions 40-79 and
     lines 16-31 inc. of full screen). **
1130 GCOL 132 : CLG : REM** Clears text viewport to
     Blue background. **
1140 GCOL 6 : REM** Sets text colour to Cyan. **
1150 VDU 5
1160 :
1170 REM** Next sequence is same as for text viewport
     above, but using VDU5 printing and graphics
     positioning commands. Compare lines with above,
     particularly size and direction of positioning
     moves in Lines 830/840. **
1180 :
1190 FOR Line% = 0 TO 16 : REM** Note extra line used,
     as no scrolling.
1200     :
1210     REM** Compare effect of using next line
     instead of the one after. It starts lines one
     pixel higher and just shows tops of characters in
     Text Line 16 - incidentally proving no scroll. **
1220     REMMOVE 640 , ( (16-Line%) * 32 )
1230     MOVE 640 , ( (16-Line%) * 32 ) - 1 : REM**
     Positions text correctly. Demonstrates care
     needed in placing top LH corner of text character
     in right place. (See article) **
1240     IF Line% = 7 THEN
1250         PRINT STRING$( 4 , "0123456789" )
1260         REMPRINT STRING?( 5 , "0123456789" )
1270     ELSE
1280         PRINT "Graphics Text Line " + STR$( Line%
     ) : REM** Note no bottom or edge scroll. **
1290     ENDIF
1300 NEXT
1310 :
1320 REM****************************************
1330 REM** Shows that VDU4 returns text cursor to place
     it was in text viewport when VDU 5 was called. **
1340 REM****************************************
1350 :
```

```
1360 pause% = GET
1370 VDU 4
1380 pause% = GET:REM** Holds program at text cursor,
     awaiting any keyboard press. **
1390 :
1400 REM——————————————
1410 :
1420 COLOUR 128 : COLOUR 7 : REM** Restore text colours
     to normal.
1430 VDU 26 : REM** Restore text and graphics viewports
     to defaults.
1440 END
```

```
Normal text printing, note flashing cursor.

1st line of VDU5 printing, at TAB(0,2) - note flashing cursor gone.
2nd line starts where combined text/graphic cursor was left after first line.

                                                                  3rd
                                                                    4t
                                                   Note no edge scrolling.
                                          Missing text written off visible scre

                                                             5th at TAB(65)
                                                              6th at TAB(66)
            Basic                                      '6th' forced to next line
                                                       as TAB(66) already passed.
Text Line 0                              Graphics Text Line 0
Text Line 1                              Graphics Text Line 1
Text Line 2                              Graphics Text Line 2
Text Line 3                              Graphics Text Line 3
Text Line 4                              Graphics Text Line 4
Text Line 5                              Graphics Text Line 5
Text Line 6                              Graphics Text Line 6
01234567890123456789012345678901234567890123456789012345678901234567890123456789
Text Line 8                              Graphics Text Line 8
Text Line 9                              Graphics Text Line 9
Text Line 10                             Graphics Text Line 10
Text Line 11                             Graphics Text Line 11
Text Line 12                             Graphics Text Line 12
Text Line 13                             Graphics Text Line 13
Text Line 14                             Graphics Text Line 14
                                         Graphics Text Line 15
```

Figure 14.1

Final screen of program 'Prog14a'.

Back to the Loan program ...

This time we are going to upgrade the Loan program twice: firstly just to show how `VDU 5` is used in practice by modifying some of the labelling on the current graphs - and then to take the program a substantial stage further.

Firstly then, ***Loan Update14a*** makes some changes to the labelling on the two graphs as currently available. That is, only `DEF`

`PROCfindNumberOfPayments` and `DEF PROCaxes(TargetLine)` from ***Loanl3b*** are changed.

Loan Updatel4a

```
10 REM> Loanl4a
20 REM** Upgraded from Loanl3b **
340 VDU4
670 Graphic2Col% = 3 :REM** Yellow **
690 Graphic4Col% = 7 :REM** White **
3120 CharWidth% = 16
3130 :
3140 VDU5
3150 :
3160 GCOL NormaITextCol %
3170 MOVE Xorig% - 16 - CharWidth% , Yorig%
3180 PRINT "0"
3190 MOVE Xorig% - 16 - (CharWidth% * LEN(STR$(L))) ,
     Yorig% + (L * Scale) + 16
3200 PRINT STR$(L)
3390 GCOL NormaITextCol%
3410 PRINT " No. of payments (N) > ";STR$(NumberUpper);
3430 PRINT * No. of equal payments (N) =
     ";STR$(NumOfPayments%) ;
3450 :
3460 VDU4
3990 VDU5
4000 PRINT " Target";
4010 MOVE BY -80 , 144
4020 GCOL MiscTextCol%
4030 PRINT "Rate Too" ;
4040 MOVE BY -96, -48
4050 PRINT "Low" ;
4060 MOVE BY -80 , -192
4070 PRINT "Rate Too" ;
4080 MOVE BY -96 , -48
4090 PRINT "High"
4100 VDU4
4110 :
4120 REM
4130 :
```

Note how the new variable `CharWidth%` (Lines 3120, 3170 and 3190) is used to offset the text to the desired positions. `CharWidth%` is assigned the value 16 because we using a display mode which is 1280 OS units wide and can have 80 text characters across it. i.e. 1280 / 80 = 16 OS units per character. There are ways to set this offset automatically to take account of any display mode, but that is beyond us yet.

Also, in Lines 4000-4090, note how relative movement (using `MOVE BY`) makes it much easier to position text which needs to be kept together. By changing only the initial cursor position (here set by the end of the previous plotting action of Line 3980 in ***Loanl3b***), the whole block of text can then be moved keeping the same relative positions.

Getting the program to converge on the answer

With the above small change in place we can pick up the main flow again where we left it in the previous chapter.You will recall that we left the Loan program drawing a single graph using a trial interest rate.

In our second upgrade in this current chapter, ***Loan Updatel4b*** takes a significant step forward by putting the number of graphs to be plotted under the control of a `REPEAT ... UNTIL` loop - together with some simple algorithms to adjust the trial interest rate. The result is that the graph eventually 'homes in' on the target line.

To make things clearer, the colour of the graph changes when it's 'getting near' - and the graph line of the final result is plotted in yet another colour.

Loan Updatel4b

```
10 REM> Loanl4b
20 REM** Upgraded from Loanl4a **
3700 Accuracy = 0.1 :REM** Required result accuracy, in
     percent **
4140 REM** Draw short bracketing lines above and below
     target line to represent 'getting near' box (10
     times larger than required accuracy) and over the
     final 5% of the x-axis. **
4150 GCOL Graphic2Col%
4160 MOVE Xend% , (Yorigin% + (VertScale * (1 +
     (Accuracy / 10)) * TargetLine))
4170 DRAW BY (-(N - INT( 0.95 * N)) * HorScale) , 0
4180 DRAW BY 0 , (-VertScale * (2 * (Accuracy / 10)) *
```

```
     TargetLine)
4190 DRAW BY ((N - INT( 0.95 * N) ) * HorScale) , 0
4200 :
4210 REM————————————————
4220 :
4230 REM** Print other graph text/labels. (Text
     printing OK in this case) **
4250 COLOUR EmphasisTextCol%
4260 :
4270 Loan$ = "Loan = £"
4280 PRINT TAB(Xprint% , Yprint%) Loan$ ;
     FNnumberToString(L , FALSE)
4290 :
4300 Premium$ = "Monthly Premium = £"
4310 PRINT TAB(Xprint% - (LEN(Premium$) - LEN(Loan$)) ,
     Yprint% + 2) Premium$ ; FNnumberToString(P ,
     FALSE)
4320 :
4330 Number$ = "No. of Payments - "
4340 PRINT TAB(Xprint% - 1 - (LEN(Number$) -
     LEN(Loan$)) , Yprint% + 4) Number$ ; N
4770 GettingNear% = FALSE
4790 :
4800 REM————————————————
4810 :
4820 REM** Calculate a suitable interest rate for first
     trial graph and set starting values **
4840 HighRate = TrialRate
4850 LowRate = TrialRate
4860 CumTooHigh% = FALSE
4870 CumTooLow% = FALSE
4880 :
4890 REM————————————————
4900 :
4910 REM** Enter graph plotting loop. Trial rate is
     adjusted after each graph - until graph ends
     within required accuracy of target line **
4920 :
4930 REPEAT
4990     VDU30
```

```
5000     COLOUR NormalTextCol%
5010 :
5020     REM** Change graph colour if within 'getting
     near' range **
5030     IF ( GettingNear% = TRUE ) THEN LineColour% =
     Graphic4Col%
5040 :
5050     PROCpause(100)
5100 :
5110     IF ( BTerm% = N ) AND ( CumVert > (1 -
     (Accuracy / 10)) * TargetLine ) AND ( CumVert <
     (1 + (Accuracy / 10)) * TargetLine ) THEN
     GettingNear% = TRUE
5120 :
5130     IF ( CumTooHigh% = TRUE ) AND ( CumTooLow% =
     TRUE ) THEN PROCconverge ELSE PROCbracket
5140 :
5150 UNTIL (BTermi = N) AND ( CumVert > (1 - (Accuracy
     / 100)) * TargetLine ) AND ( CumVert < (1 +
     (Accuracy / 100)) * TargetLine )
5160 :
5170 PROClinePlot(Graphic1Col%) :REM Repeat successful
     graph in contrasting colour.**
5180 :
5190 R = 100 * ((1 / B) - 1) :REM** Necessary as trial
     rate has been updated **
5200 :
5210 COLOUR ActionTextCol%
5220 Result$ = "Monthly Interest Rate"
5230 PRINT TAB(Xprint% - (LEN(Result!) - LEN(Loan$)) ,
     Yprint% - 5) Result$
5240 PRINT TAB(Xprint% - 8 , Yprint% - 3) "=" ;
     FNnumberToString(R , FALSE) ; "% (approx.)"
5250 :
5260 VDU30
5570 :
5580 REM* ***************************************
5590 :
5600 DEF PROCbracket
5610 REM** Seeks to establish a pair of 'too high' and
```

```
     'too low' values, before 'PROCconverge' can be
     used. **
5620 :
5630 IF ( CumVert > TargetLine ) THEN
5640     LowRate = TrialRate
5650     CumTooHigh% = TRUE
5660     TrialRate = TrialRate * 1.5
5670 ENDIF
5680 :
5690 IF ( CumVert < TargetLine ) THEN
5700     HighRate = TrialRate
5710     CumTooLow% = TRUE
5720     TrialRate = TrialRate * 0.75
5730 ENDIF
5740 :
5750 ENDPROC
5760 :
5770 REM****************************************
5780 :
5790 DEF PROCconverge
5800 REM** Progressively narrows range of 'too high'
     and 'too low' values **
5810 :
5820 IF ( CumVert > TargetLine ) THEN LowRate =
     TrialRate
5830 IF ( CumVert < TargetLine ) THEN HighRate =
     TrialRate
5840 TrialRate = (HighRate + LowRate) / 2
5850 ENDPROC
5860 :
5870 REM****************************************
5880 :
6100 DEF PROCpause(Time%)
6110 REM** Pauses program for "Time%' centi-seconds. **
6140 :
6150 StartTime% = TIME :REM** Reads the current time
     into 'StartTime%'
6160 :
6170 REPEAT
6180 UNTIL ( TIME > StartTime% + Time% ) :REM** Exits
```

```
      when TIME has increased by more than 'Time%'
      centiseconds.
 6190 :
 6200 ENDPROC
```

The first step is to decide what accuracy we will accept for the final interest rate answer - and Line 3700 declares a new variable Accuracy and sets this, arbitrarily, to 0.1%.

Then `PROCaxes()` is amended (Lines 4140-4190) to draw two short lines bracketing the target line near its right-hand end, to represent a 'getting near' area. These lines are each separated from the target line by 10 times the accuracy limit and are drawn over approximately the last 5% of the x-axis. Again, both these distances are arbitrary.

The changes to def `PROCplots()` in order to draw multiple graphs may look a little complicated but the action is straightforward if taken in the right order. Lines 4930 and 5150 install the `REPEAT ... UNTIL` loop around the already-existing graph plotting instructions and (at Line 5130) the interest rate adjusting mechanism is called after each graph is drawn.

This is a two-stage mechanism which uses four new variables (Lines 4840-4870) to keep track of the effect of the different trial rates. At the start, as can be seen, both `HighRate` and `LowRate` are set to the initial trial interest rate and the flags `CumTooHigh%` and `CumTooLow%` are both set to `FALSE`.

Line 5130 will initially cause `PROCbracket` to be called, rather than `PROCconverge`. `PROCbracket` varies the trial rate fairly coarsely, until the program registers that it has encountered two trial values which straddle the target, i.e. it has a trial rate value in the variable `LowRate` which is known to be 'too low' and another trial rate value in `HighRate` which is known to be 'too high'. This condition occurs - see `DEF PROCbracket` - when **both** `CumTooHigh%` and `CumTooLow%` become `TRUE`.

For the first few times round the loop, both `LowRate` and `HighRate` are likely to be either 'too high' or 'too low' - and hence only one of `CumTooHigh%` and `CumTooLow%` will be `TRUE`.

However, the effect of Line 5660 and/or Line 5720 will soon cause the graph to "cross over to the other side" - making both `CumTooHigh%` and `CumTooLow%` become `TRUE`.

Once this bracketing has been achieved, Line 5130 switches the interest rate adjusting mechanism to PROCconverge which - by Line 5840 -

'homes in' the `HighRate` and `LowRate` values until the right-hand end of the graph (i.e. the result of 'N' accumulations of the B terms in the equation) differs from the target value by less than the chosen accuracy.

After each graph is drawn (Line 5090 in ***Loan Update13b***), a check is also made (Line 5110) to see whether the result is within the chosen 'getting near' range. If it is, then next time round (Line 5030) the graph plotting colour is changed from cyan to white.

After the answer has been reached - and hence the loop has been exited - the final graph is **re-drawn** in red *(see later for reason)*. The overall effect (see Figure 14.2) is typically a spray of broadly similar graphs: the cyan-coloured outer ones surrounding some white ones, with a single, red graph among them representing the answer value.

However, this is not always the case, as sometimes the algorithm "gets lucky" and hits the target with very few attempts - whereas, at other times, reaching the 'getting near' criteria doesn't always prevent the next few adjustments to the trial interest rate from going outside the 'getting near' area again, before finally zeroing in.

The best visual effect is obtained by having a reasonably high interest rate (but not too high), which produces nicely curved graphs, not too close together.

Finally, because the plotting action takes place very quickly, a 1-second pause has been added to the `REPEAT ... UNTIL` loop, by `PROCpause()` in Line 5050, which has its `DEF PROC` at Line 6100.

This `PROC` uses the built-in function time which, in this case, simply reads the internal computer clock at the start of a `REPEAT ... UNTIL` loop, then keeps reading the clock again each time round the loop - until the new time differs from the initial reading by more than the number of centi-seconds we've chosen - in this case 100, or 1 second. The `REMs` in `DEF PROCpause()` should enable you to follow this very common timing action.

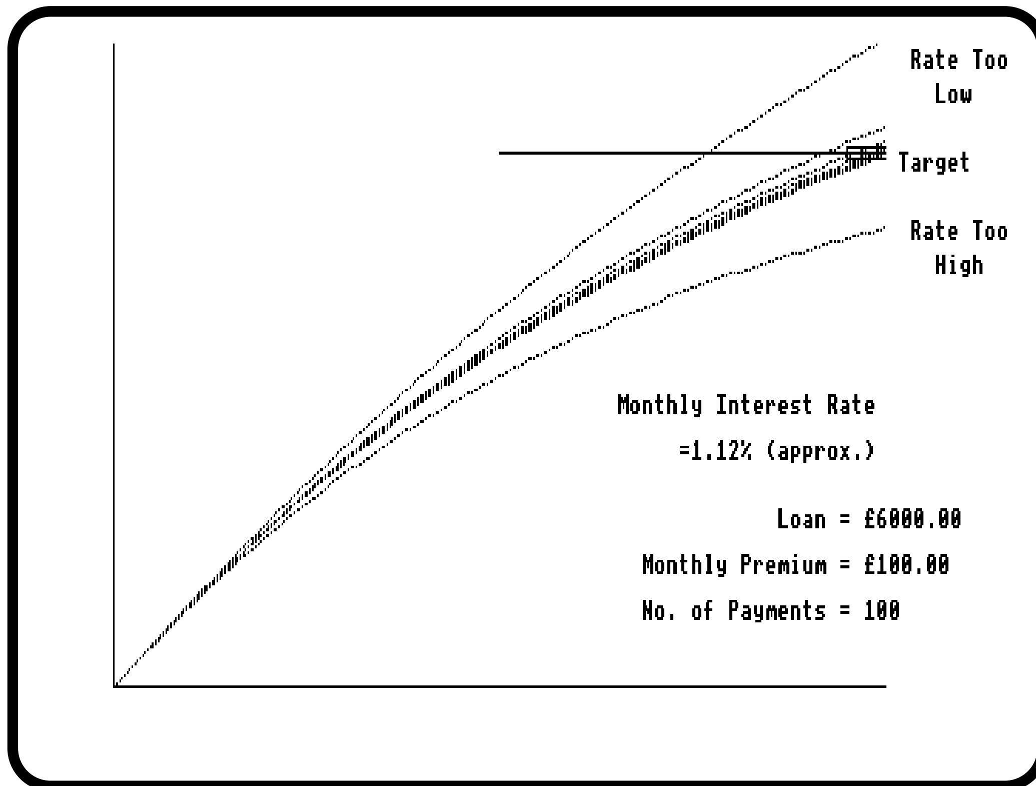


Figure 14.2

Homing in on the target line

There are a few other points worth noting in the listing. Because we need to know the result of one complete accumulation of N terms (i.e. one graph plot) before we know whether or not - and in which direction - to adjust the trial rate, the checks and adjustments need to come near the end of the `REPEAT ... UNTIL` loop. This means that the trial rate is actually adjusted once more by Line 5130, but not used, after the final answer has been achieved, just before the loop is exited.

This could be overcome by an additional flag and condition applying to Line 5130, but to keep things simple it has been counteracted here by recalculating the 'answer rate' after exiting the loop (Line 5190) - using the last-used value of B (which was calculated at the start of each loop and has therefore not been altered again before exit).

This unaltered value of B is also automatically carried forward and used to re-plot the final graph in red (Line 5170). A replot is needed because we didn't know it was the final graph when it was drawn the first time!

You can probably deduce from this particular update to `DEF PROCplots()`, that more than half the battle in getting a loop to function as required is putting the instructions in the right order. Quite often it helps to use pencil-and-paper to write down the loop in pseudo-code and/ or flow chart form.

Finally, Lines 4990 and 5260 each contain a `VDU 30` command, which has not been introduced yet. This simply resets the text cursor to the top left-hand corner of the text viewport (which is the whole screen here - the default). These two instances of `VDU 30` are not actually needed now in this particular program (!) but they were, along with other instances, during its development. So, these two have been left to demonstrate two related programming tips:

- it is good practice to reset the text cursor before starting a routine which includes screen text printing; and
- try to remember to leave the text cursor 'as you would wish to find it' after a routine.

This advice is very similar to that given earlier about always issuing a graphics cursor-placing command before a plot action - they are good habits which will prevent trouble in the long run, even though they may turn out to be unnecessary in the final program in some cases.

Expanding the action...

As it stands, ***Program Loanl4b*** works well enough, but the action at the right-hand end of the graph - in and around the 'getting near' area - is pretty cramped. It would be nice (and a good graphics exercise) to zoom into that area - and this is what happens in the next chapter, together with some other finishing touches.

15. Final steps in Loan project

Multiple graph plotting/scaling - PROC structure and input validation revisited - 'ratcheting' values - Revisiting PROC/FN to introduce local/global variables - Keyword LOCAL - Importance of free-standing PROC/FNs - Repeating a program - Final version of Loan program - Suggested areas for its further development.

In this chapter we will make three upgrades to complete the Loan program. The first will extend the program one more step and the last two will carry out some important 'tidying up'.

Adding a 'zoom box'

As we said in the last chapter, we are going to 'magnify' the action in the 'getting near' area.

The idea is simple enough: we will choose a small part of the screen and use it as a separate graph area to repeat - at a larger scale - the plots in the 'getting near' area i.e. a 'zoom box'.

The chosen location for this zoom box is on the upper left-hand side of the screen, centred vertically around the target line.

With a vertical scale 10 times the main graph scale the result is a zoom box of reasonable size in a fairly unused part of the main graph - with the advantage that we can easily relate the main and expanded views, because they share the same target line (see Figure 15.1).

The necessary additions are in ***Loan Upgrade 15a.***

Loan Upgrade 15a

```
10 REM> Loan15a
20 REM** Upgraded from Loan14b **
4390 :
4400 REM———————————————
4410 :
```

```
4420 REM** Define 'zoom' box position and calculate
     scales for it **
4430 ZoomXorigin% = Xorigin%
4440 ZoomYorigin% = Yorigin% + (VertScale * TargetLine)
4450 :
4460 ZoomXend% = Xorigin% + 400
4470 ZoomHorScale = (ZoomXend% - ZoomXorigin%) / (N -
INT( 0.95 * N))
4480 ZoomVertScale = VertScale * 10
4490 :
4500 REM————————————————
4510 :
4520 REM** Draw and label 'zoom' box **
4530 GCOL NormalTextCol%
4540 MOVE ZoomXorigin% , ZoomYorigin%
4550 DRAW BY (ZoomXend% - ZoomXorigin%) , 0
4560 VDU5
4570 GCOL Graphic2Col%
4580 MOVE BY -336 , 112
4590 PRINT "Zoom x1O"
4600 VDU4
4610 :
4620 MOVE ZoomXorigin% , (ZoomYorigin% - ((Accuracy /
     10) * ZoomVertScale * TargetLine))
4630 DRAW BY (ZoomXend% - ZoomXorigin%) , 0
4640 DRAW BY 0 , ( (2 * Accuracy / 10) * ZoomVertScale
     * TargetLine)
4650 DRAW BY -(ZoomXend% - ZoomXorigin%) , 0
4660 :
4670 REM
4680 :
4690 GCOL NormalTextCol%
4700 :
4710 ENDPROC
5460 REM** Calculates initial x and y coords of zoom
     box plot when main graph crosses zoom box start.
5470 IF ( BTerm% = INT( 0.95 * N ) ) THEN
5480 ZoomStartX = ZoomXorigin%
5490 ZoomStartY = ZoomYorigin% - (TargetLine -
     CumVert) * ZoomVertScale
```

```
5500 ENDIF
5510 :
5520 IF ( GettingNear% = TRUE ) AND ( BTerm% > INT(
     0.95 * N) ) THEN PROCzoomPlot :REM** Note '>'
     sign. **
5890 DEF PROCzoomPlot
5900 :
5910 REM** First store main graph cursor position. **
5920 LastMainX = Xorigin% + (BTerm% * HorScale)
5930 LastMainY = Yorigin% + (CumVert * VertScale)
5940 :
5950 REM** Now move cursor to zoom box and plot one
     section of zoom graph. **
5960 MOVE ZoomStartX , ZoomStartY
5970 PLOT21 , (ZoomXorigin% + (BTerm% - INT( 0.95 * N))
     *_        ZoomHorScale) , (ZoomYorigin% -
     (TargetLine - CumVert) * ZoomVertScale)
5980 :
5990 REM** Store resulting zoom box cursor position
     ready for new starting point next time round.
6000 ZoomStartX = ZoomXorigin% + (BTerm% - INT( 0.95 *
     N)) * ZoomHorScale
6010 ZoomStartY = ZoomYorigin% - (TargetLine - CumVert)
     * ZoomVertScale
6020 :
6030 REM** Finally, restore cursor to stored main graph
     position. **
6040 MOVE LastMainX , LastMainY
6050 :
6060 ENDPROC
6070 :
6080 REM****************************************
6090 :
```

The zoom box definition and drawing follows the same processes as for the main axes, so you should be able to follow the changes to `PROCaxes` (Lines 4420-4650) without difficulty.

To repeat portions of graphs in the zoom box we only need to add lines to `PROClinePlot()`. We don't want to activate the zoom box plotting until, firstly, we have registered that we are 'getting near' and, secondly, we then only want to repeat those portions of the main graph which have x-

values in the last 5% of the x-axis (where the ‘getting near’ lines are drawn). Line 5520 effects these conditions: it is located after the main graph plot has been updated and calls `PROCzoomPlot` when both conditions exist.

We also need to set the **initial** starting point for each zoom box graph - noting that as we are ‘magnifying’ part of the main graphs, each zoom box plot will start at a different vertical place, rather than at the zoom box origin. However, the x-value of this start point will always be the chosen `ZoomXorigin%` value (the graph x-origin in this case).

The y-value of the start of a zoom box plot will be the same value as its main graph counterpart when the latter reaches the left-hand end of the ‘getting near’ zone i.e. the value of `CumVert` when `BTerm%` reaches the start of the ‘getting near’ area. We simply recalculate the position of this value of `CumVert` using the expanded zoom scale and different origin. This is carried out in Lines 5460-5500. *(As listed, this calculation is carried out whether or not we are going to use the zoom box. It does no harm, but adding a further condition to Line 5470 would activate the calculation only when needed.)*

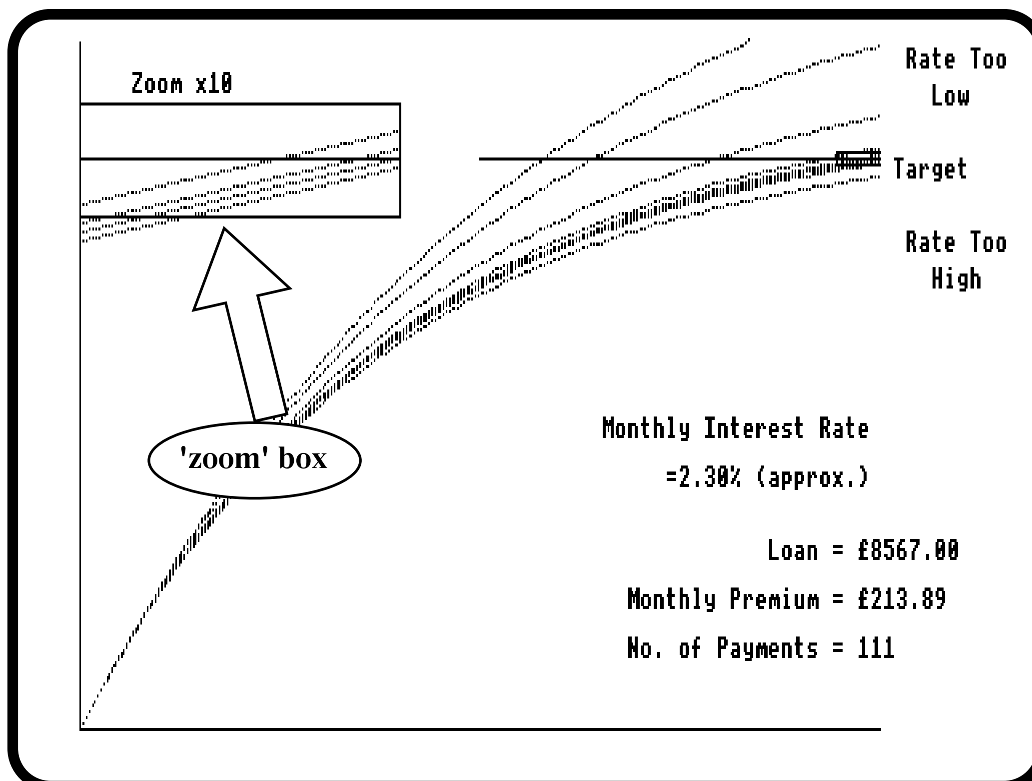


Figure15.1

Addition of 'zoom' box

When called, `PROCzoomPlot` firstly stores the current main graph coordinates (Lines 5920 and 5930). It then moves the cursor to starting coordinates to be used in the zoom box (which will be the above initial values at first), draws a new section of the expanded graph, saves the resulting zoom box cursor end position as the new starting point for the next zoom plot and moves the cursor back to the stored position on the main graph (Line 6040) - ready for the next trip round the `PROClinePlot()` loop.

This rigid sequential process involving storing, updating and restoring of ‘old’ and ‘new’ points on two different plot actions seems to occur quite frequently in graphics - including animations. So it is well worth poring over - and Figure 15.1 will help.

Usefulness of PROC structure

It is also worth pausing here to reflect how relatively easy it has been to incorporate changes into an established program once there is a good underlying proc structure. Indeed you can see that these last changes could have probably been better made by cascading the proc structure even further.

With ***Loan15a*** the functionality of our Loan program is substantially complete. The further changes are more in the nature of ‘tidying up’ - although they are not trivial points.

Input validation revisited

If you have been playing with the program (as you should have been!) you have probably already found out that you can still cause the program some headaches when R, the interest rate, is the unknown factor and certain values are input for the known items. There is a similar problem when N is the unknown. We need to prevent the user from making these undesirable entries, by changing the input limits more dynamically - to respond to the first and/or second inputs made.

Let’s solve the R case first. When the interest rate R is the unknown, any combination of L, P and N which causes (N x P) to be less than L will cause a problem - because there will then be not enough money paid to meet the loan amount, let alone the interest.

There are several ways we could eliminate the problem but as our program always presents the ‘known’ items in the same order for

inputting, we will follow this same sequence for making any adjustments to the input limits.

Thus, we will not alter the limits for the first choice (L, the loan amount), but once a value for L has been input, we will adjust, if necessary, the limits for the value of N. In turn, once L and N have been chosen, we will examine the limits for P.

These changes are made in Loan ***Update 15b***.

Loan Update 15b

