Advanced Programming for the BBC Micro

Mike James and S. M. Gee

Contents

Preface

Most programmers begin by learning a high level language such as BASIC and then immediately apply it to the task of writing programs. This seems to be a natural extension of the way they learned the language in the first place: by way of examples but examples are nearly always short and there is a world of difference between a short example and a long applications program! The difficulty is that there really are new problems to be solved in tackling a large program that were not present in the short examples. A short example of even as many as fifty lines can be 'held in the head', whereas a program that is likely to do anything useful will be composed of so many lines that this is impossible. The result is that large programs present a level of difficulty that is quite unnecessary.

There is more to programming than knowing the elements that make up a computer language. In fact, this is only the first stage on the road to becoming an advanced programmer. Learning how to apply a computer language to real problems is mostly a matter of solving the problem of writing larger programs and being aware of what your objectives are, Using the ideas presented in this book you will find not only is it possible to write large programs quickly but they are also more rewarding than any short examples. The time saved by writing large programs in an organised fashion can also be spent in dealing with details such as getting rid of bugs and making programs 'friendly'. In this sense advanced programming should be a rewarding experience for all concerned including the user!

To get the most from this book it is assumed that you have reached the stage of being able to write short programs in BBC BASIC and generally know your way around the machine. Some chapters also assume that you know something about 6502 assembler, although if this is not the case you can skip the relevant sections and come back to them at a later date. The reason for using 6502 assembler is simply that the mixing of BBC BASIC and assembler is essential to the production of efficient programs. Although most of the programming methods described can be applied either to BASIC or to assembler (or any other computer language!) without any difficulty, we have included many specific examples of 'structured assembler' and have even devoted a whole chapter to it!

The examples that are used in this book are of course larger than you would expect to find in other computer books and they are useful programs in their own right. They include a disk sector editor, a spelling checker, an execution tracer, a background clock and a disassembler. If you intend writing applications programs as large and as complex as these then you cannot afford to ignore the ideas contained within this book they make programming easier, more satisfying and more enjoyable.

Mike James and S. M. Gee

Chapter One
Advanced Programming

The BBC Micro is a computer that encourages the two approaches to programming that concern this book producing clear, bug-free programs, and using BASIC and assembler in combination. These features make the transition from simple programming in BASIC to 'advanced programming' a particularly painless and logical one.

It doesn't take long to discover that BASIC is a very simple computer language. The total number of BASIC commands can be numbered in tens and the set of commonly used ones is not many more than ten! In fact there is nothing very complex about any computer language but knowing the details of a language is not the same thing as being able to use it to achieve any given result. To make a computer work to your advantage you must not only know *what* you want it to do, you must also know how to make it do it. This is sometimes summarised by saying that our current use of computers is at the 'HOW' rather than 'WHA T' stage. The process of telling a computer how to do something is usually referred to as 'programming'. However, the subject of this book is 'advanced programming' and this raises the question of what the word 'advanced' means in this context. In this chapter what constitutes advanced programming is explored and, in passing, the topics that form later chapters are introduced.

Advanced languages?

As with so many things in computer science, it is easier to say what 'advanced' doesn't mean than what it does! Advanced programming certainly doesn't imply the use of an advanced or difficult programming language. It is a common fallacy that BASIC somehow lacks an essential ingredient and is thus unsuitable for certain programming tasks. This fallacy is reinforced by the way that academic computer scientists promote languages such as Pascal and - increasingly the newest language creation, Ada. However there is very little in either of these languages that a programmer familiar with BBC BASIC would not recognise and

understand. The sophistication of a language generally doesn't extend the range of things it is capable- of; it just makes it more or less convenient for the application in hand. At the other end of the scale, it is often supposed that it represents an advance to 'graduate' to assembly language (often referred to as 'machine code'). This idea is fostered by the fact that many commercial programs that achieve effects that seem impossible from BASIC are written in assembler. However, assembler certainly cannot be thought of as an 'advance' in any sense of the word. If anything, the continuing use of assembler is a reflection of how inadequate and crude the current generation of computers are, in that the only reason for using assembler is to gain extra speed of processing. Assembler is fast but it isn't advanced.

The fact is that the elements which make up computer programming are very simple, no matter in what language they turn up. Very roughly speaking, a program, in any language, manipulates data, uses loops to repeat actions and selects between actions using the IF statement or its equivalent. Much confusion is caused by the assumption that problems encountered when writing a program in BASIC would be solved by using aother language. In most cases the program will prove just as difficult to write in a new language as it did in BASIC because you are trying to solve the same problem with only slightly different tools. Another way to sum this up is if you don't know how to do it in BASIC you won't know how to do it any other language!

Programs small and large

Once you have learned what each of the statements in BASIC do, the next task is to use them to write programs. At first, the sort of programs that you write are short enough to be written in one sitting. They also contain very few statements so the scope for anything other than obvious bugs is limited. At this stage it is very easy to recognise a successful, working program. At the next stage, however, things are not so easy. There is a world of difference between writing a small program and writing a large program. It is important to realise that this difference is not just one of degree; there genuinely are new types of difficulty to be found in writing large programs. Some novice programmers never realise that this is the case and so fail to progress beyond short programs. Others discover ad hoe methods and create large programs through sheer enthusiasm. Both of these conditions are sad because it is very easy to learn and use techniques that make large programs not only possible, but possible without superhuman labour!

Using an organised approach to program construction is most certainly a part of advanced programming but it's not the only part. For example, writing larger programs using any method brings the possibility of introducing larger and more subtle bugs. There is also the question of making programs easy and pleasant to use. Writing programs that work as

intended is difficult, if not impossible, without being aware of the stages of 'bug detection' and 'bug location'. Good initial program design is also a help in keeping the number of bugs to a minimum but there is no fool-proof way of ensuring that a program is entirely bug-free. Don't take this as an excuse for ignoring the responsibility for removing as many bugs as possible from a program but see it as an indication that a feature of advanced programming is the need to use what time and resources are available to the best effect. While it is possible to give firm guidelines on how to deal with bugs, it is much more difficult to be definite about how to achieve user-friendly programs. After all the methods for program design and debugging have become second nature, the advanced programmer can start to tackle the real problems of writing programs that not only are useful but are a pleasure to use.

Communicating the total view

So far, it looks as though advanced programming amounts to nothing more than a collection of methods to aid program construction. However, there is a world of difference between applying these methods and appreciating how they work. Another component of advanced programming is knowing when rules and regulations can and should be broken. To this end it is important that the underlying aims and theory of programming are understood rather than being just a collection of rigidly applied methods. In this sense there is still a great deal of freedom to be exercised and there is no need to fear that the fun will be taken out of programming by trying to do it better. Indeed, there is much more fun and satisfaction to be gained from a well-conducted programming project than from one that is scraped together by sheer brute force or hours spent slaving over a keyboard.

There is another aspect of using a systematic approach to programming that is worth explaining at this early stage. As programming develops, it is becoming as much a method of communication as a way of getting things done. It is almost a cliche to say that there are more programmers alive today than ever before (the world's population is such that there is more of everything!); but it is important not to miss the fact that programs are one of the main channels of communication between programmers. A working but difficult-to-understand program fails to satisfy the function of communication and in this sense it is doomed to failure in the longer term. The reason for this is simply that no program is ever complete. It may serve the needs of the moment but the progression of computer hardware and users' expectations mean that it will inevitably become inadequate. When a program reaches the end of its useful life it is usually replaced but this can be achieved in two ways - by starting from scratch or by producing a new version of the old program. In both cases the only way to avoid re-inventing the wheel is to make sure that the ideas that the original program contains are clear for all to see. This principle of building on the

best of the past is one that is already well known in science subjects. To quote Sir Isaac Newton, the greatest English physicist and mathematician: 'If I have seen further it is by standing on the shoulders of giants'.

To make similar advances in computer science it is important that we learn to expect new software to 'stand on the shoulders' of its precursors. The skill of programming should include the ability to both read and write programs. In the past the emphasis has been on writing programs possibly because the current state of the art made reading existing programs almost impossible. Advanced programming must be about writing programs that are easy to read and learning to read programs written by other people.

Know your machine - the role of experiment

The role of any particular computer in advanced programming has been played down until this point because, in the same sense that advanced programming is nothing to do with a particular computer language, it is similarly machine independent. However, this machine-independent attitude misses the point that you have to have the experience of knowing a particular machine before you can generalise. In the same way as you must learn BASIC before you can understand the ideas of programming, you have to study a particular machine before you can understand the broader ideas of computer science and computer use and this book is specifically about advanced programming using the BBC Micro - an excellent machine for the job of illustrating almost any idea in computing!

Many pieces of hardware information seem to be 'one off facts' that turn up by accident. That is, if you didn't know them there would be no systematic way of setting out to discover them. This 'one off aspect of hardware is only true in part because there are many methods of investigating the way that a computer works that are common to all machines. It is a step in the right direction to realise that computing is an experimental science! An advanced programmer will have an idea of the possible ways that a machine can operate. Using this knowledge and perhaps some initial observations it is usually possible to guess the way things might work in a particular case — in other sciences this is referred to as an experimental hypothesis but for us the term 'guess' will do! On the basis of this guess you can predict what you would expect to find and then write a program or examine the machine in some other way to confirm or deny the correctness of your guess. This is an experiment in any subject's language, but computer scientists sometimes refer to it as 'having a look to see'!

Throughout this book there are many specific pieces of information given about the workings of the BBC Micro's hardware and software, but much of the time the emphasis is on how these facts were discovered. This means that as well as providing you with some information of practical value, it should be possible for you to see how to add to this body of knowledge. A great deal of information about the overall structure of the

BBC Micro can already be found in *The BBC Micro: An Expert Guide* (Granada, 1983). Occasionally it will be necessary to summarise information from that book so as to make this book complete in itself, but the overlap between the two volumes is slight. In the same way the overlap between this book and the *BBC Micro User Guide* is held to the minimum but it is assumed that you have a copy available. Frequent reference is made to the *User Guide* so keep it to hand.

The languages used - BASIC and assembler

As already mentioned, BBC BASIC is a suitable language for advanced programming but it has to be admitted that sometimes the speed of assembler is indispensable, For this reason, although many of the examples in this book are in nothing but BBC BASIC, it would be impossible to write a book on advanced programming that ignored assembler. Rather than introduce it with apologies wherever it proves necessary, it is better to treat it on a more equal footing with BASIC. So, as well as turning up in some of the examples, you will also find whole chapters devoted to assembler. In particular, Chapter Four deals with a subject that is almost never discussed - programming style in assembler. The most probable reason for this neglect is that it is somehow assumed that assembler is such a difficult computer language that just to be able to write programs using it is a sufficient level of proficiency. This is, as the preceding discussion of advanced programming should indicate, completely untrue. Assembler is difficult because the programming methods that are advocated for languages such as BASIC are ignored in assembler! It is as possible to write a well structured assembly language program as it is to write structured BASIC, Pascal, ALGOL . . .

   As with BASIC, it is assumed that you have reached the stage of understanding the rudiments of assembler and this book makes no pretence of being an introduction to 6502 assembler. However, this is not to say that you have to know assembler to get anything from this book. Most of the ideas are language independent and the results of the assembly language sections can be used without understanding them. The hope, of course, is that if you don't know assembly language you will be inspired to invest the time necessary to learn it!

   The twin approach of using BASIC with assembler is realistic in that it represents what really happens in writing programs. Assembler is, at the best of times, a less readable language than BASIC and there are fewer programmers who understand it. For these reasons it is important not to use it without good cause. It is rare for the bulk of a program to prove impossible to write in BASIC and the best approach is to incorporate assembly language subroutines within programs where they are absolutely essential. This combination of BASIC and assembler is the most powerful way to write programs.

Using this book

After all this discussion of what constitutes advanced programming you should be able to see how the topics dealt with in each chapter relate to the overall idea. Advanced programming is not a subject with an exact and finite syllabus and so it is impossible to include everything that is worth knowing. However, you should find that most of the important subjects are covered. The first part of the book is about programming methods in general, including structured programming, debugging and data handling. The latter part is devoted to some large example programs which serve more than one purpose. They are intended to teach aspects of programming, to iflustrate special facilities of the BBC Micro and to be useful and a-ttractive applications.

Finally, it is a little known fact that computer books should not be read in order of strictly increasing page number! Although the material in this book is organised to reflect an overall progression of information, you shouldn't suppose that you have to understand things in any predetermined order. It would be going too far to suggest that the best strategy is to read computer books backwards but you should exercise a greater degree of freedom in the order of reading than is normal. On the other hand, if you find that you do not understand something, don't give in to the very natural impulse to re-read or go back to earlier chapters press on. It is often the case that later material clarifies an earlier misunderstanding while rereading' previous sections would only reinforce it.

Chapter Two
Program Structure

Many people write BASIC programs without ever having thought very much about the resources at their disposal. Sitting down and writing a program 'as it comes' is a workable method for small programs but it is difficult to keep enough detail in your head when working on a large program. To write a large program easily and with the minimum of bugs it is important to understand the way that the instructions that constitute the BASIC language are best used to construct programs. The surprising fact is that the 'units' that make up a program are not, as most elementary BASIC textbooks suggest, single BASIC instructions such as IF or GOTO. The best way to think of, and hence write, a BASIC program is in terms of larger standard units each composed of a number of BASIC statements. For example. the action of the GOTO instruction is easy enough to understand; it transfers control to a given line number, but there are only a small number of circumstances in which it is necessary to transfer control to an out of sequence line number and therefore it is better to think in terms of the 'use' of the GOTO rather than simply what it does. These larger program 'building blocks' are often referred to as structuring elements and the programming method based on consciously using them in program construction is often referred to as structured programming. However, all these terms make it sound as though something forced and unnatural is going on in the use of a 'structuring method' when all that is really happening is the identification and conscious use of the natural structures which the BASIC programming language provides. This is akin to a skilled carpenter learning to work with the grain of the wood rather than against it. You can write programs that go against the 'grain' of BASIC but it is much better not to!

The two parts of a program

Any program has two aspects, its structure and its data. This corresponds to the two aspects of any list of instructions, they always tell you (a) what to do and (b) what to do it to.

In this sense, every program is like a simple English sentence it has a

verb, the actions that are to be carried out, and a noun, the data that is to be manipulated by the program.

These two aspects are equally important and indeed there is a trade-off that can be made between the complexity of action and the complexity of the data that the action is applied to. For example, if you use a string to hold a list of numbers then the program to manipulate those numbers is likely to be more complicated than if you had used an array.

The subject of this and the next chapter is program structure. The other side of the coin, data, is dealt with in Chapter Five. What is missing from this neat division is the interaction that occurs between the two in writing a real program. This interaction is so problem-specific that it is almost impossible to discuss it directly. However, it is hoped that as well as illustrating program and data structure separately, the larger examples will also show how the two are balanced in program construction.

The flow of control and spaghetti

One of the first ideas that any programmer has to understand is the flow of *control* through a program. The default flow of control, whereby BASIC statements are executed in order of increasing line number and from right to left, is the first to be introduced. This rarely causes any trouble but programs that use nothing but the default flow of control are so simple that they completely fail to capture any of the power of a computer and therefore it is not long before the GOTO, IF and FOR statements are introduced as ways of altering the flow of control. The trouble is that normally no guidance is given on how to use these statements to construct programs. Their action is stated and after a few examples it is supposed that the problem in hand will make it clear how the flow of control has to be changed and hence which statements should be used. This is usually referred to as *free style* programming but for reasons that will be made clear it might better be called 'spaghetti' programming.

Before it is possible to discuss desirable features of the flow of control there has to be an easy way to see it. If you look at any program its flow of control will be difficult to appreciate because of the distracting elements of variables, PRINT statements, etc. If you take a pencil and draw a line that follows all of the possible routes through a program then the result is a 'map' of the flow of control that we can examine without being distracted by the other features of the program. For example, the flow of control map of a simple program that only uses the default flow of control is a straight line down the page (see Fig. 2.1). (It is usual to treat multiple statements on a single line as equivalent to a single 'compound' statement for the purposes of mapping the flow of control.)

```
10  INPUT A
20  A = A*2
30  PRINT A
40  A = A *2
50  PRINT A
```

Fig. 2.1 The default flow of control

IF A<>0 THEN A = A–1 ELSE A = A+1
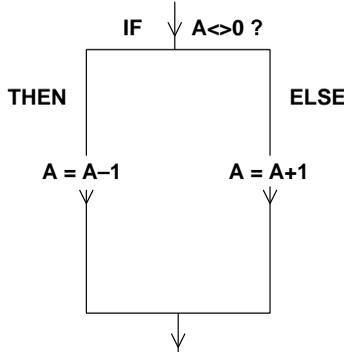
Fig. 2.2. The division of the flow of control.

The effect of an IF statement is to divide the flow of control line into two parts corresponding to the path taken when the condition is true and the path taken when it is false (see Fig. 2.2). The effect of the GOTO statement on the flow of control line is much more difficult to summarise. It certainly diverts the flow of control line but the effect depends on where it is diverted to. Using the GOTO statement it is possible to produce a program that has a flow of control line that is reminiscent of tangled string or a bowl of spaghetti - hence the term spaghetti programming! (see Fig. 2.3).


```
10   A = 0
20   GOTO 60
30   PRINT A
40   A = A + 1
50   GOTO 10
60   IF A = 0 THEN GOTO 90
70   PRINT A
80   GOTO 10
90   A = A – 1
100  GOTO 30
```
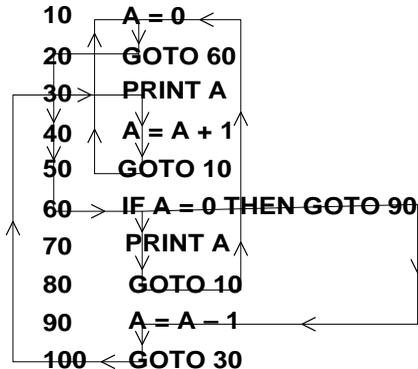
Fig. 2.3. Spaghetti!

It should be easy to see that a program that has a simple flow of

control line is easier to understand than one that is complicated and tangled. It follows from this that a simple flow of control is less likely to hide bugs because the programmer writing it has a better chance of completely understanding its action! The usual objection to this argument for simple flow of control is that complex programs can only be written using complex flow of control. To a very limited extent this is true. The simplest flow of control, the straight line running down the program, is clearly inadequate for writing any useful programs but the additions that have to be made to it before you can write *any* program are very few. In principle all you need is some method of forming a loop and a conditional branch, but in practice it is easier to use a few more ways of changing the flow of control - but not very many more and certainly not the complete freedom that the GOTO statement gives you to tie the flow of control in knots!

If you agree that spaghetti programs are a bad thing, then the next question is how to ensure simplicity and clarity without making programming an impossibly difficult activity. This question is answered in the following sections, but if you don't see the point of a simple flow of control then all of this will seem like a waste of time. It is therefore worth thinking a little about the way that you use the statements that alter the flow of control in a program before moving on to read the rest of this chapter. From now on our explicit objective will be to find easy-to-apply methods of keeping the flow of control simple. The main tool in achieving this objective is to only use a small, fixed number of ways of changing it. This approach also has the advantage that, no matter which ways of changing the flow of control are chosen as 'standard', they will be easy to recognise within a program and this in itself makes the program easier to understand.

The natural structure of BBC BASIC

The fact that any program can be written using nothing but loops and branches is remarkable and deserves a good deal of philosophical pondering in its own right. However, the subject of this book is practical programming and for this reason it is necessary to extend this limited range to include one or two other natural structuring elements. In particular, BBC BASIC provides a number of special control structures that fit very nicely into our overall objective of simplifying the flow of control fine but it is important to realise that they are not found in other, less well designed, dialects of BASIC. For this reason structured programming in BBC BASIC is a little different from the structured programming in other dialects of BASIC but the objective is still the same. (The reader is referred to *The Complete Programmer* (Granada, 1983) for details of natural structured programming in traditional BASIC.)

As BBC BASIC also includes many of the instructions of traditional

BASIC there is sometimes an over provision of facilities. The most obvious example is the ability to construct subroutines using GOSUB and procedures using PROC. Procedures and subroutines serve the same purpose but, whereas GOSUB is traditional BASIC, procedures are new and do the job much better. The presence of:

IF ... THEN line number

and

IF ... THEN list 1 ELSE list 2

where 'list 1' and 'list 2' are collections of BASIC statements, is also an example of traditional BASIC versus enhanced BASIC. In early versions of BASIC the only form of the IF statement allowed was IF . . . THEN line number and BBC BASIC has to include this form to make sure that existing programs will run. However, the second form is a much more helpful when it comes to keeping the flow of control line simple and is therefore to be preferred.

It would be nice to avoid any use of older forms of BASIC when writing naturally structured BBC BASIC, but in practice this level of strictness would add too much to the tedium of programming! You can certainly avoid the use of the extremes like GOSUB and IF . . . THEN line number but there are one or two traditional statements that are more difficult to give up. For example, it is possible to write structured BBC BASIC without ever using the GOTO! In this sense the GOTO statement is yet another part of traditional BASIC that has found its way into BBC BASIC. Some people do advocate this total abolition of the GOTO statement as the essence of structured programming and it is easy to see why. The GOTO statement is certainly responsible for most of the tangles in the flow of control line and abolishing it would make it very difficult to write spaghetti programs difficult but not impossible! While avoiding the use of the GOTO would serve our main objective of simplifying the flow of control it turns out to be more useful to examine and allow a few uses of the GOTO statement. This attitude can be summed up as 'the GOTO is a dangerous but powerful statement that it is worth learning to use properly'.

As the two fundamental forms of flow of control are selection between a number of alternatives and loops, it is worth looking at the best ways of implementing each of these in turn, but first it is necessary to consider briefly the role of line numbers in BASIC.

The trouble with line numbers

In nearly all versions of BASIC (there are one or two exceptions) each line in a program must have a line number. This is so much part of the BASIC language that BASIC programmers rarely stop to question if so

many line numbers are really necessary. BASIC line numbers serve two purposes as convenient 'markers' for editing program lines and as 'labels' that are used by GOTO and GOSUB commands. It is clear that as far as editing is concerned, for the method to work every line must have a line number but for the use of GOTO and GOS UB the only lines that need line numbers are those that are actually referred to in GOTO or GOSUB commands.

There are many reasons why the dual function that line numbers serve is a problem in the easy production of clear and bug-free programs. The most important are:

(1) The line misplacement problem

The fact that every line has to have a line number to determine its position within the program is a mechanism that is very sensitive to typing errors. You only have to mistype a single digit and, not only does a line appear to be missing at one position, an extra line appears elsewhere, possibly in part of a program that was finished and debugged some time before. In this sense, typing a single digit incorrectly causes the destruction of the overall correctness of the program and leaves a very difficult bug lurking in a part of the program that you have tested and is therefore almost above suspicion! (This aspect of line numbers is taken up again in Chapter Seven,)

(2) Fixed destinations

When a line number is used in a GOTO or GOSUB statement, it is an advantage if this line number remains unchanged throughout the life of the program. The reason for this is that the line number of the destination of a GOTO or COS UB serves to 'name' a part of the program that performs a particular task. If this 'name' changes during the development of a program then it is possible that the programmer will incorrectly use the 'old' value instead of the correct new value when re-using that part of the program. For example, if during the initial implementation of a program, subroutine 1000 prints an opening message then you are likely to use GOSUB 1000 later on to print the same message. If, however, in the intervening time it has been necessary to renumber the program so that extra lines can be inserted, fine 1000 may be assigned a new fine number, 1370, say. From this point on, the programmer must remember that what was once referred to as subroutine 1000 is now subroutine 1370, a task made even more difficult by having to remember the fine number changes for all the other subroutines in the program! The moral is that renumbering is generally a bad thing.

(3) The forward jump problem

When writing a program it is easy to use a GOTO statement to transfer control back to a point earlier in the program. The reason for this is that usually the earlier part of the program is already written and it is very easy to find the line number of the point to which you want to return. This should be compared to the situation of using the GOTO to transfer control

forward to a line later in the program. In this case the chances are that the line to which you are trying to transfer control has not yet been written. The line number that it will eventually be assigned is difficult to work out as it depends on the number of statements that will be written and the line number increment in use. The best solution to the forward jump problem is not to make a guess at the fine number but to use a symbol such as '*' to indicate that you have to return and fill in the missing line number after the line concerned has been written. If you guess the line number it is possible that you will get it wrong and this will go undetected because, even though the line number you used was incorrect, the GOTO is still valid BASIC. If you use '*' to mark incomplete GOTO statements it is impossible to forget them as they will cause the program to crash if not removed!

It is possible to imagine a version of BBC BASIC that doesn't use line numbers for editing. Program fines could be entered, deleted, fisted and modified by use of the cursor keys instead. This would be a great improvement because the confusion and difficulties that result from the dual use of fine numbers would be completely eliminated. Failing this, the best course of action is to try to avoid the use of line numbers within GOTO and GOSUB as much as possible. Once again we have a reason for minimising the use of the GOTO and GOSUB statements!

Selecting between two alternatives

The simplest form of selecting between a number of alternatives is when there are only two possibilities. This corresponds to the well known:

IF condition THEN list 1 ELSE list 2

where 'list 1' and 'list 2' are collections of BASIC statements. The action of this statement is familiar enough if the 'condition' is true then 'list I' will be executed; if the 'condition' is false then 'list 2' will be executed. You can see this causes the flow of control line to split into two possible paths which recombine only after the entire statement is complete, see Fig. 2.2. This is a particularly simple way of dividing the flow of control; you can immediately see what happens when 'condition' is true and when it is 'false'. You can also see instantly where the flow of control fine recombines. If you think that these are small considerations, compare IF. .. THEN. ..ELSE with the traditional BASIC way of selecting between two alternatives:

IF condition THEN GOTO x

. . .

list 2

. . .

GOTO y

x . . .
   list 1
   . . .
y rest of BASIC program

where 'x' and 'y' are line numbers. Using this method in a real program makes it difficult to see what happens when 'condition' is true and when it is false. It is also very difficult to see where the flow of control line recombines and continues through the rest of the program.

    The advantages of the IF. . . THEN . . . ELSE form of the BASIC statement make it worth using in preference to ail other forms. When either 'list 1' or 'list 2' contain only a few statements they can be written within the IF statement separated by colons. If the lists are at all long they should be turned into procedures and then called from within the IF statement. For example, rather than:

   IF condition THEN t1:t2:t3 . . . tn
   ELSE e1:e2:e3 . . . en

where t1 to tn and e1 to en are lists of BASIC instructions use:

   IF condition THEN PROCIist_1 ELSE PROClist_2

where PROClist_1 and PROClist_2 are defined elsewhere in the program and consist of statements t! to tn and e1 to en respectively. The subject of using procedures is discussed more fully in Chapter Three but even at this early stage they show themselves as one of the most useful features of BBC BASIC. Notice that it is never necessary to use:

   IF . . . THEN GOTO x ELSE GOTO y

or

   IF . . . THEN GOSUB x ELSE GOSUB y

where x and y are line numbers.

    It looks as though the subject of selection is completely settled in that the IF . . . THEN . . . ELSE statement is just right for the job, but there is the question of what should happen when there are more than two possible courses of action. Or, at an even simpler level, how should the very common situation of conditional execution of one fist of statements be handled ?

THEN without ELSE

If you examine almost any BASIC program you will find that the form of the IF statement that occurs most often is:

   IF condition THEN list of statements

This can be regarded as a special form of the IF . . . THEN . . . ELSE statement where the list of actions that follow the ELSE part is empty. Although this may seem like a rather academic way of looking at a perfectly simple and practical BASIC statement, it does highlight a number of issues. The 'list of statements' following the THEN will of course only be carried out if 'condition' is true but in practice the converse is just as common a requirement; that is, the list of statements will be 'skipped' if a condition is true. This corresponds to an IF statement with an empty statement list following the ELSE. While you can indeed write something like:

IF A=0 THEN ELSE PRINT "Not Zero"

there is something very difficult and confusing about reading THEN ELSE. Of course, the most commonly encountered solution is to use the NOT of the 'condition' as in:

IF NOT (A=0) THEN PRINT "Not Zero"

or even the more readable:

IF A<>0 THEN PRINT "Not Zero"

Notice that although in this simple example the final form of the IF statement looks the easiest and the one that should always be used, this is not the case if the 'condition' is at all complicated. For example, if you want to skip a list of statements when:

A=0 AND (B<>0 OR Z=100)

is TRUE, it is much easier to write:

IF NOT (A=0 AND (B<>0 OR Z=100)) THEN list

rather than try to work out what the NOT of the condition is.

It is important to notice in this discussion of the IF . . . THEN statement that there is an implicit assumption that the list of statements that follows the THEN never need contain a GOTO. In particular there is never any need to use:

IF condition THEN GOTO x

to determine whether to execute or skip a section of program. If you need to execute a section of program according to some condition then use:

IF condition THEN list

and if you need to skip a section of program according to a condition then use:

IF NOT(condition) THEN list

Once again if the 'list' of statements becomes too long to be conveniently written on one line then it should be turned into a procedure

Multiple selection - ON

It is not often that a program has to execute one of a large number of alternatives according to a condition but when it does happen it is usually a major feature such as a 'menu selection' with a great many alternatives. Traditional BASIC uses the:

ON x GOTO list of line numbers

or

ON x GOSUB list of line numbers

to select which of a number of sections of program will be executed. The value of x, a numeric variable, selects which of the 'list of line numbers' will be the destination of the GOTO or GOSUB command. For example:

ON I GOSUB 1000,2000,3000,4000

will transfer control to 1000 if I is 1, to 2000 if I is 2 and so on. In this case if I is less than I or greater than 4 the program will crash. BBC BASIC has both ON. . .GOTO and ON. . .GOSUB and it even allows the statements to be followed by an ELSE statement to avoid crashing on out of range values Thus the previous example could be written:

ON I GOSUB 1000,2000,3000,4000
ELSE GOSUB 3000

and subroutine 5000 would be called if I was less than 1 or greater than 4.

Certainly if you are going to use the ON statement you should use the ON.. .GOSUB form and always follow it with an ELSE statement to catch out of range values. However, following the earlier discussion of the sort of problems that using line numbers bring it is obviously better to avoid the ON statement if at all possible. In theory, it is always possible to use a collection of IF statements to replace a single ON statement. For example, the previous ON . . . GOSUB can be written:

IF I=1 THEN PROCaction_one
IF I=2 THEN PROCaction_two
IF I=3 THEN PROCaction_three
IF I=4 THEN PROCaction_four
IF I<1 OR I>4 THEN PROCaction_error

Notice that each of the conditions is mutually exclusive and only one of the procedures following the THEN will be executed. The correctness of

the scheme depends on one vital implied rule:

> the value of I must not be changed by any of
> the procedures following the THEN

If this rule is broken then it is possible that more than one of the procedures will be executed. Sometimes there is a temptation to break this type of rule to obtain a quick result For example, if PROCaction_one changes I to 4 then when I is initially 1 both PROCaction_one and PROCaction_four will be called. This is the worst sort of opportunist programming! Anyone reading the program would be very unlikely to spot this trick and the results could be a completely tangled and bugged program! If your intention is also to call PROCaction_four when I is 1 then make it clear by:

> IF I=1 THEN PROCaction_one:PROCaction_four

In most cases the conditions in the collection of IF statements are not as simple as I=1, 1=2 and so on and therefore it is much more difficult to ensure that only one of the procedures (or fists of statements) will be executed. In situations where the conditions to be satisfied are complicated it is usually sufficient to ensure that each condition correctly determines whether or not a procedure (or list of statements) should be executed and that nothing that would effect the condition is altered until all of the IFs are completed.

You might object that this method of selecting one of a number of actions is inefficient and slow. Surely there is no need to work through the rest of the IF statements after one of them has found a true condition? The answer is that there is no satisfactory way of gaining the slight speed advantage of skipping the remaining tests after a true condition has been found without reducing the readability and simplicity of the program. One method that is often advocated is the 'nesting' (see later) of IF statements as in:

> IF A=1 THEN PRINT "One" ELSE
> IF A=2 THEN PRINT "Two"
> ELSE IF A=3 THEN PRINT "Three"
> ELSE PRINT "Error"

By no stretch of the imagination can this be considered as clear (and therefore less likely to hide bugs) as:

> IF A=1 THEN PRINT "One"
> IF A=2 THEN PRINT "Two"
> IF A 3 THEN PRINT "Three"
> IF A<1 OR A>3 THEN PRINT "Error"

There are plenty of examples of the use of the IF statement to make choices in later chapters. Finally, it is worth recalling that what matters is

clarity and simplicity. Do not value short cuts that save a few fines of program they will cost you much more time when it comes to debugging.

Loops

The other fundamental way of altering the flow of control is the loop. It is difficult to think of a worthwhile program that doesn't contain some element of repetition. Although BBC BASIC contains two distinct and special instruction pairs FOR. . .NEXT and REPEAT. . .UNTIL there is still a role for loops built using IF and GOTO and this is one of the few cases where it is worth risking the dangerous GOTO in a program.

The basic mechanism of all loops can be seen in the infinite loop:

```
x  list of BASIC statements
   . . .
   . . .
   GOTO x
```

The GOTO x transfers control back to line number 'x' and so repeats the lines of BASIC in between forever. In this description it is assumed that there are no GOTO statements within the loop that transfer control out of it.

Infinite loops are sometimes useful in applications where a computer has to control some piece of equipment theoretically forever, but apart from this loops usually have some way of coming to an end that is they are finite loops. A finite loop is produced from an infinite loop by the addition of a conditional statement that transfers control out of the loop hence their alternative name conditional loops. The standard form of the conditional loop is:

```
x   list of BASIC statements
    . . .
    IF condition THEN GOTO y
    . . .
    list of BASIC statements
    GOTO x
y   rest of program
```

The IF statement transfers control out of the loop to fine number 'y' when 'condition' is true. This constitutes an 'exit point' for the loop and its 'condition' is an 'exit condition', see Fig. 2.4.

The only way that finite loops differ from one another is in the number and position of their exit points. There are advantages of simplicity and clarity in using only one exit point within a loop. If there is only one exit point it is easy to determine what the exit condition is and where control is transferred when the loop ends. However, there are exceptions to this one exit point rule and these will be discussed later.
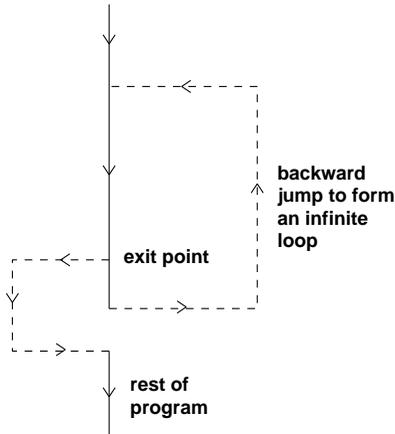
Fig. 2.4. A conditional loop

Given that there is only one exit point, the only scope for variation is in its placement. Once again, there are advantages of clarity and simplicity in placing restrictions on the way that loops are formed. Two favoured positions for loop exit points are at the very start of the loop and at the very end. Loops that have their single exit point at the start are usually called *while loops* and those that have their single exit point at the end are called *until loops*. The reason for these names is not important; they have their roots in other computer languages. What is important, however, is that BBC BASIC provides a pair of instructions, REPEAT and UNTIL, that explicitly implement until loops. Thus, rather than:

```
x   list of BASIC statements
    . . .
    . . .
    IF condition THEN GOTO y
    GOTO x
```
use

```
    REPEAT
       list of BASIC statements
       . . .
       . . .
    UNTIL condition
```
Notice that the REPEAT .. .UNTIL form of the loop completely avoids the use of GOTOs and line numbers. Unfortunately, BBC BASIC doesn't include a pair of instructions to implement a while loop and so there is no choice but to construct it from scratch using IF and GOTO.

As a while loop has an exit point at its beginning it is possible for this exit to be taken without the statements within the loop ever being executed. However, the statements within an until loop have to be executed before the exit point is reached. In other words, you can execute

a while loop any number of times including zero but you have to execute an until loop at least once.

Apart from these two simple characteristics concerning the minimum number of times that a loop can be executed, the main advantage of placing exit points at the start or the end of a loop is that all of the statements within the loop will be carried out the same number of times. For example, in the conditional loop:

```
10 I=0
20 I=I+1
30 IF I=10 THEN GOTO 60
40 PRINT I
50 GOTO 20
60 END
```

Statement 20 is executed ten times but statement 40 is executed only nine times. In general, the portion of the loop before the exit point is always executed once more than the portion of the loop that follows the exit point. The accepted wisdom is that this difference in the number of times statements are executed within a loop makes it difficult to understand, and hence debug, programs that contain anything other than while or until loops. However to restrict all conditional loops to just these two types is unnecessarily harsh. A natural form of loop that often occurs in BASIC is:

```
x   pre-exit statements
    IF condition THEN GOTO y
    post-exit statements
    GOTO x
y   rest of the program
```

where the pre-exit statements perform any actions that are necessary to discover the current 'value' of the 'condition' and the post-exit statements carry out the actions that have to be repeated if the 'condition' is false. This form of 'loop is more complicated and difficult to debug than either the while or the until loop and it should only be used when absolutely necessary. This more general conditional loop can often be found in BASIC programs where a little thought would have proved the while or the until loop perfectly suitable. In short use the while and until forms of the conditional loop unless there is a real need to use a more general form.

Multiple exit points

Although the emphasis in the last section was on the use of conditional loops with one exit point, there are times when a program can be made easier to understand by the use of more than one exit point. It is important to realise that the only reason for using such a complicated form of conditional loop is that control needs to be transferred to a number of

different points depending on what 'condition' caused the loop to end. It should never be necessary to use more than one exit point if they transfer control to the same position within a program. For example. the loop:

```
x   list of BASIC statements
    . . .
    IF condition 1 THEN GOTO y
    . . .
    IF condition 2 THEN GOTO y
    GOTO x
y   rest of program
```

can be reduced to the much simpler form:

```
x   list of BASIC statements
    . . .
    IF condition 1 OR condition 2 THEN GOTO y
    . . .
    GOTO x
y   rest of program
```

Of course there is the problem of exactly where the single exit point should be placed in the loop to give the same effective result as the pair of separate exit points but this is usually not at all difficult.

The real use of multiple exit points occurs when there are different reasons for leaving the loop and different actions that have to be carried out subsequently. For example, suppose you are searching a string array for a particular word using a loop that compares each element of the array against the 'target' string. There are two distinct reasons for leaving the loop. Either the target is found and further processing is necessary or it isn't and an error message (say) has to be printed. In this case, using a loop with two exit points is permissible and may even result in a simpler program. For example compare:

```
1000 I= I
1010 IF S$=T$(I) THEN GOTO 2000
1020 IF I=N THEN GOTO 3000
1030 I=I+1
1040 GOTO l0l0
```

which searches the string array T$ for an occurrence of the string in S$ using two exit points line 1010 exits to line 2000 if the string is found and line 1020 exits to fine 3000 if it isn't with its single exit point version:

```
1000 I=1
1010 IF S$=T$(I) OR FN THEN GOTO 2000
1020 I=I+1
1030 GOTO 1010
```

which is shorter but leaves the task of sorting out what causes the loop to end to the part of the program starting at line 2000.

It is rare for it to be necessary to use more than two separate exit points and conditions within a single loop and even then in most cases the exit points can be replaced next to each other.

Enumeration loops - FOR . . . NEXT

There is one very special and very important class of conditional loop with one exit point, the *enumeration loop*. An enumeration loop is one that causes a section of program to be repeated a number of times that is fixed before the loop begins. All versions of BASIC, including BBC BASIC, have a pair of instructions FOR . . . NEXT that are intended to be used to construct enumeration loops. For example:

```
FOR I=1 TO 10
. . .
list of BASIC statements
. . .
NEXT I
```

will repeat the 'list of BASIC statements' ten times. In some ways the existence of a 'loop counter' or 'index variable' (I in the example given above) that 'counts' the number of times that the loop has been executed is unnecessary to the idea of an enumeration loop but in practice it turns out to be an almost indispensable feature in the FOR loop, In particular, the index variable is often used in array calculation, etc., within the loop but its value should never be changed by direct assignment. If this rule is broken and the index variable's value is altered as the loop progresses the result is not fatal from the program's point of view, as the loop will run, but the task of working out how many times it will repeat is very difficult. The rule is that the number of times an enumeration loop is executed should always be clear' from an examination of the FOR statement without having to delve into the list of statements within the loop. This & so implies that an IF statement or direct assignment to the index variable should never be used to leave a FOR loop early. For example, searching a string array for a particular value is often done using:

```
1000 FOR I=1 TO N
1010 IF S$=T$(I) THEN GOTO 2000
1020 NEXT I
```

where line 1010 terminates the FOR loop when (and if) the value in S$ is found in T$(I). This is not only bad programming style, it will eventually cause the program to crash. This is because each time a FOR loop is begun an entry is created in an area of memory known as the *FOR stack*. This entry is only removed when the normal exit is made from the FOR

loop. Leaving the loop early, as in the above example, does not remove the entry and if this is repeated the FOR stack becomes so clogged with data on incomplete FOR loops that there is no room for any more hence the crash. The correct way to leave a FOR loop without crashing the program is to set the index variable to the final value:

```
1000 FOR 1=1 TO N
1010 IF S$=T$(I) THEN J=I:I=N
1020 NEXT I
```

Notice that to keep track of where the entry was found the value of I has to be saved in J before it is set to N so terminating the loop. However, as far as programming style is concerned there is no good way of exiting a FOR loop early! The whole point of an enumeration loop is that it is executed a given number of times and does not exit on any other condition. If you find that you are always exiting FOR loops before they are complete the chances are that you are confusing the purpose of enumeration and conditional loops.

It has already been stated that a FOR loop has only one implicit exit point. The remaining question is where is it placed? There are only two practical possibilities at the end of the loop (giving an 'until' form of the enumeration loop) and at the beginning (giving a 'while' form of the enumeration loop). The difference in practice is only the number of times that the FOR loop will be executed. If the exit point is at the end the loop must be carried out at least once before the test for exit is made. On the other hand if the exit point is at the start, the exit can occur before the loop is even executed once. The trouble is that different versions of BASIC use different forms of the FOR loop some use the 'until' FOR and others use the 'while' FOR. BBC BASIC uses the 'until' version of the FOR loop and so the loop is executed at least once to reach the test. For example:

```
10 FOR I=2 TO 1
20 PRINT I
30 NEXT I
```

will print 2 on the BBC Micro's screen but on some versions of BASIC the loop will not be executed even once. As long as you know which type of FOR loop you are working with there is no problem. The difficulties arise when switching between different versions of BASIC.

Finally the FOR loop has one extra, and often dangerous, refinement the STEP statement As the value of the index variable is often involved in calculations the BASIC command STEP can be used to alter the amount by which the index variable is increased each time through the loop. For example:

```
10 FOR I=10 TO 1 STEP -1
20 PRINT I
30 NEXT I
```

is an enumeration loop which is carried out ten times, but the value of the index variable decreases by 1 each time through the loop. Integer values of STEP size are fairly easy to deal with and the meaning of the resulting loop is usually quite clear. The trouble really begins when fractional values are used for the STEP size. For example you might like to try to work out the last number printed by the following FOR loop when it is run on the BBC Micro:

```
10 FOR I=0 TO 1 STEP .1
20 PRINT I
30 NEXT I
```

The answer is not 1! Ambiguities such as this are common when the STEP size is a fraction and are a consequence of the way fractions are represented in binary. Even so, occasionally it has to be admitted that a fractional STEP size is the clearest and simplest way of achieving something.

