GLENTOP

MALLARD BASIC on the AMSTRAD PCW8256/512

PROGRAM YOUR PCW!

Ian Sinclair

SECOND EDITION

PROGRAM YOUR PCW!

PROGRAM YOUR PCW

by

Ian Sinclair

Glentop Press Ltd

FEBRUARY 1987

ISBN 1 85181 004 8

First impression October 1986

Second impression November 1986

Second edition revised February 1987

Reprinted April 1987
Reprinted November 1987
Reprinted February 1988
Reprinted November 1988

Published by: Glentop Press Ltd
Standfast House
Bath Place
High Street
Barnet
Herts EN5 5XE
Tel: 01–441–4130

Originated directly from the publisher's w-p discs by
NWL Editorial Services, Langport, Somerset, TA10 9DG

Contents

Preface

The Amstrad PCW machines offer much more for both the home and business user than any previous machine at a comparable price, more even than many machines at much higher prices. Because of this, many buyers of the machine will never have made use of a computer before, certainly not a computer with the range of facilities that the PCW machines offer. You may have bought the machine only for its word processing abilities, and been delighted to find that so much more was possible. The manuals that come with the machine are among the best that have ever been produced for a small computer, but it is an impossible task to cater for every need when you are describing word processing, the use of CP/M, BASIC and LOGO in just two manuals. In particular, it's one thing to learn to operate a computer with programs that you buy, another to learn how to program it for yourself, and yet another to learn how to design your own programs. This book is aimed to offer just that type of progressive help on the latter two topics to the novice buyer who wants to take command of the PCW machine and write programs for him or herself.

There will invariably be applications for your Amstrad PCW machine that cannot be covered by programs that you buy. When your computing has moved to the stage where you need to design your *own* programs to solve your *own* problems, then you will need to learn how to program the Amstrad PCW machines for yourself. Programming, after all, is one of the features for which a computer is intended. Owning a computer and not programming it for yourself is like buying a Ferrari and getting someone else to drive it – while you pay for the petrol. If you have never programmed a computer for yourself, this book will show you how. If you have some experience of earlier makes of computers, then this book will open a new world of business-oriented programming to you. The Amstrad PCW machines do not use the same simple version of the BASIC programming language as earlier designs of computers, and this Mallard BASIC needs to be learned almost from scratch if you have previously used one of these machines. Even the BASIC programming language of the older Amstrad machines, such as the CPC464, is not an entirely adequate foundation for the Mallard BASIC of the PCW machines, though knowing

the older BASIC would allow you to skip several sections of this book. Mallard BASIC is very much aimed at the user who has serious business applications in mind, and has none of the sound, graphics and colour commands that take up a lot of space in the BASICs of the older machines. In addition, it has really excellent provision for the filing of data on disc, something that many versions of BASIC treat very poorly.

I would like to emphasise that this book was written while I was *using* an Amstrad PCW 8256, and that the listings of programs in this book were obtained from the Amstrad printer of the machine. This might seem to be an unnecessary claim, but many books still appear in which the program listings have been retyped, sometimes on another machine, with errors appearing in many of them. Every program which appears in this book, and every example of programming commands, has been tested on the PCW 8256 which I have here in front of me. Nothing has been copied untested from the manual, and where a command has operated in a way that is not obvious from the manual description, I have pointed out the difference.

As always, I am greatly indebted to many people who made this book possible. The machine was loaned to me by Glentop Publishing, and I am most grateful to all of the team there, particularly to Dr. Peter Holmes, who commissioned the book, and to Andy Savery who worked wonders with my discs and answered innumerable questions. I am also grateful to David Foster who prompted many corrections to the text and improved several examples. I am particularly grateful to Howard Fisher, of *Locomotive Software*, who offered to check the manuscript. I am sure that the result of all this work is a book that will match the capabilities of your PCW machine. In my own opinion, this is one of the best home computers that I have used to date, and I would like to congratulate Amstrad on a remarkable achievement in the computer marketplace, one that has restored the faith of users of small computers. No computer, however, is useful without software, and the Mallard BASIC from *Locomotive Software* has been a joy to use at all times. Its excellence accounts for the rapid growth and the soaring reputation of this software house on both sides of the Atlantic.

Ian Sinclair
Summer 1986

Chapter 1

Setting up

By the time you read this, you should already have connected the bits of the computer system together, unless you bought this book before you bought the computer. The advice in the manual about setting up the machine applies as much to the use of the machine as a computer as to its use for word processing. One hint might be useful, however. When you program the computer for yourself, you will need space, as much of it as you can get, and you might find it handy to place the computer system on a purpose-built stand, such as the type made by Selmor and used in many schools (figure 1.1). This keeps the computer off the desk or table, and allows you to wheel it around the room as you please. I'm going to assume that you already have the machine, either a PCW 8256 or PCW 8512, and that you have already used it, probably for word processing with the built-in *LocoScript*. You should by this time, then, have made copies of the two master discs that came with the machine. This action of making 'backups' is *very* important, because with all the care in the world accidents can happen. Some day you may leave a disc in the machine when you switch on, switch off, or reset. You might not notice at the time, but eventually you'll discover that one program no longer works properly, it has been 'corrupted'. If you are using copies of your master discs, then you can take another copy of the affected program, and keep on working. If the corruption is in your master discs, then you really will have problems, because these discs are not easy to replace, and can be costly. Later in this chapter we'll look at how a special master disc copy can be created for use with BASIC, and in appendix A you can find out how to make this a self-starting disc that need only be placed in the drive, after switching on or resetting, to get you into BASIC programming.

I shall also have to assume that you can make use of some of the important CP/M operating system actions, like DIR and DISCKIT. To prepare a disc for BASIC, you will need to be able to use PIP also, but I shall describe the action in detail when we come to it. The important point at present is to have copies of the master discs, and to have the machine housed where you can sit at the keyboard comfortably, and make lots of notes on paper. If you thought that using a computer would cut down on the amount of paper that you use, think again!

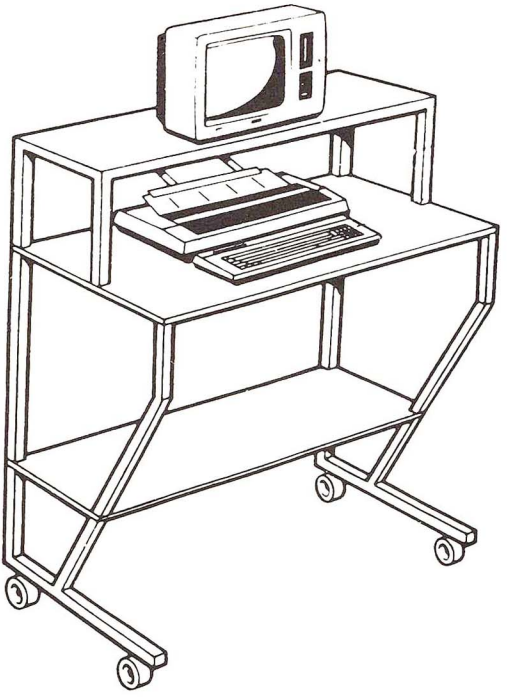


Figure 1.1 Using a *Selmor* stand to house all the bits and pieces of a PCW computer system.

Now the reasonable question to ask at this point is – why should you program the computer for yourself when for a few hundred pounds you could have as many professionally-written programs as you might need? The figure of a few hundred pounds is one good reason in itself if you are a home user, though for business purposes this might be less important. The one single overwhelming answer to this question is that only by programming for yourself do you get exactly what *you* want. Suppose, for example, you run a small mail-order catalogue group, and you want to keep tabs on who has ordered what, how much has been paid, and when delivery is made. Now there isn't a commercial program to do this, as far as I know. You could buy a set of business programs such as spreadsheets, databases and accounting packages, but you'll be overwhelmed by them. They all need a lot of learning to use, and they all do much, much, more than you need. A business accounts program, for example, keeps several ledgers going, has entries for aged debtors and all sorts of other accountancy specialties, and you spend most of your time trying to work out which bits are needed. Since you can't dispense with any of them, though, you have to try to use them all, just as if you were running a medium-scale business. By spending just a fraction of that time and a lot less money on learning to program for yourself, you could have your own program, tailored to your own needs, doing what you need of it. Remember that when you buy a program written by someone else, the program is in charge, and decides how you have to proceed. When you write your own program, you are in

charge, and you decide what the program produces. If you don't need VAT paperwork, the program doesn't produce it. If you want printed receipts, your program provides them. If you need a list of customers in alphabetical order, your program can be made to give that, or a list in order of how much they owe you – it's all up to you to decide what you want and arrange for it to be supplied. Control, then, is the main reason for wanting to program.

There's another reason, which has nothing to do with business but a lot to do with curiosity. You can use a computer as you might use a car, putting up with its odd little ways, but never doing anything to understand them. Using a computer in this way is never entirely satisfying – you always feel that the machine will have the last word. Just as by understanding what makes a car tick (or run smoothly as the case may be) you can drive it better and avoid breakdowns; you can also, by learning more about the computer become able to make better use of it. Computers are still at an early stage of development. It would not be an exaggeration to say that small computers are today in much the same state as cars were when the Model T Ford was introduced. Perhaps the PCW machines will be the Model Ts of the computer world. In any case, the more you know about the machine, the better you can drive it. In addition, programming is a very considerable aid to thinking. When you learn to program, you also learn to break a problem down into manageable pieces, and work on the pieces. If you are programming for some business reason, you'll learn a lot more about your business from writing the program, than you imagined possible. If you program for a hobby reason, then both your hobby and your computing will come on in leaps and bounds. Programming is the most stimulating mental activity that there is, and you don't have to start at university level to get a lot from it – just watch a class of 8-year olds at work with a computer!

Programming languages

A program for a computer is a set of instructions, and a programming language is concerned with how these instructions are written. When you use a computer, as distinct from programming it, you use *commands*. Each command, like the DIR and DISCKIT of your CP/M+ operating system, is set into action by typing its name and then pressing the RETURN key. In a program, the words of command are written in the sequence in which we want them to be carried out, but they are not carried out until we issue a command word. The difference is important. A direct command is carried out by typing the name, then pressing RETURN. If you want to repeat the command, you have to go through all of these steps again. A program, by contrast, can consist of a number of separate steps, written once, and which can be executed as many times as you like just by using one command, usually RUN. The words or codes that are used to mean instructions in a program are what make up the programming language, along with the way that the words must be used, which is the *syntax* of the language.

To start with, we have to look at programming languages generally. At the bottom of the heap of programming level comes machine code, writing directly in number-codes. Machine-code programming is tedious, error-prone and very difficult to check. The only reasons for using it are that it's the only way to program a completely new microprocessor for which nothing is written, because of its speed, and because of the control that it gives you. There are always features of a machine that can be controlled only by machine code. No language like BASIC can ever cope with every possibility and if you want to have your screen scrolling sideways, or to run an unusual disc system, or to use a non-standard printer, then you will probably need machine code to write the routines. There is such a continuing need for machine code that we need a programming language, assembler language, in order to be able to write machine code with less tedium and fewer bugs. Assembler language, then, comes slightly higher up the scale compared with machine code, because it's easier to write, but it still produces the fastest machine code.

What about the other end of the scale? There's one 'high-level' language, COBOL, which looks almost like a set of instructions in English when you read a program. When you write programs in a language of this type, you don't expect to have to worry about the details of the machine. You aren't interested in what microprocessor it uses (on a mainframe, you don't even have a microprocessor). You don't need to know ASCII codes, or routine addresses in RAM, or any of the things that constantly occupy the minds of assembly language programmers. You simply write your program lines, run them into the machine, and sit back like anyone else until the program crashes. The language processor (a compiler) converts the instructions of your program into machine code, and the computer executes the machine code. The name 'high-level' is a good one – you are so far above the ground level of machine code that you hardly know there's a machine there. Needless to say, the language is the same no matter which machine you happen to be programming.

Between the heights of COBOL (and also FORTRAN and ALGOL) and the ground level of machine code, there are languages at all sorts of intermediate levels. The fundamental problem is that high level languages are powerful but inefficient. They allow you to turn your problem-solving methods into programs that run smoothly, but at a great cost in memory space. They can be cumbersome, using lines and lines of program which can take all day to run. At the other end, machine code is very compact, very fast, very efficient – but sheer hell to write and debug in any quantity. The reason that we have such a large number of programming languages is that we are constantly trying to get a better balance of these different virtues. What we want is a high level language that makes it easy to express our solutions, is very compact both in statement length and use of memory, and which translates into a few bytes of machine code as would be given by an assembler. There's no such language, and probably never will be, but some come nearer than others, and some are a better solution for some kinds of problems.

The problem that BASIC was devised to solve was the problem of teaching the language FORTRAN. BASIC is an acronym for 'Beginners All-purpose Symbolic Instruction Code', and that's what it originally was – a simple language intended to serve as an introduction to programming, and

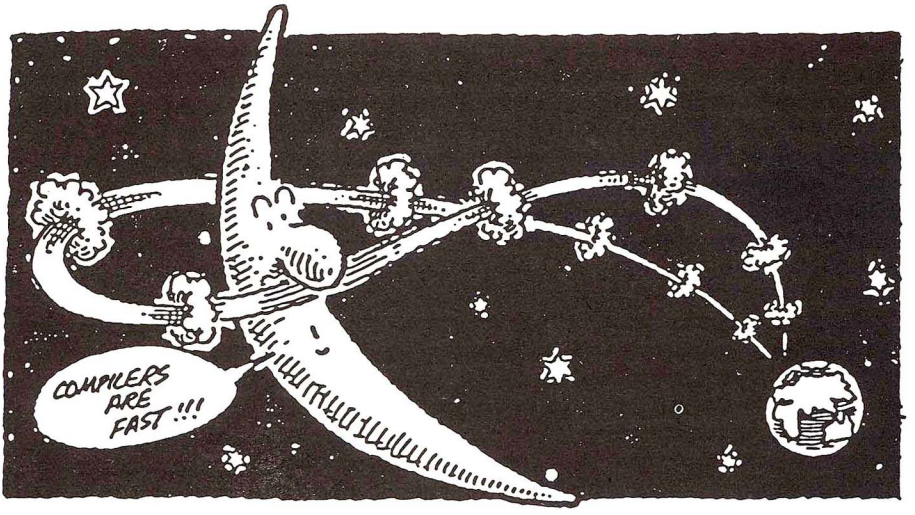
modelled on one of the great original computer languages, FORTRAN. The advantage of the original BASIC was that it was simple to learn, but close enough to the methods of FORTRAN to make the conversion easy. BASIC could be provided and used in interpreted form (see later), so that mistakes could be easily and quickly found and corrected. Finally, BASIC could be written in code that took up only a small amount of memory. It was for precisely these reasons that when microcomputers became available, they featured BASIC as their commonest programming language.

Since then, BASIC has developed a long way, and it's no longer just a language for beginners. As the language grew, it acquired more features from other languages, and without losing its simplicity, soon became a language in its own right, not just a path to an almost-forgotten FORTRAN. Because of the intensive use of BASIC versions on small computers, the language became a general-purpose one, good for all kinds of programs whether your interests were in accounts, science, engineering, text editing, or whatever. Other languages tended to remain specialised, good for only one or two selected purposes, while BASIC grew to fit the needs of users. Nowadays, more people can program in BASIC than in any other language, and they don't necessarily learn it so that they can learn another language. After all, you don't learn English so that you can later learn Icelandic.

Compilers and interpreters

Like many other languages, BASIC can be interpreted or compiled, but is nearly always interpreted. Whatever language you use to express your program in, it has to be converted into machine code before it can have any effect on the computer. Interpreting and compiling are two methods of carrying out this conversion. When a language is interpreted, each instruction is taken, converted into machine code, and then executed before carrying on to the next instruction. In practical terms, this means that each instruction word of the language calls up a set of number-codes to do the work. A compiled language, by contrast converts all of the instructions of a program into a large machine code program. This is very often recorded on disc rather than being run at the time, and the action of translating from high level language into machine code is called compiling. Once the program has been compiled, it is a machine code program, just like the `.COM` programs of CP/M. If the compiler is a good one, the machine code may be almost as compact as it would have been if created by an assembler (referred to as 'native' machine code).

Nothing's perfect, and both interpreters and compilers have merits and faults. The main merit of an interpreter is easy correction of errors. When your program stops with an error message, you can change the text, and start again. For a compiled program, this would mean loading in the text, changing part of it, compiling and recording the machine code, and then running again. The disadvantage of an interpreter is low speed, because the process of finding the correct machine codes for each instruction can be a lengthy one. When you compile, this action is also done, but once only



during compiling, not when the program is running. Your Mallard BASIC is a well-designed interpreted BASIC, allowing easy editing of programs, yet running at a very fast rate. In addition, it incorporates all of the improvements that have been made in BASIC over the years and is as up-to-date a BASIC as you are likely to find.

Principles of programming

On the surface, programming is quite straightforward. You type a list of instructions in the order that you want the machine to carry them out. You record this set of instructions, which is the program, on a disc, and load it in when you need it. That's all. It would be just as simple as this if the machine could understand English, but it can't. BASIC, like each other computing language, allows you to use a limited number of instruction words. For an extended BASIC like Mallard BASIC, this number can be large, 180 or more. It's still very small, however, compared with the thousands of words that a 'natural' language like English uses, and one of the problems of learning programming is trying to express what you want to do with such a limited number of words. This means that you have to break down any problem into small pieces that can be tackled by using a few of these 'reserved words' of BASIC. The other snag is that each reserved word or keyword has to be used in a very precise way, the *syntax* of the word. If you don't use the word correctly, the instruction cannot be carried out, and you will get an error message that reads `syntax error` to draw your attention to it. Programming requires precision, then, and how you use and place words is as important in a programming language as it is in Latin – which is why Latin scholars often make good programmers. In short, programming teaches you to analyse problems, and to tackle them with precision – and that can't be bad training for anything.

We'll see as we progress how all of this can be done, but at the start we can't really illustrate problem solving with BASIC when we don't know much

BASIC. The best way to get started is the practical way, because when you have done something for yourself, you remember it better than when you have only read about it. I'll assume, then, that you can carry out the instructions in this book as and when they appear. The first step is to prepare a disc that will allow you to load in BASIC when you want it, and to write and store programs that you have typed. The programs cannot be written nor used unless BASIC.COM has been loaded first. To do this, you need a copy of your master disc, Side 2, the one that contains the BASIC.COM program. You also need a formatted blank disc that you can reserve for use with BASIC programs. You can fit a lot of BASIC programs onto one disc, and often the limit is reached when you have filled the directory of the disc, meaning that you have 64 separate programs on the disc.

Start by switching on the machine and inserting the CP/M + disc on side 2. This will load itself, and then give you the message A>, which is called the CP/M *prompt*. If at any time when you are working with BASIC you see this symbol appear again, it's likely that something has gone very far wrong, because this 'prompt' never normally appears in BASIC. At the moment, though, it's in its correct place, telling you that the CP/M operating system is ready for a command. Now type:

```
erase m:*.*
```

and the computer will respond:

```
ERASE M:*.* (Y/N)?
```

to which you should type in a "y", **first making sure it says M: or you'll wipe out your CP/M disc!** This has cleared the memory "disc" ready for use in the following file transfers. Now type the command word PIP, and press RETURN. The disc will spin, and very soon you'll see a new prompt symbol, the asterisk *. This means that a program called PIP, which is intended for copying files, is in place and ready to work for you.

Type:

```
m:=a:*.ems
```

wait for the asterisk to reappear and type:

```
m:=a:basic.com
```

pressing RETURN after each line. The files are now held in memory (on drive M), so to transfer them put a blank formatted disc in drive A and type:

```
a:=m:*.ems
```

wait for the asterisk, then:

```
a:=m:basic.com
```

and both files will have been transferred. To return to the CP/M+ prompt, just press RETURN.

You should end up with a disc that contains two files, which you can check by typing DIR RETURN when the disc is in place. One file will be J11CPM3.EMS or J14CPM3.EMS, and the other will be BASIC.COM. This is the disc that you need to have handy in order to start programming with BASIC. To use the disc after switching on the machine at first, insert the disc and wait until you see the A> prompt. Then type BASIC RETURN and wait until the screen notice shows the copyright notice for Mallard BASIC. If you want the convenience of a disc that will automatically start up with BASIC, then see Appendix A. When BASIC starts running, the prompt sign is a rectangular green block, and you will normally see the word Ok just above the block if some action has just finished. This rectangular block is the cursor for BASIC, and is very much the same as the cursor that you use in *LocoScript*.

Disc actions from BASIC

When you use CP/M, including many programs that make use of the CP/M system, you can use commands like DIR to get the disc directory, and REN to rename a file and so on. Some of these actions are available while you are using BASIC, but not all. You can type DIR RETURN to get a file directory, and you can use DIR A: or DIR B: if you have more than one drive. You can type SYSTEM RETURN to get back to CP/M if you want to run a CP/M program instead of BASIC – but unless you have saved any BASIC program that you are using on to a disc, then it will be lost, so be careful. You can use REN to rename a file, just as you use REN in CP/M with the syntax REN oldname=newname. You can also use ERA filename to erase a file, just as in CP/M. To change drives, you can type option files b: (if logged onto drive A:), or you can use the letters b: or m:. These are the important actions for which you would otherwise have to leave BASIC to get to the CP/M system, and with these you have fairly complete control over what is on your discs. In the next section, then, we can start by assuming that you have BASIC loaded, and you're rarin' to go.

Chapter 2

Printing

When you think of what the computer does when you run programs on it, you soon see that its actions fall into four categories: typing data; seeing data printed on the screen or on paper; calculation, arrangement, or other work; and saving to and loading from disc. Very often the third type of action, the actual computing, takes the least time. The first two are of very considerable importance, because they are the actions that you are most directly concerned with. In this chapter, then, we'll look at how you can write program instructions that result in something being printed on to the screen, or on to paper. Until you have some mastery of this particular craft, you can't very well tell whether your computer is doing anything useful or not.

The first step, though, is to check that you can save and load a BASIC program. That's not because there's any difficulty about it, but it's different from the techniques that you use either with *LocoScript* or with CP/M business programs. The simplest test involves a very simple type of program, but it will give you the confidence in handling the disc recording process. It can be a very heart-breaking experience to spend a lot of time typing in a program, and then find that it vanishes when you switch off because it was not correctly recorded. If anyone tells you that they have never done it, don't believe them. We all have, and one good method of losing data in this way is to be unfamiliar with the recording system.

Start, then, by 'booting up' BASIC as described in chapter 1. When you see the copyright notice, the number of free bytes (allowing that same number of characters to be put into the memory), and the `Ok` prompt followed by the cursor, you're ready to type a program. Mallard BASIC allows you to type program instruction words in either lower-case or upper-case letters. Any instruction words, however, are converted to upper-case when the program is put into the memory. You could use the SHIFT LOCK to ensure that words appeared in upper-case, but because this key also affects the number keys it's better not to use it. One advantage of typing in lower-case is that if you mistype a command word, it won't be converted, and that's a good clue to a mistyping. When you want something reproduced on the screen, you can then use lower-case as you please. In the listings throughout this book, the instruction words will appear in upper-case.

In any variety of BASIC, you can type direct instructions that are obeyed when you press RETURN, or you can type program instructions. The machine can distinguish the two because a program instruction, or set of instructions, will always start in a line of its own with a line number. This number will be a positive whole number, and the other point about it is that the order of these numbers decides the order in which the instructions are carried out. A BASIC instruction, complete with any data that it needs to carry it out, is called a 'statement', and the simplest way of writing a BASIC program is to have one statement on each numbered line. By convention we number these lines as 10, 20, 30, 40 and so on, rather than as 1, 2, 3, 4, etc. This allows space for second thoughts because if you have typed lines 10 to 40 and you want a line between 10 and 20, you can type a line 15 at any stage. The machine will then insert it into the correct position automatically – one considerable advantage of using numbered lines. If necessary, the machine can renumber all of the lines of a program.

Now type the number 10 (1 and then 0), and then the word rem. If your 10 turns out as !), you have pressed the SHIFT LOCK down, which is why I don't recommend using this key. It doesn't matter whether you type `rem` or `REM`. This is a command word for the computer, and no matter whether you type it in upper or lower-case, the computer will (later) convert it into upper-case. Check that this looks correct, and then press the RETURN key. The effect of this is to place the instruction line '10 REM' into the memory of the machine. As you type the first digit, the character will be seen on the screen, in lower-case, at the cursor position. When you press the RETURN key, the cursor moves to the next line down. At the same time, your command stays where it was typed on the screen. If you used lower-case, then you will still see it in lower-case, like this:

```
10 rem
```

If you see a mistake as you type the line, just use the back space action, by pressing the DEL key. This moves the cursor backwards, and deletes the letter that the cursor is now over. You can then type the correct letter, which will replace the incorrect one. If this makes the line correct, pressing RETURN will enter it into the computer. If you have typed something that is incorrect, like `REN` or `RWM` then you will not be warned in any way – the computer accepts anything that you type following a number. To correct a line such as:

```
20 Ren
```

you can, instead of using DEL, just type the correct version:

```
20 rem
```

and press the RETURN key. Pressing the RETURN key enters the line into the memory. Even if your 'line' happens to take up more than one line on the screen, you don't press RETURN until you have finished and want the complete line placed into the memory. Now type the rest of the lines, as illustrated in figure 2.1, remembering to press the RETURN key after you have completed typing each line. You can check that your program looks correct by asking the computer to 'list' it. Listing means that the computer prints on the screen whatever you have stored in its memory. Using the 'line numbers' ensures that the instructions are stored, and if you type `list`, and then press RETURN, you will see your program. Don't be

```
10 REM
20 REM
30 REM
40 REM
```

Figure 2.1 A simple program for checking the disc SAVE and LOAD actions.

surprised to find that all the lower-case letters (like rem) have been converted to upper-case (like REM), because this is part of the action of the computer, along with putting line numbers in order, and leaving a space between the last digit of the number and the first letter of the command. Check from this 'listing' that the program is like the printed version in figure 2.1.

Now to make the recording. First of all you have to type:

```
SAVE "TEST"
```

and then press the RETURN key. The word SAVE is the instruction to the computer meaning that you want to save (record) a program onto a disc, and the TEST part is the filename which the computer will use to recognise the program if it is asked to. This is an action that you are likely to use in *LocoScript*, and the same restrictions on filenames (8 characters maximum in main name, optional extension of up to three characters, and so on) will apply. You will normally keep a large number of different programs on each disc, and you need the filenames to instruct the computer to load back the correct one. When you press the RETURN key, the disc will spin, and in a very short time you will see the Ok message and the cursor to inform you that the program has been saved.

Now comes the crunch. You have to be sure that the recording was valid. Type NEW and press RETURN. This should have wiped your program from the memory. At this point, it would be useful to clear the screen so as to remove the old listing. This action isn't so straightforward on the PCW machines as it is on all others, but there is a reasonably simple method that can be used for as long as you have the machine switched on. Type the following line:

```
cls$=chr$(27)+"E"+chr$(27)+"H"
```

and press RETURN. When you see the Ok and cursor, you can clear the screen by typing ?cls$. This will work until you switch off or clear the memory with NEW, and you can make it your first action when you switch on or start another program. There are ways of making a key carry out this useful action, but we'll reserve that for appendix B, because it requires rather more advanced programming. Now type LIST and press RETURN. Nothing should appear – LIST means put a list of the program instructions on the screen, and there shouldn't be any following NEW, which clears out the program.

You can now load the instructions back in from the disc. Type LOAD "test" and then press RETURN. You will once again hear the disc spin, and then see the usual Ok and cursor prompt. Type LIST now, then press the

RETURN key. You should see your program appear on the screen. Once you can reliably save programs on disc, and reload them, you can confidently start BASIC computing. When you have spent an hour or more typing a program on to the keyboard, it's good to know that a few seconds more work will save your effort on disc so that you won't have to type it again. There are several variations on LOAD and SAVE so that you can save your program in different forms, including a 'protected' form, and also so that you can load and run in one action, or load automatically whenever the BASIC.COM file is loaded. At this stage, there's no point in worrying about all of these optional extras, and they are dealt with in Appendix C.

What you have seen so far will have broken you in to the idea that the PCW machines, like practically all computers, take their orders from you when you type them on the keyboard. You will also have found that an order is obeyed when the RETURN key is pressed, but a BASIC statement that follows a line number is just placed in the memory to be executed later. You have used the command LIST which prints your program instructions on to the screen, and you know how to clear the scren when you need to. As your familiarity with the computer keyboard increases, you will want to make use of the editing commands, and these are explained in Appendix D.

An important point about all computer commands, whether they are direct commands or program instructions, is that they have to be in a precise form. The spelling of a command word must be perfect, for example, or it won't be obeyed. It must also be used in the right way. For example, you can't just type SAVE, then press RETURN, and expect the computer to do anything. The computer expects a quote mark and then a filename to follow the word SAVE, and it can't act on the command if the command is incomplete. Some commands include spaces between words, and will not work if a space is missed out. In other places, you find that a space is not important. In general, it does no harm to have spaces, called 'whitespace' between commands, or parts of commands, but it can be a problem if you leave them out unless some separating symbol like quotes is used. You have to learn these things by experience, because there is no simple guide as to when you must put in a space and when you can leave one out. You know also now, that either a direct command or a program line has to be followed by pressing the RETURN key, so that I don't have to keep reminding you by printing things like LIST (press RETURN) each time. A program can be ended with a line that consists of the word END, but if this is the last line of the program anyhow, it isn't necessary.

Let's now take a look at the difference between a direct command and a program instruction. If you want the computer to carry out the direct command to add two numbers, 1.6 and 3.2, then you have to type:

```
PRINT 1.6 + 3.2
```

and then press RETURN)

You have to start with PRINT (or print), because a computer is a dumb machine, and it obeys only a few set instructions. Unless you use the word PRINT, the computer has no way of telling that what you want is to see the answer on the screen. It doesn't recognise instructions like GIVE ME or WHAT IS, only a few words that we call its *reserved words* or *instruction words*. PRINT, which can be abbreviated to ? in Mallard BASIC, is one of these

words. Remember that there must be a space following the T of PRINT. You do not need to have spaces between the 6 and the + or the + and the 3. Curiously enough, you don't need a space if you use the ? symbol, so that you can type ?1.6+3.2 and use this with no error messages. Since using the ? mark saves so much typing, it's very useful to know when you have a lot of lines which start with a PRINT command.

When you press the RETURN key after typing PRINT 1.6 + 3.2, the screen shows the answer, 4.8. This answer is not shown in the same place as your typed command, however. It is on the next line, and starting one space in from the left-hand side. If you have just cleared the screen before typing, the answer will appear near the top left-hand corner. Once a direct command has been carried out, however, it's finished. A program does not work in the same way. A program is typed in, but the instructions of the program are not carried out when you press the RETURN key. Instead, the instructions are stored in the memory, ready to be carried out as and when you want, or recorded on disc to be used many times over. The computer needs some way of recognising the difference between your commands and your program instructions. On computers that use the 'language' called BASIC this is done by starting each program instruction with a line number. This is why you can't expect the computer to understand an instruction like 5.6 + 3=; it takes the 5 as being a line number, and the rest doesn't make sense.

Let's start programming, then, with the arithmetic actions of add, subtract, multiply and divide. I'm doing this just because these are simple to work with. Computers aren't used all *that* much for calculation, but it's useful to be able to carry out calculations now and again. Figure 2.2 shows a four-line program which will print some arithmetic results.

Take a close look at this, because there's a lot to get used to in these four lines. To start with, there's a useful way to get the line numbers put in for you automatically. This is done by typing the command word auto and then pressing RETURN. When you do this, you'll see the first line number of 10 appear on the screen, and each time you press RETURN at the end of a statement, you'll see the next number appear. You can get out of this by pressing the STOP key. Using AUTO, however, is very useful, because it means less typing and less chance of giving two lines the same number. If you repeat a line number, the most recent line replaces any line of the same number that you typed earlier.

The next thing to notice is how the number zero on the screen is slashed across. This is to distinguish it from the letter O. The computer simply won't accept the 0 in place of O, nor the O in place of 0, and the slashing makes this difference more obvious to you, so that you are less likely to make mistakes. The zero that you see on the keyboard is also slashed, it is on a different key, and is differently shaped. Type some zeros and Os on the screen so that you can see the difference. In the listings, you will see the zero slashed, but it will *not* be slashed on the output from your printer unless you carry out the instructions in appendix E.

Now to more important points. The star or asterisk symbol in line 30 is the symbol that BASIC uses as a multiply sign. Once again, we can't use the 'x' that you might normally use for writing multiplication because this is a letter. There's no divide sign on the keyboard either, so BASIC uses the

```
10 PRINT 5.6+6.8
20 PRINT 9.2-4.7
30 PRINT 3.3*3.9
40 PRINT 7.6/1.4
```

Figure 2.2 A four-line arithmetic program.

```
10 PRINT"2+2="2+2
20 PRINT"2.5*3.5="2.5*3.5
30 PRINT"9.4-2.2="9.4-2.2
40 PRINT"27.6/2.2="27.6/2.2
```

Figure 2.3 Using quote marks to show that characters have to be printed on the screen exactly as typed.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT"This is"
30 PRINT"the excellent"
40 PRINT"Amstrad PCW machine"
```

Figure 2.4 Using the PRINT instruction to clear the screen and then place words on the screen.

slash (/) sign for this. This is the diagonal line on the same key as the question mark.

So far, so good. The program is entered by typing it, just as you see it. You don't need to leave any space between the line number and the P of PRINT, because Mallard BASIC will put one in for you when it displays the program on the screen. You *must* leave a space following PRINT though, and I have to emphasise this because not all computers are quite so fussy. If you use the '?' abbreviation for PRINT, you don't have to worry about this particular space. Getting back to the program example, you will have to press the RETURN key when you have completed each instruction line, before you type the next line number. You should end up with the program looking as it does in the illustration. When you have entered the program by typing it, it's stored in the memory of the computer in the form of a set of code numbers. There are two things that you need to know now. One is how to check that the program is actually in the memory, the other is how to make the machine carry out the instructions of the program.

The first part is dealt with using the command LIST that you know already. You can use the ?cls$ dodge to wipe the screen first if you like (or the CAN key if you have followed the advice in appendix B), then type LIST and finally press the RETURN key. When you have pressed the RETURN key,

and not until then, your program will be listed on the screen. You will then see how the computer has printed the items of the program on the screen, with spaces between the line numbers and the instructions. The '?' signs, if you used them, have been converted to the word PRINT, as has the word print if you typed it in lower-case, and a space has been inserted between the word and the first digit of a number. To make the program operate, you need another command, RUN. Type RUN, then press the RETURN key, and you will see the instructions carried out. To be more precise, you will see:

```
12.4
4.5
12.87
5.428571
```

That last line should give you some idea of how precisely Mallard BASIC can carry out this type of arithmetic. If you need more than six places of decimals, then there are ways, as we shall see. You'll notice, by the way, that if you listed the program before you ran it, the results are printed under the program listing. You can avoid this by clearing the screen in the usual way before you use RUN.

When you follow the instruction word PRINT with a piece of arithmetic like 2.8*4.4, then what is printed when the program runs is the *result* of working out that piece of arithmetic. The program *doesn't* print 2.8*4.4, for example, just the result of the action 2.8*4.4.

Now this is useful, but it's not always handy to get a set of answers on the screen, especially if you have forgotten what the questions were. Mallard BASIC allows you a way of printing anything that you like on the screen, exactly as you type it, by the use of what is called a 'string'.

Figure 2.3 illustrates this principle. In each line, some of the typing is enclosed between quotes (inverted commas) and some is not. Enter this short program, clear the screen, and run it. Can you see how very differently the computer has treated the instructions? Whatever was enclosed between quotes has been printed *exactly* as you typed it. Whatever was not between quotes is worked out, so that the first line, for example, gives the unsurprising result:

```
2+2= 4.
```

Now there's nothing automatic about this result being correct arithmetic. If you type a new line:

```
15 PRINT "2+2="5*1.5
```

then you'll get the daft reply, when you run this, of:

```
2+2= 7.5
```

The computer does as it's told and that's what you told it to do. And some people think that computers could take over the world!

This is a good point also to take notice of something else. The line 15 that you added has been fitted into place between lines 10 and 20 – LIST it if you don't believe. No matter in what order you type the lines of your

```
10 PRINT"This is ";
20 PRINT"the excellent ";
30 PRINT"Amstrad PCW machine."
```

Figure 2.5 The effect of semicolons to prevent a new line from being taken.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT"This is Mallard BASIC"
30 PRINT:PRINT
40 PRINT"Ready to work for you!"
```

Figure 2.6 Spacing lines with PRINT, in this example using multistatement lines.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT 1,2,3
30 PRINT 1,2,3,4,5,6
40 PRINT"One","Two","Three"
50 PRINT"One","Two","three","Four",
"Five","Six","Seven"
60 PRINT"This item is considerably
longer","Two","Three","Four","Five"
```

Figure 2.7 How the comma causes words or numbers to be placed into 15-character columns.

program, the computer will sort them into order of ascending line number for you. In addition, you'll see that the computer has put a space between the = sign and the first digit of the answer. This is to allow for a + or - sign, and we sometimes want to put in an extra space here. This can be done by leaving a space between the = sign and the final quotemark (like = ") in the line.

With all of this accumulated wisdom behind us, we can now start to look at some other printing actions. PRINT, used alone in this way always means print on to the monitor screen. For activating the paper printer (hard copy, it's called), there's a separate variety of PRINT instruction, LPRINT, and a separate variety of LIST, called LLIST. If you load a sheet of paper into your printer, and press EXIT in the usual way, you'll get your program listed on paper when you use the command LLIST. If you want your program to print the results on paper, then each PRINT in the program must be converted into LPRINT – a job for the editor!

Now try the program in figure 2.4. You can try typing the lines in any order that you like, to establish the point that they will be in line-number order when you list the program. When you RUN the program, the screen is cleared, the cursor is shifted to the top left-hand corner, and the words appear on three separate lines. This use of separate lines is because the instruction PRINT doesn't just mean 'print on the screen'. It also means

'move to the start of a new line', and start at the left-hand side. You will also find, incidentally, that when words on the screen reach the bottom line of the screen, then all the lines appear to move up, and the top line disappears. This action is called 'scrolling', and it's the way that the machine deals with displaying lots of lines on a screen which holds only up to 32 lines altogether. The first line in this program is the command for clearing the screen and for starting at the top left-hand corner. We'll look more closely at this command in chapter 6 and in Appendix B, but for the moment it's useful to know as a way of getting that elusive screen-clear action.

Now the action of selecting a new line for each PRINT isn't always convenient, and we can change the action by using punctuation marks that we call print modifiers. Type NEW and then press the RETURN key. This clears the old program out, and you might also like to clear the screen. If you don't use the NEW action, there's a chance that you will find lines of old programs getting in the way of new ones. Each time you type a line, you delete any line that had the same line number in an older program, but if there is a line number that you don't use in the new program it will remain stored. In figure 2.2, for example, the line 15 that you added later would be left in store even when you typed a new line 10 and a new line 20. Another way of clearing out old lines is to type DELETE 1Ø-, which will delete all lines from 10 onwards. If you don't want to delete line 10, you can use DELETE 11-, and this can be very useful if you want to keep, for example, a screen-clear line in more than one program. You can also, incidentally, type a command like RUN 2Ø to make the program start at some specified line.

Now try the program in figure 2.5. There's a very important difference between figure 2.5 and figure 2.4, as you'll see when you RUN it. The effect of a semicolon following the last quote in a line is to prevent the next piece of printing starting on a new line at the left-hand side. When you RUN this program, all of the words appear in one line. It would have been a lot easier just to have one line of program that read:

```
1Ø PRINT "This is the excellent Amstrad PCW machine."
```

to do this, but there are times when you have to use the semicolon to force two different print items on to the same line. We'll look at that sort of thing later in program examples.

Rows and columns

Neat printing is a matter of arranging your words and numbers into rows and columns, so we'll take a closer look at this particular art now. To start with, we know already that the instruction PRINT will cause a new line to be selected, so the action of figure 2.6 should not come as too much of a surprise. Line 10 clears the screen and puts the cursor home to its top left-hand corner position, as before. Line 30 contains a novelty, though, in the form of two instructions in one line. The instructions are separated by a colon (:), and you can, if you like, have several instructions following one line number in this way, taking several screen lines. The only practical limit

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT TAB(42)"Centre"
30 PRINT TAB(1)"Start here..."
40 PRINT TAB(60)"60 is here"
50 PRINT TAB(87)"EDGE"
```

Figure 2.8 How TAB is used to position the cursor along the line in a PRINT statement.

to this is that it makes your instructions too hard to read if you put too many instructions together in this way. In a 'multistatement' line of this type, Mallard BASIC will deal with the different instructions in a left-to-right order. Another point about figure 2.6 is that line 30 causes the lines to be spaced apart. The two PRINT instructions, with nothing to be printed, each cause a blank line to be taken. There are other ways of doing this, as we'll see, but as a simple way of creating a space, it's very handy.

Figure 2.7 deals with columns. Line 10 clears the screen as usual, and line 20 is a PRINT instruction that acts on the numbers 1, 2 and 3. When these appear on the screen, though, they appear spaced out just as if the screen had been divided into columns. The mark which causes this effect is the comma, and the action is completely automatic. The comma is on the key next to the letter M, and if you use the apostrophe on the 6 key, you will not get the same effect! The two look rather alike on the keyboard, but completely different on the screen. As line 30 shows, you can get up to six columns, each one of which allows room for up to fifteen characters. Anything that you try to get into a seventh column will actually appear on the first column of the next line down. The action works for words as well as for numbers, as lines 40 and 50 illustrate. When words are being printed in this way, though, you have to remember that the commas must be placed *outside* the quotes. Any commas that are placed inside the quotes will be printed just as they are and won't cause any spacing effect. You will also find that if you attempt to put into a column something that is too large to fit, the long phrase will spill over to the next column, and the next item to be printed will be at the start of the following column. Line 60 illustrates this – the first phrase spills over from column 1 all the way to column 3, and the word Two is printed starting at column 4 on the same line. If you keep using commas for more than six items the columns just take up the same positions on the next line.

This action of commas is used on practically every home computer, but the Mallard BASIC adds a new twist. Type ZONE 4, press RETURN, then RUN your program again. This time, things fit better! It's as if you suddenly had more columns, and you do. The number that follows ZONE gives the number of spaces between columns, so it allows you to mark out the screen (invisibly) for printing in any way that you like. You can change the value of ZONE during a program, for example, so that different bits of printing are differently arranged. This is particularly useful if you want to make tabulated work for business purposes. For longer words, you might like to use zone 20, which will allow four columns per width of screen.

Commas are useful when we want a simple way of creating columns which all have the same spacing. A much more flexible method of placing words

1. Count number of characters in the title, including spaces.
2. Subtract this number from 90 if it is even, from 91 if it is odd.
3. Divide the result by 2.
4. Use the result as the TAB number.

Figure 2.9 The formula that is used to centre a word or phrase in a line.

on the screen exists, however. This is programmed by using the command word TAB, which has to follow PRINT. TAB is short for 'tabulate', and it means 'start printing at a stated position'. For the purpose of using TAB, we need to remember that the screen, as it exists when we switch on, is divided into 90 columns across by 31 lines down. The positions are numbered from 1 at the left-hand side to 90 at the right-hand side (you can also use TAB(0), which is the same position as TAB(1), the left-hand edge). TAB(1), then, would mean the left-hand side, and TAB(90) would be the right-hand side. The word TAB must follow PRINT, and must itself be followed by the TAB number in brackets. If you omit the brackets, you'll will see the odd effect of a zero being printed as well as your number or phrase – you'll understand why later when you read about number variables.