```
10 REM> Loan15b
  20 REM** Upgraded from Loan15a **
  1700 RateFlag% = FALSE
  1710 NumberFlag% = FALSE
  1720 IF ( NotThisLetterJ = "R" ) THEN RateFlag% = TRUE
  1730 IF ( NotThisLetter$ = "N" ) THEN NumberFlag% =
       TRUE
  1740 :
  1880 :
  1890 IF ( RateFlag% = TRUE ) THEN
  1900 IF ( INT(L / PaymentUpper) + 1 > NumberLower )
       THEN NumberLower = INT(L / PaymentUpper) + 1
  1910 IF ( INT(L / PaymentLower) - 1 < NumberUpper )
       THEN NumberUpper = INT(L / PaymentLower) - 1
  1920 ENDIF
  1930 :
  1940 IF ( NumberFlag% = TRUE ) THEN
  1950 IF ( L < PaymentUpper ) THEN PaymentUpper = L
  1960 IF ( L * ( RateLower / 100 ) > PaymentLower ) THEN
       PaymentLower = INT( L * (RateLower / 100) ) + 2
  1970 ENDIF
  2020 :
  2030 IF ( RateFlag% = TRUE ) THEN
  2040 IF ( INT(L / N) + 1 > PaymentLower ) THEN
       PaymentLower = INT(L / N) + 1
  2050 ENDIF
```

Firstly, a 'flag' (called `RateFlag%`) is introduced near the start of `DEF PROCinputKnowns()` at Line 1700 and set initially to `FALSE`. If R is chosen as the unknown this flag is changed to `TRUE` in Line 1720. This

gives a simple means of triggering the checks on the input limits.

As soon as a value has been chosen for L, we need to ensure that the lowest limit then made available for N exceeds the value (L ÷ PaymentUpper) - and that the highest limit is lower than (L ÷ PaymentLower). Otherwise, respectively, there may not be enough payments made, or our overall payment limits may be transgressed.

Any adjustment to the limits for N needs to be made in `DEF PROCinputKnowns()` after the L entry but before the N entry. If you refer to ***Loan Update 15b*** you'll see that the following sequence has been added at the right place:

```
1890 IF ( RateFlag% = TRUE ) THEN
1900 IF ( INT(L / PaymentUpper) + 1 > NumberLower )
     THEN NumberLower = INT(L / PaymentUpper) + 1
1910 IF ( INT(L / PaymentLower) - 1 < NumberUpper )
     THEN NumberUpper = INT(L / PaymentLower) - 1
1920 ENDIF
```

This uses two 'ratcheting' statements - one 'ratchets' `NumberLower` up - and one 'ratchets' `NumberUpper` down. The result is that the variable `NumberLower` ends up holding the higher of `INT(L / PaymentUpper) + 1` and the original value of `NumberLower` (our overall limit set at Line 1830 in ***Loan Update7b***) - whereas `NumberUpper` ends up with the lower of `INT(L / PaymentLower) - 1` and the original value of `NumberUpper`.

This is a very common construction where a program needs to find the highest (or lowest) value from a series of values - or needs to swap positions of items in a list if they are not currently in the order wanted e.g. in alphabetical sorting routines, where the Ascii values of the first letter of strings are compared to get them in the right order.

We need to use the keyword `INT` (e.g. in Line 1900) because we want N to be a whole number - and the +1 in this line ensures we fix the new limit to the integer above the (real number) answer to the calculation in the brackets. *As Line 1910 is dealing with the upper limit the -1 is not strictly necessary, but it keeps things symmetrical.*

Something very similar is then inserted, based on the now known value of (L ÷ N), after the N entry but before the P entry (at Line 2030) - and this time we need only be concerned with the lower limit for P. *(The unchanged upper limit may produce very high interest rates , but it will not stop the program functioning correctly.)*

A similar, but much shorter, process is now needed to eliminate the problem when N is the unknown factor.

Firstly in this case, if a value for L is chosen near the lower limit, the upper value of P may then need to be reduced so that a single payment a lot higher than the loan is not chosen. Secondly, if a high value is chosen for L, it is possible to choose a monthly premium which will not be high enough to cover even the lowest limit of our allowable interest rates.

So, we need a separate 'flag' (`NumberFlag%` at Line 1710) to register when N is the unknown, plus two appropriate statements after the input of the L value but before the input of the P value - at Lines 1940-1970. These are straightforward statements - except that we have used +2 instead of +1 in Line 1960 to overcome a possible rounding error problem.

With these changes the input limits will now adjust themselves automatically, within the preset overall limits, to values which will not stop the program working properly.

We are not claiming for one moment that these latest changes are perfect or cover the whole story. The main aims have been to ensure you end up with a program which won't easily break down and to stress that input validation is an important part of programming which is likely to crop up in every program requiring a user input.

Keyword LOCAL

The introduction of this keyword (not to be confused with `LOCAL DATA` or `LOCAL ERROR`) has been left until now because its importance can easily be overlooked if it is included in the initial introduction of `PROC/FNs`.

On the surface it appears to be simply a useful device under the heading of 'good housekeeping' - but its role is more fundamental than that. But, firstly, let's have a look at how and where it is used.

`LOCAL` is used with both `DEF PROCs` and `DEF FNs` in order to keep the value of a variable 'local' to that `PROC/FN`. A local variable means that any value assigned to it when the `DEF PROC/FN` is in active use will be confined to that `PROC/FN` **without affecting the value assigned to a variable of the same name somewhere else in the program**. This means is that you can use the same variable name inside a `DEF PROC/FN`

and somewhere else in the program as two largely-independent variables.

Some examples will explain things better. Look at the following DEF PROC from our Loan program:

```
1240 DEF PROCcentrePrint(String$)
1270 :
1280 ScreenWidth% = 80
1290 Tab% = (ScreenWidth%-LEN(String$)) DIV 2
1300 PRINT TAB(Tab%) String$
1310 :
1320 ENDPROC
```

The first thing to take on board is that the formal parameters of any PROC/FN are **automatically** made ‘local’ to their PROC/FN. Thus, String$ is automatically made local to the above PROC and the same variable name could be used ‘outside’ the proc somewhere.

What happens is that the Basic processor stores temporarily the ‘outside’ value of String$ (assuming there is one) as soon as the above proc is entered. The normal variable storage space is then used for the local String$ values inside the PROC - and the ‘outside’ value is restored to its normal place when the PROC is exited. The non-local (‘outside’) variable is called a global variable.

The keyword LOCAL gives you the option of doing exactly the same thing with other variables used in the PROC.

In the above DEF PROC, the two variables Screenwidth% and Tab% are also used - and are clearly only used in a local context here. To make them ‘local’ to this PROC, the instruction:

```
LOCAL Tab% , ScreenWidth%
```

is simply added at the very start of the DEF PROC.

With this addition, the DEF PROC can be used freely, without worrying if we’ve used the same variable names somewhere else in the program.

The order of the variables listed in a local statement is not important - and they are just separated by commas.

In our final update to the Loan program (***Loan Update 15c*** - see a little later) we include a revisit to some DEF PROC/FNs to add local statements.

However, there is more to local variables than this.

Importance of free-standing PROC/FNs

Suppose we were using, as well we might, the above `DEF PROCcentrePrint(String$)` in a program using several screen formatting `DEF/FNs`, each with a need to know the value of `ScreenWidth%`. In that case we would have probably have put:

```
ScreenWidth% = 80
```

in, say, `PROCinit` rather than Line 1280, so that we could use it in any other place i.e. we would want `ScreenWidth%` to be a global variable rather than a 'local' one. We might still want `Tab%` to be local and we could effect this with the statement `LOCAL Tab%`.

So the new routine would be:

```
DEF PROCcentrePrint(String?)
LOCAL Tab%
:
Tab% = (ScreenWidth%-LEN(String $)) DIV 2
PRINT TAB(Tab%) String$
:
ENDPROC
```

with `ScreenWidth% = 80` placed in, say, `PROCinit`.

Alternatively, we could have the best of both worlds by also making `ScreenWidth%` the second formal parameter of the above new `DEF PROC` - with no further changes. Thus:

```
DEF PROCcentrePrint( String? , ScreenWidth% )
LOCAL Tab%
:
Tab% = (ScreenWidth%-LEN(String$)) DIV 2
PRINT TAB(Tab%) String$
:
ENDPROC
```

We then use `PROCcentrePrint( AnyString$ , ScreenWidth% )` to call the `PROC` and the value of the global variable `ScreenWidth%` - declared in, say, `PROCinit` - will be passed to the (automatically) local variable `ScreenWidth%` for use inside the `PROC` i.e. one name, two variables - as explained above.

The main advantage of doing this is that `DEF PROCcentrePrint( String$ , ScreenWidth% )` can now be used in any program, without alteration. It is now 'free-standing' - totally

without any global variable inside it and yet you can still use global variables (of the same or different names) or direct values at will in the passed parameters when you call it.

Having a `PROC/FN` which is already available **and is known to work** is a tremendous advantage in program development. It reduces errors and debugging is made easier - and much time is saved.

Hence the practice of making `PROC/FNs` free of global variables is to be strongly recommended - by the use of formal parameters and the keyword `LOCAL` in combination.

There is a (very) small price to pay: the direct result of the above recommended practice is that the number of formal parameters per `DEF PROC/FN` inevitably increases - exactly as demonstrated above with `ScreenWidth%` - but this is not usually a practical problem.

You will not be surprised that most programmers tend to build up 'libraries' of useful `PROC/FNs` *- and BBC Basic caters for this with some very useful facilities which are covered in Chapter 22 and which are very important in Wimp programming.*

Having recommended the practice, you will detect that our Loan program does not always follow it! For instance, `PROCaxes()`, `PROCplots()` and their derivative `PROCs` are definitely not free of global variables.

The reason is that at the stage they were introduced, in this Beginners' book, it was more important to avoid adding more than one new topic at a time - and the concept of local variables was considered important enough to leave until now.

However, you should now be able to see that it would be better programming practice for variables such as `Xorigin%`, `Yorigin%`, `ZoomXorigin%` and several others, to be declared globally - in, say, `PROCinit` - and for a corresponding list of formal parameters to be added to the `DEFs` of `PROCaxes()`, `PROCplots()` and most of their derivative `PROCs` - all with the aim of making each `PROC/FN` free of global variables and thus 'transportable' to other programs.

Putting the whole program within a loop

At the moment our program ends after we have used it just once. It would be better if we could use it repeatedly and only end when we want to. This is easily achieved with a `REPEAT ... UNTIL` loop placed around the main structure - Lines 130-220.

We now also need to add a message to our opening screen menu to give the user the option to quit - and then ensure that the program returns to this menu each time it finishes with a calculation. The quit option will be exercised simply by pressing the letter “Q” on the keyboard, instead of L, N, P or R.

Referring to ***Loan Update 15c*** (see a little later), the extra menu message is added to `PROCsetUpMenuVariables` at Line 890 and is printed by `PROCmenu` at Line 1170. The `REPEAT ... UNTIL` loop is entered just before `PROCmenu` and exited after `PROCdisplayResults`.

As our exit criterion is going to be the letter “Q” we need to add this letter to the list of valid key presses in `FNmenuChoice` (at Line 1550). If the letter “Q” is chosen we then need to bypass the normal program actions - and this is easily achieved with an `IF ... THEN ... ENDIF` construction at Lines 170 and 230.

Finally, we also need to clear the screen after each use of the program and this is, in fact, best done at the very start of the `REPEAT ... UNTIL` loop in `PROCmenu` at Line 980.

You may wonder why we left this overall `REPEAT ... UNTIL` loop until the very end of the project - it would certainly have been a useful feature for the programmer earlier. There are several reasons: firstly, it is important to visualize the main program flow clearly before making it repeat. Secondly, until all the ‘bells and whistles’ are incorporated, it is not always possible to see the best places to insert the final `REPEAT ... UNTIL` keywords. Thirdly, when developing a program it is not unknown to inadvertently introduce errors - and meeting these early on with a major loop in place can lead to hair-tearing endless loops.

It is therefore strongly recommended that you continue to approach your own programs in the same way - get it working well before worrying about how to repeat it.

Final update

The listing changes to incorporate local and to put the whole program within a loop are both combined into the final update below.

Loan Update 15c

```
10 REM> Loan15c
20 REM** Upgraded from Loan15b **
120 REPEAT
170 IF ( Unknown$ <> "Q" ) THEN
230 ENDIF
```

```
240 :
250 UNTIL ( Unknown$ = "Q" )
890 Quit$ = "...... or press 'Q' to quit program."
970 :
980 CLS
1170 PRINT : PRINT TAB(Offset1% + Offset2%) Quit$
1250 :
1260 LOCAL Tab% , ScreenWidth%
1500 :
1510 LOCAL Position% , KeyPress$
1550 Position% = INSTR("LNPRQ",KeyPress$)
1640 :
1650 LOCAL RateFlag% , NumberFlag%
2350 :
2360 LOCAL Item , Loop% , Limit$
2870 :
2880 LOCAL TermValue , CuSum, Term%
3030 :
3040 LOCAL InverseFactor , GraphHeight% , Xorig%,
     Yorig% , Scale , Gap% , CharWidth% , TermValue ,
     CuSum , RemainingLoan , NumOfPayments%
6120 :
6130 LOCAL StartTime%
6370 :
6380 LOCAL String$, Left$, Right$, Rounding, Negative%,
     Result$
```

Final words on Loan project

Finally, we have left `DEF PROCdisplayResults` untouched - it still only contains one action line, to pause the program. If we were to delete this `PROC` we would only need to introduce the pause in some other way and at the same place in the sequence - otherwise our new overall `REPEAT ... UNTIL` loop would take us back to the opening menu before we have chance to read the actual results on screen.

So we might as well leave things as they are - and the name of the `PROC` is not entirely inappropriate in the circumstances.

Figure 15.2 shows the just-changed opening menu of the final version.

```
                Loan Calculations

There are four factors:-

              Loan Amount             (L)
              No. of Equal Payments   (N)
              Amount of Each Payment  (P)
              Interest Rate           (R)

You need to give values for any 3 of these to find the 4th.

Please choose the unknown one (L/N/P/R)

              ...... or press 'Q' to quit program.
```

Figure15.2

Opening menu of final Loan program.

The Loan program will not be updated further in subsequent chapters, although we will make reference to it from time to time - so ***Loan15c*** is the 'final version'.

Remember, it was developed as a vehicle to introduce you, the beginner, to various 'foundation' aspects of BBC Basic. It is not a model of perfection and there are certainly different ways to achieve the same ends in all the `PROC/FNs` used.

Also, it is never a 'final version' of course. Most programmers have programs in frequent use which first saw the light of day many years ago and have been added to and modified every few months since. That's part of the attraction of programming - *and also part of the bug-bear of having to upgrade commercial software regularly*.

To continue this tradition (!), here are some areas for you to work on yourself to develop ***Loan15c*** - and your confidence - further:

Convert all the `PROC/FNs` to 'free-standing' - except those early ones such as `PROCinit`, `PROCcolourDefs` etc. which are essentially unique to this program.

Make the required accuracy of the interest rate answer a user-determined item.

Make the overall input limits user-determined items.

Investigate the use of different line types (dotted, full, etc) instead of colours to distinguish the 'getting near' graph lines (Wait until we have a further look at `PLOT`).

Improve the 'bracketing' and 'converging' algorithms - if you have a mathematical bent.

Change `FNnumberToString` to allow a wider range of decimal places to be provided, by adding an extra formal parameter specifying the number of places. This might then form a useful general `FN` for other programs.

These should keep you going for a while!

The only sermon to preach is to stress that you should only attempt to change one item at a time and make your attempt on a copy of your latest version. Once you are happy with the changed version, save it and, again, use a copy of that to start work on the next item.

The end of the Loan project doesn't mean the end of the book. We still have several topics to cover.

16. VDU commands

Detailed introduction of VDU commands, concentrating on VDU control codes Keyboard <Ctrl> equivalents.

The acronym “VDU” stands for Visual Display Unit - effectively your computer screen - and part of the job of the computer operating system is to compose and send to the screen the right sort of signals to display text and graphics. This operating system function is called the ‘VDU driver’.

BBC Basic has a very extensive set of instructions which allow the programmer easy access to the VDU driver to control many visual effects.

The deceptively simple keyword is `VDU`, which was touched on more briefly in Chapters 7 and 14. The general form of a `VDU` instruction is:

```
VDU xx
```

where `xx` can be any integer number from 0-255 (&00-&FF hex). For example, if you issued the Basic instruction:

```
VDU 8
```

the number 8 would be sent to the VDU driver and it would interpret it and take the appropriate action. (In this case, it would move the text cursor back one character i.e. one step to the left in the normal English set-up.)

More than one VDU command can be sent in the same statement merely by separating the numbers with a comma. Thus:

```
VDU 8 , 8
```

will effectively action `VDU 8` twice i.e move the cursor back two character steps.

With some of the values of `xx` you need to add extra integer numbers after it to complete the instruction. These extra numbers are additional data bytes which are parameters passed to the VDU driver along with the particular `VDU xx` command.

Mostly, any extra parameter numbers are in the range 0-255, but sometimes numbers outside this range are needed i.e. when specifying OS screen units, which can be 2-byte integer numbers - *see Chapter 12*. In these cases, the number in the VDU instruction must be followed by a semi-colon in all cases (even when it is the final number of a sequence and even when the number itself is in the range 0-255). The semi-colon tells the computer that two bytes, rather than one, are to be used to store the number. *(If the number only uses one byte the second byte is made zero automatically.)*

We have already met examples of extra parameter bytes (`VDU 24` and `VDU 28`) and the use of the semi-colon (`VDU 24`) in Chapter 14 and in the demonstration program ***`Prog 14a`***, where the extra bytes defined the size and position of the graphics and text viewports.

Thus, `VDU 28` is interpreted by the VDU driver as “redefine the text viewport” - and it takes the following four bytes (in a known order) as numbers representing the position of the sides of the new viewport. *(You can still send more than one VDU command (with parameters) in the same statement, as shown above - but you do have to be careful to ensure that the required number of parameter bytes for one command is complete before the following command is started.)*

Clearly, the potential number of effects that could be covered in this way is huge - because there is no reason why each of the 256 values of `xx` could not be expanded by designating that, say, a second number from 0-255 should follow - and then another etc. - before any parameter data bytes.

Fortunately for our learning, it is not as bad as that - because the values of `xx` used are actually ASCII values *(see Chapter 6)* and the general interpretation of `VDU xx` is as follows:

VDU code	***Effect***
0-31	**VDU 'control codes' (see later)**
32-126	**print ASCII character**
127	**'delete' character to left of cursor ("backspace and delete")**
128-159	**display ASCII/defined character or teletext control codes**
160-255	**display ASCII/defined character**

Thus, the Basic instruction:

```
VDU 65
```

will simply be interpreted by the VDU driver as "Print ASCII Code 65" and "A" will be printed on the screen. (Try it in Basic Immediate Mode in a Task Window.) And a similar result will happen with any number from 32-255 (except 127) - *but bear in mind that numbers above 127 are not strictly ASCII and are defined by the particular computer set-up (and perhaps the user). So the character on the screen may differ from time to time with these higher numbers - and quite often some of them might be blank (and remember that 32 is a blank space).*

Therefore, for these single number instructions, the Basic statement:

```
VDU xx
```

acts very much like:

```
PRINT CHR$(xx)
```

and, in fact, it is not very often that you will come across `VDU` used merely to print characters - `PRINT CHR$()` is generally preferred because its intent is clearer in listings.

So, we only need to look in more detail at codes 0-31 - the non-printing, so-called 'control codes', which we first met in Chapter 6 - and it will be no surprise to learn that these numbers are also called the VDU Control Codes.

Most of these 32 VDU codes have simple effects which need no more explanation than is easily carried in a reference table and such a table is at Appendix 9.

Thus, `VDU 13` means "Move text cursor to start of current line" - and a comparison between Appendix 9 and the ASCII chart at Appendix 4 will show that this is the same as PRINT CHR$(13) i.e. the letters "CR" meaning "Carriage Return" (from typewriter days) appear in the chart at

ASCII code 13. (Again, try these equivalents for yourself in Basic Immediate Mode in a Task Window). We also covered VDU codes 4, 5, 24, 28 and 30 in Chapter 14.

Therefore, in this chapter we need only comment in more detail on some of the less straightforward of the remaining VDU codes:

`VDU 1` Sends the following byte to printer only- and therefore needs a following byte to accompany it. Note that it works only for one following byte. Thus, VDU 1,27 will send the number 27 to the printer without also sending it to the screen. If you want to send two or more bytes to the printer then each has to be preceded by a `VDU 1`. This therefore provides a means of controlling the printer from a Basic program (or from Immediate Mode) without interfering with the screen display.

It is very commonly used to send printer ‘escape code’ sequences to a printer to change the typeface etc. when printing directly from Basic - and is much used in ‘printer driver’ software (albeit probably in its machine code equivalent).

`VDU 2` and `VDU 3` These effectively turn the printer ‘on’ and ‘off respectively, from within a Basic program. Once `VDU 2` is issued, any use of `PRINT` or keyboard input goes to the printer as well as the screen - and `VDU 3` cancels this effect.

VDU 2 is therefore the main (and often the only) thing you need to get your printer to produce a hard copy of what is on the screen. If you are using text output, it is certainly worth a few experiments to see what happens. Graphics are more difficult ? you need a ‘screen dump’ program.

However, VDU 2 needs to be used with considerable caution, because it is very easy to send unwittingly some non-printing codes which ‘hang-up’ the computer - or produce reams of paper with gibberish and/or very little on them - with a hard reset being the only way out.

If you are going to try to print directly from Basic (i.e. without going via a printer driver in the modern, approved manner) always use VDU 2 and VDU 3 as a pair (like brackets) and always use the VDU 3 as soon as you can after the wanted material has been sent. *(Before going very far you will also need to delve into your printer manual in some detail to get to grips with*

'Escape codes' etc. - certainly for graphic printing. If you like to understand the 'how and why' of things then direct printing it is an interesting area to probe - but it is not necessary and certainly beyond this book.)

`VDU 6` and `VDU 21` These commands are also a pair. The second disables all output to the screen - and the former restores things back to normal. (Remember VDU 6 as a possible life-saver if you are working in Basic and the keyboard suddenly seems to be having no effect.) One common application of this is to prevent output specific to a printer from appearing on the screen. Note that it is only the output to the screen which is stopped, everything else carries on even though you may not be able to detect it.

`VDU 7` Produces a 'beep' from the computer. It is surprising how often this is useful. It is frequently used to give the user audible 'feedback' that an input action is wrong e.g. a 'beep' occurs when a wrong letter is chosen.

For a programmer, adding it to a `DEF PROC` during the development of a program will tell you when the `DEF PROC` has been called - which is often a great help to check that conditional statements are working as intended.

Needless to say, the actual sound of the 'beep' can be altered - but that is beyond the scope of this book.

`VDU 14` and `VDU 15` Another pairing. These engage/cancel 'Paged Mode' respectively. When engaged, a scrolling non-Wimp screen will halt after one screenful of text has been displayed. Pressing `<shift>` will cause the next 'page' to be shown and so on.

If the program also has an option to send the same output directly to a printer you need to ensure that paged mode is disengaged during printing - otherwise you will find that you have to press <shift> at the end of each 'page'.

`VDU 17`, `VDU 18`, `VDU 19` and `VDU 20` These concern colour control and are included in Chapter 21.

`VDU 23` This code has been left until last because it is used as a multi-purpose command to provide an enormous range of effects/controls. It takes nine parameters and even when some of these are not used they need to be entered as zero (and an easy means of doing this is provided - see below).

The second number (the first parameter) is the most important because it chooses the particular effect/control. You will need to refer to Acorn's BBC Basic Reference Manual to find the full detail of these. However, one such command is described below to demonstrate the format:

`VDU 23,1, n` This controls the appearance of the text cursor on the screen, depending on the value of n, as follows:

n	**effect**
0	Stops the cursor appearing. (Same effect as the keyword off)
1	Makes the cursor appear. (Same effect as the keyword on)
2	Makes the cursor non-flashing
3	Makes the cursor flash

Note that the way this must be entered as an instruction (for example, to make the cursor non-flashing) is:

```
VDU 23,1,2|
```

The vertical line immediately after the number 2 is ASCII character 124 (which appears on some keyboards as ¦ but on the screen as `|`). It is interpreted in a `VDU` line as ,0,0,0,0,0,0,0,0,0,

i.e. nine bytes of value zero plus an additional comma.

So, 2 | is interpreted as 2,0,0,0,0,0,0,0,0,0, As VDU 0 does nothing *(see Appendix 9)*, this device always completes the VDU statement by adding more than enough zero bytes as parameters - any extra zero bytes having no effect. *(The character `|` has a different meaning when accessing memory locations directly - so be careful! See Chapter 20)*

Note that if you want to put another `VDU` command in the same statement after the use of the character `|`, the format needs to be:

```
VDU 23,1,1|23,1,2
```

This example, as can be deduced from the list just above, turns the cursor on and then makes it non-flashing. There is no comma between the two parts - because of the extra comma

included in the interpretation of `|`. This can be a little awkward to read in a listing and probably explains why separate `VDU` statements are more usually seen in such cases.

Keyboard 'Control' equivalents

In Appendix 9 you will see a column headed `"<ctrl> +"` added to the list of VDU commands. If you are working in Basic from the Command Line, this column can be used to carry out most VDU commands directly from the keyboard - which can useful when developing a program. However, much of the original usefulness of this facility has been superseded by the fact that programming is now usually carried out in a text editor (e.g. !Edit) in the Wimp environment, rather than from the Command Line.

Nonetheless, it is as well to know that the facility exists. So, for instance, enter Basic from the Command Line by pressing `<F12>` then typing "basic". Now, hold down either one of the `<ctrl>` keys whilst pressing the letter "L". You will find that the screen clears - exactly like a `VDU 12` or a `CLS` statement. (Type `QUIT` in capitals to exit Basic. Then press the `<return>` key to take you back to where you started in Desktop Mode.)

17. Data storage/retrieval (READ, DATA and RESTORE)

Permanent and temporary storage - Use of READ. DATA and RESTORE - Data pointer.

In many programs the amount of data to be processed warrants special facilities to store it (temporarily and permanently) and to retrieve it. In this context, “permanent” means the data is not lost when the computer is closed down - whereas “temporary” storage means the data (in its temporary home) is lost when the program is exited (whether or not this latter also coincides with computer close-down).

You may be puzzled why we need temporary storage, but in the programming context it invariably means transferring the data from permanent storage to temporary storage to make it easier and/or quicker to access and/or manipulate the data - and, as you will see, all means of temporary storage are either variables or direct memory locations.

BBC Basic provides several means of data storage/retrieval and we are going to look at the subject in four linked chapters under the sub-headings:

Keywords READ, DATA and RESTORE

Arrays and the keyword DIM

Data files

Direct memory access and indirection operators

Keywords READ, DATA and RESTORE

The keyword `DATA` provides a means of permanently storing small amounts of data - and `READ` and `RESTORE` enable us to access it. The use of the word “small” is somewhat subjective: it is meant to mean the amount of data that you are prepared to type in, once, yourself. With this method, you cannot generate and load it automatically - nor type it in whilst the program is running.

The data is actually typed into a normal Basic program line and hence is stored permanently when you save the program. It is retrieved, **for processing only within the same program**, by using the keywords `READ` and `RESTORE`.

As you might expect, data items can be any of the three types: real number, integer number or string - and you can mix these at will, provided you take simple precautions when they are retrieved. Let’s look at an example in a simple program:

Program Prog17a

```
10 REM> Prog17a
20 REM** DATA/READ Demo **
30 :
40 Title$ = STRING$( 15 , " " ) : REM** To initialize
     string at max length needed. **
50 :
60 REPEAT
70     :
80     READ Volume% , Number% , Page% , Title$ :REM**
     Note order of variable types matches order of
     Data starting at Line 200. **
90     :
100     IF Volume% <> -1 THEN
110         PRINT TAB(5) "Vol." ; Volume* ?
120         PRINT TAB(11) "No." ; Number* ;
130         PRINT TAB(17) "Page " ; Page* ;
140         PRINT TAB(27) Title?
150     ENDIF
160     :
170 UNTIL Volume% = -1
180 :
190 END
200 :
```

```
210 DATA 8 , 12 , 56 , Introduction
220 DATA 9 , 1 , 76 , Variables , 9 , 2 , 43 , Keywords
     , 9 , 3 , 48 , PROC/FNs , 9 , 4 , 53 , "Menu
     Selection" , 9 , 5 , 59 , "Making an Input" , -1
     , -1 , -1 , -1
```

There is a screen shot of the output of this program in Chapter 18 - Figure 18.1. The output from **Prog17a***,* **Prog18a** *and* **Prog18b** *are deliberately identical - for easy comparison of the listings.*

Look at the last two lines first: they are typical DATA statements. After the keyword DATA the data items are typed in one by one, each separated by a comma - but no comma at the end of the line.

Strings only need to be put within quotes if the string contains spaces or commas.

(The actual data used in the example comes from a series of articles, by the author, in 'Archive' magazine - which prompted the idea of this book.)

Each DATA line is still under the constraint of a normal Basic program line i.e. it cannot be longer than about three 80-character lines - but data can be carried over as many DATA lines as you like and they can be as short as you like also. It often improves readability/understanding to break up longer data lists into logical groups by using short DATA lines.

DATA lines do not need to be contiguous - but they must be in the right sequence. When you do want/need to continue the data items beyond the end of a Basic line, just start the next line with DATA again, as in Line 220 above.

Now look at Line 80: this is the retrieval statement, using the keyword READ. It 'reads' the data items, in turn, into the variables Volume%, Number%, Page% and Title$.

In this case we have chosen to read in the values of these variables as a set of four each time, but we could have made four separate READ statements if we had wished.

Whatever we do, it is vital that the DATA items read in are compatible with the READ variables i.e. in our example program the DATA items must be of the types and order:

integer , integer , integer , string

and you can see that the DATA lines conform to this pattern.

Try putting things in the wrong order and/or with wrong types - and the error messages will be similar to those you will be used to when an attempt is made to assign a wrong type of value to a variable - as in Chapter 3.

Data pointer

data lines employ a hidden pointer. When the program is run this data pointer is initially and automatically positioned just in front of the first item of the first `DATA` line in the program.

When this first data item is read by the first `READ` operation, the pointer moves to the comma just after the first item. The pointer then waits there until the next `READ` operation (whenever it may occur in the program) and so on.

If the pointer gets to the end of a `DATA` line, it jumps to the start of the first data item in the next `DATA` line, if there is one - wherever it is in the listing - and waits there instead.

In the above program the `READ` line is within a loop. Therefore, the first time round the loop the four items in Line 200 are read one by one into the four variables and the pointer ends up at the start of the first item of Line 210. The next time round the reading carries on from there. As Line 210 contains five repeats of the same order pattern, plus the final, deliberately-artificial, set of -1,-1,-1,-1, the loop will keep repeating until it reads in the final four items - which allows the loop exit condition to be met. The data pointer will then be after the final -1.

Having chosen to read in four items at a time in our example program, we are obliged to ensure that four items - still of the right type - are also available for reading this last time around the loop, even though our exit condition looks only at one of the variables. If you omit any of these last four items you will get the “Out of data” error message. (Note also here that -1 is OK for both a numeric and a string value: it contains no spaces or commas, so will be read as “-1” in the final `READ` action.)

Other points

data lines can be placed anywhere in a program, but it is usual to put them at the end of the `PROC/FN` which reads them, or at the end of the listing. If they are encountered by the program other than by a `READ` action, `DATA` is treated as rem and the line is ignored.

The essential feature of `DATA` and `READ` is that the data is transferred from the permanent home (in the `DATA` lines) to temporary storage (variables) for processing. The method is well suited to handling fairly small amounts of unchanging data. For instance, you will often find it

used in Wimp programs to hold the text contents of the window which appears when you move the pointer over the ‘Info’ item in an iconbar menu.

The method is also sometimes used to hold longer, complete, machine-code routines in program listings intended to be typed e.g. in a magazine.The data items then being read one by one by the program into a known memory location, which is then called by the program when needed.

Keyword RESTORE

The keyword `RESTORE` (not to be confused with `RESTORE DATA` or `RESTORE ERROR`) completes the trio: this sets or resets the `DATA` pointer position and it can be used in three ways.

When used on its own, it sets/resets the pointer to the start of the first data item on the first `DATA` line of the program (i.e. the `DATA` line with the lowest line number, as at program start-up).

When used with a line number e.g. `RESTORE 210`, it sets/resets the `DATA` pointer to the first data item on the specified line - or the first `DATA` line after that if the numbered line does not have a `DATA` statement. (The line number reference must exist though, even if it is empty.)

Finally, you can also use the form `RESTORE + n`, which means set/reset the `DATA` pointer to the start of the line n + 1 after the line in which it appears. So, `RESTORE +0` means set/reset the `DATA` pointer to the very next line etc. - and this form, of course, cannot suffer from missing line numbers (assuming you have not forgotten to add the `DATA` lines!). This form is particularly useful in ‘Libraries’, which we have yet to cover.

If you are using more than one set of `DATA` statements in a program (whether or not they are listed consecutively) - or more than one set of `READ` operations on the same data set - it pays to use `RESTORE` in one of its forms just before each set of `READ` operations, even the first. This habit will ensure you always have the pointer in the right place before starting to read data (a similar policy to always putting a `MOVE` before a `VDU5` text operation). Incidentally, the “Out of data” error message is often a sign that you have started to read at the wrong place.

(There is a further nicety, using the keywords `RESTORE DATA` *and* `LOCAL DATA`*, but we do not need to cover these.)*

> *We use the output created by* ***Program Prog17a*** *in the following chapter, to help compare the different data storage methods.*

18. Data storage/retrieval (cont'd) (Arrays and the keyword DIM)

Using DIM to create arrays - Single and multi-dimensions - Subscripts - Comparison with variables - Demonstration using same data as previous chapter for comparison - Reason for declaring arrays early in program - Scope of operations on whole arrays.

In our particular example in the previous chapter, the way we wanted to process the data meant that we had no need to access any of the data items at will. We were content to handle a set of four items at a time and serially. Once we had dealt with them, we effectively discarded them and used the same four variables to process the next four etc. Very efficient of memory space - but somewhat restricting if we had needed to use one of the discarded items later.

When we need ready and unrestricted access to all data items there is not much choice but to store them (temporarily or permanently) in a more accessible way.

Again in the same example, a temporary way would have been to create 24 different variables and read all the data items into them at the start. In effect, this is exactly what Basic provides with its 'arrays', but in a much more flexible way.

Keyword DIM

The keyword dim (short for "dimension", a verb in this case) simply creates a cohesive set of variables. Examples of the format are as follows:

```
DIM RealArray(2)
DIM IntegerArray%(8)
DIM StringArray$(6)
```

These three Basic statements will 'dimension' three arrays: the first containing three real numeric variables (or 'elements') called `RealArray(0)`, `RealArray(1)` and `RealArray(2)`: the second

containing nine integer numeric variables called from `IntegerArray%(0)` to `IntegerArray%(8)`: and the third containing seven string variables called from `StringArray$(0)` to `StringArray$(6)`.

The same thing could have been achieved with:

```
DIM RealArray(2), IntegerArray%(8), StringArray$(6)
```

and, as usual, any numerical expression may be used inside the brackets instead of an actual number.

There must not be a space between the array name and the opening bracket - in either the `DIM` statement or when referring to the array elements subsequently.

Once dimensioned, an array element behaves exactly like any other variable - except for the significant advantage that the number in the brackets (called the "subscript") can be any numerical parameter i.e. you can refer to an element's subscript by any expression which equates to a number, for instance:

```
RealArray( INT( Number * 6 ) )
```

which makes reading/writing values from/to arrays very flexible.

If you refer to a subscript outside the range that you created in the `DIM` statement, you will get the error message `"Subscript out of range"` - and remember that if you use a real number to refer to a subscript, only the integer part of that number will be used i.e. `RealArray(2.5)` will be converted to `RealArray(2)` before action.

Our examples above created "single dimension arrays", but you can also use dim with two (or more) numerical parameters e.g.

```
DIM IntegerArray%(2 , 3)
```

will create a two-dimensional integer numeric array - in this case creating 12 elements/variables with names from `IntegerArray%(0 , 0)` to `IntegerArray% (2 , 3)` and two subscripts are now needed when referring to them.

The only limit to the size and dimensions of arrays is your computer memory - which neatly brings us to the point that arrays eat memory fast, particularly when they are multi-dimensional and also when using string arrays.

When arrays are first dimensioned by the `DIM` statement, all numeric array elements are assigned with the value 0 - and all string array elements with null strings. This is very convenient, but it therefore

becomes very important to note the points made in Appendix 7 and re-declare the size of string array elements to a sensible 'maximum' size.

It is worth reinforcing the point that array elements act as variables. Try the following simple sequence in a Task Window (as a program):

```
10 DIM IntegerArray%(5)
20 value% = 3
30 IntegerArray%( value% ) = 2 :REM** Variable used as
      array subscript. **
40 IntegerArray%( IntegerArray%(3} ) = 17 :REM** One
      array reference used as subscript in another
      array. **
50 FOR N% = 0 TO 5
60 PRINT IntegerArray%( N% ) :REM** Prints all
      elements, including those not altered after
      creation. **
70 NEXT
```

This sequence will print the result 0 , 0 , 17 , 2 , 0 , 0 which confirms that individual array element subscripts can be used just like variable references and also that all numeric array elements are assigned with zero when first created.

Programs Prog18a and ***Prog18b*** listed below are two further very short demonstration programs - both producing the same output as ***Program Prog17a*** but using arrays. ***Program Prog18a*** uses single dimension arrays and ***Program Prog18b*** uses a 2-dimension array. They should be easy to follow without further explanation - and a screenshot of the output is in Figure 18.1.

Program Prog18a

```
10 REM> Prog18a
20 REM** Array/DIM Demo No.1 **
30 :
40 DIM Integer%; ( 18 ) , String$( 6 ) :REM** Two
      single dimension arrays declared. **
50 :
60 FOR N% = 1 TO 6
70     :
80     String$( N% ) = STRING$( 15 , " " )
90     :
100     REM** Next line reads data into arrays in
      ascending order of subscript, from 1 upwards. **
```

```
110     READ Integer%( (3*N%) - 2 ) , Integer%( (3*N%)
     - 1 ) , Integer%( 3*N% ) , String$( N% )
120 NEXT
130 :
140 REM** Next routine prints data from arrays in same
     order. **
150 FOR N% = 1 TO 6
160     PRINT TAB(5) "Vol." ; Integer%( (3*N%) - 2 ) ;
170     PRINT TAB(11) "No." ; Integer%( (3*N%) - 1 ) ;
180     PRINT TAB(17) "Page " ;Integer%( 3*N% ) ;
190     PRINT TAB(27) String$( N% )
200 NEXT
210 :
220 END
230 :
240 DATA 8 , 12 , 56 , Introduction
250 DATA 9 , 1 , 76 , Variables , 9 , 2 , 43 , Keywords
     , 9 , 3 , 48 , PROC/FNs , 9 , 4 , 53 , "Menu
     Selection" , 9 , 5 , 59 , "Making an Input"
```

Program Prog 18b

```
10 REM> Prog18b
20 REM** Array/DIM Demo No.2 **
30 :
40 DIM Integer%( 3,6), String$( 6 ) :REM** This time
     the numeric array is two dimensional, with the
     dimensions specifically following the column/ row
     format of the intended printed output. **
50 :
60 FOR row% = 1 TO 6
70     :
80     REM** Next routine reads data into numeric array
     in column/row subscript order. **
90    FOR col% = 1 TO 3
100        READ Integer%( col% , row% )
110   NEXT
120   :
130   String$( row% ) = STRING$( 15 , " " )
140   READ String$( row% )
```

```
150   :
160 NEXT
170 :
180 REM** Next routine prints out array data row by
     row, using fixed column subscript values. **
190 FOR row% = 1 TO 6
200     PRINT TAB(5) "Vol." ; Integer%( 1 , row% ) ;
210     PRINT TAB(11) "No." ; Integer%( 2 , row% ) ;
220     PRINT TAB(17) "Page " ; Integer%( 3 , row% ) ;
230     PRINT TAB(27) String$( row% )
240 NEXT
250 :
260 END
270 :
280 DATA 8 , 12 , 56 , Introduction
290 DATA 9 , 1 , 76 , Variables , 9 , 2 , 43 , Keywords
     , 9 , 3 , 48 , PROC/FNs , 9 , 4 , 53 , "Menu
     Selection" , 9 , 5 , 59 , "Making an Input"
```

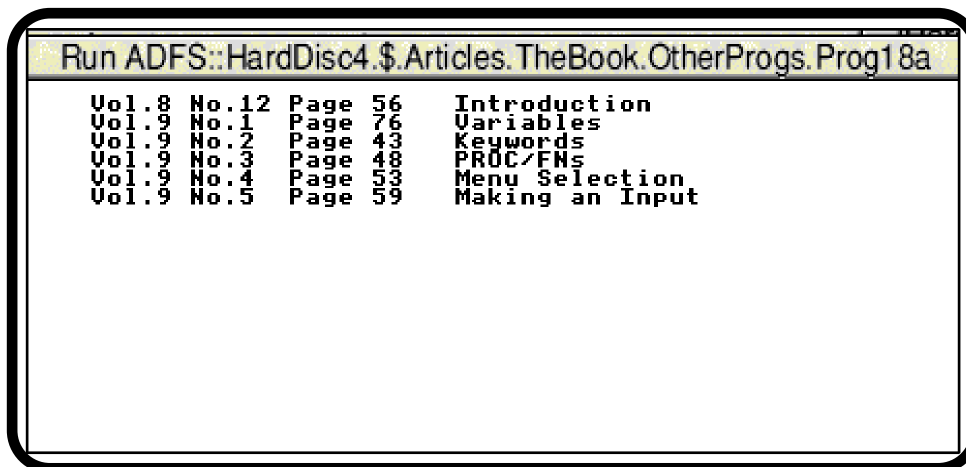


Figure 18.1

Output screen from programs 'Prog17a' , 'Prog18a' and 'Prog18b'.

Some practical points

An array name can only be dimensioned (created) once during a program run and this includes not being able to change its size. The error message “Arrays cannot be re-dimensioned” occurs in both cases. It is partially for this reason that it is good practice to try to confine all the program `DIM` statements to a `PROCinit`, called once at the start of a program run.

The more important part of the reason is that the program will then be able to tell you immediately if there is not enough memory for the array(s) - “No room for this `DIM`” - rather than give you this (fatal) error

later in the program with then, perhaps, unsaved data at risk.

Even worse, sometimes there is just enough room in a running program to dimension a late-declared array, but the action quietly overwrites the contents of a memory location by then being used for some other purpose. In this case the error message, if any, is likely to appear totally unrelated to the dim that caused it. So, make life easy for yourself!

Don't get hung up on the point that `DIM ARRAY(6)` creates 7 elements. Many programmers, unless space is particularly tight, tend consciously to ignore `array(0)` and use `DIM ARRAY(6)` if they intend to work with 6 elements. This is probably because it is an area where it definitely doesn't come natural to start counting from zero - whereas there are, at least, some logical reasons to start 'bit counting' and its derivatives from zero. Use whichever you feel easiest with. *Oddly enough, you will come across many occasions when array (0) can be usefully used for something related to the other elements e.g. to hold the sum of the other elements.*

There is a further way in which `DIM` can be used and we will come to that in Chapter 20.

Operations on whole arrays

Although we do not need to go into the detail, BBC Basic also provides a number of instructions which operate on an array as a whole. Therefore, there is a need to be able to refer to an array name unambiguously. The form of this reference is simply:

```
ArrayName()
```

with no spaces. For example, `IntegerArray%()` in the above case.

Using such an array reference, it is possible to:

- find the number of dimensions of an array;
- find the size of any dimension;
- assign a single value to all array elements;
- assign different values to individual array elements;
- add, subtract, multiply, divide and increment all elements in a numeric array with a common factor;
- add, subtract, multiply, divide the contents of all corresponding elements of two arrays of the same size;
- carry out proper matrix multiplication on suitable pairs of arrays;

use the reference as a parameter in `PROC/FNs`;

sum the contents of all the elements of a numeric array;

concatenate the contents of all the elements of a string array;

sum the lengths of strings in all the elements of a string array;

find the "modulus" (square root of the sum of the squares) of the contents of all the elements of a numeric array.

We do not need to explore them further at the moment, but some supplementary text is in Appendix 10.

19. Data storage/retrieval (cont'd) (Data files)

Basic data files in detail - OPENOUT, OPENUP and OPENIN explained - 'Lost' files - Keywords CLOSE#, PRINT# and INPUT# - Data file pointer - Common error messages - Trapping open files in error trap - Pointer manipulation - Data file structure - Keywords PTR#, EXT# and EOF# - File dumps - Format of stored data items demonstrated - Basic V/VI differences - Data type identifiers - keeping track of pointer.

So far, the only way we have seen to store data permanently is via `DATA` statements, which are not really designed for large amounts of data. By far and away the most common way of permanent storage of data is on disc and for this Basic uses 'data files'.

By this time, you will be well-used to files - at least to the extent that you know that they are each represented by a separate icon, with a separate name, inside a directory window on the desktop screen. You also know that these icons represent separately lumps of memory space on, for instance, the hard-disc or floppy-disc you happen to be looking at.

You will also be familiar with several different file types, each type having its own picture for the file icon. For example, the Basic programs you have saved are each in a Basic file with the same familiar white-on-blue icon - and Text files with their black-on-white-plus-pencil icon.

It is then a very small step to accept that BBC Basic provides a set of keywords which enable us to create and manipulate files of a type called "Data" - which has its own distinctive icon (see Figure 19.1).

Figure 19.1

Data file icon

Data files can hold numeric data and string data and, of course, any mixture of these. *(They can also hold text - i.e. strings in a format compatible with text editors - but we will not be covering this.)*

Before we can put data into ("write to") a file or retrieve data from ("read") a file, we need to "open" it - and when we have finished, we need to "close" the file. In Basic, a file is opened by asking the computer to allocate a unique communication channel to the file. We then carry out all operations by addressing our instructions to that channel, including the final instruction to sever the link when we close the file.

So, let's have a look at some of the main data filing keywords.

Keywords OPENOUT, OPENUP and OPENIN

To open a data file we use one of these three keywords. They are all functions and are all used in the same way, but for different circumstances. The format used, for all three keywords, is:

```
Channel% = OPENOUT ( "Filename" )
```

and the channel number (an integer) is returned to `Channel%`, which is a normal numeric variable of your choice and acts as the channel designator. `"Filename"` can be any string expression which evaluates to a valid filename *(see User Guide, under "The Desktop")*. The brackets are best always used, because they are optional in some cases, but not others!

`OPENOUT` is used for creating and simultaneously opening a **new** file. Once opened by this function, both read and write operations can take place.

If you use `OPENOUT` with a file name that already exists i.e. not a new file, **then the existing file will be lost and a new,**

empty, file created in its place. So be careful of this one - there is no warning!

If you have a file open as a result of this keyword, you cannot open the file again until it has first been closed. Trying to do so generates an error message saying that the file is already open.

`OPENUP` is used to open an existing file for either (or both) read and write purposes. Again, if you have a file open as a result of this keyword, you cannot open the file again until it has first been closed. Trying to do so generates the same error message as above.

If you use this keyword with a filename which does not exist (or, more often, which the filing system cannot find) then the channel number returned (to `Channel%`, in the above) is 0, **without an error message**. However you will get an error message as soon as you try to read/write to the file - so you will probably need to 'trap' the return of zero for a channel number.

`OPENIN` is used to open an existing file for read purposes only. As you cannot write to the file, there is no danger in having more than one read-only channel open at the same time and this keyword can be used to do this. Again, 0 is returned if the file doesn't exist or can't be found, so the same comments as above apply.

Lost files

As indicated, in non-Wimp programs the most common reason for the return of 0 to a channel number is that an existing data file has not been found - because the Basic processor has not been told which directory to look in. The same thing often occurs in reverse when you use `OPENOUT`. The data file is duly opened, but you can't find the directory where it has been placed. This is exactly the same problem that we mentioned in Chapter 1 when saving a new Basic program - and, as then, Appendix 1 should help you locate the file.

These problems can be avoided by setting the directory in a `!Run` file linked to your application, but that is beyond the scope of this book. *It is actually covered, very obscurely, in the User Guide, under "Command scripts".*

A less flexible, but acceptable solution for 'the privacy of your own home', is to set the directory at the start of your program by using the star command:

```
*DIR <full directory name>
```

For example:

```
*DIR :HardDisc4.$.BasicProgs
```

(N.B. The named directories must already exist)

Most ‘star’ commands *(see User Guide)* can be used in Basic programs and this one is worth noting. A star command is best used on a separate program line.

This will ensure that new files opened by the program will be put where you want them and existing files (if held in the same directory!) will be found by the program.

Keyword CLOSE#

This is the keyword used to close an open file. Its normal form is simply:

```
CLOSE# Channel%
```

which will close the file with the channel designator `Channel%` - or, more accurately, it will close the communication channel to the file.

There can be a space between the `#` (“hash” character) and the channel designator - as shown here - but not between `CLOSE` and `#`. If you seek to close a channel which is already closed an error message telling you this will occur.

You can also use the more brutal form:

```
close# 0            ( a zero, not the letter O)
```

which will close all open files. Its use is not recommended in the multitasking Wimp environment, for fairly obvious reasons, so try to stick with the normal form.

Keywords PRINT# and INPUT#

These are the two most common keywords for manipulating the contents of data files. They are used to write/read numeric and string data to/from an open file, `PRINT#` sends/writes data to the file and `INPUT#` reads data from the file - and as already indicated, each time we use these keywords the appropriate file must already be open and the statement must be accompanied by the corresponding channel number. Examples of a typical statements are:

```
PRINT# Channel% , Integer% , Real , String$
INPUT# Channel% , A% , B , C$
```

The first statement would write the values of the three variables

`Integer% , Real and String$` into the file whose open channel number is `Channel%`. The three variables must already exist and their values would then be written to the file in the order given - and without any separation between them on the file. As usual, direct values or anything which evaluates to a direct value can be used in the statement instead of variables.

The second statement reads three items in turn from an open file into the variables shown i.e. `A%, B` and `C$`. An error will occur if the file does not hold values of the right type in the same order at the point in the file being read. Note that, in this case, the three variables do not need to exist prior to this statement. If they do not, the statement acts as a variable declaration operation as well.

It is worth stressing that the data file holds values only and it is up to the programmer to know the type(s) and order of the data before retrieving it.

Demonstration program

The following example - which is in two parts - brings all the above data file keywords into play. The odd line numbering in this first part is intentional.

Program Prog19a (First Part)

