PCWsBAS.WS4

- "PCW: *Streamlined* BASIC"
  Geoffrey Childs
  PCW-World, 1990

A comprehensive guide to *streamlining* Mallard BASIC on the Amstrad PCW range

(Retyped by Emmanuel ROCHE.)

Published by PCW-World, Meadway Court, Bloomfield Street North, HALESOWEN, West Midlands B63 3RE, England.

ISBN 0 9515486 0 3

First Published October 1989
Second Edition July 1990

A disc containing the programs printed herein is supplied with this book.

Acknowledgements:

My sincere thanks are due to my many correspondents and friends who have given me ideas, answered my queries and made helpful comments which have all contributed to the preparation of this book. In particular, I would like to thank David Wilson, who has made numerous suggestions (some of which are printable) and assisted with the final editing. I am also very grateful to Gerry Austin of PCW-World, who has managed to publish this book within two months of our initial meeting -- incredible!

Geoffrey Childs, October 1989.

Table of Contents

Part 1

Introduction

Chapter 1: BASIC matters

1.1 Purpose
1.2 BASIC extensions
1.3 Your own style
1.4 Planning

Chapter 2: Everyday problems

2.1 Escape sequences
2.2 PRINT and LPRINT
2.3 Randomizing and the clock
2.4 Only 31K?
2.5 Integer variables, etc
2.6 Loops
2.7 Approximations
2.8 AND, OR, XOR
2.9 IF...

Chapter 3: Input techniques

3.1 INPUT, INKEY$, etc
3.2 INKEY$ or...
3.3 Filename checking
3.4 Date checking
3.5 Checking input
3.6 ON x GOTO or GOSUB
3.7 Menus

Chapter 4: Files

4.1 Sequential, random, or Jetsam?
4.2 Saving memory
4.3 File transfers

Chapter 5: Editing and debugging

5.1 Editing
5.2 Debugging
5.3 AUTO and RENUM
5.4 Subroutine libraries?
5.5 Mallard 'bugs'

Chapter 6: Style

6.1 To REM or not to REM
6.2 ON ERROR GOTO
6.3 Long or short lines?
6.4 Protection and OPTION RUN
6.5 Where does one put subroutines?
6.6 GOTO taboo?
6.7 PRINT

Chapter 7: It pays to increase your word power

7.1 FIND$
7.2 DEF FN
7.3 MID$ and VARPTR
7.4 LSET and RSET
7.5 INSTR

7.6 CHR$ and ASC
7.7 PEEK, POKE, CALL, and USR
7.8 LIST

Chapter 8: Writing for all PCWs

8.1 Mallard 1.29 and 1.39
8.2 Using the discs available
8.3 Machine code and compilers

Chapter 9: Intermission!

9.1 Can you INPUT a function?
9.2 UDG program
9.3 Sound and OUT

Part 2

Chapter 10: A magical mystery tour

10.1 The transport
10.2 Equipment
10.3 The map

Chapter 11: Graphics

11.1 Binary patterns
11.2 The visible screen
11.3 The screen characters
11.4 Roller RAM
11.5 SCRRUN
11.6 A new screen clear
11.7 A little frivolity!
11.8 Plotting a pixel
11.9 Saving screens
11.10 Altering the character set
11.11 Further graphics

Chapter 12: Sound

12.1 If music be the food of love...

Chapter 13: BASIC and CP/M

13.1 Input and output
13.2 DMA and FCB
13.3 Reading and writing files
13.4 The file directory
13.5 Other BDOS calls
13.6 Writing CP/M files
13.7 Combining BASIC and CP/M Plus

Chapter 14: Useful addresses...