Now try a TAB example, in figure 2.8. The first word is printed in the centre of the screen, and the second one at the left-hand side. The third word is printed at TAB(60), and the last word right up at the right-hand edge. Notice that the TAB number is the number for the position of the first letter of the word, and so we need to use TAB(87) here in order to get four letters in at the right-hand side. Now suppose you use a line such as:

```
PRINT TAB(2Ø)"A";TAB(11Ø)"B"
```

for two positions, one of which is beyond the 90-character limit. The effect is to put B directly under A, in the TAB(20) position of the line below. That's because numbers of 1 to 90 operate the TAB command, and if you use 91, then it's equivalent to starting all over again with 1 on the next line. Oh yes, there's another point here too. The word 'centre' in this example was printed in the centre of the screen by using TAB(42). Figure 2.9 shows the formula that is used to find the TAB number for making any word or phrase appear centred on the screen.

Figure 2.10 shows a rather different way of spacing out figures and letters on the screen. This uses the SPC command in line 30, and though you might think that it's pretty much like the TAB command, it isn't! When

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT TAB(10)"PCW 8256"TAB(30)"Computer"
30 PRINT TAB(10)"PCW 8512"SPC(30)"Computer"
```

Figure 2.10 Comparing TAB and SPC – note that you can have more than one of each in a line.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 DEF FNloc$(x%,y%,t$)=CHR$(27)+"Y"+CHR$(32+
y%)+CHR$(32+x%)+t$
30 PRINT FNloc$(30,20,"How about this?")
40 PRINT FNloc$(50,2,"...and this")
```

Figure 2.11 The print-at function – the principles will be explained later.

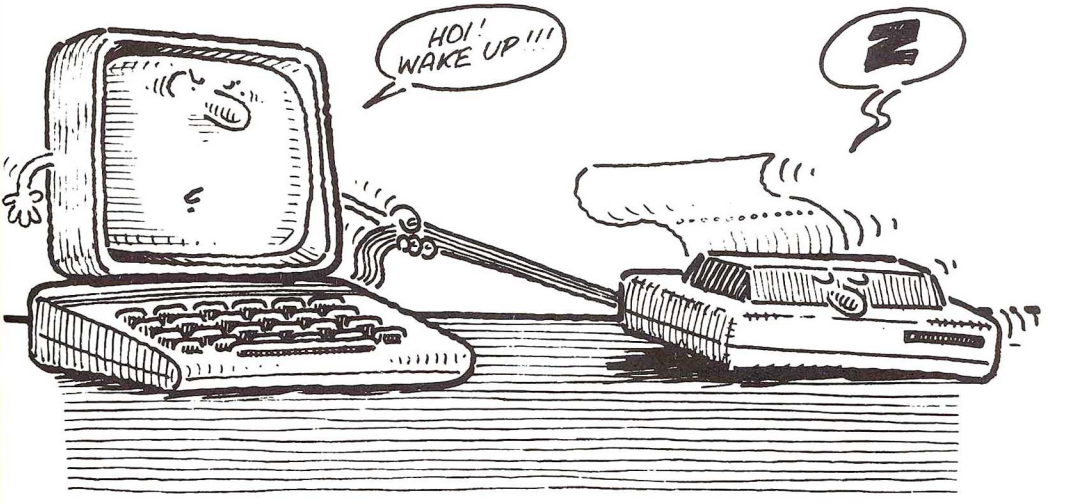
you TAB the positions, the first letter of each word is placed in the appropriate TAB position. Note, incidentally, that you can have more than one TAB in a line following a PRINT command. Unlike most computers, Mallard BASIC doesn't need any semicolons to separate bits of this command, though it doesn't object if you put them in. In line 30, SPC is used. Now this does not TAB to position 30, it prints 30 spaces between the end of `PCW 8512` and the start of `Computer`. That's quite different from having the `C` of `Computer` in TAB position 30. The general rule is that you use TAB if you want neat columns, with the first letter of each word starting in the same position of each line. You use SPC if you want to fix the amount of space between words or numbers, even if these words or numbers are of different lengths.

There's yet another way of positioning your printing, and it's even more free-ranging than TAB. At the moment, we're not quite ready for its explanation, but a demonstration is useful, and is in figure 2.11. This shows how printing can be carried out at any part of the screen and in any order. Normally, the printing action is strictly left-to-right, and top to bottom, and you can't reverse this direction. By using this trick, involving what is called a 'defined function', however, we can guide printing to any part of the screen. In the example, a phrase is printed near the bottom of the screen, and then another phrase is printed near the top. If you kept to ordinary print rules, you could not have printed these phrases in that order. The trick, like the screen-clear trick, depends on the use of letters that are printed as code instructions following the CHR$(27), of which more later.

Planning your printing

Whether you intend to print onto the screen or on to paper, you have to do some planning, because nothing looks worse than ragged printing – take a look at some of the books from 'cottage industry' publishers, for example. Like everything else in computing, it all depends on planning. If you intend to print a lot of text or figures to the screen or to the printer, it's always a good idea to plan it out on squared paper. For this purpose, paper that is ruled in the old-fashioned 1/8" squares is very useful because, if you can get it in foolscap size, you can turn it sideways, and mark off a width of 90 squares for your screen. Also remember to mark off 80 squares – because this is the width of the printer in its normal mode. If you print your text or numbers on this paper, one character per square, it will give you a very good idea of what it will all look like on the screen or on the printed sheet. In addition, you can plan

screen positions by counting squares, so that it makes it easy to find TAB numbers and SPC numbers. Planning like this can save a lot of time that would be otherwise used up in trying the printing out and changing the listing. The old motto of programmers still applies – don't enter anything into the machine until you are fairly sure how it will turn out.



Chapter 3

Exotic variations

So far, our computing has been confined to printing numbers and words on the screen. That's one of the main aims of computing, but we have to look now at some of the actions that go on before anything is printed. All that we have done so far is to print items exactly as they were typed in the program. Real computing starts when you can get things out that are *not* identical to whatever you typed in the program. One of the actions that we need to look at is called *assignment*. Take a look at the program in figure 3.1. Type it in, run it, and contrast what you see on the screen with what appears in the program. The first line that is printed is line 30. What appears on the screen is:

```
2 times 23 is 46
```

but the numbers 23 and 46 don't appear in line 30! This is because of the way we have used the letter X as a kind of code for the number 23. The official name for this type of code is a *variable name*. We can, incidentally, use `x` and `X` interchangeably, the computer will treat them as being identical.

Line 20 assigns the variable name X, giving it the value of 23. 'Assigns' means that wherever we use X, *not* enclosed by quotes, the computer will operate with the number 23 instead. Since X is a single character and 23 has two digits, that's a saving of space. It would have been an even greater saving if we had assigned X differently, perhaps as `X=2174.3256`, for example. On a few computers, you would be forced to write line 20 as `LET X=23`, but Mallard BASIC does not insist on LET, though if you do use it you will not get any error message. Line 30 then proves that X is taken to be 23, because wherever X appears, not between quotes, 23 is printed, and the 'expression' `2*X` is printed as 46. We're not stuck with X as representing 23 for ever, though. Line 40 assigns X as being 5, and lines 50 and 60 prove that this change has been made.

That's why we call X a 'variable' – we can vary whatever it is we want it to represent. Until we do change it, though, X stays assigned. Even after you have run the program of figure 3.1, providing you haven't added new lines

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 X=23
30 PRINT"2 times "X" is ";2*X
40 X=5
50 PRINT"X is now "X
60 PRINT"and twice "X" is ";2*X
```

Figure 3.1 Assignment in action. The letter X has been used in place of a number.

or deleted any part of it, you can type `?X`, and pressing RETURN will show the value of X on the screen. The listing also shows a curiosity. You can follow a phrase which is in quotes by a variable name, as in `PRINT"2 TIMES"X` in line 30. You *cannot*, however, do this if the phrase is followed by an expression, meaning something like 2 × X or X–7. You will get an error message here if you attempt to use `"is"2*X` in the program. Because of this, a lot of programmers automatically use a semicolon wherever a quote sign is followed by a variable name or expression.

Getting back to variables, this very useful way to handle numbers in code form can use a 'name' which must start with a letter, upper-case (capital) or lower-case (small). You can add to that letter other letters, making a complete word if you like, or digits, but not spaces, arithmetic symbols (`+`, `-`, `*`, `/`) or punctuation marks of any kind. Names like `TOTAL`, `lastname`, `ALLTHEREIS` and `R2D2` can all be used for number variables, and each can be assigned to a different number. This gives you a huge choice of names, and allows you to use names that help you remember what you are doing. For example, you might want to use names like `finalsum` or `LASTOFTHEM`. You can't split names up with whitespace or with any kind of dash or punctuation mark other than a full stop, though. You can't, for example, use `price_paid`, but you can use `price.paid`. One thing that you need to be particularly careful about, though, is that Mallard BASIC does not distinguish between upper and lower-case names. If you assign the variable name of `jam` to the number 45, and then assign the name `JAM` to 88, you will find that both `jam` and `JAM` have been assigned the same value, 88 in this case. This will be whichever number you assigned most recently. Another thing to watch for is the use of *reserved words*. The reserved words of Mallard BASIC are its instruction words, words like PRINT, NEW, RUN and so on. You *cannot* use these as variable names, and you will get a `Syntax error` message if you attempt to use them. Some computers won't even allow you to use words which *contain* these reserved words, so that you could not use words like 'NEWLY', for example. Mallard BASIC, however, is more tolerant, and will allow you to use any word which is not identical to a reserved word. If you ask Mallard BASIC to print the value of some word which has not been assigned, it will come up with the value of 0, rather than with an error message, as some machines do.

Just to make it even more useful, you can use similar 'names' to represent words and phrases also. The difference is that you have to add a dollar sign (`$`) to the variable name. If `N` is a variable name for a number, then `N$` (pronounced 'en string' or 'en dollar') is a variable name for a word or phrase. The computer treats these two, `N` and `N$`, as being entirely separate and different. They also have to be assigned in rather different ways. When

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 NAME$="Mallard BASIC"
30 FIRST$="The excellent":LAST$="computer lan
guage."
40 PRINT FIRST$" "NAME$" "LAST$
50 PRINT"This uses the "NAME$
60 PRINT FIRST$" "NAME$" in action!"
```

Figure 3.2 Using string variables. These are distinguished by the use of the dollar sign. The string variable 'name' can consist of as many letters as you need.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 BLURB$="The new computer "
30 PUFF$="that brings real computing at low c
ost"
40 PRINT"The PCW 8256- "
50 PRINT BLURB$;PUFF$
60 PRINT BLURB$PUFF$
70 PRINT BLURB$+PUFF$
```

Figure 3.3 Illustrating the use of string variable names for phrases of several words, and how the phrases can be printed together.

you assign a number to a number variable, using the = sign, you don't have to type a quotemark (") on each side of the number. When you assign a string variable in this way, however, you have to make use of quotemarks. We'll look at other methods of making such assignments later. In the next example, we shall make use of 'long' variable names, meaning more than two letters. Using just one letter saves memory space, but a good choice of variable name can help to remind you of what the variable represents. When you have so much memory to make use of as the PCW machines permit, you can afford to be generous with your names!

String section

Figure 3.2 illustrates *string variables*, meaning the use of variable names for words and phrases. Lines 20 and 30 carry out the assignment operations, and lines 40 to 60 show how these variable names can be used. Notice that you can mix a variable name, which doesn't need quotes around it, with ordinary text, which *must* be surrounded by quotes. You have to be careful when you mix these two, because otherwise it's easy to run words together. Note in lines 40 to 60 how spaces have been left between words. When you are printing one variable after another, the space is created by typing quotes, then pressing the spacebar, then another quotemark, like " ". To leave a space between text in quotes and a variable name, you only need to press the spacebar at the point where you need the space. You can put in semicolons at the joins when you are joining up bits of text in this way if you like, but they are not essential as the example shows.

Figure 3.3 shows another example, this time using the variable names BLURB$ and PUFF$ for longer phrases, and emphasising the three ways in

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A=2:B=3
30 A$="2":B$="3"
40 PRINT A,B
50 PRINT A$,B$
60 PRINT A "times"B" is ";A*B
70 PRINT A$" times "B$" is impossible!";A$*B$
```

Figure 3.4 String and number variables might look alike when they are printed, but they *are* different!

```
10 FIRSTNAME$="Forbes"
20 SECONDNAME$="Jones"
30 FULLNAME$=FIRSTNAME$+"-"+SECONDNAME$
40 PRINT CHR$(27)+"E"+CHR$(27)+"H"
50 PRINT"Just call me "FULLNAME$", he said"
60 PRINT"Just call me "FIRSTNAME$+"-"+SECONDN
AME$", he said"
70 A$="123":B$="456"
80 PRINT
90 PRINT"Joined string is ";A$+B$
100 PRINT"Addition gives ";123+456
```

Figure 3.5 Concatenating or joining strings. This is not the same action as addition!

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 a$="###":B$="***"
30 S$="Mallard BASIC"
40 PRINT TAB(35)A$+B$+S$+B$+A$
```

Figure 3.6 Using concatenation to make a frame for a title.

which one phrase can be tacked to the end of another. There wouldn't be much point in printing messages in this way if you wanted the message once only, but when you continually use a phrase in a program, this is one method of programming it so that you don't have to keep typing it! In line 50, the two variable names are separated by a semicolon. In line 60, they are not separated by anything, and in line 70 they are separated by a + sign. The effects are, as you can see, identical.

Strings and things

Because the name of a string variable is marked by the use of the $ sign, a variable like A$ is not confused with a number variable like A. We can, in fact, use both in the same program knowing that the computer at least will not be confused. Figure 3.4 illustrates that the difference is a bit more than skin deep, though. Lines 20 and 30 assign number variables A and B, and string variables A$ and B$. When these variables are printed in lines 40 and

50, you can't tell the difference between A and A$ or between B and B$. The only noticable difference when you see them printed in columns is that the number variables always have a space in front of the value. The difference appears, however, when the computer attempts to carry out arithmetic. It can multiply two number variables because numbers can be multiplied, but it can't multiply string variables, whether these represent numbers or not. You can multiply 2 by 3, but you can't multiply 2 LABURNUM WAY by 3 ACACIA AVENUE. The computer therefore refuses to carry out multiplication, division, addition, subtraction or any other arithmetic operation on strings. Attempting to do a forbidden operation in line 70 causes an error message when the program runs, and this error will always halt a program. The message that appears is Type mismatch in 70 and it means that you have tried to do an operation with strings that can be done only with number variables. Later on, we'll see that there are operations that we can carry out on strings that we can't carry out on numbers, and attempts to do these operations on numbers will also cause the same error message. The difference is an important one. The computer stores numbers in a way that is quite different from the way it stores strings. The different methods are intended to make the use of arithmetic simple for number variables (for the computer, that is), and to make other operations simple for strings. Let's face it, it's only a machine!

There is one operation that looks like arithmetic that can be carried out on strings, but not on numbers. It uses the + sign, but it isn't addition in the sense of adding numbers. Figure 3.5 illustrates this action of joining strings, which is often called 'concatenation'. This is nothing like the action of arithmetic, as you'll see by lines 50, 60 and 90. Line 30 concatenates (joins) two strings, and line 50 prints out this string, showing that the joining has been done. Line 60 then illustrates that the joining can be done at the time of printing the strings. Line 70 uses numbers in place of the names placed between the quotes, and line 90 illustrates the effects. Just to point out the difference, line 100 shows what would have happened if these had been number variables. Concatenation is a very useful way of obtaining strings which otherwise would need rather a lot of typing, and you have seen already how it forces one string to be tacked on to the end of the other. Take a look at figure 3.6. This defines strings A$ and B$ as characters which can be used as 'frames' around a title. The title is defined in line 30 as Mallard BASIC. Line 40 then prints a concatenated string.

Along with this business of concatenation, there's a very useful command which will join a number of identical characters into a string for you. The command is STRING$, and it has to be followed by two items, enclosed in brackets. The second item is the character that you want to use, and the first item is the number of these characters you want. For example, if you program G$=STRING$(20,"$"), this will make the string G$ contain twenty dollar signs. Figure 3.7 illustrates this STRING$ action used to make a frame for a title. Note that in this and the previous example that if

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A$=STRING$(10,"*")
30 B$=STRING$(5,"#")
40 PRINT:PRINT A$+B$+" TITLE"+B$+A$
```

Figure 3.7 Making a string of characters by using STRING$.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT"What is your name"
30 INPUT NAME$
40 PRINT:PRINT
50 PRINT NAME$" -this is your life"
```

Figure 3.8 Using the INPUT instruction. The name that you type is put into the phrase in line 50.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT"Enter a number, please"
30 INPUT N
40 PRINT:PRINT
50 PRINT"Twice "N" is ";2*N
```

Figure 3.9 An INPUT to a number variable. The quantity that you type must be a number.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 INPUT"Type your name, please ";NAME$
30 PRINT
40 PRINT"Pleased to meet you, ";NAME$
```

Figure 3.10 Using INPUT to print a phrase which requests the input.

you print a listing, it will show the OC27,73 sign, but the key that has been used is actually the hashmark. This can be avoided by using a command: `LPRINT CHR$(27)"R";CHR$(Ø)` – which will set the printer to the US character set which includes the hashmark (#). You will have real problems, however, if you ever want to display both hashmarks and pound signs in one line of a program!

What goes in

So far, everything that has been printed on the screen by a program has had to be placed in the program before it is run. We don't have to be stuck with restrictions like this, however, because the computer allows us another way of putting information, number or name, into a program while it is running. A step of this type is called an INPUT and the BASIC instruction word that is used to cause this to happen is also INPUT.

Figure 3.8 illustrates this with a program that prints your name. Now I don't know your name, so I can't put it into the program beforehand. What happens when you run this is that the words:

```
What is your name
```

are printed on the screen. On the line below this you will see a question mark, followed by the cursor. The computer is now waiting for you to type something, and then press RETURN. Until the RETURN key is pressed, the program will hang up at line 30, waiting for you. If you're honest, you will type your own name and then press RETURN. You *don't* have to put quotes around your name, simply type it in the form that you want to see printed. When you press RETURN, your name is assigned to the variable NAME$. This is one method of assignment to a string variable that doesn't need the use of quotes. The program can then continue, so that line 40 spaces down by two lines and line 50 then prints the famous phrase with your name at the start. You could, of course, have answered Mickey Mouse or Donald Duck or anything else that you pleased. The computer has no way of knowing that either of these is not your true name. Even if you type nothing, and just press RETURN, it will carry on, with no name at all. Don't listen tc the nutters who tell you that computers know everything!

We aren't confined to using string variables along with INPUT. Figure 3.9 illustrates an INPUT step which uses a number variable N. The same procedure is used. When the program hangs up with the question mark and the cursor appearing, you can type a number and then press the RETURN key. The action of pressing RETURN will assign your number to N, and allow the program to continue. Line 50 then proves that the program is dealing with the number that you entered. When you use a number variable in an INPUT step, then what you have typed when you press RETURN must be a number. If you attempt to enter a string, the computer will refuse to accept it, and you will get an error message – `redo from start`. Unlike most error messages, this does *not* cause the program to stop, it simply allows you another chance to get the INPUT step right. It doesn't mean that you have to start your program again from the start, just that you have to do the INPUT step again from the start. If your INPUT step uses a string variable name like N$, then *anything* that you type will be accepted when you press RETURN, but you will get an error message if you try to perform arithmetic on a string variable, like typing `N$*2`.

The way in which INPUT can be placed in programs can be used to make it look as if the computer is paying some attention to what you type, and to place the question mark in the line in which your reply will also be placed. Figure 3.10 shows an example – but with INPUT used in a different way. This time, there is a phrase following the INPUT instruction. The phrase is placed between quotes, and is followed by a semicolon and then the variable name NAME$. This line 20 has the same effect as the two lines:

```
15 PRINT "Type your name, please";
20 INPUT NAME$
```

and this time the question mark and the cursor appear on the same line as the question, and your reply is also on the same line – unless the length of the name causes letters to spill over on to the next line. You can also use the comma in place of the semicolon in a line like this. Try the effect for yourself. The comma causes the question mark to be deleted, but the cursor is still present. This allows you to make use of INPUT more freely, because there may be times when you don't want the question mark used.

The use of INPUT isn't confined to a single name or number. We can use INPUT with two or more variables, and we can even mix variable types in

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 INPUT"Name and number, please ";NAME$,NR
30 PRINT:PRINT
40 PRINT"The name is ";NAME$
50 PRINT"The number is ";NR
```

Figure 3.11 Putting in two variables in one INPUT step. You need to be careful about how you enter the items, however.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 LINE INPUT "Please type name(s) ";NAME$
30 PRINT:PRINT
40 PRINT NAME$
```

Figure 3.12 Using LINE INPUT, which allows the entry of commas and other characters.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 READ J
30 PRINT"Item ";J;
40 PRINT" is an ";
50 READ NAME$
60 PRINT NAME$;
70 READ NAME$
80 PRINT NAME$;
90 DATA 5, extra disc
100 DATA " drive, for the PCW 8256"
```

Figure 3.13 Using the READ and DATA words to place information into a program. This is used only for fixed items that are always part of the program.

one INPUT line. Figure 3.11, for example, shows two variables being used after one INPUT. One of the variables is a string variable NAME$, the other is the number variable NR. Now when the computer comes to line 20, it will print the message and then wait for you to enter *both* of these quantities, a name and then a number. There is just one way of doing this correctly. That is to type the name, then a comma, then the number, and then press RETURN. If you press RETURN after typing the name but before you have typed the number, you will get the `Redo from start` message, and you must then enter both the name and the number, then press RETURN. The name and number will then be printed again in lines 40 and 50 From this, you can see that you can never enter anything that contains a comma when you have an INPUT. For example, if you have an INPUT NAME$ step, you can't enter `BLOGGS, FRED`. This would be treated as two entries, and only `BLOGGS` would be accepted. Attempting to enter `BLOGGS, FRED` would cause the `Redo from start` message. Very particular about detail, these computers, and no two makes are exactly alike! There is, however, another command which allows you to enter items which contain commas. It's LINE INPUT, and you can use it to type in a whole set of characters

(up to 255 characters), including commas or other punctuation marks. When you use LINE INPUT, you must assign to a string variable, and only one string variable, since the comma is not being used as a separator. Try the example in Figure 3.12, and you'll see that this time, you can enter `BLOGGS, FRED` without any problems. You can see also that the LINE INPUT step can be used in exactly the same way as an ordinary INPUT, but that there is no question mark printed on the screen even when the semicolon separator is used. You can use the comma or the semicolon interchangeably here. There is another variation on INPUT which will be more useful to you when you have some experience in program design. This is INPUT$, and it is used in the form `A$=INPUT$(1Ø)`. In this form, a string of exactly 10 characters will be accepted, nothing more and nothing less, and no prompt sign appears. You can, of course, use your own number or number variable name within the brackets of INPUT$.

Reading the data

There's yet another way of getting data into a program while it is running, and it's used for fixed data that can be read from a list. This method involves reading items from a list, and it uses two instruction words READ and DATA. The word READ causes the program to select an item from the list. The list is marked by starting each line of the list with the word DATA. The items of the list must be separated by commas. Each time an item is read from such a list, a *pointer* is altered so that the next time an item is needed, it will be the next item on the list rather than the one that was read the last time round. We'll look at this in more detail in chapter 5, but for the moment we can introduce ourselves to the `READ..DATA` instructions. Figure 3.13 uses the instructions in a very simple way. Line 20 reads an item number, which is the first item on the list and assigns it to the variable J. This is printed in line 30, with the semicolon keeping printing in the same line so that the phrase in line 40 follows it. The semicolon at the end of line 40 once more keeps the printing in the same line, and line 50 reads the name which is the second item in the list. This is assigned to the variable NAME$ and printed in line 60. Line 70 then reads the third item of data from the other DATA line, and prints it so that the whole phrase appears. Now take a look at these two DATA lines, 90 and 100. In line 90, neither the number 5 nor the word `disc` uses quotes. The reason is that 5 is being assigned to a number variable, and `disc` is a straightforward string of characters, with no spaces or commas. In line 100, however, the string starts with a space, and contains a comma, and both of these would create trouble if we omitted the quotes. The space would be ignored, and the comma would cause the reading of that DATA item to stop at that point. If you put a string like this in quotes, the computer is forced to read it as one string.

You always have to be careful about how you match your READ and your DATA. If you use a number variable in the READ, like READ A, then what is in the DATA line being read *must* be a number. If it is not, then the program will stop with an error message. This will show that the DATA line is faulty, and present you with a chance to correct it. Perhaps this is a

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT"Press any key..."
30 K$=INKEY$:IF K$=""THEN 30
40 PRINT"END"
```

Figure 3.14 Using INKEY$ to test the keyboard. This instruction has to be repeated until a key is pressed.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT"Please type Y or N"
30 K$=INKEY$:IF K$=""THEN 30
40 IF K$<>"N" AND K$<>"Y"THEN PRINT"Incorrect
answer- Y or N only, please"
50 PRINT"Your answer was ";K$
```

Figure 3.15 Using INKEY$ to get a Y or N answer.

good time to break off and read the Appendix on editing! If you use a string variable, as in READ A$, then it doesn't matter whether your DATA line contains a number or a string. Remember that if you read a number using READ A$, then Mallard BASIC will not allow you to carry out any arithmetic on that number while it is in string form. Note that you *must* have a space following the word DATA, and that your data items must be separated by commas. If you want to use commas in your DATA items, then you will need to use quotes as illustrated.

The READ..DATA instructions really come into their own when you have a long list of items that are read by repeating a READ step. These would be items that you would need every time that the program was used, rather than the items you would type in as replies. We're not quite ready for that yet, so having introduced the idea, we'll leave it for now.

Single key reply

So far, we have been putting in Y or N replies with the use of INPUT, which means pressing the key and then pressing RETURN. This has the advantage of giving you time for second thoughts, because you can delete what you have typed and type a new letter before you press RETURN. For snappier replies, however, there is an alternative in the form of INKEY$. INKEY$ is an instruction that carries out a check of the keyboard to find if a key is pressed. This checking action is very fast, and normally the only way that we can make use of it is by making the INKEY$ instruction repeat until a key is pressed. This is possible by making a test of the variable to which INKEY$ makes an assignment. The syntax of INKEY$ is always of the form: `K$=INKEY$`, so that string variable K$ carries whatever has been assigned to it by INKEY$. If you didn't happen to hit a key just at the time when INKEY$ was being executed, then K$ contains a blank string. You

can make INKEY$ repeat, now, by testing to see if the string is blank and repeating the INKEY$ step until it is not blank. This introduces two important new topics: tests on variables and the use of repetition – of which there will be a lot more in chapter.5. Figure 3.14 shows such a test and repetition being used to produce a 'wait until ready' effect. The repetition is forced by the test part of line 30 that reads:

```
IF K$=""THEN 3Ø
```

and means as it says that if K$ is blank then start line 30 all over again. This is one obvious exception to the rule that the machine carries out your orders in the order of the line numbers, and it's an exception of a type that we shall not use too much, because there are better methods. The 'blank string' is produced by typing two quotes, with nothing between them. Each time the computer deals with INKEY$, it searches the keyboard to find if any key is pressed. If none is, then INKEY$ is a blank, and line 30 is repeated. When you press a key, however, the repetition stops (we say that the loop is broken), and the program moves on. This is useful to have at the end of a set of instructions. The user then has as much time as is needed to read the instructions, and can press any key to start things happening. As usual, 'any key' really means any character key, because several keys, such as SHIFT and CTRL have no effect, and STOP, like ALTC, will stop the program.

The INKEY$ instruction will produce a string quantity when any key is pressed, so we must assign INKEY$ to a string variable, K$. In this way, when any key is pressed, the quantity that it represents will be assigned to K$, and we can then test this string as we want. Figure 3.15 shows INKEY$ being used in this way to get a 'Y' or 'N' answer, with a form of test for incorrect answers incorporated. In line 30, INKEY$ is continually being tested. Only when a key is pressed will K$ have a value that is not blank, and the repeating action will then be broken. The value of K$ is then tested *again*, to see if the answer is acceptable. This test is quite strict. Up until now, you have been accustomed to using upper-case and lower-case letters almost interchangably, but you can't do this in such a test. If you type y or n, then this is detected as an error, because the program has tested for Y or N, and the two are not taken as being identical. If this might be a nuisance, you can get around it simply by adding a line:

```
35 K$=UPPER$(K$)
```

which will convert any lower-case letter stored as K$ into its upper-case equivalent.

Automatic definition

So far, we have defined what variable type we use as we have come to it, as part of its assignment. For example, if we use K=2, then K is a simple number variable, and if we use K$="NAME", then K$ is a string variable. BASIC is one of the few languages that does this, and in other languages it's much more common to have to define in advance what each letter will mean. Mallard BASIC allows you to get the best of both worlds by defining the meaning of the first letter of each variable name. Take, for example, the

two instructions DEFSTR and DEFSNG, which mean respectively, define string and define single-precision number. If you start a program with DEFSTR A-Z, you mean that any letter in this range will be the start of a string variable. In other words, all your variable names are string variables, and you don't have to mark them with the dollar sign. After DEFSTR A-Z, you can use variables called A, NAME, ST, ADR or whatever you like (but not reserved words), and they will all be taken as strings. If you don't want all of your variables to be strings, you can use a more restricted range, like DEFSTR A-H, or even a few letters, such as DEFSTR A,C,G. The use of DEFSNG will similarly make any name starting with the letters that you specify be the name of an ordinary number variable. Mostly, this isn't of interest, because this is the default, a name with no marker such as a dollar sign is taken as being an ordinary number. As we'll see later, however, there are two other forms of number variables that we may need to use that need different marking, and which can make use of DEFINT and DEFDBL if we want to arrange this sort of thing in advance. Remember that these instructions refer to the first letter of the variable, so that using DEFSTR A means that apple, APRICOT, AT as well as plain A will all be taken as string variables. Incidentally, the word WRITE can be used in place of PRINT, but with the odd effect that all string items are shown placed between quotes. This makes it impossible to confuse the printing of a number in string form with a number in number form.

Finally, it's as well to know how to reverse an assignment. When you make an assignment, such as A$="Name", whether it's inside a program or outside (as a direct command), then the assignment sticks until something is done about it. One thing you might do is to load in another program, using LOAD"name", which will automatically clear out all the variables of a previous program. Another method is to type NEW and press RETURN. The variables are also cleared out when you RUN the program again. A less drastic method is to type CLEAR, and then press RETURN. This will remove all variable asignments, clearing the memory of these assignments, but leaving your program in memory. Yet another option is a command COMMON RESET – but that one belongs to a book on more advanced programming.

Chapter 4

Working with numbers

Operators

An operator is a symbol for a fundamental mathematical operation. If that looks intimidating, don't worry, because the main operators that you are likely to use are the familiar signs `*`, `/`, `+` and `-` which carry out the fundamental operations of multiply, divide, add and subtract. Each of these is an operator that requires two numbers to work on, and in some books you will find them called *binary operators*. The numbers (or in some cases strings) that operators work on are called *operands*. The operators of Mallard BASIC are of four kinds, classed as *arithmetic*, *string*, *relational* and *logical*. The first group is composed of the four symbols that we are familiar with, plus the exponentiation sign, which appears on paper as ^ (the 'caret' sign) but on the keyboard and on the screen appears as ↑ (an up-arrow). The only string operator is the +, which will concatenate strings (join them together). In addition to these familiar tasks, you can use the + and – signs as *unary* operators, meaning that they can be used on a single number. You can, for example, write things like `+2.54` or `-3.6`, and these are examples of making use of the + and – operators in a unary way.

The relational operators are the set of signs that describe relationships, rather than producing an answer. The main three signs in the group are `=` (equal to), `<` (less than) and `>` (greater than), which compare the size of numbers and the ASCII codes of string characters. These signs can be combined, so that `>=` means equal to or greater than, `<=` means equal to or less than, and `<>` means not equal to. These are read left to right, so that `A>B` means 'A greater than B', and `X<Y` means 'X less than Y'. Note that you can write `<>` as `><`, reversing the position of the symbols, and you can also use `=<` or `=>`, but it's not a good habit to get into, because very few other machines will accept these combinations. You would, of course, use these operators in connection with variable names for numbers or strings rather than with number values (constants).

Finally in this list, the logic operators are the words AND, OR and NOT, sometimes called the Boolean operators in honour of the mathematicial genius George Boole whose work laid the foundations of computing science in the 1840's. The action of a logic operator is to *return* a value of *true* or *false* when it is used to test a relationship. The NOT operator is unary, and it gives a *true* result if what it precedes is *not true*. The machine expresses *true* as the number –1 and *false* as 0. If you are not accustomed to this, it can look very confusing, and a few examples will help. The secret is to work in terms of *true* and *false* only, and to start with any term that is enclosed in brackets. For example, what do you expect from the line:

```
PRINT NOT (2>1)
```

when this runs? The answer is worked out by looking first at 2>1, which is *true*. *Not true* is *false*, so the answer must be the code for *false*, which is 0. Try another one:

```
PRINT NOT ("B">"A")
```

(the quotes are important). When we compare strings, the ASCII codes count, and the ASCII code for 'B' is 66, more than the code for 'A' which is 65. `"B">"A"` is therefore *true*, and *not true* gives *false*, 0. By the same token, `NOT ("A">"B")` gives –1, *true*. Using `PRINT NOT (Ø)` will give `-1`, since *not false* must be *true*, and equally obviously, `NOT (-1)` gives Ø. If your nerves are up to it, try `PRINT NOT (7)` and see if you can explain the result. If it's baffling, please turn to Appendix F. In the normal course of programming, you should not have to be worried too much by this kind of thing, but it's as well to know, because it can sometimes make easy meat of what appears to be a difficult piece of programming.

The other logic operators, AND and OR each need two quantities to work on. These quantities can be number comparisons or string comparisons, and the important point once again is that each side of the AND or OR word should be something that can be resolved to *true* or *false*. For example, if we have:

```
PRINT (7>3)AND (5>2)
```

then we can expect the result –1. Why? Working out the items in the brackets we have 7>3 is *true* and 5>2 is *true*, so *true* AND *true* = *true*. The law of AND is that the result is *true* only if the items that are connected are also both *true*. If one item is *false*, the result is also *false*. On that basis, then you would expect the result of:

```
PRINT (5>6)AND (7>4)
```

to be 0, as it is because one term is *false*. The OR operator will give *true* if any one item is *true*, no matter whether the other is *true* or *false*. Only if both items are *false* will the result of OR be false. Figure 4.1 summarises the actions of AND and OR. Remember that you would normally be using these operators with variable names.

AND

TEST 1	TEST 2	TEST 3	RESULT
True	True	True	True
False	True	True	False
True	False	True	False
True	True	False	False
any other combination			False

OR

TEST 1	TEST 2	TEST 3	RESULT
False	False	False	False
True	False	False	True
False	True	False	True
False	False	True	True
any other combination			True

Form of use

```
(TEST1) AND (TEST2) AND (TEST3) . . .

(TEST1) OR  (TEST2) OR  (TEST3) . . .
```

Figure 4.1 Summarising the actions of AND and OR in tests.

Expressions

An expression is a set of operators and operands that provides a number or a string result. A very simple expression is `1+2`, but we usually reserve the term for lines that make use of variable names, and in which more than one operation may be carried out. A familiar pair of expressions are the incrementing expression, `X=X+1` and the decrementing expression `X=X -1`. The = sign in BASIC is used here to mean *becomes* rather than *equals*, and this is one respect in which BASIC can be very confusing to the beginner – other languages use different signs to distinguish between equality and assignment. The expression `X=X+1` therefore means that the value of X is incresed by one, and `X=X -1` means that the value of X is decreased by one.

In general, the use of more complicated expressions is something that often proves baffling to a computer user who has no experience of mathematics. This needn't be, because expressions, like anything else in computing, follow precise rules, and once you know what the rules are it's not difficult to apply them. The most important rules concern *precedence* of operators, and once you know about precedence, it's not difficult to find what an expression does. Making up an expression for yourself is another matter, and only practice can help there. The table of precedence is shown in figure 4.2. What this means is that if you have more than one operation in an expression, the operation(s) with higher precedence are carried out first. If there is no clear precedence, then the order in the expression is simply left to right. For example, if you have the expression:

```
PRINT 5+4*3-6/2
```

what do you expect? If everything obeyed a left-to-right order only, the

Order of priority

For ordinary arithmetic, order of priority is MDAS – Multiplication and Division, followed by Addition and Subtraction. The full order of priority is:

1. Brackets
2. Raising to a power, using ^
3. Unary + or –
4. Multiplication and division.
5. Integer division.
6. Integer MOD
7. Addition and subtraction.
8. Comparison, using = < >
9. AND
10. OR
11. NOT

Figure 4.2 The order of precedence for simple arithmetic actions and the logic actions.

result would be got from $5+4=9$, $9\times3=27$, $27-6=21$ and $21\div2=11.5$. It's not like this, though. Because multiplication and division have higher precedence than addition and subtraction, the `4*3=12` and the `6/2=3` are worked out first. Having done that, all that is left is of equal precedence, and we get `5+12=17` and `17-3=14`, which is the answer that the computer will give you. Remember that in a program, all or most of the quantities would be variables, and to find the numerical answer you would have to find what numbers were assigned to the variables at the time of evaluating the expression. You might, for example, be working with something like `Y=K+B*X^N`. The X^N action is carried out first, since the raise-to-a-power action (exponentiation) has highest precedence, and then the result of this is multiplied by B. Finally, the value of K is added. Precedence rules, O.K?

One important point to remember is that brackets take precedence over everything else. For example, if you have an expression which boils down to $5\times(4+3)$, then this is *not* the same as $5\times4+3$ (which is 23) but gives 35, because whatever is inside the brackets is worked out first, giving 12 in this case. When there are several sets of nested brackets, meaning brackets inside other brackets, then whatever is innermost has highest precedence. For example, $5\times(4+(8-6/2))$ gives 45, because the innermost bracket gives 5, adding this to the 4 in the next layer of brackets gives 9, and the result is 5×9. For some reason, however, it all looks much more fearsome when used with variables, particularly when there are actions like STR$ and VAL (see chapter 5) involved as well.

Translating formulae

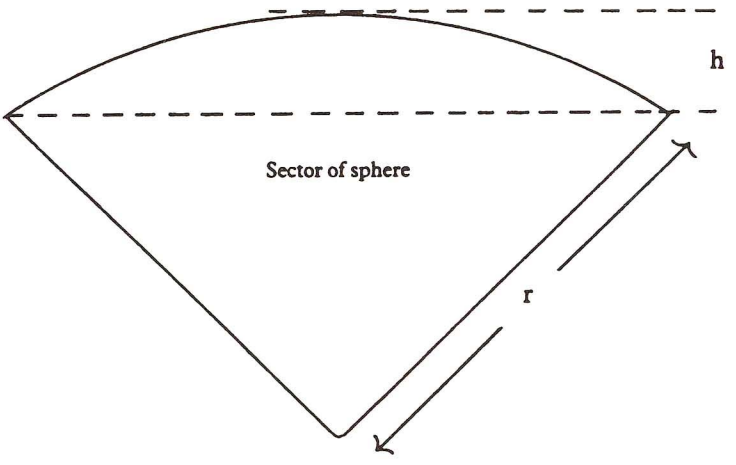
The earliest computer programming languages were for scientific and engineering use, and translating formulae so that the computer could deal with them was a very important feature. It was so important, in fact, that one of the main languages in the early days was called FORTRAN, an abbreviation of FORmula TRANslation. FORTRAN is still used, and the BASIC language which is used by all microcomputers is based very considerably on the ideas and methods of FORTRAN. For this reason, BASIC is a language not to be despised if your needs or interests are in programming of this type. A lot of languages that are more highly regarded either by academics or for business use are inferior to BASIC when it comes to working with formulae, and also, incidentally, for dealing with strings and disc files. If you haven't had some practice, however, it's not always straightforward to convert a formula written in a reference book into the form of a BASIC expression. For one thing, you have to remember that the order in which the terms of the formula are written will not usually be the order in which you want them evaluated, so that you must either change the order or make use of brackets to obtain the correct expression.

Examples help here, but one person's example is another's confusion, so please bear with me if the formulae that you want to use are not shown here. Remember that you don't have to derive the formulae for yourself for most purposes, you simply take them from a reference book. You need to know, of course, what variable values have to be supplied, and what the formula does, and you also need to know any limitations, but in general this is all. You get into a different league when you start to generate your own formulae! We'll take as a first example the formula for the volume of a sector of a sphere, shown in figure 4.3a. Now, like many formulae, this uses no sign for multiplication. Quantities that are printed together are intended to be multiplied, so that the formula requires you to multiply *2* by π by the value of r (squared) by h, and then divide the answer by *3*. Note that this uses the variable `pi`, and you would assign this value in a line such as:

```
PI=3.141592
```

so as to get the number into the form of a variable. If you are likely to be using numbers like this, even straightforward numbers like 100 and 1000, then it's always best to assign them to a variable name. The reason is that this makes a program run faster. When PI = 3.141592 is executed, the number is converted form the form that you see it into a binary code, and this takes time. Once you have done this, though, it stays in binary form, and you can then use `PI` as much as you want. If you use expressions like `R*3.14159`, then the conversion has to be carried out each time a number appears, and that might be many times.

Now in the expression, there is one power taken, and this action will have precedence no matter where we put it. It makes sense, in any case, to start with this item, getting the value of r squared. Suppose that we use variable names `R` and `H`, then the expression that is shown in figure 4.3b is the BASIC expression for evaluating the formula. The variable `V` is used for volume, and the main point to note is that we have to insert the multiplication signs (`*`) that BASIC demands, and the division sign `/`. Because all the operations apart from finding the square are of equal



Formula is: $V=\frac{2\pi r^2 h}{3}$ (a)

in BASIC: `V=R↑2*2*π*H/3` (b)

Figure 4.3 Converting a formula into BASIC, in this example the formula for the volume of a sector of a sphere.

precedence, we can write the rest of the expression in left-to-right order, and be reasonably confident. In all cases, however, if you are in any doubt, try a few examples with simple numbers and check that you get what you expect. In this example, the answers will always have more places of decimals that you would really want to use, and we'll look at how to round them off later.

The real problems come when the formula is not in the form that you want. If you have a smattering of algebra (ie. if you are over the age of 40) then you may be able to rearrange the formula to suit.

Functions

There are many quantities that we need to calculate which cannot be dealt with by an operator, or even by a reasonably simple expression. Quantities such as trigonometrical ratios, square roots, hexadecimal equivalents and so on are dealt with by the use of *functions*. A function of a number is a quantity that is obtained by the use of various actions on the number. The number (or more likely, variable) is called the *argument* of the fuctions, and for many functions has to be enclosed in brackets. In the computing sense, functions can also include actions on strings, and the main thing that they

ABS (X)	Converts negative sign to positive.
ATN (X)	Gives angle (in radians) whose tangent is X.
CINT (X)	Converts X to an integer. This will be rounded if X contains a fraction.
COS (X)	Gives the cosine of angle X (radians).
CSNG(X)	Converts value to single-precision size.
EXP (X)	Gives the value of e to the power X.
FIX (X)	Strips fraction from X.
FRE (X)	Gives amount of memory not in use or reserved.
HEX$	Converts number into hex (base 16).
INT (X)	Gives the whole-number part of X, rounded to the nearest smaller whole number.
LOG (X)	Gives the natural logarithm of X.
LOG10(X)	Gives the logarithm to base 10 (common log) of X.
MAX (X,Y,Z,...)	Gives the number which is the greatest in a list.
MIN (X,Y,Z,...)	Gives the number which is the minimum in a list.
OCT$(X,n)	Converts number X to octal string of n characters.
POS	Gives value related to position of cursor. LPOS is corresponding function for printer.
RANDOMIZE (X)	Sets new sequence of 'random' numbers.
RND (X)	Gives a random fraction between 0 and 1.
ROUND (X,Y)	Rounds X to the number of places specified by Y
SGN (X)	Gives the sign of X. The result is +1 if X is positive, −1 if X is negative, 0 if X is zero.
SIN (X)	Gives the sine of angle X (radians).
SQR (X)	Gives the square root of X.
TAN (X)	Gives the value of the tangent of angle X (radians).
UNT(X)	X must be between 0 and 65536. UNT converts it into a number between −32768 and +32767.