A family tree of loops

There is no doubt that the number of loop types is large enough to be confusing at first acquaintance. At this point in the discussion it is worth summarising the information that has been presented concerning loops in the form of a diagram see Fig. 2.5.
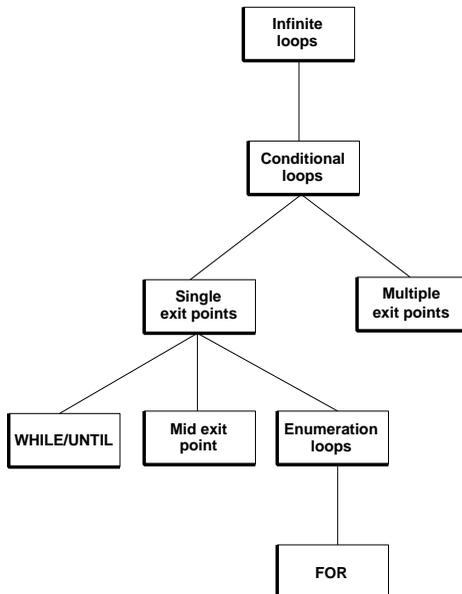
Fig. 2.5. Loops

Constructing programs sequencing and nesting

The basic idea that lies behind structured programming is that programs are constructed using nothing but the structuring elements that have been introduced in this chapter. So far nothing has been said about how these structuring elements can be put together in combination. An important factor Up be noted when combining structuring elements is that their pattern of flow of control should not be altered. In particular it is important that a GOTO instruction doesn't transfer control into the body of a structuring element. This rules out, for example, jumping into loops, etc. It is surprising to discover just how few and simple the possibilities are. In fact there are only two sequencing and nesting. Sequencing corresponds to the most obvious way of putting structuring elements together for they are simply arranged one after the other i.e. in sequence. For example, a loop may be followed by an IF statement and then another loop and so on. Sequencing is rather like writing each of the structuring elements on a card and making up a program by laying the cards out in the desired order.

   The second way of combining structuring elements - nesting - is more complicated, but very natural and familiar to all programmers. Nesting is where one structuring element occurs as part of one of the 'lists of BASIC statements' that have been present in each of the definitions given so far. The most familiar example of nesting is when a loop occurs within another loop as in:

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 PRINT I,J
40 NEXT J
50 NEXT I
```

where the FOR loop at lines 20 to 40 is completely contained, or nested, within the FOR loop at lines l0 to 50, see Fig. 2.6. Notice that the phrase 'completely contained' is implied in the very idea of a structuring element.

   If you were somehow only partially to include one structuring element within another the result would be a brand new pattern of flow of control and this would go against the very idea of using only a small and fixed number of ways of altering the flow of control. Although loops are the most often encountered examples of nesting, it is possible to have nested IF statements. Indeed, the example given in the earlier section on multiple selection used nested IFs. Nested IFs are generally difficult to understand and should be avoided or converted to a single IF. For example the nesting:
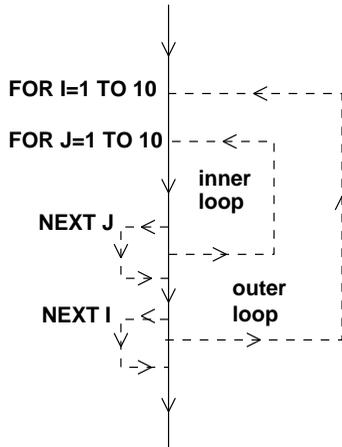
Fig. 2.6. Nesting.

    IF condition 1 THEN IF condition 2
    THEN list of BASIC statements

is better written as:

    IF condition 1 AND condition 2
    THEN list of BASIC statements

However, there is one case where nested IFs are important and this is where 'condition 2' would cause an error if it was evaluated when 'condition 1' was false.

Freedom, efficiency and structure

The account of structured programming given here is nowhere near as restrictive as that found in most books that advocate structured programming as a method of producing easy to understand, bug-free programs. Hard line structured programming generally forbids the use of GOTO and only allows the use of the until, while and FOR loops. However, the whole point of structured programming is to produce a simple and recognisable flow of control and, while this is an important objective, overall program simplicity should never be sacrificed to the application of nothing but the standard programming elements. For example, there are some occasions when the ON . . . GOSUB statement is so right for the job that it is better to use it than avoid it. The rules about which structuring elements to use are only rules to the unskilled programmer - the advanced programmer exercises judgement. Another problem that often arises is the question of efficiency. Early programmers

were taught that the main objective of programming was to write programs that work fast and use little memory. These days, with hardware being so much better and high level languages improving all the time, efficiency is much less important. Now it is the cost and reliability of software that matters and structured programming reduces cost and increases reliability.

Chapter Three
Using Procedures and Functions

The previous chapter considered the problem of keeping the flow of control simple but even this is not enough to make a large program easy to write. The key to the problem of writing large programs as easily as small programs has already been partially introduced in the form of 'nesting' structuring elements. In this chapter it is developed further into a second component of programming method that goes naturally with structured programming.

The small programs inside large programs

The best way to write a large program is as a number of small programs! This seems like an obvious approach in the tradition of 'divide and conquer' but there are a large number of interesting questions concerning exactly how the divisions should be arrived at and then how these smaller program should be assembled into one large bug-free program.

   If you look at a finished program, no matter how it has been written, you should be able to identify groups of statements that carry out particular tasks. For example a program might have an initialisation and data input section, followed by a section that performs calculations and finally a section that prints results. As the final program will have this sectional or 'modular' structure no matter how you go about writing it, then it seems like a good idea to try to recognise this structure and make use of it while constructing the program. This is the philosophy that lies behind 'modular programming' . The benefit from breaking down the construction of a large program into a number of smaller modules is easy to see. Each module can be treated as if it was a program in its own right and, of course, the module will be a smaller program than the one needed to solve the initial problem, and smaller programs are easier to write! If one of the modules turns out to be rather too large then the process can be repeated and it can be divided up into a number of smaller modules and so on. An important factor in the success of this repeated division strategy is

that each module really can be treated as a program in its own right and in isolation from the other modules.

The degree to which real modules can be treated in this way will be discussed later. For the time being all that will be said is that there are considerable advantages in implementing modules as BBC BASIC procedures. This step of identifying modules with procedures is useful but not essential. It would, for example, be possible to write sections of program as subroutines to do particular tasks or even just as lists of BASIC statements marked out by REMs, but either of these methods would be severely under-using the facilities offered by BBC BASIC

Stepwise refinement - or starting at the top

To say that a program should be made up from modules is one thing; to identify those modules when all you have is an idea of what the program should do is quite another. Stepwise refinement is an excellent technique for solving all manner of problems, not just those that demand programs as solutions. One of the best features of stepwise refinement is that even if you cannot solve the whole problem it is possible to make a start and quickly identify what the real difficulties are. It also automatically takes account of the natural modular structure of a program and uses it to advantage.

The best way to explain stepwise refinement is via a simple example. Consider the problem of producing a program that automatically sets and marks questions in simple arithmetic.

The statement of the problem is easy to understand. The program is intended to give routine practice in simple arithmetic, but unless you have written such a program before, or are exceptionally talented, you will only have a vague idea of what it should be like. If, as is normal, the complete program isn't inside your head the traditional advice is to sit down with plenty of paper and plan the program using flow charts or some other method. This is a very wasteful way of designing a program because each time you change your mind about how it should work you have to throw away a lot that is good and works, along with the unsatisfactory bits. Stepwise refinement doesn't suffer from this defect.

After very little thought it is clear that the arithmetic test program must first construct a question, then ask it, mark it and finally repeat the whole procedure until it is finished. If you examine this rough description carefully you will find that it is not so rough after all. In fact, if each of the parts of the description are interpreted as procedures, then you almost have a program. In other words after very little thought about the problem you can write:

```
10 REM arithmetic test
20 REPEAT
30 PROCset_question
```

```
    40 PROCask_question
    50 PROCmark_question
    60 UNTIL finished
```

which is perfectly good BBC BASIC apart from the fact that the PROCs do not exist as yet and the condition 'finished' in the UNTIL statement is not fully determined. You can even avoid the "No such FN/PROC at line x" error messages that trying to run this program would produce by including:

```
    1000 DEF PROCset_question
    1010 ENDPROC

    2000 DEF PROCask_question
    2010 ENDPROC

    3000 DEF PROCmark_question
    3010 ENDPROC
```

With these 'dummy' procedures present you can even begin the testing of your first attempt at the program!

The rough program given above is the first stage in the stepwise refinement process. Using it as a starting point, the second stage of stepwise refinement is to fill in the details of the dummy procedures. In this sense the dummy procedures introduced in the first stage are promises to write procedures. The second stage of refinement can also apply the same technique of using dummy procedures as promises to write the procedures needed to complete the procedures introduced in the first stage. So, for example, the devejopment of PROCset_question might produce something like:

```
    1000 DEF PROCset_question
    1010 Q$=""
    1020 PROCchoose_op
    1030 NUMl=RND(99)
    1040 NUM2=RND(99)
    1050 Q$=STR$(NUM1)+O$+STR$(NUM2)
    1060 PROCfind_ans
    1070 ENDPROC
```

Where the dummy procedure PROCchoose_op returns one of '+' ,'-', '*' or '/' in the string OS and PROCfind_ans works out the answer to the arithmetic expression held in Q$. Notice that the two dummy procedures are produced because at this stage we don't know how to go about generating one of the four arithmetic operators in O$ nor how to work out the answer to an arithmetic expression built up in Q$. These are problems that will be solved at the next stage of refinement.

This is all there is to stepwise refinement: at each stage you write as much of the program as you can using dummy procedures to put off solving any large task that has to be performed until the next stage. In practice it is sometimes better to leave a task to the next stage even when you know how to implement it at the current stage. This is the case when the task is similar to other tasks that are being deferred to the next stage, so that the resulting procedure 'fits into' the overall structure of the program. For example, some programmers would have deferred the construction of the two random numbers NUM1 and NUM2 to a procedure at the next stage (PROCnumbers say) because this fits in with the way a procedure PROCchoose_op is used to generate the operator part of the expression. There are no hard and fast rules about when to defer a task to the next level of refinement; only a feeling for the overall program can help you. However, it is worth saying that in most cases you should err on the side of using too many rather than too few procedures and steps. The advice often given concerning working out simple arithmetic holds for stepwise refinement never do at this stage what you can put off until the next!

Stepwise refinement as nesting

In the last chapter the idea of 'nesting' one structuring element inside another was introduced as a way of constructing programs. In many ways stepwise refinement is a method of program construction that proceeds by nesting one program inside another! For example if you look at the program produced in the first step in the last section (i.e. lines I0to 60) you can see that if you treat the procedure calls as single BASIC instructions then this is a very simple but complete BASIC program. Its structure is clearly one of our simple structuring elements an until loop and therefore it is very easy to understand. However if you look at it a little closer you will see that some of the BASIC statements are themselves complete programs with a structure. For example, line 30, the call to PROCset_question, is a complete program in the form of the simplest of ail structuring element the default flow of control. In practice, the modules that make up a program will contain a number of structuring elements but never so many that it is difficult to see what is happening.

It is useful to think of this nesting of modules as a number of levels present in the program. The top level is often called the 'main program' and this calls modules or procedures in the next level down and so on. For example the arithmetic test program could be described as:

```
level 1                       main program


level 2          set_question         ask_question        mark_question


level 3     choose_op      fine_ans      procs used by      procs used by
                                         ask_question       mark_question


level 4          procedures called by procedures level 3
```

and so on
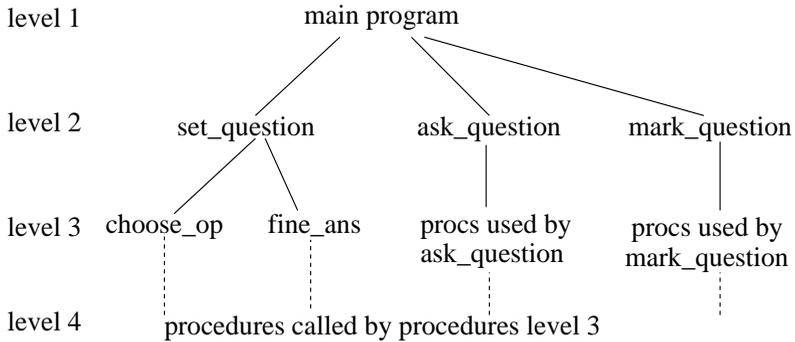
When you first look at a program constructed in this way all that you need
see to understand its overall working is the main program. As you look
closer and more carefully you work your way down through the levels,
understanding more and more of the detail of how the program works.
This is an exact reflection of the processes that were used to write the
program and this also explains why stepwise refinement is referred to as a
'top down' method.

In practice the separation between the levels of a program is not
always so clear. For example, a procedure in a lower level may be called
by a number of procedures in higher levels. In the case of the arithmetic
program PROCfind_ans might also be called by PROCset_question and
PROCmark_question or by any procedure that needs to know the answer
to the problem being set. It is better to try to build the levels in such a way
that procedures in each level onjy call procedures at the very next level
down but this is often impractical and existing procedures are often re-
used. Indeed, by reducing the total number of lines in a program, re-using
procedures is a simplifying action in its own right. There is nothing to stop
a procedure from calling one at a higher !eve] but this is a very dangerous
and sloppy programming practice. It not only destroys the clear nesting
structure of the program, it introduces the risk of the first procedure being
recalled at a later stage. For example, if PROCfind_ans for some reason
called PROCset_question then eventually PROCfind_ans would be called
again before the first call to it was completed! This recalling of a
procedure is a roundabout way of a procedure calling itself! In BBC
BASIC this is a perfectly permissible thing to do and it is called *recursion*.
Recursion is still a controversial subject in computer science; some
programmers like it and claim that it is a simplifying method, others just
find it confusing and difficult. Whatever your opinion of recursion it is
important that you know when you are using it, and the danger referred to
in a procedure calling a procedure at a higher level is simply that you
might produce a recursive program without noticing! (For more
information about recursion see Chapter Eleven of *The Complete
Programmer*.)

Module interaction - local variables and parameters

One of the advantages of using separate modules to construct a program is that each module can be treated as a program in its own right. However, as pointed out earlier it is only possible to work on a module in isolation if it really doesn't have any effect on any other modules. If you think about this statement for a moment you will realise that it is not quite accurate. If a module is going to work together with others to make up a program then it is essential that data is passed to it and returned from it. In other words, to produce a program the modules have to communicate. What is really required is that modules should only interact with one another in clear and strictly controlled ways.

As an example of how uncontrolled interaction can cause problems, consider the traditional BASIC subroutine. Any variable used in a BASIC subroutine is available to any other part of the entire program. This is fine as long as each subroutine uses its own names for variables that it doesn't want any other subroutine to use. In practice it is very difficult to keep track of a large number of variable names and when two subroutines end up using the same variable for different purposes the resulting bug is often very difficult to find.

The point is that modules shouldn't interact in ways that you never intended. Such accidental interactions are usually called the 'side effects' of a module. Fortunately, the BBC BASIC procedure (and, as explained later, function) has a perfect mechanism for ensuring that there are no side effects. Any variable that is named in a LOCAL statement at the start of a procedure will be the sole 'property' of the module. If an identically named variable already exists before the procedure is called then its old value is saved and the variable is re-initialised to zero. If the variable doesn't already exist then it is created and initialised to zero. Either way the procedure gets a nice new version of the variable that it can use without interfering with any procedures at a higher level. When the procedure has finished, the old values of all the local variables are restored.

Variables named in LOCAL statements are, not unreasonably, called local variables. By contrast any variable that is not named in a LOCAL statement is referred to as a *global* variable. Such a variable is global in the sense that its value can be used and altered by any part of the program. To make sure that procedures do not interact with one another all that is necessary is that every variable which a procedure uses should be named in a LOCAL statement. Of course, the strict application of this rule would eliminate all types of interaction including desirable communication between procedures.

A few minutes' thought indicates that the variables a procedure uses fall into four categories:

(1) Variables that pass data to the module, the so-called 'input' variables.
(2) Variables that pass data from the module, the so-called 'output' variables'.

(3) Variables that pass data into and out of the module, the so-called 'input/output variables'.

(4) Variables that are used within the module and neither pass data in or out of the procedure, the so-called 'internal' variables.

It is clear that internal variables should all be named in LOCAL statements as they are only of interest to the procedure that uses them. Input variables correspond to BBC BASIC% definition of 'parameters'. One way to think about a parameter is as a local variable that is initialised to the value stored in another variable before the procedure is called. For example in the procedure:

```
1000 DEF PROCspace(LINES)
1010 LOCAL I
1020 FOR I=1 TO LINES
1030 PRINT
1040 NEXT I
1050 ENDPROC
```

both I and LINES are local variables, that is changing their values within the procedure will not change the values stored in any variables of the same name in procedures at higher levels. However, when the procedure is called, I will be initialised to zero but LINES will be initialised to the value of the variable or expression used in between the brackets of the call. For example:

```
PROCspace(l0)
```

will cause LINES to be initialised to 10 and:

```
PROCspace(A*l0+32)
```

causes LINES to be initialised to the result of A*10+32. Notice that in BBC BASIC because parameters are local variables, they cannot be used to pass data back to the calling program. In this sense they are 'input parameters'. Some languages provide in addition both 'output' and 'input} output' parameters but unfortunately these are absent from BBC BASIC. In addition, arrays cannot be named in LOCAL statements and cannot be passed as parameters. However, this doesn't mean that you cannot use individual array elements, such as A(I) as values to be passed to procedures, only that an entire array cannot be local.

All this raises the problem of how to pass data out of a procedure. Unfortunately, there is no satisfactory answer to this problem. The only practical way of passing values back is to use global variables with the resulting risk of producing unwanted side effects. There is also no alternative if you have to pass arrays in or out of a procedure arrays are always global variables.

Putting all this together produces the following guidelines:

(1) All simple data, real numbers, integers and strings should be passed into a procedure by using parameters.

(2) Arrays have to be passed into and out of procedures as global variables.

(3) Simple data has to be passed out of procedures by using global variables.

(4) All simple variables that are not being used to pass data out of a procedure should be named in the LOCAL statement.

This scheme is the best that can be achieved using BBC BASIC and after using it for a while you should find that its only serious defect is the difficulty in making plain which variables *are* being used to pass data back to the procedure that initially called it.

Functions

BBC BASIC does supply one method of passing a single value back without using global variables the user-defined function. A function can return a single numeric or string value without using a variable because it can be incorporated into an arithmetic or string expression. For example, you could write a procedure to square a number and return the result in the global variable SQ but it is much more sensible to use a function:

```
1000 DEF FNsq(X)
1010 =X*X
```

The last line of the function can be thought of as assigning a value to the function's name and terminating the function. An 'empty' assignment of this sort anywhere in a function will set the value that it returns and terminate it. This convention leads to some very odd looking lines such as:

```
IF A=0 THEN 0
IF A>0 THEN =1 ELSE =-1
```

which are perfectly correct as part of a function definition and give rise to a user-defined version of the supplied function SON. As an example of a string function consider:

```
1000 DEF FNinsert(S$,I$,I)
1010 IF I<0 THEN =S$
1020 LEFT$(S$,l)+I$+MID$(S$,I+1)
```

which will insert the string I$ into the string S$ following the Ith letter.
User-defined functions can define their own local variables (using the

LOCAL statement). They can also use global variables to return more than one result but this is a feature that definitely should not be used. The importance of functions is that they only return one result - a function that uses global variables to return more than one result should be turned into a procedure! The reason why this one-result rule is so important is the way that functions are used within arithmetic and string expressions. For example FNsq(X) can be used in arithmetic expressions such as:

A= FNsq(B)*3

or even:

A=SQR(FNsq(B)+FNsq(C))

and most programmers do not expect expressions, or functions for that matter, to produce any side effects. In the same way, it is possible to include PRINT and INPUT statements within functions but it takes a very clever programmer to suspect an expression like:

A=FNsq(B)

of producing output or requiring input. In other words, data should only be passed to functions by parameters and they should only pass one result back in the standard manner. The main characteristic of a function is that it can be used as part of an expression and as such it should never cause an expression to do anything unpredictable.

REMs, indentations and line numbering schemes

It is often said that the use of the REM statement to explain what is happening in a program is the most important part of making a program understandable. While REMs and the comments they provide certainly do add to the readability of a program they are by no means the most important fact on A badly written program is still impossible to understand even when the REM statements outnumber the other statements many times over! On the other hand, a well structured modular program is almost self-explanatory if meaningful names are used for variables, procedures and functions. Of course, it is often the case that the occasional REM statement helps but, as most programmers admit, no matter how many good intentions you start olf with, REMs soon become a chore that is forgotten during program development, The general ru]e seems to be, if it is extra trouble then it tends not to be used! Structured modular programming isn't extra trouble; it is part of the process of program construction and it wen makes it easier! Of course, if you do find that you have no choice but to use a complicated trick then it is absolutely essential to document it within the program by using REMs, but otherwise a well written program should be almost self-documenting.

Another method of improving the readability of programs is to use a line numbering scheme of the sort that has been used without comment in all of the examples. That is the main program occupies lines 1 to 999, thus subsequent procedures occupy line numbers in the thousands. It is remarkably difficult to keep to a numbering scheme of this sort. As indicated in the previous chapter, line numbers are something that BASIC would be better without. Given that they are an evil that nothing can be done about, then using a numbering scheme is the best that can be done. However, the first use of RENUMBER destroys any scheme and so it really isn't worth worrying about inventing anything elaborate!

A very useful facility for making programs easier to read is indenting. Following LISTO 6 all FOR and UNTIL loops will be indented by two spaces in listings. This makes it easier to spot the start and end of these two structuring elements and is well worth using. However manual indenting can also be applied to other loops and blank lines can be inserted before and after procedure and function definitions either by using empty REM statements or a line that contains nothing but a colon.

Whatever you do to improve the look and readability of your programs, nothing can replace the reward of using a natural structuring method combined with the use of modules.

Chapter Four
Structured Assembler

Chapters Two and Three discussed some of the ideas involved in structured programming as applied to BBC BASIC. If you followed the main points of the argument for the use of structured programming you will realise that its objectives, namely to produce programs with a clear flow of control structure, apply to any computer language including assembler! This is not such a widely held belief as you might expect, indeed some programmers claim that it isn't possible to write well structured programs in a language as 'primitive' as BASIC, let alone assembler! The question of whether modular programming and stepwise refinement are at all useful in assembler is less controversial. Nearly every assembly language programmer quickly learns the value of subroutines and, in a language where each instruction achieves so little, stepwise refinement is an ideal way of building up large programs without getting into a position where 'the trees obscure the view of the wood'. In short, all of the programming methods that you should apply to BBC BASIC should also be applied to assembly language and any other computer language that you might use.

The first part of this chapter looks in detail at the natural structuring elements in assembler and at ways of constructing modules from subroutines. These ideas apply to 6502 assembler on any machine, not just the BBC Micro. To make things more specific, the second half of the chapter examines more advanced ways of using the BBC Micro's assembler to convert 6502 assembly language into machine code. The BBC Micro's assembler may appear at first sight to be a very simple assembler but, because it works in such close contact with BBC BASIC, it can be used in ways that are jess than obvious. In particular, it is possible to introduce better ways of defining data constants, conditional assembly and macros without any difficulty.

The natural structure of assembler

It is very important that an assembly language program has a simple flow of control. Each assembly language instruction performs so little work

that it can take many instructions to complete a reasonable task and even more to complete an entire program. Thus assembly language programs are often large and intrinsically difficult to understand. A spaghetti assembly language program is a nightmare worse than anything that can be produced in BASIC but it is a sad fact that most assembly language programs are more or less spaghetti! It is almost as if otherwise skilful and careful programmers throw every thought of good structure away as soon as assembly language is contemplated. The attitude seems to be that writing a program in assembly language is difficult enough without worrying about anything else. Of course, it is only difficult if you don't use the familiar programming methods!

The problem in applying structured programming to assembler s;eems to be that the natural structuring elements of IF . . . THEN . . . ELSE, etc., are just not present. This is not surprising as IF . . . THEN . . . ELSE is a natural structuring element of BBC BASIC there are even some versions of BASIC in which it is not present! Before it is possible to write structured assembler it is necessary to identify *its* natural structure. As in the case of BASIC there are only two categories of structure selection and looping.

Assembler selection

The fundamental commands of 6502 selection, or conditional execution, are the *conditional branches*. The usual situation is that a calculation, some test, or the simple loading or storing of a register is used to set the condition codes in the status register. Following this a conditional branch (for example, BCC Branch Carry Clear) can be used to divert the flow of control depending on the setting of the condition code bits. You should be able to see that this is very similar to the

IF condition THEN GOTO x

statement of BASIC. The 'condition' corresponds to the operations that set the condition code bits and the 'THEN GOTO x' part corresponds to the conditional branch instruction. The correspondence is not exact because more than one condition code bit may be set at a time and exactly what is being tested for also depends on the particular conditional branch that is used. For example, the BASIC instruction:

IF A= 0 THEN GOTO x

where x is a line number, is equivalent to:

CMP #0
BEQ x

where x is a label in 6502 assembler but the instruction CMP #0 not only

sets or clears the Z flag it a]so sets or clears the N and C flags. Thus:

```
CMP #0
 BMI x
```

is equivalent to:

IF A<0 THEN GOTO x

In other words, the condition that causes the branch to be taken cannot be determined solely from the operations that set the condition codes, it also depends on which conditional branch is used. Thus the form of all assembly language conditionals is:

(1) one or more operations that set the condition codes followed by:
(2) a conditional branch that branches depending on the state of one of the condition code bits.

This still leaves the question of how the transfer of control should be used to select which part of the program is to be executed? In assembler the fundamental structuring element for conditional execution is the 'conditional skip'.

```
            . . .
            operations that set the condition codes
            B** SKIP
            . . .
            list of assembly language instructions
            . . .
    .SKIP   rest of program
            . . .
```

where B** is a particular conditional branch that transfers control to .SKIP when the desired condition is true. (All the conditional branch instructions apart from BEQ and BNE are of the form BfS or BfC where 'f' is the name of the condition code bit or flag to be tested for Set or Clear.) Notice that the 'list of assembly language instructions' is only carried out when the condition is false, otherwise it is 'skipped'. As an example of a conditional skip consider:

```
            CPY #0
            BEQ SKIP
            DEY
    .SKIP     rest of program
```

Which will decrement the Y register only if it is not already zero. In other words, the DEY is 'skipped' if Y is zero. You should be able to see that this conditional skip is similar to:

```
            IF Y=0 THEN GOTO x
            Y=Y-1
    x     rest of program
```

in BASIC where x is a line number.

The conditional skip is the fundamental structuring element for all assembly language conditional execution. This might seem rather surprising as in BASIC the IF. . .THEN. . .ELSE at least makes it possible to select between two alternatives. In assembler even the choice between two sections of program is built up from a choice to skip one of them! For example to execute 'list I' when 'condition' is true and 'list 2' if it is false you would use:

```
            . . .
            operations that set the condition codes
            B** LIST1
            . . .
            list 2
            . . .
            JMP REST
    .LIST1  . . .
            list1
            . . .
    .REST   rest of program
```

which should be compared to the example of how IF. . .THEN. . .ELSE can be constructed from IF. . .THEN GOTO given in Chapter Two. A particular example may help to clear up any difficulties. The following assembly language:

```
            CPY #0
            BMI MINUS
            DEY
            JMP REST
    .MINUS  INY
    .REST       rest of program
```

will decrement Y if it is positive or zero, and increment it if it is negative. Thus, it is equivalent to the BASIC:

    IF Y>=0 THEN Y=Y-1 ELSE Y=Y+1

Once you have seen how the conditional skip can be used to select between two alternatives it is easy to use it to select between any number of alternatives. Fortunately situations that need a great many choices happen very rarely in assembly language programming because such problems are best dealt with using BASIC.

Assembly language loops

The subject of assembly language loops is identical to that of BASIC loops except of course that there are no explicit FOR . . . NEXT and REPEAT . . . UNTIL commands. All assembly language loops have to be

made up from scratch using a jump instruction to form an infinite loop and a conditional branch to form the exit point. For example:

```
          LDY #0
.LOOP     INY
          CPY #l0
          BEQ EXIT
          JMP LOOP
.EXIT     rest of program
```

is a conditional loop with its exit point at the end and so it ts the assembly language equivalent of the until loop, Y ou may notice the slight inefficiency in havjng the conditional branch immediately followed by the JMP instruction and indeed the assembly language until loop is more normally implemented as:

```
          LDY #0
.LOOP     INY
          CPY #10
          BNE LOOP
          rest of progranm
```

All of the classification of loops and guidelines for using them given in Chapter Two apply to assembly language loops. In particular try to:

(1) use only one exit point per loop,
(2) use only until and while loops,
(3) make the initial and final values of enumeration loops clear.

Modular assembly language

As well as using the natural structuring elements to put together an assembly language program it is very important to use modular stepwise refinement. The process of stepwise refinement is exactly the same for assembler as it is for BASIC; at each step the program is extended by filling in the details of dummy modules used in previous steps and introducing new dummy modules to defer things to the next step, This is easy once you have identified a method of constructing modules in assembler. Unfortunately there is no convenient way of forming modules analogous to the BBC BASIC procedure with its parameters and LOCAL variables. Assembler has only the JSR (Jump to Subroutine) and RTS (ReTurn from Subroutine) instructions and these are more like the BASIC GOSUB and RETURN than PROC and ENDPROC. In fact, using assembly language subroutines is exactly like writing modular programs in traditional BASIC using COS UB and RETURN. All the variables and registers in an assembly language program are global, in the sense that they can be used by any part of the program with the danger of creating

accidental side effects and difficult-to-find bugs. There is very little that can be done about this problem and it is something that makes the use of assembly language different from BBC BASIC. It is possible to keep the 6502's registers local to a particular subroutine by pushing them on the stack at the start of the subroutine and restoring them by pulling them off the stack at the end of the subroutine, but this is tedious and time-consuming. Equally the problem of passing data into and out of a subroutine is difficult to solve. There are many schemes that can be used. Some, for example, involve pushing data that is to be passed to a subroutine onto the stack before calling the subroutine. Similarly, data that is to be passed out of the subroutine is pushed onto the stack to be retrieved by the calling program. But, once again, such schemes are tedious and time-consuming. It is conventional for small quantities of data to be passed into and out of assembly language subroutines using the registers. Larger quantifies of data are either passed through known and fixed areas of memory or the address of the area of memory is passed in the X and Y registers. Whatever method is used it is important that it is made clear where the subroutine gets its data from, how it returns results and which memory locations it uses. Many examples of parameter passing in assembly language can be found in the applications programs in subsequent chapters.

Even though there are many disadvantages to the assembly language subroutine, it is no worse than the traditional BASIC subroutine and in both cases modular programming is still better than attempting to write large programs in one chunk. However, it is worth being aware of the increased danger of accidental side effects. Whenever an assembly language program, which was well behaved at earlier stages of stepwise refinement, suddenly starts to act in an apparently illogical manner you can be almost sure that the trouble stems from one subroutine altering variables used by another.

The role of comments

At the end of Chapter Three the idea of a well structured BBC BASIC program being almost self-documenting was introduced as a reason for not worrying too much about including hundreds of REM statements. You may think that a well structured assembly language program would share this property of self-documentation, but this is not so. While the flow of control and the subroutine nesting structure of assembler should be as clear as a well structured BBC BASIC program, the intent of the program isn't. BASIC is made up of almost self-explanatory statements such as PRINT A and IF A=1 THEN PRINT "ONE" and, as long as the flow of control is clear, it is usually not difficult to understand the program by analysing what is happening in each statement. The situation is assembly language is completely different as each instruction performs a very small part of the larger task. Without knowing what the larger task is the single

instruction often seems unconnected with anything going on in the program. Thus in assembler it is important to include comments that inform the reader of the overall idea of the action of each part of the program. Notice that this doesn't mean that you have to include comments on each instruction. Indeed, the most obvious comments in a program are the most irritating! For example, the comments in the following program section add nothing (except irritation):

```
LDA   NUM1  \ load the A register with NUM1
CLC         \ clear the carry flag
ADC   TEST  \ add TEST to the A register
and   so on
```

Whereas the single comment \A=NUM1+TEST would have said it all!

Using the assembler

All that has been said so far applies to using 6502 assembler on any machine and much of it applies to using any assembler! In this and subsequent sections we look more specifically at the way that the BBC Micro's built-in assembler can be used in some advanced ways, but first it is worth briefly going over how it works.

   To leave BBC BASIC and use the assembter all you have to do is enclose any assembly language statements in square brackets. The rule is that anything within square brackets will be treated as assembly language, anything outside square brackets is pure BASIC. (Notice that this implies that you cannot use BASIC statements within the square brackets that enclose a ptece of assembly language, but sec later). This is a very easy way of switching between BASIC and assembler but it is important to understand the very different ways that BASIC and assembler are processed. When the BASIC interpreter encounters a line of BASIC it carries out the instruction, but when the assembler meets a line of assembly language it doesn't carry out the instruction; it simply translates it to machine code. Of course, to be of any use this machine code must be carried out sooner or later but this is nothing to do with the assembler!

   When the assembler translates the assembly language it has to store it somewhere until it is needed. There are a number of places where the machine code can be stored and the most useful is in a *byte array*. A byte array is created by a special form of the DIM statement:

   DIM variable maximum_index

which creates an area in memory consisting of 'maximum index' + 1 bytes and stores the address of its first memory location in 'variable'. For example:

DIM CODE% 20

gives a byte array with 21 elements (i.e. element 0 to element 20) and stores the address of the first element in the integer variable CODE%. Notice that CODE% is a perfectly standard BASIC variable and can be used in expressions, etc., as usual. As the address of the first element of the array is stored in CODE% you can use the indirection operator '?' to store and retrieve data within the array. That is:

CODE%?I=X

will store X in the Ith element of the byte array and:

X=CODE%?I

will store the Ith element of the byte array in X. Combined with the indirection operators a byte array can be used for many things other than just storing machine code.

The second component involved in the storage of machine code by the assembler is one of the resident integer variables. The resident integer variables are, unlike all other BASIC variables, present whether you use them or not and are always stored in the same memory locations. The resident integer variables are named @%, A% to Z% and the value stored in P% is used by assembler as the address where the next item of machine code will be stored. In addition to this, every time the assembler stores an item of machine code it adds one to P% so making sure that each item of machine code is stored in a new memory location. By setting P% to a particular address before using the assembler you can determine where the assembler stores the machine code that it produces. Thus, to make the assembler store the machine code that it produces in a byte array all that you have to do is set P% equal to the start address of the array before entering the assembler. For example:

```
10 DIM CODE% 20
20 P%=CODE%
30[
40 LDA #65
50 JSR &FFE3
60 RTS
70 ]
```

Line 10 creates a byte array with 21 elements and places the address of the first element in the variable CODE%. Line 20 stores this start address in P% so that the assembler will start storing machine code in the byte array.

Lines 30 to 70 form a short assembly language program that prints the

letter A on the screen and then returns to BASIC. However, if you run the program you will only see a listing of the machine code program looking something like:

```
1951 A9 41      LDA #65
1953 20 E3 FF   JSR &FFE3
1956 60         RTS
```

If you recall that the assembler only converts 6502 assembly language to machine code this isn't at all surprising. If you look at the first line of the output then the first number, 195 I, is the address where the first item of machine code, &A9, will be stored. The second item, &41, will automatically be stored at 1952 and so on to 1956. To carry out the machine code you have to use the BASIC CALL command:

    CALL 'address'

This transfers control from BASIC to a machine code routine starting at 'address'. Notice that CALL must be carried out by BASIC so the standard sequence in writing and using a machine code program is:

(1) in BASIC   set up space for machine code, etc.,
(2) in assembler   translate assembly language to machine code,
(3) in BASIC   use CALL to transfer control to the machine code.

If you don't want to use the machine code immediately then there is nothing to stop you from saving it on tape (using *SAVE) and then loading it (using *LOAD) at a later date and possibly into a different program. The only thing that you have to ensure is that the machine code is loaded back into the same area of memory that it was originally assembled to. This problem is taken up in Chapters Eight and Nine.
    To use the machine code in the last example all you have to do is add line

    80 CALL CODE%

Notice that CODE% still holds the start address of the byte array, and hence of the machine code, whereas P% at this point holds the address of the memory location where the next byte of machine code would be stored. Now running the whole program not only results in the conversion of the assembly language to machine code but also in the execution of the machine code.

Labels and variables

So far the subject of labels has been completely ignored. In fact the BBC assembler has a very interesting way of handling labels. Although you cannot use BASIC commands within the assembler you can use BASIC

expressions! For example, instead of writing LDA #65 to load the ASCII code for "A" into the A register, you can use LDA #ASC(" A"). In the same way you can use an expression in the address field of an instruction. So you could write JMP 2*3+6, which would cause a jump to address 12, or an expression like JMP CALL%+ 10, which would cause a jump to the address given by adding 10 to the contents of the variable CALL%. There are many other ways in which this powerful idea can be applied and one of the main problems is seeing which of the many ways is useful! The key point to remember is:

> You can use any valid BASIC expression, including defined variables, anywhere that the assembler would expect data or an address as part of an instruction.

The example 'print A' program given in the last section can now be rewritten in a more readable form as:

```
10 DIM CODE%
20 P% CODE%
30 OUT%=&FFE3
40 [
50 LDA #ASC("A")
60 JSR OUT%
70 RTS
80 ]
90 CALL CODE%
```

Apart from the use of ASC(" A") already described, notice the use of OUT%, defined in line 30, gives the address to which the JSR will transfer control. In general, if you have any fixed subroutines or memory locations within an assembly language program it is better to define variables with the correct values and appropriate names within BASIC and then use these variables within the program. If you follow this simple rule your programs will be easier to understand and easier to modify.

If you are familiar with any other assemblers you will realise that the command JSR OUT% is using a normal BASIC variable as if it was a label. This is not unusual. After all a label is simply a variable that is used to hold an address. What is unusual is the fact that BBC assembler labels are also BASIC variables and this is one of the reasons that this simple assembler is so powerful!

The fact that assembler labels and BASIC variables are one and the same thing also applies to labels defined within an assembly language program. Using a label preceded by a full stop causes the assembler to store the current value of P% in it. For example, in the program:

```
10 DIM CODE%
20 P%=CODE%
```

```
30 OUT%=&FFE3
40 [
50              LDA #ASC("A")
60 .LOOP%   JSR OUT%
70              JMP LOOP%
80 ]
90 CALL CODE%
```

the variable/label LOOP% defined in line 60 is to contain the address of the JSR OUT% instruction and this is used in line 70 to jump back to this instruction repeatedly. (Of course, when the machine code is used by line 90 the result is that the infinite loop fills the screen with letter As.) It is important to notice that although LOOP% was set to a particular value by the assembler, it is still a BASIC variable and, once back inside BASIC, it can be used just like any other variable. So you could add 85 PRINT LOOP% to find out the address of the JSR instruction in fine 60.

Two-pass assembly

The only outstanding problem with using labels is related to the forward jump problem described in Chapter Two. Consider for example:

```
l0 DIM CODE%
20 P%=CODE%
30 [
40        LDA #0
50        BEQ EXIT%,
60        LDA #0
70 .EXIT  RTS
80 ]
```

The program itself doesn't do anything useful, it loads the A register with zero then jumps to EXIT% if the result of the load was zero (it always is!). However, the BBC assembler gives an error message if you run it. It is not difficult to see the reason for this error message. When the assembler reaches line 50 it needs to know the value of EXIT% but unfortunately EXIT% has not been defined and will not be defined until line 70. This is called the *forward reference* problem and the traditional solution is to use a *two-pass assembler*. A two-pass assembler, as its name suggests, takes two 'looks' at the program. The first time through it picks up all the definitions of the labels and ignores any errors. The second time through it uses the values of the labels defined in the first pass to assemble the program correctly any errors at this stage are real!

The situation with the BBC assembler seems hopeless it's a simple one-pass assembler. However this is where its close association with BBC

BASIC comes to the rescue for the first but not the last time. There is an assembler instruction called OPT which can be used to suppress error messages and program listings in the following way:

n	OPT n
0	ignore errors and don't produce a listing
1	ignore errors but produce a listing
2	report errors but don't produce a listing
3	report errors and produce a listing

(The default value of OPT is OPT 3.) The definition of OPT 0 should enable you to understand how the two-pass method works. First the assembler is run over the program using OPT 0 to define all the variables used as labels. As these variables are BASIC variables they still exist and their values are unaltered after the assembler has finished. Then the assembler is run again using OPT 3, safe in the knowledge that all the labels are defined. The only problem is how to run the assembler a second time? We could write the entire assembly language program out twice but this would soon put a stop to any serious applications! The answer is, of course, to use a BASIC FOR loop to pass the assembler over the program twice! This may sound uninteresting, especially if you have been using it for some time as directed by the User Guide, but it is our first example of using BASIC to modify the way that the assembler works. The two-pass version of the forward reference example given above is:

```
10 DIM CODE%
20 FOR PASS=0 TO 3 STEP 3
30 P%=CODE%
40 [OPT PASS
50       LDA #0
60       BEQ EXIT%
70       LDA #0
80 .EXIT  RTS
90 ]
100 NEXT PASS
```

Even this simple example exhibits one subtlety. You must be very careful to set P% to the value stored in CODE% within the loop, otherwise the second pass over the program will store its machine code after the incorrect machine code produced by the first pass. Also notice the way that the assembler must be left before trying to execute the NEXT PASS instruction. Apart from very simple programs, this two-pass method of using the BBC assembler is the norm and for this reason it is all too easy to miss its implications for using BASIC with the assembler in other ways.

Conditional assembly

We now have all the information necessary to see how to turn the BBC assembler into a 'conditional assembler'. An example is the best way to explain exactly what a conditional assembler is. Suppose you had written a communications program that worked at two different speeds. In any given application you would only want the fast or the slow version of the program so you could use something like:

IF FAST=1 THEN [LDA #60:] ELSE [LDA #30:]

where FAST is a normal BASIC variable that can be set elsewhere in the program. Depending on the value of FAST either the first instruction will be assembled into the program or the second. The only thing that you have to be careful about is to remember to leave the assembler to execute the IF statement and then remember to return to it afterwards. Notice that the IF statement isn't part of the machine code that the assembler produces, it simply affects what machine code the assembler produces.

As another example, it is often a good idea to use regular jumps to a subroutine that prints the values stored in certain variables while you are debugging an assembly language program. Once you have the program working the obvious thing to do is to take out the debugging aids but this does cause something of a problem if you then go on to find another bug! The best solution is, of course, conditional assembly.