14.1 The Mallard interpreter
14.2 CP/M Plus addresses

Chapter 15: Going places

15.1 Accessing the unaccessible

15.2 CALLing to BIOS
15.3 Screen dump
15.4 Where is the cursor?

Chapter 16: The keyboard

16.1 Keyboard information
16.2 Setting a key
16.3 Grandpa

Chapter 17: The 'M' disc

17.1 Screen saving

Chapter 18: Direct disc access

18.1 Format
18.2 Disc editor

Chapter 19: Interrupts

19.1 Excuse me, I'll only be a microsecond!
19.2 The routine timer

Appendices

1: Program disc
2: Turnkey discs
3: Documentation for DWBAS
4: Multiple POKE
5: Disassembler
6: Menu subroutine
7: Multiple input
8: Bibliography and discotheque
9: Printer fonts
10: LNER 'Mallard' listing
11: Notes to the second edition

Index

Part 1

Introduction

You may be a very good programmer, you may just think you are, or you may realise that you are only a beginner. You may have written thousands of programs... or just the one. It does not matter in the least. The less you know, the more you can learn from this book.

Will this book help you to write spectacular programs with mind-boggling graphics? Or will it just help you to write a program for your weekly budgeting that works efficiently? The answer lies solely in your hands. This book will give you the means to produce programs with all the bells and whistles. It will also help you to do the much simpler programming tasks well. What it will NOT do is to teach you how to PRINT "Hello". Nor will it give the

syntax for every Mallard BASIC word. You need an elementary tutorial for the former, and a manual for the latter.

Part 1 is about the things that you CAN officially do with Mallard BASIC. It is hoped that the discussions will improve the proficiency of your programming, so that the simple things are done effectively. Part 2 is about the things you CANNOT do with Mallard BASIC. As we shall see, there is no such word as CAN'T! Admittedly, we shall have to make some recourse to machine code in Part 2: but have no fear, you will not be asked to understand it. Any BASIC programmer can use code routines that have already been written.

The final section is a series of appendices. This includes a simple extension to Mallard BASIC and a few other programs that do not conveniently fit into the text. There are plenty of short programs and routines that are included and explained in the context of the text. Each of these will be labelled with a name such as C4P2 (short for Chapter 4 Program 2). These can be found on the program disc supplied with this book.

In a book about computers, it is impossible to avoid 'jargon' completely. I hope that most of it will be explained as we progress. It may be helpful just to explain a few words immediately. MEMORY can be thought of as the brain cells of the computer. Each cell of memory (there are over 500,000 of them in the Amstrad PCW8512) contains a number 0-255 that the computer 'remembers' until it is changed, or until the computer is switched off. CODE is an all-purpose word that means symbols, letters, words, and language that the computer can operate upon (given a little bit of help by what is already in its memory). The PROCESSOR is the electronic device that turns the numbers in memory into simple activities that the computer can carry out. MACHINE CODE is the code that the processor can understand. A programming LANGUAGE converts the code that a programmer writes into code that the computer can use. Mallard BASIC is called an INTERPRETER as it does that conversion, line by line, each time a program runs. Other languages use a COMPILER which turns a complete program (SOURCE) into a new program (OBJECT). The object program is in machine code. We shall not be much concerned with compilers. RESERVED words are words that have a special meaning to the computer in languages such as BASIC. PRINT, RANDOMIZE, and SQR are examples of reserved words in BASIC. DO is a reserved word in the Pascal language, but means nothing in BASIC.

If you did not receive a program disc when you bought the book, please let us know at PCW-World ('Cotswold House', Cradley Heath, Warley, West Midlands B64 7NF) and we will provide a disc free of charge on receipt of some evidence of the sale.

Chapter 1: BASIC matters

Just before the second world war, LNER ("London & North Eastern Railway") engine number 4468 achieved an all-time speed record for a steam engine of 126 mph (203 km/h) between Grantham and Peterborough. It could not have been done without *'streamlining'* or a competent driver and fireman. Number 4468 was named "Mallard", as is the BASIC on your Amstrad PCW. Mallard BASIC has also been designed to be very fast and, when you program with it, you are its driver and fireman. The *'streamlining',* I hope, will be found in the following pages.

1.1 Purpose

This book is not intended to be a user manual for Mallard BASIC, or even a

tutorial. It is intended for Amstrad PCW users who have experimented with Mallard BASIC and attempted to write some serious, even if simple, programs of their own, but would like to move on to bigger and better ones. If you have not reached this stage, I suggest you study some tutorial material first. Your first choice will, obviously, be the official manual, and a book by Ian Sinclair has been generally recommended. For those preferring to learn at the keyboard, 'David Wilson Computronics' have produced a comprehensive disc entitled "BASIC Tutorial".