Figure 4.4 Number functions, with brief notes. Don't worry if you don't know what some of these do. If you don't know, you probably don't need them!

have in common is that the function uses a statement word (not a symbol as an operator uses) and that it needs an operand or argument, which can be a number or a string depending on the type of function. The main number functions are listed in figure 4.4, along with their effects. Of this list, the trigonometric functions merit particular attention, because they cause a lot of trouble to programmers who are working with trigonometrical formulae. These formulae are often the simplest way of obtaining some graphics effects, and to draw graphs. The functions that are most used are SIN, COS and TAN, all of which are provided in Mallard BASIC, though graph-plotting is not (but see *Personal Computer World*, July 1986, page 190). What you need to watch, however, is that each of these functions

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 X%=7:Y%=3
30 PRINT"Integer quotient is "X%\Y%
40 PRINT"and remainder is "X% MOD Y%
```

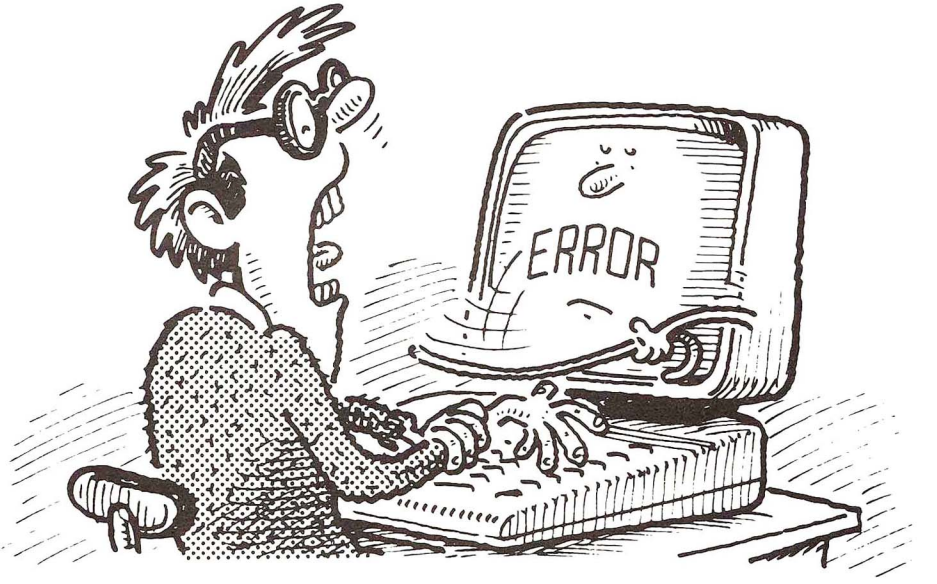
Figure 4.5 Integer quotient and remainder. Integer division gives an integer answer only, with no fraction shown.

needs an argument which is an angle in radians. Now if you are working with angles in degrees you will need to convert from degrees to radians, and there is no conversion function built into Mallard BASIC. One solution is to convert from degrees to radians. If you example, you have an angle in degrees in the form of variable D, then the conversion to radians takes the form of:

```
R=D*PI/180
```

with PI taking its usual value, assigned earlier in the program. Another type of solution is shown at the end of this chapter, in the form of a defined function, which is a do-it-yourself type of function.

The reverse problem occurs when you need to use the only inverse trigonometric function, ATN. This finds the angle (in radians) whose tangent has the value which is used as the argument. This is called an inverse function because it finds the angle rather than the function of an angle. ATN is a function which you don't need all that often, and a lot of work with trigonometry calls for the inverse SIN (Arcsin) and inverse COS (Arccos) rather than the ARCTAN. We'll look later at methods of obtaining these quantities which you are likely to need if you are programming for surveying or other geometrical work.



The other function which can cause problems is the LOG function. When you use a statement such as A=LOG (X) then what is assigned to A is the natural log of X. In textbooks, this type of logarithm is usually written as LN rather than as LOG, and the use of LOG in BASIC can be a source of considerable confusion. This is compounded by the fact that some varieties of BASIC use both LN and LOG, with their correct meanings. If you are using a function which requires the natural log, then all is well – and remember that the 'antilog' function is EXP. It's just as likely, however, that you will need to use a base-ten logarithm. One good example is in calculating decibel ratios in electronic engineering. Mallard BASIC copes with this by another function, LOG10, which gives the base-ten logarithm. Schools nowadays don't teach about logarithms, because teachers don't seem to know to what extent logarithms are used in engineering and scientific work – it's yet another reason for firms setting their own entrance exams.

Precision of numbers

One problem that turns up time and time again in computer work is precision of numbers. For some reason, computers are always thought of as being associated with mathematics, and people believe that computers can carry out arithmetic much more precisely than your average £3.50 pocket calculator. Don't you believe it! This is something that causes more trouble than anything else, and it arises because of the way that computers store numbers in the memory. Mallard BASIC allows for numbers to be stored in three different ways, called integer, single-precision, and double-precision. Up to now, we have made use of ordinary single-precision numbers without saying much about it, but now the time has come to explain more, starting with integers.

An integer means a whole number, but for the purposes of Mallard BASIC, it has a more restricted meaning of a whole number whose range is from –32768 to +32767 only. Now a number in this range can be stored in two of the memory units that we call bytes, and its value will always be precise. If you want to specify a variable as an integer, you use the % sign to mark it out, or you can use DEFINT to specify a range of letters that will be the first letters of integer variables. Because an integer requires only two bytes for storage, the use of integers will increase the running speed of a program, and we shall see shortly that this can be turned to a considerable advantage. If you can perform all arithmetic with integers, it will be fast and, with one exception, precise. The exception is division, because an integer number cannot be assigned with a fraction. If, for example, you assign A%=B%/C%, with B%=5 and C%=3, then A% will be 1, not 1.66666667, which is what you would get from PRINT 5/3. Mallard BASIC has special integer division and remainder operators, using the symbol \ (use keys EXTRA 1 / 2) for integer division, and MOD (short for modulus) for the remainder. Figure 4.5 shows this in action, with two numbers being assigned, and the integer division and remainder being found. The existence of these operators allows us to program with integers even if division is involved.

```
10 A=1/11:B=7/11:C=6/11
20 PRINT"A is "A:PRINT"B-C is "B-C
30 IF A=B-C THEN PRINT"EQUAL" ELSE PRINT"NOT
 EQUAL!"
40 IF ROUND(A,4)=ROUND(B-C,4)THEN PRINT"TRUE
 NOW"
```

Figure 4.6 Precision of arithmetic. Numbers printed on the screen have been rounded, but when two variables are compared, the numbers are not rounded, and errors can show up. The effect of ROUND is to force rounding so that such tests can be made.

You should use integer variables when:

1. The number range that you are likely to work with is small and does not require fractions.

2. The number variable is used many times, particularly for a constant value.

3. The number variable is used in expressions, particularly in loops (see chapter 5).

By keeping to these rules, you can make your programs run faster and take up less of the memory. When you first start to program in BASIC, these points aren't exactly the most pressing ones for you, but later you'll need to know and make use of these points. From now on, if it's going to be an advantage to work with integers, the examples in this book will show them in use.

Real Numbers

When you need to work with real numbers, meaning numbers which can be positive or negative, whole or fractional, and with a much greater range of size, precision then becomes a problem. In a lot of applications, we use real numbers in standard form, which means that a number such as 52400 would be written as `5.24E4`. When this is done, numbers are often approximated, so that the number 52417 might also be written as `5.24E4` on the grounds that the extra 27 represented only a small fraction of the whole number. To put it another way, the precision of the number is sacrificed so that it can be written with only three digits before the E sign. This part of the number is called the *mantissa*, and the part which follows the E is called the *exponent*. When a real number is stored in the computer, it is also stored in mantissa/exponent form, but with both in binary numbers, using the digits 1 and 0 only. The mantissa is a binary fraction, stored in three bytes, and the mantissa is a single byte integer. This is a compact way of storing numbers, but it does mean that there will always be some approximations. Figure 4.6 illustrates this, using assignments to three variables and a simple equation which should produce identical results because `1/11=7/11-6/11`. Line 20 shows that the two numbers, `A` and `B-C` give the same result on the screen, appearing in the form `9.Ø9Ø9Ø9E-2`. Because of the approximations that have to be made in storing the variables,

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A=22/7:B#=22/7
30 PRINT"A is "A
40 PRINT"B# is "B#
```

Figure 4.7 Using single and double-precision number variables.

however, the two are not identical, as the test in line 30 reveals. This difference can be a problem, because the rounding that is carried out on the numbers when they are displayed on the screen is not done when they are simply compared, and this type of thing is the result. Line 40 shows the answer in the form of the ROUND function. ROUND is followed by a number within brackets, with the number of decimal places following the number, separated by a comma. If you miss out the number of decimal places, you will round to an integer, which is not always what you want!. In this example, using items like `ROUND (A,4)` will round to four places of decimals. By using ROUND on each of the results that we are comparing, we can make the result of comparison as *true* as the result of seeing the answers on the screen. Note that you must not round everything. If you are comparing two items, then there should be only two ROUND statements. If you ROUND every part of an expression, you are likely to land up with even more problems. The sensible use of ROUND, however, can solve a lot of problems that otherwise can result from the use of single-precision numbers. If, for example, you are writing an accounts program in which two quantities are expected to balance, then rounding will be essential. This is because you will be working with single-precision numbers which include two places of decimals, and because of storage errors, there may be small fractional errors. If you are displaying your results only, these fractions are of no importance and do not appear, but they can cause any comparison to be incorrect. Two columns of figures, for example, which should give the same amount, will not necessarily give a *true* answer in a line such as:

```
IF T1=T2 THEN PRINT "BALANCES"
```

unless ROUND has been used.

Double precision

Unusually among modern versions of BASIC, Mallard BASIC offers the use of double-precision numbers. This is not a new idea, and the first home computer I ever used featured this, but it's not seen to such an extent now. A number will be stored in double precision form if you mark its variable with the # sign, or use DEFDBL for the letter that starts the name of the variable. The most noticable point about double-precision numbers is that they print out with many more decimal places than single-precision. Figure 4.7 illustrates this, showing that when the same fraction is assigned to two variables, the single-precision variable prints out with six places of decimals, and the double-precision variable prints 15 places of decimals. This added precision is seldom needed, and is obtained by using additioal storage for each double-precision variable. A double-precision variable takes 8 bytes to store, as compared to 4 for single-precision and 2 for an integer. The extra precision of these numbers is likely to be useful only if

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A#=22/7
30 PRINT"A# is "A#
40 A!=A#
50 PRINT"A! is "A!
60 A%=A#
70 PRINT"A% is "A%
```

Figure 4.8 Deliberate conversion of variable types.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A#=22/7
30 B!=6.718345
40 C%=45
50 PRINT C%+B!
60 PRINT C%+A#
70 PRINT B!+A#
```

Figure 4.9 Automatic conversion in action – the number printed is always the most precise of the types that have been used.

you are working with data for scientific or engineering purposes which for some reason has to be measured with very high precision. The vast majority of applications need nothing like this, and even for the purposes that use double-precision, it's likely that the numbers will be rounded to four places of decimals at some stage. Using double-precision variables makes your programs run slowly and the numbers take up more space in the memory. Use them only if you must.

Number roundup

Working with numbers implies the input of numbers from the keyboard, processing, and the display of numbers on the screen. As far as input is concerned, the conventional method is the statement of the form `INPUT A` (single precision), `INPUT A%` (integer), or `INPUT A#` (double precision). On Mallard BASIC, you will always get a `Redo from start` message if the number that you enter is incorrect, such as 24.5 for an `INPUT A%`, or an incorrect range for an integer. Using a number variable, incidentally, does not mean that you cannot enter *any* letters, because you can always enter a number in a form such as `1E3` when the entry is to a single-precision number, or `1D3` for a double-precision number. You are not, however, permitted to enter a number in this form to an integer variable.

One principle that is employed by a lot of programmers is to use a string for entry, such as `INPUT A$`. No entry, unless it's a string of ridiculous length, will be rejected, and it's then easy to check what has been entered and issue messages about errors. It's also easy to convert a string form of number into

number form, using STR$, dealt with in chapter 6. Before we get as far as that, however, we need to see what happens when numbers are converted from one form to another. This can be done either deliberately or by the automatic action of Mallard BASIC, and you need to know what is likely to happen and when. Figure 4.8 shows deliberate conversion, which works pretty much as you would expect it to. When you assign the content of a double-precision number to a single-precision variable, the number is chopped down to single-precision, and can be further chopped down to integer if you assign it to an integer variable. That's hardly surprising – but note that the three variables A#, A! (or A) and A% are regarded as being completely separate. You *can* run into trouble with conversion to an integer if the number that you convert is outside the integer range of –32768 to +32767. What is sometimes less expected is how automatic adjustments are made to variables. Take a look at the listing in figure 4.9, for example. In this listing, three variables are assigned, and then combinations are added. Adding an integer to a single-precision number results in a single-precision result. Adding a single-precision number or an integer to a double-precision number gives a double-precision result. In other words, the result that you see on the screen is always at the greatest precision that has appeared in the expression. If there is one double-precision number in an expression, then each other number in the expression will automatically be converted to double-precision before being used. This conversion can be comparatively slow, and you will need to watch for this kind of thing going on if you have mixtures of types like this.

Defined functions

Mallard BASIC allows you to define and use functions of your own to add to the built-in ones. These are very reasonably called defined functions, or user-defined functions, and they take a form that is rigidly fixed. To start with, a function must be defined by a line that starts DEF FN. This line must be put in *before* you want to use the function, and what follows must be contained in one line. Immediately following the 'N' of DEF FN you need a name for the function, then a list of the variable types that it will use within brackets. This has to be followed by an equality sign, then the function action with the two variable names that have been used in the brackets. One example is worth a thousand words here, so consider a very simple defined function:

```
DEF FNSum (A,B)=A+B
```

The name of the function is Sum, and it would be called into action by using FNSum. The A and B are single-precision number variables, and they do not have to be used anywhere else. Following the equality sign, A+B shows what we want done with the numbers – they are to be added. That's all, but the real meat is revealed when the function is called into action. You would do this when you had two variables, and you wanted to make use of their sum. Suppose that the variables were X and Y, then you could obtain the sum by using:

```
PRINT FNSum (X,Y)
```

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 DEF FNDif(A,B)=A-B
30 X=67.6:Y=42.1
40 PRINT"X-Y is "FNDif(X,Y)
50 PRINT"Y-X is "FNDif(Y,X)
```

Figure 4.10 Using a very simple defined function. Note the effect of the order of variables.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PI=3.141592
30 DEF FNRD(D)=D*PI/180
50 PRINT"Cos of 30 degrees is "COS(FNRD(30))
60 PRINT "Tan of 63.5 degrees is "TAN(FNRD(63
 5))
```

Figure 4.11 Using a degree to radian function.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PI=3.141592
30 DEF FNDG(radians)=radians*180/PI
40 INPUT "Size of tangent ";J
50 PRINT"Angle in degrees is "FNDG(ATN(J))
```

Figure 4.12 A radian to degree function which is useful along with ATN.

or

```
Z=FNSum (X,Y)
```

and there are several points to note here. One is that we can use X and Y in the brackets, not A and B. When you define a function like this, you define only the type of numbers (or strings) that you will use and what will be done with them. When the action is called for, all you need to specify is variables of the correct type *and in the correct order*. This is illustrated by figure 4.10, in which the function is called twice, with the order of variables reversed. The function subtracts the second variable from the first, and it's up to you to put the variables into the correct order.

One very important point is that the DEF FN part must always come early in a program, certainly before you try to use the function with an FN call. If you omit the DEF FN part, or put it later, you will get an error message about an undefined function. You should use the defined function for actions that would be repeated several times with different variable names, and which would involve a lot of typing. The names that you use in such a definition need not be used anywhere else, and you could use long names that reminded you of what was to be done. Using long variable names in a defined function is not such a strain on your typing finger(s), because they

have to be typed only once. We'll close this chapter with some illustrations that should be helpful, showing both number and string defined functions in use.

To start with, the example in figure 4.11 makes use of the relationship between radians and degrees, which is that pi radians are equal to 180 degres. By using this function, then you can make the conversion that is necessary for the functions, COS, SIN and TAN. In the example, the figures for degrees have been put in directly as numbers into the FNRD part, but these could have been variable names, as is more usual. Figure 4.12 shows a defined function which is useful for conversion from radians to degrees, which is useful for the ATN function. In this example, the variable name of radians has been used in the function as a reminder of what it is about. You can also use longer function names, such as FNSum.of-.squares, as reminders. We'll look at the use of defined functions with strings later in chapter 6.

Chapter 5

Getting repetitive

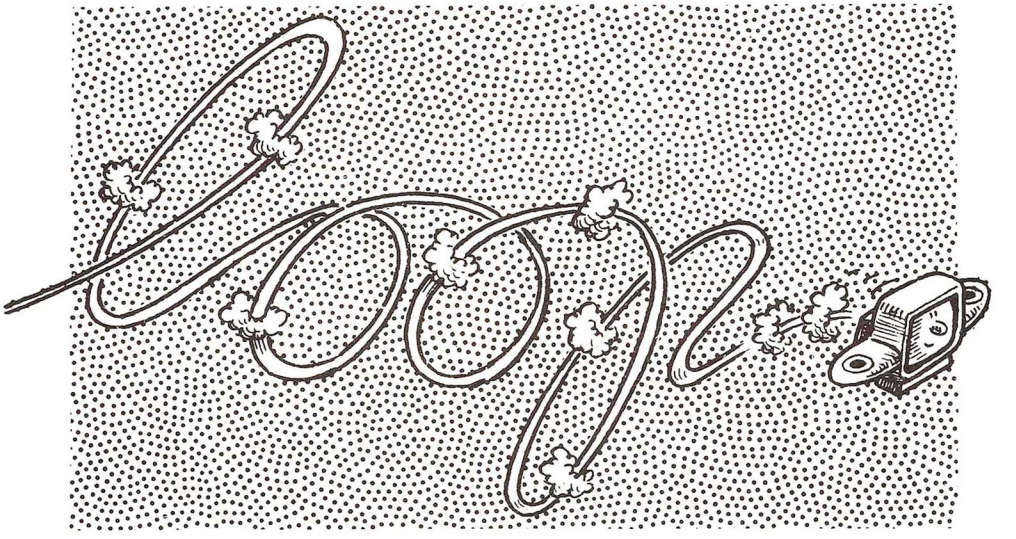
One of the activities for which a computer is particularly well suited is repeating a set of instructions over and over again and every computer language is equipped with commands that will cause repetition. Mallard BASIC is no exception to this rule, and it is equipped with more of these 'repeat' commands than is usual for BASIC. We'll start with one of the simplest of these 'repeater' actions, GOTO.

GOTO means exactly what you would expect it to mean – go to another line number. Normally a program is carried out by executing the instructions in ascending order of line number. In plain language that means starting at the lowest numbered line, working through the lines in order and ending at the highest numbered line. Using GOTO can break this arrangement, so that a line or a set of lines will be carried out in the 'wrong' order, or carried out over and over again. The command word GOTO has to be followed by a space and then a line number, and you will get an error message if the space is omitted. We have already seen this used in connection with INKEY$.

Figure 5.1 shows an example of a very simple repetition or 'loop', as we call it. Line 10 clears the screen, and line 20 assigns a starting number to an integer variable. Lines 30 and 40 contain simple PRINT instructions, and in line 50, the value of the integer N% is incremented. When line 50 has been carried out, the program moves on to line 60, which instructs it to go back to line 30 again. This is a never-ending loop, and it will cause the screen to print the words :

```
Mallard BASIC fills your screen!
```

and the number that is held in N%, each time the computer goes through the actions of the loop. We call this 'each pass through the loop'. This continues until you press the STOP key to 'break the loop', or until the value of N% exceeds 32767. Any loop that appears to be running forever can be stopped by pressing the STOP key. This does what it says, stops the program running, but not completely. If you have stopped a loop from running in this way, then you can type CONT (and press RETURN) to make



the program continue from where it left off. We'll see later that this is very useful if you are chasing faults in a program. If you want to stop the program completely, then you just don't type `CONT`!

Now an uncontrolled loop like this is not exactly good to have, and GOTO is a method of creating loops that we prefer not to use! The nasty thing about using GOTO to form a loop is that you have nothing to mark the start of the loop. You can see where the end of the loop is, because that's where the GOTO is, but to find where the start is, you have to read the number that follows GOTO. Because of this, it's all to easy to get a GOTO wrong – try for example the effect of using GOTO 20 or GOTO 40 in this simple example. This is more likely when you are working with a long program, and you can't see on the screen the line that you want to go to. A lot of varieties of BASIC offer little in the way of an alternative, but Mallard BASIC offers two, and one of them is the FOR..NEXT loop. As the name suggests, this makes use of two new instruction words, FOR and NEXT. The start of the loop is marked with the word FOR, and the end with the word NEXT. The instructions that are repeated are the instructions that are placed between FOR and NEXT. Figure 5.2 illustrates a very simple example of the FOR..NEXT loop in action. The line which contains FOR must also include a number variable which is used for counting, and numbers which control the start of the count and its end. In the example, `N%` is the counter variable, and its limit numbers are 1 and 10. The NEXT is in line 40, and so anything between lines 20 and 40 will be repeated. We have made N% an integer variable, because its values are all integers. You can, if you like, remind yourself of the variable name at the end of the loop by using `NEXT N%`, but this is not absolutely necessary.

As it happens, what lies between these lines is simply the PRINT instruction, and the effect of the program will be to print `Mallard BASIC beats the lot!` ten times. At the first pass through the loop, the value of N% is set to 1, and the phrase is printed. When the NEXT instruction is encountered, the computer increments the value of N%, from 1 to 2 in this

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 N%=1
30 PRINT"Mallard BASIC fills your screen!"
40 PRINT"Number is"N%
50 N%=N%+1
60 GOTO 30
```

Figure 5.1 A *very* simple loop. You can stop this by pressing the STOP key, or by using ALTS.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 FOR N%=1 TO 10
30 PRINT"Mallard BASIC beats the lot!"
40 NEXT
```

Figure 5.2 Using the FOR NEXT loop for a counted number of repetitions.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 FOR N%=1 TO 10
30 PRINT"Count is ";N%
40 FOR J=1 TO 1200:NEXT
50 PRINT CHR$(27)+"E"+CHR$(27)+"H"
60 NEXT
```

Figure 5.3 A program that uses nested loops, with one loop inside another. The inner loop (in line 40) is acting as a time delay.

case. It then checks to see if this value exceeds the limit of 10 that has been set. If it doesn't, then line 30 is repeated, and this will continue until the value of N% exceeds 10 – we'll look at that point later. The effect in this example is to cause ten repetitions.

You don't have to confine this action to single loops either. Figure 5.3 shows an example of what we call 'nested loops', meaning that one loop is contained completely inside another one. When loops are nested in this way, we can describe the loops as inner and outer. The outer loop starts in line 20, using variable N% which goes from 1 to 10 in value. Line 30 is part of this outer loop, printing the value that the counter variable N% has reached. Line 40, however, is another complete loop. This must make use of a different variable name, and it must start and finish again before the end of the outer loop. We have used variable J (not an integer this time), and we have put nothing between the FOR part and the NEXT part to be carried out. All that this loop does, then, is to waste time, making sure that there is some measurable time between the actions in the main loop. By using a single-precision number, we make it easier to waste time! The last action of the main loop is clearing the screen in line 50, and the NEXT is placed in line 60. The overall effect, then, is to show a count-up on the screen, slowly enough for you to see the changes, and wiping the screen clear each time. In this example we have used NEXT to indicate the end of

each loop. We could use NEXT J in line 40 and NEXT N% in line 60 if we liked, but this is not essential. It also has the effect of slowing the computer down slightly, though the effect is not important in this program. When you do use NEXT J and NEXT N%, you must be *absolutely sure* that you have put the correct variable names following each NEXT. If you don't, the computer will stop with an Unexpected NEXT error – meaning that the NEXTs don't match up with the FORs in this case. If you had omitted a NEXT, you would get a NEXT omitted message, but it would refer to the line number in which the loop started.

Even at this stage it's possible to see how useful this FOR..NEXT loop can be, but there's more to come. To start with, the loops that we have looked at so far count upwards, incrementing the number variable. We don't always want this, and we can add the instruction word STEP to the end of the FOR line to alter this change of variable value. We could, for example, use a line like:

```
FOR N%=1 TO 9 STEP 2
```

which would cause the values of N% to change in the sequence 1, 3, 5, 7, 9. When we don't type STEP, the loop will always use increments of 1.

Figure 5.4 illustrates an outer loop which has a step of –1, so that the count is downwards. N% starts with a value of 10, and is decremented on each pass through the loop. Line 40 once again forms a time delay so that the countdown takes place at a civilised speed. This is a particularly useful way of slowing the count down. If we want to speed a loop up, the easiest way is to use an integer variable such as the N% in the main loop. If we do this, however, we can't use steps that contain fractions, like .1. You'll notice also that as each number is printed, the cursor appears. Many varieties of BASIC allow this to be suppressed, but this cannot be so easily done in Mallard BASIC, so you're stuck with it in this example – see later for one method.

Every now and again, when we are using loops, we find that we need to use the value of the counter, such as N% or J, after the loop has finished. It's important to know what this will be, however, and figure 5.5 brings it home. This contains two loops, one counting up, the other counting down. At the end of each loop, the value of the counter variable is printed. This reveals that the value of N% is 6 in line 50, after completing the FOR N% = 1 TO 5 loop, and is 0 in line 90 after completing the FOR N% = 5 TO 1 STEP -1 loop. If you want to make use of the value of N%, or whatever variable name you have selected to use, you will have to remember that it will have changed *by one more step* at the end of the loop. You can, of course, use negative values of N% in loops. If you use integer variables like N% or J% for speed, however, remember that there are limits to these values. You must not attempt to use an integer greater than 32767 or less than –32768. If you attempt to make the number go outside this range you will get an 'Overflow' error message

One of the most valuable features of the FOR..NEXT loop, however, is the way in which it can be used with number variables instead of just numbers. Figure 5.6 illustrates this in a simple way. The letters A%, B% and C% are assigned as numbers in the usual way in line 20, but they are then used in a FOR..NEXT loop in line 30. The limits are set by A% and C%, and the step is obtained from an expression, C%/B%. The rule is that if you have

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 FOR N%=10 TO 1 STEP -1
30 PRINT N%" seconds and counting"
40 FOR J=1 TO 1100:NEXT
50 PRINT CHR$(27)+"E"+CHR$(27)+"H"
60 NEXT:PRINT"Blastoff"
```

Figure 5.4 A countdown program, making use of STEP to decrement the number in each loop.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 FOR N%=1 TO 5
30 PRINT N%
40 NEXT
50 PRINT"N% is now "N%
60 FOR N%=5 TO 1 STEP -1
70 PRINT N%
80 NEXT
90 PRINT"N% is  now "N%
```

Figure 5.5 Finding the value of the loop variable after a loop action is completed. This is always one more step than the specified end of the loop.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A%=2:B%=5:C%=10
30 FOR N%=A% TO C% STEP C%/B%
40 PRINT N%
50 NEXT
```

Figure 5.6 A loop instruction that is formed with number variables.

anything that represents a number or can be worked out to give a number, then you can use it in a loop like this. Where you have to be careful is in the use of integers. If you programmed your loop as `FOR N%=A% TO B% STEP B%/C%`, for example, you would get the numbers 2, 3, 4, 5 printed. This is because, having chosen to work with integers, you can't have fractions. This means that $B\%/C\%$, which should be 0.5, is taken as 1!

Loops and decisions

It's time to see loops being used rather than just being demonstrated. A simple application is in totalling numbers. The action that we want is that we enter numbers and the computer keeps a running total, adding each number to the total of the numbers so far. From what we have done so far, it's easy to see how this could be done if we wanted to use numbers in fixed quantities, like ten numbers in a set. The program of figure 5.7 does just

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 Total=0
30 PRINT TAB(32)"TOTALLING NUMBERS PROGRAM"
40 PRINT:PRINT"Enter each number as requested."
50 PRINT"The program will keep a running total"
60 FOR N%=1 TO 10
70 PRINT"Number "N%" please ";
80 INPUT J:Total=Total+J
90 PRINT"Total so far is "Total
100 NEXT
```

Figure 5.7 A number totalling program for a set of ten numbers.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT TAB(35)"Another Total Finder"
30 PRINT:PRINT"The progam will total numbers
for you"
40 PRINT"until you enter a zero."
50 Total=0
60 INPUT"number, please ";N
70 Total=Total+N
80 PRINT"Running total is "Total
90 IF N<>0 THEN 60
100 PRINT"End of totalling"
```

Figure 5.8 A running-total program which can't use FOR..NEXT. Line 90 carries out the test which decides whether or not to loop.

Sign	*Meaning*
=	Exact equality
>	left hand quantity greater than right hand quantity.
<	left hand quantity less then right hand quantity.

The signs can be combined as follows:

<>	Quantities not equal
>=	LHS greater than or equal to RHS
<=	LHS less than or equal to RHS

Figure 5.9 The mathematical signs that are used with IF for comparing numbers and number variables.

this. The program starts by clearing the screen and setting a number variable `Total` to zero. This is the number variable that will be used to hold the total, and it has to start at zero. As it happens, Mallard BASIC arranges this automatically at the start of a program, but it's a good habit to ensure that everything that has to start with some value actually does. We couldn't, incidentally, use `TO` for this variable, because TO is a reserved word, part of the FOR..NEXT set of words. You will get a `Syntax error` message when the program runs if you have used a reserved word as a variable name.

Lines 30 to 50 issue instructions, and the action starts in line 60. This is the start of a FOR..NEXT loop which will repeat the actions of lines 70 to 90 ten times. Line 70 reminds you of how many numbers you have entered by printing the value of N% each time, and line 80 allows you to INPUT a number which is then assigned to variable name J. This is then added to the total in the second half of line 80, and the loop then repeats. At the end of each pass through the loop, the variable *Total* contains the value of the total, the sum of all the numbers that have been entered so far.

It's all good stuff, but how many times would you want to have just ten numbers? It would be a lot more convenient if we could just stop the action by signalling to the computer in some way, perhaps by entering a value like 0 or 999. A value like this is called a *terminator*, something that is obviously not one of the normal entries that we would use, but just a signal. For a number-totalling program, a terminator of 0 is very convenient, because if it gets added to the total it won't make any difference. We need, however, some way of detecting this terminator, and this is provided by the instruction word, IF that we have already seen used for tests.

IF has to be followed by a condition. You might use conditions like `IF N%=2Ø`, or `IF NM$="LASTONE"` for this purpose. After the condition, you can use the word THEN, and that has to be followed by what you want to be done, all within the same line. You might simply want the loop to stop when the condition is true. This can be programmed by placing a line number following THEN, which makes this statement into the equivalent of THEN GOTO. If this number is the number of the last line in the program, the program will end when the condition is true.

Now if all of that sounds rather complicated, take a look at the simple illustration in figure 5.8. We can't use a FOR..NEXT loop here because we don't know how many times we will want to go through the loop, so we have used IF..THEN to control the loop. The instructions appear first, and we make the variable `Total` equal to zero in line 50. Each time you type a number, then, in response to the request in line 60, the number that you have entered is added to the total in line 70, and line 80 prints the value of the total so far. Line 90 is the loop controller. IF is used to make the test, and in this case, the test is to find if N is *not* zero. If it's not, then we go back to line 60. The odd-looking sign that is made by combining the 'less than' and 'greater than' signs, is used to mean 'not equal'. You can put a GOTO following THEN, or leave it out as you please. Since it's just more to type, I have left it out.

The effect, then, is that if the number which you typed in line 60 was not a zero, line 90 will send the program back to line 60 again for another number. This will continue until you enter a zero. When this happens, the

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT TAB(38)"Heads or Tails"
30 PRINT:PRINT"Press H or T - E to end"
40 INPUT A$:IF UPPER$(A$)="E"THEN END
50 N%=1+INT(2*RND(2))
60 IF N%=1 THEN PRINT"HEADS"ELSE PRINT"TAILS"
70 GOTO 30
```

Figure 5.10 a Heads-or-tails program that makes use of ELSE following IF.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 FOR N%=1 TO 1000
30 INPUT"Name, please (type X to terminate) "
; Name$
40 IF UPPER$(Name$)="X"THEN N%=1000
50 NEXT
```

Figure 5.11 Breaking out of a name-entry loop, so that you don't have to enter 1000 names.

test in line 90 fails and the program goes to line 100. This announces the end of the program, and since there are no more lines, the program stops. If you press RETURN without having typed a number, then the program takes this as equivalent to entering a zero, and stops. Not all machine behave so sensibly!

This example uses just one test with its IF, but you aren't so limited. You could, for example, decide to terminate if either a 0 or 999 was entered. In this case, your IF line could read:

```
IF N<>0 AND N<>999 THEN 60
```

making two conditions. You can also use the word OR when you make a test, as in:

```
IF X=0 OR X=999 THEN . . .
```

but you would have to be careful in a number totalling program that 999 did not get added to your total! You have to be careful about where you place some of these tests!

Figure 5.9 illustrates the type of tests that you can perform using IF. These use the mathematical signs for convenience, but remember that all of these signs will have a meaning for strings as well. For the moment, it's easy to see what = and <> might mean, but later on we'll be looking at how < and > can be used for strings.

And if not, then what?

IF..THEN forms a test which can be very useful in programs. There's another extension to IF..THEN, however. You can use the word ELSE to carry out another test and cause a different sort of action. An example makes this a lot clearer, so take a look at figure 5.10.

This is a simple heads-or-tails gamble, with no scoring. Lines 10 and 20 set things up as usual, while line 30 starts the main loop of repeated actions with the main instructions. Line 40 asks for your guess of H or T, or the use of E to end the program. Line 50 is the important gambling line. RND means 'select a number at random', and when it is followed by a positive number in brackets, it tells the machine to select a number at random, lying in the range 0–1. The number is never quite zero, however, nor does it ever reach 1. By multiplying this fraction by 2, we'll get a number which can be almost zero, or almost 2 (like 1.999999). Taking INT, the integer part, will give a whole number between 0 and 1. If we add 1 to this, we get a number which can be either 1 or 2, and that's what we want for a heads-or-tails choice. The test is made in line 40, so that if N°₀ is 1, the word HEADS is printed, and if it's not 1, 'TAILS' is printed. In this example, ELSE is being used to choose the alternative action. Normally in an IF test, the alternative is in the next line. For example, if you have the line:

```
100 IF X%=5 THEN 20
```

the test will be made, and its result will be either true or false. If the result is true (X°₀ *is* 5), then the program goes to line 20. If the result of the test is false (X°₀ is not 5), then the rest of line 100 in this example is ignored, and line 110, or whatever comes after 100 will be run. Using ELSE allows you to put the alternative into the same line, and is particularly useful if the result of the test prints messages rather than going to another line. Now its up to you to turn this into a more useful game, asking the user to guess what is coming, and keeping a score! Later on we'll look at how we can program for a larger number of tests. For the moment, it's time to look at another type of loop that allows you a lot more choice of how you terminate it.

Breakout!

Sometimes you find that you need to break out of a FOR..NEXT loop before the count has been completed. You might, for example, have a number of inputs in the course of a loop that allows 1000 inputs, and want to end after only 20. Another common option is to have a title followed by a time delay loop that waits for 25 seconds, but which allows you to break out by pressing any key if you don't want to wait. Now you can use an IF test to break out of any loop, but when the loop is a FOR..NEXT one there's a possiblity that the unfinished count can cause trouble later. Figure 5.11 illustrates breaking out of a name entry loop. The name is input in the usual way, and the next line tests the name. If the name is 'X', which could be x or X because of the use of UPPER$, then the counter variable N°₀ is set to

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT TAB(39)"The Prologue"
30 PRINT:PRINT"wait, or press a key"
40 FOR J=1 TO 20000:K$=INKEY$
50 IF K$<>""THEN J=20000
60 NEXT
70 PRINT"next item..."
```

Figure 5.12 Breaking out of a time delay loop. As before, the breakout action makes use of a reassignment of the counter variable.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT TAB(39)"More Totals"
30 Total=0:J=1
40 PRINT:PRINT"Enter numbers for totalling; 0
 to end."
50 WHILE J<>0
60 INPUT J
70 Total=Total+J
80 PRINT"Total so far is "Total
90 WEND
100 PRINT"End of program"
```

Figure 5.13 A number-totalling program that makes use of the WHILE..WEND loop. Note the 'dummy' value of J in line 30.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A$=" "
30 WHILE UPPER$(A$)<>"X"
40 PRINT A$
50 READ A$
60 WEND
70 DATA Glenfiddich, glenmorangie, Laphroaig
80 DATA Islay Mist, Glenduff, X
```

Figure 5.14 Using a WHILE..WEND loop to read data from a list.

its final value of 1000, so causing the loop to end when it is tested again in line 20. Now you could just as easily have programmed:

```
IF UPPER$(Name$)="X"THEN END
```

with the same effect, but leaving the program not properly finished. To prove this, try using this test line. When you enter X, the program seems to end, but if you type CONT RETURN, you'll find that the loop has taken up again. Ending a loop in this way can be risky, then, because if there are other parts of the program to follow, they may not work correctly. The other point, about breaking out of a time delay, is illustrated in figure 5.12. The time delay in this case must include the INKEY$ assignment, and the test of K$. The principle, however, is the same, that the loop must be ended by reassigning the counting variable of the loop to its final value.

WHILE you WEND

Very few home computers offer you any more than a simple FOR..NEXT loop, some don't even have the ELSE command. Mallard BASIC, however, offers you a very different type of loop, the WHILE..WEND. This can often make it much easier to program a loop, and it avoids the use of GOTO, even the use of IF..THEN. The principle is that you start your loop with a condition, then you have as many lines as you like of what has to be done in the loop, and finally, the word WEND (meaning While END) to mark the end of the loop.

Yes, an example would certainly help, so cast an eye on figure 5.13. This is another version of an old friend, the number-totalling program. This time, as well as making `Total=0`, we have `J=1` near the start. This is needed because of the way that a WHILE..WEND looop works, as we'll see. The start of the loop is in line 50, `WHILE J<>Ø`. What this means is that the loop will be repeated for as long as J is not zero. When the program starts, however, you will not have input any number J by this stage. This is why a 'dummy' value for J has to be included in line 30 before the loop starts. Without this J=1 step, the program would finish as soon as it got to line 50!.

The steps in the loop are familiar, and we needn't go over them again. The important one to note is line 90, WEND. This marks the end of the loop, and will automatically send the loop back to the WHILE test. There's no need for IFs and THENs here, and you can even nest WHILE..WEND loops inside each other. The only snag is that the test is made right at the start of the loop. You must have a value for whatever is being tested at this stage, or the loop simply won't run.

Take a look at another example, figure 5.14, this time using READ..DATA inside the loop. This is a program which simply reads a number of data items from a list until it encounters an `X` or `x`. In line 20, the string variable A$ is set to a space, obtained by typing a quotemark, then the spacebar, then another quotemark. The loop starts in line 30, with the condition that the loop continues until an `X` is found as the value of UPPER$(A$). The loop consists of printing the value of A$ (a blank first time round), then reading the DATA list for another value of A$. Line 60 is the WEND for the loop, sending the value of A$ back to line 30 to be tested. The overall effect is that the program prints the list of DATA items. Very tasty! Can you think what would happen if you avoided printing the blank by having lines 40 and 50 the other way round? If not, try it, and you'll see why it was written the way it is!

Figure 5.15 shows yet another use for the WHILE..WEND loop. In this case, it acts as a 'mugtrap'. A mugtrap (polite name – data validator) is a piece of program that tests whatever you have entered. If what you have entered is unacceptable, like a number in the wrong range, then the mugtrap refuses to accept the entry, shows by a message on the screen why the entry is unacceptable, and gives you another chance. Mugtraps are very important in programs where a piece of incorrect entry might stop a program with an error message. For an expert (you, after you have finished this book!) this is no problem, a crafty GOTO will get back to the program. For the inexperienced user, the error message seems like the end, and it's likely that the whole lot will be lost, even if it took all day to enter the data!

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 INPUT "type a number in range 1 to 5 ";N%
30 WHILE N%<1 OR N%>5
40 PRINT"Unacceptable answer - 1 to 5 only"
50 INPUT N%
60 WEND
70 PRINT"You picked "N%
```

Figure 5.15 Incorporating a WHILE..WEND loop into a mugtrap.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT"Press any key..."
30 WHILE INKEY$="":WEND
40 PRINT"That's it!"
```

Figure 5.16 Using WHILE..WEND with INKEY$.

In this example, then, you are invited to enter numbers in the range 1 to 5. If the number that you enter is in this range, all is well, but if not (try it!), then the WHILE..WEND loop swings into action. This prints an error message of your own, and gives you another chance to get it right. That's the essence of a good mugtrap, and the WHILE..WEND loop is ideal for forming such traps. Note, despite the emphasis on numbers in some examples, that the WHILE..WEND loop is just as much at home with strings. You can have lines like:

```
WHILE Name$ <>"X"
```

to allow you to keep entering names into a list, or

```
WHILE UPPER$(AN$) <> "Y" AND UPPER$(AN$) <> "N"
```

to make a mugtrap for a 'Y' or 'N' answer. Don't forget the use of UPPER$ to avoid having to test for `y` and `n` as well as for `Y` and `N`.

While we're on the subject, the WHILE..WEND loop is a very useful one to use with INKEY$. Figure 5.16 shows such a loop used to produce a 'wait until ready' effect. The WHILE..WEND loop will keep repeating for as long as INKEY$ is a blank, and there is no need in this case to assign a K$ to INKEY$. If, however, you need to assign the key, as you would if you were testing for a Y or N answer, then the version that we looked at in the previous chapter is simpler and easier.

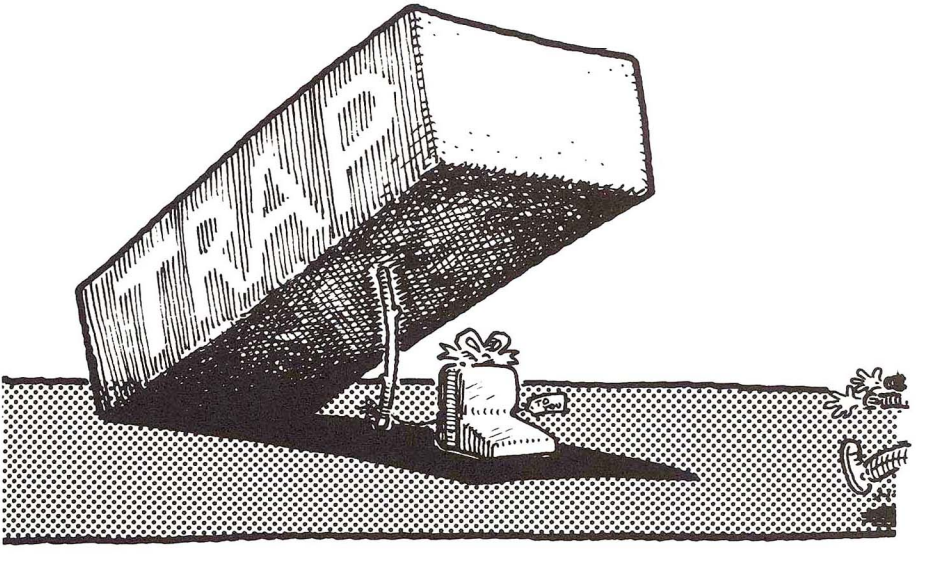
Last pass

When you start writing programs for yourself, designing a loop often appears to be difficult. It's not, but you need to approach it with some method. The best way is to write down what the loop conditions are to start

with. What conditions do you want at the start of the loop, for example? Do you need a counter in the loop which will have to be set to some starting value? In such a case, you might want to use a FOR..NEXT loop, and you then need to decide what to call the counter variable, and if it can be an integer. If your loop has nothing to do with numbers, or if it has no preset count, then the WHILE..WEND type is likely to be more useful. Once again, though, you have to look for the starting conditions. remember that the WHILE..WEND loop makes a test right at the start of the loop, and so you have to ensure that whatever is tested at the start has a suitable value.