```
10 REM> Prog19a
40 Integer% = 99
50 Real = 67.89
60 String$ = "Pathetic example"
70 Channel% = OPENOUT ( "NewFile" )
90 PRINT# Channel% , Integer% , Real , String$
110 CLOSE# Channel%
120 Channel% = OPENIN ( "NewFile" )
140 INPUT# Channel% , A% , B , C$
160 CLOSE# Channel%
170 PRINT A% ; " " ; B ; " " ; C$
180 END
```

After declaring three variables of the different types, we create a new data file called `"NewFile"` in Line 70. Line 90 then writes the three items to the file in sequence, before closing it at Line 110. Lines 120-160 then open the file again - this time only for reading - and read the three items back into new variables (but of the right type in the right order) before closing the file again. Line 170 then just prints the contents of the new variables on screen to prove that it works.

The data file pointer and common error messages

You may be wondering why we bothered to close and re-open the file in Lines 110 and 120. Are those two lines redundant here? Well, apart from proving the point that the file existed, there is another reason. If you `REM` those two lines and run the program again two other points will come to the surface.

Firstly you will get an error message (from Line 140) saying `"end of the file"` - and , secondly, when you try to run this program yet again (after removing the `REMs`) it will now persist in giving you another error saying `"file already open"`. Get out of that! *(The easiest way is to open a Task Window, type* `"basic"` *- and then type* `CLOSE# Channel%` *in Immediate mode. Then remove the* `REMs` *from Lines 110 and 120 in the usual way, before re-running.)*

Here's what has happened:

The first error effectively tells us that data files have a pointer, just like `DATA` statements. When a data file is opened, the pointer is set to the start of the file and automatically moves as we take read or write action.

In our example, we started with a write action (because the file was empty) and so, after Line 90, the pointer ends up immediately after the data from our last write action i.e. after the data from `String$` - and, as long as the file remains open, the pointer will wait here for our next instruction.

So, if we `REM` Lines 110 and 120, the next action will be the read instruction at Line 140 - but there is nothing on the file to read beyond the current pointer position - hence the first error message.

By including Lines 110 and 120 the pointer is reset to the start by Line 120 and the read operation in Line 140 can take place without a problem.

This problem can also be overcome in another way, because Basic gives us another keyword to move the pointer at will - but let's cover the second error before going on to that.

The second error is best handled by good housekeeping and it was introduced deliberately to get you into a good habit early, because the problem is a nuisance and you are sure to run into it frequently.

A good way of dealing with it is always to include a special

‘flag’ in programs with data file operations - and to change this flag from `FALSE` to `TRUE` (and vice versa) each time you open (and close) a file. The same flag is then also included early in your general error trap, where it is used to close any files which are open at the time an error occurs.

If you add the following lines to the above listing, all will become clear:

Program Progl9a (Second Part)

```
20 FileOpen% = FALSE
30 ON ERROR PROCerror : END
80 FileOpen% = TRUE
100 FileOpen% = FALSE
130 FileOpen% = TRUE
150 FileOpen% = FALSE
190 :
200 DEF PROCerror
210 IF ( FileOpen% = TRUE ) THEN CLOSE# Channel%
220 REPORT
230 PRINT " at Line " ; ERL
240 ENDPROC
```

The flag `FileOpen%` is firstly declared, prior to adding the `PROCerror` call (to prevent `PROCerror` meeting an `"unknown variable"` error). Then, simply put `FileOpen% = TRUE` immediately after opening the file and `FileOpen% = FALSE` just before closing it.

Then, if a problem occurs whilst the file is open, the error trap in Lines 200-240 will safely close the file before it reports the error.

You would, of course, need to increase the number of such flags if you are using more than one file or more than one channel simultaneously.

Data file structure and pointer manipulation

To use data files effectively you need to know a few more keywords and then to have a look at how Basic actually stores the data in the files.

Keyword PTR#

ptr# is best regarded as a special variable which always holds the current position of the pointer in an open data file. It can therefore be used on either side of the equals sign in a Basic statement. For example, the sequence:

```
Channel% = OPENIN ( File$ )
<various read/write actions)
```

```
Position* = PTR# Channel%
```

will read the current position of the pointer in the file `File$` (being accessed via `Channel%`) and assign it to the variable `Position%`.

Alternatively:

```
PTR# Channel%= PTR# Channel% - 10
```

will (if the circumstances are right) reduce the value of `PTR# Channel%` by 10 and move the pointer to the new position i.e. back 10 steps here.

By the 'right circumstances' is meant, firstly, that the value of `PTR#` cannot be made less than 0 - the start of the file. The highest value it can take depends on several things, but the main one is that if the file is open for reading only, then `PTR#` cannot be made greater than the current length ('extent') of the file.

If, however, the file is open for writing, then `PTR#` can also be made greater than the current file extent - which will automatically increase the extent correspondingly.

Note that, if you have opened more than one channel to the same file (i.e. with more than one `OPENIN` action, for reading only) then each open channel will have its own pointer, which can be moved, within the limits of 0 and the end of the file, independently of the pointers of the other open channels.

The value of the pointer position is given in bytes from the start of the file - the first byte being position 0.

You'll remember, from Appendix 5, that one byte is a small amount of memory - 8 bits - which can store any number from 0-255 (&00-&FF). Generally speaking, and certainly in the context of data files, the number of bytes is used to denote/measure the size of a chunk of memory and/or the size of a file. *(Remember also the User Guide chapter on Discs and Filing Systems')*

So, the pointer can be moved backwards and forwards in steps as small as one byte - and therefore the number of bytes needed to store the different types of data now starts to become significant to the programmer.

Keyword EXT#

This keyword operates in a similar way to `PTR#` above i.e. it is used as a special variable. In this case it holds the current size (extent) of an open file. So:

```
Extent% = EXT# Channel%
```

will assign the current extent - in number of bytes - to the variable `Extent%`.

Similarly, if the file is open for writing:

```
EXT# Channel% = EXT# Channel% + &100
```

will increase `EXT# Channel%` and the file size by &100 (256 decimal) bytes.

You will see that, for a file open for reading only, the maximum value that `PTR#` can take is `EXT#`.

Keyword EOF#

This keyword is a simple function which returns the value true only when the associated data pointer finds itself at the end of the file. It is often used in the exit condition of a `REPEAT ... UNTIL` loop reading the contents of a file. For instance:

```
PTR# Channel% = 0
REPEAT
     INPUTS Channel% , dataitem%
UNTIL ( EOF# Channel% = TRUE )
```

will read a file of integer numbers from beginning to end. The first line sets the pointer of Channel% to the start of the file and then the `REPEAT` loop operates until `EOF# Channel%` is `TRUE`.

Note that, like `READ`, you do not need a loop counter to make this loop progress through the data, because - as already described - the keywords `INPUT#` and `PRINT#` automatically increment the pointer position by the correct amount for each data item read or written.

How data items are stored

File 'DUMPs'

The easiest way to see how data is stored in a data file is to create a small data file with different types of data items and then examine its contents byte by byte, using the star command `*DUMP`. This star command is very useful in that it produces a screen/printer output showing the contents of every byte in a named file of any type.

Program Prog19b carries out these steps. It simply opens a new file (called `"ByteData"`) and writes the following six data items to it - in the shown sequence:

the integer number	`8`
the integer number	`87654321`
the real number	`5.55`
the real number	`0.0987`
the string	`"String"`
the string	`"Another string"`

However, whilst doing so, it records the various positions of the pointer so that we can analyse these subsequently - with the aid of the DUMP display.

Program Prog19b

```
10 REM> Prog19b
20 REM** Demonstration of data file pointer and file
     structure. **
30 :
40 FileOpen% = FALSE
50 :
60 ON ERROR PROCerror : END
70 :
80 DIM Pointer%(8)
90 :
100 File% = "$.ByteData" :REM** Change file path to
     suit. **
110 :
120 Integer1% = 8
130 Integer2% = 87654321
140 Real1 = 5.55
150 Real2 = 0.0987
160 FirstString$ = "String"
170 SecondString$ = "Another string"
180 :
190 Channel% = OPENOUT ( File$ )
200 FileOpen% = TRUE
210 :
220 Pointer%(0) = PTR# Channel%
230 PRINT# Channel% , Integer1%
240 Pointer%(1) = PTR# Channel!
250 PRINT# Channel% , Integer2%
260 Pointer%(2) = PTR# Channel!
```

```
270 :
280 PRINTS Channel% , Reall
290 Pointer%(3) = PTR# Channel%
300 PRINT# Channel% , Real2
310 Pointer%(4) = PTR# Channel%
320 :
330 PRINT# Channel% , FirstString$
340 Pointer%(5) = PTR# Channel%
350 PRINT# Channel% , SecondString$
360 Pointer%(6) = PTR# Channel%
370 :
380 PTR# Channel% = PTR# Channel% + 3
390 :
400 PTR# Channel% = Pointer% (6)
410 INPUT# Channel%, Test?
420 Pointer%(7) = PTR# Channel%
430 :
440 Extent% = EXT# Channel%
450 Pointer%(8) = Extent%
460 :
470 FileOpen% = FALSE
480 CLOSE# Channel%
490 :
500 PRINT:PRINT TAB(10) "File 'ByteData'"
510 PRINT
520 FOR N% = 0 TO 8
530     PRINT "&" + STR$~(Pointer%(N%)) + " ";
540 NEXT
550 :
560 PRINT:PRINT:PRINT "Extent = &"; ~Extent%:PRINT
570 :
580 REM** Next line needs to have same filepath as Line
     80 **
590 *DUMP $.ByteData
600 :
610 PRINT:PRINT "Null"+Test$+"String"
620 :
630 END
640 :
650 DEF PROCerror
```

```
660 :
670 IF ( FileOpen% = TRUE ) THEN CLOSE# Channel%
680 :
690 REPORT
700 PRINT " at Line " ; ERL
710 :
720 ENDPROC
```

Lines 220-360 carry out the data file writing actions and also read the value of `PTR#` before and after each write action. These pointer positions are assigned in sequence to the elements of the array `Pointer()`.

After this, the file is extended by three extra bytes (Line 380) - just to show that `PTR#` can be used to do this.

The pointer is then moved back to its previous position (i.e. to where it was after the `"Another string"` write action) and a read action is attempted (Lines 410-420) again recording the pointer position afterwards - which will be explained shortly.

The extent of the file is then recorded, the pointer moved to the file end (and again its position noted) and the file finally closed.

Lines 500 - 610 print the results and `DUMP` the file - with Line 590 showing that the `*DUMP` command simply needs to be accompanied by a valid file specification.

When run, the program gives the output shown in Figure 19.2 if you are using Basic V. (With Basic VI the file will be 6 bytes longer and some of the numbers will be different - see later.) We need to examine this output and compare it with the sequence of events in the program.

```
    File 'ByteData'

&0  &5  &A  &10  &16  &1E  &2E  &30  &31

Extent = &31

Address    : 00 01 02 03 04 05 06 07

00000000   : 40 00 00 00 08 40 05 39   : @ . . . . @ . 9
00000008   : 7F B1 80 9A 99 99 31 83   : . * * * * * 1 *
00000010   : 80 C1 39 23 4A 7D 00 06   : * * 9 # J } . .
00000018   : 67 6E 69 72 74 53 00 0E   : g n i r t S . .
00000020   : 67 6E 69 72 74 73 20 72   : g n i r t s   r
00000028   : 65 68 74 6F 6E 41 00 00   : e h t o n A . .
00000030   : 00                        : .

NullString
```

These pointer positions have boxes round them

Figure 19.2

'Prog19b' output and file dump

The eight printed lines below `"Extent = &31"` are the result of the `DUMP` star command. Note that all numbers in the table are in hex and your output will not have the boxes round certain numbers.

Looking now at the line after the heading, the single line of nine hex numbers is the result of logging the data file pointer position before and after each write action.

So, moving from left to right, the pointer started at `&0` (the start of the file) before we wrote anything. After writing the integer number `8`, the pointer found itself at position `&5` - which is the starting point for the next writing action, the integer number `87654321`. After writing this the pointer arrived at `&A`.

The third and fourth writing actions were both real numbers (`5.5 5` and `0.0987`) and the corresponding pointer positions after writing them were `&10` and `&16`.

The final two writing actions were strings (of different length). After writing `"String"`, the pointer arrived at `&1E` - and after writing `"Another string"` the pointer ended up at `&2E`.

We then added 3 extra bytes to the file, moved the pointer back to position `&2E` and successfully made a read action - which put the pointer at `&30`. We then moved the pointer to the end of the file and logged its

final position as &31. The output line "Extent = &31" confirms this.

The next eight printed lines comprise the DUMP table. If, for the moment, you ignore the characters to the right of the right-most colon, the table should be fairly easy to understand. It simply lists in eight columns the numeric contents of (i.e. the value stored in) each byte of the file we have created. As it is a small file, it can show the whole file in less than seven rows.

The numbers on the left show the start address of each row of bytes, starting from zero, in the top left-hand corner. So, the start-of-file address &00 (ignoring the other six leading zeros) contains the value &40; address &01 contains the value zero (&00), and so on.

The nine addresses in the dump table which correspond with the nine pointer positions listed in the second line of the output have had a box put round them - so that you can easily follow the action.

Look at the end of the file first. After the final action, the pointer ended up at &31 - which is the empty byte immediately after the last byte with something in it. (Note that the value zero (&00) is not the same as an empty byte.) Note also that when the file was extended as in Line 380 i.e. without writing additional data items, the additional bytes are all filled with the value zero and not left empty. Hence the last three addresses of the file - &2E, &2F and &30 - have 00 in them.

Now let's go to the beginning of the file - to address &00 again, where the pointer always starts when the file is opened. After the first write action the pointer arrived at &05, which was, at that time, empty. Our second writing action used byte &05 as its first byte and moved the pointer to &A - and so on.

Thus, unless the programmer intervenes by deliberately moving the pointer in some other way, the effective movement of the pointer is in steps equal to the individual length(s) of the data item(s) being written/ read - rather than one byte at a time. Here, the pointer moved from box to box at each of our steps.

Data storage formats

With the above description, if you go through the dump table byte-by-byte comparing its contents with the writing actions of the program listing, you should be able to deduce that:

an integer number (whatever its value) always occupies 5 bytes, the first byte always having the value &40 and the remaining 4 bytes somehow representing the actual value

a real number (whatever its value) occupies 6 bytes, the first byte always having the value `&80` and the remaining 5 bytes somehow representing the actual value. (With Basic VI, there will be 9 bytes occupied, with the first byte being `&88`. This permits greater accuracy of real numbers.)

the string `"String"` occupies 8 bytes; the first byte having a value &00, the second byte having the value &06 (the length of `"String"`)

the string `"Another string"` occupies 16 bytes, the first byte having a value `&00` and the second byte having the value `&0E` (the length of `"Another string"` - the space counting as a character)

for strings, the remaining values of the occupied bytes are the ASCII codes of the string characters - in reverse order.

Hence, for strings:

No. of bytes occupied = (length of string + 2)

the first byte is always `&00`;

second byte = length of string;

and the remaining bytes contain the ASCII code of the string characters (reversed).

Don't worry about the way in which the actual values of the numbers are encoded - we don't need to know at the moment. However, it is worth noting that if you assign an integer value to a real variable and write it to a file as a real number, the value will be treated as a real number i.e. 6 bytes will be used (or 9 in Basic VI).

Data type identifiers

From the above it can be seen that the numbers in the first byte of any data item (i.e. `&00` or `&40` or `&80` or `&88`) are therefore identifiers which show the type of item which follows i.e. string, integer or real (Basic V and VI) respectively.

Remember that although we have shown how the pointer moves when writing to a Basic data file, the movement is exactly the same when we read an existing file. Thus, the pointer starts at position zero on opening the file, then moves to the end of each data item after each read action.

Incidentally, it is when the pointer finds itself in an empty byte that it changes the `EOF#` marker to `TRUE`.

Getting the pointer 'out of order'

We now need to explain the actions taken by the program around file addresses `&2E-&31` (Lines 400-450 of the program). The main purpose was to show that the Basic processor is fairly single-minded in dealing with the value `&00`.

We have already established that `&00` is the identifier for the start of a string - but there are many circumstances where a byte can have a value of zero without it being the start of a string. So what happens if we deliberately set the pointer to the first of a group of zero value bytes - address `&2E` here - and then tell it to read a string data item (Line 410)?

Well, as you can see in the final line of the output (Line 610) which has nothing between the words `"Null"` and `"String"`, the answer is that a null string is read i.e. the length of the string - held in the second byte - was also zero, so a perfectly valid read action on a string data item took place and, consistent with this, the pointer ended up at address `&30` afterwards.

By a similar process, if you deliberately move the pointer to a chance occurrence of the identifier value `&00` or `&40` or `&80`(`&88`) at an address which is not the start of a data item, you will probably be able to read one apparently valid data item - although printing it out may show it is not one that you entered. But after that, you will certainly raise an error message if you try to read further items, because you will be most unlikely to land on another identifier value after this read action.

Keeping track of pointer

The above descriptions have been somewhat detailed - because it is important to know exactly what is happening. It should, if nothing else, serve to emphasize that we need to be very alert to the pointer's location before we take any read or write action - particularly a write action.

Writing to an existing file (other than adding something to the end of the file) will overwrite the existing data - without warning.

Worse still, if (in the middle of several items of data) you overwrite one data item of one type with a data item of another type, the pointer will afterwards find itself (unless you are extremely lucky) in the middle of an incomplete data item - and at a location which does not hold an identifier value (i.e. not `&00, &40, &80(&88)` nor 'empty'). This will cause an error message if you try to read the file beyond the change. *This point is returned to in the next chapter.*

The characters on the right-hand side of the DUMP output in Figure 19.2 are simply the DUMP routine's helpful attempt to print CHR$(&nn), where &nn is the number stored in a particular byte. This helps identify stored strings, but is not always a benefit in other cases.

So, the first character is @ - which is CHR$ (&40), the number stored at position &00, and so on. If the stored value is less than &20, the character is not a printable one, so a full stop is printed instead. Characters for numbers above &7F vary according to your set-up (and are likely to be different on screen compared to printer). They have all been changed to the character "*" in the above table, for simplicity.

Organisation of data files

Most data files are used to store the information which the user sees on the screen and/or printer when using a database type of application. It therefore normally follows that the data is held as a set of records; each record having the same format.

It therefore becomes important to plan the way to hold this type of data in a data file. For instance, should the records be stored 'as they come' irrespective of differing record lengths, or should they be stored in equal length chunks of file space with each item of a record starting at the same relative position in each record? How is the data to be used and accessed? What search facilities will be needed? Will amendment of records be permitted or will actions be limited to 'read only'?

These are some of the questions that need to be answered before a data file is created. Essentially, it is another case of carrying out some planning first.

Fortunately, because it is a very common problem which can be handled effectively without needing to be specific to any particular computer system, there are many good books on this topic on the library shelves (usually under the heading of "Office Automation" or similar). There is also a good book on the subject specifically for Acorn computers. It is "File Handling for All" by David Spencer and Mike Williams, published by BeeBug, St Albans.

Final point

We have referred to data files as 'permanent' storage - and this is true when we are using a hard or floppy disc filing system. However, Basic data files work just the same with a 'RAM-disc' - which, of course, is definitely not 'permanent'. So be careful if you usually configure part of your RAM in this way.

20. Data storage/retrieval (cont'd) (Direct memory access/indirection)

DIM revisited - Reserving blocks of memory - Indirection operators - Entering and retrieving data.

The preceding chapters have looked at how to store data somewhere away from the program and how to retrieve it by loading it into temporary storage created by the program. The temporary storage was either ordinary variables or array variables, which themselves are small chunks of memory in RAM, identified by the variable name (or array name plus subscript).

There is another way in which memory can be used to store data temporarily and that is by getting the program to set aside a block of 'raw' memory, filling it with the data we want, then retrieving it from the block as needed. For these actions we need to use the keyword `DIM` in a new way and to use the 'indirection' operators.

DIM revisited

To set aside a block of memory we again use the keyword `DIM` - but in a different way to that already covered in Chapter 18.

The general form of this new way is:

```
DIM BlockName% SizeOfBlock%
```

There **must** be a space between `BlockName%` and `SizeOfBlock%`.

The starting address of the block is assigned automatically to the variable `BlockName%` - which is a numeric variable whose name, chosen by the programmer, also serves to identify the data block. It does not have to be an integer numeric name (as here) - a real numeric name is also OK.

`SizeOfBlock%` is the number of **bytes** to be reserved, **less one**.

Thus:

```
DIM Data% 255
```

would set aside a block of memory 256 bytes in length and place the starting address of the block in the variable `Data%`.

If we wanted to set aside more than one such data block we can use the form:

```
DIM Data% 255 , MoreData% 15
```

where, again, there must be a space between each variable name and the number of bytes required. This time, one block would be 256 bytes long and the other would be 16 bytes long.

As usual, numeric variable name references can be used instead of direct numbers to designate the required size of the data block.

When data blocks are reserved in this way they are not initialised i.e. the values stored in the reserved memory locations are not set to zero, for instance. It is also worth noting that, unlike arrays (see Chapter 18), you **can** 're-dimension' data blocks if you wish.

As we can refer to the variable name to get the start address of such a data block, we do not normally need to know the start address. However, it is worth knowing that it will be a 'word aligned' address (i.e. divisible by four - see Appendix 8).

Indirection Operators

Having created the reserved data block we need to be able to put data in it and to retrieve it subsequently - for which we use the four 'indirection' operators:

```
?             acts on 1 byte only.
!             acts on an integer number (i.e. 4 bytes
              at a time).
|(ASCII 124)  acts on a real number (i.e. 5 bytes in
              Basic V, or 8 bytes in Basic VI).
$             acts on strings (i.e. up to 256 bytes).
```

To show how these are used it is easiest to use examples.

Entering data

The instruction:

```
?DataBlock% = 77
```

would assign the value 77 to the single byte at address `DataBlock%` i.e. in the first byte of the reserved data block. And:

```
?(DataBlock% + 1) = 66
```

or

```
DataBlock%?1 = 66
```

would place the value 66 in the single byte at address `DataBlock% + 1` i.e. in the second byte of the reserved data block.

The latter construction - with the `?` operator 'surrounded' - is called the diadic form. The number (which can also be a numeric variable) after the `?` operator offsets the address by the number of bytes given by the number - which can be zero i.e. `DataBlock%?0` means the same address as `DataBlock%` i.e. no offset.

The action of the operator is therefore very similar to assigning a value to a variable - with the 'destination' on the LH side and the value on the RH side.

Note that it is usual not to use any spaces between the `?` operator and any adjoining item(s) - although, in fact, you can use a space **after** the `?` operator (but not before it, in the diadic form).

As `?` acts only on a single byte, the maximum value that can be entered is 255 - so the actual value entered in the above examples is `<value> MOD &100`. For example, the instruction:

```
?DataBlock% = 427
```

would be interpreted as:

```
?DataBlock% = 427 MOD &100
```

which would be, effectively:

```
?DataBlock% = 171
```

The `!` operator can be used in the same ways as `?`, but acting on four bytes. Thus, the instruction:

```
!DataBlock% = 2002002
```

assigns the value 2002002 (&1E8C52), as a 4-byte number, in the four consecutive bytes starting at address stored in `DataBlock%`.

The way it does this is to store the least significant byte in the lowest address and the most significant byte in the fourth address i.e 'backwards'. Thus, in our above example, we would find that the allocations to the four bytes are as follows:

```
The address 'DataBlock%'      will contain &52
The address 'DataBlock%+1'    will contain &8C
The address 'DataBlock%+2'    will contain &1E
The address 'DataBlock%+3'    will contain &00
```

(Remember, from Appendix 8, that BBC Basic uses the 'twos complement' form of integer number representation in four bytes and therefore allows integer numbers only in the range ±2,147,483,647)

As it operates on four bytes at a time, `!` is sometimes called the 'word operator' - *see Appendix 8* - but do not let this mislead you into thinking that `!` only acts on word-aligned addresses. It acts on 4 bytes with any starting address.

The `|` operator *(ASCII code 124 - appearing as ¦ on the labelling of some keyboards)* acts on five consecutive bytes - or eight in Basic VI - and is used for entering real number values. The only difference is that this operator **cannot** be used in the diadic form - so you must use `|(DataBlock% + 7)` rather than `DataBlock% | 7`.

Finally, the `$` operator is used to place strings (up to 255 characters long) directly into a data block. Thus:

```
$DataBlock% = "ABCDEFGH"
```

will place the 8-character string `"ABCDEFGH"` into the data block starting at address `DataBlock%`.

The ASCII code of each letter is stored in each byte, in order. So:

```
The address 'DataBlock%'will contain &41 ("A")
.
etc.
.
The address 'DataBlock% + 8'     will contain &48 ("H")
```

With this operator, one extra byte is used to place a 'carriage return' to mark the end of the string. Thus, in the above example:

```
The address 'DataBlock% + 9' will contain &0D
                               (the <return> key code)
```

Again, this operator cannot be used in the diadic form i.e. you must use `$(DataBlock%+16)` rather than `DataBlock%$16`.

Retrieving data

The same four operators described above are used to read data from a data block - but this time on the RH side of an assignment expression or as a function returning a value. For example:

```
OneByteValue% = ?DataBlock% :REM Simple assignment of
                                        value in 1 byte.

OneByteValue% = ?(DataBlock% + 3) :REM Simple
                      assignment of value in 1 byte.

Value% = DataBlock%!3 :REM Simple assignment of value
         in the 4 bytes starting at 'DataBlock% + 3'.

PRINT DataBlock%!0 :REM Prints the value stored at the
                4 bytes starting at 'DataBlock% + 0'.

Real = |(DataBlock% + 4) :REM Simple assignment of
        value in the 5 bytes (8 bytes in Basic VI)
           starting at 'DataBlock% + 4'.

String$=$DataBlock% :REM Simple assignment of string
            stored at the starting address
            'DataBlock%'. Note that false strings or
               errors will occur if a string is not
               already stored at the given starting
               address.