```
 10 TEST=1
   . . .
150 [ LDA #&40
160 STA IRQC%
165 ]: IF TEST=1 THEN [ JSR DEBUG%:]
166 [
170 LDA AUXC%
180 ORA #&C0
   . . .
```

This may look a little complicated but it is easy to follow. Line 165 first returns to BASIC to carry out the IF statement which will either re-enter the assembler to assemble JSR DEBUG% or just pass on to line 166 depending on the value in TEST. While debugging the program TEST would be set to I, otherwise it would be set to 0.

By now you should be able to see how to use IF . . . THEN and IF . . . THEN . . . ELSE to conditionally assemble any list of instructions into a program and so all that is left is to decide when the facility is useful and use it!

Macros

Now that we have seen how BASIC can be used to alter the way that the assembler works, the principles behind a macro are easy. Consider the fairly common problem of carrying out a shift or rotate instruction a few times. Normally this would be done by writing the command as many times as needed. For example, four arithmetic shift lefts would be:

```
100 ASL A
110 ASL A
120 ASL A
130 ASL A
```

Using the same idea as for the two-pass assembler we could generate four ASL A instructions by placing a single ASL A instruction inside a FOR loop, but this time not resetting the value of P% each time through. To take this idea one stage further we could write a PROC with an appropriate name and a parameter that would repeat the instruction a given number of times. That is:

```
1000 DEF PROCASL(N)
1010 LOCAL I
1020 FOR I=1 TO N
1030 [ ASL A:]
1040 NEXT I
1050 ENDPROC
```

and in the main program the four ASL A instructions would be produced by:

```
100 ]:PROCASL(4):[
```

which would first leave the assembler, then call the PROC and then, after generating the required machine code, re-enter the assembler. The FROG itself is fairly straightforward apart from the need to define I as LOCAL just in case it is used anywhere else.

Once you have seen this idea in operation it 'takes off to produce all sorts of useful macros. For example, you can use the parameters passed to the PROC within BASIC to control the assembler, i.e. as in the FOR loop or in conditional assembly, or you can use them within the assembler as part of expressions. You could write a general addition macro along the lines:

```
1000 DEF PROCADD(Nl,N2,ANS)
1010 [    CLC
1020      LDA N1
1030      ADC N2
1040      STA ANS
1050 ]
```

    1060 ENDPROC

To add two number stored at memory location DATA1 and DATA2 and
store the answer in DATA3 you would use:

    PROCADD(DATA1,DATA2,DATA3)

which would generate the correct machine code at whatever position it
was used within the main program.

If you have a standard assembly language operation then the
advantages of turning it into a macro are as great as turning it into a
subroutine (using JSR . . . RTS) and in this sense macros are just a useful
as part of modular programming as subroutines. However, as use of the
macro generates the necessary machine code every time it is used there
are none of the inefficiencies incurred with jumping to and from a
subroutine.

Data definition

Macros can help with the very common problem of initialisation of
assembly language variables. The recommended method of doing this is
to leave the assembler and use the indirection operators to store the data
values directly into memory. For example, if you want to initialise two
variables, NUM I and NUM2, each consisting of a single byte to 50 and
100 respectively then you would use:

    ]
    ?P%=50
    NUM1=P%
    P%=P%+1
    ?P%=100
    NUM2=P%
    P%=P%+1
    [

If you look at the way that these two variables are constructed you will be
able to see that each one involves three stages:

(1) The '?' indirection operator is used to store the value in the next free
memory location.
(2) The address in P% is stored in the variable's name for further use in
the program.
(3) The address in P% is incremented to point to the next free location,

This is a workable but not very clear method of defining data.

Fortunately it is not difficult to define three macros (using functions)
that make data definition very easy. For example the macro:

    DEF FNequb(VA%)

```
?P%r(VA% MOD 256)
P%=P%+1
=P%-1
```

will store a single byte value in the next free memory location and return its address, The MOD 256 in the second line simply ensures that the constant is in the range 0 to 255. Using this macro, the previous definitions of NUM1 and NUM2 can be written:

```
]
NUM1=FNequb(50)
NUM2=FNequb(100)
[
```

which is a great improvement! (The function's name stands for EQUal to Byte.)

In the same way the macro:

```
DEF FNequw(VA%)
?P%=(VA% MOD 256)
P%?1=(VA% DIV 256)
P%=P%+2
=P%-2
```

will store a two byte value in the next free memory location and return its address. (The function's name stands for EQUal to Word.) The macro:

```
DEF FNequs(S$)
$P%=S$
P%=P%+LEN(S$)+1
=P%-LEN(S$)-1
```

will store all of the characters in the string SS starting at the first free memory location and returns the address of the first character. (The function's name stands for EQUal to String.) Notice that this macro does not automatically append a carriage return to the string. To construct a string complete with carriage return use:

```
]
HELLO=FNequs(" HELLO FOLKS")
CA=FNequb(&0D)
```

Some further examples of the use of these macros are given in the assembly language program in later chapters. These macros serve the same function as the EQUB, EQUW, and EQUS assembler commands found in Electron BASIC and BASIC II.

Problems with macros - OPT, BASIC and local variables

There are three complications that occur when using macros. The first is due to the default setting of OPT being used whenever the assembler is entered unless explicitly changed by an OPT statement. Thus, if you leave the assembler using ']' and then enter it again with a macro using nothing but '[', the default of OPT 3 will cause any undefined labels on the first pass to cause an error and stop the assembly. The best solution would be to use some method of finding out what the last setting of OPT was and restore it when re-entering the assembler. However, this is not possible without PEEKing system locations that might change in later versions of BASIC. The only practical solution is always to re-enter the assembler with [OPT PASS'.

The second problem is that there is no way of suppressing errors caused by variables not being defined while in BASIC. For example, if a macro uses a variable that is not defined until later as a parameter in the assembly, the program will stop and report an undefined variable error on the first pass. The only solution to this problem is to ensure that all the variables used in the BASIC part of a macro are defined before the macro is called on the first pass.

The final problem is caused by local variables being set to zero on each pass. If a macro needs a label then the best way to ensure that no other part of the program uses the same label is to name it in a LOCAL statement. This causes no trouble if the label is defined on the first pass before it is used but the usual two pass mechanism of using the first pass to define labels that are then used in the second pass doesn't work! This is because the value that a local variable acquires on the first pass is lost when the LOCAL statement is encountered on the second pass! In other words, local variables can only be used to implement backward jumps. If you need to use a forward jump within a macro the best method is to use a constant rather than a label in the address field of a conditional branch. Thus BEQ 6 will jump forward six memory locations. These restrictions make the wnting of foolproof macros a little more difficult than you might imagine but then most useful macros are surprisingly simple.

Assembly language examples

No examples of large assembly language programs have been given in this chapter to illustrate structured assembly language programming. This is not an oversight, however, as Chapters Eight, Nine and Ten all include some long assembly language programs.

Chapter Five
Structuring Data

In Chapter Two the idea that a program is made up of two parts data and instructions concerning what to do with the data was introduced. Until this chapter all of the emphasis has been placed on the instruction part of program, with the assumption that the data type needed would be obvious or didn't really matter. This is very unrealistic in that how to represent something as data inside a program is not only crucial to the simplicity of the program, it is usually very difficult. The trouble is that while there are good ways of organising the construction of a program i.e. structured programming and stepwise refinement there is no really effective theory or method of selecting and creating data types.

The main difference between the instruction and data parts of program construction is that most of the decisions about the data have to be made before the program is written. Stepwise refinement simplifies program construction for the very reason that it puts off the making of important implementation decisions until they are really essential. In this way progress can be made without having to have all of the details and principles of the program determined beforehand. In the case of data, there is no equivalent stepwise approach, you have to consider what it is you are trying to do and identify what should be represented by data and how it should be implemented. For example, if you want a program that will act as a computerised telephone directory what should be represented by data is obvious: names and telephone numbers. But the way to implement this representation is not quite as clear as you might think. Most BASIC programmers would immediately choose a string array for 'name' and a numeric array for 'telephone number'. but as will be explained later the process of adding a new name and number is greatly simplified if a more complex type of data, a *linked list*, is used. This is an example of the general principle that the more suitable the data representation is, the simpler the resulting program.

When it comes to data, the only real advantage that an advanced programmer has over a beginner is having more possibilities to choose from. If you only know about arrays you can't even consider linked lists as a method of implementing a telephone directory! You may think that even

if you know about linked lists there would be no point in considering them because the BBC Micro doesn't support anything more complex than arrays. This is quite true but it misses the point that the BBC Micro has a good range of indirection operators that can be used to construct almost any type of data easily and efficiently. This chapter serves two purposes; firstly to introduce some of the more advanced data types and secondly to show how they can be implemented on the BBC Micro. The more difficult task of indicating how to go about selecting a data type is treated by way of examples in the later chapters of this book!

Simple data and structured data

There is one similarity between program and data construction they are both 'hierarchical'. When stepwise refinement is used a program can be seen as being made up of smaller programs or modules, which in turn are also made up of other, smaller modules and so on. In the same way complex forms of data are made up from smaller, simpler forms of data. For example, a numeric array is 'built up' from simple numeric variables, The method used to organise the simple numeric variables into an array is an example of a 'data structuring' method. In other words, taking a collection of simple numeric variables and 'putting them together' so that any one of them can be specified using a name of the form A(I) (or whatever the array and index are called) is a data structuring method.

Once you have identified a data structuring method there is nothing to stop you from appjying it to other simple data types. Thus applying array structuring to simple strings gives the familiar string array. The string array is so well known that it seems unnecessary to describe it in such a sophisticated way, but as will be explained later even the humble BASIC string is quite a complicated data type!

Not only can a data structuring method be applied to simple data, it can also be appiied to data that has already been organised in some way. For example, applying the array structuring method to an existing array gives an array of arrays better known as a two-dimensional array! That is, a two-dimensional array can be thought of as a one-dimensional array, each element of which is itself a one-dimensional array. This corresponds exactly to the usual way of thinking of a one-dimensional array as a 'row' of simple numeric variables and a two-dimensional array as a table consisting of 'rows' and 'columns'. If this sounds like a very complicated way of describing something that is simple, it is because BASIC doesn't really provide the range of data types and structuring methods that would make an example credible! What is important at the moment is that the idea of organising simple data types to produce new data types is clear.

What are the simplest forms of data that can be organised to make new types of data? It is convenient to distinguish two types of elemental data *static* and *dynamic data*.

The difference between the two is simply that an element of static data

is stored in a fixed amount of memory, whereas the amount required to store dynamic data can change while the program is running. For example, a simple numeric variable is an item of static data but a string is an item of dynamic data. Static data types are much easier to implement than dynamic types and, apart from strings, they are better known to BASIC programmers. In fact, static data types are so simple that there are broadly only two types, *scalars* and *reals*.

Scalars - integer, character and Boolean data

Scalars form the largest and most useful group of data types. Essentially a scalar is a whole number or an integer but often its real nature is well hidden. For example, a single ASCII character is a scalar. To see this all you have to do is recall that each character in the ASCII set is represented by a code, i.e. an integer given by ASC("c") where c is the character. In a sense a scalar is any sort of data that can be represented by a range of integers. For example, the days of the week, Monday to Sunday, can be represented by the integers 1 to 7, or 0 to 6, or l0 to 17, etc. What is most important to a scalar type is not so much which particular integers are used but the range of integers. Indeed some computer languages, notably Pascal, allow the definition of scalar types that effectively hide the fact that integers are used in the representation. For example, in Pascal you can define a type of variable that can be used to store one of the days of the week and then write statements such as

```
DAY:=sunday
FOR DAY:=monday TO friday
```

You can achieve the same effect in BBC BASIC by explicitly using integers to represent the days of the week. That is:

```
10 MONDAY=1:TUESDAY=2:WEDNESDAY=3:THURSDAY=4
20 FRIDAY=5:SATURDAY=6:SUNDAY=7
30 DAY=SUNDAY
40 FOR DAY=MONDAY TO FRIDAY
```

This use of integers to construct what are effectively new types of data is second nature to most BASIC programmers but it helps to see that it forms part of a larger pattern. The only scalar types that BBC BASIC provides are the integers themselves, characters and the Boolean values TR UE and FALSE.

Integers can be stored either in special integer variables (distinguished by having names ending in %) or in ordinary real variables. However, there are considerable advantages in using integer variables whenever you are only using integer values. The two advantages most often quoted are that integer arithmetic is fast, and integer variables occupy less memory. While these are indeed advantages, the biggest reason for using integer variables is that they clearly show when integer values are the only ones

that are logically possible. If for some reason you my to store a real value in an integer variable the program will, quite justifiably, crash! If you had used a real variable to store the integer values then this bug would go undetected.

Character data can only be stored directly in strings or, after conversion to an integer using the ASC function, in numeric arrays. The conversion from integers back to characters is achieved using the familiar CHR$ function. The on]y other way that ASCII characters reveal that they are represented by integers is by their order relations. The order of the ASCII characters is produced by the underlying order of the codes that are used to represent them. In other words "A"<"B" is true only because ASC("A") <ASC("B"). This is simple enough to understand but often causes problems when a mixture of upper- and lower-case characters are being sorted into order. The trouble is that all of the upper-case letters come before the lower-case letters in the order and this results in all the words that begin with upper-case letters coming before all the words beginning with lower-case letters.

The Boolean data type is particularly restrictive as it consists of the two values TRUE and FALSE. In nearly all versions of BASIC, Boolean data is stored directly in either integer or real variables. (This should be contrasted with the approach adopted by other languages of creating special Boolean variables only capable of storing the values TR UE and FALSE.) BBC BASIC uses the value -1 to represent TRUE and 0 to represent F ALSE. Y on can easily verify this fact by trying:

```
PRINT TRUE,FALSE
```

This assignment of integers to TRUE and FALSE is by no means standard and you should avoid writing programs that make use of it in any way. For example, in BBC BASIC it is true that TRUE<FALSE but in other versions of BASIC it may be false! Boolean variables are considered in more detail in Chapter Twelve.

Real data

Perhaps the most confusing thing about real constants and real variables is the use of the word 'real'. In mathematics the term 'real number' has an exact technical meaning but it has been taken over by computer scientists to mean any sort of number that has a fractional part. A real variable can be used to store numbers with fractional parts. The only problem is that an integer like 3 can also be stored in a real variable and this confuses the use of real and integer variables. Indeed early versions of BASIC only provided real variables, taking the attitude that integer variables were an unnecessary and confusing luxury.

In most cases it doesn't really matter whether you use real or integer variables. However it is important to be aware of the fact that real constants are not always stored accurately. Real variables in BBC BASIC

numbers with an incredible range (2*10^38 to 2*l0^-38). No matter how large or how small a number becomes, only nine digits are actually stored the rest are made up with zeros! So the number 1234567891234 would be stored as 1234567890000 i.e. smaller by 1234. This may seem like a large inaccuracy but in fact it is only a matter of around .0000001%! In other words, real variables may store numbers with large absolute errors but the percentage error is generally small enough to ignore. However, there are occasions when these small percentage errors cause large absolute errors in a final result. For example, adding very small numbers to very large numbers often gives the original large number as the result. Try:

```
10 A=12345678912
20 A=A+1
30 PRINT A
40 GOTO 20
```

and you will see that the value of A never changes! In other words in real arithmetic it is quite possible for A=A+C to be true without implying that C is zero! Other important sources of error are subtracting large numbers that are close in value and dividing by small values. The whole subject of accuracy and error in computation is difficult and beyond the scope of this book but it is important to be aware that some odd things can happen when using real variables.

Structuring methods - arrays and records

Now that the two types of static variable scalars and reals have been described it is worth introducing the two best known ways of organising data *arrays* and *records*. Arrays are so familiar to the BASIC programmer that it is hardly worth spending much time on them. BBC BASIC provides three types of array real arrays, integer arrays and string arrays. In the next section a method of constructing arrays for any sort of data type using the indirection operators is described. An array is characterised by being a collection of identical types of data called the elements of the array under a single name. Some computer languages allow the use of the array name on its own to signify that all of the elements are to be used. For example, if A is an array then PRINT A would mean print all of the elements. However BASIC in genera] and BBC BASIC in particular only allow single elements of the arrays to be used. Any particular element is signified by the use of an index in the usual way.

The main difference between an array and a record is that the elements of a record do not have to be all of the same type. For example, to store someone's name and telephone number you would use a string for the name and a numeric variable for the telephone number. While these two data types form a single logical unit of data in a program that manipulates telephone entries in BASIC there is no way that they can be gathered

together under a single name. In many other computer languages they could be turned into a 'record' called ENTRY (or whatever). This is similar to the way the elements of an array are collected together under a single name but instead of using an index variable the individual elements called-fields of a record are accessed by having additional subnames. In the case of the telephone directory entry for example, the field that stores the name might be called NAME and the field that stores the number might be called TEL-N U M. To make clear which field in which record you were referring to, most languages require the use of all the names that a field has. So to store a name in the record ENTR Y you would use:

    ENTRY.NAME="Fred"

and so on. This should be compared to the way an element of an array is specified by giving both the array name and an index. The array name corresponds to the record name and the index corresponds to the field mame.

A field within a record is not restricted to being a simple data type, it can be an array or another record. For example, if you wanted to store a date you might define a record called DA TE with three fields, DA Y, MONTH and YEAR, which might be incorporated as a field within ENTR Y. Now to save a date along with the name and telephone number you would write:

    ENTRY.DATE.YEAR=84
    ENTRY.DATE.MONTH=4
    ENTRY.DATE.MONTH=1

and this should be compared to the use of two index variables in a two-dimensional array.

Arrays and records are the two main data structuring methods found in other computer languages. The fact that BASIC includes arrays and not records is a reflection of the fact that while it is possible to do without records it is very difficult to do without arrays (but this is a debatable point!). Records are used to form tape and disk files of information that are the computer equivalents of traditional paper-filled filing cabinets:- Y ou might think that the use of arrays is equally obvious, but in fact they are often under-used!

The use of arrays - look-up tables

The most obvious use is for storing a list of data that is going to be processed. In this role arrays are similar to the mathematical 'objects' vectors and matrices. Indeed much of the purely numerical work that computers perform involves the use of one- and two-dimensional arrays as vectors and matrices respectively. However this is not the only use for arrays within computer science. Arrays are always introduced by

examples such as finding the mean of a list of numbers or reading in a list of numbers and printing them out in reverse order. This is understandable, but because of it many programmers never see an example of how to use an array as a look-up tabble.

Look-up tables are an excellent example of how the number of actions that a program has to perform can be reduced by using a more complex data structure. For example, suppose you need to use the SIN of a particular set of angles repeatedly; the most obvious thing to do is to work out the result each time it is needed but this often produces a very slow program! An alternative method is to work out the values that you need just once and then store them in an array for later use. In other words, use an array as a look-up table. In the case of simple functions such as SIN and COS there are always two ways of obtaining results by direct calculation or by using a look-up table. However there are plenty of 'functions' that cannot be summarised by a simple formula and in these cases the only choice is to use a look-up table. For example, there is no way that you can calculate the time of departure of the third flight to a particular destination but you can use a look-up table.

The use of look-up tables can become very complicated indeed, for a good example see Chapter Eleven. There are many occasions when a two-dimensional look-up table is necessary. For example, you could use a two-dimensional look-up table to hold the distance between pairs of cities. The biggest problem encountered with using look-up tables is the amount of memory that they take and the need to initialise them with the correct values. Sometimes you can take advantage of a pattern in the data to reduce the size of the table but this depends very much on the nature of the problem. For an example of how regularities in the data can be used to reduce a two-dimensional table to one dimension see Chapter Eleven.

Constructing new data types

The BBC Micro provides all the programming facilities necessary to construct new data types in the *byte array* and the three indirection operators '?', '!' and '$'. Although it is possible to construct new data types it is not something to be taken on lightly. Use of the indirection operators can easily produce programs that are very difficult to understand.

The byte array has already been introduced in Chapter Four. The statement:

    DIM variable_name size

will create a byte array of 'size'+1 bytes and store its starting address in 'variable'. For example:

    DIM N% 10

will create a byte array of 11 bytes and store the address of the first byte in N%. Although origignally introduced as a way of reserving space for the storage of machine code programs you can think of byte arrays as a way of reserving memory locations for any purpose, including the storage of data. Notice that while the memory location of a byte array will not change once created there is no guarantee that it will be created at the same location each time the program is run.

The action of the indirection operators is easy enough to understand. The byte indirection operator '?' performs the same functions as PEEK and POKE in other versions of BASIC. That is:

   ?address=value

stores 'value', an integer between 0 and 255, in the memory location at 'address' and:

   ?address

returns the contents of the memory location at 'address'. Similarly, the word indirection operator '!' can be used to store and retrieve four bytes of data in the format used to hold data in an integer variable. Thus any value that could be stored in an integer variable can be stored in four memory locations using '!'. The final string, indirection operator '$', will store and retrieve up to 256 characters starting at the address specified. When saving characters a carriage return is added to the end of the string and when retrieving characters a carriage return is used to signal the end of the string. For examples of the elementary use of the indirection operators see the *User Guide*, Section 39. Notice that when the indirection operators are written in front of a value that forms a valid address they behave just like variables. That is, an expression like ?address will return a value if used on the right-hand side of an equals and will store a value if used on the left. The expression:

   A=123

will store 123 directly in the variable A but

   ?A=123

will store]23 in the memory location whose address is stored in A, i.e. where the value is stored in determined 'indirectly' by A. Hence the use of the word 'indirection' as applied to the indirection operators. On a slightly more advanced level it is worth pointing out that each of the three operators can be used to store the results of expressions of the correct type. For example, you can use statements like:

   ?A=I*2
   !A=I+C

and

$A=A$+CHR$(7)

If you try to store a value larger than 255 using the byte indirection operator it will be reduced to the range 0 to 255 using the MOD function. That is, if x is greater than 255

?A=x

has the same effect as

?A=x MOD 256

As already mentioned, the indirection operator stores and retrieves data in the same format as used for integer variables. Thus the numeric range that can be handled is:

-2,147,483,648 to +2,147,483,647

The second advanced feature of the indirection operators '?' and '!' is the way that an 'offset' can be specified. For example, the command

A?F=D

is equivalent to

?(A+F)=D

The way that the indirection operators work is straightforward enough. However, it is not so easy to see how to use them for anything other than single values. Perhaps the simplest data type to implement directly is the one-dimensional array, Although BBC BASIC provides integer and real arrays there are occasions when it would be useful to use one-dimensional arrays that use fewer bytes per element. For example, if you know that the values that you want to store in an array lie in the range 0 to 255 then theoretically you can get away with one byte per element, but even using an integer array takes four bytes per element. However, using a byte array and the byte indirection operator you can construct your own array that uses only one byte per element. If you need an array with N elements then use:

10 DIM ARRAY N-1

To store a value in the Ith element use:

ARRAY?I=value

and to retrieve the contents of the I th element use:

variable=ARRAY?I

Similarly, if you need an array that uses two bytes per element i.e. a numeric range of 0 to 65535, then use:

    10 DIM ARRAY 2*N-1

to reserve enough memory,

    ARRAY?I*2=value MOD 256
    ARRAY?(I*2+1)=value DIV 256

to store a value and

    variable=ARRAY?(I*2)+256*ARRAY?(I*2+1)

to retrieve a value from the Ith element.

Two-dimensional arrays can be implemented in the same way. The only complication is finding any given element. For example, if the array uses one byte per element then the space needed by an N by M array is:

    l0 ARRAY N*M-1

and the address of the I,J element is:

    ARRAY+N*J+I

Notice that each of these array definitions has two parts; the reservation of the correct amount of memory as a byte array, and a function that gives the location of any particular element This scheme can even be extended to user-defined records. For example, consider the telephone directory record given earlier. If the name field is restricted to a maximum of 20 characters and the telephone number is stored as an integer then a record can be defined as:

    10 DIM ENTRY 23
    20 NAME=0
    30 NUMBER=20

where NAME and NUMBER are offsets that define the start of each field from the first location allocated to the record. Thus:

    40 $(ENTRY+NAME)="fred"

will store a string in the name field and:

    50 ENTRY!NUMBER=12345

will store an integer in the number field. Notice that the form of a field specification when using '?' and '!' is particularly neat in that the record name can be written next to the field name using the indirection operator as a separator.

Once implemented it is possible to manipulate the elements of the user-defined arrays and records just as if they were standard variables. The only real restriction is the lack of an indirection operator that will store a real variable. This can be overcome in a number of ways. For

example you could convert the real value to a string of digits using STR$ and then store it using '$'. In general, user-defined data structures are more difficult to work with than the standard BASIC arrays. However there is one respect in which they are better than standard arrays. Although you cannot pass an array to a procedure as a parameter there is nothing to stop you from passing the address of the start of the array! For example, you could write a procedure that would add together elements I to N of an integer array and store the result in element zero:

```
1000 DEF PROCsum(NAME%,N)
1010 LOCAL I,SUM
1020 SUM=0
1030 FOR I=1 TO N
1040 SUM=SUM+NAME%!(I*4)
1050 NEXT I
1060 NAME%!0=SUM
1070 ENDPROC
```

where NAME% contains the address of the start of the byte array being used. Notice that this procedure can total any user-defined integer array. For example:

```
10 DIM ARRAY% 400
20 PROCsum(ARRAY%,100)
30 PRINT ARRAY%!0
```

will total ARRAY%. This is such a useful facility that it is worth exploring ways of extending it to the standard BASIC arrays.

BASIC arrays and indirection

All BASIC arrays are stored in a very simple format composed of two parts; a number of memory locations that hold part of the name and other information about the size of the array, and a data section that actually stores the array elements. If the start address of the array can be found then there is nothing to stop us from using it with the indirection operators to manipulate the contents of the data section of the array directly. In particular, knowing the address of the start of an array makes it possible to write procedures that will perform an operation on any array (as in the case of PROCsum in the previous section).

The most obvious way of finding the address of the start of an array (or any variable for that matter) is to write a function that searches the area of memory where the variables are stored. However this is not an easy function to write and there is a much simpler method that works just as well. In BBC BASIC the memory needed to store arrays and simple variables is allocated only when they are actually encountered during the running of a program. This means that arrays are stored in the order that

they are defined by DIM statements and as long as no other variables are used in between such definitions they will occupy adjacent areas of memory. Thus the statement:

    DIM name 0,name(n)

will create two arrays next to each other in memory. The first is a byte array consisting or a single memory location and the second is a standard array consisting of n+1 elements. As the two arrays are created next to each other you might expect the variable 'name' to contain the address of the start of the standard array minus one. However, for some reason BBC BASIC allocates three memory locations to the byte array even though you only requested one, so 'name' holds the address of the start of the standard array minus three. For example:

    10 DIM ARRAY% 0,ARRAY%(100)

creates an integer array of l01 elements (i.e. ARRAY%(0) to ARRAY%(100)) and a simple integer variable that contains three less than the address of the start of the array. The only complication is that it is not good enough simply to add three to ARRA Y%, to give the start of the array because the start of the array is not the start of the area where the data is stored. The format used to store variables is extensively described in *The BBC Micro: An Expert Guide* but all that you need to know in this context is that the end of array's name is marked by a zero and the next memory location holds a constant that indicates the start of the data area. Therefore to alter the address stored in ARRA Yg$ so that it 'points' to the data area u be:

    2000 DEF FNcorrect(START%)
    20l0 START%=START%+2
    2020 REPEAT
    2030 START%=START%+1
    2040 UNTIL ?START%=0
    2050 =START%+START%?1

as in

    ARRAY%=FNcorrect(ARRAY%)

Once the address has been corrected to point to the start of the data area of the array it can be treated exactly like the user-defined arrays given earlier. That is, you can use PROCsum to add up all the elements of any integer array. For example, try:

    10 DIM ARRAY% 0,ARRAY%(100)
    20 ARRAY%= FNcorrect(ARRAY%)
    30 FOR I=1 TO l0
    40 ARRAY%(I)=1

```
50 NEXT I
60 PROCsum(ARRAY%,100)
70 PRINT ARRAY%(0)
80 END
```

where PROCsum and FN correct have to be appended.

Dynamic variables - strings and pointers

All of the variables and data structures that we have examined so far have been static, in the sense that the amount of memory that is needed to store their values doesn't change while a program is running. By contrast dynamic variables do change their size as a program runs. To a BASIC programmer the most familiar example of a dynamic variable is the humble string. Indeed most BASIC programmers use string variables without a second thought but strings, like all dynamic variables, are very difficult to implement efficiently. So much so that many other programming languages, Pascal for example, do not provide strings as a data type. What they do provide as an alternative is the *character array*. A character array can be thought of as a fixed length string, but it is more directly related to a one-dimensional array where each element can be used to store a single character. BBC BASIC doesn't have character arrays but it is easy to see how they could be created using byte arrays and the string indirection operator.

   The details of how the BBC Micro tackles the difficult problem of implementing strings are also given in *The BBC Micro: An Expert Guide* but essentially what happens is that a certain amount of memory is allocated to a string variable when it is created. If the number of characters that it has to store exceeds this initial memory allocation a brand new and larger amount of memory is allocated to it. This process can be repeated any number of times during the course of a program, each time leaving behind a redundant area of memory that was once allocated to the string variable. In this way it is possible to use up all of the BBC Micro's memory by storing old copies of string data. If this is a particular problem within a program the solution is very simple. All you have to do is initialise all the strings to hold the largest number of characters that the program will ever want to store. In this way the initial amount of memory allocated to string variables will be large enough never to need increasing.

   Although strings are probably the most useful form of dynamic variables, they are so well known to the BASIC programmer that it is more sensible to spend time explaining how other forms of dynamic variables work and can be implemented in BBC BASIC. After the string the most important dynamic data types are:

   the stack
   the queue

the linked list and
the tree

Each of these four, along with other types of dynamic variable, are closely related to the use of *pointers*. A pointer is simply a variable that is used to store a memory address. In this sense a pointer really does 'point' at a memory location. There is nothing stopping us from storing an address in any numeric variable and indeed this has been done many times to access the contents of a memory location in combination with the indirection operators. That is, in:

?A

the variable 'A' is being used as a pointer to a memory location. The art of implementing and using dynamic data types on the BBC Micro is in setting up and manipulating pointers and this is best explained by examples of each of the dynamic data types listed.

The stack

The stack, or more accurately, the Last In First Out (LIFO) stack is well known to assembly language programmers. In action it mimics the operation of a stack of coins or plates. If you make a stack of plates you can identify two operations, adding to the stack by placing a plate on top and, conversely, removing a plate from the top of the stack. If you think about it for a moment you will see that it is a characteristic of such a stack that the order that the plates are removed is the opposite of the order that they were added for the simple reason that the plates added to the stack first will be further toward the bottom of the stack! In computing a stack behaves in the same way. Storing data on the stack is known as 'pushing data onto the stack', a PUSH operation, and retrieving data is known as 'pulling data off the stack', a PULL operation. The best way to illustrate the operation of a stack is by giving an example.

There are two components to the implementation of a LIFO stack, an area of memory that is used to store the data (often referred to as the stack itself) and a pointer that marks the top of the stack. The most obvious way to reserve some memory for a stack is once again the byte array. The pointer can be implemented as a standard BASIC variable used in conjunction with the appropriate indirection operator. When the stack is initially created the pointer variable or 'stack pointer' is set to point to the first free memory location. A PUSH operation stores data in this free memory location and then adjusts the stack pointer so that it points to the next free memory location. A PULL operation first adjusts the stack pointer so that it points at the data item on the top of the stack and then retrieves it, so freeing the location.

The following program uses an integer stack to reverse a list of ten integers:

```
 10 DIM STACK% 400
 20 POINTER%=STACK%
 30 FOR I=1 TO l0
 40 INPUT A
 50 !POINTER%=A
 60 POINTER%=POlNTER%+4
 70 NEXT I
 80 FOR I=1 TO 10
 90 POINTER%=POINTER%-4
100 PRINT !POINTER%
110 NEXT I
```

Line 10 reserves sufficient memory for a stack of up to 100 integer items (you will remember that an integer takes four bytes to store) and line 20 initialises the stack pointer POINTER% to the start of the area. The FOR loop at lines 30 to 70 reads in l0 integers and pushes them onto the stack (fines 50 and 60). The second FOR loop at fines 80 to I l0 pulls and prints each item off the stack in turn (lines 90 and 100).

You can create stacks using other data types as raw material. For example, you can use a standard BBC BASIC array indexed by a variable playing the role of the stack pointer. Stacks are used whenever the order of incoming data has to be reversed or just as a way of temporarily storing data until it can be dealt with.

The queue

Once you have seen how a stack works the queue is an obvious next step. Indeed a queue is often referred to as a First In First Out or FIFO stack. The action of a queue is exactly what you would expect from a consideration of the way people queue. The first person to join the queue is (usually!) the first person to be served. So it is with the data queue, the first item to join the queue is also the first item to leave it. Notice that the two fundamental queue operations have already been identified JOIN adds a data item to the queue and LEAVE removes an item.

The implementation of a queue involves an area of memory and a pair of pointers. One of the pointers marks the front of the queue and the other marks the end. Obviously data items are taken out of the queue using the FRONT pointer and added using the END pointer. That is, assuming each item is a single byte,

JOIN is
```
?END=value
END=END+1
```

and

LEAVE is
```
variable=?FRONT
```

FRONT FRONT+1

Initially the FRONT and END pointers point to the same memory location and as data is added to the queue the end moves up in memory and the front stays put. Similarly, as data is removed from the queue, the front moves up and the end stays put. The only trouble with this implementation of the two operations is that the front and end of the queue move ever upward! The solution is to make the queue circular by setting any pointer that reaches the top of the allocated memory back to the start of the area. For example, the following program reads in l0 numbers and adds (joins) them to a queue and then prints them out in the order that they arrived.

```
 10 DIM QUEUE% 100
 20 FRONT%=QUEUE%
 30 REAR%=QUEUE%
 40 FOR I=1 TO l0
 50 INPUT A
 60 ?REAR%=A
 70 REAR%=REAR%+1
 80 IF REAR%>QUEUE%+100 THEN REAR%=QUEUE%
 90 NEXT I
100 FOR I=1 TO l0
110 PRINT ?FRONT%
120 FRONT%=FRONT%+1
130 IF FRONT%>QUEUE%+100 THEN FRONT%=QUEUE%
140 NEXT I
```

You should be able to recognise the JOIN and LEAVE operations in lines 60 to 80 and 1l0 to 130. Also notice that, in contrast to the stack in the last section, this queue stores single byte items.

Queues are used whenever data is generated too fast to be dealt with immediately but nevertheless needs to be processed in the order that they were generated. The BBC Micro uses many queues in its normal operations. For example, the sound queue, the printer buffer, the keyboard buffer, etc.

The linked list
The queue and the stack are characterised by restrictions on where data can be added or removed. In the case of the stack, data can only be added to or removed from the top and in the case of the queue, data can only be added at the end and removed from the front. The linked list is a more versatile data type in that it is relatively easy to add and remove data from any position. In this sense the linked list is more like the everyday idea of a list of items made on paper.

The fundamental principle that lies behind a linked list is that each item in the list includes a pointer to the next item in the list. This, coupled with a pointer to the start of the list and some way of detecting the end of the list, is ail that there is to implementing a linked list. For example, a list of names can be built up using a byte array and the string and word indirection operators. If each item in the list is defined to consist of a name with a maximum of 20 characters and a single integer pointer to the next item, it is easy to work out that each item needs 24 bytes of storage. The following program will build up a list of up to l0 names:

```
 10 DIM NAME_LIST 24*10
 20 START%=NAME_LIST
 30 CURRENT% NAME_LIST
 40 CURRENT%!20=0
 50 INPUT N$
 60 IF LEN(N$)>19 THEN GOTO 50
 70 $CURRENT%=N$
 80 CURRENT%!20=CURRENT%+24
 90 CURRENT%=CURRENT%+24
100 CURRENT%!20=0
1l0 GOTO 50
```

The format of the list is as shown in Fig. 5.1. Notice that each pointer points to the start of the name and the last item doesn't store a name and its pointer is set to 0.
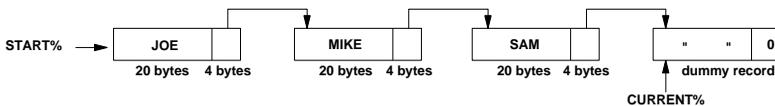

Fig. 5.1 An example of a linked list.

Notice that in the above example the pointer CURRENT% is used to add items to the end of the list but new items can in fact be added anywhere. For example, if you wanted to insert an item to follow the first, i.e. to become the second item, all you would have to do is change the pointers as shown in Fig. 5.2. The important idea is that the order that a linked list is read is determined by the pointers, not by where the item is stored.
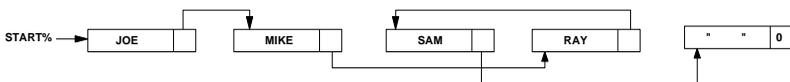

Fig. 5.2. Adding "RAY" to the linked list in Fig. 5.1. Notice that reading the list by 'following' the pointers gives the names in correct alphabetical order

The items that make up a linked list can be any data type as long as it includes a pointer to the next item. For example, the telephone directory given earlier in the chapter is best implemented as a linked fist with each

item as a record that includes a pointer to the next record. Using a linked list for a telephone directory has the advantage that when new items are added they can be inserted into the list at the correct position to preserve alphabetic order. (To do this the list is read, by following the pointers, until the first item that the new name should precede is found, then the new item is inserted by manipulating the pointers (as shown in Fig. 5.2).) Without using a linked list each new item would need a complete sort of the list to make sure that the result was in the correct order.

There are so many uses for linked lists that it is impossible to summarise them. However, it is often the case that ignorance of the existence of the linked list forces a programmer to use an array instead. The result is always a program that is much more complicated and involves a great deal of data moving. A linked list should be used whenever a list of data needs to be repeatedly rearranged.

Trees

The final dynamic data type is the most complicated of all the tree but if you have followed the use of pointers to form a linked list it should be easy to understand. Each data item in a tree is associated with two pointers the left pointer and the right pointer. The best way to think about this is as shown in Fig. 5.3 which clearly reveals why this data type is called a tree!
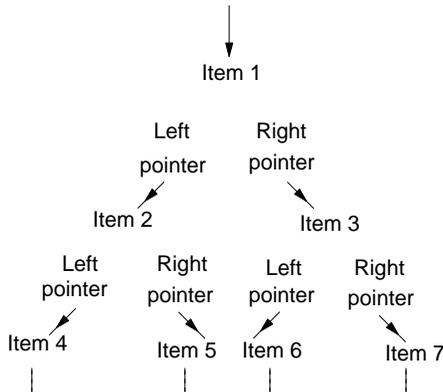
Fig. 5.3. A tree.

To implement a tree in BBC BASIC all that you have to do is reserve some memory using a byte array and use pointer and indirection operators in exactly the same way as for the linked list. (Apart from the fact that there are two pointers per item and an increased choice of where to add an item, that is!)

Trees are used to represent any sort of data that has a hierarchical structure like a famjly tree, or the command structure of an organisation. Most of their uses are found in advanced computer topics such as artificial

intelligence and are thus outside the scope of this book. (If you would like to know more about this application of trees, see *Artificial Intelligence* in BASIC by Mike James published by Newnes, 1984.) However, you should be able to see how the tree is a simple use of pointers to construct complicated data structures. In fact the type of tree that has been described is more correctly described as a binary tree, because each item has two pointers. It is quite possible to construct trees that contain more pointers per item and linked lists that contain pointers to both the next item and the previous item (sometimes called *doubly linked lists*). The range of variations is too great to go into but once you have mastered the idea of using a pointer to indicate where the next item is stored you should be able to see them all for yourself.

Too many types?

At the end of this long chapter you might feel that there are far tqo many data types and structures to cope with. However, if you make a list of the data types that have been introduced you might be surprised to discover how few there are. Simple static data reduces to nothing more than the scalars, based on integers, and the real numbers. The only two common data structuring methods are the array and the record. Even if you include the dynamic data types we only have four more: the stack, the queue, the linked list and the tree. What is amazing is that so many different problems can be tackled successfully with such a small number of types!

Chapter Six
File Storage

One important data type that has been ignored so far is the file. This is because files are generally not stored within RAM but, in the case of the BBC Micro, on tape or disk. The fact that files are not stored in RAM does affect the sort of applications that they can be used for. At the most trivial level, the fact that disk or tape has a much larger storage capacity than RAM makes it possible to write programs that handle realistic amounts of data. On the other hand, it is important not to miss the point that the idea of a file is independent of the physical device used to store it. Indeed, one of the strengths of the BBC Micro's file system is that it provides the programmer with a range of commands that are as far as possible 'device independent'. This chapter examines the BBC Micro's filing system and how it can be used from both BASIC and 6502 assembler with as much 'device independence' as possible.

The sequential file

There are two different types of file - *sequential files* and *random access files*. Of the two, only sequential files can be stored on serial devices such as tape and so these tend to be the more common type of data file. Because of this the term 'file' used on its own nearly always refers to sequential files. A file can be thought of as a 'list' of data items constructed in such a way that only one item can be read or written at a time. The only real difference between random access and sequential files is that sequential files can only be read and written in a given order. It is helpful to think of a pointer that is used to indicate the 'current' position in the file. That is, the pointer points to the item that will be read from or to the location in the list that will be written to. In the case of random access files, the pointer can be freely moved to any position within the file. Sequential files are much more restrictive. In fact there is no way that the pointer can be influenced directly. Instead each time that an item is read or written the pointer is 'moved on' to the next item. The situation is a little more complicated than this in that generally you cannot change individual items in an existing file.

When a file is being created, storage space is allocated and the pointer

is set to the beginning of this area - this is called 'opening the file for writing'. Notice that in this case the pointer points at the next free area of storage so there is no way that anything can be read from a file that has been opened for writing. As each item is written the pointer is moved on to point to a new free space .Thus, items can only be added to the end of the file and the file grows in length with each item that is written.

When an existing file is being read the pointer is initially positioned at the first item in the file this is called 'opening the file for reading'. As each item is read from the file the pointer is automatically moved on to point at the next item if there is one! It is possible to try to read more items from a file than were written to it. If you try and do this on a BBC Micro you will generate an 'end of file error'.

There is one additional operation concerning files which in many ways is the opposite of opening the file. Although in principle it is always possible for a computer to deduce when you have finished using a file, there are advantages in defining an operation that explicitly informs the computer that you are finished -- this is the close file operation. What the computer actually does when asked to close a file depends on the type of device that the file is stored on and whether the file was open for reading or writing. If the file was open for reading then a closing operation simply resets various internal variables so that the file can potentially be re-opened later in the program. If the file was open for writing then a close operation will make sure that all the data that was written to the file is actually written out rather than sitting in a buffer. After this a special marker, the 'end of file marker', is inserted to show that there is no meaningful data beyond this position.

If you are already used to sequential data files on tape or disk you may find the above description unnecessarily complicated. However, thinking in terms of a pointer to the current position in the file certainly helps when it comes to understanding random access files. As the BBC Micro only supports random access files on disk it makes more sense to leave a description of how they work until after a description of disk files in general.

BASIC file commands

The previous section introduced the sequential file in a very general way. In practice any useful computer language provides a collection of commands that manipulate files. The trouble is not only that each computer language has invented its set of file commands, so have the different dialects of BASIC! Some versions of BASIC even provide different commands for handling files that are stored on tape and on disk! The situation is not as bad as it seems from this description because all the sets of commands have a great deal in common. BBC BASIC% file handling commands are particularly logical and the only real differences between files stored on tape and disk are that cassette tape is restricted to

purely sequential files.

The *User Guide*, Section 33, gives a good introduction to the BBC Micro's file commands and as a result it is not worth going over each command in detail. However it does seem worth giving a brief summary of what the different commands are used for and how they fit into the general file scheme described earlier.

The fundamental BBC BASIC file operations are:

OPENOUT, OPENIN, CLOSE, BGET and BPUT

There are other file operations that are useful but these five are the ones from which everything else stems. The commands OPENOUT and OPENIN perform the file opening operations described earlier. OPENOUT opens the file for writing and the OPENIN command opens the file for reading. The only additional features are that opening operations are used to associate a 'filename' with a 'channel number'. The filename performs the same function for files as the variable name for variables i.e. it identifies an area of storage on the file device concerned. Although it would be possible to use the filename throughout a program to identify which file is being used, it is more convenient to use 'channel numbers'. When a file is opened it is assigned a unique channel number that is used to identify it in all further operations. The exact forms of the OPEN commands are:

variable=OPPENIN(filename)

and

variable=OPENOUT(filename)

where 'filename' can be a string expression or a constant and the channel number is returned as the result of each function and stored in 'variable'. Following either of these two commands the file is referred to by the channel number stored in 'variable' and not by its filename.

The command:

CLOSE# channel

performs the file close operation on the file assigned to channel number 'channel'. As described in the general introduction to files, you must issue a CLOSE command before trying to re-open the file and you must issue a CLOSE command when you have finished writing a file to make sure that all of the data that you have written to the file is safely stored.

The two commands:

BGET and BPUT

perform the operations of reading and writing an item of data. In this case the term 'item' refers to a single byte that can either be used directly as a number in the range 0 to 255 or can be interpreted as a single ASCII

character (using CHR$). Thus:

BGET# channel

is a function that returns a single byte from the file assigned channel number 'channel' and:

BPUT# channel,value

is a command that writes the single byte 'value' to the file assigned channel number 'channel'. As you would expect, each instruction automatically moves the pointer to the current file position on by one byte.

BBC files in use - buffering

Using just the five instructions introduced in the last section a file can be created and read back. For example:

```
 10 INPUT F$
 20 F=OPENOUT(F$)
 30 FOR I=1 to 300
 40 BPUT#F,I MOD 256
 50 PRINT I
 60 NEXT I
 70 CLOSE #F
 80 F=OPENlN(F$)
 90 PRINT BGET#F
100 GOTO 90
```

If you are using (or have selected) the tape filing system then add:

75 PRINT "REWIND TAPE"

before running this program. Line 20 opens a new file for writing, the file name is in F$ and the channel number is returned in F. Lines 30 to 60 write out 300 bytes to the file and to the screen. Line 70 closes the file and line 80 re-opens it for reading. Lines 90 and 100 form an infinite loop reading and printing a single byte at a time from the file.

If you run this program with either the tape or disk filing system selected (using *TAPE or *DISC) then you will see a number of common features about the way files are handled, Firstly, once the file has been opened for writing you will see 256 numbers printed on the screen, then the disk or tape will be activated and the program will pause. The tape will then stop (or the disk will be deactivated), the program will continue to print the remainder of the numbers on the screen and once again the tape or disk will be activated. The reason for this stop go behaviour is that the BBC Micro *buffers* all file operations. That is, bytes are not sent

directly to the disk or tape, instead they are collected in an area of memory called a buffer. Only when the buffer is full or the file is closed is anything written on the tape or disk. Thus, data transfers between the file storage device are in terms of 'buffer loads' rather than single bytes.

The same buffering behaviour can be seen when the file is read. The OPENIN command actually reads in a buffer full of bytes which the BGET command uses each time through the loop (lines 90 to 100). Of course when the buffer is empty the program pauses while another buffer load of data is read in. Buffering may seem a complicated way of handling files but it is much faster than any method based on transferring single bytes to and from the file storage device!

The above program comes to an end rather abruptly with an error message 'EOF at line 90'. The solution to this problem is to be found in the EOF function. The function:

EOF# channel

returns the value TRUE (-1) if the last byte has been read from the file and FALSE (0) otherwise. Notice that EOF goes TR UE after the last item has been read and not when you try to read beyond the last item. This means that EOF can be used as part of a REPEAT . . . UNTIL loop to process entire files. For example:

```
10 INPUT F$
20 F=OPENIN(F$)
30 REPEAT
40 PRINT BGET#F
50 UNTIL EOF#F
```

will read the file ES, byte by byte and stop immediately after reading the last byte in the file. You can see the action of EOF more clearly if you add:

```
95 PRINT EOF#F
```

to the earlier file creation and reading program. What this shows is that EOF goes TR UE when the last valid byte is read from the file but you can still read a 'dummy' value (of 254) from the file before the EOF error message is given and the program stops. This is useful because it enables a file to be read completely by a while loop, but it is also dangerous because it is possible to process a totally spurious value if the change in the EOF function is not detected immediately.

Larger data items - PRINT# and INPUT#

Although BBC Micro files work in terms of single byte data items, the data types used in BBC BASIC are generally composed of a number of

bytes. For example, a standard integer variable takes four bytes to store its value. It is possible to see how real, integer and string variables could be stored and retrieved from a file one byte at a time but this would be tedious and do commands:

    PRINT# channel,print_list

and

    INPUT# channel,input_list

where 'channel' is the channel number as used in BGET and BPUT and 'print_list' and 'input_list' are lists of variables separated by commas. Although there is a superficial similarity between PRINT # and PRINT and INPUT# and INPUT it is important to realise that their actions are very different.

   Both PRINT and INPUT handle data output and input in terms of the ASCII representation of data. For example, PRINT A will convert the value stored in the variable 'A' from its internal format to a string of ASCII digits. In this sense PRINT A performs the same set of actions as PRINT STR$(A). By contrast PRINT# and INPUT# handle data without changing it very much from its internal format. The actual format and the number of bytes of storage used within a file depends on the type of value. Integers are stored using five bytes, the first byte is always &04 and serves to identify the following four bytes as an integer value. The integer value is stored in the same format used for integer variables, that is 32-bit two's complement with the least significant byte first. Real values use six bytes, the first byte is always &FF and serves to identify the following five bytes as a real value. String values take two bytes more than the number of characters that compose the string. The first byte is always &00 to signify a string value, the second byte gives the length of the string. The actual characters that make up the string follow the 'length byte' and the only peculiarity is that they are stored in reverse order. For example, the statement:

    PRINT# F,A

will write six bytes to a file, &FF followed by the five bytes of the internal representation of A. The statement:

    PRINT# F,A%.

will write five bytes to a file, &40 followed by the four bytes that make up the internal representation of the integer stored in A%. As already explained, the number of bytes used by a PRINT# statement that writes a string to a file depends on the number of characters in the string. For exampie:

    PRINT# F,"ABCDE"

writes seven bytes to the file; &00 to indicate that a string follows, &05 to indicate the number of characters in the string and then five bytes for the ASCII codes of "E", "D", "C", "B" and "A". (Notice that the characters of the string are stored in reverse order.)

In the same way that the PRINT # statement writes out the different types of data values using the formats described above, the INPUT# will read them back into variables of the correct type. It is possible to read in an integer value to a real variable and even a real value into an integer variable (any fractional part is truncated) but trying to read a string value into a numeric variable or a numeric value into a string variable will produce a type mismatch error. This means that you can use PRINT# to create files with a mixture of data types and to read such a file back using INPUT# you have to know in which order they come. This sounds like a difficult task but in practice the pattern of data types in a sequential data file is simple. For example, in the case of a name and telephone number file, it is obvious that the pattern is always a string for the name, followed by an integer for the telephone number. If you think in terms of the data type record, introduced in the previous chapter then file organisation is easy. Any collection of data that has to be stored in a file should be thought of and manipulated as a single record. For example, the data items NAME$ and NUMBER% should be thought of as a single data item, in other words as a record. Rather than reading (or writing) the name and the number parts as and when they are needed within the program they should always be handled together. That is, always read and write *all* the information that constitutes the record. The best way to ensure that only whole records are dealt with is to confine the reading and writing of records to a pair of procedures, PROCrec_get and PROCrec_put (say). Of course, this also fits in with the philosophy of modular programming!

Allowing for different file devices

The aim of the BBC Micro's file handling commands is to be as device independent as possible. And as far as sequential files are concerned this objective has been achieved fairly well. You can use the same commands to read and write a file that resides on disk or tape. However, there are times when it is useful to know what sort of device a file is stored on. In the earlier example where a file was written, then closed, then re-opened and read, it was important to inform the user that the writing of the file was complete only if the file was being stored on tape. The reason for this is simple enough; a cassette tape has to be manually rewound before the file can be reread but a disk will look after itself.

This sort of slight difference causes no problem unless you are trying to write programs that will work with any file device. In this case if you choose to issue the "Rewind tape" message and disks are being used your program will look silly, but if you don't issue the message and tape is being used your program will be unusable. There is a way to find out the

sort of file device in use but it involves the use of the filing system for assembly language, a subject to be explained in more detail later. To find out what the current file device is the BASIC programmer can use the following function:

```
1000 DEF FNfi1e_system
1010 X%=&70
1020 Y%=0
1030 A%=0
1040 =USR(&FFDA) AND &FF
```

The value that this function returns determines which type of file device is in use according to the following table:

value	
0	no filing system
1	1200 baud cassette
2	2300 baud cassette
3	ROM pack
4	disk
5	Econet
6	Teletext/Prestel

Using this function it is possible to discover the type of file device employed and to take appropriate action.

Random access files

Although the ideas and principles of random access files are device independent it is only practical to use them in conjunction with disk drives. As a result, this section and the commands that it describes apply to the BBC Micro only plus its disk drives and the ACORN disk filing system.

If you followed the explanation of the way that a sequential file works in terms of moving a pointer through the data items that make up the file, then random access files will seem easy. In the case of random access files the 'current position pointer' is available as a 'supplied variable' with the name:

PTR# channel

When a file is opened PTR# is set to zero, so pointing either at the first byte in the file or the position where the first byte will be written. As bytes are written to or read from the file PTR# is automatically updated just as in the case of a sequential file. The difference is that now you can set the pointer variable to read or write any byte in the file directly. For example,

if you want to write the l00th byte to a file use:

```
PTR#F=100:BPUT#F,value
```

You can set PTR# to point to any byte of a file while writing, but you cannot go past the of the file while reading. So that you can avoid trying to read more of a random access file than there is, BBC BASIC provides the EXT# function which returns the number of bytes in the file.

The variable PTR# and the function EXT# are the only extras needed to manipulate random access files. However, being able to read or write any given byte in a file is only a little way along the road to useful rand om access of data. As in the case of sequential files the most useful way of organising data for storage in a random access file is to think in terms of records. Once again, only whole records should be read or written, preferably using procedures, one for writing and one for reading. To enable any record to be accessed in any order, it is necessary to associate a 'record number' with each one. If each record takes R bytes of storage then you should be able to see that to access record I, PTR# has to be set to I* R. For example, in the case of the telephone directory program used earlier, the name field of the record may be allocated 20 characters of storage and the telephone number field, being an integer, takes five bytes. Thus to write record I you would use:

```
1000 DEF PROCput_rec(I)
1010 PTR#F=I*27
1020 PRINT# F ,NAME$
1030 PTR#F=I*27+22
1040 PRINT#F,NUMBER%
1050 ENDPROC
```

where NAME$ contains the name and NUMBER% the telephone number. Line l0l0 calculates the position of record I and moves the file pointer to its start. Notice that the length of each record is 27 bytes (= 20 characters + I string code byte + I string length byte + 5 bytes for the integer). Line 1020 writes NAME$, line 1030 moves the pointer on to the start of the number field within the record and then line 1040 writes the telephone number to the file. Although fine 1020 automatically moves on PTR#, line 1030 is necessary to ensure that PTR# has been moved on to allow space for up to 20 characters. When using random access files it is important that fields within records always take the same amount of space — if they don't take the same amount of space it is much more difficult to find a given record. To read a given record of the telephone directory file use:

```
2000 DEF PROCget_rec(I)
2010 PTR#F=I*27
2020 INPUT# F,NAME$
2030 PTR#F=I*27+22
```

2040 INPUT# F,NUMBER%
2050 ENDPROC

which works in the same way as PROCput_rec. There are various other
ways of organising records within a random access file but they are all
based upon the methods described for constructing new data types given
in the previous chapter. For example, you should be able to recognise the
fixed size record random access method described in this section as being
essentially the same method used for organising a one-dimensional array
with the record number corresponding to the array index! In the same way
you can construct linked lists on disk by including within each record a
pointer to the next record.

The BBC Micro's disk filing system software is unfortunately very
primitive in the way that it allocates storage space. Disk storage is
organised into 'sectors', each capable of holding 256 bytes. A sector
represents the smallest amount of disk space that can be allocated. The
problem with the BBC disk filing system is that a file has to occupy a
block of sectors that are physically next to each other on the disk. When a
new file is opened for writing sixty-four sectors are assigned for its use. If
it doesn't use all sixty-four when the file is closed the unused sectors can
be used by another file. If more than sixty-four sectors are required
additional sectors can be acquired, but only if there are free sectors that
immediately follow the initial sixty-four! The trouble is that although
there may be more than enough free sectors on the disk to hold the entire
file, if they are not immediately next to the sixty-four already allocated the
program will crash with a 'Disk full' message. This is unlikely to happen if
the file that needs extending is being stored on a newly formatted disk, but
otherwise it is better to claim the amount of space required for a file as
soon as it is opened. That is, if you know that you are going to want to
create a random access file of 100 records, each 5 12 bytes long, you
should open the file and immediately write 51200 bytes out to it. If this
fails because there isn't enough space available as a single block of
adjacent sectors you can either change disks or use the *COMPACT
utility which moves existing files in an attempt to place all of the free
sectors together at one place on the disk.

The OPENUP problem

So far it has been assumed that a random access file will be opened using
OPENIN for reading and OPENOUT for writing. However there is
another way of opening a file which means that it can be both written and
read in the same program. The command:

variable = OPENUP(filename)

will open the file 'filename' for *both* reading and writing and will store its

channel number in 'variable' for future reference. This sounds very useful but there are differences to be found in the way that all three OPEN commands have been implemented in early and late versions of BBC BASIC. BASIC I has only OPENOUT and OPENIN although OPENIN is implemented in such a way that it can be used to both read and write a random access file. That is, BASIC I OPENIN behaves like OPENUP is supposed to. However all three OPEN commands are implemented in BASIC II. The trouble is that programs written using BASIC I do not translate as you might expect to BASIC II. In particular any OPENIN commands that you might have used in a program will have mysteriously changed to OPENUP commands! Going the other way, that is taking a BASIC II program and running it under BASIC I, is even more of a problem because any OPENUP commands are translated to OPENIN commands and any OPENIN commands will cause errors! The reason for all this is that OPENIN in BASIC I seems to use the token for OPENUP, hence the mysterious translations.

There is no simple solution to this confusing problem. If you are writing programs using BASIC I, the best course of action is to use OPENOUT and OPENIN and the program should run under BASIC II If you are writing programs under BASIC II then use only OPENUP and your programs should run under BASIC I.

Virtual arrays

As already mentioned, the method used to construct random access files based on fixed size records is essentially the method used to construct arrays in RAM. This fact can be used to construct arrays that are stored on disk so-called *virtual arrays*. For example, if you want a virtual real array then use:

```
1000 DEF FNget_array(I)
1010 LOCAL DATA
1020 PTR#F=I*6
1030 INPUT# F,DATA
1040 =DATA
```

to access the Ith element stored in the virtual array corresponding to channel F and use:

```
2000 DEF PROCput_array(I,DATA)
2010 PTR#F=I*6
2020 PRINT#F,DATA
2030 ENDPROC
```

to store DATA in the Ith element. Two-dimensional arrays are just as easy; the only real change is that the expression in lines 1020 and 20l0 becomes:

    PTR# F=I*6+N*J*6

to access the I,Jth element of the array.

Using files from assembler

The BBC Micro's filing system is almost as easy to use from 6502 assembler as from BASIC. There is a great similarity between the MOS routines used to manipulate files and the equivalent BASIC commands. For example, the routines OSBPUT and OSBGET will write and read a single byte in the same way that BPUT and BGET do. Table 6.1 gives details of the MOS routines corresponding to each of the BASIC file operations:

Table 6.1

BASIC	MOS routine	parameters
OPENIN	OSFIND (&FFCE)	A=&40, Y,X address of file name. On return Y contains the channel number. Y=0 if OSFIND cannot open the file
OPENOUT	OSFIND (&FFCE)	As for OPENIN but A=&80.
OPENUP	OSFIND (&FFCE)	As for OPENIN but A=&C0.
CLOSE	OSFIND (&FFCE)	A=0, Y=channel number of file to be closed. (If Y=0 then all files are closed.)
BPUT	OSBPUT (&FFD4)	A=byte to be written, Y=channel number
BGET	OSBGET (&FFD7)	Y=channel number. On return A contains byte read from file. Carry flag=1 if an error has occurred in which case A contains an error code. (&FF is 'end of file')

There are other MOS routines concerned with file handling but the ones given in Table 6.1 are those most often used. For example, there is a MOS routine, OSFILE, that performs the same action as the BASIC commands SAVE and LOAD. There are also a number of routines that do not apply to files stored on cassette. In particular, the needs of random access disk files are catered for by OSARGS (&FFDA). On calling this routine the X register should contain the address of the start of four memory locations in page zero. These are used to hold the input value, or the result of calling OSARGS, in the usual four byte integer format. Calling OSARGS with a channel number in Y will read the file's current position pointer if A=0, write the current position pointer if A=1, and read the file's length if A=2. A call to OSARGS with A = &FF will ensure that any alterations made to the file are actually written out rather than just sitting in a buffer. OSARGS also has a number of other functions including returning the code of the currently active filing system. This was used in the function FN file system that can be used from BASIC to discover which type of

file device is in use. There is nothing complicated about using these filing system routines as they provide the same set of operations as their equivalents in BASIC.

A disk sector editor

One of the most useful utilities that any disk user can possess is a *sector editor*. A sector editor will read in any sector of a disk and display it in hex or in ASCII characters and then allow you to write it back to disk after making any changes to the data that are necessary. This may sound like a difficult program but the disk filing system includes an extension to the MOS routine OSWORD to read or write a sector. Calling OSWORD with the A register set to &7F will read or write a sector according to the contents of a parameter block.

byte	meaning
0	drive number
1-4	address of sector buffer
5	3
6	&53=read sector &4B=write sector
7	track number
8	sector number
9	&21

This has to be set up before entering OSWORD and its address stored in the X and Y registers.

Using this information a sector editor is easy to write:

```
  10 REM SECTOR EDITOR
  20 DIM SEC_BUF% 255
  30 DIM INS_BLK% 50
  40 MODE 4
  60 PROCparm_get
  80 PROCsect_op
  90 PROCsect_print
 100 GOTO 60

1000 DEF PROCparm_get
1010 PRINT TAB(0,28);
1080 INPUT "Action Read/Write/Modify",A$
1090 IF LEFT$(A$,1)<>"R" AND
     LEFT$(A$,1)<>"W" AND
```

```
      LEFT$(A$,1)<>"M" THEN GOTO 1010
1100 IF LEFT$(A$,1)="R" THEN COM%=&53
1110 IF LEFT$(A$,1)="W" THEN COM%=&4B
1120 IF LEFT$(A$,1)="M" THEN COM%=0
1125 IF COM%=0 THEN ENDPROC
1130 PRINT TAB(0,29);:
     INPUT "Drive=",D%
1140 IF D%<0 OR D%>3 THEN GOTO 1130
1150 PRINT TAB(0,30);:
     INPUT "Track/Sector (TTSS)",T$
1160 IF LEN(T$)<4 THEN GOTO 1140
1170 T%=FNtrack(T$)
1180 S%=FNsector(T$)
1190 ENDPROC

2000 DEF PROCsect_op
2010 IF COM%=0 THEN PROCmodify:ENDPROC
2020 INS_BLK%?0=D%
2030 INS_BLK%!1=SEC_BUF%
2040 INS_BLK%?5=3
2050 INS_BLK%?6=COM%
2060 INS_BLK%?7=T%
2070 INS_BLK%?8=S%
2080 INS_BLK%?9=&21
2090 A%=&7F
2100 X%=INS_BLK% MOD 256
2110 Y%=INS_BLK% DIV 256
2120 CALL &FFF1
2130 ENDPROC

3000 DEF PROCsect_print
3010 LOCAL I
3020 PRINT TAB(0,0);
3025 FOR I=0 TO 15
3026 PRINT TAB(3+I*2 MOD 32);~I;
3027 NEXT I
3028 PRINT
3030 FOR I=0 TO 255
3035 IF I=(I DIV 16)*16 THEN
     PRINT TAB(1);~(I DIV 16);
3040 PRINT TAB(3+(I*2 MOD 32));
     ~(SEC_BUF%?I);
3050 NEXT I
3060 PRINT:PRINT
3070 FOR I=0 TO 255
```

```
3080 IF SEC_BUF%?I<32 OR SEC_BUF%?I>127
     THEN PRINT TAB(I MOD 32);"-"; ELSE
     TAB(I MOD 32);CHR$(SEC_BUF%?I);
3090 NEXT I
3100 ENDPROC

4000 DEF PROCmodify
4010 LOCAL A$,I,D
4015 PRINT TAB(0,29);
4020 INPUT "Which byte row/col",A$
4030 I=EVAL("&"+A$)
4040 IF I<0 OR I>255 THEN GOTO 4010
4050 PRINT TAB(0,30);
4060 INPUT "New value=",A$
4070 D=EVAL("&"+A$)
4075 IF D<0 OR D>255 THEN GOTO 4050
4080 SEC_BUF%?I=D
4090 CLS
4100 ENDPROC

9000 DEF FNtrack(T$)
9010 =EVAL("&"+MID$(T$,1,2))
9020 DEF FNsector(T$)
9030 =EVAL("&"+MID$(T$,3))
```

The main program consists of the usual calls to procedures that do the actual work of reading and writing sectors. PROCparm_get asks a number of questions about what tasks the program should perform. Answering the first question with either 'R' for 'Read a sector' or 'W' for 'Write a sector' causes the program to prompt for the drive number, track and sector number involved. Answering the first question with an 'M' for 'Modify' causes the buffer editor to be called so that a single byte can be changed. PROCsect_op actually performs the sector read and write operation or it calls PROCmodify so that the contents of the sector buffer SEC BUF% can be changed. Finally PROCsect_print prints the current contents of the sector buffer in hex and, where possible, in ASCII. Notice that the track and sector number has to be typed in as a single hex number, that is &F05 specifies track F sector 5. Also notice that while this program will allow a user to change any given sector it doesn't check to see if the changes are in any way sensible. For example, it doesn't give an error message if you try to access track 79 on a 40 track disk! A sector editor is extremely useful in recovering data from crashed disks, etc., but if it isn't used with care it can itself be the cause of crashed disks!

Chapter Seven
Making Programs Work

So far we have tackled the subject of advanced programming from the programmer's point of view. However, there are people other than programmers interested in programs. In particular, it is all too easy to forget the intended user, especially when the program is in some way difficult or ingenious. A program that isn't used is a wasted effort and anything that you can do to encourage people to use and trust your programs is well worth the effort.

Users are influenced by the external features of your program, that is, what it does and how it goes about it. Thus, in addition to good internal structure, a program must be reliable and convenient to use. Unfortunately there is no programming method that will guarantee that a program is reliable and convenient to use. Using modular and structured programming certainly helps in reducing the number of bugs in a program that result from confusion over how the program works. It also makes the location and elimination of bugs easier. But unless you spend time detecting the presence of bugs by testing the program, it is likely to be unreliable.

In some ways the main benefit of using the sort of programming methods described earlier m thus book is that they give you the time to concentrate on other more demanding aspects of programming. Instead of making hard work of producing a program that is only just adequate, a programming method should free you to consider overall program design and reliability. This is an important shift in emphasis, as good design and reliability are the areas where programming skill really counts. However, while it is true that there is no programming method or system that will guarantee well designed reliable programs, this doesn't provide a reason for ignoring the problem or for taking a sloppy approach. There are principles of programming testing and debugging that will ensure that the time that you do spend on program reliability is effective. There are also ways of evaluating the success of your program in terms of ease of use, and then improving it in the light of this evaluation. All in all, it seems fair to sum up the current situation by saying that the art and craft of programming lies in the testing and debugging of programs, and is measured by the ease of use of the resulting products.

The natural history of bugs

Bugs - misbehaviour of programs - are a sad fact of the programmer's and computer user's life. Only the very simplest programs can be said to be 'bug-free' and even then such statements usually only have the force of a 'hope' rather than an assertion of fact. To say that a program is bug-free usually means nothing more than no program misbehaviour has been observed for a 'period of time'. Of course as the 'period of time' increases the level of confidence in the statement that a program is bug-free also increases, but it never reaches certainty. (Theoretically there arc methods of proving that a program is correct based on Boolean logic but cum ently there is nothing of any practical value.)

The state of the art in reliable programming is that after generous testing one can say that the number of bugs remaining is likely to be small and that they are likejy to be relatively unimportant. It has been said that if the products of other branches of engineering were as unreliable as a typica] program. bridges would be falling down as fast as they could be built and motor cars would need servicing once a day! This may seem a very gloomy picture to paint at the start of a chapter that deals with program reliability but the point is that most programs are not tested and debugged at at! well. Although it is impossible to claim that all the bugs have been removed from a program, it is certainly possible to improve on the current situation. It is the typically undertested program of today that has caused programming to be categorised as the most unreliable branch of engineering! It is important not to despair because the goal of a bug-free program cannot be attained with certainty; what matters is that a program is refiable enough for failure to be the rare exception rather than the regular rule!

All this discussion of the inevitability of bugs doesn't really provide any idea of the type of failures that can occur. As a program is nothing more than a very precise set of instructions it is subject to the same types of failures that are found in written English. In general BASIC bugs fall into three categories:

(1) Incorrect use of BBC BASIC or syntax errors. For example, writing RINT instead of PR INT. Errors of this sort are the programming equivalent of spelling mistakes in English.

(2) Misunderstanding or confusion about what a part of the program actually does. For example, you may think that a variable is being used to keep a total but because another part of the program uses it, this is not the case. in English this would be equivalent to not saying what you intended to say.

(3) Not catering for a wide enough range of ways that the program will be used. If a program is intended to be used in a very exact way then it is up to the program and not the user to ensure that these conditions are met. For example, if a program is designed to work out the roots of a quadratic

equation it should not {ail because the user enters a cubic equation for it to solve! The proper response to such a request is to inform the user that only quadratic equations can be solved, but most programs in this situation would stop with a meaningless (to the user) error message. In English this is equivalent to being vague or ambiguous.

The first two types of error are in some ways more fundamental than the third in that a program that doesn't contain any of the first two types of bug will work as the programmer intended. I he third type of bug really concerns what happens when the program is used in ways that are outside the conditions that the programmer had in mind while writing the program. This is often referred to as the *robustness* of a program. The more robust a program is, the better it behaves in a wider range of situations. Unfortunately the attitude that most programmers take to this sort of 'creative' use of a program is that any error messages or strange results are the sole fault of the user. This is a very limited attitude towards programming that most users find difficult to accept- A good product should behave well in all circumstances, not just in the ones that the programmer has taken into account.

Although the reliability of programs under ail conditions Is important it is better to deal with methods of finding and eliminating the first two types of error first. Ensuring program robustness uses a very different set of techniques that are described towards the end of this chapter (see the section *Programs fit to use*).

Bug detection

The first thing to say about debugging is that it consist of two distinct phases of 'bug detection' and 'bug location'. It is one thing to know that a bug exists within a program- that is, to have detected it, and quite another to know what its cause is, that is, to have located it. Although these two phases are clear enough many programmers confuse the two and tend not to realise that there are different methods used to detect and locate bugs.

Given a program that has reached a point in its development where the programmer concerned believes that nothing extra needs to be added, the next step is bug detection. That is, the program has to be tested to discover if it performs as intended. The most common method of doing this is for the programmer simply to use the program for a while and as long as it doesn't crash or do anything obviously wrong it will be passed as working. However, this unsystematic approach to program testing is both inefficient and unreliable. The programmer may spend a long time testing the program by using it to do the same sorts of things a great many times. Obviously the most efficient use of time spent testing involves making the program do as many different things as possible. Looked at in this way, testing is about finding sets of input data that take the program through as many different situations as possible. The problem is how to construct

such sets of test data and this brings us back to the subject of the flow of control diagram introduced in Chapter Two.

Obviously, it is possible to test a program using such a limited set of test data that some parts of the program are never used. If there are parts of a program that haven't been used during testing, then no matter how long was spent testing the program it is only partially tested. An unused line wan untested line and can contain any number of bugs. Another way of thinking about this is that complete testing involves following each path through the flow of control diagram. At each IF statement the flow of control line divides into two alternative paths and the test data should include a set of values that tests each path. You can think of the flow of control diagram as a map that should be used to explore the program.

It is also important to test all parts of the program equally. Bugs tend to hide just as often in what the programmer might think of as an easy part of the program as in a hard part. In fact the difficulty of implementation of a part of a program seems to have very little to do with where bugs are found. However, programmers tend to over-test the sections of a program that were troublesome during the development of the program and ignore those that were quickly dealt with. If anything, a section of program that was quick and easy to implement is *more* likely to harbour bugs because it has been scrutinised less. The only sensible attitude is to try to forget which sections of the program were easy and which were difficult, and make sure that testing uses every part of a program.

Another problem inherent in testing is that many cases seems so trivial as to be hardly worth thinking about. The question is 'how do you know that you have found a bug'? If the program is a game or something similar, then a bug is usually immediately apparent, but in other cases it can be much more difficult to decide if the result is correct. It is important that before you try a program with a particular set of test data you predict what you expect to see in other words, you work out the correct result. The reason for this emphasis on predicting the outcome before trying the program is that it is all too easy to convince yourself that whatever the program actually does is correct Sometimes the need to believe that a program is working correctly can defeat common sense! If it is difficult to predict the correct results of a program, for example, solving a very difficult set of equations, then there are three ways of dealing with the problem:

(1) You should first try some simple test data and estimate (even if only approximately) what you expect to see.

(2) You should then try some more difficult data and make sure that the results change in the way that you expect.

(3) If there is another program that calculates the same or similar result

Without some independent method of checking the results of a program there is no guarantee that it is correct and any results it produces have to

be taken on trust. Claiming that a program is working something out using a correct method correctly applied is not a proof that the result is correct! The weak link in the argument is that you can never be sure that the method is correctly applied unless you have checked that the result is correct!

Practical testing

In theory testing is easy. All you have to do is:

(1) Construct sets of test data that take the flow of control down each of the possible routes through the program.

(2) Predict what you expect to see as the result of each set of test data and then run the program using the test data.

(3) Compare the results that you get with the results that you predicted any discrepancy and you have detected a bug

In practice, even if you could follow this plan for complete testing it would still leave some bugs undetected! The trouble is that the number of test sets of data that are necessary for medium sized programs is larger than can normally be handled during routine testing. The best that can usually be managed is to make sure that at some point during testing each branch of the flow of control diagram is tried out at least once. (Notice that full testing involves taking every possible combination of branches in the flow of control.)

One type of bug that even complete program testing will not detect is related to using up the computer's 'resources'. For example, you may have tested a section of a program that plots a shape on the screen and have used sufficient test data to explore all of the possible routes through the program but it's not until you try to plot the shape near the edge of the screen will you discover the bug caused by part of the shape going off the screen. In a sense this error is the result of trying to use more of a resource i.e. the graphics screen, than the computer has. Another, and common example, of a resource error is to be found in the misuse for FOR loops. If you indulge in the practice of jumping out of FOR loops before they are finished then you will eventually generate an error. The reason for this is that each FOR loop that you leave unfinished is still 'active' and BBC BASIC can only handle a maximum of ten FOR loops at any time. When you test the program this bug may not show itself because the test data doesn't cause the FOR loop to be left sufficiently often. However, as soon as testing is over and the program is being used 'for real' the intensity of use is much higher and hence the message "TOO MANY FORS" appears and the program crashes. (The same problem can occur with REPEAT . . . UNTIL loops.)

The only way of finding general resource bugs during testing is to

include sets of extreme test data. Of course what 'extreme' means depends very much on the nature of the program In a graphics program you should always try to make the program malfunction by extremes of position and size in the objects being drawn. In a numerical program you should try very small and very large values. However, the best way of avoiding the too many FOR or REPEAT . . . UNTIL loops bug is to read through the program and make sure that there are no GOTOs transferring control into or out of such loops. Of course if you have followed the principles of structured programming there will be very few GOTOs and none of them will transfer control into or out of FOR . . . UNTIL loops on purpose!

Some programmers and some users have the odd gift of being able to invent test data that will detect bugs. However, if you can adopt the attitude that program testing is only successful when bugs are detected then it is surprising how easy it is to think of data that makes a program fail. If this sounds obvious it is worth pointing out that most programmers are disappointed to find a bug during testing and this attitude probably subconsciously affects the test data they select. Remember that the object of testing is to find bugs not to prove that your program is working. A testing session that finds no bugs is more or less wasted time.

Bug location

Once you have detected that a bug exists the question is how to find out what is causing it. In other words, after bug detection comes bug location. The most commonly employed method of finding a bug is just looking at the program listing! The surprising fact is that this works in most cases. Often the bug causes the program to crash with an error message that names the line that contains the error. If the line named in the error message doesn't contain an obvious bug then the next thing to do is to examine lines that define variables used in the line. For example, if the error message uses the variable 'A' then examine lines that come before the line in question that also use 'A'.

So many bugs are obvious once they have been detected that some programmers never learn what to do when a bug is difficult to locate. If just looking at the program listing fails to reveal the problem after a few minutes then the chances are that 'just looking' will not work at all. If you think that debugging involves studying a listing into the early hours of the morning then the chances are that you have been wasting a lot of time. Debugging is an activity; it needs both the program listing and the computer to work efficiently. In fact the technique of debugging is an extension of program testing. You should predict what the program should be doing and compare it to what the program is actually doing. The first place that you find a discrepancy is the location of the bug. The difference between the predictions that you make during testing and debugging is the level of detail. The predictions used during testing merely concern the outward behaviour of the program, while those needed in debugging have

to predict the internal behaviour of the program. To be precise you have to specify the order in which you expect the statements to be carried out, that is the flow of control, and the values that you expect to find in each variable. If you find that the order of execution is not what you expected or that a variable has a different value stored in it, even if you have not found the actual bug you will be much closer to it.

This method of prediction and examination sounds easy enough but how do you find out the actual order of execution and the content of variables? In most versions of BASIC there is no problem with either task but BBC BASIC presents a number of difficulties. However, with the assistance of the 'trace/debug' program given in Chapter Nine debugging BBC BASIC programs is easier than any other! In assembler the problem is no more difficult and in fact the fundamental method of debugging - the *break point* - was introduced by assembly language programmers long before high level languages were invented.

BASIC and assembly language break points

A break point is a temporary halt in a program so that the order of execution can be determined and variables examined. In BBC BASIC a break point can be inserted into a program by using the STOP command. The STOP command baits the program and prints the message STOP at line xxxx where 'xxxx' is the line number of the STOP command. Thus by placing a number of STOP commands in the program you can discover the order of execution of its statements. When debugging never assume that the program is taking a particular route through the flow of control diagram; place STOP commands along the way to verify that it is. Once the program has stopped you can also discover the values stored in variables by using direct PRINT statements. Once you have finished examining variables the program can be restarted using a GOTO yyyy command entered in direct mode, where yyyy is the line number of the next instruction following the STOP command. The only trouble is that although you can restart a program following STOP using GOTO, the BBC Micro will forget about any FOR loops, REPEA T. .. UNTIL loops or procedures that it was in the middle of when the STOP was encountered. This means that when you restart the program by using GOTO the program will crash as soon as it tries to execute the NEXT, UNTIL or ENDPROC statement. Thus you cannot use STOP/GOTO to place break points within FOR loops, REPEAT. . .UNTIL loops or, most importantly, within procedures. This is an unfortunate limitation on what in other versions of BASIC is the main debugging tool.

The simplest way to overcome this problem is to add PRINT statements to the program that tell you what the current line number is and print out any variables of interest. This sound foolproof but in practice the debugging messages tend to get lost on the screen or rapidly overprinted by other messages. The best solution is to use a specially written debug

procedure:

```
9000 DEF PROCdebug(L_NO,V1S,V2$,V3$)
9010 LOCAL X,Y
9020 X=POS:Y=VPOS
9030 SOUND 1,-15, 100,10
9040 PRINT TAB(0,0);"At line ";L_NO;
9050 PRINT V1$;"=";EVAL(Vl$);
9060 PRINT V2$;"=";EVAL(V2$);
9070 PRINT V3$;"=";EVAL(V3$)
9080 REPEAT:UNTIL INKEY(-106)
9090 PRINT TAB(X,Y);
9100 ENDPROC
```

This procedure will print the line number and the names and values of up to three variables on the top line without disturbing the current cursor position. It will then wait until the COPY key is pressed before returning control to the program, For example, try:

```
10 FOR I=1 TO 100
20 PRINT I,SQR(I)
25 PROCdebug(25,"I","","")
30 NEXT I
40 END
```

Line 25 uses PROCdebug to inform the programmer that line 25 has been reached and the current value of I.

Once you have seen a procedure like PROCdebug and used it a few times you will soon be adding improvements of your own. For example, it could dump all the variables used by the program. (If you want to inspect such a routine, see the heap dump program in *The BBC Micro.- An Expert Guide*.) You could even make it prompt you for the names of the variables that you would like to examine, but of course you would have to be careful about screen layout!

If you know BBC BASIC well you might be wondering why the TRACE command has not yet been mentioned as a way of following the flow of control through a program. Following TRACE ON the line number of each statement is printed just before it is obeyed. The trace can be stopped by using TRACE OFF. Both statements can be entered in direct mode or included within a program. For example. you could include a TRACE ON command just before an IF statement to discover which way the flow of control went and turn it back off with a TRACE OFF immediately afterwards. Used in this limited way TRACE ON and TRACE OFF are extremely useful additions to the range of debugging aids but if the trace is switched on at the start of a program and left on the sheer quantity of information printed on the screen is overwhelming. A

program that enables the flow of control to be traced through a complete program without any problems is given in Chapter Nine.

Debugging 6502 assembly language programs follows the same fines as for BBC BASIC. However, there is the added problem that assembly language is not run in such a 'protected' environment as the BASIC interpreter provides for BBC BASIC. In other words, it is possible for an assembly language program to run 'wild' and obliterate system variables, etc., in such a way that the only method to recover control of the machine is to switch it off and on. The best way of handling the testing and debugging of an assembly language program is to use a special 'debug' program that provides facilities such as single instruction execution and full trace. However, for small programs the same method described for BBC BASIC can be used. That is an assembly language 'debug' subroutine can be written that will print out the current address and contents of all the registers whenever required. Obviously in assembly language debugging the address of the last instruction to be executed is needed to follow the flow of control but values stored in variables are less important than the current contents of the registers. The reason for this is that generally assembly language variables arc changed by way of storing the registers. You should be able to see that an assembly language version of PR OCdebug is not difficult in principle. However, in practice it turns out to be quite a long program!

The common errors of languages

The earlier sections of this chapter have supposed that when you are looking for a bug you have no idea where it might be, except for the evidence provided by predict and examine type debugging. However, in practice all computer languages have their 'common errors'. For example, as already mentioned the 'too many FOR loops' error often occurs in BBC BASIC because of the accidental jumping out of unfinished FOR loops. It obviously makes sense to look for the most common errors before looking for new and exotic bugs!

In all versions of BASIC the greatest source of bugs is the line numbering system. This has already been raised as a problem in Chapter Two, but it is worth describing the types of effect that mistyped line numbers produce. A single mistyped digit that is part of a fine number will result in a line of BASIC being added to the program at an effectively random position! This random scattering of legal statements within a program gives rise to a type of error that is typical of BASIC. Whenever a program starts to behave in apparently illogical ways it is worth checking that there are no stray lines embedded in the program. This type of check is one of the few debugging activities that is best conducted with a program listing away from the computer. The problem of eliminating

misplaced lines in the first place is very difficult. If you are entering a large number of lines at one time a mistyped digit will not only add a line at another position, it will also leave a gap in the program where you intended the line to be. In practice, it is the absence of a line that was recently typed that is detected first. Unfortunately the usual reaction is to suppose that for some reason the machine just didn't accept the line. Because of this the line is just typed in for a second time and there is no immediate search for the version of the line that went missing! The misplaced line is found sooner or later but often after a great deal of time-consuming debugging. The rule is that you should always check that any line or block of lines that you have typed has been added to the program in the correct place. Any missing lines should not simply be retyped, but the program should be searched and the lines found and removed.

Another common BASIC bug the unwanted interaction between modules was introduced in Chapter Two. One of the most likely causes of subtle and difficult-to-find bugs is the use of the same variables by different procedures for different tasks. This is a problem that can be largely eliminated by using local variables wherever possible but even BBC BASIC sometimes suffers from this distressing bug.

The most common assembly language bugs are due to confusion over addressing modes. For example, it is all to easy to mean:

LDA #&70

and actually write:

LDA &70

Other assembly language bugs are often caused by misunderstanding how instructions set and affect the condition codes. Never assume that a branch is being taken just because it is self-evident. Check that the instructions just before the branch really do change the condition codes in the way that you expect. Apart from these two bugs, assembly language also suffers from the 'unwanted interaction between modules' problem encountered in BASIC.

Stepwise testing and debugging

The question still remains of what to do once you have detected and then located a bug. Should you fix the bug and carry on testing or carry on testing and put off changing the program until later? The answer to this question is not at all easy. If you change the program by fixing the bug then you are no longer testing the program that you started with and in theory at least you should go back and start testing all over again! If you think that this is rather extreme advice, it is worth saying that many embarrassing bugs result from making apparently minor last minute changes that cause problems which would never be guessed at in well

tested parts of the program. If you make *any* change to a program. your level of uncertainty about its correctness must drop back to where it was before you started testing. In practice, common sense has to be applied to which tests to repeat and which not to.

An advantage of using stepwise refinement is that the program can also be tested in a stepwise manner. As each module is added to the program, its specific action should be tested. In this way, by the time the program reaches its final form, most of the worst bugs are likely to have been removed. However, this is not to say that stepwise testing will produce a bug-free program As already explained in previous sections, some bugs arise from unwanted interactions between modules and to detect these it is important to test the whole program. Thus, stepwise testing and debugging should be used but not as an excuse to limit or avoid final testing!

Programs fit to use

Reducing the number of bugs in a program is a difficult and time consuming task but it is not enough to ensure the success of a program! Indeed, a well debugged program is the minimum requirement For a program to be successful it has to be easy to use and well behaved. It is virtually impossible to explain how to write programs that are easy to use. The best that can be done is to suggest guidelines and ways of evaluating the ease of use of a program. However, it is possible to describe ways of producing programs that are well behaved.

Essentially, a well behaved program should never crash or produce results that it was not intended to. A program that is free of bugs can crash in one of two ways. Firstly, the user can enter values that the program was not designed to handle. For example, entering a value that is too big, or too small or even of the wrong type. Secondly, an internal or external condition can cause an apparently correct program to fail. For example, if a string becomes longer than 255 characters then it cannot be stored in BBC BASIC and the program will crash. Another example is trying to read a file that doesn't exist, or a disk error.

The first type of crash can be avoided by testing all input values for the correct type and range. This is not as difficult as it sounds. Every INPUT statement should be followed by an IF statement that checks the input values and either issues an error message reporting out of range values or simply transfers control back to the input statement. Any INPUT statement that is not followed by such a check for illegal values is a potential route for garbage into the program - and every programmer is familiar with the well known saying 'Garbage In. Garbage Out'. In reality this saying is more often realised as 'Garbage In, Nothing Out' because garbage data usually crashes the program! You can see examples of input

protection in most of the programs in this book.

The second type of crash is very difficult to avoid. In many ways the causes of such crashes are very like ordinary run-of-the-mill bugs. The difference is that unlike bugs they are the consequence of perfectly correct BASIC. For example, the part of a program that tries to read a file that doesn't exist is perfectly correct BASIC; it just happens to be working in an environment where it cannot carry out the task it was designed for. Sometimes R is possible to detect and program around such situations, but often the biggest problem is realising that such an event can happen. Often it isn't even practical or possible to check for the conditions that cause the crash for example, the work involved in checking that a string will not become longer than 255 characters as a result of the next operation is possible but not practical. The only practical way of handling such exceptional circumstances is to use the ON ERROR 'statement' command. When an error occurs the BBC BASIC interpreter will carry out the 'statement' part of the most recent ON ERROR command. Thus, ON ERROR can be used to detect and handle errors as they happen. That is, instead of trying to write BASIC statements that will detect situations which will cause a crash before they happen, the ON ERROR statement can be used to detect the situations by the fact that it does cause the system to crash! However, the problem with detecting a crash after the situation that caused it has already occurred, is that it is very difficult to 'backtrack' to the point just before the crash.

To be precise, after an ON ERROR statement has been acted upon all of the variables have the value that they had just before the error occurred but any active FOR loops or REPEAT . . . UNTIL loops are finished and any FROG calls are 'forgotten'. This means that following an ON ERROR statement you cannot transfer control back into a FOR or REPEAT . . . UNTIL loop or procedure. This limits the way that you can restart the program to a transfer of control back to a point in the main program. This is very restricting and often it is better simply to issue an error message explaining the problem that has arisen and then re-run the entire program by using RUN as a program statement. To enable the error handling routine to discover the nature of the error, BBC BASIC provides the variables ERR, which holds the error number of the last error, and ERL. which holds the line number that the last error occurred in.

Following this description you should be able to see that error handling m a BBC BASIC program is very simple and limited. An ON ERROR GOTO xxxx statement should be included at the start of the program where 'xxxx' is the line number of an error handling routine. The error handling routine should test ERR and ERL to find out what sort of error has occured and where. On the basis of this test messages may be issued, some variabies initialised and some files closed. Then the error handling routine should use a GOTO statement to transfer control to a procedure call within the main program but outside any FOR or REPEAT .. .UNTIL loops. Because of the need to use of GOTOs and line numbers it is better to add the ON ERROR type of error handling at a point where the

program is very nearly finished. This also has the advantage that during testing you can keep notes of any crashes that the error handler should deal with. Often, however, the only solution is for the error handler to make any valuable data safe and then offer the user the chance to re-run the program. For example. if you detect a disk error within a text processing program it is unacceptable not to try to handle the error, but it is equally unacceptable to deal with it by losing am, text that was entered and re-running the program from scratch (there are a number of commercial text processors that do behave in this way in response to relatively minor errors). The correct way of handling such an error is to give the user the opportunity to change disks and save the text immediately. The rule is that during all error handling the main concern should be to minimise the user's wasted effort and this usually means minimising the loss of data or results. The applications programs in the remainder of this book have all been written with crash- proofing in mind.

Given a fully debugged and crash-proofed program the only remaining consideration is its ease of use. As mentioned in the introduction to this section it is wry difficult to say in genera! what makes a program easy to use. The range of user 'interfaces' found in programs is too wide to comment on in detail but it seems obvious that for a program to be easy to use it must tend to do things in the order that the user requires. It should also give the user as much freedom to determine the course of action as possible but without forcing the user to specify everything in minute detail. Programs that drag the user through tasks in an order determined solely by the programmer convey a sense of restriction and claustrophobia soon sets in. However. programs that are completely free and respond to the user's very detailed commands are very tedious to use after a while. The way to find out if your program does things in the right order is to take note of the sort of mistakes that users make in the early stages of running your program. If you ask for the maximum value followed by the minimum value and users keep on entering the minimum value followed by the maximum value then it is your program that is at fault! Remember the user is (nearly) always right.

Apart from watching the way users react to your program there are some general points that are worth keeping in mind while you are designing and implementing it.

(1) Try to keep the screen display dear and uncluttered. It is important to realise that a screen display you have developed in the course of the program will look simple to you because it is familiar, but to a new user it might appear a jumble

(2) Use colour and sound to draw attention to the important aspects of your program, screen display or whatever. As the important aspects are likely to be few in number, this implies that sound and colour should be used sparingly. For example, do not use a different colour and sound effect for every message. Highlight only the most important messages on

the screen and only use sound to draw attention to a mistake or to reinforce very important points. Loud, colourful programs quickly become tedious.

(3) Always ask clear and unambiguous questions and do not use jargon unless it is the user's jargon. For example, do not ask 'Dimension of array?' when you mean 'How many entries?' If your program is intended for users that have their own jargon then it makes the program more friendly if you use their jargon. However, make sure that your idea of their jargon is up to date there is nothing less likely to instil confidence than using a term that has been redundant since 1800!

(4) Never present too much information at once and never allow information to scroll off the screen before the user can read it. Make your program respond to single key presses that indicate when the user has finished reading whatever information is on the screen. For novice computer users a slower rate of printing information on the screen is also helpful. The usual BBC rate of printing can convey the impression that the computer is so fast they cannot keep up! It is also worth nothing that messages given in upper- and lower-case are much easier to read than upper-case only.

(5) Always try to allow the user to correct mistakes and abort program actions. Always use INPUT rather than GET$ or INKEY$ so that the user can correct entries with the delete key. Allowing the user to abort program actions is much more difficult. For example, how often have you had to sit through a program doing something that you didn't really want it to do because you entered a command by mistake and can find no way to stop it! If at all possible it is a good idea to monitor the pressing of the ESCAPE key using the ON ERROR statement. If the ESCAPE key is pressed then the error handler will be entered with ERR set to 17. Following this, the error handler's problem is to work out how to stop the program's current action without causing a real error and then how best to return control to the user.

(6) Finally, do not give error messages that are either too technical or too friendly. That is, messages such as "INTEGER OVERFLOW CONDITION ENCOUNTERED" and "YOU GOT IT WRONG" are equally to be avoided. Try to inform the user of exactly what is wrong without going into too much detail.

Similar guidelines could be added indefinitely, becoming increasingly more specialised. The most important thing, however, is to think about how the user wants to do the job and then try to make sure that your program helps rather than hinders. In the final analysis, it is the user who either likes or dislikes your program so take notice of any friendly users you can find to try your programs!

Chapter Eight
A Spelling Checker

This chapter and the following three each present a large applications program. The first of these is a 'spelling checker' which can, in principle, be implemented using nothing but BASIC. In practice, however, it is worth using some assembly language to achieve a reasonable performance. The program in Chapter Nine an execution tracer needs to be implemented entirely in assembly language because of the way that it fits into and modifies the workings of the BBC Micro's system software. The program in Chapter Ten is another example of an all assembly language program that again fits into the existing system software but it is of a different character to the execution tracer in that it uses the BBC Micro's interrupt and event handler. Chapter Eleven is devoted to the implementation of a 6502 disassembler - written exclusively in BASIC.

None of these programs have been refined to the point where they could be considered completely finished. (Something that all advanced programmers realise is that no program is ever entirely finished!) But they are taken to a stage of development where they do something useful in thia sense they are finished all but the frills. However, if you have been following the programming philosophy outlined in this book you will soon realise how much time and effort programming the 'frills' takes!

A spelling checker – design specification

The first thing that you should do before starting work on any program is to consider in as much detail as possible what the program is intended to do. This establishes a clear objective that your finished program can be judged against. Although it is often necessary to modify initial objectives in the light of what is possible in a reasonable amount of time, at least you will know what you are sacrificing for a working program. The second thing that you should do is to consider general ways of achieving your objective. In other words, you should try to work out rough outlines of how program would go about the task. The more ways of doing something that you can think of at this stage the better. It is also important not to get

bogged down with detail at this early stage. What you should aim for is an English description of the way the program will work that contains enough information for you to identify the data structures involved.

The spelling checker is an especially large and complicated program but its full objective is easy to state:

> Check each word in a text file against a list of correctly spelt words (a dictionary).

> Words that are not found in the dictionary are either new words or misspellings.

> To discover which, each word is presented to the user and a decision is requested.

> New words can be added to the dictionary so that the program can 'learn' to spell them.

> Misspelt words should be identified in the text file and the user given the opportunity of correcting them.

From this specification it is not difficult to see that a spelling check program has four actions to perform:

> (1) look up the words in a dictionary
> (2) ask the user to identify misspellings
> (3) add new words to dictionary
> (4) correct misspellings in the text

To make the program suitable as an example it makes sense to drop the last action and simply print out a list of misspellings in the text so that the user can use a text processor} editor to make corrections. This is not at all unreasonable and produces a spelling checker that is almost as useful as one that implements the full specification.

The next question is how to go about the three actions that we are going to implement. Clearly the major problem is how to look up each word in the text file in a dictionary file. The straightforward approach is to read in each word from the text file and search through the dictionary for it. However, this approach is very time inefficient in that very common words would be looked up each time they occurred in the text. A much better method would be to read the entire text file and build a list of all the words used. This list is almost certainly going to be much shorter than the text file because common words such as 'the', 'and', etc., would appear only once in the list irrespective of the number of times that they occurred in the text. This reduction of the number of words also makes it feasible to store the list of words in memory rather than as a file.

Once this list has been built each word can be looked up in the dictionary in turn. This is a task that is made much faster if both the list of words and the dictionary are sorted into alphabetical order. If they are not

in order then the entire dictionary has to be searched for each word in the list. If they are in order then you can abandon the search for a word as soon as you have gone past the position that it should occupy in the dictionary. Each word that is found in the dictionary can be deleted from the list as we are only interested in the words that are not in the dictionary The rest of the program is just a matter of simple operations on the list of unidentified words.

It is important to realise that in a program of this sort it is vital to settle on a method that is efficient. As each word might have to be checked against a large dictionary file any slow operations will because of their repetition make the program too slow to use.

The first stage

The spelling checker program is so large that it makes sense to tackle it in a two parts the construction of the sorted list of words and then the dictionary search and updating, This is not stepwise refinement but it does provide the opportunity to get something working before going on to extend it. Attempting a smaller problem that can be extended to include additional desired features is a good strategy if you are working alone because it provides some reward for your efforts in a shorter time than by trying to write the full program.

Before the first stage of implementation we have to decide on a data structure suitable for the storage of a list of words. The most obvious choice is a one-dimensional string array with each word stored in a different element of the array. The disadvantage of this is that the management of the string array is left to the BASIC interpreter which, for this application, would mean wasting both time and memory. Once a string array has been rejected the only other possibility is to construct a data structure using a byte array. Since the list needs to be sorted into order as words are read in, the most obvious contender is a linked list. Using a linked list new words could be inserted into their correct position without having to move existing words to make room. Attractive though it is for this application, a linked list has one important disadvantage it is slow to search for a particular word. The reason for this is that the only search method that can be used is to start at the first word and examine each entry in turn until the word is found or until you have searched beyond the point at which the word would appear if it was present in the list. This searching method is known as a *linear search* and it is very inefficient when compared to alternatives such as a *binary search*. (For details of searching methods, see Chapter Eight of *The Complete Programmer*, Mike James (Granada, 1983).) As the creation of a sorted fist of words represents only a small part of the work of the program, the linked list does not appear to be an attractive option. Although it is more difficult to construct a sorted list using a BASIC string array, it is a much

easier data structure to apply advanced searching methods to. The obvious solution is to construct a special purpose string array using a byte array and keep the storage management within the program. To be more precise, the words can be stored in a byte array using the '$' indirection operator, If a word has to be inserted into the list all of the words that follow it have to be moved up to make room but this is more efficient than using BASIC to manage the array.

Now that the data structure has been decided upon the main program can be written:

```
 10 MODE 7
 20 SIZE%=5000
 30 DIM WORD% SIZE%
 40 FIN%=WORD%+SIZE%
 50 FILE_END=FALSE
 60 PROCinitialise
 70 PROCopen_file
 75 REPEAT
 80 PROCread_word(F)
 90 IF WORD$<>"" THEN PROCadd_word(WORD$)
100 UNTIL FILE_END
110 PROClist_words
120 STOP
```

Line 30 reserves 5000 memory locations as a byte array for the word list, i.e. a total of 5000 characters. The variable SIZE% is used to set the size of the array throughout the program so that this can be easily changed and FIN% is set to the address of the end of the array by line 40. PROCinitialise is included to take care of any initialisation that has to be done just once at the start of the program. PROCopen_file is responsible for getting the file name and opening the file. The resulting channel number is returned in F. The main structure of the program is a REPEAT . . . UNTIL loop formed by lines 75 to 100. This reads words from the file usfing PROCread_word and adds them to the list using PROCadd_word. Thd REPEAT . . . UNTIL loop is brought to an end by reaching the end of the file, signalled by FILELEND becoming TRUE (see Chapter Twelve for information on the use of Boolean variables). The IF statement in line 90 is necessary to allow for the possibility of the file coming to an end without PROCread word returning a word in WORD$. Finally PROClist_words is used to examine the list for correctness.

The second stage

The second stage of refinement presents no real problems.

```
1000 DEF PROCinitialise
1010 WORDCNT%=0
1020 CURRENT_END%=WORD%
1030 ENDPROC

2000 DEF PROCopen_file
2010 CLS
2020 PRINT TAB(1,5);
2030 INPUT "File name of text to be"'
     " proof read ",F$
2040 IF LEN(F$)>10 THEN
     PRINT"Name too long":GOTO 2020
2050 PRINT
2060 F=OPENIN(F$)
2070 ENDPROC

3000 DEF PROCread_word
3010 LOCAL CHAR%
3020 WORD$=""
3030 REPEAT
3040 CHAR%=FNread_cap(F)
3050 UNTIL NOT(FNseparator(CHAR%))
     OR FILE_END
3060 WORD$=WORD$+CHR$(CHAR%)
3065 IF FILE_END THEN GOTO 3110
3070 CHAR%=FNread_cap(F)
3080 IF FNseparator(CHAR%) OR
     FILE_END THEN GOTO 3110
3090 WORD$=WORD$+CHR$(CHAR%)
3100 GOTO 3065
3110 IF WORD$<>"" THEN
     WORDCNT%=WORDCNT%+1
3120 PRINT TAB(0,8);
     "Number of words =";WORDCNT%
3130 ENDPROC

4000 DEF PROCadd_word(WORD$)
4020 PROCfind_word(WORD$)
4025 PROCtime(2)
4030 IF FOUND% THEN ENDPROC
```

```
4035 PROCstart
4040 PROCmake_space(WPOINT%,LEN(WORD$)+1)
4045 PROCtime(3)
4050 $WPOINT%=WORD$
4060 ENDPROC

6500 DEF PROClist_words
6505 CLS:VDU 14
6506 PROCprinter
6510 WPOINT%=WORD%
6520 IF WPOINT%=CURRENT_END% THEN
     VDU 15,3:ENDPROC
6530 PRINT $WPOINT%
6540 WPOINT%=WPOINT%+LEN($WPOINT%)+1
6550 GOTO 6520
```

PROCinitialise and PROCopen_file are straightforward. The only real point of interest is that PROCinitialise sets CURRENT_END% to the start of the byte array. CURRENT_END% is used to mark the extent of the byte array that has been used to store words, i.e. it marks the division between the used and unused parts of the array.

The first procedure worth examination is PROCread_word. The main problem in implementing this procedure is the definition of a 'word'. You might think that a word was any group of letters enclosed by blanks. However, this is not a sufficiently wide definition to include words that are terminated by commas, full stops or other punctuation. The easiest way to define a word is to say that it is a group of letters enclosed by any of a number of legal separator characters. With this definition the algorithm of PROCread_word is:

   (1) Read characters until the first non-separator is found and then store it in WORD$.

   (2) Read characters until a separator is found and add each non-separator to WORD$.

Each of these steps can be clearly seen as loops in the procedure. The first is a REPEAT . . . UNTIL loop (lines 3030 to 3050) and the second is a additional loop with an exit point in the middle (lines 3070 to 3100). The exit point in the middle could be avoided and the loop turned into a REPEAT . . . UNTIL loop but in this case it is more natural to use the more complicated sort of loop. PROCread_word uses no additional procedures but it does use two functions. FNread_cap(F) returns the ASCII code of the next character read from the file after it has changed lower-case letters to upper-case. FNseparator(CHAR%) is an example of a predicate function in that it returns a value of TRUE if CHAR% is a

separator and FALSE otherwise (predicate functons are described in Chapter Twelve). PROCadd_word first checks to see it WORD$ is already in the list (using PROCCfind_word). If it is then nothing else needs to be done. If the word does have to be added to the list then PROCmake_space creates a 'gap' in the list by moving up all of the existing entries. PROCfind_word returns two results, a Boolean variable FOUND% that is TRUE if the word is in the list and FALSE otherwise, and WPOINT% which gives either the position of the word in the byte array or the position that it should occupy if it is not in the array.

The final procedure is PROClist_words and this uses WPOINT% to record the address of each word in the byte array. IF WPOINT% is the address of the start of the first word then PRINT $WPOINT%. will print the first word and WPOINT%=WPOINT%+LEN($WPOINT%)+1 is the address of the second word. The only procedure that PROClist_words calls is PROCprinter which asks the user if the words should be listed on the printer or not.

Third and fourth stages

The procedures to be defined at the third stage of refinement are PROCfind_word, PROCmake_space and PROCprinter. The functions yet to be defined are FNread_cap(F) and FNseparator. Of these PROCfind_word and PROCmake_space are the most difficult.

```
5000 DEF PROCfind_word(WORD$)
5020 FOUND%=FALSE
5030 WPOINT%=WORD%
5040 IF WPOINT%=CURRENT_END% THEN ENDPROC
5050 IF $WPOINT%>WORD$ THEN ENDPROC
5060 IF $WPOINT%=WORD$ THEN
     FOUND%=TRUE:ENDPROC
5070 WPOINT%=WPOINT%+LEN($WPOINT%)+1
5080 GOTO 5040

6000 DEF PROCmake_space(FROM%,AMOUNT%)
6005 LOCAL I
6010 IF CURRENT_END%=FROM% THEN
     GOTO 6050
6020 FOR I=CURRENT_END%-1 TO FROM% STEP -1
6030 I?AMOUNT%=?I
6040 NEXT I
6050 CURRENT_END%=CURRENT_END%+AMOUNT%
6060 IF ABS(FIN%-CURRENT_END%)<20 THEN
     PROCnospace
6070 ENDPROC
```

```
6600 DEF PROCprinter
6605 LOCAL A%
6610 PRINT TAB(0,4);
6620 INPUT "Do you want to use
     the printer (Y/N)",A$
6630 IF A$<>"Y" AND A$<>"N" THEN
     GOTO 6620
6640 IF A$="Y" THEN VDU 2 ELSE VDU 3
6650 ENDPROC

9000 DEF FNseparator(C%)
9010 IF C%=ASC(" ") THEN =TRUE
9020 IF C%=ASC(",") THEN =TRUE
9030 IF C%=ASC(".") THEN =TRUE
9040 IF C%=ASC(":") THEN =TRUE
9050 IF C%=ASC(";") THEN =TRUE
9070 IF C%<32 THEN =TRUE
9080 IF C%>127 THEN =TRUE
9090 =FALSE

9100 DEF FNread_cap(F)
9110 LOCAL C%
9120 C%=BGET#F
9125 IF EOF#F THEN CLOSE#F:FILE_END=TRUE
9140 IF C%>96 THEN C%=C%-32
9150 =C%
```

PROCfind_word simply searches through the list of words until it reaches the end of the list (line 5040), finds the word (line 5060) or reaches the point in the list where the word should be stored (line 5050). The body of the procedure is in the form of a loop with multiple exit points (three to be exact) all grouped at the start of the loop. In fact, the first two exit points could be combined into one, that is:

```
5040  IF WPOINTf%=CURRENT_END% OR
      $WPOINT%>WORD$ THEN ENDPROC
```

but the form used in the program is just as clear.

PROCmake_space is simple in principle. All it has to do is to use a FOR loop to move all of the elements of the array from FROM% to CURRENT_END% up by AMOUNT% bytes. However, in practice it is quite difficult to get all the details right. If you are writing procedures that perform complicated movements of data then it is a good idea to go through the steps involved on paper below trying to write any BASIC. It is possible for PROCmake_space to use up all of the byte array. This

condition is checked for in fine 6060 but the real question is what to do when the program runs out of space, In the tradition of stepwise refinement, the answer to this difficult question is put off until the next stage of refinement by calling a procedure PROCnospace!

The functions FNseparator and FNread_cap are not difficult to implement but they both contain important points of detail. FNseparator simply checks C% against each possible type of separator character. The separators in lines 9010 to 9050 are obvious but some text processors embed control codes in the text in place of blanks and so any code lower than 32 also has to be considered a legal separator (9070). The same holds for codes above 127 but it is far less common for these to occur. Notice that the advantage of using a function to determine whether or not a character is a separator is that it is very easy to change or add to the set of characters that are considered separators and without having to delve into the inner areas of the program! For example, after you have run the program on a few samples of text you will soon discover that there are two omissions. The characters '(' and ')' have to be considered valid word separators but it is left as an exercise for the reader to add them to FNseparator. FNread_cap reads in a character code from the file and then immediately tests to see if the end of file has been reached. [fit has, then it sets the FILE_END flag so that the rest of the program knows that there is no more data and actually closes the file. Line 9140 converts all lower-case characters to upper-case. This function could protect the rest of the program from trying to read data from a closed file by including the line:

9115 IF FILE_END THEN =0

but it is probably better to know that the program is trying to read from a file that it shouldn't and deal with the problem higher up in the structure.

The fourth stage in refinement is now just one procedure, PROCnospace. It is often said that what a program should do when everything is working is easy it's when errors occur that things get tough! The temptation is to write a procedure that gives an error message "NO ROOM" and stops the program. This is quick and easy for the programmer but the user would feel that the whole experience was a waste of time. A reasonable compromise is to inform the user of the problem and then proceed to process as many words as can be held in memory. This is what PROCnospace does. It displays a message for a short time, closes the file and sets FILE_END to TRUE so that the rest of the program will think that there is no more text to process.

```
 6700 DEF PROCnospace
 6710 PRINT TAB(0,10);"Not enough
      memory!!"
 6720 PRINT "Closing file - but will
      process"
```

```
6730 PRINT "words already read"
6740 FILE_END=TRUE
6750 CLOSE#F
6760 TIME=0
6770 REPEAT:UNTIL TIME>500
6780 ENDPROC
```

The dictionary

At the end of the last section the program was developed to the point where it would read a text file, construct a sorted list of words in the byte array starting at WORD% and then print the results either on the screen or additionally on the printer. After testing this version of the program on a number of texts you shouldn't be able to find any bugs (if you do then fix them!) although you should be able to find ways of crashing the program by specifying non-existent filenames, etc. The next stage in producing the program is to move on from this working, but extremely limited, version of the program by the addition of procedures to handle the dictionary file. Once again before committing anything to BASIC there are a few design decisions to be made. The main consideration is the form of the dictionary tile. Up to this point the program has been independent of the file system in use i.e. the program will work with either tape or disk. The problem with the dictionary becomes apparent as soon as you consider how it should be updated. If the dictionary file is to be kept in order then adding new words would mean some very complicated random access file handling. In fact it would be better to avoid the random access file handling by simply rewriting the entire dictionary file, adding the new words in their correct positions. Even this simpler method of adding words to the dictionary file requires two files to be open at the same time one for reading and one for writing and this is something that the tape system cannot usually handle.

As an alternative it seems worthwhile to drop the requirement that the dictionary file should be sorted into order. The advantage is simply that new words could be added on to the end of the file i.e. they could be appended. The disadvantage is that the search time is greater. However, this can be minimised by a number of changes to the search algorithm. Firstly the list of words stored in memory is smaller than the dictionary so it makes sense to read a word from the dictionary and search for it in the list nt words rather than vice versa. It also seems reasonable to physically remove each word that has been identified from the word list so that subsequent searches are on a smaller list. Neither of these two improvements produce a program that is as fast as one that uses a sorted dictionary but the advantages in terms of simplicity are great.

It would be possible to work out a scheme whereby new words could be added to the end of an existing cassette file but this would be fairly

complicated. The cassette system not only doesn't support random access files, it doesn't allow existing files to be extended. The simplest method of adding data to a cassette file is to write an additional file following it. That is the dictionary data would be stored on tape stored on tape not in one file but in as many files as needed named DICT1, DlCT2 and so on. You would have to introduce your special 'end of all files' marker so that you would know when you had reached blank tape but apart from this there are no real problems. However, this scheme entails a complication unnecessary to the essence of the spelling checker and the remainder of the program will assume that the disk filing system is in use. Changing it to work on the tape system is left as an exercise for the reader.

The first stage in adding the dictionary handling to the spelling checker involves changes to the main program:

```
 10 MODE 7
 20 SIZE%=5000
 30 DIM WORD% SIZE%
 40 FIN%=WORD%+SIZE%
 50 FILE_END=FALSE
 60 PROCinitialise
 70 PROCopen_file
 75 REPEAT
 76 PROCstart
 80 PROCread_word
 85 PROCtime(1)
 90 IF WORD$<>"" THEN PROCadd_word(WORD$)
100 UNTIL FILE_END
110 PROCreport
120 PROClookup
130 PROClist_words
140 PROCadd_dict
150 END
```

PROCreport informs the user that reading the text file has finished and that the dictionary lookup is about to begin. PROClookup, as its name suggests, performs the actual look up operation and deletes any words that have been found from the word list. PROClist words is the procedure developed for the first version of the program that prints the list of words on the screen and additionally on the printer. Now that it follows PROClookup it gives the user a list of only those words that are not in the dictionary and are therefore possible misspellings. Finally PROCadd_dict performs the task of adding words to the dictionary.

In the second stage of refinement only PROClookup and PROCadd_dict are at all complicated:

```
7000 DEF PROCreport
7010 CLS
```

```
7020 PRINT TAB(0,5);"Text file has been
     read."
7030 PRINT "A total of -";WORDCNT%;
     " words"
7040 PRINT
7050 PRINT "Ready to check spelling
     against"
7060 PRINT "dictionary file."
7070 ENDPROC

7100 DEF PROClookup
7110 F=OPENUP("DICT")
7115 FILE_END=FALSE
7120 REPEAT
7130 INPUT#F,WORD$
7135 IF WORD$="" THEN FILE_END=TRUE
7140 PROCfind_word(WORD$)
7150 IF FOUND% THEN
     PROCremove(WPOINT%,LEN($WPOINT%)+1)
7160 UNTIL FILE_END
7180 PTR#F=PTR#F-2
7190 ENDPROC

8000 DEF PROCadd_dict
8005 LOCAL A$
8010 PRINT "Do you want to add correct"
8020 PRINT "words to dictionary (Y/N) ";
8030 INPUT A$
8040 IF A$<>"Y" AND A$<>"N" THEN
     GOTO 8010
8050 IF A$="N" THEN CLOSE#F:ENDPROC
8060 WPOINT%=WORD%
8070 IF WPOINT%=CURRENT_END% THEN
     PRINT#F,"":CLOSE#F:ENDPROC
8080 WORD$=$WPOINT%
8090 PRINT WORD$
8100 INPUT "A(dd) or I(gnore)",A$
8110 IF A$<>"A" AND A$<>"I" THEN
     GOTO 8100
8120 IF A$="A" THEN PRINT #F,WORD$
8130 WPOINT%=WPOINT%+LEN($WPOINT%)+1
8140 GOTO 8070
```

PROClookup assumes that the dictionary file is always called DICT, opens it and proceeds to read words from it. Notice that the DICT file is read using INPUT and written using PRINT so that the BBC Micro's

filing system looks after the internal format of the file. The actual search of the word list is performed by PROCfind_word which was developed for the first version of the program and so alrendy exists. The only new procedure needed is PROCremove which is the logical opposite of PROCadd_word in that it removes a word by shifting all of the entries in the word list down to close up the space that the word occupied.

PROCadd_dict is a very simple routine that asks the user if each word in the word list should be added to the dictionary. Notice that there is no need to open the dictionary file because PROC1ookup leaves the file open and positioned at the end of file marker ready for new words to be added.

The final stage of refinement involves writing PROCremove which is closely modelled on PROCadd_word.

```
7500 DEF PROCremove(FROM%,AMOUNT%)
7510 LOCAL I
7520 FOR I=FROM% TO CURRENT_END%-AMOUNT%
7530 ?I=I?AMOUNT%
7540 NEXT I
7550 CURRENT_END%=CURRENT_END%-AMOUNT%
7560 ENDPROC
```

This completes this version of the program. All that is necessary to 'start the ball rolling' is the creation of first dictionary file using:

```
10 *SAVE DICT 0000 8000
20 F=OPENUP("DICT")
30 PRINT#F,""
40 CLOSE#F
```

This results in a single entry, the word "A", being stored in the dictionary lie DICT. Following this the program can be run and the dictionary extended by the addition of correctly spelt words from a range of text files. in this way the dictionary is slowly built up rather than created by transferring a traditional paper dictionary to disk.

Evaluation

At this stage in program development it is always worth evaluating the program so far. In use the program shows no obvious problems apart from its tendency to crash if the correct files are not present on disk. This clearly nidicates the need for the addition of some error handling. The internal structure of the program is quite good although PROCadd_dict is a little on the large size for a single procedure and might better have been broken down into two smaller procedures. It would also have been better if PROClookup had used separate procedures to open and read the dictionary file this would make any future modifications to the file

handling easier. However these criticisms are not so serious that any immediate action needs to be taken although you might want to disagree!

Perhaps the most obvious problem with this program is that it is slow. And it gets increasingly slow as words are added to the word list. There are two possible answers to this problem:

(1) Improve the search method.
(2) Use machine code.

It is quite obvious that a great saving in time could be made by replacing PROCfind_word with a procedure that uses a binary search (see *The Complete Programmer*). However, as the spelling checker is being used here as an example we will go straight to the machine code option as a way of illustrating how BASIC and machine code can be used together.

Locating inefficiency

The main thing to remember about using assembly language is that it is better not to use it at all! Given that the bulk of any program is only executed a few times it is possible to implement it using BASIC without any serious problems of inefficiency. However, any section of a program that is executed repeatedly can very easily become a problem. Even a section of code that only takes a fraction of a second to execute in BASIC may add up to a considerable delay if it is carried out thousands of times. Obviously the best thing to do is to identify any such 'critical' pieces of program and replace them by assembly language versions. In this way BASIC and assembler can be used to produce efficient programs with the minimum of effort.

In the case of the spelling checker it is obvious that searching the word list and adding new words to it are the two most likely candidates for assembly language routines. However, for the sake of example we will suppose that this is not quite so obvious and try to identify where the inefficiencies lie. To do this we need to know how much of the program's running time is taken by each procedure. The following three procedures can be used to record the cumulative time that any procedure is used for:

```
10000 DEF PROCstart
10010 TIME=0
10020 ENDPROC

10100 DEF PROCtime(I)
10110 T(I)=T(I)+TIME
10130 TIME=0
10140 ENDPROC

10200 DEF PROCtresult(N)
```

```
10210 LOCAL I
10220 FOR I=1 TO N
10230 PRINT I,T(I)
10240 NEXT I
10250 IF INKEY$(0)="" THEN GOTO 10250
10260 ENDPROC
```

PROCstart zeros the clock. PROCtime(I) adds the time since it was last called or from the last call to PROCstart to (I). PROCtresult(N) prints the cumulative times stored in T(I) to T(N). To use these procedures to find out how long it takes to read all the words in a file, search for them in the word list and finally add them to the list all we have to do is add the lines:

```
   1 DIM T(10)

  76 PROCstart

  85 PROCtime(1)

 105 PROCtresult(3)

4025 PROCtime(2)

4035 PROCstart

4045 PROCtime(3)
```

Lines 76 and 85 record the time spent in PROCread_word, line 4025 records the time spent in PROCfind word and lines 4035 and 4045 record the time spent in PROCmake_space (on common sense grounds these are likely to be the most time-consuming procedures).

Running the spelling checker, a 150 (approx.) word text gave the following results:

PROCread_word	18.07 seconds
PROCfind_word	28.29 seconds
PROCmake_space	43.36 seconds

Clearly PROCmake_space is the first procedure that should be replaced by assembly language!

Using machine code with BASIC

Using assembly language within a BBC BASIC program is easy but it is

still worth minimising the total amount of assembly language used. An examination of PROCmake_space suggests that the most time-consuming section is the FOR loop that actually does the move of the words in the list. This suggests that we need an assembly language routine that will move all of the characters in the list from FROM% to CURRENT_END% up by AMOUNT% bytes. It is clear that we are going to have to pass FROM%, CURRENT_END% and AMOUNT% to the routine as parameters and this implies the use of the CALL instruction. That is, the new version of PROCmake_space using an assembly language routine MOVE% to implement the move is:

```
6000 DEF PROCmake_space(FROM%,AMOUNT%)
6010 IF CURRENT_END%=FROM% THEN
     GOTO 6050
6020 CALL MVE%,FROM%,AMOUNT%,CURRENT_END%
6050 CURRENT_END%=CURRENT_END%+AMOUNT%
6060 IF ABS(FIN%-CURRENT_END%)<20 THEN
     PROCnospace
6070 ENDPROC
```

Notice that only the FOR loop (lines 6020 to 6040 in the old version) has been replaced and BASIC is still used to check for a word added to the end of the byte array (line 6010). updating CURRENT_END% (line 6050) and checking that there is enough space to add more words (line 6060). These operations would all have taken a great deal of assembly language to implement and the resulting gain in speed would be negligible.

All that now remains is to write the assembly language that constitutes MVE%!

The 6502 has a reputation for being a fast and efficient microprocessor but it lacks any registers that can hold a full 16-bit number or address. This is a problem for this particular application in that it involves the manipulation oi 16-bit addresses. The solution lies in the use of 'indirect' addressing. For example,

```
LDY #0
LDA (POlNT%),Y
```

will load the A register from the memory location whose address is stored in the pair of locations POINT% and POINT%+1. In this sense POINT% and POINT%+1 act like a 16-bit index register i.e. they hold the full 16-bit address of a memory location that is involved in an instruction. The only complication is that this sort of indirect addressing only works with memory locations in page zero, that is &00 to &FF. Page zero memory locations are so important and in such short supply that most of them are

already used by BBC BASIC and the MOS, but & 70 to &8F are set aside for use by application programs.

If POINT% is the address of the lowest pair of memory locations in page zero, then the basis of the main algorithm of the MVE% routine can be written:

```
LDY #0
LDA (POINT%),Y
LDY AMNT%
STA (POINT%),Y
```

The first two lines load A from the memory location pointed at by POINT% and the final two lines store A in the memory location AMNT% higher up in memory. This should be compared to

POINT%?AMNT%=?POINT%

which performs the same operation in BASIC.

Assembly language programming follows the same stepwise refinement principle used for BASIC and so the first stage of MVE% is:

```
8610 .MVE%    JSR PARM_GET%
8620         JSR START_LOOP%
8630 .MLOOP% LDY #0
8640         LDA (POINT%),Y
8650         LDY AMNT%
8660         STA (POINT%),Y
8665         JSR TST_END%
8666         BEQ MEXIT%
8670         DEC POINT%
8675         LDA POINT%
8676         CMP #&FF
8680         BNE MLOOP%
8690         DEC POINT%+1
8700         JMP MLOOP%
8710 .MEXIT% RTS
```

Subroutine PARM_GET% moves the parameters passed to MVE% from the parameter block at &600 into page zero so that they can be used as indirect addresses. Notice that the CALL instruction stores the addresses of the parameters, not their values in the parameter block. START_LOOP% initialises POINT% to the address stored in CURRENT_END% and stores the value in AMOUNT% in AMNT%. Lines 8630 form the assembly language equivalent of the FOR loop in the original version of the procedure. Subroutine TST_END% compares

POINT% to the address in FROM% and returns with the Z flag set accordingly. Finally lines 8670 to 8690 perform a 16-bit decrement of the value stored in POINT% notice that the 6502 has no 16-bit operations.

The next stage of refinement gives:

```
8740 .PARM_GET%       LDX #0
8750                   LDY #0
8760 .PLOOP%           LDA &601,Y
8770                   STA &70,X
8780                   INX
8790                   INY
8800                   LDA &601,Y
8810                   STA &70,X
8820                   INX
8830                   INY
8840                   INY
8850                   DEC &600
8860                   BNE PLOOP%
8870                   RTS
8880 \
8890 .START_LOOP%      LDY #0
8900                   LDA (CRNT_END%),Y
8910                   STA POINT%,Y
8915                   LDA (AMNT%),Y
8916                   STA AMNT%
8920                   INY
8930                   LDA (CRNT_END%),Y
8940                   STA POINT%,Y
8950                   RTS
8955 \
8960 .TST_END%         LDA POINT%
8965                   LDY #0
8967                   CMP (FRM%),Y
8969                   BNE TEXIT%
8970                   INY
8972                   LDA POINT%+1
8974                   CMP (FRM%),Y
8976 .TEXIT%           RTS
```

PARM_GET simply transfers the parameter addresses from the parameter block at &600 to zero page locations starting at &70. To understand this subroutine all you have to know is that &600 contains the number of parameters and each parameter is in the form of a two byte (16-bit) address and a single byte giving the type of the parameter. As all the parameters used in MVE% are integer variables, the type bytes are

ignored and at the end of PARM_GET%, &70 and &71 hold the address of FROM%, &72 and &73 hold the address of AMOUNT% and &74 and &75 hold the address of CURRENT_END%. That is after PARM_GET%.

```
LDY #0
LDA (&70), Y
```

would load A with the first (least significant) byte of FROM%. To avoid confusion the following labels are defined:

```
8545 FRM%=&70
8546 AMNT%=&72
8547 CRNT_END%=&74
```

START_LOOP% transfers the value stored in CURRENT_END% into POINT% and POINT%+1 and stores the first byte of AMOUNT% in AMNT%. Notice that following START_LOOP% AMNT% no longer contains the address of AMOUNT%, it contains the first byte of its value. (It is assumed that no word is longer than 256 characters!) Finally TST_END% is the assembly language equivalent of a predicate function. Instead of a truth value TST_END% returns with the Z condition code set according to whether the l6-bit value in POINT% is equal to the 16-bit value stored in FROM%. Notice that this is done in two stages; first the least significant bytes are compared (line 8967 and 8969) and then the most significant bytes are compared (lines 8972 and 8974). If either pair of bytes is different then the subroutine returns with the Z condition code clear. Whenever you have a complex condition to test for in assembler use a predicate subroutine like TST_END%.

After including all of the assembly language and the necessary BASIC to assemble it, the complete program is:

```
  1 DIM T(10)
 10 MODE 7
 15 DIM CODE% 200
 20 SIZE%=5000
 30 DIM WORD% SIZE%
 40 FIN%=WORD%+SIZE%
 50 FILE_END=FALSE
 60 PROCinitialise
 70 PROCopen_file
 75 REPEAT
 76 PROCstart
 80 PROCread_word
 85 PROCtime(1)
 90 IF WORD$<>"" THEN PROCadd_word
100 UNTIL FILE_END
```

```
 105 PROCtresult(3)
 110 PROCreport
 120 PROClookup
 130 PROClist_words
 140 PROCadd_dict
 150 END

1000 DEF PROCinitialise
1010 WORDCNT%=0
1020 CURRENT_END%=WORD%
1030 PROCasm(CODE%)
1040 ENDPROC

2000 DEF PROCopen_file
2010 CLS
2020 PRINT TAB(1,5);
2030 INPUT "File name of text to be"'
     " proof read ",F$
2040 IF LEN(F$)>10 THEN
     PRINT"Name too long":GOTO 2020
2050 PRINT
2060 F=OPENIN(F$)
2070 ENDPROC

3000 DEF PROCread_word
3010 LOCAL CHAR%
3020 WORD$=""
3030 REPEAT
3040 CHAR%=FNread_cap(F)
3050 UNTIL NOT(FNseparator(CHAR%))
     OR FILE_END
3060 WORD$=WORD$+CHR$(CHAR%)
3065 IF FILE_END THEN GOTO 3110
3070 CHAR%=FNread_cap(F)
3080 IF FNseparator(CHAR%) OR
     FILE_END THEN GOTO 3110
3090 WORD$=WORD$+CHR$(CHAR%)
3100 GOTO 3065
3110 IF WORD$<>"" THEN
     WORDCNT%=WORDCNT%+1
3120 PRINT TAB(0,8);
     "Number of words =";WORDCNT%
3130 ENDPROC

4000 DEF PROCadd_word
```

```
4020 PROCfind_word
4025 PROCtime(2)
4030 IF FOUND% THEN ENDPROC
4035 PROCstart
4040 PROCmake_space(WPOINT%,LEN(WORD$)+1)
4045 PROCtime(3)
4050 $WPOINT%=WORD$
4060 ENDPROC

5000 DEF PROCfind_word
5020 FOUND%=FALSE
5030 WPOINT%=WORD%
5040 IF WPOINT%=CURRENT_END% THEN ENDPROC
5050 IF $WPOINT%>WORD$ THEN ENDPROC
5060 IF $WPOINT%=WORD$ THEN FOUND%=TRUE
     :ENDPROC
5070 WPOINT%=WPOINT%+LEN($WPOINT%)+1
5080 GOTO 5040

6500 DEF PROClist_words
6505 CLS:VDU 14
6506 PROCprinter
6510 WPOINT%=WORD%
6520 IF WPOINT%=CURRENT_END% THEN
     VDU 15,3:ENDPROC
6530 PRINT $WPOINT%
6540 WPOINT%=WPOINT%+LEN($WPOINT%)+1
6550 GOTO 6520

6600 DEF PROCprinter
6605 LOCAL A%
6610 PRINT TAB(0,4);
6620 INPUT "Do you want to use
     the printer (Y/N)",A$
6630 IF A$<>"Y" AND A$<>"N" THEN
     GOTO 6620
6640 IF A$="Y" THEN VDU 2 ELSE VDU 3
6650 ENDPROC
6700 DEF PROCnospace
6710 PRINT TAB(0,10);"Not enough
     memory!!"
6720 PRINT "Closing file - but will
     process"
6730 PRINT "words already read"
6740 FILE_END=TRUE
```

```
6750 CLOSE#F
6760 TIME=0
6770 REPEAT:UNTIL TIME>500
6780 ENDPROC

7000 DEF PROCreport
7010 CLS
7020 PRINT TAB(0,5);"Text file has been
     read."
7030 PRINT "A total of -";WORDCNT%;
     " words"
7040 PRINT
7050 PRINT "Ready to check spelling
     against"
7060 PRINT "dictionary file."
7070 ENDPROC
7100 DEF PROClookup
7110 F=OPENUP("DICT")
7115 FILE_END=FALSE
7120 REPEAT
7130 INPUT#F,WORD$
7135 IF WORD$="" THEN FILE_END=TRUE
7140 PROCfind_word
7150 IF FOUND% THEN
     PROCremove(WPOINT%,LEN($WPOINT%)+1)
7160 UNTIL FILE_END
7180 PTR#F=PTR#F-2
7190 ENDPROC

7500 DEF PROCremove(FROM%,AMOUNT%)
7510 LOCAL I
7520 FOR I=FROM% TO CURRENT_END%-AMOUNT%
7530 ?I=I?AMOUNT%
7540 NEXT I
7550 CURRENT_END%=CURRENT_END%-AMOUNT%
7560 ENDPROC

8000 DEF PROCadd_dict
8005 LOCAL A$
8010 PRINT "Do you want to add correct"
8020 PRINT "words to dictionary (Y/N) ";
8030 INPUT A$
8040 IF A$<>"Y" AND A$<>"N" THEN
     GOTO 8010
8050 IF A$="N" THEN CLOSE#F:ENDPROC
```

```
8060 WPOINT%=WORD%
8070 IF WPOINT%=CURRENT_END% THEN
     PRINT#F,"":CLOSE#F:ENDPROC
8080 WORD$=$WPOINT%
8090 PRINT WORD$
8100 INPUT "A(dd) or I(gnore)",A$
8110 IF A$<>"A" AND A$<>"I" THEN
     GOTO 8100
8120 IF A$="A" THEN PRINT #F,WORD$
8130 WPOINT%=WPOINT%+LEN($WPOINT%)+1
8140 GOTO 8070

8500 DEF PROCasm(START%)
8510 LOCAL PASS
8520 FOR PASS=0 TO 3 STEP 3
8530 P%=START%
8540 REM CALL MVE%,FRM%,AMNT%,CRNT_END%
8545 FRM%=&70
8546 AMNT%=&72
8547 CRNT_END%=&74
8548 POINT%=&76
8600 [OPT PASS
8610 .MVE%          JSR PARM_GET%
8620                JSR START_LOOP%
8630 .MLOOP%        LDY #0
8640                LDA (POINT%),Y
8650                LDY AMNT%
8660                STA (POINT%),Y
8665                JSR TST_END%
8666                BEQ MEXIT%
8670                DEC POINT%
8675                LDA POINT%
8676                CMP #&FF
8680                BNE MLOOP%
8690                DEC POINT%+1
8700                JMP MLOOP%
8710 .MEXIT%        RTS
8720 \
8740 .PARM_GET%     LDX #0
8750                LDY #0
8760 .PLOOP%        LDA &601,Y
8770                STA &70,X
8780                INX
8790                INY
8800                LDA &601,Y
```