Next, you have to think of what is to be done in each pass of the loop. This might be some string action, or some number action, or a bit of both. The really important bit, however, is to decide what will end the loop. You might want to end a counting FOR..NEXT loop before the true end of the count, in which case please heed the words of wisdom earlier in this chapter. Ending the WHILE..WEND loop is easier – you simply alter whatever is tested at the WHILE part.

Finally, it's possible to get yourself tied up in testing a loop. Suppose, for example, you have designed a loop that calls for the entry of 1000 names, and you want to check that it ends correctly when you enter an X, or after 1000 entries. Needless to say, you don't check it by going through all 1000 entries. You can either alter the count to 5 for testing, or you can alter the count after stopping. For example, suppose that your loop starts with `FOR N%=1 to 1ØØØ`. At some stage in the loop, you can press the STOP key, which will halt things temporarily. You can then type `N%=998` RETURN, and then type `CONT` RETURN to re-enter the loop. This will then show how the loop finishes when you are using numbers of this size. For that reason, this is a better test than altering the loop size to a count of 5.



Chapter 6

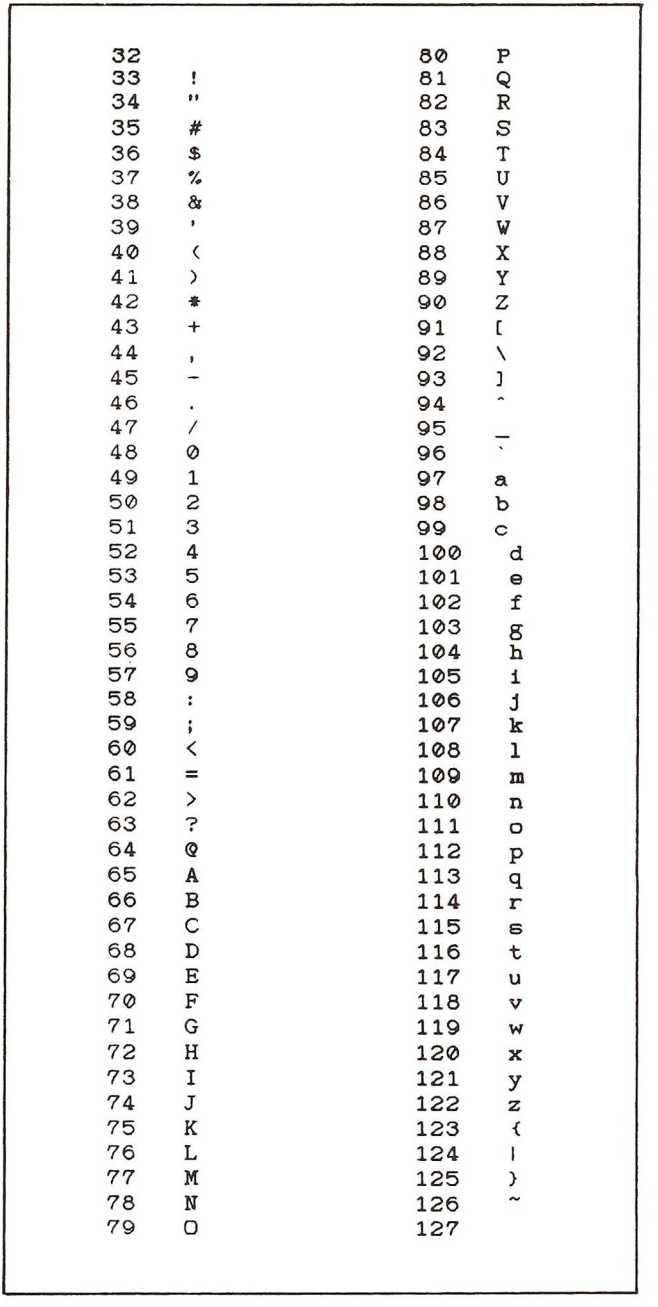
Strings and things

String functions

In Chapter 4, we took a fairly brief look at number operators and functions. If numbers turn you on, that's fine, but string functions are in many ways more interesting. What makes them that way is that the really eyecatching and fascinating actions that the computer can carry out are so often done using string functions. What's a string function, then? As far as we are concerned, a string function is any action that we can carry out with strings. That definition doesn't exactly help you, I know, so let's go into more detail.

A string, as far as Mallard BASIC is concerned, is a collection of characters which is represented by a string variable, a name which ends with the dollar sign or whose starting letter has been used in a DEFSTR statement. You can pack practically as many characters as you are likely to need into a Mallard BASIC string – a maximum of 255 characters per string, which is a longer string than most of us will ever need. Like other computers, Mallard BASIC stores its strings in a way that is very different from the way that is used to store numbers, making use of what is called ASCII code. The letters stand for American Standard Code for Information Interchange, and the ASCII (pronounced Ass-key) code is one that is used by most computers. Figure 6.1 shows a printout of the ASCII code numbers and the characters that they produce on the Amstrad printer, with the printer in its US character set.

Each character is represented by a number, and the range of numbers is from 32 (the space) to 127. On the printer, 127 produces nothing, but on the Mallard BASIC screen, you'll see a zero appear. You can assign characters to a string variable by using the equality sign. When you assign in this way, you need to use quotes around the characters. You can also assign using INPUT or LINE INPUT, and using READ..DATA, when no quotes are needed. If you want to use quotes within a string, then this can be achieved with the use of LINE INPUT.



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

Figure 6.1 The standard ASCII code numbers.

Now all number variables are represented in a different type of coding, one that uses the same number of codes no matter whether the value of the variable is large or small. There's one type of number coding for integers, and another for ordinary real numbers which comes in two sizes according

to whether you want to use single or double precision numbers. Because a string consists of a set of number codes in the memory of the computer, one code for each character, we can do things with strings that we cannot do with numbers. We can, for example, easily find how many characters are in a string. We can select some characters from a string, or we can change them or insert others. Actions such as these are the actions that we call 'string functions'.

Len in action

One of these string function operations that I mentioned was finding out how many characters are contained in a string. Since a string can contain up to 255 characters, an automatic method of counting them is rather useful, and LEN is that method. LEN has to be followed by the name of the string variable, within brackets, and the result of using LEN is always an integer number so that we can print it or assign it to an integer number variable. You don't need to put a space between the N of LEN and the opening bracket.

Figure 6.2 shows a simple example of LEN in use. Line 20 assigns a variable which is then printed in line 30, and then line 40 tells you how many characters are in this variable. Note the word 'characters', it's not the same as 'letters'. Each space, full stop, comma and so on counts as a character for the purposes of a string, because each one is represented by an ASCII code number. The quotes that mark the start and end of a string in an assignment are *not* counted, however. Lines 50 and 60 illustrate how you can find the length of a string which you have entered. This is something that is useful if you want to ensure that a name entered at the keyboard is not too long for the computer to use. You might, for example, have a program that places names into a printed form, with columns of restricted width, such as fifteen characters. If too long a name is entered, it could spill over into another column. You'll probably notice, if you have a long name or address, that when you get letters that have been computer-addressed, some part of the name or address may have been shortened. Now you know why, and how!

All this is hardly earth-shattering, but we can turn it to very good use, as figure 6.3 illustrates. This program uses LEN as part of a routine which will print a string called T$ centred on a line. This is an extremely useful routine to use in your own programs, because its use can save you a lot of

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A$="Mallard BASIC in action"
30 PRINT A$
40 PRINT"There are "LEN(A$)" characters here"
50 LINE INPUT"Try a phrase for yourself  ";B$
60 PRINT"There are "LEN(B$)" characters here"
```

Figure 6.2 Introducing LEN, a member of the string function family.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 T$="The splendid Mallard BASIC"
30 PRINT TAB((90-LEN(T$))/2)T$
```

Figure 6.3 Using LEN to print titles that will automatically be centred.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 N$="22.5":V%=2
30 PRINT:PRINT N$" times "V%" is "V%*VAL(N$)
40 PRINT
50 V$=STR$(V%)
60 PRINT"there are "LEN(V$)" characters in "V%'
 !"
70 PRINT
80 PRINT"does "N$" added to"V$" give " N$+V$"?'
```

Figure 6.4 Using VAL and STR$ to interchange string and number values. Watch for the added space!

tedious counting when you write your programs. The principle is to use LEN to find out how many characters are present in the string T$. This number is subtracted from 90, and the result is then divided by two. If the number of characters in the string is an even number, then the TAB number will contain a .5, but this is completely ignored by TAB when the string is printed. You can, incidentally, use 90 or 91. Whichever one you use, you will find that words are reasonably well centred – 90 works better with phrases which have an even number of characters, and 91 works better with phrases which have an odd number of letters. Yes, you could write a program that adjusted the position with a few IF..THEN..ELSE steps! We can use a routine of this type to centre anything that has the name T$. In Chapter 8, we'll be looking at the idea of *Subroutines*, which allow you to type the set of instructions (for centering a title, for example) just once, and then use them for any string that you like.

STR$ and VAL

You know by now that there are some operations that you can carry out on numbers but not on strings, and some which you can carry out on strings but not on numbers. This might be inconvenient, but as it happens, we can convert numbers from one form into another quite easily. This allows us to perform arithmetic on a number that has been in string form, and also to use string functions on a number that was formerly only in number form. Take a look at figure 6.4. To start with, we make N$ a number in string form, and V% a number in integer-number form. Line 30 then shows how we can carry out arithmetic with N$. By typing `VAL(N$)` in place of `N$` alone, the number value of N$ is used in the calculation, and the correct result is obtained. In line 50, number V% is transformed into a string, V$ by the use of STR$(V%). There's a warning here, however, as line 60

shows. Number V% was equal to 2, a single digit number. When STR$ has been used to convert to string form, however, the number of characters is increased mysteriously to two. This is because of that invisible space which is put in to allow for a plus or minus sign. The STR$ routine always includes the space, so that the length of a string which has been obtained from a number is always one character too much unless there has been a minus sign in the number. Line 80 is there to remind you of what can happen if you forget about VAL and try to add two strings!

VAL is used mainly when you have had an INPUT to a string variable, and after testing for items like length of the string, it is converted to number form using VAL so as to be tested for correct range. If necessary, a number that will eventually be an integer can be converted into single-precision form for its range test. This will ensure that the program is not stopped by an error message if the size of the number is outside the range of an integer. If testing shows that the number is acceptable, it can be changed into integer form by a statement such as X%=X. The use of STR$ is less common, but can be applied when numbers have to be placed in strings with a suitable format. The usual requirement is for numbers to be lined up with the decimal points in a vertical column, and this type of lining up is much easier if the numbers are in string form. See Chapter 11, however, for another method of doing this with PRINT USING.

A slice in time

The next group of string operations that we're going to look at are called slicing operations. The result of slicing a string is another string, a piece copied from the longer string. String slicing is a way of finding what letters or other characters are present at different places in a string. You can also use it to select different parts of a string so that these can be printed or reassigned.

All of that might not sound terribly interesting, so take a look at figure 6.5. A set of strings are assigned in lines 20 to 50, using variable names A$ to

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A$="City lights"
30 B$="of two tries"
40 C$="Truly"
50 D$="rotton"
70 PRINT"Name a Mallard rival!"
80 PRINT"Press any key for answer"
90 WHILE INKEY$="":WEND
100 Z$=LEFT$(A$,5)+LEFT$(B$,3)+LEFT$(C$,3)+LE
FT$(D$,2)
110 PRINT Z$" (rail buffs only)
```

Figure 6.5 Extracting letters from strings by using LEFT$.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT:INPUT"Your surname, please ";SN$
30 PRINT:INPUT"Your first name, please ";FM$
40 PRINT:PRINT
50 PRINT"You'll be known as "LEFT$(FM$,1)+"."
+LEFT$(SN$,1)+". round here."
```

Figure 6.6 Using the LEFT$ slicing action to get initials from a name.

```
10 DEF FNabrevit$(name$,size%)=LEFT$(name$,size%
20 PRINT CHR$(27)+"E"+CHR$(27)+"H"
30 PRINT"Enter a long name"
40 INPUT N$
50 PRINT"Enter a number smaller than "LEN(N$)
60 INPUT length%
70 PRINT"First"length%"letters are ";FNabrevit$
(N$,length%)
```

Figure 6.7 Using LEFT$ in a defined string function to slice any name for as many letters as is needed. Note the form of the string function name.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A$="Mallard magic"
30 PRINT:PRINT
40 PRINT"It's all "RIGHT$(A$,5)" to me!"
```

Figure 6.8 Using the RIGHT$ slicing action to extract letters from the right-hand side of a string.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 INPUT "Your name, please ";A$
30 N%=LEN(A$)
40 FOR J%=1 TO N%
50 PRINT LEFT$(A$,J%);TAB(21)RIGHT$(A$,J%)
60 NEXT
```

Figure 6.9 Using left and right slicing actions in a loop to print out words in a very odd way!

D$. You are then asked a question for railway buffs, and asked to press any key for the answer. What's printed in line 110 is a phrase which uses letters that have been selected from the left-hand sides of the other strings. Now how did this happen? The instruction LEFT$ means 'copy part of a string starting at the left-hand side'. LEFT$ has to be followed by two quantities, within brackets and separated by a comma. The first of these is the variable name for the string that we want to slice, A$ to D$ in this example. The second is the number of characters that you want to slice (copy, in fact) from the left-hand side. The effect of `LEFT$(A$,5)` is therefore to copy the first five characters from '`City lights`', giving '`City `' – and note that the

space has been copied. The next string from line 30 is then treated in the same way and tacked on (concatenated) to the first part, and in the same way, the other strings are sliced and the slices added in for form Z$. The results of all this is the phrase which is printed on the screen in line 110.

For a slightly more serious use of this instruction, take a look at figure 6.6. This has the effect of extracting your initials from your name, and it's done by using LEFT$ along with a bit of concatenation. The INPUT steps in lines 20 and 30 find your surname and first name, and assign them to variable names SN$ and FM$. We can't use the more obvious FN$ for forename, because FN is a reserved word in BASIC, and you are not allowed to use reserved words as variable names. You will get a `Syntax error` report if you try to do this. Line 50 then prints your initials by using LEFT$ to extract the first letter of each string. The letters are then assembled along with full stops, using concatenation in line 50. If you have two players in a game, it's often useful to show the initials and score rather than printing the full name, but the full names can be held stored for use at various stages in the game. You can also make this type of action into a defined function. When you use a defined function that returns a string, the only difference is that the name of the function must be a string name. In the example of figure 6.7, the name is `FNabrevit$`, and that final dollar sign is *very important* – without it, you'll get a `Syntax error` message. The action of the defined function is to produce the left slice of a string, which is why the defined function must have a string name. The function as demonstrated allows you to enter a name, and then to select the left-hand side of it as you please. You don't need to use a defined function for this straightforward action, but it's a convenient place to point out how string defined-functions can be used.

All right, Jack?

String slicing isn't confined to copying a selected piece of the left-hand side of a string . We can also take a copy of characters from the right-hand side of a string. This particular facility isn't used quite so much as the LEFT$ one, but it's useful none the less. Figure 6.8 illustrates the use of these instructions to avoid having to type a word over again. There are, of course, more serious uses that this. You can, for example, extract the last four figures from a string of numbers like `Ø1Ø-242-7Ø16`. I said a *string* of numbers deliberately, because something like this has to be stored as a string variable rather than as a number. If you try to assign this to a number variable, you'll get a silly answer. Why? Because when you type `N = Ø1Ø-242-7Ø16` then the computer assumes that you want to subtract 242 from 10 and 7016 from that result. The value for N here would be –7248, which is not exactly what we had in mind! If you use `N$="Ø1Ø-242-7Ø16"` then all will be well.

Now we can get quite a lot of interesting effects from LEFT$ and RIGHT$. Take a look at figure 6.9 for an example, which does odd things with the letters of your name. The program prompts you to enter your name in line 20, and the name is assigned to A$. In line 30, we use LEN so that the number variable $N^{o}{}_{o}$ contains the total number of characters in

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A$="Mallard BASIC"
40 L%=LEN(A$)
50 FOR N%=1 TO L%
60 PRINT MID$(A$,N%,1);" ";:NEXT
70 PRINT:PRINT
80 FOR N%=1 TO L%
90 PRINT MID$(A$,N%,1)+"+";:NEXT
```

Figure 6.10 Slicing from any part of a string, using MID$ with a position number and a number of characters. These numbers can, of course, be replaced by variables.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 INPUT" Your name, please ";NM$
30 L%=LEN(NM$):C%=L%/2+1
40 FOR N%=1 TO C%
50 PRINT TAB(45-N%)MID$(NM$,C%-N%+1,N%*2-1)
60 NEXT
```

Figure 6.11 Making a letter pyramid to show the action of slicing with an expression.

your name. This will include spaces and hyphens – nobody's likely to use asterisks and hashmarks! Line 40 starts a loop which uses the total number of characters as its end limit. Line 50 is the action line. When J% is 1, line 50 prints the first letter on the left of your name on to the left-hand side of the screen, and the first letter on the right of your name at a position further over to the right-hand side. On the next pass through the loop, a new line is selected, and two letters are printed. This continues until the entire name is printed. If you use a LEFT$ or RIGHT$ with a number that is more than the number of letters in the string, then you simply get the whole string.

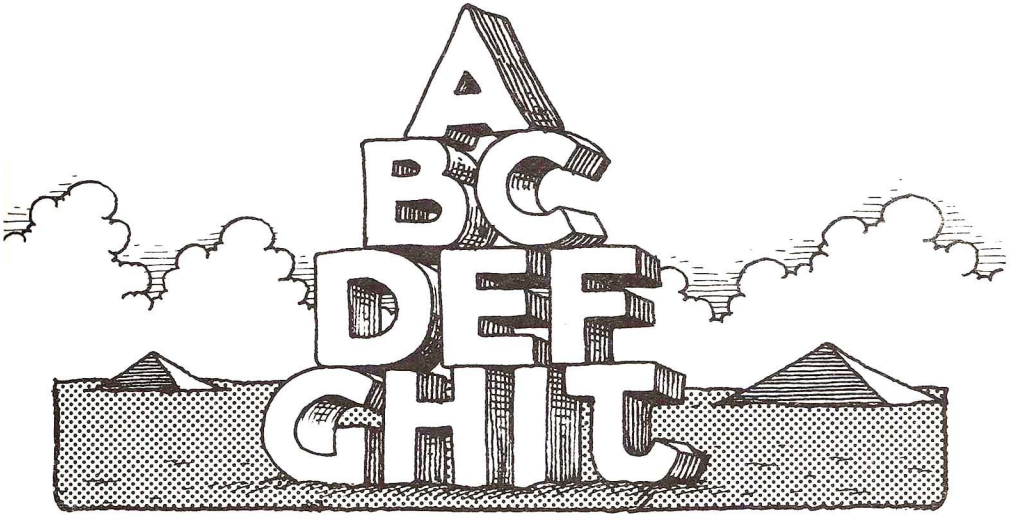
Middling along

There's another string slicing instruction which is capable of much more than either LEFT$ or RIGHT$. The instruction word is MID$, and it has to be followed by three items, within brackets, and using commas to separate the items. Item 1 is the name of the string that you want to slice, as you might expect by now. The second item is a number which specifies where you start slicing. This number is the number of the characters counted from the left-hand side of the string, and counting the first character as 1. The third item is another number, the number of characters that you want to slice, going from left to right and starting at the position that was specified by the first number.

It's a lot easier to see in action that to describe, so try the program in figure 6.10, which illustrates the power that this statement gives you over the display of strings. Line 20 assigns `Mallard BASIC` to A$, and line 30 finds

L%, the number of characters in this phrase. The loop that starts in line 40 then prints letters taken from the word phrase. With the value of N% equal to 1, the letter that is sliced is M, because its position in the word is 1, and we're copying one letter from this position. If we used `MID$(A$,1,2)`, we would get Ma, and if we used `MID$(A$,3,2)` we would get la. As it is, we select a letter at a time, and print a space. The semicolon in line 60 then ensures that the next sliced letter is printed on the same line. The nett effect is that the letters are printed spaced out. The second loop in lines 80 and 90 performs the same kind of effect, but places a + sign between the letters rather than a space. On versions of BASIC that permit graphics shapes to be generated , this type of slicing can be used to create giant print by using each sliced letter to create a giant version, and printing the giant one.

One of the features of all of these string slicing instructions is that we can use variable names or expressions in place of numbers. Figure 6.11 shows a more elaborate piece of slicing which uses expressions. It all starts innocently enough in line 20 with a request for your name. Whatever you type is assigned to variable NM$, and in line 30 a bit of mathematical juggling is carried out. How does it work? Suppose you type DONALD as your name. This has six letters, so in line 30 L% is assigned 6, and C% is the whole number part of L%/2 (equal to 3), plus 1, making 4. Line 40 then starts a loop of 4 passes. In the first pass you print at TAB(44) – because N=1 and 45–N is 44 – the MID$ of the name using C%–N% + 1, which is 4–1 + 1 = 4, and N% × 2–1, which is also 1. What you print is therefore MID$(NM$,4,1), which is A in this example. On the next run through the loop, N% is 2, C%–N% + 1 is 3, and N% × 2–1 is also 3. What is printed is `MID$(NM$,3,3)`, which is NAL. The loop goes on in this way, and the result is that you see on the screen a pyramid of letters formed from your name. It's quite impressive if you have a long name! If your name is short, try making up a longer one!



```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A$="Effective management"
30 B$="How to manage purchasing"
40 C$="Keeping a menagerie"
50 F$="manage"
60 IF INSTR(A$,F$) THEN PRINT F$" is in "A$
70 IF INSTR(B$,F$) THEN PRINT F$" is in "B$
80 IF INSTR(C$,F$) THEN PRINT F$" is in "C$
```

Figure 6.12 Illustrating the action of INSTR to find if one string is contained within another.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A$="Mallard BASIC computing":PRINT
30 FOR N%=1 TO LEN(A$)
40 PRINT ASC(MID$(A$,N%,1));" ";
50 NEXT
```

Figure 6.13 Using ASC to find the ASCII code for letters.

Inside, upstairs and downstairs

There are some string functions that are not so easy to place in groups, but one, INSTR, definitely belongs close to the slicing functions. The use of INSTR is to find if one string is contained within another. The syntax is:

```
INSTR(main string, short string)
```

and the result is a number value. The number is the position number in the main string where the short string starts. If the short string does not exist in the main one, then `instr` gives zero, and this number-or-zero choice can be used in a very simple way to pick out strings, as figure 6.12 illustrates. In this example, three strings contain phrases which might be book titles and the program is set to work to scan the titles looking for a word `manage`. The test is arranged so that if the word is not found, the zero result of INSTR will make the IF test fail, so that the phrase is not printed. In the following chapter, you will see how a list of phrases could be arranged to be tested like this in a loop. Though the example does not use the facility, the fact that the position number for the start of the string that is being looked for can be obtained from INSTR is useful. In addition, you can specify in the INSTR statement that a search should start at some specific number position. This is done by placing the number as the first part of the INSTR argument, just before the main string name and separated from it by a comma.

Two functions that act on the characters of a string have already been mentioned, UPPER$ and LOWER$. UPPER$ converts each lower-case letter in a string into upper-case, and LOWER$ converts each upper-case letter into lower-case. Notice that these refer to letters, not characters generally, and only the letters of the alphabet are affected. Using UPPER$ or LOWER$ is useful in comparing strings, because if you are searching for

Smith, it's useful to know that the entries of SMITH, smith and even sMith will be correctly matched! The provision of UPPER$ and LOWER$ is therefore very useful for writing programs that search for matching strings, and also for arranging strings into correct alphabetical order.

Another one in this set is SPACE$. This is used to create a string of spaces using syntax in the form `S$=SPACE$(2Ø)`. SPACE$(n%), where n% is an integer number, creates a space with length equal to the value of N%, and this, being a string can be assigned to any string variable, or concatenated to any string variable. Using SPACE$ avoids the need to type spaces, and it's use in a program makes reading the program rather easier, because you don't have to count the spaces that have been assigned. Finally, STRIP$ is a rather specialised statement that will convert ASCII codes with values of 128 and above into codes in the normal range. It is used in the form `$=STRIP$(B$)`.

More priceless characters

It's time now to look at some other types of string functions. If you hark back a few pages, you'll remember that we introduced the idea of ASCII code. This is the number code that is used to represent each of the characters that we can print on the screen. We can find out the code for any letter by using the function ASC, which is followed, within brackets, by a string character in quotes or a string variable (no quotes). The result of ASC is a number, the ASCII code number for that character. If you use `ASC("Mallard BASIC")`, then you'll get the code for the `M` only, because the action of ASC includes rejecting more than one character. Figure 6.13 shows this in action. String variable A$ is assigned in line 20 and in line 30 a loop starts which will run through all the letters in A$. The letters are picked out one by one, using `MID$(A$,N%,1)`, and the ASCII code for each letter is found with ASC. Watch how the brackets have been used! The space between quotes, along with the semicolons in line 40 make sure that the codes are all printed with a space between the numbers, and without taking a new line for each number. Simple, really.

ASC has an opposite function, CHR$. What follows CHR$, within brackets, has to be a code number, and the result is the character whose code number is given. The instruction `PRINT CHR$(65)`, for example, will cause the letter `A` to appear on the screen, because 65 is the ASCII code for the letter A. We can use this for coding messages. Every now and again, it's useful to be able to hide a message in a program so that it's not obvious to anyone who reads the listing. Using ASCII codes is not a particularly good way of hiding a message from a skilled programmer, but for non-skilled users it's good enough. Figure 6.14 illustrates this use. Line 40 is a WHILE..WEND..INKEY$ loop to make the program wait for you. When you press a key, the loop that starts in line 50 prints 13 characters on the screen. Each of these is read as an ASCII code from a list, using a READ..DATA instruction in the loop. The PRINT CHR$(D%) in line 60 then converts the ASCII codes into characters and prints the characters, using a semicolon to keep the printing in a line. Try it!. If you wanted to

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT:PRINT"What's the key to compute-it-y
ourself?"
30 PRINT"Press any key to reveal the secret"
40 WHILE INKEY$="":WEND
50 FOR J%=1 TO 13
60 READ D%:PRINT CHR$(D%);
70 NEXT
80 DATA 77,97,108,108,97,114,100,32,66,65,83,
73,67
```

Figure 6.14 Using ASCII codes to carry a coded message, and then using CHR$ to obtain the character that corresponds to a code number.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A$="Understanding Electronic Components"
30 F$="electronic"
40 K%=INSTR(UPPER$(A$),UPPER$(F$))
50 IF K%=0 THEN 110
60 PRINT LEFT$(A$,K%-1);
70 PRINT CHR$(27)+"p";
80 PRINT MID$(A$,K%,LEN(F$));
90 PRINT CHR$(27)+"q";
100 PRINT RIGHT$(A$,LEN(A$)-K%+1-LEN(F$))
110 END
```

Figure 6.15 Using CHR$(27) with various letters to produce screen display effects.

conceal the letters more thoroughly, you could use quantities like one quarter of each code number, or 5 times each code less 20, or anything else you like. These changed codes could be stored in the list, and the conversion back to ASCII codes made in the program. This will deter all but really persistent decoders! This example, incidentally, illustrates the use of READ and DATA in a loop. We would normally use READ and DATA only for information that we particularly wanted to keep stored in a program like this. Another feature is the use of integer variables such as J% and D%. This takes less memory space, and using J% in the loop makes it operate significantly faster.

The CHR$ action, however, can do rather more than this example suggests. There are several ASCII codes that do not correspond to characters that can be printed on the screen, and CHR$ allows us to use them. Throughout this book, in fact, we have been using a CHR$ statement to clear the screen, and the key to this is printing `CHR$(27)`. This is called the ESC character, because many computers have a key marked ESC which generates this character. The nearest you get to this on the PCW machines is the EXIT key, but when you are using Mallard BASIC, the EXIT key does not give the code 27, though it can, like all other keys, be redefined if you know enough about CP/M. The point, however, is that by printing or LPRINT-ing strings that contain CHR$(27), you can carry out many effects that

make your programs considerably more effective. Unfortunately, the manual for Mallard BASIC has only a brief mention of these codes, and illustrates only the clear-screen and cursor position ones, using a neat and useful defined function. The full list of ESC codes for the screen is in Book 1 of the PCW manuals, on pages 140–141 (in my edition). The ESC codes for the printer are shown on pages 126 to 137. In the space of this book, we cannot go over every possible action of these codes, but in this chapter I shall show a few as an illustration of what is possible, and we shall refer to the use of the codes in following chapters.

The listing in figure 6.15 is another illustration of the use of INSTR, along with UPPER$, but in this case showing how a found string can be highlighted. The inverse video effect of black-on-green is switched on by using `?CHR$(27)+"p"`, and it's important to note that this must be a lower-case p. The effect is switched off again by using `?CHR$(27)+"q"`, and in the example, the main string is sliced so that the first part is printed normally, the found part is printed in inverse video, and the last part is printed normally again. It looks very impressive! This listing illustrates how the ESC codes can be used, and you will find that you can enable or disable the cursor, underline words, and delete whole lines, shift the cursor to any point on the screen, and so on. The list of possible actions is very impressive, and it takes quite a time to realise just how much can be achieved. One of the major actions concerns the use of windows , or screen viewports, as the manual refers to them, and this will be dealt with in Chapter 11.

There's more still, because the use of ?CHR$(27) also allows you to print on screen characters that have code numbers in the range 0 to 31. The listing in figure 6.16 shows this in action, with the character list of 0 to 31 being printed. The one that does not appear for some reason is the down-arrow sign, code 9. If you want to switch to some national character set, then the use of `?CHR$(27)+"n"` will do this, using the number of the character set as n. Remember that characters whose codes are in the range 128 to 255 can be printed directly, just by using `?CHR$(200)`, for example; and this, of course, applies also to any characters in the range 32 to 127 as well.

These ESC characters can also be sent to the printer by using LPRINT in place of PRINT (or ?). When this is done, you can control all of the printer actions from BASIC, including page sizes, line spacings, characters per line, print style (Italic, Bold, superscript etc), and all of the other features that will be familiar from the use of *LocoScript*. You can even send codes to the printer that will make it print graphics, something that is much less easy to achieve on the screen. This type of programming, however, belongs in a book on advanced techniques.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 FOR N%=0 TO 31
30 PRINT CHR$(27)+CHR$(N%);" ";
40 NEXT
```

Figure 6.16 Printing the characters for ASCII codes 0 to 31. These codes normally produce actions (or have no effect), and the characters appear only if CHR$(27) has been used.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT"Which list do you want?"
30 N%=0:WHILE (N%<1 OR N%>3)
40 INPUT" 1, 2, or 3 ";N%
50 PRINT CHR$(27)+"I"+CHR$(27)+"1";
60 WEND
70 IF N%=1 THEN RESTORE 160
80 IF N%=2 THEN RESTORE 170
90 IF N%=3 THEN RESTORE 180
100 A$=""
110 WHILE Q$<>"X"
120 PRINT Q$;" ";
130 READ Q$
140 WEND
150 END
160 DATA Opel,Mercedes,Porsche,X
170 DATA Renault,Peugeot,Citroen,X
180 DATA Fiat,Alfa-Romeo,Lancia,X
```

Figure 6.17 How RESTORE can be used to select different DATA lines.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 A$="QWERTY"
30 PRINT:INPUT"Type a word ",B$
40 IF A$=UPPER$(B$) THEN PRINT"Same as mine!"
:END
50 IF A$>UPPER$(B$) THEN SWAP A$,B$
60 PRINT"Correct order is "A$" then "B$
```

Figure 6.18 Comparing words to decide on their alphabetical order.

Restoration comedy

There's another instruction which belongs with READ and DATA, and has a useful effect when we want to read strings in particular from DATA lines. This one is RESTORE. When you use RESTORE by itself, it means that the DATA pointer is to be reset. For example, if you have just read six items from a DATA line that holds only six, you can't have another READ, because there is nothing more to read. If you use RESTORE just before another READ, however, you will read the first item again. There's another twist to the use of RESTORE, however. RESTORE followed by a line number means that the DATA list will start again from the beginning of that numbered line. You must make sure, of course, that the line number which you have chosen *is* a data line! Take a look at figure 6.17. This offers a choice of data to be read by making use of RESTORE along with a

number. the number input has been put into a WHILE..WEND loop, with a bit of ESC coding used to move the cursor up and delete the line when the answer is given. This allows the screen to remain at its INPUT stage until a number in the correct range is input. When you pick a number in the correct range, it is used in lines 70 to 90 to carry out a RESTORE command which has a line number for the set of data that has been requested by number. It's unfortunate that the RESTORE command does not allow an expression to be used. If it did, we could program this more tidily, such as `RESTORE (1000*N%)`. Each DATA line contains three items ending with X, and the loop which reads the DATA is arranged so that it will stop when the X is read. We could have used a FOR..NEXT loop for reading, since each line contains the same number of items. By using this method, however, we allow any number of items to be used in each list, and unequal numbers as well, which is a lot more useful. RESTORE, used in this way is a useful method of selecting from a number of lists which will be selected each time the program is used.

The law about order

We saw earlier that the symbols =, < and > can be used to compare numbers. We can also compare strings, using the ASCII codes as the basis for comparison. Two letters are identical if they have identical ASCII codes, so it's not difficult to see what the identity sign, =, means when we apply it to strings. If two long strings are identical, then they must contain the same letters in the same order. It's not so easy to see how we use the > and < signs until we think of ASCII codes. The ASCII code for A is 65, and the code for B is 66. In this sense, A is 'less than' B, because it has a lower ASCII code number. If we want to place letters into alphabetical order, then, we simply arrange them in order of ascending ASCII codes. This is not totally straightforward, because the ASCII codes for the lower-case letters are greater than the ASCII codes for the upper-case letters. This would lead to the letter `Z` being placed before the letter `a` if we did not use UPPER$ on each string we compare. Languages with no UPPER$ statement are at a considerable disadvantage here!

This process can be taken one stage further, though, to comparing complete words, character by character. Figure 6.18 illustrates this use of comparison using the = and > symbols. Line 20 assigns a nonsense word – it's just the first six letters on the top row of letter keys. Line 30 then asks you to type a word. The comparisons are then carried out in lines 40 and 50. If the word that you have typed, which is assigned to B$ is identical to `QWERTY`, ignoring differences in case, then the message in line 40 is printed, and the program ends. If `QWERTY` would come later in an index than your word, then line 50 is carried out. If, for example, you typed `PERIPHERAL`, then since Q comes after P in the alphabet and has an ASCII code that is greater than the code for P, your word B$ scores lower than A$, and line 50 swaps them round. This is done by using the very useful `SWAP A$,B$` statement, another one that appears in very few versions of BASIC. Line 60 will then print the words in the order A$ and then B$, which will be the correct alphabetical order. If the word that you typed comes later than `QWERTY`, for

example, TAPE, then A$ is not 'greater than' B$, and the test in line 50 fails. No swap is made, and the order A$, then B$, is still correct. Note the important point though, that words like QWERTZ and QWERTX will be put correctly into order – it's not just the first letter that counts.

Chapter 7

Complex data

Put it on the list

The variable names that we have used so far are useful, but there's a limit to their usefulness. Suppose, for example, that you had a program that allowed you to type in a large set of numbers. How would you go about assigning a different variable name to each item? It would be very unsatisfactory if you had to assign a new variable name to each item, and if you had to put in an INPUT line for each input. It would be much more satisfactory, in fact, if you could have just one INPUT routine that could be used in a loop, but to do this we need a different type of variable. Figure 7.1 illustrates this. Lines 10 to 40 generate an (imaginary) set of examination marks. This is done simply to avoid the hard work of entering the real thing! The variable in line 30 is something new, though. It's called a *subscripted variable*, and the 'subscript' is the number that is represented by N%. The name 'subscripted' that we use has nothing to do with computing, it's a name that was used long before computers were around How often do you make a list with the items numbered 1, 2, 3 . . . and so on? These numbers 1, 2, 3 are a form of subscript number, put there simply so that you can identify different items. Similarly, by using variable names A(1), A(2), A(3) and so on, we can identify different items that have the common variable name of A. A member of this group like A(2) has its name pronounced as *A-of-two*, and you can type this into the computer using either round brackets or square brackets as you please – other versions of BASIC are not so generous. In the listing, the more familiar round brackets have been used, and the use of ESC characters gives the heading effect. We have also used TAB to good advantage to make the listing neat – without the TABs it looks very ragged as you'll see if you remove them.

The usefulness of this method is that it allows us to use one single variable name for the complete list, picking out items simply by their identity

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 FOR N%=1 TO 10
30 A%(N%)=1+(RND(1)*100)
40 NEXT
50 PRINT:PRINT CHR$(27)+"p"
60 PRINT TAB(18)"MARKS LIST";SPACE$(18)
70 PRINT:PRINT CHR$(27)+"q"
80 FOR N%=1 TO 10
90 PRINT TAB(10)"Item "N%;TAB(19)"received "A%
(N%);TAB(34)" marks."
100 NEXT
```

Figure 7.1 An array of subscripted number variables. It's simpler than the name suggests!