PRINT "Data string is " + $(DataBlock% + 44) :REM
                      Construction OK, but will give
                      an error if total length of new
                      string exceeds 255 characters.
```

Some of the rem comments above are very similar to those made in Chapters 17 and 19 about the importance of keeping track of the data pointer. The key issue is that, in all cases, the retrieval operation is merely interpreting the value(s) held in the byte(s) at the address(es) specified.

For instance, if you inadvertently call for a string to be output from a particular data block address and a string was not previously stored with that starting address, then the processor will not know. It will inexorably start reading the bytes until it runs into a value `&0D` (by chance, or perhaps at the end of a string stored at a higher address) - or until it has counted 256 bytes without finding one.

To demonstrate the point, try this sequence in a Task Window:

```
DIM Block% 400
Block%!0 = &0D444142
$(Block% + 4) = "Test string"
PRINT $Block%
```

and you will get the string `"BAD"` - because the processor faithfully looked for a string starting at `Block%` (where we had not assigned a string) and read three bytes before it came across `&0D` in the fourth byte. It therefore thought it had reached the end of a string and duly printed out the characters represented by ASCII codes `&42`, `&41` and `&44` i.e. “BAD”.

Now try:

```
PRINT $(Block + 6)
```

and

```
PRINT ~Block%!2
```

and you will get the answers `"st string"` and `&65540D44` respectively. With the above explanation, you should be able to work out why these answers occur.

Wimp programs

It is worth mentioning here that data blocks are essential tools in Wimp programming and Chapter 23 takes this a little further.

21. Colour

Non-Wimp colour - Number of colours available - Colour numbers - Default palettes in 2-, 4- and 16-colour modes - Changing palettes in these modes - Keywords COLOUR, GCOL, CLS and CLG - Demonstration of colour control/ choices - 256-colour modes and TINT - Brief introduction of GCOL with two parameters - AND, OR, EOR and NOT with colours - Demonstration - Keyword POINT(- TINT revisited.

As with the chapters on graphics and display modes, this book has to limit its introduction to colour - and, also as before, we will be confining our scope to non-Wimp applications. *(Colour in Wimp applications is handled differently and is beyond the scope of this book.)*

We also make the assumption that you appreciate that any colour can be produced by mixing the right amounts of three **primary** colours. As infants, we invariably ended up with muddy browns by mixing all the paints together. It's the opposite with light on your cathode ray tube - mixing 'all' the colours (red, green and blue) now produces white - and 'no colours' now brings a blank screen i.e. black. We also have the means of altering the brightness (intensity) of each primary colour.

Number of colours and Colour numbers

In non-Wimp programs the number of different colours available on the screen simultaneously is either 2, 4, 16 or 256 - depending on which screen mode you have chosen. The User Guide gives the details of which modes correspond with which number of colours and there is a choice of different screen resolutions with each. Naturally, more colours means more memory is used for display purposes.

When you first enter your chosen mode, a default set of colours (a 'palette') is assigned - corresponding to the number of colours available in that mode - and each colour is given a colour number. Also, the default foreground and background colour numbers (for both text and graphics) are chosen so that the foreground is white on a background of black.

For non-Wimp programs, in the 2, 4 and 16-colour modes BBC Basic provides the means to change the palette - and choices can be made from a total palette of 4096 colours. However, for non-Wimp in 256-colour screen modes, things are handled differently and it is best to leave the default palette as it is.

2, 4 and 16-colour screen modes

For the 2, 4 and 16 colour modes, these colours and colour numbers are as in Figure 21.1.

2-colour modes	16-colour modes	
0 = Black	*0 = Black*	*8 = Black/White**
1 = White	*1 = Red*	*9 = Red/Cyan**
	2 = Green	*10 = Green/Magenta**
4-colour modes	*3 = Yellow*	*11 = Yellow/Blue**
0 = Black	*4 = Blue*	*12 = Blue/Yellow**
1 = Red	*5 = Magenta*	*13 = Magenta/Green**
2 = Yellow	*6 = Cyan*	*14 = Cyan/Red**
3 = White	*7 = White*	*15 = White/Black**

** flashing*

Figure 21.1

Default Colour sets ('palettes')

If your program refers to a colour number higher than the range available in a given mode - as is quite often the case if you develop the program in one mode and change it later for some reason - the computer automatically reduces the colour number to come within the right range, by using the mod operator. For example, in a 16-colour screen mode, the instruction:

```
colour 22
```

would be read as:

```
colour (22 mod 16)
```

with the result being:

```
colour 6
```

In 2 and 4-colour modes the same instruction would be read as `COLOUR (22 MOD 2)` and `COLOUR (22 MOD 4)` respectively - giving results of `COLOUR 0` and `COLOUR 2` respectively.

Thus programs will not return an error in these cases, but you will almost certainly see a different colour to what was intended - which, of course, might render text/graphics invisible if the original background/ foreground colour contrasts are eliminated.

You will note that colour numbers 8-15 in 16-colour modes are flashing colours by default. You can exercise considerable control over the flashing effect with the `VDU 23,9` and `VDU 23,10` commands - and the flashing effect can also be eliminated.

You may find your computer is configured to prevent the flashing effect. The probable answer is to issue the instruction:

```
SYS "OSByte", 9, 25
```

before your program prints!draws anything on the screen - and:

```
SYS "OSByte", 9, 0
```

at the end of the program if you want to return things to as they were. We introduce `SYS` *calls later.*

Keywords COLOUR and GCOL (plus CLS and CLG)

We have briefly touched on these keywords before. They are used to choose the required foreground and background colours (from the current palette) prior to writing text or drawing graphics on the screen, as follows:

`COLOUR ColourNumber%` (for text)

or

`GCOL Colour Number%` (for graphics, including 'VDU5 text')

Thus, assuming that we are in a 16-colour mode and using the default colours as in Figure 21.1, the sequence of instructions:

```
COLOUR 7
COLOUR 132
GCOL 6
GCOL 132
```

would set both graphics and text background colours to Blue, with the text in White and any drawings (or 'VDU5 text') in Cyan. (Cyan is a sky-blue colour.)

These instructions would only choose the colours for the subsequent text/graphics actions. To see them on the screen we need to take, for instance, some `PRINT` or `PLOT` actions - and if we want the whole of the text and/or graphic viewport to have the chosen background colour(s) we need to 'clear' the viewports first, using the keywords:

`CLS` (for the text viewport)

and

`CLG` (for the graphics viewport)

Because it is more normal to want to see the viewport with the chosen background colour, it is very common to see the combination:

```
COLOUR 132 : CLS
```

or

```
GCOL 132 : CLG
```

as a two-statement line - a fairly harmless exception to the one-statement-per-line rule.

If you do not clear the screen to the background colour first, you will find that the chosen text background colour shows as a box around text written on the screen - but the same does not with the graphics background colour when using “VDU5 text‘.

Finally, note that `VDU 17` (with 1 extra byte - *see Appendix 9*) can be used instead of `COLOUR` in the above usage.

Changing the palette

It is worth just pausing to think what we mean by changing the palette. We have already seen that we use the colour numbers in the Basic statements to select which colours to show on the screen - in both text or graphics actions. So, to change the palette, we need a means to assign a different colour to a particular colour number - and we therefore have the concept that the colour numbers themselves are entities which are independent of the actual colour assigned to them at any time. We call these numbers the **logical colour numbers** (or just “the logical colour”), because they do not, in themselves, imply any particular colour.

In fact, in our use of `COLOUR` and `GCOL` above (i.e. to select the colour for subsequent print/graphics actions) it is the logical colour number that is used in the variable `ColourNumber%` - and the colour currently assigned to the chosen number will be used.

In the 2, 4 and 16-colour screen modes it is very easy to change the palette, using the keyword `COLOUR` with four parameters. The general form of the instruction is:

```
COLOUR  LogicalNumber% , Red% , Green% , Blue%
```

where `Red%`, `Green%` and `Blue%` define the amounts (intensity) of each of the primary colours to be mixed to produce the new colour - to be assigned to logical colour number `LogicalNumber%`.

`LogicalNumber%` needs to be within the correct range for the screen mode in use - and so will be reduced automatically in the same way as before.

`Red%`, `Green%` and `Blue%` are numbers in the range 0-255, with 0 meaning none of that primary colour to be used and 255 meaning the maximum intensity of that primary colour to be used. However, there are only 16 different different levels of intensity currently available - so values 0-15 all have the same effect as ‘intensity level 0’; 16-31 as ‘intensity level 1’ and so on. In other words, choose a level from 0-15 for the amount of each colour to be mixed - then multiply each amount by 16 in the instruction.

With 16 intensity levels available for each of the three primary colours, there are 4096 separate actual colours to choose from - and there is no bar to assigning the same actual colour to more than one logical colour number.

Unfortunately, there is no easy way to show all 4096 available colours on the screen at the same time - but ***Program Prog21a*** - which produces the screenshot in Figure 21.2 - shows 15 colours at a time, picked randomly - together with their component amounts (in intensity levels, 0-15) and this will be a help. Each time you press a letter key the colours will change. Press `<Esc>` to exit.

Figure 21.2

Output from program 'Prog21a'

Program Prog2la

```
 10 REM> Prog2la
 20 REM** Demonstration of available colours for 2-, 4-
      and 16-colour screen modes. **
 30 :
 40 MODE 12 :REM** A 16-colour mode. **
 50 :
 60 COLOUR 0 , 255 , 255 , 255 :REM** Makes background
      colour white - background defaults to Colour 0.
      **
 70 :
 80 REPEAT
 90     FOR Box% = 1 TO 15
100          :
110          Red% = 16 * (RND(16)- 1)
120          Green% = 16 * (RND(16)-1)
130          Blue% = 16 * (RND(16) -1)
140          :
150          COLOUR Box% , Red% , Green% , Blue%
160          :
170          GCOL Box%
180          RECTANGLE FILL (64) , (Box% MOD 16)*64 , 64
      , 32
190          :
200          VDU5
210          MOVE BY 64 , 0
220          PRINT "Red = " ; STR$(Red% DIV 16);
230          :
240          MOVE BY 64 - (16 * LEN(STR$(Red% DIV 16)))
      , 0
250          PRINT " : Green = " ; STR$(Green% DIV 16);
260          :
270          MOVE BY 64 - (16 * LEN(STR$(Green% DIV
      16))) , 0
280          PRINT " : Blue = " ; STR$(Blue% DIV 16)
290          VDU4
300      NEXT
310      :
320      G=GET
330      CLG
```

```
340     :
350 UNTIL FALSE
360     :
370 END
```

Using colours from the 16-colour default palette

BBC Basic provides special facilities for the 2, 4 and 16-colour screen modes if you want to use any of the colours from the **16-colour default palette** - by using the keyword `COLOUR` with only two parameters. The general form is:

```
COLOUR LogicalNumber% , ActualNumber%
```

where the `ActualNumber%` is the number of the colour shown in Figure 21.1 for 16-colour modes. For example:

```
COLOUR 1 , 13
```

will assign flashing Magenta/Green to logical colour 1 in any of the 2, 4 and 16 colour screen modes.

In a 2-colour mode - assuming you have not also changed logical colour 0, which will be Black and also the default background colour - all your foreground text/graphics will therefore be limited to flashing Magenta/ Green.

In a 4-colour screen mode, you will still have logical colours 2 and 3 available to be separately assigned if you so wish - and logical colours 2 to 15 still to be assigned in 16-colour modes.

Other points

It may be obvious, but there is only one palette available at any one time - for both text and graphics actions. So, changing **the palette** with colour (with 2 or 4 parameters) means that both text and graphics colours are affected. Our previous use of colour (with one parameter) to set the text colour only, may have left some confusion on this point.

If there is already something on the screen when you change the palette, those colours affected will change instantly. This can be useful: for instance, to erase a message/graphic by writing it on the screen using a certain logical colour number and then changing that logical colour to the background colour - and doing the reverse to reveal the message/graphic again.

Note that the pre-set flashing colours 8-15 (assuming they are flashing in your configuration!) can be regarded as being available **in addition** to any of 4096 colours i.e. you can define/redefine some logical colours using `COLOUR LogicalNumber% , Red% , Green% , Blue%` and others using `COLOUR LogicalNumber% , ActualNumber%`. *(Numbers 0-7 of the 16-colour default palette can also be obtained in the '4096 method'. but you can use either method to assign these eight colours.)*

`VDU 19` can be used instead of `COLOUR` in this section.

256-colour screen modes (and the keyword TINT)

At this level of introduction **it is best to regard the 256-colour modes as having a fixed palette.**

Like the other modes described above, colours are chosen by colour numbers, but this time the numbers run from 0-63 - each of which can have four different shades, giving 256 colours in all.

The colours associated with the numbers 0-63 are derived from the binary form of these numbers *(see Appendix 5)*. All the integer numbers from 0-63 can be represented in binary by no more than six bits - 63 being 111111. By splitting these six bits into three pairs, each pair is used to define the amount of each of the three primary colours in the resulting colour. Thus, using a space between the binary pairs for emphasis:

Colour Number	Binary	Primary Colour
1	00 00 01	dark red
2	00 00 10	mid-red
3	00 00 11	bright red
4	00 01 00	dark green
8	00 10 00	mid-green
12	00 11 00	bright green
16	01 00 00	dark blue
32	10 00 00	mid-blue
48	11 00 00	bright blue

Numbers 0 and 63 are Black and White respectively - and all the numbers between those listed are mixtures of more than one primary colour. For example, colour number 45 is binary 10 11 01 and hence is the colour resulting from mixing mid-blue (10 `gg rr`) and bright green (`bb` 11 `rr`) and dark red (`bb gg` 01). It is not unusual to refer to colour numbers derived in this fashion as `rrggbb`.

The shade/tint is chosen simply by adding the new keyword `TINT` to the colour selection instruction. The general form is:

```
COLOUR ColourNumber% TINT TintLevel%
```

for text, or:

```
GCOL ColourNumber% TINT TintLevel%
```

for graphics. *(Remember - TINT is only available in 256 colour screen modes.)*

`TintLevel%` is a number from 0-255 where 0 means the darkest shade and 255 means the brightest shade - but there are only four shades available. Hence, `TintLevel%` values 0-63 all produce the darkest shade - and so on.

It is easier to consider the shade levels as 0, 1, 2 and 3 and then multiply the choice by 64 to assign a value to `TintLevel%`.

For example:

```
COLOUR 16 TINT (3*64)
```

would select the brightest shade of dark blue (!) as the foreground colour for subsequent text printing. Or:

```
GCOL 20 TINT 64
```

would select the 'level 1' shade of a colour (010100) formed by mixing dark blue (01ggrr) with dark green (bb01rr).

These examples also give a strong clue why you are unlikely ever to see the 256 colours listed in a table similar to Figure 21.1. Choosing acceptable colour names would be very difficult! The easiest thing is to show them all on the screen - which is what ***Program Prog21b*** does.

The program produces four, apparently identical, blocks of 64 colours - but closer examination will show that each block is in a different shade/ tint (lightest at the top of the screen). In addition, an "X" has been printed in each colour box which has equal amounts of the three primary colours i.e. these boxes contain the different shades of grey, from white to black.

Program Prog21b

```
10 REM> Prog21b
20 REM** Colour palette in 256-colour screen modes **
30 :
40 MODE 15
50 :
60 FOR Tint% = 0 TO 192 STEP 64
70     :
80     FOR Colour% = 0 TO 63
90         :
100        R% = Colour% AND 3
110        G% = (Colour% AND 12) >>> 2
120        B% = (Colour% AND 48) >>> 4
130        :
140        GCOL Colour% TINT Tint%
150        RECTANGLE FILL 128 + (64 * (Colour% MOD
     16)) , (Tint% / 8) + (4 * Tint%) + (Colour% DIV
     16) * 56 , 56 , 48
160        :
170        IF R% = G% AND G% = B% THEN
180            VDU5
190            MOVE BY -32 , -8
200            GCOL 0
210            PRINT "X";
220            VDU4
230            MOVE BY 16 , 8
240        ENDIF
250        :
260    NEXT
270    :
280    :
290 NEXT
300 END
```

Default TINT levels

Note that, if you do not add any `TINT` instruction the default is that foreground colours will be given tint level 3 (the brightest shade/tint) and background colours will be given tint level 0 (the darkest tint/shade).

This means that if you set the **same** colour number to foreground and background in 256-colour screen modes (i.e. using `ColourNumber%` and `128+ColourNumber%`) - without using `TINT` - they will in fact be different shades of that chosen colour.

Special use of TINT to change tint

The shade/tint can also be selected in a sometimes helpful alternative way, using `TINT` with two parameters:

```
TINT Factor% , TintLevel%
```

`TintLevel%` is exactly as before, but `Factor%` is a number from 0-3 with the following meanings:

0 Current text foreground colour has tint level set

1 Current text background colour has tint level set

2 Current graphics foreground colour has tint level set

3 Current graphics background colour has tint level set

(There is a further use for the keyword tint which we will leave until later in this chapter.)

Final comments on keyword COLOUR

We have used colour with the conventional 'English-English' spelling in this book, but it is worth noting that the 'American-English' spelling `COLOR` can be used also - it will be changed automatically to colour on listing the program.

More on GCOL

So far, we have used GCOL with one parameter only - to select the graphics foreground and background colours - but it has a much wider use, which we will touch upon.

The general form is:

```
GCOL ActionNumber% , ColourNumber%
```

`ColourNumber%` is exactly as we have already seen it (i.e. a logical colour number) and `ActionNumber%` is an integer number in the range 0-7, meaning:

Action	Meaning
0	'paint' with `ColourNumber%` on the screen
1	`OR` the colour already on the screen with ColourNumber%
2	AND the colour already on the screen with `ColourNumber%`
3	`EOR` the colour already on the screen with `ColourNumber%`
4	Invert the colour already on the screen, (ignoring `ColourNumber%`)
5	Do nothing
6	`AND` the colour already on the screen with (`NOT ColourNumber%`)
7	`OR` the colour already on the screen with (`NOT ColourNumber%`)

Clearly, this table requires an understanding of Appendix 6 to establish the results of these actions - and then might still raise a query about why anyone would want to take these actions anyway!

As usual, it is more complicated to read about it than to do it, so a demonstration will help and ***Program Prog21c*** provides this.

The demonstration plots overlapping squares on the screen and prints the resulting logical colour numbers which occur at various points on the screen after each step.

The program also serves to introduce the keyword `POINT(` - which we first mentioned in passing in Chapter 12 - and it will be best to describe this before giving the listing.

Keyword POINT(and TINT re-visited

The keyword point (is a function that simply returns the number of the logical colour currently on the graphics screen at the point specified. Thus:

```
PixelColour% = POINT(512 , 512)
```

would assign to `PixelColour%` the graphics logical colour number at the centre of the default standard screen - using a number in the range 0-15

for 16-colour screen modes and correspondingly reduced ranges for 2- and 4-colour screen modes.

Note that if the specified point is not within the currently-defined graphics viewport, the value returned is `-1`.

In 256-colour screen modes, the colour number returned by `POINT(` is in the range 0-63 and the keyword `TINT` - used this time as a function in the same way as `POINT(` above - will return the tint, with a value in the range 0-255.

Program Prog21c

```
 10 REM> Prog21c
 20 REM** Demonstration of use of GCOL with two
     parameters. **
 30 :
 40 MODE 12
 50 :
 60 REM COLOUR 3 , 6
 70 REM COLOUR 5 , 7
 80 :
 90 VDU 5
 100 :
 110 GCOL4
 120 RECTANGLE FILL 200 , 200 , 800
 130 PROCprintColour
 140 G=GET
 150 :
 160 GCOL 1 , 3
 170 RECTANGLE FILL 400 , 400 , 800
 180 PROCprintColour
 190 G=GET
 200 :
 210 GCOL 1 , 5
 220 RECTANGLE FILL 300 , 100 , 500
 230 PROCprintColour
 240 G=GET
 250 :
 260 VDU20
 270 PROCprintColour
 280 G=GET
```

```
290 :
300 VDU4
310 END
320 :
330 DEF PROCprintColour
340 GCOL1
350 MOVE 250 , 900
360 PRINT; POINT(250 , 900)
370 :
380 MOVE 450 , 900
390 PRINT; POINT(450 , 900)
400 :
410 MOVE 1100 , 900
420 PRINT; POINT(1100 , 900)
430 :
440 MOVE 300 , 150
450 PRINT; POINT(300 , 150)
460 :
470 MOVE 350 , 300
480 PRINT; POINT(350 , 300)
490 :
500 MOVE 450 , 450
510 PRINT; POINT(450 , 450)
520 :
530 ENDPROC
```

The actions in Lines 60 and 70 should firstly be `REMed` (as in the listing) to ensure the default 16-colour palette is used in mode 12. The sequence is then as follows:

`GCOL4` is used, so the first action is a Blue square plotted on a Black background - and the various printed points show 4 and 0 respectively.

A second square, overlapping the first square, is now plotted using `GCOL 1 , 3` i.e. `ORing` the screen colour(s) with logical colour 3 (Yellow in default). Now,

`4 OR 3 = 7` (White by default)

and

`0 OR 3 = 3` (Yellow by default)

So where the second square overlaps the first the result is logical colour 7 (White by default) - and where there is no overlap the result is logical colour 3 (Yellow by default). These logical colour numbers are now shown at the appropriate points - changing from their values after the first step.

A third square is now plotted overlapping the two previous squares and the background, using GCOL 1 , 5 i.e. ORing the screen colour(s) with logical colour 5 (Magenta by default). The resulting logical colours this time are:

`0 OR 5 = 5` (Magenta by default)

`4 OR 5 = 5` (Magenta by default)

`7 OR 5 = 7` (White by default)

`VDU 20` returns everything to the default palette - so here there is no change with this last step.

If you now re-run the program with the REMs in Lines 60 and 70 taken out - one at a time, then both - you will see that the logical colour **number sequences on the screen stay exactly the same** as in the first sequence above, despite the fact that we see some different colours - because Lines 60 and 70 alter the default palette. In particular, with Line 70 in action, there will be two areas of White on the screen after one step - but with different logical colour numbers.

This proves that the `GCOL` operations use logical colour numbers - showing on screen the actual colours assigned to the logical numbers at the time - see Figure 21.3 also.

`Program Prog21c` can easily be adapted to demonstrate the other `GCOL ActionNumber%` values and this is recommended ‘homework’.

Finally, `VDU 18` (with two extra bytes - *see Appendix 9*) can be used instead of `GCOL` in this usage.

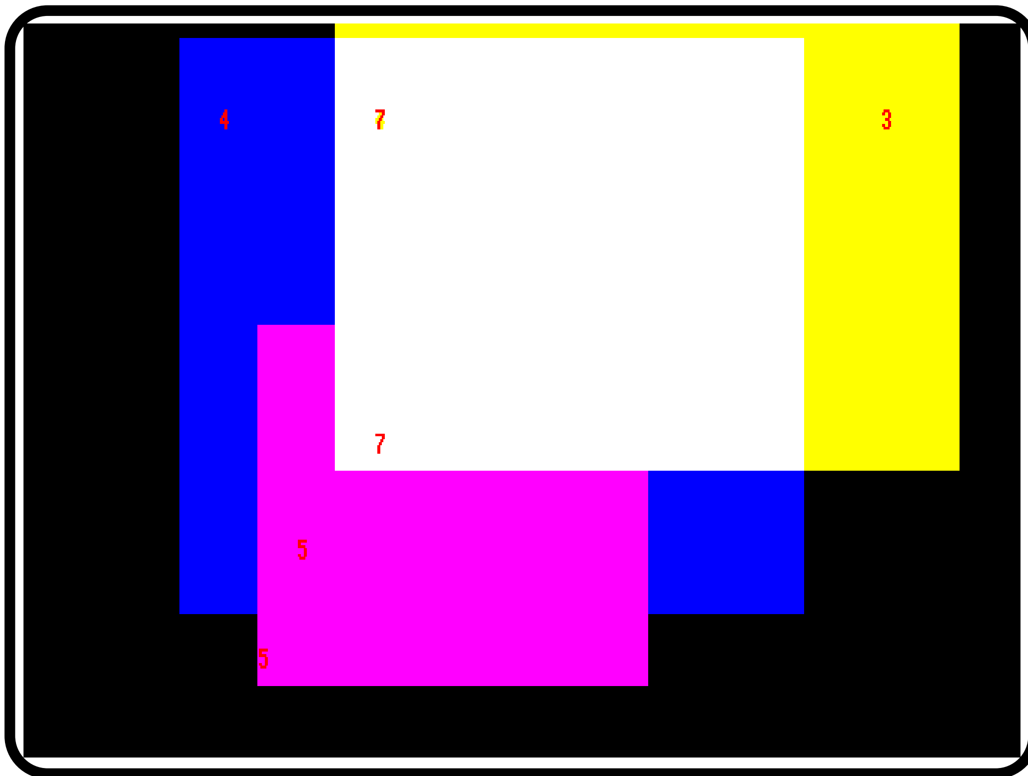


Figure 21.3

Effect of OR with colours, program 'Prog21c'.

These GCOL actions are very much used in graphics programs - particularly animated graphics and CAD applications - as they provide the means of simulating the effect of objects at different distances from the viewer. Thus objects can move realistically in front of a background picture and/or behind a foreground feature.

22. Libraries

Importance of libraries to Wimp programming - Keywords LIBRARY, INSTALL and OVERLAY introduced and compared - Building a library, with example.

We now move on to a topic which is very much a key part of Wimp programming, but can be used equally effectively in the non-Wimp environment.

You will recall - back in Chapter 5 - that one of the main uses of PROC/FNs is to hold routines which are going to be used often within a program. It is a natural extension of that to find that many programs have a need for the same PROC/FNs.

Whilst it would be entirely feasible to copy such PROC/FNs from program to program as needed, it is often much more convenient to use a 'library' of PROC/FNs.

A library is simply a collection of DEF PROC/FNs, in normal Basic program form and saved as a normal Basic file. This file is then identified to/by the program that wishes to use any of the PROC/FNs in the library - and thereafter these PROC/FNs can be called by the program in the usual manner.

There are three ways to use libraries and the corresponding keywords are library, install and overlay.

Keyword LIBRARY

This is probably the easiest method to use. LIBRARY is a normal Basic statement (i.e. normally used within a program) which takes the form:

```
LIBRARY FileSpec$
```

where FileSpec$ is a string which needs to evaluate to a valid file

specification (for the filing system in use) of the library file you wish to use.

The program then finds the file and loads it into RAM just above the program which called it. For all practical purposes, the effect is that all the PROC/FNs in this library are now part of your program, just as if you had added them to the program listing yourself. You can call them at will during the program but they will disappear like ordinary variables when a new program is loaded/run.

You can use LIBRARY to load as many library files as you wish - within the limits of the RAM available for Basic operations. When a PROC/FN is called, the most recently loaded library file will be searched first (but after the normal program DEF PROC/FNs, if any). This is not usually a significant issue but it's nice to know in case you want to hone a program finely for speed.

Because it allocates memory space and produces 'fatal' errors if the library file cannot be found, it is best to use library very early in the program and it is usual to see it (or them) as the first real statement i.e. after any introductory REMs.

Keyword INSTALL

This keyword is a command which cannot be used within a program. Therefore, before loading a Basic program file:

```
INSTALL FileSpec$
```

must be typed in from the command line or Task Window or contained in a command script file such as an 'Obey' file - *see User Guide*.

It works in a very similar fashion to LIBRARY, but this time the library(ies) are loaded right at the top of the available Basic RAM and the RAM available to Basic programs is then reduced correspondingly.

Thus, this method is a little more 'permanent' than using library and the loaded library(ies) will not be lost until you quit the Basic environment. This means that new Basic programs can use the loaded library(ies) without further ado.

You can, of course, use INSTALL and still use LIBRARY. On calling a PROC/FN any INSTALL libraries are searched (again, starting with the most recent), but after any LIBRARY libraries.

Keyword OVERLAY

This final method uses less memory space but is more difficult to manage in practice. It is included here solely to complete the set, so to speak, and because the description only needs to be short - but it is unlikely that beginners will need it.

Its syntax is different from the above methods and the general form is:

```
OVERLAY Lib1$ , Lib2$ , etc.
```

where `Lib1$` etc. are each valid file specification strings of library files. When used (as a normal Basic programming statement), the processor firstly checks which is the longest file in the given list and reserves space in RAM for this.

When a `PROC/FN` is called which does not have a `DEF` in the normal program listing, the processor will first search any libraries loaded by `LIBRARY` and `INSTALL` and - assuming it does not found a match - it will then **load each of the given `OVERLAY` libraries in turn** into the same reserved space until it finds the matching `DEF`. (Each loading overwrites the previous `OVERLAY` loading.) The library containing the match then **stays loaded** until another proc/fn is called and cannot be found elsewhere.

Once a match is found the processor 'remembers' the library in which it was located and loads that library straightaway if the `PROC/FN` is called again and the correct library is not still loaded.

The advantages and disadvantages of `OVERLAY` are probably clear to see. The memory space needed is much reduced - but a good deal of thinking and library file organisation is needed to avoid problems. For instance, you cannot call a `PROC/FN` in one `OVERLAY` library from a `PROC/FN` in another `OVERLAY` library. However, you can get rid of libraries from the overlay list if you know it (or they) will not be needed later in the program - by just re-issuing the `OVERLAY` statement with null strings to replace those library files no longer needed.

It has to be said that you need to be really up against the end stops on memory space to consider using overlay - indeed, for the 'amateur' it would have been a most useful facility in the early days of BBC Basic - when it was not available and the same thing had to be done in less 'hygienic' ways!

Building PROC/FN libraries

There are a few points to watch when building your own set(s) of libraries.

Program statements in libraries must not use any Line Number references - so, if `DATA` lines are included within library listings the `RESTORE+` form needs to be used to access the `DATA` items safely.

The line numbers you use in your library listings are not too important - you can start from 10 in each library if you like and it will not cause conflicts with your program numbering or other libraries. *However, there is a small advantage in starting a library listing from a high number (e.g. 20000) because this will reduce the chances of problems if you ever want to copy any of the* `DEFs` *into programs directly.*

All `DEF PROC/FNs` ought to be 'free standing' as described in Chapter 15. That is, all variables in the `DEF` should be made local, either by use of `LOCAL` or by including them as formal parameters. This is very important in order to achieve 'transportability' to any program.

Each library should best include one `DEF PROC` specially-produced to print out information about each of the `DEF PROC/FNs` contained therein. This needs only to be a series of print statements which put the details on screen when the `PROC` is called, from Basic Immediate mode for instance.

Each library should use its first line as a `REM` containing a descriptive heading (including a Version number) for the library, plus the name of the above special `PROC` so you can see which `PROC` to call to see more information.

This is more than good housekeeping, as there is a BBC Basic command `LVAR` which can be used to list all defined variables, the names of any `PROC/FNs` which have been called plus the first line of any loaded libraries (in the order they will be searched). `LVAR` *is fairly easy to use in the non-Wimp environment. First enter Basic from a Task Window, load and run the program of interest; and then type* `LVAR` *to get the list.*

Be careful to keep separate Wimp and non-Wimp libraries. Several `SYS` calls, for instance, are meaningless or will not

work in the non-Wimp environment. Conversely, as we have stressed, several Basic keywords ought not to be used in Wimp programs. It is always a good idea to check out any proposed library `PROC/FN` in both environments first if you are not sure.

The listing ***Program Prog22a*** (which, for once, cannot be run and is not on the disc) shows the form of a typical library with the above features. `DEF PROCprintLibHelp` is an example of the special `PROC` referred to in the fourth point above.

Program Prog22a (not for running!)