As I develop my approach, I will not be afraid to PEEK and POKE about in memory from time to time, but I will not assume that you have any knowledge of machine code programming -- or even that you want to descend to that level! Very occasionally, a machine code sequence will be included in an example routine, but you will not need to understand it.

My central theme will be programming style, though some might think that I am the last person on Earth who ought to be writing about that (ROCHE> Including me, since Geoffrey Childs *NEVER* indents his programs!). If someone wanted to compliment my programming, I would accept -- with typical modesty -- such words as "knowledgeable", "ingenious", "economical", and even "eccentric". But not even my best friends -- nor, for that matter, my worst enemies -- would call me a stylish programmer. However, what we are considering is your style, not mine. There are certain aspects of programming that I consider to be bad programming rather than bad style and, in these cases, I shall be quite dictatorial, telling you a few things that you must not do. Apart from that, I hope that you will accept most of what follows as suggestions, rather than instructions. Very often, the question is one of speed of operation, or the comfort of the user, which is what good footplate crew have always been concerned about.

Any form of programming is a skill and, however individual a skill may be, the best practitioners will try to learn from others. If you think that you always program perfectly, who am I to disagree? You may find some of my ideas and methods reprehensible -- again, I will not disagree. What I can claim is experience -- the fact that I wrote "Lightning BASIC" proves that -- and that most of my programs do work, eventually.

1.2 BASIC extensions

Mallard is a powerful and fast implementation of BASIC. It is fair to say that the more I have used it, the more I like it. I did find it rather strange at first, after using Microsoft and Xtal BASICs on other machines. However, any new system needs patience before it becomes familiar.

Having said this, there is no question that Mallard is extremely clumsy at doing a few simple things, the most obvious of which is clearing the screen! (ROCHE> BASIC, and CP/M, were first used on computers which had no screen, only an ASR-33 Teletype through which all interaction with BASIC/CP/M was done. So, no screen = no CLS. Anyway, who needs a CLS when pressing [RETURN] the appropriate number of lines is enough to clear the screen?) There is a method behind Mallard's madness. It is not that Locomotive were incapable of devising a CLS which would have done the job. By using control codes, Mallard files become portable when saved in ASCII form, and can be used on computers other than the Amstrad PCW. (ROCHE> I have been using Mallard-86 BASIC for MS-DOS for 10 years, now, and have *NEVER* used ASC files, only BAS files when transfering my old Amstrad PCW programs...)

Your masterpiece (and mine) are probably only intended to run on the Amstrad PCW. (ROCHE> I started as a COBOL programmer on IBM Mainframes. Then ventured

into minicomputers. Then finally used micro-computers (which went from 8 bits to 16 bits, then 32 bits!). For me, Mallard BASIC is a high-level way of porting my programs: they run under CP/M Plus, MS-DOS, and in a "DOS Box" under Windows.) It makes sense to make a few extensions to Mallard BASIC, so that the simple jobs can be done quickly.

Much of my programming on the Amstrad PCW has been devoted to writing various extensions to Mallard BASIC, so that not only does it do the simple jobs easily, but becomes more powerful generally.

I will admit that I do most of my BASIC programming using "Lightning BASIC" but, then, having written it myself, I have not had to pay for it! You would not like it if, having raided your piggybank to buy this book, I told you to send 24.95 Pounds to CP Software to buy Lightning BASIC. (But, of course, it is worth every penny...)

DWBAS is a much shorter extension (also written by me) -- this can be RUN and LISTed from the program disc. You may have this for nothing. I do not like typing in programs with lots of figures, that is why I have provided DWBAS (and the other programs in this book) on disc. If you have the Amstrad PCW9512, use DW139 instead, this is also on the program disc. This is an amended version that runs with Mallard-80 BASIC Version 1.39. (The Amstrad PCW8256 and PCW8512 use Mallard-80 BASIC Version 1.29.) The explanation of these extensions is given in Appendix 3. The programs in the text do not rely on DWBAS, but those in Appendices 6 and 7 do need DWBAS loaded first.