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 DIM A%(12),N$(12)
30 PRINT TAB(10)"Please enter names and marks."
40 FOR N%=1 TO 12
50 INPUT "name ",N$(N%)
60 INPUT"Mark ",A%(N%)
70 NEXT
80 PRINT CHR$(27)+"E"+CHR$(27)+"H"
90 Total%=0
100 PRINT TAB(16)"MARKS LIST":PRINT
110 FOR N%=1 TO 12
120 PRINT TAB(10)N$(N%);TAB(50)A%(N%)
130 Total=Total+A%(N%)
140 NEXT
150 PRINT:PRINT
160 PRINT"Average mark was "Total/12
```

Figure 7.2 Using strings in one array, and numbers in another. The arrays have been dimensioned, using DIM, because numbers greater than 10 will be used.

numbers. Since the number can be a number-variable or an expression, this allows us to work with any item of the list. The example shows the list being constructed from the FOR..NEXT loop with each item being obtained by finding a random number between 1 and 100, and then assigned to A%(N%). Since A% is an integer name, this is an integer array, and the N% is the subscript number which *must* be an integer. You can have arrays of any data type, number or string, but the subscript numbers must always be integers. You can use any type of number variable as the subscript, but it will be chopped to an integer, and you can't expect to be able to refer to item A(2.5)! In the example, ten of these 'marks' are assigned in this way, and then lines 60 to 100 print the list. It makes for much neater programming than you would have to use if you needed a separate variable name for each number.

So far, so good, but one point has been omitted so far. Try altering the loop, so that it reads `FOR N%=1 TO 11`, and then run the program. You'll get an error message that says `Subscript out of range in 3Ø`. The computer is prepared for the use of subscript numbers of up to 10, but no higher. The computer has to be prepared by an additional statement for the use of numbers greater than 10 – the preparation consists of getting some more memory ready to receive the data. When you use DIM (meaning 'dimension'), the memory is allocated for the array. A line such as `DIM A%(11)` actually allows you up to *twelve* items in an array, in fact, because we can use `A%(Ø)` if we like, but you must not attempt to use `A%(12)` or any higher number. You will get the error message again if you do so. You can choose to begin any array at item 1 instead of item 0 if you like by using the statement `OPTION BASE 1` early in the program. The default is to start with item 0, and since you aren't forced to use it it does no harm apart from taking up memory space.

The important DIM instruction, then, consists of naming each variable that you will use for arrays, and following the name with the *maximum number*, within brackets, that you expect to use. You aren't forced to use this number, but you must not exceed it. If you do, and your program stops with an error message, you will have to change the DIM instruction and start again – which will be tough luck if you were typing in a list of 100 names! Note that you can dimension more than one variable in a DIM line, as figure 7.2 shows. Even though you don't have to use DIM when you use numbers of 10 or less, it's a good habit to do so. The reason is that it avoids wasting memory space, by making the most efficient use of the memory.

Figure 7.2 extends this use of arrays variables another step further. This time you are invited to type a name and a mark for each of twelve items. When the list is complete, the screen is cleared and a variable called `Total` is set to zero in line 90. The list is then printed neatly, and on each pass through the loop the total is counted up (in line 130) so that the average value can be printed at the end.The important point here is that it's not just numbers that we can keep in this list form. The correct name for the list is an array of one dimension, and this example uses both a string array (names) and a number array (marks). The dimension in this case is the single subscript number that is used to identify each item, and later on we'll be looking at arrays that have more than one dimension.

With the added facility of arrays, we can now begin to think of much more powerful programs, because we can now deal with lists of items without having to program any more than we would for one single item. We can, for example, enter items into a list with the type of routine that is illustrated in figure 7.3. In this example, a WHILE..WEND loop has been used, but there has to be a test for number, because the dimensioning of the loop must not be exceeded. This is always a problem, because if you use a very large dimension number, you will be reserving a lot of memory that you might in fact not use; and if you dimension too little, you may run out of room for your data. In this example, the main entry loop uses WHILE and WEND, with the condition for continuing written as:

```
(N%<=1ØØ AND Nr#<>Ø)
```

meaning that the loop will continue for as long as both conditions, N% not exceeding 100 *and* number `Nr#` not zero, are true. The program makes a lot

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT TAB(41)CHR$(27)+"r"+"NUMBERS"+CHR$(2
7)+"u"
30 N%=1:DIM A#(100):Nr#=1
40 WHILE (N%<=100 AND Nr#<>0)
50 PRINT CHR$(27)+"Y"+CHR$(44)+CHR$(42);
60 PRINT CHR$(27)+"K";
70 PRINT"Please enter number ";
80 INPUT Nr#:A#(N%)=Nr#
90 N%=N%+1
100 WEND
110 PRINT CHR$(27)+"E"+CHR$(27)+"H"
120 PRINT TAB(39)CHR$(27)+"p NUMBER LIST";CHR
$(27)+"q"
130 PRINT:PRINT
140 FOR J%=1 TO N%-2
150 PRINT TAB(41) A#(J%)
160 NEXT
```

Figure 7.3 Entering items in a list in such a way that dimensioning is not exceeded.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 DIM J%(100)
30 FOR n%=1 TO 100
40 J%(N%)=2*N%
50 NEXT
60 PRINT TAB(46)CHR$(27)+"p NUMBERS"+CHR$(27)+"q"
70 PRINT:PRINT
80 FOR N%=1 TO 100
90 IF n%/20<>INT(N%/20) THEN 130
100 PRINT"Press any key":WHILE INKEY$="":WEND
110 PRINT CHR$(27)+"Y"+CHR$(34)+CHR$(32)
120 PRINT CHR$(27)+"J"
130 PRINT TAB(48)J%(N%)
140 NEXT
```

Figure 7.4 Displaying a list, with paging to ensure that you have time to read the list. You can also use ALTS to 'freeze' the screen display (or restart it).

of use of the ESC character codes to underline the title in line 20, to position the cursor in line 50 and to print a title in inverse video in line 120. The important points, however, are that a number is input and allocated to an array place. In this example, the number has been allocated indirectly, using `INPUT Nr#`, followed by `A#(N%)=Nr#`. This is not essential, because you could use `INPUT A#(N%)` just as easily. When you want to do a lot of checking on an entry, however, it's easier to use an intermediate, and it's also useful to have a record of the entry to use in the WHILE condition. Since N% is incremented in the loop, you couldn't test `A#(N%)<>Ø` in the

WHILE part of the loop and you would have to use A#(N%-1) instead, and assign some dummy value to A#(Ø). Using an intermediate variable name can make for a lot less typing, and a neater looking program. Incidentally, did you test that the WHILE loop would terminate correctly? It's easy enough to test for the ending when zero is entered, but to test the alternative ending for 100 entries, you need to break the loop by pressing the STOP key. Then type N%=98 RETURN, and follow this with CONT RETURN. This will get you back to the loop, and you can enter the remaining numbers, and watch as all 100 entries, most of them zeros, are listed.

Another aspect of working with arrays is display. As you will have seen from the previous example, the display can be far from satisfactory if the numbers just scroll the screen, leaving you with no time to look at them. What is needed is some method of paging the listing so that the numbers can be inspected, and figure 7.4 shows what can be achieved. The array is created for you this time to save the labour of typing it, and the important point is the display. I'm assuming that the display has to be in the form of one item per line, because if you can accept more than one item per line then large amounts of data can be displayed by using a PRINT A#;" "; action in the printing loop. In this example, the list is paged by testing the value of the control number in the FOR..NEXT loop, N%. The test is for N%/20<>INT(N%/20), which is true except for N% values of 20, 40, 60, and so on, all multiples of 20. When the test is true, the lines 100 to 120 are skipped, and the numbers are printed normally. When the test is not true, at a multiple of twenty, the intermediate lines swing into action. Line 100 causes a pause until a key is pressed , and then the next two lines organise the paging. Line 110 places the cursor on the first line of the numbers, leaving the title untouched, and line 120 clears the screen from the cursor position to the foot of the screen. This action, however, leaves the cursor still in position for the next page, which will be printed whenever you press any key.

Manipulating arrays

The main point of using arrays is to manipulate data that is held in the form of array items. Manipulation can mean almost anything that you want to do with the data, but it usually boils down to two particular actions, searching and sorting. More has been written on these actions than on anything else in computing, but a lot that has been written was written a long time ago, and some of it is very hard to follow unless you happen to be a student of computing theory. In this chapter, we can't exactly go into great detail about these actions, but we'll look at one example each of methods that we can use. These methods apply equally to any type of array, but you have to make suitable adjustments. For example, if you make a test for a number array that uses A%, then adapting the program to a string array that uses A$ means that the wording of the test must be changed. For that reason, I'll demonstrate one number action and one string action.

We'll start with searching. There's no problem in searching an array for the 56th item – you just do a PRINT A%(56), or whatever you want. Even if this

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 DIM A%(100)
30 FOR N%=1 TO 100
40 A%(N%)=1+(RND(1)*100)
50 NEXT
60 PRINT"list ready":PRINT
70 FOR N%=1 TO 100
80 IF A%(N%)/7<>INT(A%(N%)/7)THEN 100
90 PRINT A%(N%)" is divisible by 7"
100 NEXT
```

Figure 7.5 Searching through a list, in this example for numbers that are exactly divisible by 7.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 Size%=100
30 DIM List$(100)
40 REM create list of 'words'
50 PRINT"please wait...creating list"+CHR$(27)
+"f"
60 FOR N%=1 TO Size%:A$=""
70 FOR J%=2 TO 2+(RND(1)*10)
80 A$=A$+CHR$(65+(RND(1)*25))
90 NEXT:List$(N%)=A$:NEXT
100 PRINT"List ready...now sorting"
110 REM sort routine
120 Y%=1
130 WHILE Y%<Size%:Y%=2*Y%:WEND
140 WHILE Y%<>0
150 Y%=(Y%-1)/2
160 IT%=Size%-Y%
170 FOR T%=1 TO IT%:J%=T%
180 Z%=J%+Y%
190 IF List$(Z%)>List$(J%)THEN 220
200 SWAP List$(Z%),List$(J%)
210 J%=J%-Y%:IF J%>0 AND Y%>0 THEN 180
220 NEXT:WEND
230 REM end of sort
240 FOR N%=1 TO Size%
250 IF N%/25<>INT(N%/25)THEN 280
260 PRINT"Press any key...":WHILE INKEY$="":
WEND
270 PRINT CHR$(27)+"E"+CHR$(27)+"H"
280 PRINT List$(N%)
290 NEXT
300 PRINT CHR$(27)+"e"
```

Figure 7.6 A Shell-Metzner sort routine for strings. Most of the time in this program is spent in generating the random 'words' used to demonstrate the sorting process.

means an input step, it amounts only to `INPUT X%:? A%(X%)`, and is no problem. The searching arises when you want to know if you have any items that are divisible by 7, or any names that start with 'R'. This involves looking at each item in the array and testing it, which is what searching is all about. The simplest type of search is through all of the items, but if the items are arranged in some order it's sometimes possible to devise quicker and more elaborate searches, called binary searches. Leaving these complications for the more advanced programmer, let's see what a search through a number list looking for numbers divisible by 7 would be like. The example in figure 7.5 first generates an array using random numbers, and you can see by the time it takes to produce the `List ready` report that this takes some time. Since the list consists of numbers generated almost at random, they are in random order and some of them must be divisible by 7. The search uses a FOR..NEXT loop which tests each item on the list using the same type of test as we used earlier in the example of paging a display. If the test in line 80 is true, and the number does not divide evenly by 7, then line 100 is carried out, so choosing another item. If the number is divisible by 7, then line 90 prints the fact. A common mistake in your first effort at searching is to find only the first matching answer, and to stop the search at that point. Unless you know that there can be only one item that answers the description this can be a cause of problems. If this were a string list, you might be looking for all of the items that started with a particular letter. You would therefore have an input line to find what letter you wanted to look for, and you would assign this to some variable like search$. You would then use a similar loop, but with a test such as:

```
IF LEFT$(search$,1)<>LEFT$(J$(N%)) THEN 100
```

and following this a line that printed the string items that were found.

Sorting into order is a very much more difficult business. At one time, all books on BASIC would show a routine called the *Bubble sort*, whose only merit was that it was easy to explain. Since a Bubble sort on a long string array can take *hours* to complete, it's not one that is of much use to programmers, unless you are working with a string that is almost completely sorted. The problem is that any sorting routine that is efficient is very difficult to explain, and the best way of understanding it is to go through it step by step. A faster routine that is the favourite in terms of speed and comparative simplicity is called the Shell–Metzner sort, after the names of its original programmers. It is based on the idea of comparing items that are some way apart in the list, starting by dividing the list into two equal parts, and looking at the first item in each part. These items are compared, and if they are in the wrong order, they are swopped. The spacing is then decreased and the exercise repeated until the items that are being compared are next to each other. Another set is then taken, and the process repeated until all the items of the whole list are in order. Figure 7.6 shows this in action, with a list that is made up from 'words' created at random – and that's a routine that you might want to use for your own tests! The cursor has been turned off during the routine by the ESC code in line 50, and on again at the end in line 300. The sort routine then fixes a number that is a power of two to decide where to start splitting the list, and it then starts making comparisons. If you want to see what items are being compared, then add a line:

```
185 PRINT Z%,J%
```

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 DIM N$(3,2)
30 FOR N%=1 TO 3
40 FOR J%=1 TO 2
50 READ N$(N%,J%)
60 NEXT:NEXT
70 FOR N%=1 TO 3
80 PRINT TAB(10)N$(N%,1)TAB(30)N$(N%,2)
90 NEXT
100 DATA Horse,Foal,Cow,Calf,Dog,Puppy
```

Figure 7.7 Making a matrix of rows and columns.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 DIM A$(50,2)
30 FOR N%=1 TO 50
40 PRINT CHR$(27)+"Y"+CHR$(37)+CHR$(42);
50 INPUT "Name ",A$(N%,1)
60 PRINT CHR$(27)+"Y"+CHR$(39)+CHR$(42);
70 INPUT "Tel. No. ",A$(N%,2)
80 PRINT CHR$(27)+"E"+CHR$(27)+"H"
90 NEXT
100 PRINT"List complete"
110 INPUT"Pick an initial letter ",J$
120 FOR N%=1 TO 50
130 IF UPPER$(LEFT$(J$,1))<>UPPER$(LEFT$(A$(N
%,1),1)) THEN 150
140 PRINT"name "A$(N%,1)"  Tel. No. "A$(N%,2)
150 NEXT
```

Figure 7.8 Using a name and number matrix for a simple telephone directory application.

but don't keep this in because it greatly slows down the rate of sorting. You can use this routine for sorting numbers rather than strings, simply by substituting a number array for `List$` – you could, for example, use `List%`, `List!` or `List#` for integers, single-precision, or double-precision numbers respectively.

Rows and columns

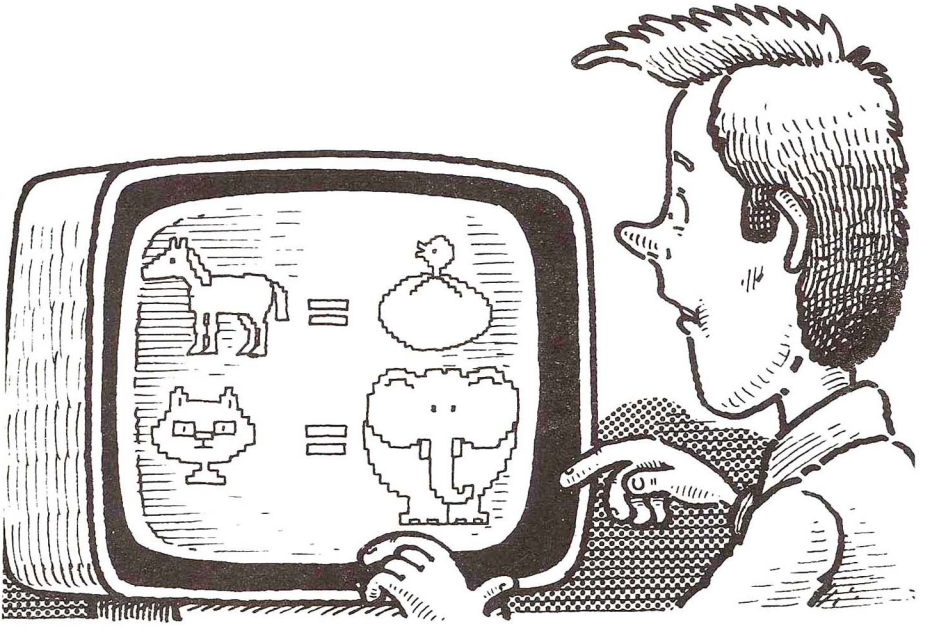
You can imagine an array as a list of items, one after the other, but there is a variety of array which allows a different kind of list, called a matrix. A matrix is a list of groups or items, with all the items in a group related. We could think of a matrix as a set of rows and columns, with each group taking up a row, and the items of a group in separate columns. Take a look at

figure 7.7 to see how this works. We use here a variable N$ which has two subscript numbers. The first number is the row number and the second is the column number. We need two FOR..NEXT loops to read data into this matrix. This is carried out in lines 30 to 60. The items are then printed in columns by the loop in lines 70 to 90. In this loop, the variable N% is used as the row number and we use the column numbers 1 and 2. The rows contain animal names, and the columns separate the different names that we use for adult and for young animals respectively. This example has used a *string matrix*, but a number matrix is also possible. If your maths has progressed to A-level or beyond, you probably know a variety of uses for number matrices, particularly in the solution of simultaneous equations. Once again, though, that's material for a more advanced book than this one.

Figure 7.8 shows a rather more ambitious matrix program. The idea is to store sets of names and telephone numbers which are fed in by you in the course of the loop in lines 30 to 90. Once the matrix has been filled, you can pick an initial letter for a name, and ask the computer to print out the name and number that it has located. I've left out mugtraps just to keep this example reasonably short, but you would certainly need some sort of mugtraps, even if only in the form of a message like:

```
PRINT" Sorry, can't find ";J$;" ENTRIES"
```

Normally, having set up something like this, you would want to use it more than once. A loop is needed, so that once you have found one name, you can go back to line 110 to pick another. You might also want to think about a suitable way of dealing with it when the name is not found. This will be when the program has finished without ever having run line 140. There will be clues about this later! Lastly, you might want to alter the program so as to make use of INSTR. The test for J$ being the first letter of a name could be better written by testing to see if what you typed was any part of the



name. In this way, you could find a telephone number for someone when you knew only part of the name. Before you rush in and make this into a working program, I should perhaps tell you that due to the expertise of our elected representatives in drawing up a law to protect people against abuse of databases by large companies, it might be illegal to have a name-and-number program on your computer without registering under the Data Protection Act. At present, only about half of the major computer users in the country have registered, and the rest are pointing out that it's also illegal to videotape TV programs, but at least they don't charge you £22 to check if you do or not!

One final point. When your memory is packed with large arrays, programs will run slowly because of lack of room. Once you have finished using an array in a program, then, it's useful to be able to erase it, and this can be done with ERASE, used in the form `ERASE A$,B,C#`, using the array name but with no brackets or numbers.

Chapter 8

Menus, subroutines and programs

Several chapters ago, we bumped into the idea of making a choice, by pressing a single key. In that example, the choice of keys was limited, Y or N. A choice of two items, isn't exactly generous, and we can extend the choice by a program routine that is called a Menu. A menu is a list of choices, usually of program actions. By picking one of these choices, we can cause a section of the program to be run. This is an important idea, because it allows us to break programs into small sections, and it also introduces some very important ideas in program design. One way of making the choice is by numbering the menu items, and typing the number of the one that you want to use. Figure 8.1 shows what a typical menu of this type would look like on the screen. With a computer which was fitted with stone-age BASIC, we could use a set of lines such as:

```
IF K =1 THEN 1ØØØ
IF K =2 THEN 2ØØØ
```

and so on. There is a much simpler method, however, which uses the Mallard BASIC instruction words, ON..GOTO or ON..GOSUB. These represent two different ways of carrying out the same task, and we'll look at both of them closely.

To start with, suppose we want to pick from four items by typing numbers 1 to 4 on the keyboard. The first thing that you have to ensure is that only the numbers 1 to 4 are accepted, not numbers like 0, 5, –10 and so on – in other words, a bit of mugtrapping. The second point is that the use of INPUT is a bit tedious, because you have to press RETURN after you have typed your number-key. We'll use INKEY$, then, to get the reply (the number-key) for single-digit numbers,then convert to number form, and it only remains to check that you have chosen a number that is in the correct range. Following this check, we'll use the number that was entered to find a line number and run the piece of program that starts at this line.

In the example of figure 8.2, lines 10 to 60 print a title and a menu, using TABs to make the printing reasonably neat. This is followed by advice on how to choose (it might be obvious to you, but not to any other user!). The

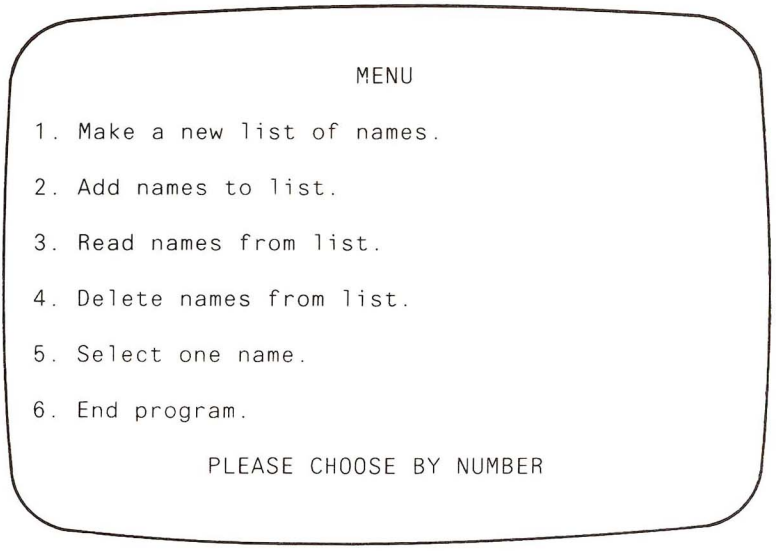


Figure 8.1 A typical menu as it would appear on the screen.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT TAB(27)CHR$(27)+"pMENU"+CHR$(27)+"q"
30 PRINT:PRINT TAB(10)"1. Daily takings."
40 PRINT:PRINT TAB(10)"2. Stock situation."
50 PRINT:PRINT TAB(10)"3. Purchases."
60 PRINT:PRINT TAB(10)"4. Summary."
70 PRINT:PRINT TAB(5)"Please select by number
 1-4"
80 K$=INKEY$:IF K$=""THEN 80
90 K%=VAL(K$)
100 IF K%>4 OR K%<1 THEN PRINT"Incorrect rang
e- please try again":GOTO 70
110 ON K% GOTO 130,150,170,190
120 END
130 PRINT"Takings answer"
140 GOTO 120
150 PRINT"Stock answer"
160 GOTO 120
170 PRINT"Purchase answer"
180 GOTO 120
190 PRINT"Summary here"
200 GOTO 120
```

Figure 8.2 A menu choice which uses INKEY$ and VAL to obtain the selected number, then selects via ON K% GOTO.

action starts in line 80. In this line there is an INKEY$ loop, which will keep looping while no key is pressed. If a key *is* pressed, however, K$ will have a value which is a string, and the next line converts this into a number. Note that you can't use a WHILE..WEND loop in a straightforward way here because you need to make the assignment to K$. Line 100 tests this number, to make sure that it is in the range of 1 to 4 that comprises our menu. If the number is not in the correct range, you get a polite message (it keeps the user on your side!), and a reminder of what the correct range is. If a letter key was pressed, incidentally, the K% = VAL(K$) step will convert the letter string into the number 0, and this will be rejected by line 100. Never put a number choice of zero into a menu, because it's a lot harder to distinguish between letter and number then!

At line 110, with the value of K% in the correct range, the choice is made by the statement

```
ON K% GOTO 13Ø,15Ø,17Ø,19Ø
```

This list *must* be in the order that will serve the menu choices. In other words, the routine which starts at line 130 should attend to the needs of menu item 1, the line which starts at 150 should attend to the needs of menu item 2, and so on. These lines need not be numbered 130, 150 and so on, of course. The numbering might have been 7000, 600, 150, 2010 *provided that these were the correct starting lines* for the relevant routines. In this example, each 'routine' here is just a printed message followed by a GOTO 120. This makes sure that the program ends after each routine. If we had used GOTO 10, we would have made the program RETURN to the menu. This is very often what is needed. When the program always returns to the menu, one of the menu choices should be `End this program`, so that you can stop without having to switch off the machine.

Sectional programming

A lot of programs consist of a title, some instructions, and then a menu. Depending on what menu choice you have made, some part of the program is run, and the program ends, or returns to the menu again. The ON K GOTO type of menu selection is useful, but an even more useful method makes use of subroutines. A subroutine is a section of program which can be inserted anywhere that you like in a longer program. A subroutine is inserted by typing the instruction word GOSUB, followed by the line number in which the subroutine starts. When your program comes to this instruction, it will jump to the line number that follows GOSUB, just as if you had used GOTO. Unlike GOTO, however, GOSUB offers an *automatic* RETURN. The word RETURN is used at the end of the subroutine lines, and it will cause the program to RETURN to the point immediately following the GOSUB that called it. Figure 8.3 illustrates this. When the program runs, line 10 contains GOSUB 1000 in place of the usual screen-clearing routine. In fact, what has been placed at line 1000 *is* the usual screen-clear action, and it is followed in line 1010 by RETURN. What happens, then, is that the action shifts to line 1000 to clear the screen, then

```
10 GOSUB 1000
20 PRINT"this is a ";
30 x%=41:y%=12
40 GOSUB 1020
50 GOSUB 1040
60 GOSUB 1060
70 GOSUB 1080
80 x%=10:y%=23
90 GOSUB 1020
100 PRINT"subroutine"
110 END
1000 PRINT CHR$(27)+"E"+CHR$(27)+"H";
1010 RETURN
1020 PRINT CHR$(27)+"Y"+CHR$(32+y%)+CHR$(32+x%);
1030 RETURN
1040 PRINT CHR$(27)+"p";
1050 RETURN
1060 PRINT"YELLOW";
1070 RETURN
1080 PRINT CHR$(27)+"q";
1090 RETURN
```

Figure 8.3 Using a subroutine – this is the key to more advanced programming.

to line 1010, which brings it back to line 20, the instruction immediately following line 10. Line 20 then prints a phrase, with the semicolon used to prevent a new line from being selected. Line 30 then assigns two variables, x% and y% which are going to be used to position the cursor, using another ESC sequence. The important point here is that x% and y% will be used in the subroutine, but their values can be allocated at this point. We describe this as *passing parameters to the subroutine*. The GOSUB 1020 in line 40 then causes the cursor to be shifted to line 12, column 41 by using the values of x% and y% in the ESC sequence – this sequence is illustrated in the manual so I won't describe it in detail here. Once again, the RETURN in the next line, 1030, makes the program RETURN to just following where it came from, to line 50 this time. In line 50, there's another GOSUB, this time to line 1040. This one uses ESC "p" to switch on inverse video, and the RETURN will this time be to line 60. Here again there is another GOSUB, to print the word YELLOW. Because of the subroutine of line 1040, this word will be printed in inverse video. Next comes line 70 which calls to line 1080 to switch off the inverse video, and control then goes to line 80. In this line, x% and y% are allocated again with different values that will put the cursor at the bottom of the screen and close to the left-hand side. After calling the subroutine that operates this, in line 1020, line 100 prints the word "subroutine", and the program ends. This example is, of course, of a yellow subroutine.

Now there's a lot here to ponder over. It's a clumsy-looking example, of course, mainly because it's an example for the sake of an example and doesn't really do anything useful. The points that it demonstrates are important, however. One is that any actions that you might want to use

more than once should be put into the form of a subroutine. In this way, instead of typing a set of instructions over and over again, you type them once and use a GOSUB each time you need them. The screen-clear action is a good candidate for this type of treatment, though it's even better to assign it as a string. In the course of any serious program you are likely to need the screen cleared many times. One method, then, is to put the

```
?CHR$(27)+"E"+CHR$(27)+"H" :RETURN
```

into a line at the end of the program and use a GOSUB to this line at each place in the program where you might want the screen cleared. Even this one routine put into this form can save you a lot of typing, and you only have to get it right once!

The next point is that a subroutine can make use of variables that have been assigned in the rest of the program (or even in another subroutine). The subroutine that shifts the cursor position, for example, makes use of variables for line number and column number, so that the subroutine can remain the same no matter where you want to move the cursor. It would be absurd to have to type out the whole routine again just because you wanted the cursor at line 10, column 20, and yet without this facility that's just what you would have to do. By using variables in the subroutine that can be allocated with values in the main program, we can once again use a single subroutine over and over again.

Subroutines are useful even if they are used only once. The reason is that they permit you to break a program into sections. For example, suppose that you are writing a line that prints out some data. You know that at that part of the program you want to print data in a particular form, but you haven't yet decided what might be best, and you might want to alter it. You can type a line such as:

```
500 FOR N%=1 TO 22:GOSUB 2000:NEXT
```

in which the GOSUB 2000 is the printing action that you haven't written yet! Note that you can put a GOSUB as part of a multistatement line like this, and when the subroutine returns it will be to the NEXT part of this statement, not to the next line of the program. Subroutines offer us a way of designing programs in a simple and staightforward way, and this is of even greater importance than their other useful actions, as we'll see later.

Now for something more serious. Figure 8.4 shows subroutines in use as part of a (very imaginary) games program. Lines 10 to 80 offer choice of evil objects, and line 90 invites you to choose. The choice is made by using a string AN$ which contains the permitted answers, and by calling a routine. This routine contains the familiar INKEY$, but in an unfamiliar form, in line 10000 on. The idea here is to assign a string to INKEY$ and then test this string for being contained in the permitted list. The WHILE..WEND loop will continue until a suitable answer is obtained. There is no error message delivered with this type of routine (courtesy of David Foster), but since the list of permitted answers is staring you in the face, a message is redundant anyhow. The subroutine assigns K% with the number value of K$, whatever that happens to be provided that this is in the approved list, and then returns; in this case to line 110. This is the line that causes the choice to be carried out, with ON K% GOSUB acting to direct the GOSUB to the correct line in the same way as ON K% GOTO. This time, however, the

```
10 GOSUB 500:PRINT
20 PRINT TAB(10)"Choose your monster."
30 PRINT
40 PRINT TAB(5)"1. Vampire.":PRINT
50 PRINT TAB(5)"2. Werewolf.":PRINT
60 PRINT TAB(5)"3. Zombie.":PRINT
70 PRINT TAB(5)"4. Dalek":PRINT
80 PRINT TAB(5)"5. Flying picket.":PRINT
90 PRINT:PRINT TAB(10)"Select by number, plea
se.":PRINT:PRINT
100 ANS$="12345":GOSUB 10000:REM inkey$ routi
ne
110 ON K% GOSUB 1000,2000,3000,4000,5000
120 PRINT:PRINT"Want another choice? Type Y o
r N"
130 ANS$="YN":GOSUB 10000:IF K%=1 THEN 10
140 END
500 PRINT CHR$(27)+"E"+CHR$(27)+"H":RETURN
1000 PRINT:PRINT"Blood, blood, from the Vampi
re-State building."
1010 RETURN
2000 PRINT"Howl, moan, I'm a liberated animal"
2010 RETURN
3000 PRINT"Obey, obey, it's in the post - I'l
l ring you back"
3010 RETURN
4000 PRINT"Exterminate, exterminate, this is
 Dalek-lib"
4010 RETURN
5000 PRINT"Blood, howl, exterminate - where
do we go next?"
5010 RETURN
10000 K%=0:K$="":WHILE K%=0 OR K$="":K$=UPPE
R$(INKEY$)
10010 K%=INSTR(ANS$,K$):WEND
10020 RETURN
```

Figure 8.4 Using ON K% GOSUB in a (very) imaginary game.

program will RETURN to whatever follows the choice, line 120. For example, if you pressed key 1, then the subroutine that starts at line 1000 is carried out, and the program returns to line 120 to check if you might also want subroutines 2000, 3000 4000, or 5000. Line 130 then makes use of the subroutine again, with the choice now contained in the string YN, and K% giving the position number of 1 for Y and 2 for N. One thing that you have to watch for is that a choice number such as K% is not used in any of the subroutines. Suppose, for example, that you picked choice 1. This makes K% = 1, and the first subroutine runs. Now suppose that this subroutine,

by some oversight, used the variable name of K%, and assigned this with the number 4. The result would be to run the subroutine at line 4000! This is because ON K% GOSUB works like any other GOSUB, and returns immediately following where it has been called. If K% = 1 ran the first subroutine, then this would RETURN to line 110 just following the 1000 choice, so that another choice will be made if K% is not at a different value. This can cause a lot of confusion, and the golden rule to avoid it is to make the selecting variable one that you don't use in any subroutine. It helps, in fact, if you make the name something more elaborate, like `select%`, to remind you about this important point.

A subroutine is extremely useful in menu choices, but it's even more useful for pieces of program that will be used several times in a program. In figure 8.4, by way of an example, the INKEY$ routine has been written as a subroutine, because it's one that you are likely to use many times in the course of any program. Putting the INKEY$ into a subroutine means that you need to type these program lines once only. Wherever you need a select action, you simply assign the ANS$, then type GOSUB 10000 (or whatever line number you have used), and the routine will be used when the program runs. Notice that in each case, the subroutine is placed in lines that can't normally be RUN. If you removed the GOSUB instructions from these programs, the subroutines couldn't run, because there is an END or STOP line before the computer could naturally get to the subroutine. This is important, because if the computer gets to a subroutine line by accident (when GOSUB has not been used), the program will stop with an error message `Unexpected RETURN in 1Ø1Ø` – using whatever line number the RETURN is in. Getting into a subroutine the wrong way is called 'crashing through', and you must avoid it by placing an END at the end of your main program, before the subroutine lines start.

Figure 8.5 shows an elaboration on the INKEY$ subroutine. The trouble with INKEY$ is that it doesn't remind you that it's in use, there's no question mark printed as there is when you use INPUT. The subroutine in lines 1000 to 1030 remedies that by causing an asterisk to flash while you are thinking about which key to press. The asterisk is flashed by alternately printing the asterisk and a set of characters that will delete it and leave the cursor at the same position. The cursor is removed by using ESC "f", so

```
10 GOSUB 500
20 PRINT:PRINT"Choose 1 or 2, please"
30 GOSUB 1000
40 PRINT"Your choice was ";K$
50 END
500 PRINT CHR$(27)+"E"+CHR$(27)+"H":RETURN
1000 K$="":PRINT CHR$(27)+"f";:WHILE K$=""
1010 PRINT"*";:K$=INKEY$:GOSUB 2000
1020 PRINT CHR$(27)+"l"+CHR$(8);:GOSUB 2000:WEND
1030 PRINT CHR$(27)+"e";:RETURN
2000 FOR n=1 TO 200:NEXT:RETURN
```

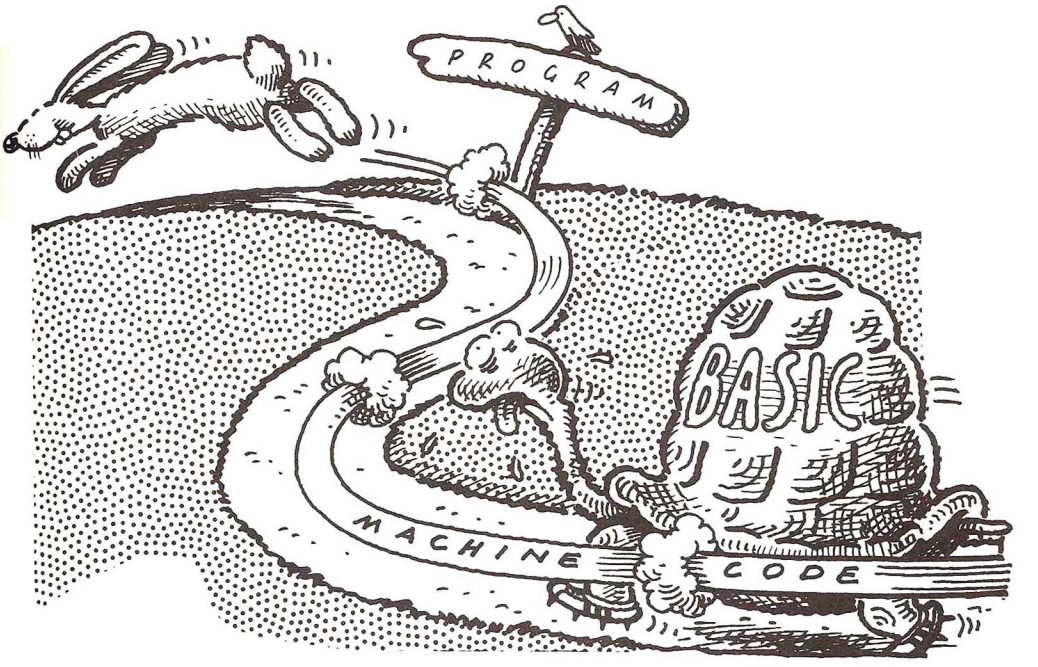
Figure 8.5 An INKEY$ subroutine that flashes an asterisk while waiting for a key to be pressed.

that the asterisk then acts as a reminder without any confusion caused by the cursor – if you remove this part of the listing you'll see why I removed the cursor. The routine then enters a WHILE..WEND loop which will continue until K$ acquires a value by the pressing of a key. In the loop, an asterisk is printed at the cursor position, and a time delay is used (GOSUB 2000) to keep this on screen for a short time. The asterisk is then removed by using ESC "l" (that's a lower-case L, not a number one), which deletes the line, and also CHR$(8) which causes the cursor to move one step back again. Once again, there is a time delay before the action repeats. Try this pair of subroutines in your programs, and see what a difference they make. Meantime, make friends with subroutines. They are not just a useful way of obtaining an action at several points in a program, they are an indispensible aid to program planning, of which there's much more just about to land on you.

Your own show

You can get a lot of enjoyment from your PCW series computer when you use it to enter programs from discs that you have bought. You can obtain even more enjoyment from typing in programs, using Mallard BASIC, that you have seen printed in magazines such as *Personal Computer World*. Even more rewarding is modifying one of these programs so that it behaves in a rather different way, making it do what suits you. The pinnacle of satisfaction, as far as computing is concerned, however, is achieved when you design your own programs. These don't have to be masterpieces. Just to have decided what you want, written it as a program, entered it and made it work is enough. It's 100% your own work, and you'll enjoy it all the more for that. After all, buying a computer and not programming it yourself is like buying a Ferrari and getting someone else to drive it for you.

Now I can't tell in advance what your interests in programs might be. Some readers might want to design programs that will keep tabs on a stamp collection, a record collection, a set of notes on food preparation or the technical details of vintage steam locomotives. Programs of this type are called *database* programs, because they need a lot of data items to be typed in and recorded. On the other hand, you might be interested in adventure games, expert systems, or producing graphics patterns with the printer. Programs of that type need instructions that are beyond the scope of this book, and it's not likely that these were the applications that most buyers of the PCW machine had in mind. What we are going to look at in this section, then, is the database type of program, because it's designed in a way that can be used for all types of programs. Once you can design simple programs of this type, you can progress, using the same methods, to design your own business programs with Mallard BASIC. Remember, though, that most of the very fast moving or elaborate programs that you see are not written in BASIC. The reason is that BASIC is interpreted, and unless you have Mallard BASIC loaded in, you can't run a Mallard BASIC program. Most of the bought business programs are written in machine code, a set of number-coded instructions aimed directly at the microprocessor at the heart of the computer. This type of code makes no use of BASIC and doesn't need to have BASIC taking up any space in the memory. Such coding is also very much more difficult to write. A lot of the machine code



programs are, in fact, written with the aid of other programs (compilers) on much bigger computers, and then loaded into the smaller computers. If you learn how to design programs in BASIC, however, you will be able to learn other languages or machine code later. All you need is experience – a lot of it.

Two points are important here. One is that *good* experience counts in the design business. If you make your first efforts at design as simple as possible, you'll learn much more from them. That's because you're more likely to succeed with a simple program first time round. You'll learn more from designing a simple program that works, perhaps after a bit of thought and modification, than from an elaborate program that never seems to do what it should. The second point is that program design has to start with the computer switched off, preferably in another room! Program design needs planning, and you can't plan properly when you have temptation in the shape of a keyboard in front of you. Get away from it!

Put it on paper

We start, then, with a pad of paper. For myself, I use a student's pad which is punched so that the sheets can be stored in a ring binder. This way, I can keep the sheets tidy, and add to them as I need. I can also throw away any sheets I don't need, which is just as important. Even a very simple program is probably going to need more than one sheet of paper for its design. If you then go in for more elaborate programs, you may easily find yourself with a

Aims

1. Present the name of an animal on the screen, picked at random.
2. Ask what its young is called.
3. Reply must be correctly spelled.
4. User must not be able to read correct answer from the listing.
5. Give one point for each correct answer.
6. Allow two chances at each question.
7. Keep a track of the number of attempts.
8. Present a score as number of correct answers out of number of attempts.

Figure 8.6 program outline ... your starter!

couple of dozen sheets of planning and of listing before you get to the keyboard. Just to make the exercise more interesting, I'll take an example of a program, and design it as we go. This will be a very simple program for the use of young children, but it will illustrate nearly all of the skills that you need.

Start, then, by writing down what you expect the program to do. You might think that you don't need to do this, because you know what you want, but you'd be surprised. There's an old saying about not being able to see the wood for the trees, and it applies very forcefully to designing programs. If you don't write down what you expect a program to do, it's odds on that the program will never do it! The reason is that you get so involved in details when you start writing the lines of BASIC that it's astonishingly easy to forget what it's all for. If you write it down, you'll have a goal to aim for, and that's as important in program design as it is in life. Don't just dash down a few words. Take some time about it, and consider what you want the program to be able to do. If you don't know, you can't program it! What is even more important is that this action of writing down what you expect a program to do gives you a chance to design a properly structured program. Structured in this sense means that the program is put together in a way that is a logical sequence, so that it is easy to add to, change, or redesign. If you learn to program in this way, your programs will be easy to understand, take less time to get working, and will be easy to extend so that they do more than you intended at first.

As an example, take a look at figure 8.6. This shows a program outline plan for a simple game. The aim of the game is to become familiar with the names of animals and their young. The program plan shows what I expect of this game. It must present the name of an animal, picked at random, on the screen, and then ask what the name of its young is. A little bit more thought produces some additional points. The name of the young animal will have to be correctly spelled. A little bit of trickery will be needed to prevent the user (son, daughter, brother, or sister) from finding the answers by typing LIST and looking for the DATA lines. Every game must have some sort of scoring system, so we allow one point for each correct answer. Since spelling is important, perhaps we should allow more than one try at

1. Display title, then instructions.
2. Display name of animal.
3. Ask for name of young.
4. Use INPUT for reply.
5. Compare reply with correct answer.
6. If correct, add 1 to score, and ask if another one is wanted.
7. If incorrect first reply, allow another try without incrementing number of tries.
8. If second reply incorrect, select another question.
9. Game ends when user types **n** in reply to 'Do you want another one'.

Figure 8.7 The next stage in expanding the outline.

each question. Finally, we should keep track of the number of attempts and the number of correct answers, and present this as the score at the end of each game. Now this is about as much detail as we need, unless we want to make the game more elaborate. For a first effort, this is quite enough. How do we start the design from this point on?

The answer is to design the program in the way that an artist paints a picture or an architect designs a house. That means designing the outlines first, and the details later. The outlines of this program are the steps that make up the sequence of actions. We shall, for example, want to have a title displayed. Give the user time to read this, and then show instructions. There's little doubt that we shall want to do things like assign variable names, dimension arrays, and other such preparation. We then need to play the game. The next thing is to find the score, and then ask the user if another game is wanted. Yes, you have to put it all down on paper! Figure 8.7 shows what this might look like at this stage.

Foundation stones

Now at last, we can start writing a section of program. This will just be a foundation, though. What you must avoid at all costs is filling pages with BASIC lines at this stage. As any builder will tell you, the foundation counts for a lot. Get it right, and you have decided how good the rest of the structure will be. The main thing you have to avoid now is building a wall before the foundation is complete!

Figure 8.8 shows what you should aim for at this stage. There are only nine lines of program here, and that's as much as you want. This is a foundation, remember, not the Empire State Building. It's also a program that is being developed, so we've hung some 'danger – men at work' signs around. These take the form of the lines that start with REM. REM means REMinder, and any line of a program that starts with REM will be ignored by the computer. This applies also to a multi-statement line, so that though the computer will get as far as the REM part in each of these lines, it will ignore

```
10 GOSUB 800:REM clear screen
20 GOSUB 1000:REM title
30 GOSUB 1200:REM instructions
40 GOSUB 1400:REM setup
50 ans%=1:WHILE ans%=1:GOSUB 2000:REM play
60 GOSUB 3000:REM score
70 GOSUB 4000:REM ask for another
80 WEND
90 END
```

Figure 8.8 A 'core' or 'foundation' program for the example.

anything that follows. That means, incidentally, that you should not try to put any instructions in the same line following a REM. You can type whatever you like following REM, and the point of it all is to allow you to put notes in with the program. These notes will not be printed on the screen when you are using the program, and you will see them only when you LIST. In figure 8.8, I have put the REM notes on the same lines as the GOSUBs. A more common way is to use lines which are numbered just 1 more than the main GOSUB lines. This way, it's easy to remove all the REM lines later. How much later? When the program is complete, tested, and working perfectly. REMs are useful, but they make a program take up more space in memory, and run slightly slower. I always like to keep one copy of a program with the REMs in place, and another 'working' copy which has no REMs. That way I have a fast and efficient program for everyday use, and a fully detailed version that I can use if I want to make changes.

Let's get back to the program itself. As you can see, it consists of a set of GOSUB instructions, with references to lines that we haven't written yet. That's intentional. What we want at this point, remember, is foundations. The program follows the plan of figure 8.7 exactly, and each part of it is carried out by a GOSUB. The variable `ans%` is being used to keep the loop running, and we shall write a subroutine which will use INKEY$ to look for a `y` or `Y` being pressed, and then allocate `ans%` accordingly. This will be in the subroutine at line 4000, which deals with the `Do you want another game?` step that we planned for earlier.

Take a good long look at this nine-line piece of program, because it's important. The use of all the subroutines means that we can check this program easily – there isn't much to go wrong with it. We can now decide in what order we are going to write the subroutines. The wrong order, in practically every example, is the order in which they appear. Always write the title and instructions last, because they are the least important to you at this stage. In any case, if you write them too early, it's odds on that you will have some bright ideas about improving the game soon enough, and you will have to write the instructions all over again. A good idea at this stage is to write a line such as:

```
1 GO TO 40
```

which will cause the program to skip over the title and instructions. This saves a lot of time when you are testing the program, because you don't have the delay of printing the title and instructions each time you run it.

The next step is to get to the keyboard (at last, at last) and enter this core program. If you use the GO TO step to skip round the title and instructions temporarily, you can then put in simple PRINT lines at each subroutine line number. We did this, you remember, in the program of figure 8.4, so you know how to go about it. To save typing, these can be simple REM and RETURN lines, and you can try an INPUT ans%:RETURN for the routine at line 4000. This allows you to test your core program and be sure that it will work before you go any further.

The next step is to record this core program, with its temporary GOTOs and REMs, and then keep adding to the core. If you have the core recorded, then you can load this into your computer, add one of the subroutines, and then test, using a STOP line to prevent the program going further than you want. When you are satisfied that it works, you can record the whole lot under another filename. Next time you want to add a subroutine, you start with this version, and so on. This way, you keep files of a steadily-growing program, with each stage tested and known to work. Again, this is important. Very often, testing takes longer than you expect, and it can be a very tedious job when you have a long program to work with. By testing each subroutine as you go, you know that you can have confidence in the earlier parts of the program, and you can concentrate on errors in the new sections.

Designing the subroutines

The next thing we have to do is to design the subroutines. Now some of these may not need much designing. Take, for example, the subroutine that is to be placed in line 4000. This is just the type of INKEY$ routine that we looked at earlier, along with a bit of PRINT, so we can deal with it right away. Figure 8.9 shows the form it might take. The subroutine is straightforward, and that's why we can deal with it right away! Type it in, and now test the core program with this subroutine in place. Incidentally, there's a bit of cunning here, because this subroutine can be used in two ways. By using GOSUB 4000, you get the full treatment of message and Y/N choice. If you use GOSUB 4020, however, you can assign your own ANS$ earlier, and use this part to make whatever selection you want! This type of programming is something that you learn by experience, and by modifying programs so that they take up as little space as possible.

Now we come to what you might think is the hardest part of the job – the subroutine which carries out the 'Play' action. In fact, you don't have to

```
4000 PRINT"Would you like another game?"
4010 PRINT"- please answer Y or N.":ANS$="YN"
4020 K$="":WHILE K$="" OR ans%=0
4030 K$=UPPER$(INKEY$):ans%=INSTR(ANS$,K$)
4040 WEND:RETURN
```

Figure 8.9 The 'askmore' subroutine, written starting at line 4000.

1. Keep the answers as DATA lines containing ASCII codes.
2. Keep list of animals in another DATA line, to be placed in a string.
3. The number that selects the animal also selects the data line for the answer, using RESTORE.
4. Use variable TR% for number of attempts.
5. Use variable SC% for score.
6. Use variable GO% to record number of attempts at one question.

Figure 8.10 Planning the 'Play' subroutine.

learn anything new to do this. The Play subroutine is designed in exactly the same way as we designed the core program. That means we have to write down what we expect it to do, and then arrange the steps that will carry out the action. If there's anything that seems to need more thought, we can relegate it to a subroutine to be dealt with later.

As an example, take a look at figure 8.10. This is a plan for the Play subroutine, which also includes information that we shall need for the setting-up steps. The first item is the result of a bit of thought. We wanted, you remember, to be sure that some smart user would not cheat by looking up the answers in the DATA lines. The simplest deterrent is to make the answers in the form of ASCII codes. It won't deter the more skilled, but it will do for starters. I've decided to put one answer in each DATA line in the form of a string of ASCII codes, with each code written as a three-figure number. Why three figures? Well, the capital letters will use two figures only, the small letters three, so making them all into three figures simplifies things. You'll see why later – what we do is to write a number like 86 as 086, and so on. That's the first item for this subroutine.