```
8810                    STA &70,X
8820                    INX
8830                    INY
8840                    INY
8850                    DEC &600
8860                    BNE PLOOP%
8870                    RTS
8880 \
8890 .START_LOOP%  LDY #0
8900                    LDA (CRNT_END%),Y
8910                    STA POINT%,Y
8915                    LDA (AMNT%),Y
8916                    STA AMNT%
8920                    INY
8930                    LDA (CRNT_END%),Y
8940                    STA POINT%,Y
8950                    RTS
8955 \
8960 .TST_END%     LDA POINT%
8965                    LDY #0
8967                    CMP (FRM%),Y
8969                    BNE TEXIT%
8970                    INY
8972                    LDA POINT%+1
8974                    CMP (FRM%),Y
8976 .TEXIT%       RTS
8990 ]
8995 NEXT PASS
8999 ENDPROC

9000 DEF FNseparator(C%)
9010 IF C%=ASC(" ") THEN =TRUE
9020 IF C%=ASC(",") THEN =TRUE
9030 IF C%=ASC(".") THEN =TRUE
9040 IF C%=ASC(":") THEN =TRUE
9050 IF C%=ASC(";") THEN =TRUE
9070 IF C%<32 THEN =TRUE
9080 IF C%>127 THEN =TRUE
9090 =FALSE

9100 DEF FNread_cap(F)
9110 LOCAL C%
9120 C%=BGET#F
9125 IF EOF#F THEN CLOSE#F:FILE_END=TRUE
9140 IF C%>96 THEN C%=C%-32
```

```
 9150  =C%

10000  DEF PROCstart
10010  TIME=0
10020  ENDPROC

10100  DEF PROCtime(I)
10110  T(I)=T(I)+TIME
10130  TIME=0
10140  ENDPROC

10200  DEF PROCtresult(N)
10210  LOCAL I
10220  FOR I=1 TO N
10230  PRINT I,T(I)
10240  NEXT I
10250  IF INKEY$(0)="" THEN GOTO 10250
10260  ENDPROC
```

When this version of the program is run on the same text file of 150 (approx.) words the timing results are:

PROCread_word	18.65
PROCfind word	28.55
PROCmake_space	2.49

which for PROCmake_space is an improvement of more than a factor of twenty! This excellent result should convince you to change PROCfmd_word to an assembly language routine.

Conclusion

The spelling checker has proved to be a very large program indeed and a completely finished version is well beyond the scope of this book. You should, however, be able to extend and modify it by the continued use of modular and structured programming. In doing so you should be able to prove for yourself how much easier it is to program using an organised method. The second point that this example illustrates is how BASIC and assembler can be used together to make programs efficient. There are very few occasions where a program has to be written in nothing but assembler and BASIC is always easier!

Chapter Nine
An Execution Tracer

The idea of testing a program by following the order of execution of statements was introduced in Chapter Seven. The use of the command TRACE ON was also described, as were the problems presented by the quantity of output this command can produce. Ideally what is needed is some way of watching the execution of a program line by line without cluttering up the screen. The object of the program developed in this chapter is to provide the BASIC programmer with a 'real time' indication of the order of execution. Roughly speaking it prints the line number of statement that is currently being obeyed at a fixed location at the top of the screen but, to allow the programmer time to read this number before it is overwritten, it also has to slow down the execution rate of the BASIC interpreter. This program is particularly attractive in that, as well as being a useful programming tool in its own right, it opens up the way to a very wide range of other utilities.

Intercepting BASIC

If a machine code program is going to print the line number of each line of BASIC as it is executed we obviously have to find some way of intercepting the BASIC interpreter as it either starts or finishes obeying a line. At first sight this seems like an impossible problem but the key to its solution lies in the way the BASIC interpreter communicates with the MOS. Whenever the interpreter needs to call the MOS to carry out some action it does so by calling one of the fixed addresses high in memory but these routines immediately 'indirect' through a number of 'jump vectors' stored in RAM. For example, if the interpreter wants to print a character on the screen it loads the A register with the character code and jumps to the subroutine OSWRCH at &FFEE. However, the routine at &FFEE immediately does an indirect jump using &20E (known as WRCHV) i.e. JMP (&20E). In other words, the routine at &FFEE transfers control to a routine whose address is stored in &20E and, as &20E is a location in RAM, it can be changed.

The principle behind intercepting the BASIC interpreter as it starts or finishes executing a line is to force it to make a jump to the MOS which could be diverted to a new machine code routine. (Notice that there is no way that this routine could be written in BASIC!) After examining the possible jump vectors that could be altered (given in section 43 of the User Guide) the first idea to present itself was to force an error at the start of each ine by including an illegal character. This would cause the BASIC interpreter to jump through BRKV at &202 but the problem with this method is that there is no obvious way of restarting the BASIC interpreter after an error of this sort. None of the other jump vectors seemed to be of my use and so it looked as though a completely different method would have to be employed. However, after leaving the problem for a few hours, inspiration struck! Suddenly the possibility of using the existing TRACE command became clear. Following a TRACE ON command the line number of each line that is executed is printed on the screen in a fixed format:

[line number] space

The printing is, of course, achieved by using OSWRCH and this routine can be intercepted quite easily. Thus the overall plan is to change the RAM vector used by OSWRCH to point to a new machine code routine that checks each character printed for '['. If it finds an opening square bracket then it can assume that what follows, up to the matching closing square bracket is a line number. The only problem with this method is that any programs that print square brackets as part of their normal output or that contain assembly language will behave strangely. This seems a small price to pay for such a useful utility so easily implemented!

To summarise the execution tracer should:

(1) Detect '[' and move the text cursor to a fixed location where the digits that follow '[' will be printed.

(2) Detect ']' and move the text cursor back to its original position and suppress the next blank that is printed.

Assembling a program

If you are writing a lot of assembly language then it makes good sense to write a standard BASIC program that can be re-used with each assembly language program. For example:

```
10 DIM CODE% 500
20 PROCasm(CODE%)
30 CALL *****
40 STOP
```

```
1000 DEF PROCasm(START%)
1005 FOR PASS=0 TO 3 STEP 3
1010 P%=START%
1020 PROCprog
1030 NEXT PASS
1040 PRINT
1050 PRINT P%-START%;" Bytes"
1060 ENDROC
```

Line 10 sets up a byte array called CODE% to hold the machine code produced. (For large programs 500 bytes might prove insufficient.) Line 20 calls PROCasm which actually does the job of assembling the program. The parameter START% is used to indicate where assembly should begin. This is usually set to CODE%, but as explained later this is not always the case. Line 30 calls the assembled machine code for testing. Of course, ***** would be replaced by the name of the routine to be run. Lines 1005 to 1030 perform the two pass assembly as described in Chapter Four. Line 1050 prints the number of bytes that the assembled machine code occupies and it is worth keeping an eye on this figure to make sure that sufficient space has been allocated to the byte array CODE%. The actual assembly language to he assembled is written within PROCprog and this is the only procedure that ever needs to be significantly altered. If you want to write an assembly language program simply write it as part of PROCprog (enclosed in square brackets) and add it to the BASIC program given above.

The first stage

Assembly language program development proceeds in exactly the same way as for a BASIC program that is, as a number of stages of refinement. There are two parts to the main program of the execution tracer, the part that 'installs' it by changing the jump vector and the part that checks for square brackets. There is an important difference between the two in that the part that installs the program is only run once to change the jump vector but the check for square brackets is executed each tune a character is printed out - in this sense the part that changes the jump vector isn't really an integrated part of the execution checker and it can be written later.
    The main program has to do three things:

(1) Check for '[' and carry out the appropriate action.
(2) Check for ']' and carry out the appropriate action.
(3) Check for the 'trailing' space after ']' and suppress it.

The first two actions are obvious but you might wonder why the trailing

space has to be suppressed. The reason for this is a matter of screen formatting and will become clear later. The main program can now be written quite easily:

```
3100  .MAIN%       CMP #ASC("[")
3110               BEQ DOTRACE%
3120               CMP #ASC("]")
3130               BEQ FINTRACE%
3140               CMP #ASC(" ")
3150               BEQ SPACE%
3160               JMP (&70)
```

DOTRACE%, has to move the current printing position to the top of the screen so that the digits that follow the '[' are always printed in the same place. FINTRACE% has to restore the printing position back to where it was before DOTRACE% moved it! This implies that DOTRACE% also has to store the existing printing position before moving it elsewhere. SPACE% has the rather odd job of checking to see if each space printed follows a closing square bracket, but more of this later. If the character to be printed is anything other than a square bracket or a blank the main program does an indirect jump through location &70. The reason for this is that it is assumed that the part of the program that changes the jump vector WRCHV to point to the start of the execution tracer will store its original value in &70 and &71. Thus JMP (&70) will transfer control to the machine code in the MOS that implements OSWRCH.

Second stage

The machine code routines that have to be written as part of the second stage of refinement are DOTRACE%, FINTRACE% and SPACE%. As described in the last section, DOTRACE%, has to save the current position of the text cursor and then move it to the position that the line number is to be printed at. Finding the current position of the text cursor is easy OSBYTE call 134 returns the x and y co-ordinates of the text cursor in the X and Y registers respectively. In practice, this is a little too easy in that there is a hidden trap inherent in using any of the MOS from within the execution tracer. The trouble is that whenever DOTRACE% is called, the BASIC interpreter will be in the middle of printing a '[' on the screen and if it is going to resume what it was doing before it was intercepted by the execution tracer the contents of all the registers and system variables have to be preserved. The easiest way to save and restore all or any of the registers is to use the system stack. However, pushing all the registers on the stack and then pulling them all off is a very long and boring list of assembly language statements. The obvious thing to do is to use a pair of macros:

```
9300 DEF PROCsave
9400 [OPT PASS
9410      PHA
9420      PHP
9430      TXA
9440      PHA
9450      TYA
9460      PHA
9470 ]
9480 ENDPROC

9500 DEF PROCrestore
9510 [OPT PASS
9520      PLA
9530      TAY
9540      PLA
9550      TAX
9560      PLP
9570      PLA
9580 ]
```

The instruction:

```
]:PROCsave:[OPT PASS
```

will generate the code to push each of the 6502% registers onto the system stack in the order A,P ,X, Y. Similarly:

```
]:PROCrestore:[OPT PASS
```

will pull the registers off the system stack in the reverse order i.e. Y,X,P,A thus restoring their original values. (The system stack is a LIFO stack, see Chapter Five.)

Now that the problem with preserving the registers has been solved DOTRACE% is easy to write:

```
3200 .DOTRACE%   :]PROCsave:[OPT PASS
3210             LDA #134
3220             JSR OSBYTE%
3230             STX &72
3240             STY &73
3250             LDA #31
3260             JSR OSWRCH%
3270             LDA #20
3280             JSR OSWRCH%
3290             LDA #0
```

```
3300                    JSR OSWRCH%
3310                    :]PROCrestore:[OPT PASS
3320                    LDA #0
3330                    JMP (&70)
```

Lines 3210 to 3240 find the current location of the text cursor (using OSBYTE call 134) and store the X value in &72 and the Y value in &73. Lines 3250 to 3300 use the machine code equivalent of TAB(x,y) to move the text cursor. This is achieved by sending ASCII code 31 to the VDU driver (see Chapter Ten) followed by the desired x and y values (20 and 0 in this case). The last part of the routine restores the registers (line 3310) and then loads the A register with 0 the ASCII code for null and does an indirect jump through location &70 (line 3330). As already explained, this transfers control to machine code that implements OSWRCH in the MOS and thus completes the call to OSWRCH made by BASIC.

Notice that this simple routine, DOTRACE%, has a very subtle and potentially confusing action. It calls OSWRCH apparently directly using JSR OSWRCH%, but WRCHV (the jump vector for OSWRCH) has been changed to point to the start of the execution tracer. In other words, the instructions JSR OSWRCH% actually calls the execution tracer for a second time! This works because none of the routines within the execution tracer try to print any of the characters tested for by the main program and so all OSWRCH calls that originate from within the execution tracer are correctly passed on to OSWRCH (i.e. by line 3160). You might like to work out what happens if DOTRACE% attempts to print "[" or "]"!

FINTRACE% is quite easy:

```
3400 .FINTRACE% :]PROCsave:[OPT PASS
3405                JSR PBLANK%
3410                LDA #31
3420                JSR OSWRCH%
3430                LDA &72
3440                JSR OSWRCH%
3450                LDA &73
3460                JSR OSWRCH%
3470                LDA #&FF
3480                STA SPFLG%
3485                JSR DELAY%
3490                :]PROCrestore:[OPT PASS
3500                LDA #0
3510                JMP (&70)
```

After first saving the registers using PROCsave it prints two blanks using PBLANK% (a subroutine to be written at the next level of refinement).

The reason for printing these blanks is to clear any trailing digits belonging to previous line numbers. Lines 3410 to 3460 restore the text cursor to its original position once again using the machine code equivalent of TAB(x,y). Line 3480 sets SPFLG% to &FF to indicate that the next space Ibm BASIC tries to print should be suppressed. Line 3485 then calls DELAY%. which, as its name suggests, serves to slow down the speed of execution of BASIC so that the fine numbers can be read. Finally a null is printed to complete the OSWRCH call originally made by BASIC.

Now that FINTRACE% has set SPFLG% to &FF when a space has to be suppressed, SPACE%, can be written as a simple test on SPFLG%.

```
3600 .SPACE%       PHA
3610               LDA SPFLG%
3620               BEQ SPEXIT%
3630               PLA
3640               LDA #0
3650               STA SPFLG%
3660               JMP (&70)
3670 .SPEXIT%      PLA
3680               JMP (&70)
```

After saving the contents of A (line 3600) SPFLG% is loaded and tested to see if it is zero. If it is, line 3260 transfers control to 3670 where the value in A is restored and printed (line 3680). If SPFLG% isn't zero then A is loaded with zero and this is used both to zero SPFLG%, and to print a null character in place of the space. Notice the way that FINTRACE%, and SPACE% use SPFLG% as a method of communication. FINTRACE% sets SPFLG% to &FF when the next space should be suppressed and SPACE% resets it to zero so that only a single space is supppressed. A variable or memory location used in this way is often referred to as a 'flag'.

Third stage and changing WRCHV

At the third stage only PBLANK% and DELAY% remain to be implemented. These are both trivial:

```
3700 .DELAY%       LDY #PAUSE
3710 .DLOOP%       LDX #255
3720 .D1%          DEX
3730               BNE D1%
3740               DEY
3750               BNE DLOOP%
```

```
3760                    RTS

3800  .PBLANK%          LDA #ASC(" ")
3810                    JSR OSWRCH%
3820                    LDA #ASC(" ")
3830                    JSR OSWRCH%
3840                    RTS
```

The only detail worth comment is the way that the X and Y registers are used as separate counters to provide a long enough delay. Lines 3710 to 3730 form an inner loop nested within an outer loop formed by lines 3700 to 3750. Each time through the outer loop the inner loop is executed 255 times. The total delay is set by the variable PAUSE% which for convenience can be defined as part of the BASIC program that assembles the execution tracer.

All that now remains to be implemented is a short program to change the jump vector WRCHV to point to the start of the execution tracer (i.e. MAIN%). This simply involves transferring the existing contents of &20E to &70 and &20F to &71 and then storing the low and high bytes of MAIN% in &20E and &20F respectively. That is:

```
3000  .SETUP%           LDA &20E
3010                    STA &70
3020                    LDA &20F
3030                    STA &71
3040                    LDA #(MAIN% MOD 256)
3050                    STA &20E
3060                    LDA #(MAIN% DIV 256)
3070                    STA &20F
3080                    RTS
```

The complete program

The complete program, including macros, data definitions and the procedures that assemble it, is given below:

```
  10 DIM CODE% 500
  15 PAUSE=255
  20 PROCasm(CODE%)
  30 CALL SETUP%
  40 STOP

1000 DEF PROCasm(START%)
```

```
1005 FOR PASS=0 TO 3 STEP 3
1010 P%=START%
1020 PROCprog
1030 NEXT PASS
1040 PRINT
1050 PRINT P%-START%;" Bytes"
1060 ENDPROC

2000 DEF PROCprog
2010 OSBYTE%=&FFF4
2020 OSWRCH%=&FFEE
2030 OSWORD%=&FFF1
2990 [OPT PASS

3000 .SETUP%    LDA &20E
        \save WRCHV in &70
3010           STA &70
3020           LDA &20F
3030           STA &71
3040           LDA #(MAIN% MOD 256)
        \store MAIN % in WRCHV
3050           STA &20E
3060           LDA #(MAIN% DIV 256)
3070           STA &20F
3080           RTS
3090 \
3100 .MAIN%     CMP #ASC("[")
3110           BEQ DOTRACE%
3120           CMP #ASC("]")
3130           BEQ FINTRACE%
3140           CMP #ASC(" ")
3150           BEQ SPACE%
3160           JMP (&70)
        \print the character in A
3170 \
3200 .DOTRACE% :]PROCsave:[OPT PASS
        \save the registers
3210           LDA #134
        \get the current cursor position
3220           JSR OSBYTE%
3230           STX &72
        \and store it in &72 and &73
3240           STY &73
3250           LDA #31
        \TAB(20,0)
```

```
3260               JSR OSWRCH%
3270               LDA #20
3280               JSR OSWRCH%
3290               LDA #0
3300               JSR OSWRCH%
3310 :]PROCrestore:[OPT PASS
        \restore registers
3320               LDA #0
3330    JMP (&70)
        \print a null
3340 \
3400 .FINTRACE% :]PROCsave:[OPT PASS
        \save registers
3405               JSR PBLANK%
        \print two blanks
3410               LDA #31
        \TAB(X,Y) where x and y are
3420               JSR OSWRCH%
        \old cursor position
3430               LDA &72
3440               JSR OSWRCH%
3450               LDA &73
3460               JSR OSWRCH%
3470               LDA #&FF
        \set flag to supress next space
3480               STA SPFLG%
3485               JSR DELAY%
        \make BASIC slower
3490 :]PROCrestore:[OPT PASS
        \restore registers
3500               LDA #0
        \print a null
3510    JMP (&70)
3520 \
3600 .SPACE%     PHA
3610               LDA SPFLG%
        \test flag
3620    BEQ SPEXIT%
  \IF SPFLG=0 THEN PRINT A ELSE PRINT ""
3630               PLA
3640               LDA #0
3650               STA SPFLG%
3660               JMP (&70)
3670 .SPEXIT%     PLA
3680               JMP (&70)
```

```
3690 \
3700 .DELAY%     LDY #PAUSE
         \PAUSE sets the length of delay
3710 .DLOOP%    LDX #255
3720 .D1%       DEX
3730            BNE D1%
3740            DEY
3750            BNE DLOOP%
3760            RTS
3770 \
3800 .PBLANK%   LDA #ASC(" ")
3810            JSR OSWRCH%
3820            LDA #ASC(" ")
3830            JSR OSWRCH%
3840            RTS

9000 REM MACROS

9010 DEF FNequb(VA%)
9020 ?P%=(VA% MOD 256)
9025 IF PASS=3 THEN PRINT ~P%;"=";~?P%
9030 P%=P%+1
9040 =P%-1

9100 DEF FNequw(VA%)
9110 ?P%=(VA% MOD 256)
9115 IF PASS=3 THEN PRINT ~P%;"=";~?P%
9120 P%?1=(VA% DIV 256)
9125 IF PASS=3 THEN PRINT ~P%+1;"=";~(P%?1)
9130 P%=P%+2
9140 =P%-2

9200 DEF FNequs(S$)
9210 $P%=S$
9220 IF PASS=3 THEN PRINT ~P%;"=";S$
9230 P%=P%+LEN(S$)+1
9240 =P%-LEN(S$)-1

9300 DEF PROCsave
9400 [OPT PASS
9410    PHA
9420    PHP
9430    TXA
9440    PHA
9450    TYA
```

```
9460     PHA
9470  ]
9480  ENDPROC

9500  DEF PROCrestore
9510  [OPT PASS
9520     PLA
9530     TAY
9540     PLA
9550     TAX
9560     PLP
9570     PLA
9580  ]
9590  ENDPROC
```

If you run this program and then add to it the lines:

```
1 REM
2 GOTO 1
```

and then run it after typing TRACE ON you will see the line numbers '1' and '2' printed repeatedly at the top of the screen following the execution of the infinite loop. To disable the trace just type TRACE OFF. Notice that if you try to trace the execution of the program that assembles the execution trace program you are doomed to crash the machine. The reason for this is that the first pass of assembly overwrites the existing machine code with an incomplete version with obvious consequences.

Using the execution tracer

If the execution tracer is going to be used to test BASIC programs it is clear that the machine code that it produces has to be stored somewhere out of harm's way so that another BASIC program can be loaded and run without destroying the machine code. There are locations where machine code can hide which do not need to be 'setup' and thus the user can be spared the problem of reserving memory. The trouble is that these readymade places are all used for something else by the BBC Micro. For example, the execution tracer could be stored in the RS423 buffers (transmit at &0900 to &09FF and receive at &0A00 to &0AFF) but this would cause a problem if the serial port was being used at the same time.

The alternative to using one of these existing areas is to reserve some memory by increasing the value stored in PAGE. However, for a utility such as the execution tracer this seems very inappropriate and likely to cause trouble if you load the machine code over an existing BASIC program by forgetting to set PAGE! After weighing up the alternatives, in

this case it seems better to use the RS423 receive buffer at &0A00 to store the machine code. This still allows a serial printer if required without disturbing the machine code Of course, if the program that was being debugged using the tracer used the RS423 Input then it would have to be re-assembled In tun in the memory reserved using PAGE. (An example of the use of PAGE to reserve memory can be found in Chapter Ten in connection with the background clock program.)

To assemble the machine code into the RS432 receive buffer all you have to do is to change line 20 to read:

20 PROCasm(&A00)

The resulting machine code should then be saved on disk or tape by using:

*SAVE TRACER A00,AFF,A00

This will save the entire RS432 buffer from &A00 to &AFF. Following this you can install the execution tracer at any time by typing:

*RUN TRACER

and using TRACE ON/OFF to control it.

Evaluation and modification

The execution tracer is not an easy program to write in that it 'tampers' with the normal working of the MOS. The program is already in a state where it is a useful tool but there are one or two simple extras that would be worth incorporating. It is very easy to install the execution tracer but currently the only way to remove it completely is to press BREAK. It would be better if another program was made available to remove the execution tracer by changing WRCHV back to its original value. It would also bean advantage to incorporate a method of allowing the user to alter the delay factor without having to re-assemble the program, but this is a more complicated addition.

The main worry with any program that modifies the workings of the MOS is that it will introduce unwanted and unpredictable side effects. Such side effects are always difficult to pin down because they are generally only observed after a great number of different things have happened and in this sense they are not repeatable. The execution tracer does have a serious problem that was first identified as a side effect-Occasionally while tracing the progress of a graphics program a line would be drawn to the wrong point. This happened very infrequently but often enough to cast doubt on the execution tracer. The cause of this erratic behaviour remained a mystery for some time until a simple program that consistently produced the misbehaviour was found. If you try:

```
10 VDU 23,224,0,91,0,91,0,91,0,91
20 PRINT "A"
30 GOTO 10
```

with the execution tracer installed and turned on you will find that the "A" is not pruned although the trace claims that line 20 is repeatedly executed. The explanation lies in the fact that ASC("[") is 91! A number of VDU codes are followed by a list of parameter bytes, all of which are sent to the VDU driver. For example, in the case of line 10 the VDU statement sends ASCII code 23, then 224, and then 0 and 91 four times. The execution tracer is thus fooled into thinking that a stream of opening square brackets are being sent to the screen when in fact they are all part of a character definition!

The solution to this problem is to detect those VDU codes that are followed by a sequence of parameter bytes and make sure that the execution tracer ignores them. This is quite easy in principle but in practice it takes quite a few extra lines of code. The modifications necessary to the execution tracer to enable it to ignore parameter bytes following certain VDU codes are given below:

```
3090 .MAIN%            PHA
3091                   LDA SUP_COUNT%
      \IF SUP_COUNT<>0 THEN GOTO SKIP%
3092                   BNE SKIP%
3093                   PLA
3094                   JSR VCODE%
      \check for codes and set SUP_COUNT%
3109                   CMP #ASC("[")
3110                   BEQ DOTRACE%
3120                   CMP #ASC("]")
3130                   BEQ FINTRACE%
3140                   CMP #ASC(" ")
3150                   BEQ SPACE%
3160                   JMP (&70)
3165 .SKIP%            DEC SUP_COUNT%
      \SUP_COUNT=SUP_COUNT-1
3166                   PLA
      \PRINT A
3167                   JMP (&70)

4000 .VCODE%           STX &74
      \save X
4010                   LDX #9
      \load X with number of bytes that
       follow each code
4020                   CMP #23
```

```
          \test for code
 4030                   BEQ COEX%
          \IF A=code THEN GOTO COEX%
 4040                   LDX #8
 4050                   CMP #24
 4060                   BEQ COEX%
 4070                   LDX #5
 4080                   CMP #25
 4090                   BEQ COEX%
 4100                   LDX #4
 4110                   CMP #28
 4120                   BEQ COEX%
 4130                   LDX #4
 4140                   CMP #29
 4150                   BEQ COEX%
 4155                   LDX #2
 4156                   CMP #31
 4157                   BEQ COEX%
 4160                   LDX #0
 4170 .COEX%            STX SUP_COUNT%
 4180                   LDX &74
       \restore X
 4190                   RTS
 5000 ]
 5010 REM DATA
 5020 SPFLG%=FNequb(0)
 5030 SUP_COUNT%=FNequb(0)
```

The basic idea behind this extension is to record in SUP_COUNT%, the number of bytes that should be ignored by the execution tracer. If SUP_COUNT% is zero then the program works as before apart from calling VCODE% to check to see if the code in the A register is followed by any parameter bytes. If SUP_COUNT% is non-zero then the code in the A register is not examined by the execution tracer and is simply passed to OSWRCH. Each code that is ignored in this way causes SUP_COUNT% to decrease by one. Thus the number of bytes that are ignored depends on the initial setting of SUP_COUNT%. Setting SUP_COUNT% is the responsibility of subroutine VCODE%. This repeatedly loads the X register with the number of bytes to be ignored following a particular code and then tests for that code. For example, line 4010 loads X with 9 and then line 4020 tests for code 23 and indeed nine parameter bytes always follow code 23. In this way the X register always contains the number of bytes to be ignored when the subroutine reaches COEX%. This value is stored in SUP_COUNT% and then control is returned to the main program. There are many other small but important

details of how this extension works but you should be able to make sense of them one by one.

Conclusion

The execution tracer started life as a fairly simple idea for an assembly language program. As with many assembly language programs it quickly grew to be larger and more subtle than expected! The use of stepwise refinement and modular programming makes it much easier to cope with the unexpected as well as the planned! There is still a small problem with the execution tracer that results from the way that BBC BASIC handles the comma within a PRINT statement. This is because the 'zone' spacing is performed by BASIC, which uses the COUNT function to send the correct number of blanks to the screen, rather than by the MOS. There is nothing simple that can be done to correct this problem without using memory locations that might be changed in future versions of BASIC. Fortunately the trouble it causes doesn't detract loo much from the usefulness of the execution tracer.

Chapter Ten
The MOS – A Soft Machine

Of the two pieces of system software stored in ROM (BBC BASIC and the MOS Machine Operating System) BASIC gets by far the greater amount of attention and praise. This is only to be expected because BBC BASIC is a fairly advanced dialect of a very well known language. However, BBC BASIC relies very heavily on the software support of the MOS and many of the actions that appear to be part of BASIC are simply passed over as a request for action to the MOS. The MOS is also important in that it is the only piece of 'fixed' ROM software in the BBC Micro. For example, you might replace the BASIC ROM by another language but the new language would still use the MOS as support software.

Some of the details of the workings of the MOS have already been described in *The BBC Micro: An Expert Guide* and the aim of this chapter is to enlarge on some of the ideas presented in this earlier book. The MOS is a highly structured piece of software and there is much that can be learned from it about the way that the BBC Micro works. The first part of this chapter builds up a general picture of the different sections of the MOS and the tasks that they perform. The final part presents a short project that makes use of a very special feature of the BBC Micro, 'event handling', to produce a clock display that allows you to run another program while still displaying the time.

What is a machine operating system?

The reason why there is, in general, so little comment on the MOS probably has something to do with it being an operating system. The purpose of implementation of a language such as BASIC is obvious, but the purpose of an operating system is not so clear. The quality of a dialect of BASIC is therefore easy to judge while it is difficult to set a standard for an operating system. Tradition has it that the purpose of an operating system is to make the facilities of a machine available to other programs and ultimately the user, and this is indeed what the MOS does. However. if an operating system offers up a facility in such a way that it is difficult

to use, does the fault lie in the operating system or the hardware of the machine? You might think that the answer to this question depends on the particular case; sometimes the software would beat fault and sometimes the hardware. The surprising thing is that it is always the software! No matter how horrible the hardware is it is possible for the software contained within the operating system to deliver it to the user in a way that is easy to use. The only thing that you can blame the hardware for is lack of speed and this might make the software's job pointless - an easy-to-use but incredibly slow system is not something that anyone wants! An ideal operating system should, so to speak, 'tame the machine's hardware' so that other software can concentrate on putting it to good use.

Operating systems are not often considered an important factor in personal computing and machine designers have often overlooked the need for good operating systems and concentrated on the quality of the BASIC. This has forced BASIC to grow extensions that look after whatever extra hardware that the machine has. An example of this approach at its most extreme can be seen in the APPLE II's so-called Disk Operating System or DOS. This is so much an afterthought to APPLESOFT BASIC that it is difficult to see it as a separate piece of software! The BBC Micro was possibly the first machine fully to use an operating system both to make facilities available and to enhance the performance of the hardware. The MOS was designed as an essential and integral part ofthe BBC Micro and without it the hardware would look distinctly incomplete! Indeed the MOS is best regarded as a software extension of the hardware and in this sense the BBC Micro is very much a creation of software. In other words, it can be viewed as a 'soft machine'.

The structure of the MOS

The main design decision that seems to have given rise to the MOS is that all of the machine's I/O would be handled by it and it alone. In this sense the MOS can be thought of as a layer of software that separates other system software, such as BASIC, from the hardware (see Fig. 10.1). This separation makes it possible to enable languages such as BASIC to be hardware independent. At first it may be difficult to see any payoffs for the user of this hardware independence but it is, for example, responsible for the uniform way that files are handled no matter what the file device concerned tape, disk, network or telesoftware (see Chapter Six). As far as file devices are concerned, there is just one set of I/O commands and the variability in the storage device is 'absorbed' by the filing system software.

Miscellaneous and character-oriented devices, such as the sound generator and the serial interface, are always difficult to build into an operating system in a regular way but the BBC MOS does its best. The text and graphics display is such an important part of the machine that it is given a section of the MOS all to itself the VDU drivers. Other I/O) devices are combined with miscellaneous operations and dealt with in two

classes those that require only a small amount of information to be passed and those that require a large or variable amount of information to be passed. The reason for this division is entirely practical because it enables a simple method of communicating with the MOS to be used wherever possible.
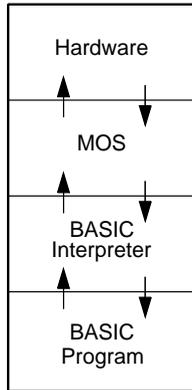

Fig. 10.1. The flow of data.

The MOS also acts as a sort of 'supervisor' program for any other ROMs that might be present in the sideways ROM sockets. For example, when the machine is first switched on the MOS selects, or 'pages', the ROM that is in the numbered ROM socket with the highest number (see later). A more important example of the role of the MOS in ROM paging is in the processing of 'service' requests. When the ROM currently in use makes a call to the operating system (via OSBYTE, for example) to perform some I/O Oor any other task the MOS will attempt to carry out the request. However, if the M OS doesn't recognise the request as something it is capable of dealing with, it will ask each of the sideways ROMs in turn if they can deal with the request. Thus the MOS is responsible for implementing a fairly sophisticated paging system that can effectively extend the BBC Micro's memory space to well over the 64K of memory that the 6502 can address directly.

The final, and perhaps the least obvious, action of the MOS is to look after interrupts. Put simply, an interrupt is a signal that stops a computer from carrying out its current task, switches its attention to another task and then returns it to its current task as if the interruption had never occurred. Thus interrupts can be used to give a computer the appearance of doing more than one thing at a time by repeatedly and very rapidly switching between a number of tasks. The BBC Micro is an advanced machine in that interrupts are an essential part of the way it works not, as in the case of so many machines, an afterthought added to control a special I/O device or provide a clock. An interrupt-driven machine has a very special 'feel' about it for both programmers and users. In particular, programmers have to be aware ofthe potential of a machine that runs

under interrupts and of the consequent problems. Apart from providing the real time clock in the pseudo variable TIME, the interrupts are used to service and maintain the extensive system of I/O buffers and queues and provide information to user programs about when certain events have occurred. To get the best from your BBC Micro it is very important to understand the nature of an 'interrupt driven' machine. To this end the final part of this chapter gives an example of how interrupts can be used within a program.

Interrupt
+ event
handler

File system
OSFIND
OSGBPB
OSBPUT
OSBGET
OSARGS
OSFILE

Misc I/O
OSRDCH (OSWRCH)
OSBYTE
OSWORD

VDU/
GRAPHIC DRIVER
OSASCI
OSNEWL
OSWRCH

ROM
PAGING
System
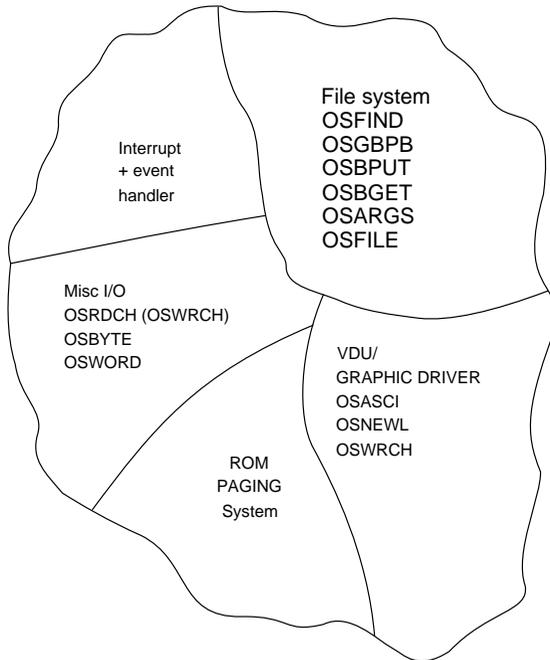
Fig. 10.2. The structure of the MOS.

The structure of the MOS as described above can be seen in Fig. 10.2, along with the names of the machine code subroutines that BASIC and other software use to communicate with each section. Notice that although the interrupt and event handler looks as though it has no way of letting either BASIC or a user program gain access to it, this is not the case, as will be explained later. As with any piece of real software there are always odds and ends that don't fit into the overall classification and the MOS is no exception. For example, there is a 'command line interpreter' which while not essential does make the MOS easier to use. Now that the structure of the MOS is clear it is time to look more closely at each of its parts.

The file system

In most computer systems it is useful to distinguish two types of I/O devices those that work in terms of named files and those that work by transferring small units of data one after the other. File devices include the well known cassette recorder and floppy disks of all kinds (Chapter Six has already described file handling as it applies to these two types of device). However, for the BBC Micro the range of file devices extends to exotic systems such as networks, telesoftware and ROM cartridges. Such diverse systems might lead you to believe that a different method of handling is appropriate and indeed necessary for each one. This is not so! As already mentioned at the start of Chapter Seven, the idea of a file and the operations that manipulate it are independent of the actual physical device used to store it. The physical device affects the speed of storage and retrieval (and very occasionally it may not allow a particular operation) but it doesn't alter the principle underlying handling files.

The MOS uses a standard set of six machine code subroutines for all file operations, no matter what type of device is in use. Two of the subroutines, OSFIND and OSARGS, are concerned with general file operations such as 'opening', 'closing', 'deleting', etc. The remaining four are used to read and write files. OSBGET and OSBPUT are used to read and write a single byte from and to a file that is already open. OSFILE reads or writes an entire file in one operation. The final subroutine OSGBPB will write or read a variable sized block of bytes to the file this can only be used with devices that can sensibly support random access such as the disk system.

Each one of these subroutines is used by referring to a fixed location in the MOS ROM but, as these locations immediately transfer control to jump vectors in RAM, the position of the code that actually carries out the operation can easily be changed. In this way the code used when you call one of the subroutines can be made appropriate for the currently selected filing system tape, disk, network, etc. These subroutines are used by BASIC to implement its file I/O and you should be able to recognise the correspondence between the BASIC commands GET, PUT and LOAD/ SAVE with the MOS subroutines OSBGET, OSBPUT and OSFILE. Of course the more familiar INPUT and PRINT make repeated use of the MOS subroutines to handle their multiple data transfers. The use of these machine code subroutines has already been described with reference to tape and disk in Chapter Six.

The VDU driver

The VDU driver is responsible for performing all of the text and graphics operations provided by the BBC Micro. If you are familiar with the hardware details of how the BBC Micro produces its TV display (see The BBC Micro: An Expert Guide for this hardware information) you will

realise that the VDU driver has a lot to do simply to make a character appear on the screen. As well as this fundamental job, it also has to provide the colour control, implement the graphics commands, look after the scrolling, implement and maintain the text and graphics windows and look after just about every other feature of the display. With so many different things to do you might expect that the VDU driver would consist of a great many different subroutines, each with a different name and method of use. As in the case of the file system, it is possible to make things simpler by careful planning and observing how information is normally generated and sent to a video display. As text is usually sent to the display in the form of a stream of ASCII codes it makes sense to continue to use the same method to communicate all the data to the VDU driver. In other words, the VDU driver can use a single entry point as long as it examines the stream of ASCII codes that is being sent to it for special 'control codes'.

The single entry point is OSWRCH and the ASCII codes are extended to include all of the text and graphics operations. For example, if the VDU driver detects ASCII code 12 then it will clear thetext screen, a code of 1 6 will clear the graphics screen and so on. Sometimes the amount of information needed for a text or graphics operation is greater than a single ASCII code can convey. In this case a sequence of ASCII codes is required. For example, following ASCII code 31 the next two codes are not interpreted in the usual way but are taken to be the new X and Y position of the text cursor.

The familiar BBC BASIC text and graphics commands are implemented by sending the appropriate ASCII codes to the VDU driver. For example, the command CLS results in the code 12 being sent to the VDU driver. In this sense the BASIC statements:

    CLS
    PRINT CHR$(12);

and

    VDU 12

all achieve exactly the same results - they send code 12 to the VDU driver and so clear the screen and are therefore the same! TAB(X,Y) first sends code 31 and then two codes whose values are given by X and Y respectively. In other words, BBC BASIC doesn't manipulate directly the contents of the RAM that stores the screen data. Instead it converts its requirements into the appropriate sequence of ASCII codes and sends them to the VDU driver which then changes the screen data. All programs running on the BBC Micro, whether in BBC BASIC or machine code,should use the VDU driver and avoid direct manipulation of the screen memory if at all possible.

For the sake of convenience, it is useful to have three entry points to the VDU driver; OSWRCH, which sends the code stored in the A register

to the VDU driver; OSASCI, which inspects the code and, if it is a carriage return, sends a new fine code to the VDU driver, and finally. OSNEWL, which is an entry point that automatically sends a new line followed by a carriage return code. Each of these subroutines is entered via a fixed address within ROM which then immediately transfers control to a routine whose address is stored in RAM. This makes it possible to intercept all output going to the screen by changing the contents of the RAM vector and to replace or alter OSWRCH, etc. An example of this can be found in the trace program given in Chapter Nine.

The power of this simple system of using a stream of ASCII codes to control the complex text and graphics operations takes some time to appreciate. One of its advantages is that both the BASIC programmer and the assembly language programmer control the display in exactly the same way. The sequence of codes that the BASIC programmer sends to the VDU driver, using, say, the VDU command, is exactly the same as the set of codes that the assembly language programmer has to send to the VDU driver by calling OSWRCH directly.

Miscellaneous I/O

There are two machine code subroutines OSBYTE and OSWORD which control a wide range of MOS activities. Exactly what OSBYTE and OSWORD do is controlled by the code stored in the A register before they are called. Once again the programmer is spared from having to deal with a large number of separate and specialised subroutines by the simple expedient of using a single entry point and an action code. OSBYTE is made available to BASIC programmers in two ways. Firstly, it is used to implement many BASIC commands, For example, the INKEY function calls OSBYTE to read the keyboard. Secondly, many OSBYTE actions are available via the familiar *FX command. Only OSBYTE actions that do not return information to the program can be used via the *FX command, because although the * FX command can transfer data to OSBYTE it has no mechanism for returning it. The command *FX a,x,y places the value 'a' in the A register, 'x' in the X register and 'y' in the Y register and then calls OSBYTE. In this sense it is exactly equivalent to the assembly language:

```
LDA #a
LDX #x
LDY #y
JSR OSBYTE
```

If 'x' or 'y' are omitted the command will automatically load the appropriate registers with zero.

The OSWORD subroutine works in a very similar way to OSBYTE

but, as the amount of data involved is more than just the A, X and Y registers can hold, the *FX or an equivalent cannot be used to access OSWORD from BASIC. The only access to OSWORD that a BASIC programmer has is indirect and via the BASIC commands that call OSWORD to implement their operations. For example, the SOUND command sets up a data block of information about pitch, volume, etc., then it loads the A register with 7, sets the X and Y registers to point at the data block and jumps to OSWORD which does the real work of setting up the sound generator. OSWORD calls are listed in Section 43 of the *User Guide*.

The *FX commands are also well documented in the BBC Micro's manual and so it isn't worth repeating the list here. However, there area number of OSBYTE and OSWORD calls that are useful from BASIC but no direct command is provided to access them. For example, the OSWORD call with A set to 10 will return the dot definition of any character. The ASCII code of the character is stored in the first location of a data block and the dot pattern is stored in the following eight bytes of the block when OSWORD returns. Although this OSWORD call isn't available in BASIC it is easy to write a procedure to implement it:

```
1000 DEF PROCshape(CODE%)
1010 ?&600=CODE%
1020 A%=10
1030 Y%=&06
1040 X%=&00
1050 A%=USR(&FFF1)
1060 FOR A%=1 TO 8
1070 DOT%(A%)=A%?&600
1080 NEXT A%
1090 ENDPROC
```

The eight bytes of the dot pattern for the character corresponding to CHR$(CODE%) are returned in the array DOT% which must be dimensioned in the main program before PROCshape is used. The area of memory starting at &600 is used for the data block because it is unlikely to be in use for anything else. (It is normally used for the parameter block for a CALL statement.) As another example the OSWORD call with A set to 11 returns the logical to physical colour assignment asset by any previous VDU 19 commands. This is also not available as a direct BASIC command but it is easy to add it as a function:

```
2000 DEF FNphyscol(L%)
2010 ?&600=L%
2020 A%=11
2030 Y%=&06
2040 X%=&00
2050 A%=USR(&FFF1)
2060 =?&601
```

This will return the physical colour code assigned to the logical colour code stored in L%. You should be able to see the similarity between this function and the previous procedure.

The final two I/O procedures, OSRDCH and (for a second time) OSWRCH, are used to read and write single bytes of data to any of the character-oriented devices. Normally OSRDCH will return the ASCII code of any key pressed on the keyboard and OSWRCH will send the ASCII code of a character to the VDU driver. This is because the keyboard and the screen are the default I/O devices. It is possible to change this default selection using the OSBYTE subroutine and in this sense OSRDCH and OSWRCH are general purpose single character input and output routines. For example, following *FX 3,3 OSWRCH sends a character to the printer port and following *FX 2,1 OSRDCH gets characters from the serial port. (See the short example in the section on the interrupt and event handler.)

ROM paging

The ROM paging activities of the BBC Micro are unlikely to be of practical interest unless you plan to produce your own software in ROM. However, the general method used by the MOS to handle paging is interesting for its own sake and will be described in this section. If you do feel up to tackling the difficult and time-consuming problem of putting software into ROM then you will find all of the technical details that you need in a booklet entitled *BBC Micro ROM Paging Systems Explained*, produced by Watford Electronics (35/37 Cardiff Road, Watford, Herts. They can also supply all the extra hardware that you will need).

Although the 6502 inside the BBC Micro can only address 64K of memory directly, this has been extended by allowing any one of a possible 16 ROMs to occupy the 16K region from &8000 to &C000. You can think of these 16 ROMs as the pages of a book and at any one time the book can only be opened to reveal one of the pages. This method of sharing a limited amount of address space is usually called paging. From a hardware point of view the BBC Micro's paging system is relatively simple. If we assume that there are 16 ROM sockets numbered 0 to 15 then to select (or 'page in') the ROM in socket x all we have to do is to write the value of'x' to the ROM page register at &FE30. An unmodified BBC Micro has only four paged ROM sockets on the main PCB but this number can easily be extended to the full 16 sockets by the use of an expander board. (Some expander boards will even allow you to install 16K of RAM in the address range &8000 to &C000 and this makes ROM software much easier to develop.)

The hardware side of ROM paging is indeed very simple but the real difficulty inherent in any paging system is how to make sure that the required page is enabled when und only when it is needed this is clearly a software problem. To be more precise it is a problem that the MOS has to

solve. The 16K of software that constitutes the MOS is always in position at &C000 to &FFFF no matter which of the paged ROMs is selected. This makes it the ideal (in fact the only!) candidate for looking after the paging process.

The simplest paging action that the MOS carries out is detecting which ROM sockets actually have a valid ROM installed and then paging the one with the highest number into the address space. This happens when the machine is first switched on or when BREAK is pressed. If you are using an unmodified BBC Micro then it might be of use to know that the ROM sockets are numbered from 12 on the left-hand side making the right-most socket number 15. (The socket on the far left of the row of five is not a page ROM socket but holds the MOS ROM instead.) In other words, this paging action makes sure that when the machine is first switched on, or when BREAK is pressed, the ROM in the right-most socket is enabled this is usually BASIC but it could equally well be any other language or, say, a word processor.

For a ROM to be detected its first few bytes must conform to the following format:

Offset (from &8000)	Size (in bytes)	
0	3	JMP language
3	3	JMP service
6	1	ROM type
7	1	Length of information area.
8	1	Version number (in binary).
9	as required	Title string ending with 00.
-	as required	Version string ending with 00.
-	as required	Copyright string starting with '(C)' and ending with 00.
-	4	Tube relocation address.

The most interesting of these entries are the JMP language and JMP service instructions. Each ROM can consist of two parts a language part and a service part and these two instructions are used by the MOS to enter either part as required. The language part of a ROM is characterised by being a candidate for transfer over the Tube to run on a second processor and for this reason it must avoid any direct reference to the main processor's I/O locations it must use the MOS for all I/O operations, A service part of a ROM is not a candidate for transfer over the Tube and as such it can make direct use of the main processor's I/O locations and generally get involved with the hardware. Service ROMs are usually concerned with providing additional software support to other programs - i.e. they provide a service to other programs. For example, the Acorn disk operating system is a service ROM. On the other hand language ROMs contain programs like the BASIC interpreter or word processors that make use of service ROMs to handle the hardware. This distinction isn't clear cut, however, and most ROMs have both a service and a language entry.

The rest of the entries are mainly about describing the nature of the ROM. The type byte of each valid ROM that is detected is stored in a table, the address of which can be found by using OSBYTE call 170. (The start of the table is returned in the X and Y registers.)

If the selection of a single ROM from a number of possibilities was the only sort of paging that the MOS carried out then there would be little advantage in using it! In fact the ROM paging system is used a great many times during the normal running of a program. The language part of any ROM can be entered at any time by using a *FX 142, ROM number command or its OSBYTE equivalent. This OSBYTE command can be issued by any of the paged ROMs or the. MOS itself. If it comes from another paged ROM, control is first passed to the MOS which then performs the necessary paging operation.

In the same way, the service part of a ROM can be entered using *FX 143 or its equivalent OSBYTE call. However, in this case the call is somewhat more complicated. The full form of the *FX command is:

   *FX 143,call number,parameter

where 'call number' describes the reason for the calland 'parameter' supplies any additional information that is necessary. Notice that unlike the language call this service call does not name the particular ROM socket number that should deal with the call. The reason for this is that this number will usually be unknown to the MOS and instead it will page in each of the ROMs in turn and enter the service part with the 'call number' in the A register. In this way the service part of each ROM is given the chance to either take notice of or act on the call. The effects of most of the call numbers are difficult to explain exactly but the following table gives a summary.

Call Number	Action
0	NOP - no operation - if a service ROM acts upon a call and needs to make sure it will be ignored by the remaining ROMs it can set A to zero before returning to the operating system.
1	Work space required each ROM may state its needs for working memory.
2	Private space required each ROM may state its need for private, i.e. non shareable, work space.
3	Auto-boot when BREAK is pressed each ROM is given a chance to test the keyboard to see if its 'start up' key is pressed. If it is then the ROM may transfer control to its language section or run any other program it likes.

4	Command not recognised by MOS. If a command starting with '*' is not one that the MOS recognises it offers it to each of the ROMs in the hope that one of them will recognise and act on it. On entry Y,(&F2) points to the start of the command.
5	IRQ not recognised. The MOS cannot identify the source of an IRQ interrupt and asks each ROM in turn to try to identify it. If a ROM is not using any interrupts then it simply returns control to the MOS.
6	BRK a BRK instruction has to be executed i.e. the system is in an error condition.
7	OSBYTE call not recognised. An OSBYTE call that the MOS doesn't recognise is passed to each of the ROMs to see if it is one that they can handle. The OSBYTE parameters A,X and Y are stored in &EF,&F0 and &F1 respectively on entry and are also used to return any results.
8	OSWORD call not recognised. As in the case of an unrecognised OSBYTE call each ROM is given a chance to process the unrecognised OSWORD call. Notice that OSWORD calls in the range &80 to &FF indirect via USRV rather than cause ROM paging.
9	HELP information required each ROM should print its title line, etc.
A	Claim main work space a ROM wishes to use the work space.
B	NMI released.
C	NMI claimed.
D	Initialise *ROM filing system.
E	Return a single data byte for *ROM filing system.
F	Claim MOS indirection vectors.
10	Filing system is about to close *SPOOL and *EXEC files.
11	Character set about to implode/explode.
12	Initialise filing system.
FE	Tube post initialisation.
FF	Tube hardware present.

Many of these calls simply inform any ROMs that might be active of a system condition but four of them effectively extend the facilities offered by the MOS. Call 4 extends the range of '*' commands that are recognised, call 5 extends the range of interrupts that are recognised (seethe next section), call 7 extends OSBYTE and call 8 extends OSWORD. For example, suppose you have written a new word processor and you want to call it 'BWORD'. If you included a service entry that recognised the letters of'BWORD' then you could be paged in by the command '*BWORD'. When the MOS encounters this command it would find that it wasn't one

that it recognised and it would begin to page in the ROMs one by one and enter their service section with call number 4 unrecognised command. Each ROM would then check to see if they recognised the command. Of course, if you have chosen the name 'BWORD' carefully only the service section of your word processor ROM will recognise it. To get the word processor going the service section of your ROM would call the MOS using *FX 142,ROM number which would then enter your ROM at the language entry point.

As a more simple example, suppose your new ROM implements a command that makes a specific sound effect and you want it to be added to the MOS as OSBYTE &A0. When the MOS is entered via an OSBYTE call with a code of &A0 it cannot identify what action to take and so it starts to page each ROM in turn using call number 7. Each ROM will inspect the unknown OSBYTE call and return control to the MOS if it too fails to recognise it. Of course when your new ROM is paged it will recognise the OSBYTE call and so will proceed to make the required sound effect. Finally your ROM should set A to zero and return to the MOS so that none of the remaining ROMs will even bother to examine the call.

There is much more to say about how the MOS ROM paging mechanism can extend the MOS. In particular, there is the whole subject of filing systems and how new filing systems can be written so as to integrate with the existing software but all this is beyond the scope of this book. If you are interested then you will find the extra details that you need in the booklet from Watford Electronics mentioned at the start of this section and this, coupled with the overall description given here, should be sufficient for you to implement your own ROM software.

The interrupt and event handler

While the BBC Micro is running interrupts are generated every hundredth of a second by one of the timers inside VIA A. Each time one of these regular interrupts is received the interrupt handler, whose address is stored at &204, is called. This updates the pseudo variable TIME and then checks to see if anything else needs attention. For example, it is responsible for altering the parameters of the sound generator for the duration of a sound controlled by an envelope. It is this action that makes the BBC Micro's sound generator so powerful. From a hardware point of view it is not at all impressive but when it is added to the frequent and periodic attention that the interrupt handler provides it is as good as more complex hardware and far more flexible.

The interrupt handler also looks after the system of queues which are such an important feature of I/O through the MOS. Using the sound generator as an example for the second time, it is easy to see that at each interrupt the interrupt handler has to check to see if the current sound has reached its specified duration or not. If it hasn't then the sound generator

is left alone. If it has then the sound queue associated with that channel is inspected. If it is empty then the channel is switched off, otherwise the parameters stored in the queue are used to start the next tone and the queue is moved up by one. This interrupt servicing of the sound queue is responsible for the BBC Micro apparently being able to do two things at once - run a BASIC program and generate a sound. The printer and serial ports are both associated with queues or buffers that are serviced by the interrupt handler and, as in the case of the sound generator, this greatly enhances the performance of the entire system. For example, it is quite possible for a BASIC program to print out a large quantity of data while getting on with another job. All it has to do is send data to the printer buffer until it detects, using ADVAL(-4), that it is full. Then it can get on with another job and the interrupt handler will automatically send a character at a time to the printer, at the rate that the printer can accept. Of course, if left in this state all that would have been printed is a single buffer full of data but if, while carrying out its other jobs, the BASIC program occasionally checks to see if the buffer is once again empty (using ADVAL(-4)) and transfers enough data to keep it topped up, any amount of data can be printed without stopping the BASIC program unnecessarily. As an illustration of this method the following short program will print powers of two on the printer and a sequence of integers on the screen.

```
  10 I=0
  20 J=2
  30 IF ADVAL(-4)=63 THEN PROCprint(J)
  40 I=I+1
  50 PRINT I
  60 GOTO 30

1000 DEF PROCprint(J)
1010 REPEAT
1020 VDU 2,21:PRlNT J:VDU 6,3
1030 J=FNupdate(J)
1040 UNTIL ADVAL(-4)<10
1050 ENDPROC

2000 DEF FNupdate(J)
2010 =J*J
```

The first part of the program, consisting of lines 10,40,50 and 60, is easily recognised as a simple infinite counting loop. Line 30, however, tests to see if the printer buffer is completely empty. If it is then PROCprint is called to refill it. This ensures that, as the values of I are printed on the screen, the printer is periodically given something to keep it busy. PROCprint simply sends as many values of J as the printer buffer can hold and then returns control to the main program. Notice that the REPEAT loop, lines 1010 to 1040, comes to an end ten characters before the buffer

is full. The reason for this is that the PRINT statement adds a number of characters to the buffer each time it is executed and if the buffer became full during this print statement the program would have to wait until space was made available. This would of course defeat the object of the program. Also notice that following each addition to the buffer FNupdate is called to provide a new value for J. In practice FNupdate would usually be a function to read single characters from a text tile stored on disk. so allowing a file to be listed while the program gets on with something else. You might be a little worried by line 1020 as it uses VDU 21 and VDU 6 to disable and enable screen output respectively, rather than the *FX called mentioned in the section on miscellaneous I/O. The reason for this is that *FX 3 appears not to work when the printer is the only selected output device (i.e. *FX 3,2 will not select the printer). This presents no real difficulty as VDU codes 21 and 6 can be used to send output to the printer without affecting the screen. It is quite possible to write a machine code version of the above program that would print the contents of a file while an entirely different program was being used. How this could be done will be easier to see after the following example.

A background clock

Interrupts are such an important part of the BBC Micro that it is worth giving an example of a complete project that makes use of this facility. The aim of the project is to add a 'background clock' that will display hours, minutes and seconds on the screen no matter what the BBC Micro is doing and without interfering with the running of any other programs. The screen display part of this problem has already been solved in connection with the trace program described in Chapter Nine and the only really new element is how to update the display periodically. Perhaps the most obvious method of updating a clock is to use the regular interrupts to enter a routine that increments the current time. However, interrupts occur every hundredth of a second and the background clock only has to 'tick' once every second. It would be possible to count one hundred interrupts and then increment the time but in the case of the BBC Micro this would be a complicated way of doing things. As well as a five byte timer that is used to provide the pseudo variable TIME, the MOS also maintains a separate 'interval timer' that is incremented with each interrupt. This interval timer can be read using OSWORD 3 and set to any given value using OSWORD 4. For this particular project the most important feature of the interval timer is that it can be used to cause a periodic event that repeats after any given interval. If the 'interval timer crossing zero' event is enabled using *FX 14,6 then an event will occur each time the event timer reaches a count of zero. As the event timer counts up to & FFFFFFFFFF and then resets itself to zero, the time between these events can be set by storing an appropriate value in the event timer. In this case we need an event to occur after one second, i.e. after one hundred

interrupts, so the value that should be stored in the interval counter is & FFFFFFFFFF-100 (because this will make the interval timer reset to zero after being incremented 100 times).

Now that we have a method of making the clock 'tick' every second the rest of the project is mamly about routine details such as how the data should be represented, etc. (The rest of this discussion assumes that you know something about simple binary and BCD) representation - if this is not the case then you will find these ideas dealt with at length in Chapter Twelve.) Obviously we are going to have to store a two digit value for each of hours, minutes and seconds, increment them in such a way that a count of 60 seconds adds one to the minutes and a count of 60 minutes adds one to the hours. We could use simple binary to store the values but this would make printing them out as decimal digits rather difficult. Instead it makes sense to use a BCD (Binary Coded Decimal) representation. BCD is explained more fully in Chapter Twelve but essentially BCD uses four bits to represent a single decimal digit in the range 0 to 9. Thus a single memory location can store a pair of BCD digits and this is particularly convenient for our application as a single memory location can be used for the two digit hours, minutes and seconds.

After so much discussion, writing the main program is easy:

```
3000 .TICK%
3010 ]PROCsave:[OPT PASS
3020          JSR UPDATE%
3030          JSR DISPLAY%
3040          JSR RESET%
3045 ]PROCrestore:[OPT PASS
3050 RTS
```

PROCsave and PROCrestore are the two macros introduced as part of the trace program in Chapter Nine which save and restore all of the 6502 registers - this has to be done for all but the simplest event handling routine as the register values have to be preserved. UPDATE% is simply a routine to increment the time. DISPLAY% prints the time at the top of the screen and then restores the cursor's position. Finally RESET% sets the interval timer's value so that an event will occur again after one second has passed.

The second stage of refinement is fairly easy. The UPDATE% routine simply adds one to SEC% and checks to see if it has reached 60. If it has then it is zeroed and one is added to MIN% which is also checked to see if it has reached 60. If it has then it too is zeroed and one is added to HOUR% which is then checked to see if it has reached 24 (the background clock is a 24 hour clock!). If it has then it is also zeroed. The DISPLAY%, routine is essentially the same as the routine used in the trace program to alter the cursor position. The only difference is that it now calls DISPNUM% to print the time before returning the cursor to its

original position. RESET% simply uses a call to OSWORD 4 to set the interval timer and needs no comment.

The third and fourth stages of refinement are now quickly implemented as only DISPNUM% remains undefined. DISPNUM% is straightforward, if a little tedious in that it uses a pair of subroutines, L_NUM% and RNUM%, to convert the left (i.e. bit 7 to bit 4) and the right (i.e. bit 3 to bit 0) digits of a BCD number into valid ASCII digits and then uses OSWRCH to print the results.

The only part of the program that remains is a routine that will alter the 'event vector' at &220 to contain the address of the start of theclock program and enable the interval timer to reach zero event. This can be seen in the complete version of the program given below as INIT%.

```
  10 DIM CODE% 500
  20 PROCasm(CODE%)
  30 CALL INIT%
  40 STOP

1000 DEF PROCasm(START%)
1005 FOR PASS=0 TO 3 STEP 3
1010 P%=START%
1020 PROCprog
1030 NEXT PASS
1040 PRINT
1050 PRINT P%-START%;" Bytes"
1060 ENDPROC
2000 DEF PROCprog
2010 OSBYTE%=&FFF4
2020 OSWRCH%=&FFEE
2030 EVNTV%=&220
2040 OSWORD%=&FFF1
2990 [OPT PASS

3000 .TICK%
3010 ]PROCsave:[OPT PASS
3020           JSR UPDATE%
3030           JSR DISPLAY%
3040           JSR RESET%
3045 ]PROCrestore:[OPT PASS
3050           RTS
3060 \
3070 .UPDATE%   SED
3080           CLC
3090           LDA SEC%
```

```
3100                    ADC #&01
3110                    CMP #&60
3120                    BNE UP2%
3130                    CLC
3140                    LDA MIN%
3150                    ADC #&01
3160                    CMP #&60
3170                    BNE UP1%
3180                    CLC
3190                    LDA HOUR%
3200                    ADC #&01
3210                    CMP #&24
3220                    BNE UP3%
3230                    LDA #&00
3240  .UP3%            STA HOUR%
3250                    LDA #0
3260  .UP1%            STA MIN%
3270                    LDA #0
3280  .UP2%            STA SEC%
3290                    CLD
3300                    RTS
3310  \
3320  .DISPLAY%        LDA #134
3330                    JSR OSBYTE%
3340                    STX XTEMP%
3350                    STY YTEMP%
3360                    LDA #31
3370                    JSR OSWRCH
3380                    LDA #20
3390                    JSR OSWRCH%
3400                    LDA #0
3410                    JSR OSWRCH%
3420                    JSR DISPNUM%
3430                    LDA #31
3440                    JSR OSWRCH%
3450                    LDA XTEMP%
3460                    JSR OSWRCH%
3470                    LDA YTEMP%
3480                    JSR OSWRCH%
3490                    RTS
3500  \
3510  .DISPNUM%        LDA HOUR%
3520                    JSR L_NUM%
3530                    JSR OSWRCH%
3540                    LDA HOUR%
```

```
3550              JSR RNUM%
3560              JSR OSWRCH%
3570              LDA #58
3580              JSR OSWRCH%
3590              LDA MIN%
3600              JSR L_NUM%
3610              JSR OSWRCH%
3620              LDA MIN%
3630              JSR RNUM%
3640              JSR OSWRCH%
3650              LDA #58
3660              JSR OSWRCH%
3670              LDA SEC%
3680              JSR L_NUM%
3690              JSR OSWRCH%
3700              LDA SEC%
3710              JSR RNUM%
3720              JSR OSWRCH%
3730              RTS
3740 \

4000 .L_NUM%      LSR A
4010              LSR A
4020              LSR A
4030              LSR A
4040              ORA #&30
4050              RTS
4060 \
4070 .RNUM%       AND #&0F
4080              ORA #&30
4090              RTS
4100 \
4110 .RESET%      LDX #(T_LEN% MOD 256)
4120              LDY #(T_LEN% DIV 256)
4130              LDA #4
4140              JSR OSWORD%
4150              RTS
4160 \

4200 .INIT%       LDA #(TICK% MOD 256)
4210              STA EVNTV%
4220              LDA #(TICK% DIV 256)
4230              STA EVNTV%+1
4240              JSR RESET%
4250              LDA #14
```

```
4260                 LDX #5
4270                 JSR OSBYTE%
4280                 RTS

5000 ]
5010 REM DATA
5020 HOUR%=FNequb(0)
5030 MIN%=FNequb(0)
5040 SEC%=FNequb(0)
5050 XTEMP%=FNequb(0)
5060 YTEMP%=FNequb(0)
5070 T_LEN%=FNequb(155)
5080 dummy=FNequw(&FFFF)
5090 dummy=FNequw(&FFFF)
5110 ENDPROC

9000 REM MACROS
9010 DEF FNequb(VA%)
9020 ?P%=(VA% MOD 256)
9025 IF PASS=3 THEN PRINT ~P%;"=";~?P%
9030 P%=P%+1
9040 =P%-1

9100 DEF FNequw(VA%)
9110 ?P%=(VA% MOD 256)
9115 IF PASS=3 THEN PRINT ~P%;"=";~?P%
9120 P%?1=(VA% DIV 256)
9125 IF PASS=3 THEN PRINT ~P%+1;"=";
     ~(P%?1)
9130 P%=P%+2
9140 =P%-2

9200 DEF FNequs(S$)
9210 $P%=S$
9220 IF PASS=3 THEN PRINT ~P%;"=";S$
9230 P%=P%+LEN(S$)+1
9240 =P%-LEN(S$)-1

9300 DEF PROCsave
9400 [OPT PASS
9410 PHA
9420 PHP
9430 TXA
9440 PHA
9450 TYA
```

```
9460 PHA
9470 ]
9480 ENDPROC

9500 DEF PROCrestore
9510 [OPT PASS
9520 PLA
9530 TAY
9540 PLA
9550 TAX
9560 PLP
9570 PLA
9580 ]
9590 ENDPROC
```

If you run the above program you will see the clock display appear at the top of the screen as soon as INIT% has been called. This display will remain until you press CTRL and BREAK, so forcing a full system restart or until you load another program over the machine code stored in CODE%.

Obviously to make the program of permanent use it has to be installed in the system in such a way that it cannot be overwritten by accident. The only real way of doing this is to increase the value of PAGE and store it in the free memory that this creates. To be precise:

(1) Save the above program on tape or disk and then perform a full system reset (either by switching the machine off and on or by pressing CTRL and BREAK).

(2) Enter

PRINT~PAGE

and note down the result.

(3) Now enter PAGE=PAGE+256 and once again PRINT PAGE and note down the result. This reserves 256 bytes for the clock program.

(4) Now load the clock program and change line 20 to read PROCasm(x) where x is the original value of PAGE noted down in step 1. Also remove line 30 to stop the clock from being started.

(5) Now run the program and enter PRINT INIT% and note down the result. This gives the address of the start of the program.

(6) Finally using the information collected in steps (1), (3) and (5) enter:

*SAVE TIMER x y z

where x is the value of PAGE as obtained instep (1), y is the value of
PAGE obtained in step (3) and z is the value of INIT% obtained in step
(5). Following steps (1) to (5) all you have todoto run the timer is to
remember to reserve the memory space that it needs using PAGE=PAGE+
256 and then enter the command *RUN TIMER.

　　The background timer given above certainly works and does the job
that was intended, but it suffers from a few shortcomings. The most
obvious is that there is no way to set M to the current time! This is not a
serious problem as it is not difficult to write a BASIC program that will
store the values of the current time in the memory locations used tor
HOUR%, MIN% and SEC%. A more serious defect is that the
DISPLAY% routine affects the working of the COPY key by moving the
copying cursor as well as the text cursor back to its 'original' position.
This is typical of the sort of unexpected side effect of an interrupt or event
driven program. In this case the only solution is to provide a subroutine
that will print the time on the screen without using the MOS and this is a
very large project. A much smaller criticism of the program is that it
doesn't check to make sure that the event that has happened really is the
interval timer reaching zero after all there might be other enabled events.
As each type of event corresponds to a different event code stored in the A
register, this is easy to remedy. The code for the event timer crossing zero
is 5 so, adding:

```
2991 CMP #5
2992 BEQ TICK%
2993 RTS
```

will ensure that the clock is only updated by the correct event.

　　You can use the methods outlined in this project to use events in your
own programs. In particular, you should now be able to see how to print
the contents of a file while running other programs by using the 'print
buffer empty' event to call a routine that will fill the print buffer.
However, it is worth mentioning that programs that use interrupts or
events can be very difficult to understand and very difficult to debug so
proceed with care!

Chapter Eleven
Project – A 6502 Disassembler

The BBC Micro comes equipped with a very good 6502 assembler that makes the translation of 6502 mnemonics to 6502 machine code easy. The reverse process – converting 6502 machine code to 6502 mnemonics – may be something that seems pointless until you attempt to decipher how a machine code program works. Obviously if you have the assembly language source for the machine code then all you have to do is study it. If for any reason you don't haw access to the assembly language source then one way or another you have to attempt to reconstruct it from the machine code.

A program that changes machine code to assembly language is called a *disassembler*. The main theme of this chapter is the production of a 6502 disassembler but this takes us into some surprising areas concerning thevery fundamentals of microprocessors, look-up tables and program design. As well as providing plenty of scope for discussion, the final program is something that every BBC Micro owner needs now and again. The reason for this is that the BBC Micro contains 32K of very important machine code only software – BBC BASIC and the MOS. Some of the topics are admittedly rather advanced. In fact. one or two are the sort of thing that you would normally only encounter on a computer science degree so you can be pleased it they cause you no trouble! On the other hand. if you find any of the material tough going do not despair, you can still make use of the disassembler and understand many of the points of its implementation without understanding everything in this chapter.

The trouble with tables

The most obvious approach to constructing a disassembler, or an assembler for that matter, is to use a look-up table (see Chapter Five). The basic algorithm of a 'table driven' disassembler is very simple the contents of each memory location is used to index a pair of tables that give:

(1) the three letter mnemonic to which it corresponds and
(2) the addressing mode (e.g. absolute, immediate, etc.) that is in use.

If the contents of a memory location do not correspond to a legal 6502 instruction then a special 'invalid operation' mnemonic, "***" for example would be returned and the next memory location examined. Values within a machine code program that do not correspond to any 6502 operation are most likely to be generated by data within the ortginal assembly language source. However, the whole question of how to differentiate automatically between areas of code and data is one stage beyond the simple disassembler that is currently under discussion.

The only real difficulty with the look-up tables for the mnemonics and addressing modes are their size. To allow an entry for every possible single byte value needs two arrays each with 256 elements. That is,

    DIM M$(255),AM(255)

where M$(1) holds the three letter mnemonic for the op code 1 and AM(1) holds an integer that indicates the addressing mode. For example, M$(&65) would hold "ADC" and AM(&65) would indicate zero page addressing. (&65 is the machine code for ADC using zero page addressing!) If you think about the pattern of entries in these tables you will quickly realise that there are many duplicate entries. For example, the mnemonic for ADC occurs eight times, once for each of its possible addressing modes. Entering the complete set of 6502 mnemonics to form a table is a tedious enough task without entering some of them as many as eight times!

If you examine a 6502 instruction table (there is one at the back of the BBC Micro's *User Guide*) you should be able to see a pattern in the values for the op codes corresponding to a single instruction in each of its different address modes. Any pattern in the entries in a look-up table is worth investigating because it is often possible to make use of it to reduce the number of entries in the table.

The 6502 instruction format

To a certain extent the 6502 chip within the BBC Micro has the problem of machine code disassembly every time it executes a program. After reading an op code from memory the 6502 has to decode it and determine what actions have to take place. For example, after reading the op code & 65 the 6502 has to decode it to 'discover' that it means ADC using zero page addressing mode and then carry out the addition. The restrictions of building logic circuits on silicon make it necessary to find a way of making the decoding process as simple as possible. For this reason there is a very regular pattern to be found in the op code values used in any microprocessor and the 6502 is no exception.

By examining the table of instruction codes it is possible to deduce that the internal format of a 6502 op code is:

b7 b6 b5 b4 b3 b2 b1 b0

instruction code	mode	group
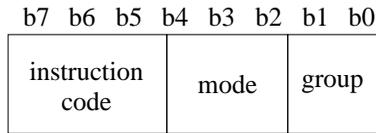
Only the top three bits (b7, b6 and b5) of each opcode are used to indicate the instruction that is to be carried out. The middle three bits (b4, b3 and b2) indicate which addressing mode is in use. However, using only three bits to determine the operation would limit the 6502 to a total of eight different instructions! To increase the total number of instructions the last two bits of the op code are used to provide four different groups of instructions. Thus the ADC zero page instruction can be analysed by writing down its op code in binary as a group one instruction:

&65 =

011	001	01
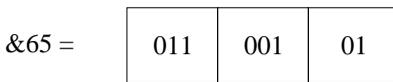
The instruction code for ADC is 3 and the mode code for zero page addressing is 1. In the same way ADC immediate is:

&69 =

011	010	01
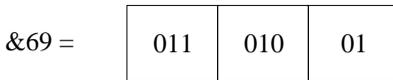
which again shows that ADC is a group 1 instruction and its instruction code is 3. The only change to the op code in going from ADC zero page to ADC immediate is carried in the three mode bits, the addressing mode code for zero page addressing being 1 and for immediate addressing 2.

You should be able to appreciate that this division of the bits that make up an op code into an instruction code, an addressing mode code and a group code can greatly simplify the decoding that is involved in both a microprocessor and a disassembler. For example, if the first two bits of an op code indicate a group one instruction and the top three bits give an instruction code of 3 then the command is ADC. Subsequent examination of the mode bits to give the current addressing mode complete the decoding.

6502 instructions by group

Before the internal structure of the op code can be used to produce a simplified disassembler, it is necessary to classify each 6502 instruction by group code and produce tables relating instruction codes to mnemonics and addressing mode codes to actual addressing modes. The way that this task was tackled originally was with the aid of the following program:

```
10 INPUT A$
20 A=FNdec(A$)
30 GR=A AND &03
40 MO=(A AND &1C)/4
50 OP=(A AND &E0)/32
60 PRINT "Code=";~A;" Group=";~GR;" OP=";OP;" Mode ";MO
70 GOTO 10
80 DEF FNdec(A$) EVAL("&"+A$)
```

This will analyse any 6502 op code and print out its group, instruction code and addressing mode code.

For reasons that will become apparent, it is better to start with an examination of the group 1 table:

| Instruction | | | | Mode | | | | |
Code	0	1	2	3	4	5	6	7
0	ORA	ORA	ORA	ORA	ORA	ORA	ORA	ORA
1	AND	AND	AND	AND	AND	AND	AND	AND
2	EOR	EOR	EOR	EOR	EOR	EOR	EOR	EOR
3	ADC	ADC	ADC	ADC	ADC	ADC	ADC	ADC
4	STA	STA	STA	STA	STA	STA	STA	STA
5	LDA	LDA	LDA	LDA	LDA	LDA	LDA	LDA
6	CMP	CMP	CMP	CMP	CMP	CMP	CMP	CMP
7	SBC	SBC	SBC	SBC	SBC	SBC	SBC	SBC
address mode	(z,X)	zero page	imm	abs	(z,Y)	z,X,	c,Y	c,X

(z = zero page address and c = two byte constant)

From this table you can see that all eight group 1 instructions are operations m the A register. Another striking feature is that each instruction can be used in any of the eight possible addressing modes. (As will be demonstrated this is not the case with the other instruction groups.) Also notice that the use of just three bits to code the addressing mode limits any 6502 instruction to eight addressing modes at most despite the fact that the 6502 supports 13 different addressing modes. This, to a certain extent, accounts for the odd collection of rules about which addressing mode can be used with which instruction.

As far as the disassembler is concerned, group 1 instructions are very easy to decode. All that has to be done is to use the instruction code bits to index a table of the eight mnemonics and the address mode bits to index a table that gives the addressing mode in use. The mnemonic table is simply:

Instruction Code	Mnemonic
0	ORA
1	AND
2	EOR
3	ADC
4	STA
5	LDA
6	CMP
7	SBC

If the 13 addressing modes of the 6502 are coded as:

mode	imm	abs	zero page	acc	imp	(z,X)	(z),Y	z,X	c,X	c,Y	rel	indir	z,Y
code	1	2	3	4	5	6	7	8	9	10	11	12	13

then correspondence between the addressing mode code in group 1 instructions and actual addressing mode is:

addresing mode code (b4,b3,b2)	actual mode code
0	6
1	3
2	1
3	2
4	7
5	8
6	10
7	9

This may seem a little complicated at first but the reward is reducing the 64 entries in the simple look-up tables to 8. For example, to decode the opcode &1D it is first split up to give:

Group=1, Instruction code=0, Mode=7

As this is a group 1 instruction the two tables given above yield ORA for the mnemonic and address mode 9 or c,X (absolute indexed addressmg using the X register) which are both correct.

The situation with group 0 instructions is not so simple. The table for group 0 is:

Instruction Code	Mode 0	1	2	3	4	5	6	7
0	BRK	***	PHP	***	BPL	***	CLC	***
1	JSRabs	BIT	PLP	BIT	BMI	***	SEC	***
2	RTI	***	PLA	JMP	BVC	***	CLI	***
3	RTS	***	PLA	JMPind	BVS	***	SEI	***
4	***	STY	DEY	STY	BCC	***	TYA	***
5	LDYimm	LDY	TAY	LDY	BCS	***	CLV	***
6	CPYimm	CPY	INY	CPY	BNE	***	CLD	***
7	CPXimm	CPZ	INX	CPX	BEQ	***	SED	***
address mode	impl	zero page	impl	abs	rel		impl	

(where the addressing modes are as shown at the bottom of each column unless otherwise indicated within the table next to the instruction concerned)

This table reveals a much more complicated pattern. In particular, the instruction code part of the op code is not the sole determinant of the instruction type. Indeed, for group 0 instructions the addressing mode also determines the instruction type. For example, there are five different instructions corresponding to Instruction code=3 – that is, RTS, PLA, JMP, BVS and SEI. However this is not to say that the neat pattern of group 1 instructions is entirely lost. In the main, the instructions corresponding to a particular value of mode do use the same addressing mode. For example, all the instructions with mode=1 use zero page addressing, those with mode 2 use implied addressing, those with mode=4 use relative addressing and those with mode=6 use implied addressing. When mode=0 or mode=3 the situation is a little more difficult. The instructions in mode 0 are a mixture of implied addressing (BRK, RTI and RTS), absolute addressing (JSR) and immediate addressing (LDY, CPY and CPX) while all but one of those in mode 3 are absolute and JMP is indirect. Even so you should be able to see that there is a good overall regularity which simplifies the decoding process for the 6502's logic circuits. (For example, all mode 4 instructions are branches and all mode 6 instructions clear or set flags.) As far as the disassembler is concerned the simplest solution is to use the table as it stands i.e. as a two-dimensional look-up table indexed by the instruction and address mode part of the op code. The look-up table for the addressing modes is best teated as a one-dimensional table with corrections, as in the following:

addressing mode code	actual addressing mode
0	5

1	3
2	5
3	2
4	11
5	0
6	5
7	0

In the above table actual addressing code 0 is used to indicate that there are no 6502 instructions using this mode in group 0. Of course this table is not always correct - a mode 0 instruction doesn't always use implied addressing and mode 3 instructions don't always use absolute addressing but the departures can be easily detected and corrected using IF statements.

Group 2 instructions are also messy but in a different way to group 0 instructions:

Instruction code				Mode				
	0	1	2	3	4	5	6	7
0	***	ASL	ASL	ASL	***	ASL	***	ASL
1	***	ROL	ROL	ROL	***	ROL	***	ROL
2	***	LSR	LSR	LSR	***	LSR	***	LSR
3	***	ROR	ROR	ROR	***	ROR	***	ROR
4	***	STX	TXAimp	STX	***	STXz,Y	TXS	***
5	LDX	LDX	TAXimp	LDX	***	LDXz,Y	TSX	LDYc,Y
6	***	DEC	DEXimp	DEC	***	DEC	***	DEC
7	***	INC	***	INC	***	INC	***	INC
	imm	aero	acc	abs		z,X	impl	c,X

(where the addressing modes are as shown at the bottom of each column unless otherwise indicated within the table next to the instruction concerned)

The interesting thing about this table is that the top half (i.e. Instruction code = 0 to 3) is entirely regular but the bottom half is irregular. The look-up table for this group could also be created as a two-dimensional array as for the group 0 instructions. However, overall the regularities in the table outnumber the irregularities and it is easier to use a one-dimensional table based on the pattern of mnemonics in the mode=1 column and use IF statements to correct for the departures from regularity. In the same way the addressing mode look-up table can be one-dimensional:

Addressing mode code	Actual addressing mode used
0	1
1	3
2	4
3	2
4	0
5	8
6	5
7	9

The departures from this pattern are taken care of by using IF statements.

After the complications introduced by the irregularities of group 0 and group 2 instructions it is a relief to discover that the 6502 doesn't have any group 3 instructions!

How good is the 6502?

Apart from reducing the size of the look-up tables needed for a disassembler, this study of the structure of the 6502 instruction set explains many of the odd patterns of addressing modes that can be used with any particular instruction. One opinion is that an advanced microprocessor should allow every instruction to use any and all addressing modes that make sense. From this point of view the 6502 is a very poor micro indeed as its rules for which addressing modes can be used with which instructions contain many seemingly arbitrary restrictions. For example, why can't any addressing mode that uses the Y index register be used with the ROL or ROR instructions? A simple explanation would be to say that ROL and ROR are group 2 instructions but this misses the point that there are three unused addressing modes in the group 2 table and the whole of the group 3 table is empty! However, this argument can be countered by pointing out that by using only a single byte for the op code (other microprocessors can use two or even three byte op codes) the 6502 provides a good range of instructions, each of which can be used with an appropriate collection of addressing modes. From this point of view the 6502's design sacrifices simplicity of use for a compact and efficient instruction set.

The disassembler

After so much analysis the 6502 disassembler is fairly easy to write. The

procedures used are shown in Table 11.1.

Table 11.1

Name	Line number	Function
initialise	1000	Sets up the look-up tables for each instruction group.
getparams	2000	Input start and end address for the disassembly.
code(D)	3000	Returns the mnemonic (in M$) and addressing mode (in ATYPE) corresponding to the op code in D
gzero(OP,MO)	4000	Performs the look-up for group 0 instructions.
gone(OP,MO)	4200	Performs the look-up for group 1 instructions.
gtwo(OP,MO)	4400	Performs the look-up for group 2 instructions.
gthree(OP,MO)	4600	Returns M$=*** and ATYPE=0 for non-existent group three instructions.
add(ATYPE)	5000	Uses ATYPE to return a string (A$) that contains the address field of the instruction. Also returns the contents of the memory locations that it uses to construct the address as hex numbers in B$.

PROCinitialise sets up six different look-up tables, one pair for each instruction group. G1$ and A1% hold the mnemonics and addressing modes for group one instructions, GO$ and A0% hold the mnemonics and addressing modes for group 0 instructions and G2$ and A2% hold the mnemonics and addressing modes for group two instructions. If you look at PROCcode you will see that, apart from dividing up the three parts of the op code (lines 3020 to 3040) all it does is to call the correct PROC to deal with the particular group that the instruction belongs to (lines 3050 to 3080). PROCone is the simplest because the look-up tables can be indexed by OP and MO and the result used without correction. PROCzero uses the two- dimensional look-up table for the mnemonic code and a one-dimensional table for addressing mode. Unlike PROCone the results from the look-up table have to be corrected for irregularities in the addressing modes such as JSR (in the column corresponding to mode=0) using absolute addressing rather than implied. These addressing mode irregularities are corrected by the IF statements in lines 4030 to 4050. PROCgtwo uses a pair of one- dimensional look-up tables like PROCgone but in this case there are a large number of corrections to both addressing mode and mnemonic. These are dealt with by the IF statements lines 4430 to 4550. PROCgthree is almost a dummy routine that returns "***" and ATYPE=0. It is included more for completeness than anything else. PROCadd uses the information about the addressing mode stored in A TYPE to construct the address field that goes with the instruction. For example, if A TYPE is 1 then the addressing mode is immediate and the

address field is constructed by 'peeking' the value in the next memory location, converting it to a string of hex digits and then adding a "#" in front. Each of the adddresing modes has a corresponding IF statement that constructs the instruction's address field in AD$. The only other complication is the need to return the contents of any memory locations that are used to construct the address field as hex digits in B$ so that they can be listed alongside the disassembled instructions.

Three functions are also used:

Table 11.2

Name	Line number	Action
dec(A$)	9000	Converts hex string to decimal number.
rel(A)	9010	Returns positive of negative offset used in relative addressing from the positive value returned by 'peeking' th address field.
hex(A)	9030	Converts decimal number in A to a hex string.

The complete program is:

```
 10 PROCinitialise
 20 PROCgetparams:
    REM SADD,EADD returned
 30 A=SADD
 40 REPEAT
 50 PRINT TAB(0);~A;" ";
 60 PRINT TAB(8);FNhex(A);" ";
 70 PROCcode(?A):REM M$,ATYPE returned
 80 PROCadd(ATYPE):REM AD$ returned
 90 PRINT B$,M$;" ";AD$
100 UNTIL A>EADD
110 END

120 DEF PROCinitialise
130 LOCAL I,J
140 DATA ORA,AND,EOR,ADC,STA,LDA,CMP,SBC
150 DATA 6,3,1,2,7,8,10,9
160 DATA BRK,JSR,BIT,RTS,***,LDY,CPY,CPX
170 DATA ***,BIT,***,***,STY,LDY,CPY,CPX
180 DATA PHP,PLP,PHA,PLA,DEY,TAY,INY,INX
190 DATA ***,BIT,JMP,JMP,STY,LDY,CPY,CPX
200 DATA BPL,BMI,BVC,BVS,BCC,BCS,BNE,BEQ
```

```
210 DATA CLC,SEC,CLI,SEI,TYA,CLV,CLD,SED
220 DATA 1,3,5,7,11,0,5,0
230 DATA ASL,ROR,LSR,ROR,STX,LDX,DEC,INC
240 DATA 1,3,4,2,0,8,5,9
250 DIM G1$(7)
260 FOR I=0 TO 7
270 READ G1$(I)
280 NEXT I
290 DIM A1%(7)
300 FOR I=0 TO 7
310 READ A1%(I)
320 NEXT I
330 DIM G0$(7,7)
340 FOR I=0 TO 7
350 FOR J=0 TO 7
360 IF I=5 OR I=7 THEN G0$(I,J)="***"
    ELSE READ G0$(I,J)
370 NEXT J
380 NEXT I
390 DIM A0%(7)
400 FOR I=0 TO 7
410 READ A0%(I)
420 NEXT I
430 DIM G2$(I)
440 FOR I=0 TO 7
450 READ G2$(I)
460 NEXT I
470 DIM A2%(7)
480 FOR I=0 TO 7
490 READ A2%(I)
500 NEXT I
510 ENDPROC

520 DEF PROCgetparams
530 LOCAL A$
540 REPEAT
550 INPUT "START AT (HEX)",A$
560 SADD=FNdec(A$)
570 INPUT "END AT (HEX)",A$
580 EADD=FNdec(A$)
590 UNTIL SADD<EADD
600 ENDPROC

610 DEF PROCcode(D)
620 LOCAL MO,GROUP,OP
```

```
630 OP=(D AND &F0)/32
640 MO=(D AND &1C)/4
650 GROUP=D AND &03
660 IF GROUP=0 THEN PROCgzero(OP,MO)
670 IF GROUP=1 THEN PROCgone(OP,MO)
680 IF GROUP=2 THEN PROCgtwo(OP,MO)
690 IF GROUP=3 THEN PROCgthree(OP,MO)
700 ENDPROC