One important feature of DWBAS (and Lightning BASIC) is a multiple POKE. The command POKE 50000,2,3,4 will POKE 2 into 50000, 3 into 50001, and 4 into 50002. I find this so important that I shall INSIST that you have a multiple POKE when we come to Part 2 of the book. If you refuse to use DWBAS, Appendix 4 gives a briefer program which will install this feature. Naturally, you will not need this if you use DWBAS.

1.3 Your own style

Part 1 of this book is intended to help you develop your own style of programming. There are usually many ways of achieving the desired result in a program. Often, the programmer has to make a choice between economies in memory and speed on the one hand, and attractive presentation and easily readable programs on the other. Programming style generally develops subconciously, but it is not a bad idea to have an introspective examination of one's programs from time to time. In this section, we have a preliminary look at some of the questions that you might consider.

Writing computer programs has parallels with writing English. It is easy to identify bad style, but good style is not so clear cut or uncontroversial. In most of our literary works, we would try to avoid spelling sausages as 'sosiges', but it would not matter if we put 'sosiges' on our weekly shopping list! In an idle quarter of an hour, last week, I used a computer program to work out the Daily Telegraph Brain Teaser. It does not matter in the slightest how badly the program was written, provided it works!

Another parallel with English is that the style of a computer program depends on the intended user. You would not write a love letter in the same style as a technical report, and it is unlikely that you would write a "Space Invaders" game in the same style as an accounts system.

Clearly, there are some general rules that all programmers should follow, and they can be summarised in one sentence. Write your program in a style that

will be suitable for the people you expect to use it. The balance between lengthy prompts, excessive input checking and confirmation on the one hand, and a fast flowing program on the other, must be made with the expected experience of the average user in mind. Decorative screens and musical jingles (not so easy on the Amstrad PCW) must be balanced between pleasure and irritation potential! It is sometimes hard to remember that the person using your program, probably a philistine, will not be admiring your beautiful programming skills, but using your program as a means to an end.

Having made these general didactic points, it is time to emphasise the main point I want to make in this section: **DEVELOP YOUR OWN STYLE.**

What is right for you, may not be right for me, and vice versa. If you find that you can write programs that work well without bothering about structure and subroutines, then write that way -- at least until you find that it might be wiser to modify your views. If you find that planning on paper and having a complete knowledge of exactly what your program will do is essential -- do it that way. Academic writers about computer programming talk some sense, but I think that they talk a lot of rubbish, too!

I am going to finish this section by asking a number of questions. They are question which you may feel strongly about, and your answers may be opposite to mine. You may feel that the answer is sometimes 'yes' and sometimes 'no'. I hope that some will be questions that you have not even considered. In a short section like this, it is impossible to cover all the questions one might ask about style. Provided you are thinking about these propositions, it matters not whether you become more pig-headed in your views, or more willing to modify them.

Should variable names be long: "age.of.employee" or short: "a"?

Do you religiously, occasionally, or never put in REMs?

Do you leave blank lines, such as "200 :", to make your listings look pretty?

Should documentation be non-existent, in REMs, on-screen, or in a separate file or booklet?

Do you use "Press any key..." or a specific "Press SPACE..."?

Do you always use INPUT, even for one-letter replies?

Do you believe in right-justifying or centring screen printing?

Do you make sure the user sees your name on any programs you write?

Do you have any little habits by which a program you wrote would be identified as likely to be yours?

Should you avoid POKEs or CALLs that make your programs machine or model specific?

Do you use reverse video often, occasionally, or never?

Do you write your subroutines before the rest of your program, or whenever they crop up?

Do you think that you make too little or too much use of subroutines?

Do you like or dislike programs where the menu choice is made by cursor control (or a mouse), rather than by pressing a key?

Is it better to keep most program lines to a single statement?

What are your feelings about GOTO? That it should be never used. That is has occasional uses. That you do not care what the academics think, and it is just about your favorite BASIC word!