The next item allows us to keep the names of the animals in an array. This has several advantages. One of them is that it's beautifully easy to select one at random if we do this. The other is that it also makes it easy to match the answers to the questions. If the questions are items of an array whose subscript numbers are 1 to 10, then we can place the answers in DATA lines, one set of numbers in each data line, and read these also as a string array.

The next thing that the plan settles is the names that we shall use for variables. It always helps if we can use names that remind us of what the variables are supposed to represent. In this case, using SC% for the score and

```
2000 GO%=0:V%=1+INT(10*RND(1))
2010 PRINT CLS$:PRINT"The animal is - "Q$(V%)
2020 PRINT:PRINT"The young is called - ";
2030 INPUT X$:TR%=TR%+1
2040 GOSUB 5000:REM correct answer
2050 RETURN
```

Figure 8.11 The 'pick' subroutine that generates the random number and picks a name.

TR% for the number of tries looks self-explanatory. The third one, GO% is one that we shall use to count how many times one question is attempted. Finally, we decide on a name for the array that will hold the animal names – Q$.

Play it once at least, Sam

Figure 8.11 shows what I've ended up with as a result of the plan in figure 8.10. The steps are to pick a random number, use it to print an animal name, and then find the answer. That's all, because the checking of the answer and the scoring is dealt with by another subroutine. Always try to split up the program as much as possible, so that you don't have to write huge chunks at a time. As it is, I've had to put another subroutine into this one to keep things short.

We start the subroutine at line 2000 by *clearing* a variable. The size of GO% is set to 0, to make absolutely sure that this variable has the correct size each time this subroutine is started. If this variable has value 0, then two goes at the answer are allowed. If GO% = 1, then only one more attempt can be allowed. The second part of line 2000 then picks a number, at random, lying between 1 and 10. Lines 2010 to 2030 are straightforward stuff, except for the PRINT CLS$. This is a replacement for using a subroutine to clear the screen. In the preliminary suroutine we shall assign CLS$ to a string of ESC characters that will clear the screen and home the cursor. This is easier than using either a subroutine or a defined function. We then print the name of the animal that corresponds to the random number, and ask for an answer, the young of that animal. The last section of line 2030 counts the number of attempts. This is the logical place to put this step, because we want to make the count each time there is an answer. Now it's chicken-out time. I don't want to get involved in the reading of ASCII codes right now, so I'll leave it to a subroutine, starting in line 5000, which I'll write later. The REM in line 2040 reminds me what this new subroutine will have to do, and the Play subroutine ends with the usual RETURN.

Details, details

With the Play subroutine safely on disc, we can think now about the details. The first one to look at should be one that either precedes or follows the Play step, and I've chosen the Score routine. As usual, it has to be planned, and figure 8.12 shows the plan. Each time that there is a correct answer, the number variable SC% will be incremented, and we can go back to the main program. More is needed if the answer does not match exactly. We need to print a message, and allow another go. If the result of this next go is not correct, that's an end to the attempts. Later on, after you have read chapter 11, you might want to improve on these details, perhaps by having question

1. For a correct answer, increment SC%, go to next question.
2. For a first incorrect answer, with GO% = 0, allow another try. Decrement TR%, and increment GO%.
3. For a second incorrect answer, with GO% = 1, pass to the next question, and make GO% = 0 again.

Figure 8.12 Planning the 'score' subroutine.

```
3000 WHILE GO%=0 AND X$<>A$:GOSUB 3300:WEND:R
EM wrong!
3010 IF X$=A$ THEN GOSUB 3200 ELSE GOSUB 3100
3020 RETURN
3100 PRINT"No luck - try the next one.":GOSUB
 7000:RETURN
3200 SC%=SC%+1:PRINT"Correct- your score is n
ow";SC%
3210 PRINT"in"TR%" attempts.":GOSUB 7000:RETUR
3300 PRINT"Not correct- but it might be your
spelling!"
3310 PRINT"You get another go free.":TR%=TR%-1
3320 GO%=1:GOSUB 7000:GOSUB 2010:RETURN
```

Figure 8.13 The 'score' subroutine written. This allows for the three possibilities of the plan.

and answer in different windows. Because each action has its own subroutine, it's easy to make these kinds of alterations because the rest of the program can be left undisturbed.

Figure 8.13 shows the program subroutine that has been developed from this plan. This is in several sections, with each section dealing with a different type of answer. We have several possible answers. First of all, the answer might be correct, in which case we want to increment the score. If the answer is incorrect, we have to check if this is a first go or a second. If this is a first go, then a message will announce this and another go is awarded, with the number of tries being decremented to allow the go to be a 'free' one. If this has been a second attempt at the same question and is still wrong, then a message will be printed and another name selected. Line 3000 deals with a first go, because GO% is 0, and an incorrect answer, because X$<>A$. The loop will run until there is a correct answer or until GO% = 1, and it includes the GOSUB 3300 to print the message, decrement TR% (free try), and GOSUB 2010 to get another answer. Notice here again that we are entering a subroutine one line on. We don't want to find another value of V%, but just repeat the old one, so line 2000 is skipped.

After line 3000 has checked for an incorrect answer, with GO% = 0, line 3010 runs. This checks for the answer when GO%=1, and if this is correct then the GOSUB 3200 runs; if incorrect then GOSUB 3100 runs. The

```
1400 TR%=0:SC%=0:GO%=0
1410 CLS$=CHR$(27)+"E"+CHR$(27)+"H"
1420 DIM Q$(10),A$(10)
1430 FOR J%=1 TO 10:READ Q$(J%):NEXT
1440 FOR J%=1 TO 10:READ A$(J%):NEXT
1450 RETURN
```

Figure 8.14 The 'setup' subroutine for dimensioning and array filling.

```
5000 A$="":FOR J%=1 TO LEN(A$(V%))STEP 3
5010 A$=A$+CHR$(VAL(MID$(A$(V%),J%,3))):NEXT
5020 RETURN
```

Figure 8.15 The lines that extract the ASCII codes and get the answer from them.

subroutine at 3200 increments the score and comes back, the one at 3100 prints a sympathetic message and comes back. In any event, they all come back to line 3020, which returns to the main program again. All of this might not seem to be the way that you would write it, so try for yourself. The thing in this example that never seems right to a beginner is the order of testing. This has been done to allow the use of a WHILE..WEND loop, rather than on a set of tests with a lot of GOTO's. You may not feel this way at the moment, but writing it in this way is easier to follow!

Now that we've got the bit between our teeth, we can polish off the rest of the subroutines. Figure 8.14 shows the subroutine that deals with dimensioning and arrays. Line 1400 sets all the variables for the scoring system to zero, and line 1410 defines CLS$, the string that wil clear the screen when printed. Line 1420 dimensions the array Q$ that will be used for the names of the animals, and A$ which will be used for the numbers that give the answers. Line 1430 then reads the names from a data list into the array Q$, and line 1440 reads the numbers into A$ – and that's it! We can write the DATA lines later, as usual.

Next comes the business of finding the answer. We have planned this, so it shouldn't need too much hassle. Figure 8.15 shows the program lines. The variable V% is the one that we have selected at random, and it's used to select one of the strings of ASCII numbers, A$(V%). Since each number consists of three digits, we want to slice this string three digits at a time, and that's why we use STEP 3 in the FOR..NEXT loop in line 5000. Line 5010 then builds up the answer string, which we call A$. Remember that A$, used alone, is not confused with the A$(V%) array. A$ is set to a blank in the first part of line 5000 to ensure that we always start with a blank string, not with the previous answer, which would also be A$. The string A$ is then built up by selecting three digits, converting to the form of a number by using VAL, then to a character by using CHR$. This character is then added to A$, and this continues until all the numbers in the string have been dealt with. That's the hard work over. Figure 8.16 is the subroutine for the instructions, and figure 8.17 is the title subroutine, including a pause. Finally, figure 8.18 shows the DATA lines and the delay subroutine.

```
1200 PRINT CLS$:PRINT TAB(39)"INSTRUCTIONS"
1210 PRINT:PRINT TAB(21)"The computer will su
pply you with the name of an"
1220 PRINT TAB(21)"animal. You should type th
e name of its young -"
1230 PRINT TAB(21)"and make sure that your sp
elling is correct and"
1240 PRINT TAB(21)"that you start each name w
ith a capital letter."
1250 PRINT TAB(21)"The computer will keep the
score for you. You"
1260 PRINT TAB(21)"get two shots at each name
."
1270 PRINT:PRINT TAB(31)"press the spacebar t
o start"
1280 ANS$=" ":GOSUB 4020:RETURN
```

Figure 8.16 The instructions – always leave these until you have almost finished.

```
1000 PRINT CLS$
1010 PRINT TAB(37)CHR$(27)+"pYOUNG  ANIMALS"+
CHR$(27)+"q"
1020 GOSUB 7000:RETURN
```

Figure 8.17 The title program lines.

```
6000 DATA Dog,Cat,Cow,Horse,Hen,Fox,Kangaroo,
Goose,Lion,Pig
6001 DATA 080117112112121
6002 DATA 07510511611610111O
6003 DATA 067097108102
6004 DATA 070111097108
6005 DATA 06710410509910710111O
6006 DATA 067117098
6007 DATA 074111101121
6008 DATA 07111111510810511O103
6009 DATA 067117098
6010 DATA 08010510310810111O
7000 FOR C=1 TO 2000:NEXT:RETURN
7010 RETURN
```

Figure 8.18 The DATA lines with the name of the animals and the coded answers.

Now we can put it all together, and try it out. Remember that Mallard BASIC allows you to record pieces of program, and then MERGE them together. Because it's been designed in sections like this, it's easy for you to modify it. You can use different DATA, for example. You can use a lot more data – but remember to change the DIM in line 1420. You can make it a question-and-answer game on something entirely different, just by changing the data and the instructions. You can add more insteresting screen effects like windows, inverse video, underlining and so on.. One major fault of the program as it stands is that once an item has been used, it can be picked again, because that's the sort of thing that RND can cause. You can get round this by swapping the item that has been picked with the last item (unless it *was* the last item), and then cutting down the number that you can pick from. For example, if you picked number 5, then swap numbers 5 and 10, then pick from 9 only. This means that:

```
1 + INT(1Ø*RND(1))
```

will become

```
1+INT(D*RND(1))
```

where D starts at 10, and is reduced by 1 each time a question has been answered correctly.

There's a lot, in fact, that you can do to make this program into something a lot more interesting. The reason that I have used it as an example is to show what you can design for yourself at this stage. Take this as a sort of BASIC 'construction set' to rebuild any way you like. It will give you some idea of the sense of achievement that you can get from mastering Mallard BASIC. As your experience grows, you will then be able to design programs that are very much longer and more elaborate than this one by a long way. As it is, in the two following chapters, we'll be diving into the subject of disc filing, which is the next logical step in programming.

There's just one more thing. Suppose your own program doesn't perform as you expect it to? How do you set about sorting out problems? For a brief look at this, read appendix D, which is concerned with editing and troubleshooting.

Chapter 9

BASIC filing techniques

What is a file?

I have used the word 'file' in the course of this book to mean a collection of information which we can record on a disc. Programs in BASIC are one type of file, and the only type, incidentally, which permits the use of the LOAD and SAVE commands, because both CP/M programs and *LocoScript* files are dealt with quite differently. As it happens, BASIC pograms can be saved in three different ways. The staightforward way, using something like `SAVE "file"`, saves the program in a coded form, with one code character representing each key reserved word in the program. Another form of coding is obtained when you use `SAVE"file",P`. this saves the program in *protected* form so that it can only be loaded and run by using `LOAD"file"`, then `RUN`, or `RUN"file"`, and cannot be stopped and listed unless you know how to break the protection (by reading the Amstrad programmer's manuals!). Another option is to use `SAVE "file",A`. This will save the program in text form (as a set of ASCII codes instead of as *tokens*), it is a more versatile type of file for BASIC programs because it can be used both by BASIC and by some CP/M routines, including CP/M word-processors (including ED, but *not LocoScript*). BASIC programs saved as ASCII files do take up rather more space on the disc, and do take marginally longer to save and to load, but these differences are hardly significant, even for fairly long programs. If you intend to exchange BASIC programs with users of other machines, perhaps through the CP/M User Group, you should save in this format. In this chapter, however, I shall use the word 'file' in a narrower sense. I'll take it to mean a collection of data that is *separate* from a program. For example, if you have a program that deals with your household accounts, you would need a file of items and money amounts. This file is the result of the data-gathering action of the program, and it preserves these amounts for the next time that you use the program. Taking another example, suppose that you devised a program which was intended to keep a note of your collection of vintage 78 rpm recordings. The program would require you to enter lots of information about these recordings, such as title, artists, catalogue number, recording company, date of recording, date of issue and so on.. This information is a file, and at some stage in the program, you would have to record this file.

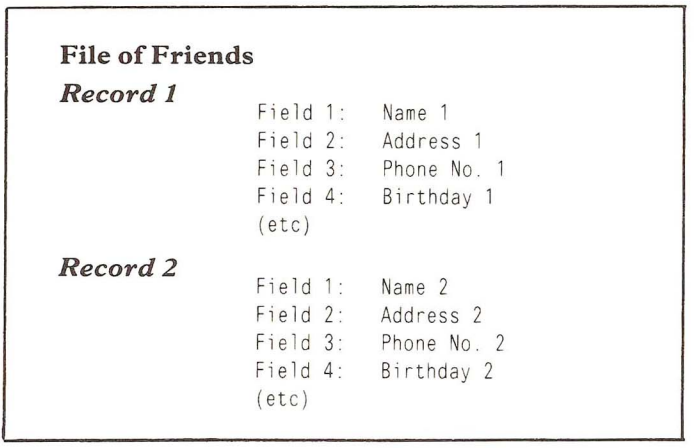


Figure 9.1 The meaning of *record* and *field*.

Why? Because when you load a BASIC program and RUN it, it starts from scratch. All the information that you fed into it the last time you used it has gone – unless you recorded that information separately. This is the topic that we're dealing with in this chapter, recording the information that a program uses. The shorter word is 'filing' the information. In this chapter, we look at the roots of the subject and at one type of filing that Mallard BASIC can carry out.

Knowing the names

You can't discuss filing without coming across some words which are always used in connection with filing. The most important of these words are 'record' and 'field', illustrated in Figure 9.1. A record is a set of facts about one item in the file. For example, if you have a file about vintage steam locomotives, one of your records might be used for each locomotive type. Within that record, you might have wheel formation, designers name, firebox area, working steam pressure, tractive force . . . and anything else that's relevant. Each of these items is a 'field', an item of the group that makes up a record. One record might, for example, be the *Scott* class 4-4-0 locomotives. Every different bit of information about the Scott class is a field, the whole set of fields is a record, and the Scott class is just one record in a file that will include the Gresley Pacifics, the 4-6-0 general purpose locos, and so on. Take another example, the file 'British motor-bikes'. In this file, B.S.A. is one record, A.J.S. is another, Norton is another. In each record, you will have fields. These might be capacity, number of cylinders, bore and stroke, gear ratios, suspension system, top speed, acceleration . . . and whatever else you want to take note of. Filing is fun – if you enjoy arranging things in order. The importance of filing is that all of the information can be recovered very quickly, and that it can be arranged in any order, or picked out as you choose. If you have a file on British Motor-bikes, for example, it's easy to get a list of machines in order of cylinder

capacity, or in order of power output, or any other order you like. You can also ask for a list of all machines under 250cc, which ones used four-speed gearboxes, which were vertical twins, which were two-strokes. Rearranging lists and picking out items is something which is a lot less easy when the information exists only on paper.

Disc filing

In this book, because we are dealing with Mallard BASIC and its very advanced filing systems, we'll ignore filing methods that are based on DATA lines in a BASIC program. Though you may be experienced with the use of filing with cassette systems on other machines, I'll explain filing from scratch in this chapter. If it's all familiar to you, please bear with me until I come to something that you haven't met before. To start with, there are two main types of files that we can use with a disc system, *serial files*, and *random access files*. The differences are simple, but very important ones. A serial (or *sequential*) file places all the information on a disc in the order in which the information is received, just as it would be placed on a cassette. If you want to get at one item, you have to read all of the items from the beginning of the file into the computer, and then select. There is no way in which you can command the system to read just one record or one field. More important, with such files you can't easily change any part of a record, or add more records in the middle of such a file. Any such action has to be done by reading the whole file and making alterations while the file in in memory, then re-recording the whole file. A random access file does what its name suggests – it allows you to get from the disc one selected record or field without reading every other one from the start of the file. You might imagine that, faced with this choice, no-one would want to use anything but random access files. It's not so simple as that, though, because the convenience of random access filing has to be paid for by a lot more complication. For one thing, because random access filing allows you to write data at any part of the disc, it would be very easy to wipe out valuable data with a program that was badly designed. There is also a problem of choosing which record to read. We'll start, then, by looking at serial files.

Serial filing

We'll start by supposing that we have a file to record, called `CAMERAS`. On this file we have records (such as `Nikon`, `Pentax`, `Canon`, `Yashica` and so on). For each record we have fields like `Model`, `Film Size`, `Shutter speed range`, `Aperture range (standard lens)`, `Manual` or `Automatic`, and so on. How do we write these records? First of all, we need to arrange the program that has created the records so that it can output them in some order. Figure 9.2 for example, shows how we might arrange this part of a BASIC program so as to input a number of records, with five fields to each record. The number of fields is five, so the fields are input from the keyboard using a `FOR N%=1 TO 5` loop. The number of records isn't fixed, so we use a GOTO loop, which keeps putting out records until it finds one called `X` or `x`, which is the terminator. If we can make a test for the

```
10 CLS$=CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT CLS$:RC$="":X%=0:DIM Field$(5)
30 PRINT TAB(40)"DATA ENTRY":PRINT:PRINT"Type
 X to end entry"
40 PRINT:INPUT"Record name - ";RC$:IF UPPER$(
RC$)="X"THEN 110
50 REM need to record this on disc
60 X%=X%+1:FOR N%=1 TO 5
70 PRINT"Field item "N%" ";:INPUT Field$(N%)
80 REM need to record this also
90 NEXT
100 GOTO 40
110 REM end of entry
120 PRINT"There are "X%" records on this file
."
```

Figure 9.2 The form of program lines that are needed to generate record and field data.

terminator at the start of a loop, then it's better to use a WHILE..WEND loop. If, however, the test must be made at the end of the loop, or in the middle, then we have no alternative but to use GOTO in Mallard BASIC, because there is no simple way out of the WHILE..WEND loop. Note that we haven't used an array for holding these items, because an array has to be dimensioned, and we don't know in advance how many items we will have. Instead of storing the items in an array for future use, they will be recorded on disc. The points where the disc recording routine would be fitted are shown in the REM lines 50 and 80. Each item, field or record, is treated as a string. This is because strings are easier to work with – you will not, for example, get any `Redo` messages at the INPUT stage because of a mismatch of variable type. The other good reason for using strings is that a string is a set of ASCII characters, and these files are *always* recorded using ASCII representation.

That deals with the organisation of the data for putting on to disc, but how do we actually put it on the disc? There are several stages, and the first one is to *open* the file. This means assigning a reference or *channel* number, and a *filename* (which will be recorded on the disc). There are two variations of this statement, one for writing and one for reading. The reference number is a purely temporary thing which is used from the time that a file is opened until it is closed. Having assigned a reference number, you must not OPEN again with the same number until the first file that used the number has been closed. Apart from that, though, you are free to assign reference numbers as you please. The default range of numbers is 1 to 3, but if you should ever need a larger range, this can be set by using `CLEAR,,,5` (or however many you want). The three commas are part of this command and must not be omitted – they mark where other quantities can also be changed by this statement. The OPEN statement will cause the disc to spin and the filename to be recorded.

Once the file has been opened for writing, with the filename on the disc, you can place data on to the disc using a statement of the form `PRINT#1` (or

WRITE#1). The 1 in this example is the same as the reference number or channel number that was used in the OPEN statement, and this is how the machine recognises files. Each time you want to make use of a file that has been opened, then, you must use this file reference number which has been established by the earlier OPEN statement.

The purpose of using the filename and the reference number is to organise data. The disc stores all data in units of 512 bytes. It wouldn't make sense to spin the disc and find a place on the disc just to record one byte at a time, so when you record or read a disc, it's always one complete sector, or as much of a sector as possible, at a time. Some of the memory of the computer has to be used to hold data which is being gathered up for recording, or which is being replayed. The reference number is an identifying number for the piece of memory, called a *buffer*, that is being used, so that the machine finds the correct data in the correct part of the memory. Using this reference number avoids the need for you to have to allocate parts of the memory to use in this way as buffers. The memory which is used for this purpose lies above the top of the useable range, and is changed when you use CLEAR, or the MEMORY command, to create more space for file buffers.

Opening the file

After that short diversion, back to our filing program. Before we start to gather the data together for filing, we need to open the file for the data. This is done using the OPEN"O",1," command. OPEN has to be followed by either "O" for output or "I" for input when you are working with a serial file, and following the comma is the reference number which will normally be in the range 1–3. You must then put in another comma, and then a filename of no more than eight characters. In addition, it helps if you can give the filename some useful extension label. The 'standard' extension for data is .DAT, so it makes sense to use this unless you have some pressing reason to use something else. When you are developing a program that will go through a large number of stages, it's a good idea to put the stage number into the main title as, for example, CAMERAØ1.DAT. When this first line is executed, the disc will spin for a short time, preparing for the file, and the filename will now appear on the directory/catalogue if you interrupt the program and type DIR. The buffer space will also be prepared in the memory and allocated to the reference number which in this example is 1. As always, you can place the drive letter ahead of a colon if you have more than one drive.

The use of the OPEN statement opens a file – which means that we can now make use of the file for writing data on to the disc. It also means that the disc is prepared for the file. Any file that exists on the disc already and has the same name of aircraft will *not* prevent you from opening this file. This means that you have to be rather careful about how you use files, because one file will replace another of the same name, making the old file a .BAK type. You can, of course, recover this old file by renaming it with REN or NAME. This feature makes it very easy to update and modify files, as we shall see. If you want to lock a file after using the progam, you can make use of the CP/M SET command. The manual mentions the use of the word LOCK as part of the OPEN statement, but this is not accepted as part of the Mallard BASIC for the PCW machines, only for larger versions of the language.

```
10 OPEN"O",1,"aircraft.dat"
20 RC$="":X%=0:CLS$=CHR$(27)+"E"+CHR$(27)+"H"
30 PRINT CLS$:PRINT TAB(40)"DATA ENTRY":PRINT
40 PRINT TAB(10)"Type X to end entry.":PRINT
50 WHILE UPPER$(RC$)<>"X":INPUT"Record Name -
  ";RC$
60 IF UPPER$(RC$)<>"X"THEN GOSUB 110
70 WEND
80 REM End of file
90 PRINT"There are"X%" records on the file."
100 CLOSE:END
110 PRINT #1,RC$
120 X%=X%+1:FOR N%=1 TO 5
130 PRINT"Field item ";N%" is ";
140 INPUT Field$
150 PRINT #1,Field$
160 NEXT:RETURN
```

Figure 9.3 Printing data to a disc file. The CLOSE statement is **very** important.

Printing to the file

It's at this stage that we need to make use of loops in the writing program. Within these loops, we need to have a line something like:

```
PRINT#1,Field$
```

in which `PRINT#` means put the information out to the buffer whose reference number is 1, the reference number that was selected by the OPEN statement, so that `PRINT#1` will *eventually* put out to the disc system the data that follows. The PRINT#1 statement can be used exactly as you would use the PRINT statement, and can be followed by strings, numbers, commas and so on just as a PRINT would be. There's an alternative, WRITE#1, which is used particularly where strings are concerned and for most general purposes, WRITE#1 is more useful if you are recording more than one item in each statement. In this chapter, you will see examples of both in use. In this example, what is being printed to the file is `Field$`. There is no need to use an array, such as `Field$(N%)` here, because the items are recorded to the disc/buffer in order and no array is needed. We also need to write the record *name*, and this is done within the loop, by using a line such as:

```
PRINT#1,RC$
```

Figure 9.3 shows the complete example of a short and simple program of this type which has been adapted from the first example. The programming has been changed to make use of a WHILE..WEND loop, with a subroutine used for the field inputs and the disc filing statement. You can enter anything you like into this, but it makes more sense to enter something that you can easily check. Since the file is called `aircraft.dat`, you could make each record name the name of an aircraft type, and each field some feature of the aircraft, like country of origin, aircraft type, engine details, weight

```
10 CLS$=CHR$(27)+"E"+CHR$(27)+"H"
20 OPEN"I",2,"aircraft.dat"
30 WHILE NOT EOF(2):PRINT CLS$:PRINT TAB(37)"
AIRCRAFT DETAILS"
40 INPUT #2,Name$
50 PRINT:PRINT TAB(10)"Name is "Name$:RESTORE
60 PRINT:FOR n%=1 TO 5
70 INPUT #2,Gen$:READ Field$
80 PRINT:PRINT TAB(15) field$;"  ";Gen$
90 NEXT
100 PRINT:PRINT"Press spacebar for next record"
110 WHILE INKEY$="":WEND
120 WEND
130 CLOSE:PRINT"END":END
140 DATA Country of origin,Type,Power,Empty wei
ght,Accommodation
```

Figure 9.4 Reading a file from disc until the EOF character is read. Using this 'terminator', you do not need to know how long the file is.

unladen, number of crew, and so on. You can, of course, easily change this program so that it has another title that suits the information that you might want to use. Before we move on, consider what this program has done. It has created a file called `aircraft.dat`, and allocated a channel number of 1 to this file. It has then stored the data as it came along, in the sequence of RECORD, then FIELDS. Finally, the file has been recorded and closed by using CLOSE. This last step is *very* important. For one thing, you don't actually record on the disc *any of the information* in this short program until the CLOSE statement is executed. That's because it would be a very time-consuming business to record each item in a file one at a time. What the filing system does, remember, is to gather the data together in a buffer in memory. This memory is placed just above `HIMEM`, and it will be written to the disc only under one of two possible circumstances. One is that the buffer is full, so that there is a complete buffer-load to write. The other is that there is a CLOSE type of statement in the program. For a large amount of data, the disc will spin and write data each time the buffer is full. The CLOSE statement then writes the last piece of data, the one which doesn't fill the buffer. It also writes a special code number, called the end-of-file marker (EOF). This can be used when the file is read, as we'll see later. If you forget the CLOSE statement, you'll leave the buffer unwritten, with no EOF – and cause a lot of problems both in your programs and possibly with that disc. It's possible to use something like `CLOSE 1`, but since CLOSE deals with *all* open files, this is easier and more certain. The biggest danger comes when you are testing a program. If there is an error, such as a syntax error, which stops the program from running, there will be no CLOSE carried out, and the files will be left open. If you had typed a lot of data, you would lose it if you then went on to correct the program and run it again. The correct procedure is to close all of the open files. In this example, it's easy – you only have to type `CLOSE` and press RETURN. For a large program, you would probably find it better to write an ON ERROR GOTO line which, when an error occurred, closed files and ended. This

would automatically ensure that files are never left open. The CLOSE ensures that your data on all file reference numbers will be recorded. When you use an INPUT# statement, as in this example, to gather up the data, you can find that with a lot of data you will hear the disc start and stop at intervals. That's an indication of the buffer transferring data to the disc. You can't use the keyboard while the transfer is taking place, but the time that's needed to write a sector is fairly short. In this example, there is nothing like enough data to fill a buffer. You will hear the disc spin when the OPEN command is executed, and again when the CLOSE command is executed, but not at any time between these two unless you enter a huge amount of data.

Getting your own back

Having created a file on disc, we need to prove that it has actually happened by reading the file back. You can read this file with the CP/M `TYPE` or the BASIC `DISPLAY"filename"` commands, though not in a very neat form. You can also read *and edit* the file with the CP/M utility ED, but you need to be fairly experienced with the use of ED to do this. You can rename the file, or delete it with ERA or KILL. In general, you will use a BASIC program to read and make use of this file. A program which reads a file must contain, early on, a command which opens the file for reading. This is `OPEN"I",1"filename"`, and it must use the same filename as was used to write the file. If we recorded a file using the name `aircraft.dat`, then we must not expect to be able to read it if we use `cameras` – or any other name. Mis-spelling can haunt you here, because if you specify a filename such as `aircraft` or `AIRCARFT.DAT` you get no response! Once the file has been opened, we can read data with `INPUT#2` (or `LINE INPUT#2`), just to establish that we don't have to use the same name as we had when we recorded the file. This then will be followed by the variable name that we want to assign to each item. This statement reads an item from the disc, and will allocate it to a variable name, for printing the item or other use, according to what we have programmed. The number of reads can be controlled by a FOR..NEXT loop, if the number is known, or it can make use of the EOF marker, if the number is unknown. By testing for `NOT EOF(2)`, meaning not the end of file for file number 2, then, we can make the program stop reading the file at the correct place. The example of Figure 9.4 shows both methods in use. The number of fields has been five, so that a FOR..NEXT loop can be used to control the input of the fields. The number of records, however, has not been fixed by a FOR..NEXT loop, so that we have to keep reading the file until the EOF byte is found. This is done in line 30 by testing EOF in a WHILE..WEND loop. If EOF is found, then the file is closed, and the program ends. You can, incidentally, find the size of an open file by using `LOF(channel number)`, but this does not give a size in bytes, and is useful mainly because its value is zero for an empty file. As you can see, the EOF test has been put into the WHILE..WEND loop, because the EOF needs to be tested for *before* another item is read. If you read again from a file like this, you will get the `EOF Found` error message, and the program will stop. Unless you have arranged for an ON ERROR GOTO line to close files for you, the files will still be open. Leaving a

```
Name is Air Metal AM-C 111

      Country of origin  W. Germany
      Type  STOL transport
      Power  2 1120 shp turboprop
      Empty weight  8000 lbs.
      Accommodation  2 crew 24 passengers


Name is Beechcraft Super King Air 200

      Country of origin  USA
      Type  Exec. transport
      Power  2 P&W 850 shp turboprop
      Empty weight  7650 lbs.
      Accommodation  2 crew 6 passengers


Name is GAF Nomad

      Country of origin  Australia
      Type  STOL utility
      Power  2 Allison 400 eshp turboprops
      Empty weight  4330 lbs.
      Accommodation  1/2 crew 15 passengers max
```

Figure 9.5 An example of the readout from the AIRCRAFT.DAT file, made by using LPRINT statements in place of PRINT.

reading file open is not quite such a disaster as leaving a writing file open, but it's still very undesirable. Note that the disc does *not* spin each time you press a key to get another record. This is because a complete chunk of data is read each time, and if the information that you want is all in one buffer load, the disc need not be used. Sorry if I seem to be labouring this point, but a newcomer to disk filing sometimes finds it difficult to remember. Figure 9.5 shows the information from my test file printed by altering some of the PRINT lines into LPRINT, and altering some TAB numbers. You can see that we have the essence of a useful filing system here.

Now this simple example shows a lot about serial filing that you need to know. We have used `PRINT#1` mainly because there was only ever one item at a time to put on to the file. If you have a whole set of items to put at one time, it's better to use `WRITE#1`, because you don't get any problems with formatting, such as you get with `PRINT#1`. For example, if you use `PRINT#1,A$,B$` you will set 15 spaces for each string because of the use of the commas, which is exactly as it would be when you use `PRINT A$,B$`. The name that is used with OPEN (`I` or `O`) is the file name for the file on the disc. Any other file that is later recorded with the same name will not overwrite this file, because the old file changes to a `.BAK` file. The system

```
10 X%=0:CLS$=CHR$(27)+"E"+CHR$(27)+"H"
20 OPEN"I",1,"aircraft.dat":OPEN"O",2,"aircra
ft.dat"
30 PRINT CLS$:PRINT:PRINT TAB(10)"Please wait
..."
40 WHILE NOT EOF(1)
50 INPUT #1,Name$:PRINT #2,Name$
60 FOR N%=1 TO 5
70 INPUT #1,Gen$:PRINT #2,Gen$
80 NEXT:WEND:CLOSE 1
90 PRINT CLS$:PRINT:PRINT TAB(40)"ADDITIONS":
PRINT:PRINT
100 WHILE UPPER$(Name$)<>"X":INPUT"Aircraft n
ame- ";Name$
110 IF UPPER$(Name$)<>"X" THEN GOSUB 160
120 WEND
130 PRINT"You have added"X%" item(s) to the f
ile.":PRINT
140 CLOSE
150 PRINT"END":END
160 X%=X%+1:PRINT #2,Name$
170 FOR N%=1 TO 5
180 PRINT"Field item"N%" - ";:INPUT Gen$
190 PRINT #2,Gen$
200 NEXT:RETURN
```

Figure 9.6 How a serial file can be extended. This means reading in each item and re-filing it, then closing the old file and adding items to the new one.

therefore provides for easy file replacement, *and* for reasonably good file security. In addition, a file is closed by writing the EOF character, something that is done automatically by the system when you use CLOSE (or RESET). How, then, can you update a file, particularly if you want to add more items to the end of the file?

Updating the file

There are two answers, if we stick to serial filing. One possibility, which is the simplest one for short files, is to load the whole file into the memory of the computer, make the alterations (your BASIC program will have to be written so as to provide for this), and then write the file again, wiping out the earlier version. The other possibility is to open two files, one for reading and the other for writing. You don't need to have dual disc drives for this, though it makes life much simpler if you do, using a disc drive letter (A or B) at the start of the filename. Working with two files open means that the computer will maintain two buffers. You read one record from the reading file and you can, if you want, display it. If it's all right, it's then written (to the buffer initially). If the record has to be modified, you can do so. If extra records have to be added, this is equally simple. Each time a buffer empties, the disc will spin and a read or write will take place. This 'simultaneous'

operation is possible because of the use of different OPEN commands, which control different buffers. In practice, it's a matter of writing your program to suit. Figure 9.6 shows a simple program which allows you to extend the file that was created by the program of Figure 9.3. Note, however, that the files use *the same* names, even though I have assumed that both files will be on the same disc. This is because the `OPEN"O"` file and the `OPEN"I"` file are treated separately, using different buffers because of their different reference numbers. This saves any problems of deleting the old file and changing the name of the newly created file, which you would have to do otherwise. This is a particularly useful feature of the Mallard BASIC serial filing system, which you don't find on many other serial disc filing systems. The operating system will see to it that the new file is recorded as `aircraft.dat`, and the old file is renamed `aircraft.bak`. One point we have to be *very* careful about, however, is closing files. The CLOSE 1 statement is used whenever the program has finished with reading the old file, and the CLOSE statement is used whenever the last of the new files has been added. If you made the first statement a CLOSE, then the writing file would also be closed, and no extension of the file would then be possible.

Looking at the program in detail, line 20 opens two files *with the same name*. One, however, is an input file, and the other is an output file. The input file will be used by INPUT#1, and the output file by PRINT#2, so there should be no conflict between them, since they use separate buffers. Line 30 clears the screen and issues a `PLEASE WAIT` notice while the reading and writing of the existing file is done. If your files are long, it may take the disc some time to do all of this reading and writing, and this notice is a reminder that it's all happening. Never leave a user with a blank screen, even if the user is always yourself! When you have tackled chapter 11, you might want to put this and any other notices into a 'window'. In the lines 30 to 80, data will be read in from the old file and written out to the new one until the EOF marker is found. When this happens, the WEND in line 80 takes effect, and the next command is CLOSE 1, which shuts down the reading file. The writing file is still open, however, with its buffer containing data that has been read so far. You can now add more data, using the same lines as you used in the program of Figure 9.3. When an `X` or `x` is typed in response to the request for a record name, then the program displays the number of the added items, closes the write file (so recording the file), and stops. Quite easy, really, but in this program, no provision has been made for altering any of the records that are read from the old file. This is a routine which we can easily add – and that's the next thing to look at.

Changing a record

It's not difficult to alter a record on a file. You read the item, print it, and then change the item before re-recording the file. The main problem is to find a neat way of doing this. The program of Figure 9.7 shows one approach which I use in my own file programs. This is to read the whole of one record, display it on the screen, and give the user the chance to edit or leave as need be. The editing is visual, rather than by the old-fashioned method of numbering the entries and asking for a number to be entered. When the record is displayed, a flashing arrowhead points at the first field. If you want to leave the record as it is, then you press the COPY key, and this will bring up the next record. If you *do* want to change a record, you

```
10 X%=0:CLS$=CHR$(27)+"E"+CHR$(27)+"H":DIM Ge
n$(5)
20 DEF FNlocate$(x%,y%)=CHR$(27)+"Y"+CHR$(32+
y%)+CHR$(32+x%)
30 OPEN"I",1,"aircraft.dat":OPEN"O",2,"aircra
ft.dat"
40 PRINT CLS$:PRINT FNlocate$(39,12);:PRINT"P
LEASE WAIT"
50 WHILE NOT EOF(1):INPUT #1,Gen$(0)
60 FOR N%=1 TO 5:INPUT #1,Gen$(N%):NEXT
70 GOSUB 110
80 FOR N%=0 TO 5:PRINT #2,Gen$(N%):NEXT
90 WEND:PRINT CLS$:PRINT FNlocate$(39,12);:PR
INT"PLEASE WAIT":CLOSE
100 PRINT"END":END
110 PRINT CLS$
120 PRINT FNlocate$(5,22)"Press arrow key to
move cursor,"
130 PRINT"SPACE to alter item, COPY to end ed
it.";
140 PX%=5:PY%=4
150 FOR N%=0 TO 5
160 PRINT FNlocate$(PX%,PY%+N%);
170 PRINT Gen$(N%):NEXT
180 PX%=1:PY%=4
190 WHILE K$<>CHR$(23):PRINT FNlocate$(PX%,PY
%);:REM copy key
195 K$=INKEY$
200 PRINT">";:GOSUB 320
210 PRINT FNlocate$(PX%,PY%);
220 PRINT" ";:GOSUB 320
230 IF K$=" "THEN GOSUB 290
240 IF K$=CHR$(31)THEN PY%=PY%-1
250 IF K$=CHR$(30)THEN PY%=PY%+1
260 IF PY%<4 THEN PY%=9
270 IF PY%>9 THEN PY%=4
280 WEND:K$="":RETURN
290 PX%=5:PRINT FNlocate$(PX%,PY%);:PRINT SPC
(34);
300 PRINT FNlocate$(PX%,PY%);:INPUT Gen$(PY%-
4)
310 PX%=1:RETURN
320 FOR j=1 TO 50:NEXT:RETURN
```

Figure 9.7 Altering items in a file. This program also illustrates a form of visual menu system.

move the arrowhead to the record, using the cursor-up and cursor-down keys, the arrowed keys on the right-hand side of the keyboard. When the arrowhead points to the field that you want to change, you press the spacebar. This wipes out the field-name on the screen, but not in the memory. You can now type a new field name or number, and terminate it with the RETURN key. Only when you have pressed the RETURN key is this new field entered, and if you want to change your mind, you can delete your entry and type the old entry again. When a change has been made in this way, the arrowhead still points to the same field, and you can make another change to this or, by shifting the arrowhead, to any other field. When you have finished editing the record, you can press the COPY key to bring up the next record. The process will continue for as long as there are records to read. The amended file will be recorded as `aircraft.dat`, and the old file will be renamed to `aircraft.bak`. Note, however, that the visual editing system is useful only if the fields are short – a field which spreads over more than one line will cause problems.

How it works

Line 10 sets up in the usual way, and line 20 defines a function `locate$` which will place the cursor at a specified x,y position on the screen. Lines 30 to 90 follow the pattern which should be familiar to you by now. The files are opened, one for reading and the other for writing, and the WHILE NOT EOF(1) loop will load in each record until the end of the file. The record name is assigned to `Gen$(Ø)`, however, so as to make it easier to work with the fields in one array. The new items start with the GOSUB 110 in line 70. This carries out the editing, and when editing of a record is complete, this subroutine will return. The amended or unamended record is then put into the new file by line 80, and the WEND in line 90 brings up the next record.

In the subroutine, the screen is cleared, and a message about the editing commands is printed at the bottom of the screen. Lines 150 to 170 then print the record name and each field on to the screen. Note that this works only if each field is of less than 75 characters, because the whole method depends on using one line for each entry. In line 180, new X and Y positions for the cursor are assigned to PX% and PY%, and a loop starts in line 190. In the loop, the > character is printed at the cursor position, held for a short time, then removed. Four keys are then tested by using the value of K$ which has been obtained from the INKEY$ in line 195. CHR$(23) tests for the COPY key, and causes a return from the subroutine if this key is pressed. K$=" " tests the SPACEBAR, and if this is pressed, then GOSUB 320 calls up the replacement routine. The other two tests are for the cursor movement keys, and if one of these keys is pressed, then the value of PY% is altered. Lines 260 and 270 then test the value of PY%, to ensure that it does not stray outside the line limits, and line 280 completes the loop.

When the SPACEBAR is pressed, the first action in line 290 is to change the PX% number so as to locate the first character of the entry, and the SPC(34) clears most of the line. You could make this space larger if your entries are longer than this. The next `PRINT FNlocate$` instruction places the cursor back at the start of the field-name, and the INPUT then allows you to make the change. By using `en$(PY%-4)` you assign the new entry to

its correct place in the array. Line 310 then restores the value of PX% to place the arrowhead correctly, and the routine returns. The effect is quite impressive, though the key actions in the loop are slightly 'sticky' because of the time delays. In any case, this and the previous routine demonstrate how serial filing can be used with advantage on a disc system. In many cases, you will find this type of filing to be more useful than random-access filing, which is never easy with any disc system. If you need random-access filing, however, chapter 10 indicates how the Mallard BASIC comes to your assistance with a whole programming system, JETSAM, aimed at making random-access filing easier to program.

Chapter 10

Random access filing

Random access filing allows us to get at any record (or set of records) on the disc *without* having to read all of the earlier records as well. 'Random' doesn't mean that we have to create a random access file by scattering bytes all over the disc (though we could), simply that it should be possible directly to locate and read any byte on the disc. In general, we will create a file initially, and record it, using much the same general techniques as we have used for serial filing. We can then add to the file, replace items, change items, and select items *at random*, using random-access techniques. The important point about such random-access filing is that it's possible to extend a file so that the file length is much greater than could be held in the memory of the computer. As an alternative, of course, we can use the random-access filing methods that we need for extending a file in order to create the file in the first place. With a random-access file, we can extract single items or groups as we please, amend or delete or replace items, all without the need to read the whole file into the memory and then back to disc as we did with the serial files. This freedom, however, is bought at a price.

Random access commands

The important random access commands of Mallard BASIC are OPEN, FIELD, PUT, GET and a set of field-positioning and number-conversion instructions, of which more later. Of these, the one that needs most explanation is FIELD, since this has no counterpart in serial filing. Random access filing is possible only if each record in the file consists of the same number of fields, and each field in the record is of a specified size. There's nothing magical about random access filing. The name of the file is stored on the disc directory, and the directory also contains the starting position of the file. From then on, it's easy. If you know that each record is 100 characters (bytes) long, for example, then it's easy to find the 4th

record (numbering from 1) because it will start at the 300th byte in the file, and the 17th record will start at byte 1600. Similarly if the 100 byte record consists of four names each of 25 characters, then it's easy to find the third name in a record by getting to byte 75. Without these fixed lengths, random access filing is not possible, not in this comparatively simple form anyhow. The FIELD instruction exists to allow you to specify each field, its name in the file, and its length as a number of characters/bytes.

The form of a FIELD statement is simple. It starts with `FIELD 1` or `FIELD file%`, for example, using the file channel (or reference) number. As usual, unless you have some rooted objection to the number, you might as well use 1. Using a variable such as `file%`, which can be assigned the number 1 at some earlier stage, allows you some freedom when you are designing a program, in case you want to change the channel number (to avoid conflict with another file, perhaps) without having to change all the statements that use it. Having used the FIELD statement and the channel number, you then put in each variable name that you want to use *in the file* and its maximum size. The syntax here is:

```
size AS name
```

and each name *must* be a string name. You can, for example have such items as `25 AS A$, 32 AS Add$` and so on. There must be one of these `number AS string` statements for each field that you will use, and they are separated from each other and from the start of the FIELD statement by commas.