```
10000 REM> PrintLib : Version 1.2 *** Text and number
      formatting routines (for both Wimp and non-Wimp
      programs). Call 'PROCprintLibHelp' for details of
      each PROC/FN ***
10010 REM***************************************
10020 DEF PROCprintLibHelp
10030 REM*** Prints out details of all PROC/FNs in
      library. **
10040 PRINT "PROCcentrePrint(String$)"
10050 PRINT "Places a string centrally on a
      80-character screenwidth line"
10060 PRINT
10070 PRINT "FNnumberToString(Number,DecPlaces%)"
10080 PRINT "Converts any real number to string form,
      correctly rounded to chosen number of decimal
      places. Adds leading or trailing zeros as
      necessary."
10090 PRINT
10100
. (etc.)
.
.
10300
10310 ENDPROC
10320
10330 REM***************************************
10340
```

```
10350 DEF PROCcentrePrint(String$)
10360
. (definition lines)
.
.
.
10560
10570 ENDPROC
10580
10590 REM****************************************
10600
10610 DEF FNnumberToString(Number,DecPlaces%)
10620
. (definition lines)
.
.
.
10870
10880 =NumberString$
10890
10900 REM****************************************
10910
10920 (etc.)
.
.
```

In our brief introduction to Wimp programs (Chapter 24) it will become clear why libraries are essential for sanity.

23. Using the Operating System (SYS calls)

Access to Operating System - Keyword SYS - Detailed examination of SYS call structure - SWIs - Use of commas to avoid ambiguity of input/output parameters - Parameter blocks.

To function properly, a computer needs to carry out some fairly basic tasks over and over again e.g. write a character onto the screen; read the mouse position; keep track of text and graphic cursor positions; manage the memory; deal with keyboard presses etc. These sort of fundamental tasks need to be carried out during any program, whatever language it is written in - and the management of them is the job of the operating system (OS). (Called the "RISC OS" in Acorn machines).

In the same way that Basic uses `PROC/FNs` for frequently-used Basic routines, any OS has a large number of machine code routines at its disposal, which it uses for its repeated tasks.

These routines can sometimes be very useful to the Basic programmer and, from the earliest version of BBC Basic, great importance has been given to making it easy for the programmer to get access to them.

In the earlier versions of BBC Basic the star command `*FX` was provided for this purpose (and is still available today, for backward compatibility). This allowed very many OS features to be selected/deselected e.g. enable/disable the arrow keys; set flash rate of flashing colours; etc. This worked very well with 'set/reset' types of actions but was limited and somewhat awkward to handle when input parameters needed to be passed to, or output results were wanted from, the OS routines.

In BBC Basic V a much-improved facility is made available with the keyword `SYS`.

Keyword SYS

The general format of `SYS` is best described with a real example:

```
SYS "OS_ReadModeVariable" , -1 , 1 TO ,, ScreenWidth%
```

This particular SYS call, as the words imply, reads information about the screen mode - but let's concentrate on the general structure of the SYS statement rather than the particular example.

In its simplest form, a SYS statement breaks down into three parts: name, input parameters (if any) and output variables (if any) and the example above contains all three.

Name

The first item after the keyword SYS is `"os_ReadModeVariable"` and this one-word string (i.e. no spaces) is the name of the particular routine. Note that the name is very similar in construction to a variable name and the use of capital letters at the start of each 'sub-word' is the same policy used in this book for naming variables. As with variables, the match must be exact i.e. the string name is case sensitive - so `"os_Readmodevariable"` will not do.

Acorn's Programmer's Reference Manual (PRM) lists 18 pages (!) of these routines - which are referred to as SWIs ("Software Interrupts" for the name-droppers). *With such a large number of SWIs available there really is no option other than using the PRM if you need to know fully what is available. Fortunately, we need to know only a few SWIs for the purposes of this book and they will be introduced as needed.*

Each SWI also has a number, which can be used instead of the string name. Thus, in our example:

```
SYS &35 , -1 , 1 TO ,, ScreenWidth%
```

would do the same thing. (It is usual to use hex numbers for SWIs.) The string version makes it far easier to read what a listing intends, so you are recommended to keep to this - although the numbers are processed faster. *(As you might expect, there are SWIs to convert SWI names to numbers and vice versa. They are* `"OS_SWINumberToString"` *(&38) and* `"OS_SWINumberFromString"` *(&39).)*

Variables can also be substituted for the direct SWI name/number if you wish. Thus:

```
ModeInfo% = &35
SYS Modelnfo% , -1 , 1 TO ,, ScreenWidth%
```

or:

```
ModeInfo$ = "OS_ReadModeVariable"
SYS ModeInfo, -1 , 1 TO ,, ScreenWidth%
```

would each work equally well.

Input parameters

After the SWI name/number, but before the `TO`, appear some numbers separated by commas - including the comma after the SWI name/number. These are input parameter values being passed by the programmer to the routine. The parameter values need to be in the order and of the type specified (by the PRM) for the particular SWI, which may be either an integer number or a string - real numbers are not used here. In particular, memory locations (integers) are very commonly required as input parameters.

These parameters are passed to special storage locations called "registers", of which up to eight are available to Basic, named `R0 to R7`.

Not all SWIs use all eight registers and in our example only two are used and both need integer numbers (i.e. here ro is -1 and rl is l). Again, the PRM is needed to find the details of how many parameters/registers are used, what each of them means and what type and range of values each can take.

In our example, `R0` is used to pass the screen mode number we are interested in, or `-1` is used to denote "the current mode in use". So, we could have put, say, `12` here if we had wanted to know about `MODE 12`. The next parameter/register (`R1`) is used to choose which item of mode information we want to know about - and the PRM lists thirteen items to choose from. Our example has used `1`, which means "How many text characters per line in this mode?" (Other items include No. of text lines, maximum no. of colours, pixel size, etc.)

When a string is passed as a parameter, it is actually the memory address of the string that is passed to the register - and this is done automatically by Basic, so the programmer need only enter the string in the right place in the `SYS` statement (directly in quotes, or by reference to a string variable name).

Entering parameter values

If we are providing input parameter values to be placed in the registers, it is pretty obvious that we have to ensure that we list them in a manner which causes no doubt as to which value is to go to which register.

Generally, this is simply achieved by listing the input values in register number order (R0 first, then R1 and so on). However, it is quite common for an SWI to use only a few registers, **not necessarily starting with** R0 and/or not necessarily consecutive. To cope with all variations unambiguously, commas are used to “acknowledge the presence” of any register not used but preceding a used register.

This is easier to show than to explain in words. So, imagine an SWI requiring input parameter values to be placed only in registers R2 and R4 - with a string and a number respectively. To be unambiguous, our SYS statement would therefore need to show the input parameter values as follows:

```
SYS "SWI_NameString" , , , "R2InputString" , ,
      R4InputInteger%
```

Effectively, what this shows is that each input parameter value needs to be regarded as being preceded by a comma i.e. <, Value> - and if the value is not given then the comma must still be used, to show that a value is missing. In our mythical example above, the first two commas show that values for R0 and R1 are missing; the third comma is the one which precedes "R2lnputstring"; the fourth comma shows that an R3 value is missing, and the fifth comma is the one preceding R4InputInteger%.

If the SWI is being used only to make an input (i.e. no output results to be returned) then the SYS statement simply ends after the sequence of input parameters.

Output variables (with the keyword TO)

If an SWI provides any output results, then the **same set of registers** (R0-R7) are used for the result values - which again may be integer numbers or strings. (Thus, any input values placed in the registers may well be overwritten by output values from the routine.)

The SYS statement helpfully allows these output results to be assigned directly to Basic variables. This is done by using the keyword to after the input parameter values (if any) and then listing the required ‘destination variable’ names of the right type, separated by commas, in the order corresponding to the registers. As before, commas are used if necessary to cope with registers whose value is not required for output.

Thus, in our earlier real example:

```
SYS "OS_ReadModeVariable" , -1 , 1 TO ,, ScreenWidth%
```

we have used TO followed by ,, Screenwidth% - because the PRM tells us that the output we want is an integer number and it is going to be

placed in register R2 in this case. Further, the PRM tells us that registers R0 and R1 will be left unaltered from their input values.

Hence, we are not interested in the output values in R0 and R1 but we have to 'acknowledge their presence' and this we do by putting a comma to represent each of them i.e. two commas in this case. We can then follow these commas with the variable name Screenwidth% and there will now be no doubt that this is our required destination for the R2 output result.

Note that there is an important difference in the use of commas between the input parameters and the output results. Assuming that both are present in the SYS statement, then there is always a comma in front of the **input** R0 value (i.e. after the SWI name/number) but **not** in front of the R0 output result variable (i.e. after the TO). If TO happens to be followed immediately by a comma it means that we have decided not to assign the R0 output result to a variable.

To demonstrate the point we can expand our previous mythical example to:

```
SYS "SWINameString" , , , "R2InputString" , ,
     R4InputInteger% TO R0OutputVariable%, ,
     R2OutputVariable%
```

when output results from R0 and R2 only are required (and assuming they are both integer numbers).

Or:

```
SYS "SWI_NameString" , , , "R2InputString" , ,
     R4InputInteger% TO , , R2OutputVariable%
```

if only the result from R2 is wanted.

Parameter blocks

Frequently in Wimp programs, we need to send/receive rather a lot of data to/from an SWI in a SYS call - far more than can be done with eight registers. For instance, when using the SWI for defining a window on a Wimp screen we need to pass the size, colours, heading text etc. - maybe 20-30 items. In such cases, great use is made of "parameter blocks".

A parameter block is simply a block of memory set aside by the programmer for the purpose of holding the required data; maybe a few bytes or maybe several 'pages' of memory.

The starting address of the data block is then passed as a parameter to the SWI - and the SWI automatically reads the data from the block

(previously placed there by the programmer) and the SWI often puts some or all of the output result data into the same block (this time for the programmer to extract and use).

It is the PRM, of course, which details which parameters need to be of this type and the minimum data block necessary and where each item of data needs to be placed within it.

To set aside a block of memory for this purpose we use the keyword dim in the way already covered in Chapter 20 when introducing the indirection operators. For example:

```
DIM BlockName% SizeOfBlock%
```

remembering that there must be a space between `BlockName%` and `SizeOfBlock%`.

Final points

By and large, with 18 pages of available SWIs, there are very few aspects of the OS that you cannot directly interface with the `SYS` keyword. It is probably worth noting that SWI names are very helpfully structured - as our earlier example demonstrated:

```
SYS "OS_ReadModeVariable"
```

The first part of the name - before the underscore character - gives the heading of the group of SWIs that this one belongs to - the ‘OS’ group in this case. There are ‘headings’ for printer drivers, sound, Wimp, fonts, colour etc. so it is easier to find what you are looking for.

In Wimp programming, even from the very start, an understanding of `SYS` calls is essential.

The above choice of `SYS "OS_ReadModeVariable"` *as a vehicle to introduce the subject was not arbitrary. If you cast your mind back to* `DEF PROCcentrePrint(String$)` *at Line 1240 of our Loan program, you will see that Line 1280 specified a screen-width of 80 characters - because we knew that we would be operating in a screen mode with that width. However, with this SYS call we could now make that* `PROC` *work correctly with any screen mode, automatically.*

If we change Line 1280 of our Loan program to:

```
1280 SYS "OS_ReadModeVariable" , -1 , 1 TO ,,
     ScreenWidth%
```

the `PROC` *would correctly centre a string in any screen mode. A similar change could be made to Updatel4a, where an identical issue arises.,*

24. The nature of Wimp programming

Wimp and non-Wimp compared - The Wimp - Interaction between application program and Wimp - Wimp Poll - Reason codes - Structure of Wimp program - Example of possible Window for Wimp version of Loan program - Wimp programming tools.

We are not going very far into Wimp programming - merely enough to set the scene and, hopefully, make your further reading more meaningful.

Comparison of non-Wimp and Wimp programs

As we have seen in this book, a non-Wimp program - however small - monopolises the computer whilst it is running. Even when we do not specify a screen mode change and the program appears to run within a Task Window on the desktop screen, we cannot ‘get out of that window until the program has ended and we are invited to `"Press SPACE or click mouse to continue"`.

In contrast, a Wimp program adds something to what is already on the desktop screen. It doesn’t prevent user interaction with other applications already present and - memory space permitting - it doesn’t stop even more applications from being added. This is the obvious first difference between non-Wimp and Wimp programs.

The second main difference is the way the user makes an input. A Wimp program puts pictorial symbols (icons) on screen and most input is carried out by placing the screen pointer on the icons (by moving the mouse) and clicking a mouse button. In non-Wimp programs, the keyboard is used much more. *(The mouse and pointer can be used in non-Wimp programs but not as easily as in Wimp programs.)* Further, when a non-Wimp program needs user input, the computer sits there idly waiting until the

input is made - and nothing else can happen until then.

A third main difference, although it is not yet evident, is that Wimp programs tend to need a lot more program lines to be written initially i.e. before a program can meaningfully be run and cause something to appear on the screen.

Program structure and the Acorn Window Manager ("the Wimp")

If we look at the structure of a non-Wimp program - like our Loan program, for instance - we will see that the programmer never 'releases the reins'.

Because the non-Wimp program monopolises the computer, the program proceeds line by line in exactly the way the programmer has structured it. There may be loops, branches and conditional statements etc. but they were put there by the programmer alone and you can usually follow the flow from start to finish in the listing without undue difficulty.

The fundamental change when we come to Wimp programs is that - in order to permit more than one program to be active on the screen simultaneously - the programmer has to release the reins for some of the time (or, more accurately, release some of the reins all the time).

Roughly what happens is that a manager program (called the Acorn Window Manager, or just "the Wimp"), within the operating system of the computer, is used to handle all the input and output to/from Wimp application programs.

Any Wimp application program is then structured to ask the Window Manager repeatedly "Anything happened lately, concerning my program?" For much of the time the answer will be "Nope". But if, for instance, the user makes a mouse click (an input action) over one of the program's icons on the screen, the Window Manager records that click and next time the question comes the Window Manager gives the details.

The program then takes corresponding action in whatever way it has been programmed to do - and then reverts to repeating its earlier question, until some further positive response occurs.

What the application program doesn't realise is that the Window Manager is being asked the same question repeatedly by every other active Wimp application program - and they all share the available time, usually without any apparent difficulty from the user's viewpoint.

In order to join in with this action, a Wimp application program needs to:

-log on with the Window Manager.

-keep asking ‘the question’.

-respond without delay to a positive answer.

-log off when the program exits.

Thus, the programmer still retains full control over what the program does - but not over precisely when it does it. Having said that, we are dealing with timings in milliseconds and the time-share aspects are usually invisible to the user.

The Window Manager more than compensates for imposing itself in this way by relieving the programmer of a host of boring tasks.

For instance, when the programmer wants a window or menu shown on the screen (or erased) he simply tells the Window manager to do it using a sys call, passing the details (e.g. size, screen location, heading text, etc.) in a parameter block.

The Wimp poll and reason codes

The particular control process used by the Window Manager is a polling mechanism, called the Wimp Poll - and the heart of all Wimp programs is usually just a fairly simple proc, normally placed inside a while . . . endwhile loop, which repeatedly ‘asks the question’ and temporarily branches the program along an appropriate path when a positive response is received.

Thus, the outline structure of a typical Wimp program is:

```
PROCinit :REM** Sets up DataBlock%, etc. and logs on to
     Window Manager **

WHILE (Quit% = FALSE)
     PROCpoll :REM** Calls PROCpoll repeatedly **
ENDWHILE

PROCexit: REM** Logs off from Window Manager **

END
```

```
DEF PROCpoll
SYS "Wimp_Poll" , 0 , DataBlock% TO Reason% :REM**
     'Asks the question' and designates variable and
     data block for answer **
CASE Reason! OF : REM** Deals with a range of possible
     different answers **
     WHEN 0 : PROCaction0
     WHEN 1 : PROC action1
     .
     .
     etc.
ENDCASE
ENDPROC
```

The heart of the action is the line:

```
SYS "Wimp_Poll" , 0 , DataBlock% TO Reason%
```

which 'asks the question' and puts the response into the variable `Reason%` (often also putting associated detailed output data into `DataBlock%`).

What is often initially surprising to the beginner is that the core of every Wimp program tends to look the same. The differences come in the `PROCaction` area.

We have used Reason% as the name of the variable to receive the answers because the Wimp Poll gives its answers by using "reason codes", which currently can take any of the values 0-19.

For instance, reason code 6 means that a mouse click has occurred - and the details (which window, which icon, which button, type of click etc.) are automatically put into `DataBlock%` by the Window Manager, for the programmer to retrieve if needed.

Similarly, reason code 9 means that an item from a Wimp menu has been chosen and, this time, `DataBlock%` will be used to say which item on which menu etc.

So, the Wimp programmer's main task is to develop the responses to the various reason codes - and even with this brief description you can see that frequent reading and writing to `DataBlock%` is going to be necessary.

For example, if the programmer wants the above choice of a menu item to cause a window to open on the screen then the response to reason code

9 would need to load a parameter block with the details of the window and tell the Window Manager to open it.

Any further action by the user on this new window would, of course, be the subject of another question and answer session with the Window Manager via the repeated call to the Wimp Poll.

Sometimes the Window Manager needs to prod the program to take some action, as a result of a user action not apparently directly associated with your program. For instance, if you drag an open window of your program to a different part of the screen or resize it, the Window Manager will start issuing reason code 2 - meaning that you need to take action to open that window again. (Moving an object is essentially a repeated deletion and redraw.)

So, nearly every Wimp program needs, as a bare minimum, a routine to respond to reason code 2, otherwise the window simply will not drag. The fact that opening a window means telling the Window Manager to open it should not exasperate you (!) because you do have to ensure that the right window is opened by loading up the data block correctly first.

Similarly, if you click on the window ‘close’ icon the Window Manager will issue reason code 3, which asks you to take action to close the window - which again means telling the Window Manager to do it after you have identified the window correctly in a data block.

A flow diagram of a Wimp program therefore tends to comprise multiple main loops, all entering and leaving the Wimp Poll and each representing a particular response to a reason code.

Descriptively, the program runs in a repeated series of long pauses followed by short activity when a positive reason code eventually comes - where ‘long’, ‘short’ and ‘eventually’ need to be looked at in terms of the computer’s pace of life.

Wimp version of Loan program?

Production of a Wimp version of our earlier Loan program is well beyond the scope of this book, but it is interesting to compare our non- Wimp version with a possible main window of a Wimp version - and such a possibility is shown in Figure 24.1.

The four boxes at the right-hand side of the window would be “writable icons”, where the known values of three of the items would be entered by the user. After entries had been made in any three of these boxes, the user would be prevented from making an entry in the fourth box - and the “Calculate” box would ‘brighten up’ and wait for the user to press `<select>` with the pointer over it.

The numerical answer would then appear (in a different colour) in the fourth box.

When the No. of Payments or the Interest Rate are the unknown factors a second window would open to show the appropriate graphs.

The Wimp version could use substantially the same calculation and graph routines as our non-Wimp version - but now linked to the mouse keypresses via the reason code responses of the Wimp Poll routine.

Figure 24.1

Possible main window of a Wimp version of Loan program.

‘Shell packages’ and similar Wimp programming tools

Because many Wimp programs have identical features centred on the Wimp Poll, it becomes perfectly feasible to produce a skeleton Wimp program which will serve most of them with relatively minor modification.

Thus, packages are available to do just this - sometimes called ‘shell’ programs - and maybe with a library of associated PROC/FNs and other useful ‘tools’.

These do not remove the need for the programmer to devise particular routines specific to each program, but it can relieve him/her from much of the repeated ‘drudgery’ of the skeleton. *Naturally, you do not get something*

for nothing - the resulting programs are likely to need more memory than if you tackled each one from scratch. "You pays your money - you takes your choice."

Further reading

There is one very good book introducing Wimp programming from scratch - it assumes you are competent in Basic but nothing more. It is "Wimp Programming for All" by Lee Calcraft and Alan Wrigley, published by Beebug, St Albans.

It uses the same tutorial method as used in this book i.e. a continuously developing program through several chapters - with a disc available.

Appendix 1. ‘Current Directory’

If you have created a new file by typing `SAVE` in a Task Window as in Chapter 1 (or, in other cases, without specifying a full path specification in a ‘Save box’) you may not be able to find the file on your desktop screen - even after moving the windows around and/or opening other directories where you think it might be. Don’t panic!

Unless told specifically, the computer will always put such files into the ‘Current Directory’ (sometimes called the ‘Currently Selected Directory’) and the User Guide explains what this is.

Therefore, if you can’t find the file, it almost certainly means that the Current Directory is not open on the desktop screen and/or it is not what you think it is. For instance, the Current Directory may still be set for the last application you were using, or an application triggered by your particular start-up configuration.

So, firstly, open a few of the likely directories to have a look.

If you have no luck, you will have to try the longer route:

open the root directory of your hard disc by clicking `<select>` over the hard disc icon on the iconbar;

click `<menu>` in the root directory window just opened (but not over any of its icons) - and `<select>` `"Select all"` from the menu items that appear. This will reverse the colours on all the icons in the root directory, showing that all are now selected;

click `<menu>` again in the root directory window and then move

the pointer over `"Selection"` and off to the right to open another menu and

click `<select>` on `"Find"`. This will open a writable icon in which you can type the name of the file you are looking for (e.g. `'Progla'`) followed by <return>;

then sit back and wait. It may take a minute or so, but the computer will go through all your hard disc directories in turn until it finds the file. It will then report with a message giving you the full path of the file's location, plus the option to open the corresponding directory for you;

`<select>` that option and you will then see the file. (Remember to use the scroll bars if need be!)

This is the time to move (i.e. `<shift> + drag`) the file to somewhere more convenient if you like.

The above takes much longer to read than to do! It emphasizes the importance of the Current Directory and may prompt you to re-read the User Guide on that subject.

Appendix 2. Variable names

There are some restrictions on what can be used for variable names:

- no spaces allowed within the name.
- no punctuation marks or arithmetic operator signs allowed. In fact, any symbol which is used in Basic for some special purpose must not appear in a variable name e.g. # or ! When % and $ are used at the end of a variable name to designate its type, they are not regarded as part of the name - but they must not be used elsewhere in the name.
- must not start with a numerical digit - but digits can be used after the start.
- must not start with any Basic keyword - for obvious reasons. But don't forget that all keywords are in upper case, so their lower case counterparts (or a mixture) are OK.

The rules for `PROC/FN` names are less restrictive because they can start with a digit or a keyword - and the symbol @ can also be used.

(Filenames have their own separate rules - see Use Guide. They are not controlled by Basic.)

Appendix 3. TRUE and FALSE

The meanings of TRUE and FALSE in Basic are quite precise and it is unsatisfactory to rely upon the normal English usage of the words.

It is best to explain the meaning in small steps. *(Remember, from Chapters 6 and 7 that most of Basic's constructions/loops have 'condition statements'. These true/false rules will apply to all of them.)*

If you type the following as a short Basic program and run it:

```
Marker% - TRUE
Flag % = FALSE
PRINT Marker%
PRINT Flag%
```

you will get -1 and 0 as the results.

This tells us that BBC Basic uses the numbers -1 and 0 to represent `TRUE` and `FALSE` respectively. This is an important piece of knowledge, so embed it in your memory!

Now let's go a little further by typing and running the following new program. *(The condition statements are in brackets to emphasize them.)*