Do you use integer variables whenever possible, occasionally, or never?

Do you RENUM when a program is complete, or do you leave numbers as they stand, so that you may be able to recall them better later?

Do you ever use DEF FN, apart from escape sequences? Or PEEK, POKE, CALL, and VARPTR?

Do you use ON ERROR throughout a program, for occasional bits where a specific error could be expected, or never?

Do you protect your programs? If not, do you make your listings as easy (or as difficult!) as possible for the user to follow?

Which type of files do you use most often: sequential, random, or Jetsam?

Do you think Jetsam is the best thing about Mallard BASIC, could be used on occasions, or is too complex to try to understand?

What is your attitude to a finished program that works? "I am not going to touch it again", or "I will take every opportunity to gild the lily"?

Do you consider it bad programming if the screen scrolls during operation?

Some of the questions will not be discussed any further, but there are other questions on which I do have definite views, and which we shall think about in more detail. Where this happens, I shall be happy to preach to heathens, doubters, or the converted!

1.4 Planning

If you are a competent pianist, it is possible to sit down at a piano, put your hands on the keys, have no idea what you are going to play, and improvise happily and tunefully. You cannot do this with a computer. All programs do need an element of planning. At the very minimum, you must know what the program aims to achieve.

Flowcharts were once one of my pet hates. Perhaps I do not read the right books these days, but thankfully we do not hear so much about them, now. The idea of a flowchart is to put into words the various stages of a program, and use symbols and arrows to define its possible courses. For a simple program, this seems a waste of effort and, for a complex program, it is difficult to imagine the size of paper that you would need! I would find drawing the flowchart much harder than writing the program -- which is not the purpose of the exercise! I know some people do find them helpful. If you like them, do not let me stop you using them. You are probably more methodical than I.

Some methods of planning are needed. If I am writing a Mallard BASIC program, I usually code at the computer. Occasionally, I may use the back of an envelope for a section of code that looks tricky. I may write a very rough menu outline for a long program on paper. If I am writing a machine code routine, I usually write it in assembler on an envelope, and translate it into

code at the machine. Not recommended, unless you also are in an advanced state of madness! The stationers do not make a fortune out of me. I appreciate that most people would disagre with so little paper planning. Ask yourself if more time spent on paper planning would make the eventual writing easier or more efficient. Only you can decide.

You may have kindly thoughts like: "Yes, but you are such a genius that you do not need to plan." Or unkind ones like: "No wonder your programs turn out so badly." Wrong on both counts. I do plan: I probably spend more time on it that you do. If an idea for a substantial program comes to mind, I do not sit down at the computer to write it. I carry the idea, maybe for months, and think about it. I do nothing at the computer until I know that I have sufficient time to write the program with a minimum of breaks. By the time I start, I have a clear outline plan in my mind. While I am writing the program, I will be thinking about it and the little extras that can be introduced to improve on the outline. When the program is written, I hope to have nothing more to do. Usually, this is optimistic because it is often someone else who finds the bugs.

'Structured programming' is a computer buzz-word. To some people, it is closer to a religious utterance. A structured program is contained in modules. On a computer such as the BBC, these modules are called PROCEDURES. On the Amstrad PCWs, we have to make do with subroutines, which are nearly the same thing! The procedures are written, and all the main program does is to call them in the correct sequence. My advice is not to make structured programming into a religion, but make good use of subroutines, so that your programs have a semblance of structure -- even if served with a helping of spaghetti!

If I use any formal technique, it is probably closer to the 'top-down' method than anything else. I usually a program at line 5000, where I put the general purpose subroutines -- some will have been pinched from other programs. I may write some more specific subroutines, such as filing ones, starting at line 6000, which may only need to be chained to certain options on the menu. I use 7000+ for error traps. The main program is divided into sections, starting at line 1000, 2000, etc. I write the program a section at a time, when the subroutines have been written. Sometimes, I also use lines above 8000. Eventually, the main menu and housekeeping goes at the beginning of the program.