The OPEN statement is more straightforward, because much of it is as you would use it for a serial file. The keyword OPEN is followed by `"R"` for a random-access file, rather than the `"I"` or `"O"` that would be used in a serial file. Following the `"R"` is a comma, and then the file channel number. Another comma separates this from the file name – and that's often all that you need. Using this form, for example, `OPEN"R",1,"ranfil.dat"` will open the file either for reading or writing using Channel 1 and with the filename of `ranfil.dat`. Any existing file called `ranfil.dat` will be opened by this statement and if no such file exists, one will be created. No `BAK` files are created when you use random-access filing, so there's nothing to stop you writing new data all over a valuable file if your program is badly designed.

So far, though, nothing has been used to determine how large a record is. The FIELD statement decides the size of each field, and you might imagine that the operating system would simply add up the sizes in the FIELD statement and make this sum equal to the record length. It doesn't. The record length is *fixed* at 128 characters, the standard buffer size, unless you specify something different in the OPEN statement. For quite a surprising number of applications, 128 characters is a convenient record size, but if you are filing something small, perhaps only 10 characters per field, it would be a waste of disc space to mark out a set of 128-character spaces each holding 10 characters. The OPEN statement therefore allows you to reserve more or less space by adding a number following the file-name that specifies record size. For example, using `OPEN "R", 1, "ran-fil", 1Ø` would specify that each record in `ranfil` would be of 10 characters only.

Padding

Since each field in a random-access file must be of a fixed length, what do we do if the information does not fit? To start with, we ensure that any numeric information *will* fit, using methods that we can look at later. The awkward items are strings like names and addresses. These will be chopped if they are too long and padded out with blanks if they are too short. Unlike some versions of BASIC, Mallard does not require you to write a lot of tedious loops in order to pad out short names. Instead, you can use three statements that will have a very familiar ring to anyone (like me) who used the good old TRS-80 disc system, or who have more recent acquaintance with BASIC-A on the IBM PC. These statements are LSET, RSET and MID$. The names indicate the action, but the syntax will be unfamiliar unless you have used a similar filing system. LSET, for example, sets a string to the left of its field, fills out the right-hand side with blanks, and assigns the field to the record. For example, using `LSET A$=name$` will assign `A$` so that it contains `name$` followed by enough blanks to make up the correct field size, and put this field in place. This type of statement must follow after a FIELD statement, so that the string A$ (in this example) has been defined already. If the FIELD statement has used `25 AS A$`, then the name in `name$` will be padded out to 25 characters. As you might expect, if `name$` consists of more than 25 characters, it will be chopped to 25 characters by omitting characters at the right-hand side.

The word RSET works in a similar way, but will set the string over to the right, putting the padding blanks on the left. This is most likely to be used when you have converted numbers to strings and want the numbers placed to the right-hand side of a field. MID$ is rather more complicated, and is used mainly when you want to pad with characters that are not spaces. Used like this, MID$ is always preceded by RSET so that the name, or whatever you want to pad, is at the right-hand side of the field. Suppose, for example, that we used a field of 10 characters, and then assigned `RSET A$="IAN"`, followed by `MID$(A$,1,7)="##########"`. This would pick characters 1 to 7 from the string of hashmarks, and place them into the same positions in A$, giving `#######IAN` for A$. You would normally want to use this statement in the form `MID$(A$,1,1Ø-LEN(X$))`, with X$ being the string that is later RSET as A$. You can't use `LEN(A$)` because the length of A$ is 10 characters, fixed by the FIELD statement.

Numbers are treated in a rather special way. Since all items are stored in a random-access file in string form, the obvious thing to do is to convert each number into a string by using STR$. This, however, is a slow action, and if you have to work with large files and a lot of numbers it can noticeably increase the time that you spend just waiting. In addition, the conversions that are made by STR$ are to quite long strings, particularly for double-precision numbers, and you have to make your fields long enough to hold all the characters. Mallard BASIC provides three sets of reserved words to cope with this conversion in a different and more useful way – again familiar to former TRS-80 (and present IBM) programmers. The principle is to convert into string form not the number itself, but the code that is used to store the number. The result of this will be strings that are often of very weird characters, if you try to print the result, certainly not recognisable as numbers. The advantage is that an integer always codes as a 2-character

```
10 A%=65:B%=16706
20 X=22.45
30 d#=216.45169#
40 A$=MKI$(A%):PRINT A$;"   ("LEN(A$)")"
50 B$=MKI$(B%):PRINT B$;"   ("LEN(B$)")"
60 X$=MKS$(X):PRINT X$;"   ("LEN(X$)")"
70 D$=MKD$(D#):PRINT D$;"  ("LEN(D$)")"
80 PRINT CVI(A$);"   ";CVI(B$)
90 PRINT CVS(X$);"   ";CVD(D$)
```

Figure 10.1 Converting numbers to string form for random access filing. This conversion is not the same as is achieved using STR$.

string, a single-precision number as a 4-character string, and a double-precision number as an 8-character string. Figure 10.1 illustrates this with a short program that uses the codings, prints the characters and the lengths of the strings, and shows how we can convert back into printable numbers.

The number-to-string conversions are done by MKI$ *(integers)*, MKS$ *(single precision)* and MKD$ *(double precision)*, using fixed string lengths of 2, 4 and 8 bytes respectively. For an integer, you can see reasonably easily how the scheme works. The number 65 is character A in ASCII code, so when the string version of 65 is printed, you get an A printed. The next number, 16706, is more of a surprise, because it gives BA. The ASCII code for B is 66, with A being 65, and 16706 is $66 + 256 \times 65$. This corresponds with the way that larger integers are stored as two bytes. The strings that correspond to single and double-precision numbers are considerably more complicated unless you have a taste for mathematics. The program shows that the length allocated to each string has no connection with the number of digits in the number, and is fixed only by the type of number, integer, single-precision or double-precision. Finally, the conversion back to number form is carried out by using CVI (integer), CVS (single-precision) and CVD (double-precision) respectively. The use of these words allows us to work with numbers in random-access files using smaller fields than would be possible if we had to convert with STR$ and VAL, and also with faster conversions.

An integer file

It's time to illustrate a random access file in action. It's tempting to illustrate this with a really elaborate filing program, but for several reasons, I haven't done so. The main reason is that random access filing, as we shall see, has serious limitations in this form, and Mallard BASIC provides a variation on random access filing, JETSAM, that is much superior. The real effort, then will be reserved for JETSAM, and we'll content ourselves with a simpler illustration, an integer file, of random access work. The fact that this is a simple illustration should not put you off – the techniques are exactly the same as you will need if you choose to use random access filing for more serious purposes.

```
10 ON ERROR GOTO 290:REM close files
20 bel$=CHR$(7):cls$=CHR$(27)+"E"+CHR$(27)+"H"
30 file%=1:limit%=100
40 OPEN"R",file%,"integers.rnd",2
50 FIELD file%,2 AS A$
60 FOR N%=1 TO limit%
70 A%=1000-N%:REM recognisable number!
80 LSET A$=MKI$(A%)
90 PUT file%
100 NEXT
110 CLOSE
120 GOSUB 260
130 OPEN"R",file%,"integers.rnd",2
140 FIELD file%,2 AS N$
150 n%=1
160 WHILE n%<>0
170 INPUT "number ";n%
180 GET file%,n%
190 X%=CVI(N$)
200 PRINT"Number is ";X%
210 PRINT bel$
220 FOR j=1 TO 1500:NEXT
230 PRINT cls$
240 WEND
250 GOTO 290
260 PRINT"Press any key to replay"
270 WHILE INKEY$="":WEND
280 RETURN
290 CLOSE:END
```

Figure 10.2 A simple random access filing program for integers, demonstrating the techniques.

The example is in Figure 10.2. the first line is an automatic way of ensuring that files will be closed if an error is found. We haven't dealt with the ON ERROR GOTO statement yet, but in this example its actions are clear enough. If an error is found as the program runs then instead of the program stopping with the usual error message, it goes to line 290, which will close files and end. The snag here is that you get no error message in this simple application, and it's a good idea to omit this line while you are testing the program. Line 20 then contains assignments to two string variables. The variable `bel$` contains CHR$(7), which when placed after a PRINT command will cause a beep to be sounded. The `cls$` is the screen-clearing string that ought to be familiar by this time. Line 30 then defines two variables, `file%` assigned the value 1 and acting as a channel number, and `limit%` to set a limit to a FOR..NEXT loop of items. The point of using variables for these quantities is to illustrate how easy it is to adapt a program to other numbers when variables are used in this way. In this short example, of course, you could just as easily have used numbers 1 and 100 in the appropriate places.

1. OPEN file, using channel number and file name. Record length added if not 128 characters.
2. FIELD to show size of each field that will be used.
3. LSET places data in field, set to left and padded out with spaces if needed.
4. PUT places the record in the file – the disc will spin only if the buffer is full.

Figure 10.3 A summary of the main processes in creating a random access file.

Line 40 then starts the creation of the file, using the OPEN statement. The channel is as fixed by `file%`, equal to 1, and the name chosen for the file is `integers.rnd`. This suggests that the integers are random, but in this examble they quite certainly are not for reasons that we'll look at in a moment. The last part of the OPEN statement fixes the record size at 2, since only 2 bytes are needed to store an integer. Following the opening of the file, which will start the disc spinning, the FIELD statement uses the channel number again, and states that the variable that will be filed is called `A$` and consists of 2 bytes only.

Having declared the sizes of record and of field, identical in this case since there is just one field per record, we can now put numbers into the file. The loop that does this starts in line 60, and uses the ending number set by `limit%`. In the loop, the numbers that are to be filed are assigned to A% in line 70, using 1000–N%. The reason for doing this in a test is to provide easy recognition. If you use truly random numbers, you can't be sure when you replay the file if a given number is really the one that was originally filed in that position. By using a scheme like this, you can be sure that number 1 is 999, number 2 is 998 and so on, all easily recognised later. This is a useful tip for testing any filing system, because otherwise it's easy to miss a fault that makes the replayed item incorrect. Line 80 is the important one at this point. This line places the string version of the number A% into a string A$, and sets to the left. Since the string consists of only two characters, this looks rather un-necessary *but it is essential.* If you omit the LSET, you will find that `A$=MKI$(A%)` will run, but that you cannot find any numbers in the file when you replay it. The reason is that LSET, RSET and MID$ are not just field padding statements, they are the statements that actually place the value of a variable *into the file.* You can't omit them! In normal circumstances, there would be several lines of LSET, RSET or MID$ statements to place the fields into the record in correct form, and following these you need to put the record on to the file. This is done in the following line here, using `PUT file%`. If the record is of the default size of 128 bytes, this statement will place the data on to the disc, but if the records are small, as they are in this example, the records are stored in a 'buffer' piece of memory until the total length is around 128 bytes, upon which the assembled records are filed. Figure 10.3 shows a summary of these processes. At the end of the loop, the file is closed by using the CLOSE statement. Once again, `CLOSE file%` could be used, but CLOSE by itself closes all files.

The second part of the example shows how such a file can be read back. One golden rule here is that unless you are trying to do something very clever (or risky), you should try to use statements in the same form when replaying as you did when recording. Accordingly, the replay program starts with `OPEN "R", file%, "integers.rnd",2` just as the recording program used. Note that the OPEN statement is identical, no matter whether we are wrtiting, reading or extending the file. We can once again use FIELD to declare the form of the fields, and this time we have used `2 AS N$`, with a different variable name. The variable name of A$ need not be used this time – all that is going to be done is to take two characters from the file and assign them to a string variable, and provided that you declare *some* variable, it doesn't matter what the name is so long as it does not conflict with anything else that you use. With the file open and the FIELD statement executed, you can then use the file. In this case, we are going to read the file, and to do so we have to specify which record we want to read. This is done by record number, and this is the weakness of random access files of this form. Unless you know the record numbers, you can't read the records. If the records that you are using will have some 'natural' number, like the employee record number that is illustrated in the manual, all is well. If not, then you have to find some way of associating each record with its *accession* number, as this is called. In this simple example, of course, you simply type a number between 1 and 100, and the number appears. Since you know that the number that appears should be 1000-record number, you can check that the numbers are the correct ones. Asking for a number greater than 100 *is not an error*, you merely get the result of zero.

Now before we leave this example, it's important to point out that random access allows much more than is illustrated here. Once the file is opened and the FIELD statement executed, we can amend or extend the file as we like. This is done by specifying a record number with PUT in the same way as we have used GET in line 180. In other words, if you assign something to N$ in this second part of the program, you can use a statement like `PUT file%,45` to change record number 45 to some other number. You can also use a statement like `PUT file%,12Ø` to create a record that did not exist in the original. Neither the OPEN nor the FIELD statement puts any limit on the number of files that you can create, and there's no restriction about the order in which you create them. The system that is followed is that if PUT is used with no record number, then the records are put into the file in order starting with 1, or with any specified number that is used in a preceding PUT statement. This freedom can cause difficulties, however, because it's quite possible to specify a record number that cannot be filed because there is no more room on the disc. Because of this, random access files have to be planned carefully, usually by setting a limit to the number of entries and making a dummy file with this number in order to clear space on the disc. The illustration of random access filing in the manual shows most of the techniques that you will need. Meantime, we'll move on to another topic, JETSAM.

JETSAM files

Serial files are very useful for a lot of purposes, but not if you want to be able to obtain one record out of a very large number. With a long program in the memory of a PCW machine as well as BASIC itself, there would not be much memory left to store a great number of records at a time. This means that you would have to read your records in as an array, test each one to find if it was the one you wanted, and then read another lot in if you couldn't find the one that you wanted. The obvious alternative is random access filing, and we have just dealt with that topic. Random access filing, however, in Mallard BASIC, suffers from one considerable snag. Unless you know the number of a record, you cannot get random access to that record. You can't, for example, get the record for `SINCLAIR, I` just by typing this name, and unless you happen to know that this is record number 273, that's it, you will have to search through the records one by one just as you would with a serial file. Programming with random access filing so that you can recover items by name, or so that you can quickly list in alphabetical order of name, is by no means easy – but these are often precisely the actions for which we most need random access!

The solution to the problem was, however, developed a long time ago, and is called by various names like indexed random access, relative filing, ISAM (Indexed Sequential Access to Memory) and so on. The JETSAM name is derived from ISAM (JET because of speed, perhaps) and it has all of the advantages of random filing with few of the disadvantages. We'll spend the rest of this chapter, then, looking at JETSAM files and how to use them. Rather then use examples alone, I'll show the development of a longer program (compared to our examples) that makes use of JETSAM files to achieve a useful random access database. In other words, this will be a program that allows you to file facts on the disc and get them back in useful form whenever you want them.

JETSAM File facts

A JETSAM file has to be made up from a number of records, like any other file. In each record, as usual, each field must have a fixed maximum length. The record length, however, must be set to a number that is *two bytes* greater than the total of field lengths. This is vitally important, because these two extra bytes are used to store an index number by which the record will be found. Another vitally important point that is easy to miss if you come to JETSAM after using other types of files, is that the CLOSE statement *must* use the channel number of the JETSAM file. You *must not* rely on the simple CLOSE statement that suffices with other files.

The most important feature of JETSAM filing is that the disk operating system keeps a track of the random access file for you. This is done by setting up an additional 'key file', which is a simple serial file that stores 'keys' to your records. You have considerable freedom in choosing these keys. You might, for example, specify that each key should be the first five letters of the surname that is the first field of each record. By typing these

letters, then, you can locate the key, and the key file will then find the item in the random access file. This could be done in other ways, but only with a lot of hard programming work. You can have more than one such key in the file, so that you can get at your records by more than one way, and you can, if you like, have one entry in a key file referring to more than one record. Each key file is kept in sorted order. If the keys are numbers, then the numbers are kept in low-to-high order, and if the keys are short names, they will be kept in alphabetical order. Each time you add a record and insert its key(s), then the new key(s) will be placed into the correct position in the key file(s).

JETSAM rules

On any computer, there is always a price to be paid for convenience, and in this case, the price is that you have to construct your filing program to a set of rules. We have seen that you need to fix the number of fields per record and the number of characters per field. You have to use a special form of OPEN command for the JETSAM file, and you must ensure that the record length is two bytes more than the sum of field lengths. You also have to CREATE a new file when you first start a file on a disc side, not simply open it. There are also a large number of new statements for working with the JETSAM file, and all of these are functions. In other words, when you use one of these statements, it will return a number just as LEN(A$) returns a number. This number can be used in checking for errors and in guiding the program to more efficient use of the file. At this point, simply listing the features of a JETSAM file becomes boring and confusing, and from now onwards, we'll look at this form of filing by considering the creation of a program using it. The program is one that indexes your collection of books (after all, they might not all be by me), and is one of the few database uses that does not appear to be covered by the Data Protection Act of 1984. In this program, because it can be used as the basis of a useful database, several types of statements will appear that will be new to you. They will be explained in more detail later, because in this chapter, I want to concentrate on the JETSAM filing system, rather than on the methods that are used to present the data.

The Library file

Figure 10.4 shows the short core program for the Library program. Line 10 is a way of dealing with one particular type of error that can arise, and we'll look at that later when we come to it. The rest of the core follows lines that should be fairly familiar, with the main subroutines annotated with REMs so that we can follow what's happening. Line 60 contains a choice from five menu items. Line 70 contains two GOSUB statements. The GOSUB 2060 closes all files and the GOSUB 2020 restores the normal full size of screen. As usual, `PRINT cls$` is used to clear the screen. This line is put in to

```
10 ON ERROR GOTO 2080
20 GOSUB 270:REM INITIALISE
30 GOSUB 2140:REM TITLE/INSTRUCTIONS
40 GOSUB 430:REM MENU
50 GOSUB 530:REM CHOICE
60 ON choice% GOSUB 610,710,950,1510,110
70 GOSUB 2060:GOSUB 2020:PRINT cls$
80 PRINT"To return to menu, press spacebar"
90 GOSUB 580
100 IF K$=" " THEN 40
110 GOSUB 2060:REM CLOSE FILES
120 GOSUB 2020:PRINT CHR$(27)+"w":REM cancel
 wordwrap
130 PRINT cls$
140 END
```

Figure 10.4 A core program for a JETSAM filing program.

```
270 cls$=CHR$(27)+"E"+CHR$(27)+"H"
280 tell$="Please press any key to proceed"
290 yn$="Please press Y or N key"
300 DEF FNwindow$(t%,len%,d%,w%)=CHR$(27)+"X"
+CHR$(t%+32)+CH
R$(len%+32)+CHR$(d%+31)+CHR$(w%+32)
310 ques$="Do you want another?"
320 hard$="Press P for hard copy, any other k
ey for screen."
330 scrl$="Use ALT-S to stop/start screen scr
olling."
340 BUFFERS 10
350 flag%=0
360 fil%=1
370 main$="BOOKS.DAT"
380 inx$="INDEX.DAT"
390 user$="Library Index"
400 ln%=92:REM record length + 2
410 PRINT CHR$(27)+"v":REM Word-wrap on
420 RETURN
```

Figure 10.5 The initialisation routine. Note that line 300 is very long and has been split here for convenience of printing.

ensure that after each subroutine that uses files has been carried out, all files will be closed, and the full screen will be available. There is then a request to press the spacebar to return, followed by an INKEY$ action in the form of GOSUB 580. The file-closing and full-screen steps are then repeated to ensure that there is no chance of files being left open or the screen size restricted. These lines are redundant in the program as it stands, but should be kept in case you extend the program to include a second menu following the first one. In line 120, a word-wrap is removed. This is a feature that can be useful for long entries, because it prevents words being split at the edge of the screen. Here again, word-wrap is not really needed in this program, but the feature might be useful for your own work. The word-wrap *on* command is located in a subroutine.

Before we start to look at the file subroutines, we must go over the introductory routines briefly. The first of these starts in line 270, and carries out initialisation, figure 10.5. This assigns various phrases, most of which need no comment. Line 300, however, is new. This is a defined function that creates a *window*. A window is a limited area of screen that the computer will treat as its full screen. If, for example, we define the bottom half of the screen as a window, then we can clear and print on this half leaving the upper half unaffected. Using windows is a useful way of allowing different parts of the screen to be used for different purposes, and the principle has been lavishly used in this program. The definition allows us to to create a window by a statement of the form `PRINT FNwindow$(top,left,depth,width)`. Note that if you omit the `FN` in such a `PRINT` statement, you will get an `Out of Memory` message! Line 340 is BUFFERS 10. This assigns ten chunks of memory for buffers, and is used to speed up file access. If you omit this line, your filing actions will run rather slowly, particularly when you have a lot of data in the file. The more buffers you can assign, the faster the program can run, but you must not, of course, take up so much memory with buffers that there is no room for the program to operate. Lines 350 to 400 assign variable names for quantities that will be used in the program. This gathering together of these quantities makes it easy to alter the program to suit your own needs, with `fil%` used for channel number, `main$` used for the main filename and `index$` for the key files. There is also `user$`, which is a 'user name' that can be recorded on the disc with the file. Its presence is not necessary, and no use is made of it here, but in a longer program it might be used for identification, such as, for example, to identify different sets of data on different discs. Line 410 turns on the word-wrap feature, referred to earlier.

The next subroutine is shown in figure 10.6, and contains the title and instructions, for which we need no comment at the moment, and figure 10.7 is the menu. This provides the items that are shown, including the important `End program` choice. The statement in line 500 then creates a one-line piece of screen that can be used for a message like `Pick by number`, though, once again, it has not been used in this case. The assignment to `mx%` in line 510 ensures that only numbers 1 to 5 will be permitted in the reply – this is the only mug-trapping that has been used for the menu choices in the program. The GOSUB 2020 then opens up the full screen again, and the subroutine returns. The choice of menu item is then made in the subroutine in figure 10.8. This uses an INKEY$ type of choice, with the value tested for no key, value less than 1 or value greater than `mx%`. Putting the conditions in the WHILE loop ensure that the loop runs until a suitable key has been pressed. When a suitable choice has been made, the number value

```
2140 GOSUB 2020
2150 PRINT cls$;:PRINT TAB(41)"LIBINDEX":PRINT
2160 PRINT" This program allows you to make a comprehensive
index of all your books."
2170 PRINT"You can create a file on a disc, and from then on
 fill it with details of"
2180 PRINT"your books, using author's name, title and subjec
t area as filing 'keys'."
2190 PRINT"At any time, you can list your books in alphabeti
cal order of title, name"
2200 PRINT"of author, or subject, and this list can be on pa
per if you like. At present"
2210 PRINT"the printer is set up for pages."
2220 PRINT
2230 PRINT"  You can add to the list as you please, and you
can also look for a book"
2240 PRINT"by specifying its title - or the first six letter
s of the title. If there"
2250 PRINT"is more than one book with similar titles (same f
irst six letters), then"
2260 PRINT"you can move through the list forwards or backwar
ds until you find the one"
2270 PRINT"that you want. This title can then be erased or c
orrected."
2280 PRINT
2290 PRINT"  The lists are kept in alphabetical order, but t
his will be upset if"
2300 PRINT"a title is inserted using the 'correct' facility.
 If you want to put in"
2310 PRINT"a new entry, always use the 'Add book titles' cho
ice. You can delete"
2320 PRINT"any title from the list after finding it in the l
ist.":PRINT
2330 PRINT"Press any key to start"
2340 GOSUB 580
2350 RETURN
```

Figure 10.6 The title and instructions subroutine – this is always written last of all!

```
430 PRINT cls$;TAB(43)"MENU":PRINT
440 PRINT FNwindow$(3,10,6,70);
450 PRINT"1. Create new file/disc."
460 PRINT"2. Add book titles."
470 PRINT"3. List books in order."
480 PRINT"4. Find books, change, delete."
490 PRINT"5. End program."
500 PRINT FNwindow$(22,10,1,70);
510 mx%=5:GOSUB 2020
520 RETURN
```

Figure 10.7 The Menu subroutine.

```
530 K$="":WHILE K$=""OR VAL(K$)<1 OR VAL(K$)>mx%
540 K$=INKEY$:WEND
550 choice%=VAL(K$)
560 GOSUB 2020
570 RETURN
```

Figure 10.8 The choice subroutine, using INKEY$.

```
610 PRINT cls$
620 PRINT FNwindow$(2,10,2,70);"Creating file..."
630 PRINT FNwindow$(23,10,3,70);"Press N to stop"
640 GOSUB 580:IF UPPER$(K$)="N" THEN 680
650 PRINT cls$
660 CREATE fil%,main$,inx$,2,ln%,user$
670 REM creates file on new disc
680 GOSUB 2020:PRINT cls$;
690 CLOSE fil%
700 RETURN
```

Figure 10.9 The CREATE subroutine, needed when a new file is started.

```
580 K$="":WHILE K$=""
590 K$=INKEY$:WEND
600 RETURN
```

Figure 10.10 The INKEY$ subroutine used to get a single-key reply.

is assigned to `choice%`, the full screen is opened again, and the subroutine returns. Once again, at this point the opening of the full screen is redundant, but it has been put in so that the action will be carried out when the subroutine is called from other places. When this subroutine returns, it will implement the menu choices by the calls that are shown in line 60, figure 10.4. These are the filing routines, and we shall now scrutinise them in detail.

Filing with JETSAM

A JETSAM file cannot be started simply by an OPEN statement, and OPEN is used only for files that have already been created. To create a JETSAM file from scratch on a disc, we need a once-and-for-all CREATE statement, and figure 10.9 shows this action. The first four lines are concerned with screen display, and allowing you to have second thoughts by pressing the N key. This uses the short routine in figure 10.10 to detect a key so that the CREATE action is aborted when the N key is pressed. If any other key is pressed, then a new file will be created. This will use the names `BOOKS.DAT` and `INDEX.DAT` that were provided for in the initialising

routines, along with several other variables. In line 660, then, we have immediately following CREATE the channel number `fil%`. This quantity is one that is used extensively, so the assignment to `fil%` avoids the need to keep having to check what number has been used. Following `fil%`, we have the essential descriptive variables for the file, all separated by commas. We have noted `main$` and `index$` before as the filenames that will be used on the disc for the two sets of files. The number `2` is a 'file lock' number, and though file-locking is used only in multi-user versions of Mallard BASIC, the number 2 has to be used here to ensure compatibility. The variable `ln%` contains the number 92 that is the record length, allowing for the two extra bytes, and the optional `user$` at the end identifies the file, but is not used in this example. The routine ends by calling a subroutine (figure 10.11) to restore the full screen, clearing the screen, and then closing the file by using `CLOSE fil%`.

This last point is *very* important. The channel number must follow the CLOSE statement, otherwise files will not be correctly closed, and you will not be able to use what you have on the disc. In addition, it's possible that the CREATE statement in line 660 may not operate. If the disc that you are using has these files on it already, you are not permitted to create new ones, and this will cause an error message. Now in the normal way of events, this error message would also stop the program. The program, stopped at this stage, would have open files, causing subsequent trouble unless you then typed `CLOSE fil%` as a direct command. To avoid this, the error detecting line, line 10, forces the program to go to line 2080 in the event of any error. In this line (see later) the error type is analysed, and if it happens to be the error of trying to create files that already exist, a message is printed, there is a pause, and the program returns to line 670, allowing normal service to be resumed, and files closed. This is an excellent illustration of the usefulness of the ON ERROR GOTO statement, much better than any artificial example could be. If your disc contains no book data files, then the files are created, and if it does, the instruction is ignored. Once the files have been created, we can fill them with data, or make use of the data, and that's what the rest of the Menu is concerned with. The CREATE section is used only once per disc side.

Adding data

Once the files have been created, we have to key in information. Now nothing so far has done more than create file names and allocate record sizes. We have done nothing to establish field sizes, nor to state how many keys will be used and what the keys will be. It's at this stage that we need to look at these points, and of course they would be dealt with at the planning stage of the program, not as late as this (I hope). The adding of data is dealt with in the subroutine illustrated in figure 10.12.

At the planning stage, not illustrated here, the program provided for storing the author's name, surname first, the book title, and the general subject (Travel, Computing, Play, Novel and so on). The field sizes are 20 for author, 50 for title and 20 for subject, giving 90 in all. Each of these fields

```
2020 PRINT FNwindow$(0,0,30,90);:RETURN
```

Figure 10.11 The subroutine that restores the full screen size after the use of windows.

```
710 PRINT cls$
720 PRINT FNwindow$(2,10,2,70);TAB(25)"BOOK DATA"
730 GOSUB 2030
740 PRINT FNwindow$(5,10,6,70);cls$;
750 LINE INPUT"Author- surname first ";A$
760 PRINT TAB(15);:LINE INPUT "Title ";B$
770 PRINT TAB(13);:LINE INPUT "Subject ";C$
780 LSET aut$=A$:LSET ttl$=B$:LSET sub$=C$
790 IF flag%=1 THEN RETURN
800 err%=ADDREC(fil%,0,0,UPPER$(LEFT$(aut$,6)))
810 IF err%<>0 THEN GOSUB 920
820 rn%=FETCHREC(fil%):REM find record number
830 err%=ADDKEY(fil%,0,1,UPPER$(LEFT$(ttl$,6)),rn%)
840 IF err%<>0 THEN GOSUB 920
850 err%=ADDKEY(fil%,0,2,UPPER$(LEFT$(sub$,6)),rn%)
860 IF err%<>0 THEN GOSUB 920
870 PRINT FNwindow$(20,10,3,70);
880 PRINT ques$:PRINT YN$
890 GOSUB 580:REM YES/NO
900 IF UPPER$(K$)="Y" THEN 740
910 RETURN
```

Figure 10.12 The subroutine for inputting data to the file. This is used by two other routines, hence line 790.

```
2030 OPEN "K",fil%,main$,inx$,2,ln%,user$
2040 FIELD fil%,20 AS aut$,50 AS ttl$,20 AS sub$
2050 RETURN
```

Figure 10.13 The subroutine that opens files and establishes field sizes.

will also be used as a key, so that we can pick out a record by using the author's name, the title, or the subject. The key files will use just the first six letters of each field. This means that if we want to pick one book by its title, there is a possibility that more than one title will use the same key – for example, books called *Understanding Electronic Components* and *Understudy for Monty* would use the same index of UNDERS. The JETSAM system gets around this by allowing us to call up, very quickly, the next record or the previous record to check for the one that we want.

At the entry stage, there are several things that must be done. The data must, of course be entered and placed in its fields. This uses the same techniques as in random access filing, with LSET used to put each string into the left of its field. At the same time, the letters are extracted for use in the index file. This involves making the extract and using one line to enter each key. With three keys in this file, we need three such lines to ensure that each record can be accessed in its three different ways.

```
920 PRINT FNwindow$(10,20,2,70);
930 PRINT"Cannot proceed - error code "err%
940 RETURN
```

Figure 10.14 The error report line for JETSAM file errors – this will never run in normal service!

Getting down to the detail of figure 10.12, the screen is organised in the first three lines, and the call to 2030 (figure 10.13) opens the files and establishes the fields. We shall be using `aut$`, of 20 characters, for author's name, `ttl$` of 50 characters for title, and `sub$` of 20 characters for subject. With this subroutine completed, the inputs can start in line 750. A LINE INPUT has been used so that names can be typed surname first, using a comma between surname and initials. The same principle is used for the other strings, with the PRINT TAB statements being used to line up the inputs. Temporary variables `A$`, `B$` and `C$`, are used to hold these inputs. Line 790 is used because you may need to correct an existing entry at times rather then add a new one. This means using up to line 780 and then returning (to avoid the add-to-file steps), and the 'flag' variable `flag%` is used as a signal. This quantity is normally 0, so that line 790 has no effect in this case, and we shall look later at the case where this is used.

With data entered, we now have to put it into the files. Adding a record is carried out by the ADDREC statement, which also places a key entry into the key files. In this case, we will use the author's name as the key, and make this key 0. This number is referred to in the Manual as the 'rank' of the file, so that this will be the rank zero key. In the ADDREC statement of line 800, the word ADDREC is followed in brackets, by the channel number `fil%`, a zero (the lock number), then another zero (the rank of this key), then the key string itself. In this case, the key string is the first six letters of the author's name, `aut$`. This remember, has already been padded to a total length of 20 characters, so that if the author has a very short name, the key string will include a blank or two. You have to be careful here, because if you used `LEFT$(A$,6)` rather than `LEFT$(aut$,6)` you might not get these padding blanks in a very short name, and this would cause trouble later. The whole entry is written in the form of an assignment to `err%`, and this number will be zero if the action is successful. Any fault, unlikely in the Amstrad version of JETSAM, will cause a different number to be allocated to `err%`, and this will be dealt with by line 920, see figure 10.14.

This ADDREC step puts the entire record of 92 bytes into the main random access file, `BOOKS.DAT`, and it puts one entry, the first six letters of the author's name, into the index file, earmarked as rank 0. It does *not*, however, create any other keys. For a lot of simple file actions, one index to each record might be all you needed, and the subroutine would be almost over by this point. In this example, however, I want to show how other keys to the same record can be added at this same time. This requires the use of the record number, the same record number as you use to identify the record in a random access file. Now you don't know this number, so that a routine is needed to find it. This is shown in line 820. It's simple enough, because all that you want is the record number for the record that is being processed, and `FETCHREC(fil%)` is all that is needed. This gives the record

```
950 PRINT cls$;FNwindow$(2,10,2,70);"LIST BOOKS"
960 GOSUB 2030
970 PRINT FNwindow$(5,10,6,70);
980 PRINT"List by -"
990 PRINT TAB(9)"1. Author."
1000 PRINT TAB(9)"2. Title."
1010 PRINT TAB(9)"3. Subject
1020 mx%=3:PRINT FNwindow$(20,10,3,70);
1030 GOSUB 530
1040 GOSUB 2020:PRINT cls$;
1050 ON choice% GOSUB 1090,1230,1370
1060 PRINT"Return to main menu?":PRINT YN$
1070 GOSUB 580:IF UPPER$(K$)="N" THEN PRINT
cls$;:GOTO 970
1080 choice%=0:RETURN
```

Figure 10.15 The listing option subroutine.

rn%, which will be zero if for some reason the record cannot be located. Having obtained the record number in this variable, we can then add the other keys with the ADDKEY statement.

The first of these is in line 830. As usual, it is written as an assignment to err%, and the quantities that follow ADDKEY are enclosed in brackets. These are the channel number fil%, the lock number 0, the key rank of 1, the key itself, and the record number. As before, the key string is taken from the first six characters, in this case of the title. The keys in rank 1, then, are title keys, with the keys in rank 0 being the author keys. As usual, err% is tested so that a message can be delivered if anything goes wrong (any number other than zero). Line 850 then adds the rank 2 key, the first six letters of the subject name. With these keys added, the user is then asked if he/she wants to make another entry. If the answer is Y, this causes a return to book entry so that another record can be created. If the answer is any other key, the program returns. We now have records in the main file, and keys in the index file.

Listing the records

Option 3 of the main menu provides for listing the entries, and since three keys have been used, we can list by author, by title, or by subject, always in alphabetical order. The menu choice leads to the subroutine that starts at line 950, figure 10.15. This is a sub-menu, in which you have to choose to list by one of the three possible methods, so that lines 950 to 1050 follow a familiar pattern. The detailed action is then delegated to the subroutines in lines 1090, 1230 and 1370. Starting with the list by author, figure 10.16, lines 1090 to 1120 deal with the screen and some messages. The user is asked if hard-copy (on paper) is needed, and reminded that if the screen is used, the display can be stopped and started by using ALTS. Line 1130 then

```
1090 PRINT FNwindow$(1,0,2,90);cls$;
1100 PRINT TAB(37)"LIST BY AUTHOR"
1110 PRINT FNwindow$(28,0,2,90);cls$;hard$
1120 PRINT scrl$;
1130 GOSUB 580: IF UPPER$(K$)="P" THEN pr%=1
ELSE pr%=0
1140 PRINT FNwindow$(3,0,24,90);cls$
1150 err%=SEEKRANK(fil%,0,0)
1160 IF err%<>0 THEN GOSUB 920
1170 WHILE err%=0 OR err%=101
1180 GET fil%
1190 GOSUB 150
1200 err%=SEEKNEXT(fil%,0)
1210 WEND
1220 RETURN
```

Figure 10.16 Listing by author. SEEKRANK is used to find the first item, and SEEKNEXT gets the next in order until the end of the file.

```
150 PRINT aut$;" - ";ttl$: IF pr% THEN LPRINT
 aut$;" - ";ttl$
160 PRINT"Subject - ";sub$: IF pr% THEN LPRIN
T "Subject - ";sub$
170 PRINT: IF pr% THEN LPRINT
180 RETURN
```

Figure 10.17 The printing routine for author-order listing. This provides for both screen and paper printout.

```
1230 PRINT FNwindow$(1,0,2,90);cls$;
1240 PRINT TAB(38)"LIST BY TITLE"
1250 PRINT FNwindow$(28,0,2,90);cls$;hard$
1260 PRINT scrl$;
1270 GOSUB 580: IF UPPER$(K$)="P" THEN pr%=1 E
LSE pr%=0
1280 PRINT FNwindow$(3,0,24,90);cls$
1290 err%=SEEKRANK(fil%,0,1)
1300 IF err%<>0 THEN GOSUB 920
1310 WHILE err%=0 OR err%=101
1320 GET fil%
1330 GOSUB 190
1340 err%=SEEKNEXT(fil%,0)
1350 WEND
1360 RETURN
```

Figure 10.18 The title-order list subroutine.

allows the P key to alter a variable pr% which will control the printer. The important line now is line 1150, written as usual as an assignment to err%. The effect of SEEKRANK is to find the first entry in a specified rank, which will be for the first record in the alphabetical order of this rank. The SEEKRANK statement needs the channel number fil%, the lock number of 0, and the rank number which in this case is 0 also. Now in this case, a successful action will give err% equal to 0, and anything else will give a different number. Having located this first key, however, we need to read the record, and then find the next key in the same rank. This can give rise to two possible numbers for err%, 0 or 101, both of which indicate success. The number 0 means that the key is as requested originally, 101 means that this is another in the same rank, also acceptable when we want to list all the items in the same rank. The loop that starts in line 1170 will therefore allow err% to take either value. In this loop, the record is read by using GET fil% (note – no brackets), the result is printed by a call to 150, and the SEEKNEXT statement in line 1200 gets the next record. Since the keys will be read in strict alphabetical order, this loop will get the records in this order, the order of author's name, and print each until there are no more. The end of the key file is marked with a number 103 assigned to err% by the SEEKNEXT statement.

While we're on the subject of reading the list, we can look at the printing routine, figure 10.17. This is straightforward except for the provision for using LPRINT (hard copy) if the variable pr% is 1. If you had chosen to use hard copy, you will see each record on the screen, and it will also be sent to the printer. Using ALTS will suspend the screen output, but has no immediate effect on the printer because of the printer buffer that is used. The printer is normally set up for paged output, so unless you have used a printer-control file to alter this setting, you can get your listing in paged form, though it's better to use continuous stationery and cut to pages later.

The methods that are used to get a listing in alphabetical order of author's name are used in almost identical form to get a listing in order of title (figure 10.18 and 10.19) or in order of subject (figure 10.20 and 10.21). These routines are so similar that it would be possible to deal with all three choices with a single subroutine, but for the sake of clarity, I have kept them separate – it's not as if there is any shortage of memory space. The printouts are different in each case, showing the quantity that is used for listing the records as the first item, but for many applications, the same routine could be used for all three keeping to the order of author, title and subject.

Pick, Change, Delete

The last main item on the main menu allows one title to be picked, its details checked, and then for this item to be changed or deleted if required. This is the most complicated set of actions in a JETSAM file, and the subroutine that controls all of it starts in line 1510, figure 10.22. This starts off in a straightforward way up to the point where the title is entered. Line 1570 then assigns key$ as the left 6 characters of the title, which has been

```
190 PRINT ttl$:IF pr% THEN LPRINT ttl$
200 PRINT"Author - ";aut$;"   Subject - "sub$:
PRINT
210 IF pr% THEN LPRINT "Author - ";aut$;"  Su
bject - "sub$:LPRINT
220 RETURN
```

Figure 10.19 The title-order print subroutine.

```
1370 PRINT FNwindow$(1,0,2,90);cls$;
1380 PRINT TAB(34)"LIST BY SUBJECT"
1390 PRINT FNwindow$(28,0,2,90);cls$;hard$
1400 PRINT scrl$;
1410 GOSUB 580:IF UPPER$(K$)="P" THEN pr%=1 EL
SE pr%=0
1420 PRINT FNwindow$(3,0,24,90);cls$
1430 err%=SEEKRANK(fil%,0,2)
1440 IF err%<>0 THEN GOSUB 920
1450 WHILE err%=0 OR err%=101
1460 GET fil%
1470 GOSUB 230
1480 err%=SEEKNEXT(fil%,0)
1490 WEND
1500 RETURN
```

Figure 10.20 The subject-order list subroutine.

```
230 PRINT sub$:IF pr% THEN LPRINT sub$
240 PRINT"Author - ";aut$;"   Title - "ttl$:P
RINT
250 IF pr% THEN LPRINT "Author - ";aut$;"  ti
tle - "ttl$:LPRINT
260 RETURN
```

Figure 10.21 The subject-order print subroutine.

padded out with six extra blanks. The reason for the padding is that a very short title would otherwise give a short `key$` that would never match the stored 6-character key. Having obtained the title in this 6-character upper-case form, which means that you don't have to type the whole title and can ignore upper/lower-case, the SEEKKEY action is used. This allows a key and its record to be found, and requires the channel number, lock number (always 0), rank number and the key string to be specified. If this title cannot be found, line 1590 deals with the error with a call to the subroutine in figure 10.23, then a jump over the rest of the routine. This avoids problems that would be caused if the record-stepping commands were used from an indeterminate position. If the key has been located correctly, line 1600 gets the file, and line 1610 prints it on the screen. No provision has been made for hard copy here, because it doesn't appear necessary, but

```
1510 PRINT cls$;FNwindow$(1,10,2,70);
1520 PRINT TAB(25)"FIND BOOK"
1530 GOSUB 2030
1540 PRINT FNwindow$(5,10,6,70);
1550 PRINT TAB(9)"Please type title: ";
1560 LINE INPUT B$:PRINT
1570 key$=UPPER$(LEFT$(B$+"          ",6))
1580 err%=SEEKKEY(fil%,0,1,key$)
1590 IF err%<>0 THEN GOSUB 2360:GOTO 1700
1600 GET fil%
1610 PRINT cls$;:GOSUB 150:REM print it
1620 GOSUB 1900 :REM forward or backward
1630 PRINT FNwindow$(25,10,5,70);
1640 PRINT"Do you want to - "
1650 PRINT TAB(16)"1. Correct this entry?"
1660 PRINT TAB(16)"2. Delete this entry?"
1670 PRINT TAB(16)"3. Proceed, no changes"
1680 mx%=3:GOSUB 530:PRINT FNwindow$(3,0,26,90);
1690 ON choice% GOSUB 1730,1770,1890
1700 PRINT cls$;:PRINT ques$:PRINT YN$:GOSUB 580
1710 IF UPPER$(K$)="Y" THEN PRINT cls$:GOTO 1540
1720 RETURN
```

Figure 10.22 The subroutine for picking one title. This must allow for more than one title using the same key letters.

```
2360 REM Deal with incorrect title
2370 PRINT:PRINT"No such title found"
2380 FOR j=1 TO 1000:NEXT
2390 PRINT FNwindow$(3,0,26,90);
2400 RETURN
```

Figure 10.23 Dealing with the case of no title found.

since the printing subroutine contains the LPRINT statements, making `pr%` equal to 1 in this subroutine would give hard copy.