710 DEF PROCgzero(OP,MO)
720 M$=G0$(MO,OP)
730 ATYPE=A0%(MO)
740 IF MO=0 AND OP=1 THEN ATYPE=2
750 IF MO=0 AND (OP=0 OR OP=2 OR OP=3)
    THEN ATYPE=5
760 IF MO=3 AND OP=3 THEN ATYPE=12
770 ENDPROC

780 DEF PROCgone(OP,MO)
790 M$=G1$(OP)
800 ATYPE=A1%(MO)
810 ENDPROC

820 DEF PROCgtwo(OP,MO)
830 M$=G2$(OP)
840 ATYPE=A2%(MO)
850 IF ATYPE=0 THEN M$="***"
860 IF MO=0 AND OP<>5 THEN M$="***"
870 IF MO=2 AND OP=4 THEN M$="TXA":
    ATYPE=5
880 IF MO=2 AND OP=5 THEN M$="TAX":
    ATYPE=5
890 IF MO=2 AND OP=6 THEN M$="DEX":
    ATYPE=5
900 IF MO=4 THEN M$="***"
910 IF MO=5 AND (OP=4 OR OP=5) THEN
    ATYPE=13
920 IF MO=6 THEN M$="***"
930 IF MO=6 AND OP=4 THEN M$="TXS":
    ATYPE=5
940 IF MO=6 AND OP=5 THEN M$="TSX":
    ATYPE=5
950 IF MO=7 AND OP=4 THEN M$="***"
960 IF MO=7 AND OP=5 THEN M$="LDX":
    ATYPE=10
```

```
 970 IF MO=7 AND OP=6 THEN M$="DEC"
 980 ENDPROC

 990 DEF PROCgthree(OP,MO)
1000 M$="***"
1010 ATYPE=0
1020 ENDPROC
1030 DEF PROCadd(ATYPE)
1040 IF ATYPE=1 THEN A=A+1:
     AD$="#"+FNhex(?A)
1050 IF ATYPE=2 THEN A=A+1:
     AD$=FNhex(?A+256*?(A+1)):A=A+1
1060 IF ATYPE=3 THEN A=A+1:
     AD$=FNhex(?A)
1070 IF ATYPE=4 THEN AD$="A"
1080 IF ATYPE=5 THEN AD$=""
1090 IF ATYPE=6 THEN A=A+1:
     AD$="("+FNhex(?A)+",X)"
1100 IF ATYPE=7 THEN A=A+1:
     AD$="("+FNhex(?A)+"),Y"
1110 IF ATYPE=8 THEN A=A+1:
     AD$=FNhex(?A)+",X"
1120 IF ATYPE=9 THEN A=A+1:
     AD$=FNhex(?A+256*?(A+1))+",X":A=A+1
1130 IF ATYPE=10 THEN A=A+1:
     AD$=FNhex(?A+256*?(A+1))+",Y":A=A+1
1140 IF ATYPE=11 THEN A=A+1:
     AD$=FNhex(FNrel(?A)+A+1)
1150 IF ATYPE=12 THEN A=A+1:
     AD$=FNhex(?A+256*?(A+1))+")":A=A+1
1160 IF ATYPE=13 THEN A=A+1:
     AD$=FNhex(?A)+",Y"
1170 A=A+1
1180 IF M$="***" THEN AD$="":ATYPE=0
1190 B$=FNhex(?(A-1))
1200 IF ATYPE=2 OR ATYPE=9 OR ATYPE=10
     OR ATYPE=12 THEN
     B$=FNhex(?(A-2))+" "+FNhex(?(A-1))
1210 IF ATYPE=4 OR ATYPE=5 OR ATYPE=0
     THEN B$=""
1220 ENDPROC

1230 DEF FNdec(A$)=EVAL("&"+A$)

1240 DEF FNrel(A)
```

```
1250 IF A<127 THEN =A ELSE =A-256
1260 DEF FNhex(A)
1270 LOCAL B,I,A$
1280 FOR I=1 TO 4
1290 B=A AND &F
1300 A=A DIV &10
1310 IF B<10 THEN A$=CHR$(B+48)+A$
ELSE A$=CHR$(B+55)+A$
1320 NEXT I
1330 IF MID$(A$,1,2)="00" THEN
     A$=RIGHT$(A$,2)
1340 =A$
```

The main program is easy enough to understand by reference to the procedure table (Table 11.1). The only other information that might help is that A contains the address of the memory location that is currently being examined for a possible op code.

Program structure and error handling

The disassembler listed above is a debugged first version. That is, although a few lines have been changed and even a few lines added during debugging the overall structure, the procedures and their internal operation are unchanged from the first attempt. After the extensive analysis of the form of the look-up tables given in the first part of this chapter it is not surprising that no major changes were necessary but this does not mean that the structure of the program is entirely up to standard. PROCinitialise could be made more compact by using a pair of two-dimension arrays for the mnemonic and address mode look-up table but this would make the rest of the program much more difficult to follow. The most unsatisfactory procedure in the whole program is PROCadd. An examination of the statements that follow the THENs shows immediately that there is much duplication of effort. It would be better to assemble the data items that most of the IF statements use before the IF statements are executed. For example, FNhex(?A) and FNhex(?A+256*?(A+1)) should all be worked out and stored in string variables at the start of the procedure. Also the variable that keeps track of the current location in memory, A, should not be incremented by each of the IF statements. This makes the task of printing the hex values stored in the memory locations alongside the disassembled instruction much more difficult than it need be (hence lines 1190 to 1210).

Even though there is plenty of scope for improvement definite proof that it is often better to rewrite first attempts at procedures the program

works and is a useful tool.

After using the program a few times the only fatal error that occurred was BAD HEX due to mistyping of hex numbers in PROCgetparams. This is easily cured by adding:

1 ON ERROR RUN

This is a crude but effective error catch-all.

Using a dissassembler

Using a disassembler is as easy as specifying the start and end address of the block of code that you want disassembled! Of course interpreting the output is much more difficult. Knowing where the machine code program starts is a great help but even then it is possible to run into an area of data and illegal op codes. Although illegal op codes are an almost sure sign that you have encountered a data area it is possible for a data area to contain legal op codes and so give the impression that you are still disassembling part of a program. In practice, data areas are identified both by containing the occasional illegal op code and by being referenced in address fields of other parts of the program.

The other main technique in understanding a disassembly is tracing the flow of control. Starting from the first instruction of the program it is possible to follow through each branch, JMP and JSR instruction and mark their destinations. In this way all of the possible paths through the program can be identified before the task of trying to understand what is happening in each part is begun. Going through and marking RTS instructions serves to identify candidates for the end points of subroutines. Similarly, the instructions that follow them are possible starting points of subroutines.

Even after using these hints you will still need a great deal of skill and ingenuity to decipher a disassembly. The one guiding principle is that you should always work out how you would write a program or subroutine to achieve the same result before you look at the output of a disassembler. If you have chosen the same method as the program being disassembled then you will quickly recognise the essential features of the method. If you haven't, then the disassembly will remain unfathomable and after a while you should stop studying it and try to think of another way of achieving the same ends. When you have identified the overall algorithm that is being used it is surprising how quickly the details fall into place.

Super disassemblers

The disassembler described in this chapter is far from the last word in sophistication on how to produce an easy-to-understand disassembly of a

program. The following are some suggestions that you might like to add to the program, none of them very difficult:

(1) Allow the user to specify the location of known data areas. In these data areas the disassembler should output the contents of memory as hex and, where posslble, ASCII characters.

(2) Allow the user to specify a list at known labels and their corresponding addresses. Each address that the disassembler produces should be checked against the list to see if a label has been defined. If it has then the label should be used in preference to the numeric address. For example, instead of JSR &FFF4 the disassembler would print JSR OSBYTE (assuming that OSBYTE=&FFF4 had been included in the list of label definitions).

(3) Mark all branch, JSR and JMP instructions automatically so that they can be found quickly. Also mark their corresponding destination addresses. In other words, mark all the exit and entry points within the disassembly.

It is possible to produce disassemblers that automatically identify program and data areas by tracing all of the possible paths for the flow of control through the program but this is a much more difficult problem.

Chapter Twelve
Bits, Binary and Boolean Logic

It may seem odd to leave a discussion of the sort of thing that is often included in introductory courses on computing to the very end of a book on advanced programming. However, although bits, binary numbers and Boo lean logic do form part of the foundations of computing, most of the efforts of computer scientists have been concentrated on hiding this fact! For example, high level languages such as BASIC were invented to avoid a face-to-face confrontation with the underlying principles of computing.

The BBC BASIC interpreter will accept commands to do arithmetic in terms of the familiar decimal notation, convert everything to a binary representation, work out the result using binary arithmetic and will then convert this back to decimal for printing! In fact, it is possible to use a computer without ever realising that it has anything to do with bits, binary or Boolean logic and for the most part this is how it should be. For the ordinary user a computer is a tool and its internal workings should be hidden. The question of how much of the internal workings a programmer should know is debatable. On the one hand most programs can be written without reference to 'low level' concepts such as bits. In this sense languages such as BASIC have been successful in making programming more accessible. However, there are still many reasons why BASIC and other high level languages are not 'powerful' in terms of speed, memory use, etc., and this occasionally forces the use of assembler and other 'low level' languages.

Assembler is closer to the underlying hardware and so it works in terms of bits and binary numbers and this is often the main reason that BASIC programmers find it difficult to adapt to assembly language programming. The point is that while it is easy to understand what assembly language instructions do, and even relate them to statements in BASIC, it is not easy to see how to use them the problem is that even if the operation is familiar, it is performed on primitive forms of data not normally found in BASIC. For example, the assembly language ADC (ADd with Carry) performs the familiar operation of addition, but on eight-bit binary numbers.

It is clear that if you want to use assembler effectively you must have at least a working knowledge of some of the ideas at the heart of

computing. What is less obvious is that the need for this knowledge also arises in BASIC. A BASIC program that interacts dtrectly wnh the hardware of a machine usually has to manipulate data at the level of binary numbers. Also, it is sometimes possible to save both memory and time by manipulating data directly in terms of its underlying binary representation and so avoid having to use assembler to achieve the same result.

All this suggests that once you have learned to use a high level language then the next thing to do is to learn about bits, binary and other technical topics before beginning the task of learning assembler! Bits, binary and logic may be fundamental to computing but they are definitely not the first things you should learn. If you follow this order, learning assembler will be easier because you will know about the sort of data and operations that make up an assembly language program but imagine how difficult learning BASIC would be if you had to also learn all about decimal numbers on the way! If you have already studied assembler then the contents of this chapter may help you to apply what you know.

If the idea of learning about such mathematical concepts is off-putting then it worth saying at this point that there is much in this chapter that is directly about programming! If you find any of the topics difficult at first reading, then just look them over and return to them when you actually need to know something about them.

Bit patterns

The raw material of computing is the *bit pattern*. All the other types of data described in Chapter Five are represented in the computer's memory using bit patterns. A bit pattern is nothing more complicated than its name suggests; that is, a pattern of zeros or ones. A memory location can store only a fixed size of bit pattern. In most of the current micros, including the BBC Micro, a single memory location always stores a bit pattern consisting of eight bits. A group of eight bits is such a common unit that it is given the name of *byte* probably the best known piece of all computer jargon. However, it is important to realise that this use of eight bits is entirely arbitrary and it will change as new generations of computers with greater memory capacities are produced.

A bit pattern can be used to represent all sorts of information. For example, suppose that you live in a house with eight rooms then you could record which rooms had lights on and which had them off by using one bit per room and a code such as (0=off and 1=on. In this way the bit pattern 00000011 would mean that all but two of the rooms had their lights off. It is important to realise that the information a bit pattern holds is entirely in the eye of the beholder. If, for example, we take 0 to mean 'light on' and 1 to mean 'light off' then the same bit pattern as above now records the fact that all the lights are *on* bar two! A bit pattern doesn't have a built-in meaning; it all depends what you use it for.

Simple binary

The most familiar use of a bit pattern is as a binary number. In some ways the familiarity of binary numbers is a problem in that it tends to obscure the more fundamental idea of a bit pattern. In fact, we are generally so familiar with numbers that it is often difficult to see how they work. Although the use of a bit pattern to represent a number is almost identical to the way that we use patterns of digits to represent numbers, if often causes a great deal of trouble for beginners. The best advice is that if you are in any doubt about how binary, or any other sort of number system, works then compare it with the way that decimal numbers work.

The standard decimal representation of a number uses a 'place value' system. For example, the number 123 means I 'lot' of one hundrgd, 2 'lots' of tens and 3 'lots' of I. In other words, the meaning of a digit depends on its 'place' within a number. Another way of expressing the meaning of a decimal number like 123 is:

$$1*100 + 2*10 + 3*1$$

and you can see that the value of the number is arrived at by multiplying each digit by a constant, or 'weight', that depends on its position. In the case of a decimal number the weights follow a regular pattern, starting with a weight of I for the right-hand digit and this weight increases by a factor of l0 for each position moved to the left. If the digit positions are numbered with the first position on the left as 0, the weight for position I is simply $10\wedge1$. (To see that this equation works you need to know that $10\wedge0$ is 1. Indeed, it is a strange fact that any value raised to the zero is 1.)

The most obvious way of using a bit pattern to represent a number is to modify the decimal place value system to take account of the fact that each bit has only two states (0 or 1) unlike the ten different states that a decimal digit can take (0 to 9). Apart from the binary weights being powers of two rather than powers of ten no other modifications are required. in other words the bit pattern l0 l0 interpreted as a binary number is:

$$1*8 + 0*4 + 1*2 + 0*1$$

Notice that only 0 and 1 can occur in a binary number and the weights are given by $2\wedge1$ where I is the bit position numbered starting from 0. If you work out the sum of each bit multiplied by its associated weight you will, of course, get the value that the big pattern represents in decimal. This simple observation is all that is needed to write a BBC BASIC function that will convert binary to decimal:

```
 10 INPUT B$
 20 PRINT FNbin_to_dec(B$)
 30 GOTO l0

9000 DEF FNbin_to_dec(B$)
```

```
9010 LOCAL I,D
9020 D=0
9030 FOR I=1 TO LEN(B$)
9040 D=D+EVAL(MID$(B$,I,1))*2^(I-1)
9050 NEXT I
9060 =D
```

You can use this function to convert a string of zeros and ones to decimal but notice that it doesn't check to see that no other characters are used in the string and that it isn't the fastest method of converting to decimal.

Once you understand the way that the binary place value system works, binary numbers are easy. For example, the smallest number that can be stored in a single eight-bit memory location is 00000000, or just 0 in decimal, and the largest is 11111111, or 255 in decimal. If you need to use a larger numeric range then you can extend the number of bits available by using more than one memory location. For example, using the bit pattern stored in two memory locations provides sixteen bits to represent a number and this gives a range of 0 to 65535. One of the memory locations has to be used to store the eight bits associated with the smaller place values i.e. bit 0 to bit 7 this is called the 'least significant' byte and the other is used to store the bits associated with the larger place values i.e. bit 8 to bit 15 and this is called the 'most significant' byte. The only trouble is that the byte indirection operator 'T only handles single memory locations. That is:

    ?A=100

will first convert the decimal number 100 to an eight-bit binary representation and then store the bit pattern in the memory location whose address is in 'A'. You cannot use'?' to store a vaine that needs sixteen bits to represent it directly. For example:

    ?A=3000

will not convert 3000 to a 16-bit binary representation and store it in the two memory locations indicated by 'A'. To store a 16-bit value it is necessary to separate out the most and least significant bytes. This is most easily done using the operators MOD and DIV:

    Most significant byte = value DIV 256
    Least significant byte = value MOD 256

Putting the two bytes back together is just as easy. If A contains the least significant byte and A+1 the most significant then:

$$value=?A+256*(A?1)$$

The reason why these equations succeed in splitting and recombining 16-bit numbers will be clear after the section on bit manipulation.

Negative numbers or two's compliment binary

With simple binary it is only possible to use bit patterns to represent postive integers. It is important to realise at this stage that there are a number of different ways of making a bit pattern represent negative numbers. For example, we could use the standard decimal method of a special symbol to indicate that a number is negative - +6 is a positive number and -6 is a negative number. This is known as *sign magnitude* representation because each number is composed of a sign that indicates whether it is positive or negative and a number that indicates its size or 'magnitude'. The sign magnitude method can be used with bit patterns to represent both positive and negative numbers. In binary the sign magnitude method is just as easy. The sign of a number is indicated by one of the bits, usually with 0 standing for positive and 1 for negative and the remaining bits represent the magnitude using simple binary. For example, an eight-bit sign magnitude number would conventionally use bit 7, the most significant bit, to indicate the sign of the number so:

00000111

would be +7, because bit 7 is 0 for a positive number and the remaining bits represent 7 in simple binary. On the other hand:

10000111

would be -7 because bit 7 is 1 for a negative number and the remaining bits still represent 7 in simple binary.

Although sign magnitude representation is simple to understand it is not easy to do arithmetic with. For this reason most computers use a different method of representing positive and negative numbers known as *two's complement* representation. This is the method used by the 6502 and by BBC BASIC so it is worth explaining in detail.

It is a fundamental property of a negative number that when added to a positive number of the same magnitude the answer is zero. For example 7+(-7) is 0. In fact this is more than a fundamental property, it is the definition of a negative number! That is, the negative of X is a number Y such that:

X+Y=0

if you can find any number Y that satisfies this equation then it is the negative of X. This looks like an impossible task until you examine the way that binary arithmetic is performed on computers. Unlike paper and pencil arithmetic computer arithmetic is always carried out to a fixed number of bits. For example, using 8-bit numbers and 8-bit arithmetic the result of adding 1 to 255 will be:

```
         1 1 1 1 1 1 1 1
         0 0 0 0 0 0 0 1
         ─────────────────
      1   0 0 0 0 0 0 0 0
```

or 256, but 256 needs nine bits to represent it and the computer only keeps the first eight. Thus, in computer arithmetic 255 + 1 is 0. As 1 + 255 equals 0 you should be able to see that 255 can be regarded as a representation of -1. To be precise, it is the two's complement representation of -1. Starting from 0 it is possible to find the representation of the negative of each value. That is the negative of 0 is 0, the negative of 1 is 255, the negative of 2 is 254 and so on. Notice that apart from zero the negative of X is simply:

256 - X

Also notice that whereas an 8-bit simple binary number could represent values in the range 0 to 255, an 8-bit two's complement number can only represent positive numbers from 0 to 127. This is because 128 to 255 are used to represent negative numbers in the range -128 to -1. In binary the fact that 0 to 127 represent positive numbers and 128 to 255 represents negative numbers means that it is possible to use bit 7 to test if a number is negative or positive.

In the same way, 16-bit arithmetic simply ignores any 17th bit that is generated as part of the result Thus in 16-bit arithmetic 1+65535 equals 0 and by the same reasoning 65535 can be taken to be the 16-bit two's complement representation of -1. At first sight it may seem confusing that the two's complement representation of -1 depends on the number of bits that are used but it is not so difficult if you recall that the largest number that can be stored before returning to zero depends on the number of bits. In the case of 16-bit two's complement representation the range that can be represented is —32768 to +32767 and the negative of X is given by:

65536 - X

The great advantage of two's complement numbers from the point of view of the assembly language programmer is that they can be added together without having to worry about whether they are positive or negative. This is not the case with sign magnitude representation. If you want to add 6 and -3 you actually have to notice that the second number is negative and do a subtraction i.e. 6 - (+3). With two's complement this is unnecessary. The same sum in 8-bit two's complement, for example, is 6 + 253 (253 is the 8-bit two's complement representation of 3) which gives 259 or, after ignoring all but the first eight bits, the correct answer, 3. In the same way two's complement numbers can be subtracted without having to take any notice of the sign of any of the operands.

BBC integers

Rather than explore the implications of 8-bit and 16-bit two's complement numbers, it is probably more useful to concentrate on the format used by BBC BASIC to represent integers. That is 32-bit two's complement. Obviously a 32-bit number takes four memory locations to store. The bytes are stored starting with the least signficant byte so the arrangement in memory is:

byte	A+3	A+2	A+1	A
	b31 to b24	b23 to b16	b15 to b8	b7 to b0

where A is the address of the first byte. The range of values that can be represented is a staggering:

$$-2,147,483,648 \text{ to } 2,147,483,647$$

and to find the two's complement representation of X use:

$$4294967296 - X$$

However, the word indirection operator '!' will automatically convert decimal numbers into the correct two's complement representation and vice versa. For example, !A = -2147483648 will store a bit pattern consisting of all ones in the four memory locations starting at A.

Fractions - floating point binary

After extending the use of bit patterns to represent negative numbers, the next obvious step is to find a way of representing fractions. Just as in the case of negative numbers there are many ways of achieving this. The simplest method is just to work as if there was a 'binary point' in the bit pattern. The binary point works in a way analogous to the familiar decimal point. Bits to the left of a binary point have place values that increase by a factor of two, but bits to the right of a binary point have place values that decrease by a factor of two. For example, the bit pattern:

weight	8	4	2	1		1/2	1/4	1/8	1/16
	0	1	1	0	.	1	1	0	0

represents 6.75 using eight bits with the binary point fixed between bit 3 and bit 4. This representation is known as *fixed point binary* and used to be very popular but it has fallen from favour because of its limited numerical range. However, it is often useful in assembly language programs where a limited amount of arithmetic is required.

The most common way of representing fractional numbers and the method that the BBC Micro uses is called *floating point binary*. This is the binary equivalent of the familiar decimal exponential notation. In decimal a number can be written as:

mantissa * 10^exponent

This form of representation is used by BBC BASIC when the number is too large or too small to print conveniently on the screen. If you try:

PRINT 2^32

you will see the result 4.2949673E9 which means 4.294967*10^9 (the E stands for 'Exponent'). You can also enter numbers in this format in response to an INPUT staternent.

A floating point binary representation has the same overall format as a decimal exponential representation. In the case of BBC BASIC a single byte is used to store the exponent and four bytes are used for the mantissa:

byte	A	A+1	A+2	A+3	A+4
	exponent		four byte mantissa		
		high	$\rightarrow$	low byte	
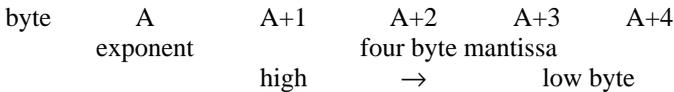
where A is the address of the first byte. The exponent part of the representation has to allow for the possibility that the exponent might be negative. For this reason an 'excess 128' representation is used. This is yet another way to use a bit pattern to represent both positive and negative integers. As you might guess, an excess 128 representation means that 128 is added to the true exponent before it is stored. Thus 129 would represent an exponent of +1 and 127 would represent an exponent of -1. The value of the number stored is given by multiplying the value of the mantissa by 2^(E-128) where E is the contents of the first memory location. Notice that it is 2 raised to the power of the exponent and not 10! The value stored in the mantissa is represented in rather an odd way. Firstly, it is always assumed that the binary point is just before the first bit and the value that the mantissa represents is more than .5 and less than 1. You can always make sure that the value of the mantissa is more than .5 and less than 1 by altering the value of the exponent. With these two conditions in mind the exponent is represented as a simple binary fraction. However, as this fraction always lies between .5 and less than I you should be able to see that the bit that comes after the binary point is always a I. (The place value of this bit is .5). Always setting a bit to the same value is a waste of storage and so it is used instead as a sign bit. Thus, the mantissa is a modified sign magnitude fixed point representation! It is important to notice that the mantissa is *not* stored as a two's complement value in the way that an integer is.

This description of floating point is a little complicated so it is worth giving a few examples. The bit pattern:

byte	0	1	2	3	4
	sign bit				
	$\downarrow$				
	10000001	00000000	00000000	00000000	00000000
decimal	129	0	0	0	0
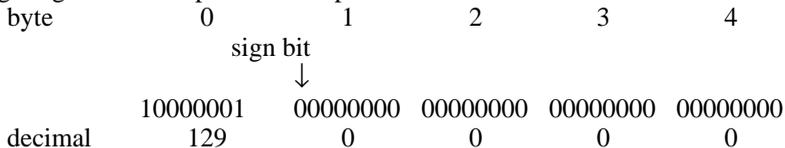
represents an exponent of 1 (=129-128) and positive mantissa of value .5. (The sign bit, that is bit 7 of byte 1, is 0 so the number is positive and although all the other bits of the mantissa are also 0, the first bit after the binary point is always 1). Thus the value represented is $.5*2^1=1$. The bit pattern:

byte	0	1	2	3	4

                      sign bit
                         ↓
         10000011   11100000   00000000   00000000   00000000
decimal   131   224   0   0   0

represents an exponent of 3 (= 131 - 128) and a mantissa of -.875 (-=.5+ .25+.125) which gives a value of -7 (=.875*2^3).

If you know the floating point representation it is possible to' 'pick up' values stored in BASIC variables from assembler but any sort of floating point arithmetic is far better handled in BASIC.

Binary Coded Decimal

To close this discussion of how bit patterns can be used to represent numeric values it is interesting to give an example of a completely different method. Binary Coded Decimal (or BCD) representation is a sort of cross between a decimal representation and simple binary. The principle is that instead of converting the entire decimal number to binary each digit is converted separately. As the largest decimal digit is 9, or 100! in binary only four bits are needed per digit. For example the number 1234 would be represented in BCD as:

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 0001 & 0010 & 0011 & 0100 \end{array}$$

Using four bits per digit means that it is possible to store two digits per memory location and this ghes a range of 0 to 99 which should be compared to the 0 to 255 that can be achieved using simple binary. The main advantage of BCD is that conversion to binary is easy and can be done as digits are entered from a keyboard, say. BCD arithmetic is also an exact parallel of what happens in decimal and this can be a advantage for some applications where accuracy is essential.

The BCD representation is particularly useful in assembler and to this end the 6502 can work in a 'decimal mode'. The instruction SED will SEt Decimal mode and CLD will CLear Decimal mode, i.e. return the 6502 to normal binary arithmetic. In decimal mode two bytes are added together using ADC or subtracted using SBC as if they each represented a pair of decimal digits in BC D. For example,

```
SED
CLC
LDA #20
ADC #86
```

will produce the result 112 in the A register. That is the bit patterns that represents 20 and 86 are:

20 = 0001 0100
86 = 0101 0110

which corresponds to 14 and 56 in BCD. Now 14 + 56 =70 and this is represented in BCD by the bit pattern:

0111 0000

which if treated as a simple binary number and converted to decimal is 112.

Notice that during this example the bit pattern was regarded as representing different things. For the purposes of the assembler the bit pattern was expressed in terms of a decimal number but for the purposes of BCD arithmetic it was a pair of digits. This is quite proper and is typical of the sort of 'mental gymnastics' that an advanced assembly language programmer often goes through in thinking about data. A bit pattern can be interpreted and hence written down in many ways. For example the bit pattern :

10000111

is 135 in decimal if it is interpreted as simple binary; it is 7 as a sign magnitude number; it is 121 as a two's complement number; it is 87 in BCD and so on ... The important point is that the reality is the underlying bit pattern what it represents depends on the way we interpret it and the operations that we subject it to.

Hexadecimal

BBC BASIC can work with both decimal numbers and hexadecimal numbers. The main advantage of hex is that it is very easy to convert to binary. For this reason it is a particularly simple way of specifying a bit pattern. Hex numbers use a place value representation with weights that are powers of l6. This use of weights larger than l0 means that it is necessary to invent some extra symbols to stand for values between 10 and IS, As explained in the BBC User Guide the conventional choice for these new symbols is A to F. Thus counting in hex goes ()to 9 as usual and then A, B, C, D, E and F. After F (15) the place value system comes into play and the next number is 10 standing for

1*l6 + 0*1

or 16 in decimal. The conversion of hex numbers to decimal follows the method used to convert binary to decimal that is each digit is multiplied by the appropriate weight and the results are totalled. The only

complication is the need to work out unfamiliar expressions such as E*16 which are tackled by converting the extra symbols A to F to decimal before trying to perform the multiplication i.e. E*16 is simply 15*16.

However, as already mentioned, the importance of hex numbers is that they are remarkably easy to convert to binary and vice versa. The reason for this is that each symbol that makes up a hex number can be converted to binary without reference to the rest of the number. For example, to convert :

F7C3

all that you have to do is to write down the binary equivalent, of each symbol, that is:

F	7	C	3
1111	0111	1100	0011

giving 1111011111000011 as the answer. Note that you have to be careful to remember to write down four bits for each symbol (because the range 0 to 15, that is 0000 to 1111, takes four bits to represent). To convert a binary number to hex, form groups of four bits starting from the right (adding zeros to the final group if it has less than four bits). Then each group of four bits can be separately converted to hex. For example:

10110110111

can be converted to hex by first forming groups of four bits:

0101	1011	0111

(Notice the extra zero added to the group on the far left.) Then each group is converted to a single hex character:

5	B	7
0101	1011	0111

giving 5B7.

This method of converting hex to binary and binary to hex is so simple that it is always to be preferred as a way of writing down a bit pattern. As a single hex character corresponds to four bits and vice versa it only needs a pair of hex characters to represent the contents of a single memory location. For example, the most common explicit use of bit patterns in BBC BASIC is in user-defined characters. A row of eight dots can be represented by a pattern of eight-bits and this pattern is usually written in a VDU 23 statement as a pair of hex digits.

It is well known that in BBC BASIC a hex number is distinguished by writing '&' in front. A feature that is easy to miss from even a careful reading of the User Guide is that "~" can be used to force a value to be printed in hex. The easiest way to input hex numbers is to use the EVAL function, as in:

```
    INPUT N$
    N=EVAL("&"+N$)
```

Unfortunately "~" cannot be used to convert decimal numbers to hex strings as it can only be used in PRINT statements. However, a function to convert decimal to hex can be found in the disassembler in Chapter Eleven.

Bit patterns and logic

The use of bit patterns to represent numbers is in many ways a very sophisticated idea that might be better introduced after the rather simpler topic of using bit patterns in Boolean logic. Boolean logic may be simple but it is much less familiar than arithmetic and for this reason it has been left until a later stage.

Boolean logic involves the manipulation of two values usually called *true* and *false*, although in practice they can be any two distinct values. Obviously it is possible to make a bit pattern represent a number of Boolean values as each bit can be interpreted as meaning true or false. Thus the bit pattern 0 l0! could be taken to mean 'false, true, false, true' if we assume that 0 is representing false and 1 is representing true. The fundamental operations of Boo lean logic, AND, OR and NOT, are fairly well known even to non-programmers as they are very similar to their everyday English equivalents. That is x AND y, where x and y are Boolean values (or truth values) is TRUE if, and only if, x and y are both TRUE; x OR y is TRUE if either x or y are true; and NOT x is TRUE if x is FALSE and vice versa. A good way of explaining the action of a logical operator is by a truth table that lists the result of the operation for each possible value of x and y. For example AND may be characterised by:

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

You may have noticed that while the idea of using a bit pattern to represent a number of logical values was introduced at the start of the section the logical operators have been introduced in terms of their effect on a pair of single bits. That is, while the result of I AND 0 can be easily worked out to be 0, so far no meaning has been attached to expressions such as 01 l0 AND 1010. There are in fact a number of ways that this can be done but the most common is the so-called bitwise extension to the standard logical operations. A bitwise operation manipulates pairs of bits that occupy the same positions within their respective bit patterns. For example, the bitwise evaluation of 0110 AND 1010 first calculates bit zero of the result by ANDing together the bit zeros of each bit pattern -

that is 0 AND 0. Then the second bit of th result is calculated in an analogous manner using the second bit of each bit pattern and so on. This is best visualised as:

$$
\begin{array}{r}
0110 \\
\text{AND} \quad 1010 \\
\hline
0010
\end{array}
$$

Bitwise versions of OR and NOT can be similarly arrived at. For example, the bitwise implementation of 0110 OR 1010 is:

$$
\begin{array}{r}
0110 \\
\text{OR} \quad 1010 \\
\hline
1110
\end{array}
$$

and NOT 0110 is simply obtained by 'inverting' each of the bits in turn giving 1001 as the result.

Bitwise logical operations are particularly useful for carrying out 'bit manipulation', that is independently changing some bits within a bit pattern without altering others. For this reason it is the most common method used to extend the logical operators to bit patterns, but it is important to realise that it is by no means universal - see for example ZX BASIC as implemented on the ZX81 or Spectrum.

BASIC logic

In BBC BASIC TRUE and FALSE are represented by integers, -1 for TRUE and 0 for FALSE. The logical operators are AND, OR, NOT and EOR (EOR stands for Exclusive OR and will be described later). Ifyou use these operators on the values 0 and —I then they behave exactly as you would expect. For example -1 AND 1 is -1 and NOT 0 is -1. However, if you try using them with other integer values you will find that, rather than printing an error message, the BBC Micro will return another integer as a result. For example, if you try PRINT 16 OR 3 you will see the result 19 printed. From the last section you should be able to see that BBC BASIC is in fact implementing its logical operators in a bitwise fashion. In fact the logical operators are applied bit wise to the four bytes that are used to represent an integer. That is, the result of l6 OR 3 is:

OR	00000000	00000000	00000000	00010000
	00000000	00000000	00000000	00000011
	00000000	00000000	00006000	00010011

which is, of course. the binary representation of 19.

The only complication to this simple bitwise implementation is that an integer is represented using two's complement For example, what is -1 OR

7? The answer is easy as long as you remember or work out the bit pattern used to represent -1 (using two's complement):

-1	=	11111111	11111111	1111111	11111111
7	=	00000000	00000000	00000000	00000111

| -1 OR 7 | = | 11111111 | 11111111 | 11111111 | 11111111 |

That is -1 OR 7 is simply -1. If you find all of this conversion to two's complement difficult or confusing then the best thing to do is to use hex. For example PRINT ~(&E3 OR &3F) gives the result &FF.

Finally, it is worth pointing out that while TRUE and FALSE are -1 and 0 respectively, the IF statement will treat any non-zero value as TRUE. That is, the IF statement is more accurately described as:

IF numeric expression THEN list 1 ELSE list 2

and 'list 1' will be executed if 'numeric expression' is non-zero and 'list 2' will be executed if 'numeric expression' is zero.

Logic in assembler - EOR

The 6502 performs all of its logical operations as bitwise instructions working on eight-bits at a time. Of course, you can manipulate larger bit patterns by repeating the operations on eight-bits at a time. For example:

AND #&40

will perform a bitwise AND of the current contents of the A register and &40 leaving the result in the A register. The logical operations included in the 6502% instruction set are AND, OR and EOR. Y ou might be puzzled by the absence of a NOT command and the presence of EOR. In practice, there are a number of ways of producing the same effect as a NOT command and one of them is to use EOR. EOR stands for Exclusive OR and its action is best explained by way of a truth table:

x	y	x EOR y
0	0	0
0	1	1
1	0	1
1	1	0

If you examine this table you will see that x EOR y is true if x or y is true, but not *both*. This corresponds more closely to the English use of 'or' than OR. For example, if you say 'you can have marmalade or jam' you generally mean that the choice is one or the other but not both! EOR can be used to produce the same effect as a NOT instruction. If you want to perform a bitwise NOT on the contents of the A register then use:

EOR #&FF

This works because each bit in the A register is EORed with a '1' bit in the bit pattern and if you look at the truth table you will see that 0 EOR 1 is 1 and 1 EOR 1 is 0 which is exactly what is needed to implement a NOT operation.

Bit manipulation

The need to manipulate and test the value of individual bits within bit patterns is fundamental to many areas of programming, particularly where any direct interaction with the hardware is involved. Using the bitwise logical operators it is easy to set any bit within a bit pattern to zero or one or to change its existing value from 0 to 1 or vice versa.

In particular, you can set any bit or group of bits to zero by ANDingthe bit pattern with another specially constructed bit pattern called a *mask*. For example, suppose you want to set the first three bits in a bit pattern to zero then you would AND it with a mask consisting of all ones apart from the first three bits. To see how this works examine the following:

        value   = 10101010
        mask   = 11110000
        result =1010 1000

Notice that the last three bits will always be zero no matter what the last three bits in 'value' are (because 0 or 1 when ANDed with a 0producesa 0). ln the same way OR can be used to set any bit or group of bits to I and EOR can be used to NOT or 'flip' any bit or group of bits.

To be precise:

(1) *To set any group of bits to 1* construct a mask value consisting of zeros in every bit position apart from the bit positions that you want to set to 1. This mask is then ORed with the bit pattern that you want to alter.

(2) *To set any group of bits to 0* construct a mask value consisting of ones in every bit position apart from the bit positions that you want to set to 0. This mask is then ANDed with the bit pattern that you want to alter.

(3) *To 'flip' any group from their current value* construct a mask consisting of zeros in every bit position apart from the bit positions that you want to alter. This mask is then EORed with the bit pattern that you want to alter.

For example, if you want to 'flip' the first three bits of an eight-bit value, the mask you would use would be:

    00000111

or &07 in hex. If you EOR &07 with any value you will discover that as

required the first three bits are unchanged, but the rest of the value is unchanged. For example, &55 EOR &07 gives the result &52.

Extracting information - USR and shifts

Setting groups of bits to zero without affecting the rest of the bit pattern is often used in conjunction with *shift* operations to extract information from a bit pattern. A shift operation, as its name suggests, will shift all the bits in a bit pattern one place to the right or the left. For example a right shift of the bit pattern:

10011011

gives

01001100

Notice that as well as shifting all the bits one place to the right we have also 'produced' a zero that has been shifted into bit 7's position and the old value of bit 0 has been 'lost'. Apart from the direction of shift i.e. left or right shift operations can also differ in the value that they shift 'into' the bit pattern and what happens to the bit that is shifted out. Shifting a zero in and losing the bit that is shifted out is called a *logical shift*.

Using a logical shift and the logical operators it is possible to isolate any group of bits within a bit pattern. For example, a JSR function returns a value that is best considered as a bit pattern made up in the following way:

P |Y |X |A

where P is the contents of the status register, Y is the contents of the Y register, X the contents of the X register and A the contents of the A register. Notice that each register value is composed of eight-bit and hence the [JSR function returns a four byte bit pattern that is interrupted by BBC BASIC as an integer. Suppose that the particular [JSR function that you have written returns its result in the X register and the values in the P, Y and A register are irrelevant. To isolate the byte that comes from the X register from the other three bytes, all that you have to do is to set all of the unwanted bits to zero and then perform eight logical right shifts. That is, first AND the value with:

&0000FF00

and then perform the eight logical shift rights. For example, if the value returned by USR is &B1123486 then the result of ANDing it with the above mask is &00003400 and the result of eight right shifts is & 00000034 or just &34.

This method can be used directly in 6502 assembler because the 6502

has both logical operations and logical shifts. However, BBC BASIC doesn't have a shift operation and so it looks as though this method cannot be used. Fortunately this is not the case as there is a correspondence between right shift operations and division by two and left shift operations and multiplication by two. Dividing a number by two and ignoring any fractional part has the same effect as shifting the bit pattern one place to the right. In general BASIC this can be implemented as INT(V/2) where V holds the bit pattern to be shifted to the right. In BBC BASIC a better and faster way is to use the DIV operator which performs integer division and so automatically ignores any fractional part. Thus, to isolate the X register's value in BBC BASIC use:

(V AND &0000FF00) DIV 256

(Dividing by 256 is the same as dividing by 2 eight times, i.e. 256=2^8, and so performs eight right shifts as required.)

Using shifts (implemented as divisions and multiplications in BASIC) and logical operations it is possible to isolate, test and generally alter any bit or group of bits within a bit pattern and this opens up all sorts ofmemorysaving methods of 'packing' data into the smallest possible space. Bit manipulation is a common tool in advanced programming.

Logic functions - predicates

For the final section in the final chapter of this book we return to the subject of program clarity. Sometimes it is necessary to write a very simple IF statement that is complicated by the difficulty of the 'condition' that it tests for. For example, suppose you had written a program that counted the number of letters (as opposed to numbers, spaces and punctuation characters) a string contained. The IF statement that actually does the counting is simple enough in conception:

IF C$ = a letter THEN COUNT COUNT+1

where C$ contains the character being tested. The trouble is that the condition C$ a letter is not BASIC and turning it into valid BASIC obscures its meaning:

IF (C$>="A" AND C$<="Z") OR (C$>="a" AND C$<="z") THEN COUNT=COUNT+1

The first bracket tests to see if C$ is in the set of upper-case characters and the second tests to see if it is in the set of lower-case characters. Obviously if either of these tests is true then CS is a letter and hence they are connected together using OR.

It is not possible to simplify this 'letter test' condition and so you might come to the conclusion that there is nothing that can be done to improve

the clarity of the program - this is not so. A simple idea from logic can be applied with great success in BBC BASIC - the predicate function. A predicate function is, roughly speaking, a function that returns a Boolean value. As BBC BASIC represents TRUE and FALSE by integers there is nothing stopping us from writing functions that return Boolean values. For example the 'letter test' condition can be easily turned into a predicate function :

```
1000 DEF FNletter(C$)
1010 LOCAL T%
1020 T%=C$>="A" AND C$<="Z")
1030 T%=T% OR (C$>="a" AND C$<="Z"
1040 =T%
```

Notice the way that within the function the complex condition can be worked out in stages so that it can be understood in the same way this is always a good idea. Once this function has been defined the IF statement can now be written:

```
IF FNletter(C$) THEN COUNT=COUNT+l
```

or as

```
IF FNletter(C$)=TRUE THEN COUNT=COUNT+1
```

depending on which you find the clearest expression of your intentions.

Theory

If you really want to be confident about computing, there is a lot more theory to learn about binary numbers, Boolean logic, etc. Most of it is quite straightforward as long as you keep askingyourselfthe question 'what does this theory really mean in terms of bit patterns and operations'. The best way of ensuring this practical outlook is to learn theory as and when you need it in your programming. A great deal of this book has been about increasing your awareness of the possibilities rather than getting involved in a great deal of detail. In computing, finding out what you don't know and what you need to know is a substantial part of the battle!

Further Reading

Even though this is a long book it still does not include everything you might want to know about BASIC programming, 6502 assembler and the BBC Micro and while using it you may occasionally need to refer elsewhere for more information.

An introductory book on BBC/Acorn BASIC that uses the approach of natural structure to explain the use of each command is *The Electron Programmer* by S. M.Gee and Mike James (Granada, 1983). If you need a more advanced but general guide to programming then try *The Complete Progranuner* by Mike James (Granada, 1983). If you are programming in assembly language then look out for *Introducing BBC Micro Machine Code* by A. P. Stephenson (Granada, 1983). A comprehensive, if rather long, book on 6502 assembly language programming is: *Beyond BASIC* by Richard Freeman (BBC and the National Extension College Trust, 1983.)

For specialised information on ROM paging see *BBC Micro ROM Paging Systems Explained* from Watford Electronics, 35/37 Cardiffitoad, Watford, Herts. They can also supply the necessary hardware required.

You will find the hardware and many details of the inner workings of BBC BASIC in *The BBC Micro: An Expert Guide* by Mike James (Granada, 1983).

If you want to write applications programs that use the BBC Micro's graphics then we recommend both *BBC Micro Sound and Graphics* by Steve Money (Granada, 1983) and *The Complete Graphics Programmer* by Mike James (Granada, 1984) which is a more advanced book about graphics techniques.

Finally, if you would like more examples of how to use the programming methods described in this book, this time applied to games programs, you are referred to: *The BBC Micro Gamesmaster* by Kay Ewbank, Mike James and S. M. Gee (Granada, 1984) whicheontains not only the final program listings but a description of how each program was created and its problems overcome.

Index