There is no magic in this particular numbering system, but there is plenty of point in establishing a pattern for yourself and sticking to it, if only for the fact that anything done consistently is more easily remembered.

However well you plan, you will probably spend at least as much time on testing and debugging your program as on writing it, but we will leave these little treasures until later.

Chapter 2: Everyday problems

In this chapter, we are going to look at some Mallard BASIC commands that most of you will have used. Familiarity can breed contempt, and good or bad programming is often more evident in the simple everyday operations that most programmers will be using frequently. The topics may seem disjointed, but the common theme is that we should think about the routines that we know we can program, as well as those that we know will offer a new challenge.

2.1 Escape sequences

One of the first things most of use notice about Mallard BASIC is that you cannot clear the screen with a simple CLS (or something similar). I have come to respect Mallard in many ways, but I still think that this is ridiculous, even though the reason is that Mallard BASIC is meant to be portable.

If you are using DWBAS, you will be happy not having to use escape sequences like

```
PRINT CHR$ (27) + "E" + CHR$ (27) + "H"
```

any more.

This will be the case if you only want to program for yourself. Some of you will wish to write programs for magazines, for example, and you cannot rely on readers having DWBAS. Life will be simpler with DWBAS, but there will still be times when you have to use the conventional sequences.

So, let me get rid of a couple of bees in my bonnet. I hate the people who write

```
esc$ = CHR$ (27)
```

or even worse:

```
escape$ = CHR$ (27)
```

The idea is, presumably, to shorten things, so WHY NOT

```
e$ = CHR$ (27) ?
```

It also irritates me greatly to see the PRINT AT sequence used without a DEF FN, and the horrible CHR$(32+r) appearing all over a program. If you must use escape sequences, my advice is to write a little subprogram including all the regular ones, and start each new program with it. Unused escape sequences can be deleted when the program is completed. (End of burst of bad temper!)

It is worth studying pages 140-141 (in my copy) of Manual 1 to note the rarer escape sequences. Some knowledge of the ones to use with the printer is also advisable. The effect of escape sequences is, of course, quite different in PRINT statements from that in LPRINT statements.

2.2 PRINT and LPRINT

Many programs will contain sections that offer the user a choice of printing on the screen, or sending the same text to the printer for hard copy. Quite often, these program sections are fairly long, and it seems to be wasted effort when the whole thing has to be written out again with all the PRINTs changed to LPRINTs. Of course, it is possible to save some of the burden by suitable use of AUTO and RENUM, but there is a much easier way.

The idea is very simple. We temporarily send all PRINT commands to the address for LPRINT commands. When we have done the LPRINTing, we change the address back to normal. POKE 18527,90 changes PRINT to LPRINT. POKE 18527,100 reverts to normal. On the Amstrad PCW9512, a different version of Mallard-80 BASIC (Version 1.39, as opposed to Version 1.29) is provided, and the addresses are different. The corresponding POKEs are POKE 18591,0 and POKE 18591,10. If you are writing for users who might be in either version of Mallard BASIC, you will have to test. The address I use (out of many possibles) is 5103. If PEEK

(5103) = 197, then you are using Mallard-80 BASIC Version 1.39.

There is also a simple POKE to make the PRINTing go to the screen and then to the printer. POKE 8792,205 in Version 1.29 and POKE 29161,205 in Ver. 1.39. This make a machine code jump into a call (like a GOSUB): when it does the PRINT routine it returns, and where has it got to? The LPRINT routine! Making this change is rather more comprehensive than the previous POKE. Not only does PRINT get echoed, but also everything else that goes to the screen, including the "Ok" prompt! To go back to normal, you POKE 8792,195 with Ver. 1.29, and POKE 29161,195 with Ver. 1.39.

An omission in Mallard is that there is no official way to TYPE or DISPLAY a file to hard copy. Byt try this in Mallard 1.29:

```
POKE 8793, 234 : DISPLAY "FILENAME.TYP" : POKE 8793, 239
```

Change 8793 to 29162 with Ver. 1.39. If you run an Amstrad PCW9512, you will probably be using a version of CP/M Plus named 'J21CPM3.EMS' (found on your Amstrad master disc). In this case, you should replace the 234 and 239 mentioned above with 246 and 251, respectively.