```
Marker% = 6 - 7
Flag% = 8 - 8
IF (Marker% = -1) THEN PRINT "1st Condition is True"
IF (Flag% = 0) THEN PRINT "2nd Condition is True"
```

When run, this will print both messages; showing, as you would expect here, that the 1st and 2nd condition statements both “evaluate to `TRUE`” (“are true” in normal English), because we have deliberately made the numbers accord with the required condition.

Now add the following two lines to the end of the program and run it again:

```
IF (Marker%) THEN PRINT "3rd Condition is True also"
IF (Flag%) THEN PRINT "4th Condition is True also"
```

Now, the 1st, 2nd and 3rd messages will be printed - but not the 4th.

So, the evaluation (`Marker%`), on its own, is `TRUE` - because `Marker%` holds the value -1.

However, whereas the evaluation of (`Flag% = 0`) is `TRUE` (because `Flag%` does equal zero!), the evaluation of (`Flag%`), on its own, is `FALSE` - because `Flag%` holds the value 0.

It makes sense, but you might have to go through it a couple of times to ensure you follow it properly.

Now, change the last line to:

```
IF (NOT Flag%) THEN PRINT "4th Condition is True also"
```

and re-run. This time the 4th message is added to the output, because `NOT` changes `TRUE` into `FALSE` and vice versa. *(`NOT` is one of the operators covered in Appendix 5.)*

So far so good. But now, with all the above changes still in place, change the first line to:

```
Marker% = 9 - 6
```
(or any sum that does not give 0 or -1 as a result)

and you will now get 2nd, 3rd, and 4th messages. The surprise result is that the 3rd message still appears. What does this tell us?

It indicates that Basic regards **any non-zero value, on its own** (i.e. as the complete condition statement), to be `TRUE`.

Check this out by adding the lines:

```
Number% = 77
IF (Number%) THEN PRINT "5th Condition is True too!"
```

and the 5th message will be added to the output.

To convince yourself thoroughly, try it with `Number%` set to -77, 0 and -1 in turn. Only when the value is 0 will the 5th message not appear.

Because the above points can be a little difficult to absorb straight away, beginners are advised to:

(a) try to use full condition statements wherever you can. It is easier to understand and less likely to produce surprises e.g. use:

```
IF (Marker% = TRUE) THEN ....
```

rather than:

```
IF (Marker%) THEN .....
```

(b) always put the brackets round the condition statement to help you remember what is happening.

Appendix 4. ASCII codes

&0 NUL 0	&10 DLE 16	&20 space (SP) 32	&30 0 48	&40 @ 64	&50 P 80	&60 ` 96	&70 p 112
&1 SOH 1	&11 DC1 17	&21 ! 33	&31 1 49	&41 A 65	&51 Q 81	&61 a 97	&71 q 113
&2 STX 2	&12 DC2 18	&22 " 34	&32 2 50	&42 B 66	&52 R 82	&62 b 98	&72 r 114
&3 ETX 3	&13 DC3 19	&23 # 35	&33 3 51	&43 C 67	&53 S 83	&63 c 99	&73 s 115
&4 EOT 4	&14 DC4 20	&24 $ 36	&34 4 52	&44 D 68	&54 T 84	&64 d 100	&74 t 116
&5 ENQ 5	&15 NAK 21	&25 % 37	&35 5 53	&45 E 69	&55 U 85	&65 e 101	&75 u 117
&6 ACK 6	&16 SYN 22	&26 & 38	&36 6 54	&46 F 70	&56 V 86	&66 f 102	&76 v 118
&7 BEL 7	&17 ETB 23	&27 ' 39	&37 7 55	&47 G 71	&57 W 87	&67 g 103	&77 w 119
&8 BS 8	&18 CAN 24	&28 (40	&38 8 56	&48 H 72	&58 X 88	&68 h 104	&78 x 120
&9 HT 9	&19 EM 25	&29) 41	&39 9 57	&49 I 73	&59 Y 89	&69 i 105	&79 y 121
&A LF 10	&1A SUB 26	&2A * 42	&3A : 58	&4A J 74	&5A Z 90	&6A j 106	&7A z 122
&B VT 11	&1B ESC 27	&2B + 43	&3B ; 59	&4B K 75	&5B [91	&6B k 107	&7B { 123
&C FF 12	&1C FS 28	&2C , 44	&3C < 60	&4C L 76	&5C \ 92	&6C l 108	&7C | 124
&D CR 13	&1D GS 29	&2D - 45	&3D = 61	&4D M 77	&5D] 93	&6D m 109	&7D } 125
&E SO 14	&1E RS 30	&2E . 46	&3E > 62	&4E N 78	&5E ^ 94	&6E n 110	&7E ~ 126
&F SI 15	&1F US 31	&2F / 47	&3F ? 63	&4F O 79	&5F _ 95	&6F o 111	&7F DEL 127

Appendix 5. Hex and Binary numbers

Our normal method of counting numbers is based on the ten different digits 0-9 and is called a **decimal** system. Thus we get the familiar pattern:

0 1 2 3 4 5 6 7 8 9

10 11 12 13 14 15 16 17 18 19

etc.

There is nothing sacrosanct about the decimal system. It happens to suit us with our 10 fingers, but it is not convenient for all purposes - and there is no reason why numbering systems cannot be based on other numbers of digits.

Binary

In electronic circuits there are many devices which can only have two, rather than ten, different states. The ‘On’ or ‘Off of a switch is perhaps the most obvious example - but computer memory is another example and is the one that concerns us most here. In these cases, using a numbering system with only two digits - a **binary** system - is more appropriate.

So, using (for convenience only) the same two symbols as we use in our decimal system for the first two digits i.e. 0 and 1, we get the pattern:

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

for the binary equivalent of the decimal numbers 0-16.

The phrase 'Binary digiTS' is normally contracted to the word 'bits'.

The particular pattern above for the binary numbers does not show all the important relationships very clearly, but by examination you should be able to deduce the following rules:

- the binary numbers 1, 10, 100, 1000, 10000, etc. represent 2^0, 2^1, 2^2, 2^3, 2^4 etc. i.e. decimal 1, 2, 4, 8, 16 etc. or 'the number of zeros equals the power of 2' (N.B. any number raised to the power 0 equals I)
- moving all the bits to the left by one bit (and adding an extra 0 to the right-hand end) doubles the number, e.g. binary 101 (decimal 5) shifted left by one bit becomes 1010 (decimal 10).
- If the original binary number has a 0 at its right-hand end, then shifting everything to the right by one position (and dropping off that original right-hand 0) will halve the number, e.g. binary 1100 (decimal 12) becomes binary 110 (decimal 6). If you do the same with a binary number with a 1 at its right-hand end, the answer will be 'the integer part of half the number'. A right-shift therefore gives the same result as `<number> DIV 2` in Basic.
- In binary arithmetic:

  Adding: 1 + 1 = 10 (1 + 1=2 in decimal)

  "1 + 1 equals 0 and carry 1 to next column. In next column, 1+0 equals 1. Hence answer is 10."

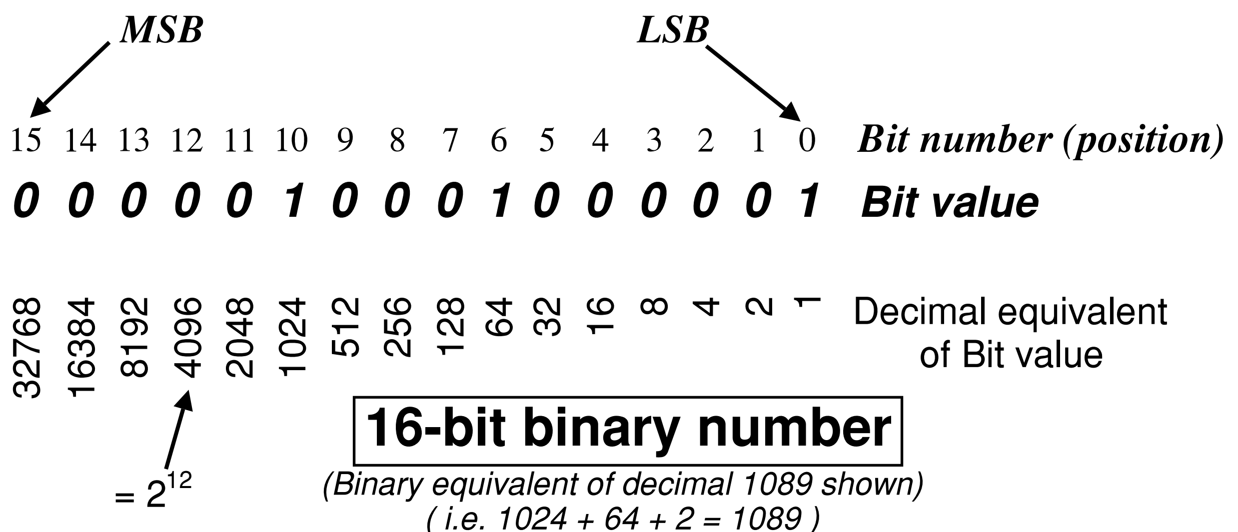
Subtracting: 100 - 11 = 001 (4-3=1 in decimal)
"0-1 can't be done, so 'borrow 1' from next column, to get 10 - 1 = 1. In second column, 'pay back 1'. As bottom line of second column is already 1, paying back changes it to 0 and carry 1 to third column. Then, in second column 1-0=0. In third column, 1-1=0. Hence answer is 1."

Bit positions

In BBC Basic, we often need to look at the contents of specific bits within a binary number. For this purpose the bit positions are invariably numbered (in decimal!) from the right and starting with zero.

The right-hand bit (i.e. 'bit0') of a number is also called the Least Significant Bit (LSB) - because it has the least effect on the size of the number represented by that group of bits. Similarly, the extreme left-hand bit is called the Most Significant Bit (MSB).

This diagram of a 16-bit binary number will help.



You have to be a little careful about the MSB. because the computer is likely to be working with a fixed number of bits (e.g. 16 or 32 bits) whatever the number being represented, whereas we humans tend not to remember all the 'leading zeros' which exist to the left of the highest 'I' in a particular binary number. For example, referring to the above

picture, we would probably write the binary number as 10001000001 - and then easily regard bit 10 as the MSB. But a 16-bit computer will be recording the number with all the leading zeros (as in the picture) with bit 15 as the MSB.

The diagram also shows how to work out the decimal equivalent of a binary number i.e. check which bit positions contain 1, then add up the corresponding powers of 2:

$2^{10} + 2^6 + 2^0$ in the above, or 1024 + 64 + 1 = 1089

You will get used to handling at least bits 0-7 fairly easily - and that might well be enough for most purposes, for reasons which will soon become clear.

Finally, a few small points:

- it is common programming parlance to say that a bit is ‘set’ when its value is 1, and ‘unset’ when 0.
- a group of 8-bits is frequently called a ‘byte’. So the picture above shows a ‘2-byte’ number. Note that a byte can therefore store any number from 0-255 (Binary 00000000 to 11111111)
- BBC Basic allows us to input binary numbers by preceding them with % e.g. instead of the statement `Integer% = 27`, we could write `Integer% = %11011`. Unfortunately, there is no corresponding built-in means of getting Basic to give you an output in binary form. (However, a `FN` to simulate this is listed at the end of this Appendix.)
- the !SciCalc application supplied with most Acorn computers allows conversion between decimal and binary numbers (and other bases also). There is also a very convenient calculator included with !DeskEdit, if you use that for program editing.

Other numbering bases

The above shows how a binary (‘base 2’) numbering system is constructed and we could go through a similar process for any other base. In practice, you are only likely to run across one more - or maybe two. Hexadecimal (base 16) is fairly common in computing - and very occasionally you might meet Octal (base 8). *Both are included in the !SciCalc application by the way.* We will only cover Hexadecimal (‘hex’) here.

Hexadecimal numbering ('hex')

The first thing we have to do if we want a numbering system to base 16 is to decide what symbols we will use to represent the 16 digits.

It is convenient to use the same symbols as our decimal system for the digits 0-9 and conventionally we use the letters A, B, C, D, E and F for digits 10-15 - usually in upper case, but not always.

So, the hex pattern is:

0 1 2 3 4 5 6 7 8 9 A B C D E F *(decimal 0-15)*
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F *(decimal 16-31)*
20 21 ...... etc.
etc.

90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F *(decimal 144-159)*
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF *(decimal 160-175)*
B0 B1 ...... etc.
etc.
.... FD FE FF *(... to decimal 255)*
100 101 102 103 104 ...... etc. *(decimal 256 onwards) .....)*

It does take a bit of getting used to but it definitely is more convenient to use than decimal for many programming purposes.

To distinguish hex numbers from decimal numbers, it is normal to precede a hex number with the '&' symbol - and there could be real confusion without this: i.e. does 19 mean 'decimal 19' - or 'hex 19' (decimal 25)? Writing &19 makes it clear.

In BBC Basic we can input hex numbers instead of decimal by using the '&' symbol - and we can get a hex output by using the '~' symbol (the 'tilde'). See Chapter 10 for how to do this.

Binary/Hex link

Because $16 = 2^4$ there is a very strong link between binary and hex numbers. Have a look at the following equivalents:

decimal	1099
binary	%0000 0100 0100 1011
hex	&044B

(The binary number has deliberately been separated into groups of 4 bits - and leading zeros have been included in both the binary and hex numbers.)

Now, take each 4-bit section of the binary number separately and work out their values in hex. You ought to get 0, 4, 4 and B reading from left to right - which conforms with the hex equivalent.

In effect, a single hex digit represents 4 binary digits (4-bits) - which is perhaps not too much of a surprise when $16 = 2^4$.

Note also that, using the same principle as we did for binary numbers earlier, the hex digits represent (in turn and starting from the right) the number of times that 16^0, 16^1, 16^2, 16^3 etc. are needed to get the result - or 1, 16, 256, 4096, etc. Let's check it by calculating the decimal equivalent of &044B:

&044B	=	(0 x 163) + (4 x 162) + (4 x 161) + (&B x 16°)
	=	(0 x 4096) + (4 x 256) + (4 x 16) + (&B x 1)
	=	(0 + 1024 + 64 +11)
	=	1099

Basic Program with FN to produce output in Binary

Program 'BinaryOut'

```
10 REM> BinaryOut
20 REM** A FN to produce the 32-bit binary equivalent
     (in string form) of an integer number. **
30
40 ON ERROR REPORT:PRINT " at Line ";ERL:END
50
60 REM** The next few lines are used to demonstrate the
     FN and to print the resulting 32-bit binary
     number as eight separated groups of four bits,
     for better visibility. (The FN itself produces
     the result without the spaces. **
70 REM** Change DATA line to change numbers
demonstrated - ending with zero (which will be
     converted also) **
80
90 REPEAT
100  READ number%
110
```

```
120   temp$ = FNbinary(number%)
130   PRINT TAB(1) "%" ;
140
150   FOR group% = 1 TO 32 STEP 4
160       PRINT MID$(temp$ , group% , 4) + " "
170   NEXT
180
190   PRINT " " + STR$(number%)
200   PRINT
210
220 UNTIL number% = 0
230
24 0 END
250
260 DATA 1, 8, 30, 64000,&7FFFFFFF, -2147483647, -3,0
270
280 REM****************************************
290 REM****************************************
300
310 DEF FNbinary(N%)
320 REM** Returns a string of the binary number
     conversion of parameter N% **
330 REM** THE LARGEST VALUE OF N% ALLOWED (+ or -) IS
     2,147,483,647 (&7FFFFFFF)
340
350 WordLength% = 32 :REM** No. of bits in result, DO
     NOT ALTER **
360
370 negative% = FALSE
380
390 REM** Next line converts negative numbers correctly
     before processing **
400 IF N% < 0 THEN negative% = TRUE : N% = &80000000 +
     N%
410
420 skip% = FALSE
430 Binary$ = " "
440
450 PROCrecur
460
```

```
470 REM** Leading zeros are now added to the resulting
     Binary$ and then only the required number (0-31)
     of binary digits are extracted from the
     right-hand end of the extended Binary$. **
480 Binary$ = RIGHT$(STRING$(WordLength% , "0") +
     Binary$ , WordLength%)
490
500 REM** Finally, substitute a 1 at bit 31 if N% is
     negative.
510 IF negative% THEN Binary$ = "1" + RIGHT$(Binary$ ,
     WordLength%-1)
520
530 = Binary$
540
550 REM** * ************************************
560
570 DEF PROCrecur
580 REM** Recursive PROC i.e. calls itself **
590
600 REM** Firstly, N% is tested for odd/even and 1/0
placed in binary output correspondingly **
610 IF N% MOD 2 = 1 THEN Binary$ = "1" + Binary$ ELSE
     Binary$ = "0" + Binary$
620
630 REM** N% is then halved. As result is assigned to
     Integer Variable (N%) only integer part of
     division will remain (which may be 0). Equivalent
     to binary shift one bit to right (and discarding
     bit that 'falls off RH end')
640 N% = N% / 2
650
660 REM** If remainder is 1 or more then sequence is
     repeated by recursive call - thus adding binary
     digits to LH end of Binary$ each time. This
     continues until integer remainder is zero.
670 IF N% >= 1 THEN PROCrecur
680
690 END
```

Appendix 6. The operators AND, OR, EOR, NOT, <<, >>, and >>>

Logical ('bit-wise') operators

BBC Basic has four 'logical operators' which can be used on integer numbers in a similar fashion to the arithmetic operators - see Chapter 11 - but with very much different results.

Their difference is they act separately on each bit of the binary version of the integer numbers *(and if you enter a real number, they will act only on the integer part of it)*.

The rules are easy to state:

AND needs two numbers and produces a single new number from them *(it must not be confused with '+' sign, which gives a totally different result)*. `AND` compares the two numbers bit by bit ('bit-wise'). If both bits in a particular position are 1 then the bit in the same position in the new number is 1 - otherwise it is 0.

OR needs two numbers and produces a single new number from them. It compares the two numbers bit by bit ('bit-wise'). If either bit (or both bits) in a particular position is 1 then the bit in the same position in the new number is 1 - otherwise it is 0.

EOR ('Exclusive `OR`') needs two numbers and produces a single new number from them. It compares the two numbers bit by bit ('bit-wise'). If either bit (**but not both**) in a particular position is 1 then the bit in the same position in the new

number is 1 - otherwise it is 0.

NOT acts on a single number and produces a new number by simply changing every binary bit. Every 0 is changed to a 1, and every 1 to a 0. This is called 'inverting' the number - and is **not** the same as the 'two's complement' - nor the same as 'putting a minus sign in front of it'.

The first three operators are fairly straight-forward and here are some examples:

AND

`3 AND 5` produces the result `1`.

```
3   %011
5   %101
________
1   %001
```

Only in the LSB ('bit 0') is there a 1 in the binary versions of both numbers 3 and 5 - so the result is `&001`. More closely, note that a 1 in any position on the bottom line of an `AND` operation will leave the bit in the same position in the top line unchanged - and a 0 in the bottom line produces a 0 in the same position in the result, whatever the bit in the top line.

OR

`17 OR 11` produces the result `27`.

```
17  %10001
11  %01011
__________
27  %11011
```

Bit 2 is the only one which does not have a 1 in either (or both) positions, so the result is `%11011` which is decimal 27. More closely, note that a 0 in any position on the bottom line of an or operation leaves the bit in the same position in the top line unchanged - and a 1 in the bottom line produces a 1 in the same position in the result, whatever the bit in the top line.

EOR

`17 EOR 11` produces the result `26`.

```
17  %10001
11  %01011
__________
26  %11010
```

This is similar to the previous example except that the LSB (bit 0) is 1 in both numbers, so the result is %11010, which is decimal 26. More closely, note that a 0 in any position on the bottom line of an `EOR` operation leaves the bit in the same position in the top line unchanged - and a 1 in the bottom line inverts the bit in the same position in the result.

NOT

As not inverts all 32 bits (0-31) the numerical results are somewhat artificial because they are very much affected by the ‘two’s complement’ method of storing integer numbers (see Appendix 8). For example:

`NOT 3` produces the result `-4`

and, in fact:

`NOT <number >` produces the result `-(<number> + 1)`

More importantly, if you `NOT TRUE` you get `FALSE` - and vice versa - because they are represented by -1 and 0 respectively (see Appendix 3).

In fact, you are not likely to need `NOT` very much, except to reverse `TRUE` and `FALSE`.

Check the above examples in Basic Immediate Mode in a Task Window, by typing:

```
PRINT 17 OR 11
```

etc.

Bit manipulation and ‘masking’

It is probably obvious that the above operators are not a great deal of use for number manipulation. They come into their own for bit manipulation. In particular, they invaluable for extracting (or changing) parts of a multi-byte ‘word’ or just parts of single bytes - and this is often necessary - particularly in the Wimp environment.

In Chapter 6 an example of this was given. There, if you recall, we wanted to ensure that any keyboard letter was returned as its upper case version - irrespective of the fact that either upper or lower case might have been generated by the keyboard itself. To do this we 'ANDed' the ASCII number of the keyboard press with 223 - and hey presto!

The binary equivalent of 223 is %ll0111ll i.e. all bits except bit 5 are set. So, whatever number is put in the expression:

```
<number> AND 223
```

will produce a result with bit 5 definitely 'unset' and the others unchanged. Referring to the ASCII Chart at Appendix 4, this means that the ASCII number for an upper case keypress (all in the range 65-90) will stay the same, but a lower case keypress (all in the range 97-122 i.e. 32 higher than upper case, by design) will have its bit 5 value changed from 1 to 0 and thus become its upper case equivalent (32 lower) i.e we have placed a 'mask' or 'sieve' over the binary number to change it.

There are many examples where similar patterns have been designed-in to a set of binary numbers and where, therefore, bit-wise manipulation is helpful. Printer 'Escape Codes' are a particularly common case.

Finally, you will find that we often only want to know the value of the number represented by, say, the 8 least significant bits of a 16-bit or 32-bit number. This can be extracted with:

<number > AND &FF (remembering that &FF is %llllllll)

This operation will 'let through' (unaltered) bits 0-7 of <number>, but will change the value of all higher bits to 0. Thus, the result is a number equal to the contents of bits 0-7 of <number >. *(Compare this with the action of* MOD &100 *- see Chapters 11 and 20.)*

The bit shift operators ('<<',' >>' and '>>>')

Basic provides three arithmetic operators specifically designed to make binary bit manipulation easier. They operate on 32-bit integer numbers and are best described with examples:

Number % = 10 << 1 (or Number% = %1010 << 1 in binary)

will result in Number% being assigned with 20 (%10100 binary).

```
Number% = 10 << 3
```

will result in Number% being assigned with 80 (% 1010000 binary).

Thus, the `<<` ("left-shift") operator shifts the values of all the bits in the preceding number (here, 10 (decimal) or %1010 (binary)) to the **left**, by the number of times specified by the following number (1 and 3 respectively here) - and a corresponding number of zeros are added to the right-hand end afterwards.

In decimal, the effect of each left shift is to double the original number, as the examples show.

This operator is often used in Wimp programs to place particular values accurately into multi-bit parameters.

The "right-shift" operators `>>` and `>>>` both work in a similar fashion to the above but shifting the bit values to the **right** - 'dropping off the original right-most value.

Shifting to the right means that all 32 bits are always involved - and because of the particular way Basic stores integer numbers (see Appendix 8), which uses the left-most bit ('bit 31') to indicate positive or negative, two operators are needed to provide a choice about how to deal with 'bit 31' after the right-shift has taken place i.e. should the left-most bit be made a 0 or 1?

So, `>>>` shifts all 32 bit values to the right and introduces zeros at the left-most bit position - whereas `>>` shifts all bit values to the right and then makes 'bit 31' the same value as originally existed there. For these reasons, `>>>` is called the "unsigned right-shift" and `>>` the "signed right-shift" (or "sign extended right-shift").

Thus, `>>` will retain the original positive or negative status of a number and can therefore be used to carry out arithmetic division by two (a right-shift of values then being the equivalent of `Number% DIV 2` - as the right-most value is 'dropped off).

Accordingly, `>>>` is more often used to manipulate bit values in multi-bit parameters i.e. where the 32-bit 'number' is being used as a small 'parameter block' rather than a normal number.

Appendix 7. Declaring string variables

In program listings (a prime source of information for beginners) you may see a sequence like this:

```
String$ = STRING$( 20 , " " )
String$ = " "
```

and wonder why the programmer has declared the string to be 20 characters long and then immediately changed it to a shorter string, or even a null string (as shown here). *A similar sequence might well be seen after dimensioning a string array - and seem particularly pointless bearing in mind that string array elements are automatically initialised to null strings!*

If you are using a string variable whose contents are going to vary through the course of the program run, then it is good practice to make an initial declaration of that variable with a string of the **maximum length that the variable is likely to need during the program run**. The reason, oddly enough, is to economize in the use of the memory space needed to run the program.

If a Basic variable is storing a string of a certain length and it then is called upon to assign a shorter string, there is no problem - the new string is stored within the same space as the original string. However, if the new string is longer, the Basic interpreter will allocate a completely new storage space for the longer string - **retaining, but not using, the previous space occupied.**

So, for example, if the initial string was 6 characters long and the new string is 7, then a total of 13 spaces (plus two lots of 'storage overhead')

will now be taken up. Do this a few times and a lot of waste space will be generated.

You can now see the reason for the opening sequence: the programmer wants `String$` to start as a null string, but he knows it is going to meet other values, up to 20 characters long, during the run. The sequence therefore assigns a dummy string 20 characters long at the start - before re-setting it to a null string - thus ensuring that only 20 memory spaces (plus one ‘overhead’) are used.

If the programmer gets it wrong and a string of length greater than 20 arises, then the program will start to waste space - but almost certainly not as much as would have occurred without the opening sequence.

This practice is particularly important when string arrays are declared - because of the larger amounts of memory they can need.

Appendix 8. Memory and the storage of numbers

The ‘random access memory’ (RAM - see WG) of the computer is used, inter alia, to store applications and other programs and their associated data. To use the RAM effectively, it is necessary to have the means to store data at specific memory locations and/or to read data from specific locations. Hence, a memory addressing system is needed.

Memory addresses and bytes

It is normal for memory addresses and the size of a computer memory (or portions of memory) to be measured in bytes (see Appendix 5) and for memory addresses to be quoted in hex numbers (also Appendix 5).

Thus, the lowest RAM address is &00 and the highest depends on the size of your RAM. For instance, the RAM address typically allocated for the start of the memory space to hold a Basic program is &8F00 (decimal 36608) - and the end of this same memory space is typically &A8000 (decimal 688128) i.e. a memory space of size &A8000-&8F00 = &9F100 bytes (or 651520 bytes, in decimal).

Thus, one byte is the smallest addressable memory location in Basic and each byte (being eight bits) can store numbers from 0 to &FF (0-255 decimal).

‘Pages’

A chunk of memory 256 bytes long with a starting address at a memory location having its least significant byte equal to zero (i.e. having 00 as its right hand end in hex form) is often called a page of memory. For example, the addresses &100, &A00, &2300 etc. are all addresses of

'page' starts - and the values &8E00, &8F00, &9000, &9100 (for example) mark the start of four **consecutive** memory 'pages'.

(Do not confuse this with the keyword PAGE, which is a pseudo-variable holding the start address of a Basic program.)

'Words'

As was mentioned in Appendix 5, some operations in computers are carried out on fixed-length chunks of memory e.g. 16-bit or 32-bit chunks.

In these cases the chunks are often referred to as ' 16-bit words' or '32-bit words'. This area is computer-specific, so the phrase definitely needs to be read in the specific context - and you can sometimes meet more than one 'word length' used in one computer, for different purposes.

In Acorn RISC OS computers, a 'word' is 32 bits (4-bytes) long - and, moreover, the chunk of memory in which a 'word' is stored must have a starting address which is divisible by 4 i.e. addresses whose least significant **byte** is &x0, &x4, &x8 or &xC only. Such addresses are called 'word aligned'.

How Integer and Real Numbers are stored

Integers

For storing integer numbers - positive or negative - BBC Basic uses 32-bit binary numbers in a special way. Bits 0-30 (31 bits) are used for the number value and bit 31 is used to show positive or negative - bit 31 is 'set' in negative numbers and 'unset' in positive numbers.

Bits 0-31 are used in normal binary fashion for positive integers; but negative integers are stored in 'two's complement' format. This is best explained by considering any pair of +/- numbers such as:

+6 and -6

If we add these together arithmetically we get zero - so we have to ensure that the binary representation of these numbers also produces zero. In 32-bit binary, +6 will be:

0000 0000 0000 0000 0000 0000 0000 0110

So, -6 must be:

1111 1111 1111 1111 1111 1111 1111 1010

(Starting from the right, adding these two numbers together binary-wise produces a 0 in bit positions 0-31, with the 'carry 1' generated at bit 1 rippling all the way through to bit 32 - and thus 'drops off the end' - leaving 0 at all locations.)

Thus, if you add a number to its 'two's complement', the answer is always zero.

The highest number that can be represented in this way is therefore:

2^{31} - 1 i.e. 2,147,483,647

(or &7FFFFFFF)

(or %0111 1111 1111 1111 1111 1111 1111 1111)

and the lowest is:

-2,147,483,647

(or %1000 0000 0000 0000 0000 0000 0000 0001)

Reals

For real numbers, Basic V uses 5 bytes (40 bits) and Basic VI uses 8 bytes (64 bits) to store numbers in a special way which we do not need to know at this stage. The range of numbers possible is:

$\pm 1.7 \times 10^{38}$ to $\pm 1.5 \times 10^{-39}$ (Basic V)

$\pm 1.7 \times 10^{308}$ to $+1.5 \times 10^{-323}$ (Basic VI)

Appendix 9. VDU ‘control codes’

Chapter 16 explained that the VDU codes 0-31 (which correspond with the ‘non printing’ ASCII codes) are called the VDU control codes.

The following table gives their meanings:

VDU Code	<ctrl> +	Extra bytes needed	Meaning
0	2 (or @)	0	Does nothing
1	A	1	Sends next character to printer only *
2	B	0	‘Enables’ printer *
3	C	0	‘Disables’ printer *
4	D	0	Text written at text cursor †
5	E	0	Text written at graphics cursor †
6	F	0	‘Enables’ VDU driver *
7	G	0	Makes a ‘beep’ sound *
8	H	0	Moves cursor back one character
9	I	0	Moves cursor forwards one character
10	J	0	Moves cursor down one line
11	K	0	Moves cursor up one line

12	L	0	Clears text viewport
13	M	0	Moves cursor to start of current line
14	N	0	Turns on ‘Page mode’ *
15	O	0	Turns off ‘Page mode’ *
16	P	0	Clears graphics viewport
17	Q	1	Defines text colour ‡
18	R	2	Defines graphics colour ‡
19	S	5	Defines logical colour ‡
20	T	0	Restores default logical colours ‡
21	U	0	‘Disables’ VDU driver *
22	V	1	Selects screen mode
23	W	9	*Multi-purpose command* *
24	X	8	Defines graphics viewport †
25	Y	5	Equivalent to plot command
26	Z	0	Restores default viewports
27	[	0	Does nothing
28	\	4	Defines text viewport †
29	]	4	Defines graphics origin
30	6 (or ^)	0	Moves text cursor to its ‘home’ position †
31	-(or _)	2	Moves text cursor to specified position

† More detail is given in Chapter 14

* More detail is given in Chapter 16

‡ More detail is given in Chapter 21

Appendix 10. Operations on whole arrays

As indicated in Chapter 18, there are several operations which use a reference of the form `ArrayName()` to refer to an array as a whole. Notes on these are given below.

The keyword DIM as a function

From Basic in a Task Window, type the following as a program or in Immediate Mode:

```
DIM Array( 2 , 3 , 3 , 2)
NumberOfDimensions% = DIM( Array() )
SizeOfDimension% = DIM( Array() , NumberOfDimensions%)
PRINT NumberOfDimensions% , SizeOfDimension%
```

The first line uses `DIM` in the normal way to declare/dimension a new array - here of four dimensions.

The second line uses `DIM` as a function - with the ‘whole array’ reference of `Array()` as its argument. The function `DIM( Array() )` returns the number of dimensions in the named array.

Similarly, in the third line, `DIM( Array() , n )` returns the size of the nth dimension of the array.

Hence here, the fourth line will print the results 4 and 2 respectively.

Remember, there must not be a space between `DIM` and the first bracket, nor anywhere in the reference array name.

Assigning values to elements

The statement:

```
Array() = ( Number )
```

will assign the value of the variable `Number` to every element of the named array. As usual, any numeric expression can be used, but **it must be enclosed in brackets.**

The same format can be used with string arrays. Thus:

```
Array$() = ( String$ )
```

and it is safest always to enclose the right-hand side in brackets.

You can assign different values to different elements by:

```
Array() = 1 , 2 , 3 , 4 , 5 , 6 , 7
```

and the order of the assignment "changes the right hand subscript most frequently" e.g. in our 4-dimension example, the above numbers would be assigned as follows:

```
Array( 0 , 0 , 0 , 0 ) = 1
Array( 0 , 0 , 0 , 1 ) = 2
Array( 0 , 0 , 0 , 2 ) = 3
Array( 0 , 0 , 1 , 0 ) = 4
Array( 0 , 0 , 1 , 1 ) = 5
Array( 0 , 0 , 1 , 2 ) = 6
Array( 0 , 0 , 2 , 0 ) = 7
```

and the assignment of other elements would not be changed, as there are no more numbers in the list.

Operating on element values

All elements can be increased, decreased, multiplied or divided, using the form:

```
Array() = Array() + ( 7 )
Array() = Array() / ( 6 )
```

etc. or the form:

```
Array() += ( 12 )
```

Also, with string arrays, you can use:

```
Array$() = Array$() + ( String$ )
```

to add `String$` to the end of the existing string in every element.

In all these cases it is safer to put a pair of brackets round the RH side, because it is necessary in some cases but not in others.

Arrays of **identical size** can be added, subtracted, multiplied and divided using the form:

```
Array1() = Array1() * Array2()
```

in which case the operation is carried out on **each pair of corresponding elements**.

You can also use the form:

```
Array3() = Array1() * Array2()
```

provided that `Array3()` **has already been dimensioned** (to the identical size).

Again, you can do the same with string arrays, within the constraint that only the + sign has a meaning.

Matrices

For those who understand mathematical matrices, BBC Basic also provides proper matrix multiplication, using the full-stop character as the operator sign, for example:

```
Array3() = Array1() . Array2()
```

Limitations

The only limitations on the above series of array operations are that they cannot usually be extended to more than one operation - and you can only use them (as above) on the RH side of an ‘equation’ and where the LH side is an array reference of an existing array. Thus:

```
Array4() = Array1() * Array2() + Array3()
```

cannot be used - it would need to be split into two steps. Nor:

```
PRINT Array() + (3)
```

which is meaningless.

Appendix 11. Supplementary notes on Tutorial program formulae

The two formulae used in the Loan program and first introduced in Chapter 9 are:

$L \div P = B + B^2 + B^3 + B^4 + \dots\dots\dots\dots + B^{N-1} + B^N$

..........(Formula 9.1)

where B = 100 -r (100 + R), and:

L=Loan Amount (£)

P= Amount of each monthly payment (£)

N=Number of equal monthly payments

R=Monthly interest rate (%)

and:

Loan Remaining after N equal payments

$= (L \times B^N) - P \times (1 + B + B^2 + \dots + B^{N-1})$

..........(Formula 9.2)

where, now, B = (100 + R) ÷ 100 i.e. the **inverse** of the previous relationship; and the other factors are the same as before.

Both these formula are derived from the same starting point, as follows.

If our loan amount is L and we pay equal monthly premiums of P - which is designed to gradually pay off the loan and to pay the interest (which is quoted as a monthly rate R) - then:

at the end of the first month of the loan we owe the loan amount plus the first month's interest i.e.

$$L + L(R/100)$$

This can be re-arranged to

$$L\ (100 + R) / 100$$

or, if we substitute B for (100 + R) / 100, simply to:

$$L/B$$

After we have paid the first monthly premium (of P) the amount of the loan outstanding is:

$$(L / B) - P$$

which is the starting point for the second month.

If we repeat the process we will end up with:

Loan Remaining (after 2 months and 2 payments)

$$= L\ (1 / B)^2 - P\ [\ 1 + (1 / B)]$$

and after X months:

Loan Remaining

$$= L\ (1 / B)^X - P[\ 1 + (1 / B) + (1 / B)^2 + ... + (1 / B)^{X-1}]$$

which we will call our 'baseline formula'.

If we now look at the special case where 'Loan Remaining = 0', this formula effectively becomes:

$$L\ (1 / B)^X = P[\ 1 + (1 / B) + (1 / B)^2 + ... + (1 / B)^{X-1}]$$

which, on re-arranging, becomes:

$$L / P = [1 + [1 / B) + (1 / B)^2 + ... + (1 / B)^{X-1}] / (1 / B)^X$$

or:

$$L/ P = [1 + (1 / B) + (1 / B)^2 + ... + (1 / B)^{X-1}] \times B^X$$

which reduces to:

$$L/P = B + B^2 + B^3 + B^4 + ... + B^{X-1} + B^X$$

and this is identical to ***`Formula 9.1.`***

Returning to our 'baseline formula', if we invert our definition of B (as used in ***`Formula 9.2`*** earlier) to:

$$B = (100 + R) \div 100$$

then it is easy to see that our 'baseline formula' is already effectively identical to ***`Formula 9. 2`***.

With these derivations a few further comments about how they are used in the Loan program may assist.

`DEF PROCfindLoanAmount`

This is the simplest case. ***`Formula 9.1`*** is rearranged to:

$$L = P (B + B^2 + B^3 + B^4 + ... + B^{N-1} + B^N)$$

The `DEF PROC` firstly calculates B (called `RateFactor` in program), then evaluates P multiplied by the summation of the B terms, using FNsummation(). This FN uses two main variables TermValue and CuSum. The first of these (Line 2930) successively takes the value B, B2, B3 etc. as the FOR next loop progresses. The other (Line 2940) accumulates the sum of these terms.

`DEF PROCfindPaymentAmount`

This is very similar to the above; merely using the rearrangement:

$$P = L / (B + B^2 + B^3 + B^4 + ... + B^{N-1} + B^N)$$

DEF PROCfindNuraberOfPayments

The ‘Loan Remaining’ formula (***Formula 9.2***) is used in this DEF PROC. The elements of the calculation are very similar to the previous two PROCs (except that B is now the inverse and called InverseFactor).

However, a FOR ... NEXT loop cannot be used because we don’t know N, so we use a REPEAT ... UNTIL loop, which continues until the **loan remaining plus one month’s interest on it** is less than one monthly payment (P). We then use the exit value of the loop counter NumOfPayments% to give us the value of N.

You can immediately see that this method is an approximation and it is arguable whether or not we should add one to NumOfPayments% for the ‘real’ answer. But this does not matter too much for our purpose of introducing Basic programming.

Index

! 234
!DeskEdit 18, 288
!Edit 10, 18, 36
!Run file 217
!SciCalc 288
!Zap 18
218
$ 33, 234
% 32, 288
& 289
&0D 238
* 12, 31
*DIR 218
*DUMP 223
*FX261
+ 31, 119-120
+= 32
- 31
-= 32
. 309
/ 12, 31
16-colour default palette 245
2, 4 and 16-colour modes 240
256-colour screen modes 246
2-byte integer numbers 194
32-bit words 302
<< 293
<> 96
<backspace-and-delete> 19, 89
<Caps Lock> 79
<ctrl>+ 199
<ctrl-X> 19
<Delete> 12
<Esc> 243
<menu> 275
<return> 7, 79, 89, 92, 99
<select> 275
<shift> 197
> 11
>> 293
>>> 293
? 90, 234
¦ or | (ASCII 124) 198, 234
~ 120, 289
‘ 91

A

abbreviated form 36
ABS 131
Acorn Window Manager 268
adding strings 120
alphabetical sorting 183
amending and listing 15
amounts (intensity) of primary colours 242
AND 73, 82, 293
AND with colours 250
animation 181, 254

argument 45, 120, 131-132
arithmetic operators 31, 127
arrays 34, 207
ASC 78
ASCII 78, 120, 124, 198, 236, 296
ASCII codes 0-31 79
ASCII codes 0-127 78
ASCII codes 128-255 79
aspect ratio 136
audible ‘feedback’ 197

B

background colour 136, 160
base 16 288
base 2 288
base 8 288
Basic environment 11, 18
Basic file 215
Basic file icon 16
Basic Immediate mode 258
Basic options 19
Basic prompt 11, 90
Basic statement 7
Basic variables 264
beep 197
binary 82, 246
binary system 285
binary/hex link 289
bit 82
bit position 82, 287
bit shift operators 296
‘bit-wise’ operators 293
block of memory 233
block plot actions 146
brackets 127, 129, 132
branching 103
brightest shade 247
building PROC/FN libraries 257
BY 137-138
byte 222, 288, 301

C

CAD applications 254
calculations 31
called 51, 52, 56, 97
caret 10, 90
carriage return 78, 195, 236
CASE 92, 103, 125
changing the palette 242
channel number 216
character grid 159
character positions 42
characters per line 67
CHR$ 71, 78
CHR$(GET) 81
CIRCLE 139, 145
CLG 241
CLOSE# 218
CLOSE# 0 218
closing/opening directories 2
CLS 199, 241

colon 7, 13, 17, 48, 104
COLOR 249
Colour 67, 136, 239-241, 245
colour number 239, 242
comma 39, 43-44, 55, 90, 91, 104, 160, 185, 193, 198, 203
comma used to "acknowledge the presence" of register 264
Command Line 10, 256
Command Line mode 8, 10
common error messages 48
communication channel 216
concatenate 120, 213
condition 72, 92
control codes 78, 195
control loops 71
conventional x-y coordinates 133
coordinate system 133
COS 131-132
count from zero 42
counting and incrementing 32
CR 195
Current Directory 16, 275
Currently Selected Directory 16, 275
cursor 10, 43
customised prompts 91

D

darkest shade 247
DATA 201
data block 234
data file 215, 216
data file pointer 220
DATA pointer 204
data storage formats 228
data type identifiers 229
decimal places 123
decimal system 285
declaring a variable 30
declaring string variables 299
DEF 57
default 20, 38-39, 133, 157
default foreground and background colour numbers 239
default INPUT prompt 90
default justifications 44, 45
default output for numbers 42
default set of colours 239
default TINT levels 248
defining special characters 79
defining viewports 159
DEF FN 184
DEFPROC 52, 184
DEG 131
degrees 132
delete 19
desktop 2
desktop mode 8
desktop screen 215
desktop window 18
destination variable 264

diadic form 235
dialects 6
DIM 207, 233
DIM as a function 307
dimension 207
direct memory locations 201
direct printing from Basic 38
direct values 17, 104
directory 16, 20
directory window 20, 215, 275
display mode 133, 135
display resolution 135
DIV 68, 129, 297
double-click 20
drag 19-20
drag & drop 2, 38
DRAW 137-138, 145
drawing a graph 155
‘DUMP' output 231

E

elements 207
ELLIPSE 139-140, 145
ELSE 92-93
empty byte 228
empty lines 17
empty PROCs 57, 86
END 15, 47, 57
end condition 72
end of file 223
ENDCASE 92, 103, 125
ENDIF 93
endless loops 188
ENDPROC 52
ENDWHILE 74
entry condition 74
EOF# 223, 229
EOR 293
EOR with colour 250
erase a message 245
ERL 47
ERROR 46
error message 11, 46, 55, 92, 122-123, 160, 204-205, 208, 211, 217-218, 220
error trap 26, 47, 161, 221
“evaluates to TRUE” 73, 280
Escape Codes 296
Exclusive OR 293
exit condition 72, 97
expressions 104
EXT# 222
extent 222

F

FALSE 73, 92
fatal errors 47
file name 20, 277
file specification 226, 255
file types 215
file ‘DUMPs’ 223
files 34

FILL 140
filled shapes 140
flag 182, 184, 221
flashing colours 241, 246
floating point 33
FN 51, 56, 104, 115
FN names 53, 277
FN to produce Binary 290
FOR ... NEXT loop 71, 109
foreground colour 136
formal parameter 54, 95-97, 154, 186, 258
formal parameters automatically made ‘local’ 185
free-standing PROC/FNs 185, 186, 258
full-stop character 309
functions 56, 80

G

GCOL 136, 158, 241, 249
GET 68, 71, 79
'GET AND 223’ 82
GET$ 68, 71, 79, 106, 123
‘GET OR 32’ 82
global variable 185-186
graph drawing 155
graph plotting colour 174
graphic colours 67, 136
graphical solution 149
graphics 133
graphics cursor 137, 158
graphics logical colour number 250
graphics plotting 137
graphics screen 157
graphics viewport 161
graphics ‘play area’ 133

H

“hash” character 218
heading of group of SWIs 266
HELP . 35
HELP <keyword> 35
hex 119, 227, 288, 301
hex and binary numbers 285
hexadecimal (hex) 78, 82, 119, 130, 288
histogram 148
home position 38
hours:minutes:seconds 130
how data items are stored 223
how integer and real numbers are stored 302

I

icon 18, 215, 267
iconbar 18
identifiers 229
identify a data block 233
IF ... THEN ... ELSE ... ENDIF 89, 92
Immediate Mode 12, 14, 90

indenting 73, 76
indirection operators 233-234
‘initiation’ PROC 67
INKEY 86
INKEY$ 86-87
INPUT 89
input and output to/from Wimp application programs 268
input from the keyboard 79
input prompt message 99
input validation 96, 117, 184
INPUT# 218
INSTALL 256
INSTR(71, 77
INT 131, 183
integer division 129
integer numeric variable 29, 32, 79
integer part 48, 129
integer remainder 130
integers 8
intensity (amounts) of primary colours 239, 242
invalid inputs 97
inverting a colour 250
inverting 294

J

jaggedness 135

K

keep asking ‘the question’ 268
keeping track of pointer 230
keyboard 'Control’ equivalents 199
keywords 7, 35

L

Least Significant Bit 287
least significant byte 236
left justified 44
LEFT$(121
left-shift 297
LEN 68, 122
length of a Basic program line 8
length of a string 33
LET30, 37
levels of intensity 243
libraries 187, 205, 255
libraries (separate Wimp and non-Wimp) 258
LIBRARY 255
line-feed-plus-carriage-return 38
line number 8, 14
line number references 257
line types 191
LIST 15, 36
LOCAL 184
LOCAL DATA 184
LOCAL ERROR 184
local variable 184

locked to a grid 142
LOG 131
log on/off with the Window Manager 268-269
logical ('bit-wise') operators 127, 293
logical colour 242
logical colour numbers 242, 253
loop counter 72, 223
loop exit condition 116, 204
lost files 217
lower case 81
lower case letters 34
lowest address 236
LSB287
LVAR 258

M

machine code 5
machine code routines 261
masking 81, 295
mathematical functions 127
matrix multiplication 212, 309
meaningful variable names 34
meaningful PROC names 55
meaningful prompts 90
memory addresses 301
memory locations 263
MID$(121
MOD 129-130
MODE 67
modifiers 42, 90
more than one match (CASE) 104
Most Significant Bit 287
most significant byte 236
mouse button 267
MOVE 137, 145
MOVE BY 169
move the pointer 222
MSB 287
multi-line statements 104
multi-statement line 7
multi-tasking 10, 22
multiple conditions for exit 73
multiple graphs 173

N

negative arguments 132
negative INKEY 86
negative integers 129
negative numbers 126
'nesting' 93, 94, 129
'nested' loops 73, 76
new line 45
new page 78
NEXT 109-110
non-printing codes 78
non-Wimp 22
NOT 250, 280, 293
null string 33, 79, 92, 208
number of colours 66
number of different colours 239

O

objects at different distances 254
octal 288
OF 92, 103, 125
offset values 146
ON 47
one graph line 155
ON ERROR 47
open a data file 216
open an existing file 217
OPENIN 216
open a new file 216
OPENOUT 216
OPENUP 216
Operating System units 133, 161
operations on whole arrays 212, 307
OR 73, 293
OR with colour 250, 252
organisation of data files 231
OS units 133, 161
OTHERWISE 92, 103, 125
output stream 38
'overhead' 61
OVERLAY 256
overwriting 159

P

page of memory 301
paged mode 197
palette 239
parameter blocks 265
parameter passing 54
parameters 45, 157, 194, 263
pause 68, 79
'Pause% = GET' 100
permanent storage 201
PI 131
picture elements 134
pixel grid 135
pixel resolution 136
pixel size 140
pixels 134
planning 22
PLOT 145
plot a graph 150, 155
PLOT mode blocks and offsets 145, 147
POINT 140
POINT(141, 250
POS100
positive INKEY 87
post-entry checking of input 80
pounds and pence 123
primary colours 239
PRINT 38, 158
print directly from Basic 196
print field 42
print modifiers 39
PRINT# 218
printer 38
printer 'escape code' 196

priority rules 127-129
PROC definition 52
PROC names 53, 277
procedures (PROCs) 51-52
procedures/functions 34
PROCinit 67, 211
PROCs as structural aid 55
program compactors 34
program line 7
programming language 5
programming mode 13
prompt 8
pseudo-code 23, 150, 175
PTR# 221

Q

quote marks 48, 203

R

RAD 131, 132
radians 132
RAM 256
RAM-disc 231
random number 132
‘ratcheting’ 183
READ 202
read the current data pointer position 222
‘ready to send’ 78
real and integer inputs 49
real numeric variable 29, 33, 101
reason codes 269-270
RECTANGLE 139
registers 263-264
REM 26, 37, 57, 117, 204, 220
REM> 16, 38
remove line numbers 20
REPEAT ... UNTIL loop 71, 72, 97
REPORT 47
reset the text cursor 176
resident integer variables 33
RESTORE 202, 205
RESTORE + 205, 257
RESTORE DATA 205
RESTORE ERROR 205
returning a value 56
reveal a message 245
right justification 42, 149
RIGHT$(121
right-shift 297
RISC OS 2 61
RND 131, 132
root directory 275
rounded 43
rounding error 33, 184
rounding numbers 125
‘rounds to zero’ 129
RUN 17

S

SAVE 15, 38
'Save as' window 20
'Save box' 20, 275
saving the program 15
screen 133
screen background colour 67
screen display mode 38, 45, 66, 136, 239
screen pointer 267
screen resolution 66, 142, 239
scrolling 46, 160
selection from a menu 71
semi-colon 39, 43-44, 90, 158-159, 161, 194
set bit 288
set/reset DATA pointer position 205
setting the directory 217
shade/tint 246-247
shadow effects 159
'shell packages' 272
sieve 82, 296
sign extended right-shift 297
signed right-shift 297
SIN 131
single dimension arrays 208
single point/dot 141
single quote 39, 45, 91
single statement lines 8
size of file 222
skeleton Wimp program 272
smallest 'dot' 135
Software Interrupts 262
spaces 20
SPC99
SQR 131
square root 131
standard graphics screen 133, 136
star command 217
starting address 228, 233
starting applications 2
STEP 109-110
step through line-by-line 100
storing integer numbers 302
STRS 119
string 7, 30, 33
string manipulation 77, 119
string variable 29, 33
STRING$(123
StrongEd 18
subscript 208
SWI name/number 263
SWIs 262
syntax 35
SYS241, 258, 261
SYS &35 26
SYS "OS_Byte" 241
SYS "OS_ReadModeVariable" 262
SYS "Wimp_Poll" 269

T

TAB 26, 91, 149, 159
tab key 26
tab value 44, 78
TAB(38, 43, 45
TAB(0, 0) 45
TAN 131
Task Window 9-10, 36, 90, 110, 147, 256
temporary storage 201
text and graphics viewports 157, 160
text character position 159
text colour 67, 158
text coordinates 134, 160
text cursor 38, 158
text file 215
text lines 66
text screen 157
text-in-quotes 90
THEN 92, 94
tilde 120, 289
TIME 174
time limit 87
timing action 174
TINT 246, 250-251
TO 109, 262-264
transportability 187, 258
trigonometric and logarithmic functions 117
TRUE 73, 92, 106, 229
TRUE and FALSE 279
turn printer'on'and'off 196
tutorial program formulae 311
two's complement 236, 294, 302
two-dimensional array 208
typing errors 26, 48

U

underscore character 266
unset bit 288, 296
unsigned right-shift 297
upper and lower case 13
uppercase 34, 36, 81
use of commas 265
user friendliness 81
User Guide (UG) 2
user input 89, 99
user-friendly 90, 100
using the Operating System 261

V

VAL 120, 131
variable declaration 30
variable names 30, 34, 36, 277
variable types 32, 55
variables 17, 25, 30, 104, 201
variable's value 30
VDU 100, 157, 193, 195
VDU0 198
VDU 1 195
VDU 2 196

VDU 3 196
VDU 4 158
VDU 5 157
VDU 6 196
VDU 7 197
VDU 8, 9, 10, 11 307
VDU 12 199
VDU 13 195
VDU 14 197
VDU 15 197
VDU 17 197, 242
VDU 18 197, 253
VDU 19 197, 246
VDU 20 197, 253
VDU 21 196
VDU 23 197
VDU 23, 9 241
VDU 23, 10 241
VDU 24 159, 161
VDU 26 161
VDU 30 176
VDU 31 100
version 6
version number 11, 258
Version V (Basic) 6
Version VI (Basic) 6
vertical scale 149
viewing window 133
viewports 157
Visual Display Unit 193
VPOS 100

W

waiting for a valid keypress 81
warnings 47
Welcome Guide (WG) 2
WHEN 92, 103, 125
WHILE ... ENDWHILE loop 71, 74, 269
Wimp 1, 22, 89, 268
Wimp poll 269
Wimp programming 119, 238, 255, 267
Wimp programming tools 272
word-aligned addresses 234, 236, 302
word operator 236
word wrap 25, 46
words 302
writable icons 89, 271

Y

"Yes/No" 80

Z

zero 92
zoom box 177