The next problem is that this may not be the correct record! As we noted earlier, it's possible for more than one record to give the same six letters of a title for the key, and SEEKKEY will always give the first one in the sorted list. We need, then, to be able to move forward through the records to find the correct one, and just in case we move too far, we need also to be able to move back. Line 1620 deals with this movement, and we'll look at it in detail shortly. Assuming that the correct record has been located (and it will be the *current* record), then lines 1640–1690 offer the choice of correcting the entry, deleting it, or just going on. Having made this choice and carried it out, you are then offered a chance to select another title or to return to the main menu.

```
1900 PRINT FNwindow$(20,10,3,70);
1910 PRINT"Press E for next, X for previous, S i
f correct"
1920 GOSUB 580:PRINT FNwindow$(5,10,6,70);:PRINT
1930 IF UPPER$(K$)="X" THEN GOSUB 1960:GOTO 1900
1940 IF UPPER$(K$)="E"THEN GOSUB 1990:GOTO 1900
1950 RETURN



1960 err%=SEEKPREV(fil%,0)
1970 IF err%<103 THEN GET fil%:PRINT cls$;:GOSUB
150:ELSE PRINT"No file"
1980 RETURN


1990 err%=SEEKNEXT(fil%,0)
2000 IF err%<103 THEN GET fil%:PRINT cls$;:GOSUB
150:ELSE PRINT"No file"
2010 RETURN
```

Figure 10.24 (a) The forward or backward menu stage; (b) the 'backward' routine; and (c) the 'forward' routine.

```
1730 PRINT FNwindow$(10,0,10,90);:flag%=1:GOSUB
750
1740 flag%=0:PUT fil%
1750 PRINT FNwindow$(3,0,26,90);
1760 RETURN
```

Figure 10.25 The short routine for making a *small* correction. Large changes can result in an item being in incorrect order. For large changes, delete the old item and add a new one as replacement.

```
1770 rn%=FETCHREC(fil%)
1780 GET fil%:key$=UPPER$(LEFT$(aut$,6))
1790 PRINT key$;" has been deleted."
1800 err%=DELKEY(fil%,0,0,key$,rn%)
1810 IF err%>103 THEN GOSUB 920
1820 key$=UPPER$(LEFT$(ttl$,6))
1830 err%=DELKEY(fil%,0,1,key$,rn%)
1840 IF err%>103 THEN GOSUB 920
1850 key$=UPPER$(LEFT$(sub$,6))
1860 err%=DELKEY(fil%,0,2,key$,rn%)
1870 IF err%>103 THEN GOSUB 920
1880 RETURN
```

Figure 10.26 Deleting a record. The complication is caused by the need to delete more than one key.

```
1890 RETURN:REM Important!


2060 CLOSE fil%
2070 RETURN


2080 IF ERR=55 THEN GOTO 2100
2090 ON ERROR GOTO 0
2100 PRINT"File already exists."
2110 FOR j=1 TO 1000:NEXT
2120 RESUME NEXT
2130 END
```

Figure 10.27 Odds and ends: (a) A RETURN used in the 'find book' menu; (b) the CLOSE fil% routine; and (c) the error-catching routine, set up to detect the 'File exists' error.

Now for the actions called from the selecting routine. First of all Figure 10.24 shows how we can move forwards and backwards through the records after having located a title. Line 1910 shows the keys used for this action, deliberately chosen to be the keys used by many word-processor and other programs. Lines 1930 and 1940 call up movement subroutines as requested, and if the S key was pressed, the subroutine simply returns at line 1950. The movement routines start at 1960 and 1990. The SEEKPREV action is to get the previous file in the list, and needs only the channel number and the lock number. Similarly, SEEKNEXT gets the next record. A value of `err%` less than 103 indicates success so that the record can be got from the main file and printed, using established subroutines. The routines are arranged to loop, using GOTO 1900, so that you can search through the records until you find the correct one. Do not overdo this facility, or you will come to the end of the file. This does not cause any problems, however, since you can then press the S key to return, followed by the choice of 3 to leave the record.

Now we need to look at what we do with a record that has been found, assuming that we want to change or delete. It's important to note here that 'change' means correct, and the correction should not be a major one. The reason is that this facility is a simple one, used only to correct minor errors. If you have an entry whose title is `WANDERING IN THE AUVERGNE`, then you can, with this action, change it to `GUIDE TO MALLARD BASIC`. It will still have the index keys of the original title, however, so that your Guide to Mallard BASIC would be listed with the 'W's in the title list, and with the position of the author of the original text, and in the same subject group. In other words, if you use this facility to change a title completely, it will always be in the wrong place in the list! The change routine is in figure 10.25, and is very simple. The call to 750 gets the original entry subroutine but because `flag%` is set, the routine returns at line 790. The `PUT fil%` then places the new record in the file and the routine returns.

Deleting a record is more involved. A record is automatically deleted when all of its key entries have been deleted, so if you are using records with just

one rank of keys, then deleting is quite straightforward. In this case, though, three keys have to be deleted, and the action requires the record number to be used. The routine is illustrated in figure 10.26, and starts with the FETCHREC action, yielding the record number `rn%`. The file is then fetched so that the correct keys can be extracted, and in line 1780, the author key is extracted. This is used in line 1800 with DELKEY, specifying the channel number, lock number, rank, key and record number. The other two keys are deleted in the same way. If only one key exists, a shorter version of DELKEY can be used.

Now we can tidy up some odds and ends, figure 10.27. The line (a) is a RETURN that is used in the FIND BOOK routine when choice 3, `Proceed, no changes` is used. By using this RETURN, the program then goes to line 1700 without doing anything. In (b), the `CLOSE fil%` closes the files in an orderly way, seeing that the files are in order. When a file is deleted, there can be differences between the index file and the main file that will cause problems until the files are corrected. This is done when the correct `CLOSE fil%` statement is issued. If for any reason, you want to carry out this correction at another time, there is a CONSOLIDATE statement that can be used.

Finally, (c) shows the error-trap routine. As the program exists, there is only one error that specifically needs to be trapped, the 'File exists' error that may be encountered when creating a new file. This is detected by its error number in line 2080, and calls up the routine that prints the message, pauses, and then resumes normal action on the next line. If the error is of any other type, line 2090 restores normal error-trapping and the program will stop with the error showing. You might like to add a GOSUB 2060 in a line 2085 to ensure that files will be closed in such an event. During program tests, no other errors appeared, so that this step was not put in, but if the program is expanded you might need to add more traps, and also guard against leaving files open. The only thing that can cause open-file problems in normal use is pressing the STOP key, and this can be disabled as described in the manual, using OPTION RUN. Adding a line:

```
15 OPTION RUN
```

will allow the program to become uninterruptible. Unfortunately, it also disables the use of ALT S for stopping the screen scrolling, so that you will need to add screen-paging commands (see chapter 7) if you still want to be able to carry out this action. The simplest method of doing this is to put a GOSUB 580 at the end of each print routine, or in the calling routines so that you have to press a key each time you want to see a record.

Chapter 11

Last roundup

This chapter contains a rag-bag of topics that were not vitally important in learning to use Mallard BASIC, but which you might find interesting now that your muscles are better developed, as it were. We still can't possibly cover all of the commands of this very large and useful version of BASIC, simply because there are so many that are for advanced users only and which would need many pages of explanation unless you have become very familiar with programming. In this chapter, then, we'll deal only with the more important outstanding topics, because by the time you have reached this stage you will be able to extract information much more easily from the manual.

Merge and Chain

If you are fairly experienced in writing and using programs in BASIC, you will be able to make use of the other methods of loading a program, apart from LOAD and RUN, that Mallard BASIC permits. One of these extra methods is MERGE, which allows you to add a BASIC program to one that is already held in the memory. The normal use of LOAD automatically erases any program in the memory, and loads the new program into the same piece of memory. By using MERGE correctly, you can join a program which is on the disc to one that is in the memory to form a longer program. There are, however, several rules for using MERGE that you have to obey rather carefully. First and most important is line numbering. The line numbers of the program on the disc must not be the same as the line numbers of the program in the memory. If any line numbers are identical, then the lines on the disc program that is MERGEd in will replace lines of the same number in the memory instead of adding to them. The second point is that merging two programs together will not ensure that they can

```
10 REM Second
20 x=2*a
30 c$="NEW "+B$
40 PRINT C$;" ";x
50 END
```

```
10 REM main program
20 a=6: b$="Example"
30 PRINT B$;" ";A
40 CHAIN "SECOND",,ALL
50 END
```

Figure 11.1 Using CHAIN to pass variables from one program to another.

run as one program. If the first program has an END statement, for example, then that's where it ends, and the other does not run! You cannot run two separate programs after a merge in this way, the two have to combine into one program. Even if the two do make correctly into one program, you have to be certain that the variable names are the same in both, and are used in the same way.

When you LOAD, RUN, or MERGE, any variables in the programs are automatically set to default values of zero for numbers and blank for strings. This normally means that you can transfer variable values only by saving variables on disc from one program and reading from another, but Mallard BASIC allows another pair of options. The command CHAIN will load a program in and run it, deleting the original program, and CHAIN MERGE will do the same, but delete only lines that are specified by an extra part of the statement. For most purposes, CHAIN is the statement that you are more likely to use, and the Manual shows the differences in detail. Both can allow variables from a running program to be preserved so that they can be used by another program. A filename is required following CHAIN or CHAIN MERGE, and if this alone is specified, the new program will be loaded, and will run from its first line. You can also force the chained program to start at a specified line number, so that `CHAIN"addit",200` would load the program `addit` and run it starting at line 200. The CHAIN command allows variable values to be passed on if the word ALL is added, and the merged program to be run from a specified line (as would be normal in this case). An additional facility of CHAIN MERGE, however, is to delete lines of the old program so as to economise on memory. For example, the command:

```
CHAIN MERGE "addit",200,DELETE 10-100
```

would delete lines 10 to 100 of the program that is running and then add program `addit` and run it from line 200. The short listing of figure 11.1 shows CHAIN in action. You should type and save program SECOND first, then the other program. When the main listing is run, it will print values, then the CHAIN action will load and run SECOND, printing the new values *which make use of the older ones*. You will see, when you list, that

program SECOND has completely replaced program MAIN, because we used the same range of line numbers in each. The reason for the two commas between the filename and ALL is that no starting line number has been used, but the program requires the commas as separators.

Print Fielding

The topic of print fielding will be completely new to you if your computing has been learned on other machines, and even if you have used one of the older Amstrad models, you may not be entirely familiar with the principles. It's a topic that rightly belongs in the category of more advanced programming, so that we shall do no more than outline it here. Print fielding is concerned with how a quantity, string or number, is presented on the screen. Fielding is not so much concerned with the position of print on the screen, which is dealt with by TAB, SPC and the printat and window functions that we shall look at later, but with the actual quantity itself. The field that is referred to is the space in which something is printed, and fielding means how the printed text or numbers will be arranged *in this space*. For example, you might want a heading centered, or placed hard against the right-hand edge of the screen or paper. You might want a number printed with only two places of decimals, with decimal point lining up, with dollar or pound signs and so on.

Figure 11.2 shows an example of fielding carried out with text. The screen is cleared, and line 30 then prints the name USING the field string \ \ with three spaces between the slash signs. These slash signs are the forward slashes, obtained by using EXTRA 1/2 (and I found that I needed to press the keys *twice* to get the symbol). The number of characters in this string, counting each slash and each space means that the field will be of this width, five characters. When the name is printed then, only five characters will appear, and it's the first five as you'll see when you run this. If the name is of fewer characters than is needed to fill the specified space, then it is left-justified, with the first letter of the name at the left-hand side of the field. The next one is less complicated. The field string is assigned to X$, and it consists of an exclamation mark only. This forces the print action to print only the first character of the string. Both of these actions could be carried out by using LEFT$, but sometimes the use of USING is more convenient. You can, for example, prepare a set of strings to specify string width and use whichever you like.

```
10 cls$=CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT cls$
30 PRINT USING"\    \";"Sinclair"
40 B$="Amstrad"
50 X$="!"
60 PRINT USING X$;B$
```

Figure 11.2 Fielding text with a PRINT USING string.

```
10 cls$=CHR$(27)+"E"+CHR$(27)+"H":PRINT cls$
20 FOR N%=1 TO 5
30 RESTORE 130
40 FOR X%=1 TO N%
50 READ F$:NEXT
60 RESTORE 140
70 FOR J%=1 TO 5
80 READ D#
90 PRINT USING F$;D#
100 NEXT:PRINT
110 FOR x=1 TO 1000:NEXT
120 NEXT
130 DATA "##,###.##","##.#","#,###.#","#.#","
#####"
140 DATA 1.57,11.236,10143.2,1071623,237.145
```

Figure 11.3 Numbers printed with different USING strings, showing rounding actions.

```
      1.57
     11.24
10,143.20
%1,071,623.00
   237.15

 1.6
11.2
%10143.2
%1071623.0
%237.1

    1.6
   11.2
%10,143.2
%1,071,623.0
  237.1

1.6
%11.2
%10143.2
%1071623.0
%237.1

    2
   11
10143
%1071623
  237
```

Figure 11.4 The printout from the program of 11.3.

Formatting control strings have a few applications for printing strings, but they really come into their own when you want to print numbers, particularly money amounts and items in columns. The characters that will be used here consist of the hashmark, the full stop/period/decimal point, and the comma. Used in a formatting string, the number of hashmarks and other characters specifies the total number of characters in a field, and the dot and comma show where a decimal point (one only), and commas can be placed. The program in figure 11.3 prints a set of numbers, each set using a different field specification. The loop that starts in line 20 will print five sets of numbers, and the correct field specification is obtained by lines 30 to 50. The statement RESTORE 130 ensures that the data for the field specification is taken from line 130, and by using the loop `FOR X%=1 TO N%`, you will end up with F$ assigned to the Nth. item in the `DATA` line. The RESTORE 140 in line 60 then picks the number data, and the loop that starts in line 70 reads various numbers so that they can be printed with the required format. When each loop of number printing is completed, the loop in line 110 makes the program wait for a few seconds so that you can take a look at the group of figures before the next fielding string is assigned.

Figure 11.4 shows the results of all this as printed on paper, substituting LPRINT for PRINT in line 90. The first fielding string uses a total of nine characters, so all fields are nine characters long. The numbers are justified to the right of the field, and since two hashmarks have been shown following the decimal point, there will be two decimal figures, whether the number contains decimals or not. Numbers with three places of decimals are *rounded* down to two, numbers with one decimal place have a trailing zero added. This ensures that the decimal points line up, which is ideal for displaying tabular data. One number cannot be displayed neatly in this field, because it has 7 figures and no decimal point. It would therefore consist of ten characters if we added a decimal point and two figures following the point. In such a case the number *is printed*, but not in the correct form and the % sign is used to show that this number does not fit. Note the effect of rounding on 11.236, which is printed as 11.24. This automatic rounding can save you a lot of program effort, but you need to be careful that it is doing what you expect it to. If, for example, you are working with such things as compound interest, you may be calculating to four decimal places, but displaying a rounded amount only. You will have to ensure then that the rounding does not work against you! Note that if you have not set your printer to the US character set, the hashmarks will appear as pound signs.

The other sets of figures follow this pattern, with rounding when the decimal places are not displayed, and the % signs showing if the total number of places is insufficient. Note in the last set that the effect of the formatting string ##### on 1.57 is to print this number as 2, rounding up to the nearest whole number, which is what the format calls for. The figure of 1.49 would be rounded to 1 in this position. In general, when you are working with numbers, it's better to allow for more places than you need preceding the decimal point, and for as many as you want following. If you need to display with three decimal places, and you are expecting numbers up to 502614 for example, use #,###,###.### as your formatter so that if a larger number comes along, you will not be treated to the sight of a set of percent signs and numbers in the wrong positions.

```
10 cls$=CHR$(27)+"E"+CHR$(27)+"H":PRINT cls$
20 F$="$$###,###.##"
30 FOR N%=1 TO 5:READ D#
40 PRINT USING F$;D#
50 NEXT
60 DATA .25,1.76,23.4,2176,31260.22
```

Figure 11.5 The floating dollar effect.

Another facility provided in Mallard BASIC is DEC$, which allows numbers to be put into string form and formatted. The syntax is:

```
A$=DEC$(number,format-string)
```

and if A$ is printed later you will not need to have a PRINT USING since the formatting has already been carried out. The action is for numbers only – see the manual for details.

Money amounts

Printing money amounts is always likely to be a requirement for a lot of applications, but the most useful USING format of Mallard BASIC for money amounts provides only for the dollar sign to be put into money amounts. This isn't too much of a hardship if you know how to alter the keyboard and printer characters (see any good book on business software on the PCW machines). The simplest way of placing a money unit sign, however, is to have it at the head of a formatting string, as for example:

```
F$="£##,###.##"
```

which would put the British pound symbol at the head of a ten-character field. Obviously, you could place whatever symbol you needed in this position, or in a similar string. Using the money symbol at the head of the field is not always satisfactory in visual terms, however. If you allow for five figures between the pound sign and the decimal point, for example, the pound sign will always be placed there, so that you get amounts such as £ 4.56 printed, with the pound sign looking much too remote from most of the figures. What is needed is what is called a 'floating' money symbol, one which will *just precede* the first figure of a money sum.

This ability to float the symbol is provided in PRINT USING – but only for the dollar sign. To float the dollar (sounds like an economist's prescription) requires putting another dollar sign preceding the dollar sign, then the usual set of hashmarks that mark out the field size for your money amounts. Figure 11.5 shows the listing that uses a floating dollar sign and you can see when you run this that the dollar sign just precedes the money amount in each case. The display is much more satisfactory than that of the fixed monetary sign, but only for dollars! Another facility is to use the asterisk in a format string so that you get asterisks in place of blanks – this is very often used in cheque printing.

Print positioning and windows

Though Mallard BASIC differs from the Locomotive BASIC that is used in the CPC machines, it can in many cases carry out similar actions in different ways. Two notable absences from Mallard BASIC are PRINT AT and WINDOW. PRINT AT would allow you to print at any position on the screen, and WINDOW creates a screen size on which all printing is done until a new WINDOW is defined. We have used examples of these actions in examples, and the way in which they are obtained in Mallard BASIC is by the use of ESC characters, meaning characters that are printed following CHR$(27). Note, incidentally, that there is an important difference here. If you use `PRINT CHR$(27)` followed by a character, it will have some effect on the screen display. If you use `LPRINT CHR$(27)` followed by some character, it will have some effect on the printer. The two are quite separate, and have different effects. The codes and their effects and listed in Book 1 of the PCW manuals, pages 126 to 137 for the printer, and pages 140, 141 for the screen.

At this point it's useful to note why these actions are carried out in different ways. Actions like clearing the screen and printing in specific parts of the screen are likely to be carried out more than once in the course of a program. For that reason, it's undesirable to have to type out the full statement each time it's needed, and we have seen how the clear-screen sequence can be assigned to a string, `cls$` so that the statement `PRINT cls$` will have the effect of carrying out the action. Other actions, however, need parameters. To print at any point on the screen, for example, requires you to supply two numbers, the row number and the column number. These will normally be different each time the action is required, so that the values must be passed on, and this is most easily done by using a defined function. For setting a window, four numbers must be passed on so that this too requires a defined function.

```
10 cls$=CHR$(27)+"E"+CHR$(27)+"H"
20 PRINT cls$
30 DEF FNat$(r%,c%)=CHR$(27)+"Y"+CHR$(r%+32)+
CHR$(c%+32)
40 DEF FNwindow$(R%,L%,H%,W%)=CHR$(27)+"X"+CH
R$(32+R%)+CHR$(32+L%)+CHR$(31+H%)+CHR$(31+W%)
50 PRINT FNat$(15,42);"MIDDLE"
60 GOSUB 1000
70 PRINT FNwindow$(0,0,30,90);
80 FOR n%=1 TO 30*90:PRINT"A";:NEXT
90 GOSUB 1000
100 PRINT FNwindow$(7,30,7,30);cls$;
110 FOR n%=1 TO 7*30-1:PRINT"B";:NEXT
120 PRINT FNwindow$(0,0,30,90);
130 END
1000 FOR x=1 TO 1500:NEXT:RETURN
```

Figure 11.6 The AT and WINDOW actions in defined string function form.

```
10 ON ERROR GOTO 1000
20 PRINT"Type a word, please"
30 INPUT A$
40 L=LEN(A$)
50 PRINT 1/L
60 END
1000 PRINT"Word has no letters"
1010 RESUME 20
```

Figure 11.7 A simple illustration of ON ERROR GOTO.

Figure 11.6 shows these actions written in function form, both of which we have used previously. The FNat$ needs row and column numbers, both of which can be represented by integer variables. These have to be placed in the function as CHR$ numbers, with 32 added to each. The action is illustrated in line 50, where it has been used to print in the centre of the screen. The action uses row and column numbers that apply to whatever screen is in use. If, for example, you are working with a screen size of only 10 columns and 5 rows, then a FNat$(2,5) will place the cursor on the 3rd. row (counting from zero) and the 6th column, also counting from zero. If you use numbers for this function that cannot be applied (such as numbers that would place the cursor off the screen, the cursor will be put to the nearest edge of the screen.

The use of windows is another step in screen control. A window is defined as a rectangular part of the screen of any size up to the full size. Instead of specifying the positions of the four corners, however, the window is defined by the top row number, the left-hand column number, the number of rows and the number of columns. In this case, the numbers are counted from 1 to the maximum number available. The importance of a *window* in Mallard BASIC is that once a window is defined, it constitutes the only piece of screen that will be used until another *window* is defined. This is illustrated in Lines 100, 110, in which a small window is defined in a screen filled with As, and this space cleared to be filled with Bs. Neither the clearing of the screen nor the printing of letter B has any effect on the screen outside the defined window. Note how the defined function for the window is written, with 32 added to the start numbers, and 31 added to the height and width numbers. When a window within another piece of screen is being printed in, you should try to avoid printing in the last character position in the window, because this always causes scrolling, leaving a gap. To see this in action, remove the -1 from the loop statement in line 110.

Good screen use demands planning, and if you can get hold of graph paper or other planning paper with 90 × 30 squares, this planning will be made much easier. Before you write any program in which such use will be made of the screen, plan out the windows on the paper and assign a number to each. You can then make use of these numbers when you come to use the program. One way is to have window defining lines such as:

```
T1%=0:L1%=0:H1%=30:W1%=90
```

defining the numbers for a full screen, and :

```
T2%=10:L2%=20:H2%=5:W2%=70
```

defining a smaller screen. It's then easy to switch windows in the program by using statement such as:

```
PRINT FNwindow$(T2%,L2%,H2%,W2%);
```

and if you want to change the window sizes you don't then have to change each FNwindow$ statement, only the assignment lines.

Error Trapping

At several stages in this book we have come across the idea of mugtrapping. This, politely referred to as 'data validation', is a way of checking data that has been entered at the keyboard, to see if it makes sense or not. Mugtrapping is normally carried out by using lines such as:

```
16Ø IF LEN(A$)=Ø THEN GOTO 1ØØØ:GOTO 5Ø
```

and you need a separate type of mugtrap for each possible error. This can be fairly tedious, and it usually turns out that there is one other error that you haven't spotted. In addition, there are actions that will generate errors which stop the program, like trying to work with a disc file that doesn't exist because you are using the wrong disc. Mallard BASIC is one of the exclusive few versions of BASIC that offers you mugtrapping statements that will intercept such errors, and the main one is ON ERROR GOTO. Figure 11.7 gives a very artificial example – a real-life example would involve too much typing. In this example, the length of a word is measured, and the number inverted (divided into 1). This is impossible if the length is zero, and the ON ERROR GOTO is designed to trap this. You could get a zero entry, for example, by pressing RETURN without having pressed any other keys. Now normally, when this happened, you would get an error message, and the program would stop. The great value of using ON ERROR GOTO is that the program does not stop when an error is found, instead it goes to the subroutine. In this example, the subroutine prints a message, then resumes on line 20. It's delightfully simple, but it's something that calls for experience. You see, if your program still contains things like syntax errors, these also will cause the subroutine to run, and this can make the program look rather baffling as it suddenly goes to another line. Always omit the ON ERROR GOTO line until you have tested the program sufficiently to remove all syntax errors. You can also make the error-handling point out errors other than the ones that you are particularly looking for, as we shall see.

RESUME can be used in any of three ways. Firstly, used on its own, it will cause the whole program to run from the place where the error was found when RESUME is executed. This is not always useful, because if the error cannot be corrected (like a `disc full` error) in the error routine lines, then you will get an endless loop. Secondly, as shown, with a line number, it will cause the program to resume at that line. The third method of using is RESUME NEXT, in which case the program will resume at the line *following the line in which the error was trapped.* Once again, you may have to be careful here, because you need to make sure that the statement you want

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H";
20 ON ERROR GOTO 1000
30 n%=-5
40 FOR j%=n% TO 5
50 PRINT LOG(j%)
60 PRINT"Line 60":NEXT
70 END
1000 IF ERR=5 THEN PRINT"There is an error";E
RR;" in line ";ERL
1010 RESUME NEXT
```

Figure 11.8 Allowing resumption of action, ignoring an error.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H";
20 ON ERROR GOTO 1000
30 FOR N%=1 TO 10
40 ON ERROR GOTO 0
50 PRINT"The next message has been forced by
the ERROR comand."
60 ERROR 61
70 END
1000 IF ERR=26 THEN PRINT"NEXT missing, but
program continues."
1010 RESUME 40
```

Figure 11.9 Using ERROR with a 'standard' number to simulate the effect of an error. This is very useful for testing actions that you would not normally encounter.

to use is in such a line and not, for example, part of the line in which the error occurred, separated by a colon.

Figure 11.8 shows another example. Line 20 sets up the ON ERROR GOTO command, while lines 30 to 60 generate a set of errors by using invalid (negative) numbers in the LOG function. This would normally cause an `Improper argument` error message to be displayed, stopping the program. Because of the use of ON ERROR GOTO 1000, however, the detection of the error causes this line to be run, and in line 1000 the type of error is checked. If it is indeed an 'Improper argument' error, then the error message will be printed, showing the error line by use of the variable ERL and the error number by ERR. Irrespective of what type of error occurred, the RESUME NEXT in line 1010 will make the program go to line 60, because the error is detected in line 50.

There is another command word, ERROR, which can be used to make an ON ERROR GOTO execute, and which is very useful for testing the effect of error trapping. You may, for example, want to find out how your program might react to a disc being full, which is not something that you can easily cause during testing. ERROR can be used in either of two ways, either with the numbers that are listed in pages 352 to 358 of the Manual, which are the BASIC error numbers, or using numbers outside that range,

particularly in the range 150 to 200. If ERROR is used with the standard error codes, it will make the program's error trapping respond as if one of the standard errors had occurred. Using ERROR with a number that is not a standard one allows you to create your own error messages. The standard use of ERROR is indicated in figure 11.9, in which the routine at line 1000 detects a missing NEXT in a loop – not something that you would normally leave in a program! Line 40 then turns off the error trapping, so that when ERROR 17 is invoked, you can see the normal error message for this error appear. The normal error-trapping is turned on by using ON ERROR GOTO 0. Note that the missing NEXT is detected *before* line 40 runs, which shows that the program checks for this before executing the statement. Finally, figure 11.10 illustrates a 'home-made' error number in action. Normally, the error trap would include a line ON ERROR GOTO 0 in order to avoid any chance of an error causing an endless loop, but you'll find that if this line is included when these non-standard codes are used, then the error message `unknown error in 30` will appear.

```
10 PRINT CHR$(27)+"E"+CHR$(27)+"H";
20 ON ERROR GOTO 1000
30 ERROR 190
40 PRINT"END"
50 END
1000 IF ERR=190 THEN PRINT"Home-made error here"
1010 RESUME NEXT
```

Figure 11.10 Using a non-standard ERROR number for other purposes.

Appendix A

A self-starting disc

You can make your BASIC program disc self-starting, meaning that you can use it in the drive immediately after switching on, to load in CP/M and then BASIC, automatically. The method of doing this and many other 'set-up' actions is provided on your Master disc set in the form of a program called RPED, which is a form of simple text-editor for 'command files' that will cause CP/M actions to be carried out. In this book, which deals with Mallard BASIC, there is no space to go into detail of the CP/M operating system, so that the instructions that follow assume that you have some experience with file copying using PIP.

1 Copy the CP/M file, the one that has the extension EMS to a fresh disc, and copy also BASIC.COM, the two RPED files, and SUBMIT.COM
2 Type SUBMIT RPED.SUB RETURN
3 Press key F1 to create a new file and give the old name as RPED.SUB
4 Type as the new name PROFILE.SUB
5 Edit out lines, leaving only BASIC in the list (unless your RPED file contains only BASIC.COM)
6 Press RETURN then EXIT
7 Press EXIT to quit.

You should now have the files that are needed for self-starting. To test this, remove disc(s), and restart the machine. Now insert your new disc. It should start up CP/M and then load in BASIC. About 75K of the disc space is used by CP/M, BASIC and the other files.

Appendix B

The CLS key

The PCW machines provide for any key to have its actions (alone, with SHIFT, with ALT and with EXTRA) redefined, and also for some keys to be used for 'expansion codes'. An expansion code is a complete phrase that can be assigned to a key so that pressing the key will provide the phrase. A particularly useful phrase to assign in this way is the statement for clearing the screen, and the method is illustrated below. The manual gives details of the expansion keys and how to assign different number codes, but for most purposes it's easier just to use the keys that have already been assigned with numbers for this purpose. One such is the CAN key, which has the code 139.

This key can be used to clear the screen by writing a short file consisting of an instruction:

```
E 139 "PRINT CHR$(27)+CHR$(69)+CHR$(27)+CHR$(72) ^M"
```

the effect of which is to assign the screen clear string and the carriage return (coded as ^M) to the key. Because the string to be assigned must be enclosed in quotes, you cannot have items such as E and H in the string, and these have been replaced by their ASCII codes, using CHR$. To create the file, you can use RPED, and the suggested steps are as follows – when I refer to the 'disc' I mean the disc that will contain your copy of BASIC and programs in BASIC.

1 With the computer running CP/M, place the System disc, side 1, in the drive and type SUBMIT RPED.SUB RETURN
2 When the menu appears, press key F1, and enter when prompted the filename RPED.SUB. Press RETURN.
3 Remove the System disc, and put in your disc.
4 In response to the prompt, type the filename clsk, with no extension; or whatever filename you want to use.
5 When the file appears, delete the BASIC.COM entry, and any others, and type the line shown above. Remember that the up-arrow is obtained by using EXTRAU. Press the RETURN key at the end of the line, and then press EXIT. This places the file on the disc.

6 The menu appears again, and you can once again press EXIT.
7 You must have SETKEYS.COM on your disc, so copy this over from the System disc, using PIP, if you do not have it already.
8 To activate the key, type from CP/M SETKEYS.CLSK. This will make the key produce the phrase, but it cannot be used in CP/M, only in BASIC.
9 When BASIC is running, pressing the CAN key will clear the screen.

Note: You can make the assignment of this key part of your start-up routine. If you are using a self-starting file as detailed in appendix A, then adding to the file prior to the *basic* line, the line:

```
setkeys clsk
```

will activate the CAN key before BASIC is loaded in. You can also add a line that produces a slashed zero on the printer, see appendix E.

Appendix C

The OPTION commands

Mallard BASIC contains several reserved word phrases using the word OPTION. This is associated with other words to produce some effects that are useful, but some of them are only of interest to the advanced programmer, and some are useable only by machine code programmers.

OPTION BASE allows array subscripts to start either from 0 or from 1 by typing one of these digits following OPTION BASE. The default is 0, so that every array in Mallard BASIC will have a zero item, A(0), B$(0) and so on. If you use only members starting at A(1), B$(1) and so on, and have a large number of such arrays, you can save memory by starting your program with OPTION BASE 1.

OPTION FIELD Ø is a way of altering JETSAM so that the two bytes that are used for the record number have to be declared in a FIELD statement. It is most unlikely that you would ever need this facility.

OPTION FILES "letter" is a way of changing the drive letter to A, B or M. You can also change the user number within a program in this way by using a number between the quotes. This can save a lot of typing if you have more than one drive and want to keep files on the second drive. You can return to drive A with OPTION FILES "A".

OPTION INPUT, OPTION LPRINT, OPTION PRINT must not be used unless you have a thorough understanding of machine code routines.

OPTION NOT TAB and OPTION TAB affect the way that BASIC treats the TAB character, and are not likely to be of interest to most programmers.

OPTION RUN prevents the STOP key, and the actions of ALTC or ALTS, from stopping a program. Normally, these keys are tested at regular intervals, so that using this option allows BASIC programs to run faster as well as being uninterruptible. The action is cancelled when RUN is used, so that this option must be used as a program line. The action is reversed by using OPTION STOP.

Appendix D

Editing and debugging

Editing means changing something that has already appeared on the screen. You can, of course, delete a character by using the DEL key, or you can retype a faulty line. You can also delete a range of lines by using the DELETE command (such as `DELETE 1Ø-1ØØ`, `DELETE 2Ø-`, `DELETE -5ØØ`). Editing, however, means changing one feature of a line without having to change anything else. Any feature of a line, including its line number, can be changed by editing. The editing process can be carried out:

(a) While a line is being entered, before RETURN has been pressed,
(b) At a later stage, after a line has been entered, but before the program is run,
(c) When an error is signalled during running,
(d) When you see an asterisk appear against a line number during entry, using AUTO, to signal that this line contains a statement already.

Dealing with these in order:

(a) While a line is being entered, all of the line-editing commands, below, can be used. Editing is completed by pressing the RETURN key.
(b) When the line has been entered, but the program has not been run, you can use `EDIT` *`linenumber`* as detailed below.
(c) When the program stops with an error message, the line in which an error has been traced will appear on the screen, with the cursor on its line number. You can then edit the mistake.
(d) When a line number selected by AUTO corresponds to an existing line, the computer enters edit mode automatically.

Mallard BASIC uses line editing, which means that you have to call up the line that you want to change, using a command like `EDIT 2ØØ` (to edit line 200). The cursor can then be moved over the line to locate and correct the mistake. If you don't already have the line on the screen with the cursor on it (as when an error is signalled), then use the `EDIT` *`linenumber`* command to get the line in place. Remember that there must be a space between the `T` of `EDIT` and the first digit of the line number. The result will be to place the

line on the screen with the cursor over the first digit of the line number. You can now move the cursor, using the arrowed keys. You can delete a character which the cursor is over by pressing the DEL (forward) key. By holding this key down, you can clear everything that is to the right of the cursor. Alternatively, you can delete whatever is *to the left of the cursor* by using the DEL (back) key. Typing a character will insert that character *at the cursor position*. Press RETURN to complete editing a line. Remember that you can edit a line in this way while you are entering it. If you have typed:

```
PRINT THIS IS THE END"
```

A useful variation, when you have a number of consecutive lines to edit, is to type `AUTO`, followed by the line number of the first line you want to edit. This will bring the line to the screen for editing, and when you finish editing it, the next line will be presented. You can stop the process by pressing STOP. If you don't want to change one of these lines, you simply press RETURN when the line is presented for editing. You can also use this method to renumber a block of lines, but you must remember to delete the old numbers afterwards. In addition, GOTO and GOSUB destinations are not automatically renumbered as they are when you renumber normally.

Digging out the bugs

In computing language, a fault in a program is called a 'bug', and someone who puts the faults there is called, of course, a programmer. Your programs can exhibit many kinds of bugs, and these are indicated by the error messages that you get when you try to run a program. Some of these messages are pretty obvious. `Line does not exist`, for example, means that you have used a command like `GOTO 1000` or `GOSUB 1000` and forgotten to write line 1000. It can also appear if you have tried to `DELETE 1000` with no line 1000, or if you have a `THEN 1000 ELSE 2000` following an `IF` somewhere.

The most common fault message is `Syntax error`. This means that you have wrongly used some of the reserved words of BASIC. You might have spelled a word incorrectly, like `PRIBT` instead of `PRINT`. You might have missed out a bracket, a comma, a semicolon, or put a semicolon in place of a colon. The most common mistake, however, is to miss out a space before or after a command word. Any spelling errors of command words, or missing spaces which allow command words to be joined to other words, can be found easily if you always enter your programs *using lower-case*. Since the computer always converts all reserved words to upper-case, you will find that mistakes are highlighted by being still in lower-case. Machines can't tell what you meant to do, they can only slavishly do exactly what you tell them. If you haven't used BASIC in exactly the way the machine expects, you'll find a syntax error being reported. Another common error is `Improper argument`. This usually means that something silly has happened involving a number. You might, for example, have used `LOG(-5)`, which is an obvious mistake, but it's not so obvious if the command happens to be `LOG(N%)`, and `N%` has been changed to a value of –5. Anything that makes use of numbers, like MID$, LEFT$, RIGHT$, STRING$, and others can

have an incorrect number used – and this will cause the Improper argument error. You will also find that using a negative number in SQR(N), a negative or zero value in LOG(N), and other mathematical impossibilties will cause this error message. The cause shouldn't be hard to trace, because the machine tells you which line caused the trouble.

Even when you have eliminated all of the syntax errors and improper arguments, you may still find that your program does not do what it should. Mallard BASIC does what any BASIC of the '80s should do – it gives you a lot of ways of finding out exactly what has gone wrong. One of the most powerful of these is the ALTS key combination. This, as you know, stops the action of the machine when you use it once, and restarts it when you press ALTS again. This can be very useful for bug-hunting in screen display routines, but for other program sections, pressing STOP is more useful. This stops the program, and prints the line number in which the program stopped. What you probably don't know, however, is that you can print out the values of variables, and even alter values while the program is stopped, and then you can make the program resume by using a CONT or a GOTO. Of the two, CONT is preferable, because it will continue from where the program left off. You cannot, however, make use of CONT if you have called up an edit, or changed the program in any way. This will give a message such as Cannot Continue or possibly Unexpected NEXT (if you stopped in a FOR..NEXT loop). Suppose, for example, you are running a program which uses a slow count, and you press STOP at some early stage. The program stops, and you get a message like Break in 40. That line number, 40, is important, because you can start the program again at that line or another one *providing* you don't edit, delete or add to any of the lines of the program. Suppose that the counter variable is N%. Try typing ?N%, and RETURN. This will give you the value of N%. Now try N%=998 (say) and press RETURN. Type CONT, or GOTO 40 (or whatever line number you stopped in). You will then see the count start again – but at 998!. This is an excellent way of testing what will happen at the end of a long loop. Testing would be a rather time-consuming business if you had to wait until the count got there by itself.

Tracing the loops

One way in which a program can be baffling is when it runs without producing any error messages – but doesn't run correctly. This is really a sign of faulty planning, but sometimes it's an oversight, and the use of the STOP key method of tracing a fault can be very handy, because it allows you to print out the state of the variables at any stage in the program, and then carry on. Sometimes you want a simpler form of tracing, though. If your program contains a lot of IF..THEN..ELSE lines, it often happens that one of these does not do what you expect. In such a case, Mallard BASIC provides help for you in the form of two commands TRON and TROFF.

TRON means TRACE ON, and its effect is to print on the screen the line number of each line as it is executed. The line numbers are put between square brackets, and they are printed at the start of a screen line, in front of

anything the program prints. Try typing TRON and then running a slow-acting program. TRON is particularly useful if you aren't sure what a program is doing, and it can be very handy in pointing out when something goes wrong with a loop. Remember that you can combine TRON with other debugging commands. You can, for example, stop the program, alter the variables, and then continue, with TRON showing you which lines are being executed. Typing TROFF RETURN switches off this tracing process.

Appendix E

The slashed zero and the hashmark

For listings and for many other purposes, it's useful to have the zero slashed, but the printer does not use this as a default. The slashed zero can be obtained by sending an ESC code to the printer, and this code can be sent either from BASIC, using a BASIC statement in a program or as a direct command, or from the operating system at the time when a self-starting disc is loading in its files.

From BASIC, the statement that is needed is:

```
LPRINT CHR$(27)+"X"
```

and it's often useful to change to the US character set so that you have the use of the hashmark. This is done by using:

```
LPRINT CHR$(27)+"R"+CHR$(Ø)
```

If you want to prepare a file that can be used in a self-starting disc, then use RPED as detailed in Appendix B, but with the following file lines:

```
^'ESC'X
^'ESC'R^'Ø'
```

and save with the file name of `zeroslsh`. Using `SETLST ZEROSLSH` in your SUBMIT file will then set the printer as required. Note that you will need `SETLST.COM` on your disc in order to run this file.

The same techniques can be used to set the printer in any other way that you want, either as part of a BASIC program or during loading of BASIC. The manual details the printer control codes that can be used to change typestyle, paper sizes, character assignments and so on.

Appendix F

Boolean actions

The logic actions of AND, OR and NOT appear at first sight to have very odd effects when used on numbers. The effects can be understood only if you know how numbers are stored in binary form, and if this is a closed book to you, then I suggest that you look at a book on machine code programming relating to the Amstrad CPC464 or other Amstrad machines. The important point as far as we are concerned here is that Mallard BASIC stores all numbers in binary form, and the logic actions work on these binary numbers. If you type NOT(7) for example, then you are acting on the binary form of 7, which if stored as an integer is 0000000000000111. The action of NOT is to change each 1 to 0 and each 0 to 1, so that the result is 1111111111111000. This is the number 66528 in denary, but Mallard BASIC follows the normal convention that any integer number above 32767 is a negative number, and its true value is obtained by subtracting 65536. This gives –8, which is the result of the command `?NOT(7)`. Figure F.1 shows how the actions of OR and AND can be interpreted for numbers.

AND rule is 1 AND 1 is 1, all other cases 0
OR rule is 0 OR 0 is 0, all other cases 1.

When two numbers are ANDed or ORed, the result is a number which is the binary equivalent of the action on each bit. For example, the binary equivalent of the number 22 is ØØØ1Ø11Ø, and the binary equivalent of the number 14 is ØØØØ111Ø. If these are ANDed, the bits in each column are compared, and a 1 is written as a result only if each bit is a 1. In the example, this gives:

```
ØØØ1Ø11Ø
ØØØØ111Ø
--------
ØØØØ11Ø
```

and this result is the binary equivalent of 6. Thus programming `PRINT 22 AND 14` will give the answer of 6.

If we use the same numbers to illustrate OR, we get:

```
ØØØ1Ø11Ø
ØØØØ111Ø
--------
ØØØ1111Ø
```

which corresponds to 3Ø in ordinary (denary) numbers. The result of `PRINT 22 OR 14` is therefore 3Ø.

Mallard BASIC contains no routines for converting denary to binary, so that to become proficient with these logic actions, you need to be able to convert numbers into binary form and back. Any good book on machine-code for any of the Amstrad machines will cover this topic.

Figure F.1 Interpreting the actions of OR and AND on number quantities.

Appendix G

Items omitted

In any text dealing with BASIC, some items must be omitted, because to explain them in detail would need another book. In particular, the reserved words that deal with machine language operation or with memory manipulation have been omitted, and you will need to consult a book on machine code for explanations of such words. These include CALL, DEF USR, HIMEM, INP and INPW, MEMORY, OUT and OUTW, PEEK, POKE, USR, VARPTR and WAIT.

The remaining reserved words are those which you are most unlikely to use until you are thoroughly proficient in BASIC, at which time you should be able to make use of the details in the manual more effectively than you might on first contact. These are COMMON, FILES, FIND$, OSERR, and WIDTH.

INDEX

PROGRAM YOUR PCW!

This book is about how to use Mallard BASIC on Amstrad's PCW computers. It starts out from the very beginning, assuming you know nothing about BASIC or programming. In easy stages, it takes the reader through displaying messages on the screen, data statements, formulae and functions, loops, and string handling.
Later chapters deal with subroutines and with data files. Both serial files and random-access files using JETSAM are explained in easy to understand terms.

By the end of this book, readers should be able to write programs suitable for their own special needs, be they in business or in the home.

GLENTOP

Glentop Press Ltd
1 Bath Place
Barnet
Herts EN5 5XE

ISBN 1-85181-004-8

9 781851 810048