2.3 Randomizing and the clock

You may think that this is an odd pair to take together, but I shall not discuss the Amstrad PCW clock at any other point. Many moons ago, I suspected that a clock must exist, and found it without too much effort. When I published this result, it was received with interest... but I don't think that the discovery should have been beyond any competent Mallard BASIC programmer prepared to PEEK and POKE around a little. Everybody knows about the Amstrad PCW clock these days -- but just in case...

The clock is contained in the bytes 64502-64504, in the CP/M Plus area. When the computer is switched on, they are set to 0, 0, and -- guess what -- 0! They represent hours, minutes, and seconds. The clock is constantly updated through the interrupt system, and PEEKing these bytes will give you the time since you switched on the computer. Er, well, it will if you think in hexadecimal! If PEEk (64504) gives 37, it actually means 25 seconds, as 37 = 2 * 16 + 5. I can sense some of you writhing. Writhe no more.

For reading the clock: DEF FN a (x) = INT (x / 16) * 10 + x MOD 16. For setting the clock: DEF FN b (x) = INT (x / 10) * 16 + x MOD 10. Then, to read: h = FN a (PEEK (64502)) : m = FN a (PEEK (64503)) : s = FN a (PEEK (64504)). To set: POKE 64502, FN b (h) : POKE 64503, FN b (m) : POKE 64504, FN b (s). It should be obvious what the variables h, m, and s represent...

You can amuse yourself by finding other ways of doing this, by using HEX$.

Mallard is not one of the easiest BASICs to use when you need a sequence of random numbers. Perhaps random sequences are used mainly for games, but they can also serve serious purposes, such as statistical simulations. If you simply put RND in your program at various points, the same sequence of random numbers will appear every time. Each time you play your murder adventure... the butler did it! So, to be clever, you put a line like RANDOMIZE 23 at the start of your program. This time, the vicar dunnit. But the vicar will go on doing it every time.

What is needed is a different sequence of random numbers each time. This is where the clock can be useful. You 'seed' the random number generator with a number that is itself (approximately) random:

```
        RANDOMIZE PEEK (64504)
```

The sequence now will be one of sixty possibles, depending on the exact number of seconds showing on the clock when the game starts. For a serious application, 60 sequences may be insufficient. You could use: RANDOMIZE PEEK (64504) + 256 * PEEK (64503).

An alternative approach is to use the same sequence of random numbers, but enter the sequence at a different point. Quite easy...

```
        20 PRINT "Press any key to start" : z$ = INKEY$ : z$ = "" : WHILE z$ =
"" : r = RND : z$ = INKEY$ : WEND
```

(ROCHE> Indented, this gives:

```
        20 PRINT "Press any key to start"
        30 z$ = INKEY$
        40 z$ = ""
        50 WHILE z$ = ""
        60       r = RND
        70       z$ = INKEY$
        80 WEND
)
```

For a game, either approach is adequate. For a more serious application, I would recommend using BOTH.

2.4 Only 31K?

>From time to time, a letter is published in one of the Amstrad PCW magazines, bemoaning the fact that, while the writer has a 512K computer, there is only 31K left for programs. I would have a little bet that none of those writers has, actually, used more than 5K in a Mallard BASIC program. When I started programming, I was PLEASED when my programs were so long that this much memory had been used!

The reason for the 31K limit is that, in any one microsecond, the Amstrad PCW can only access 64K of memory. A single machine code command can replace part of this 64K with some of the other memory in the computer. The principle of Mallard BASIC is to have the program and the interpreter in the blocks of memory that are normally the ones accessed. There is another 80K of memory that Mallard BASIC (or CP/M Plus) uses to operate the keyboard, screen, discs, printer, and other things. This memory is not entirely sacrosanct, but most Mallard BASIC programmers will be quite happy to let the interpreter use this memory 'behind closed doors'. The rest of the memory is the M disc. Don't worry if you think this is too brief to be comprehensible. We shall go into the memory mapping in much greater detail in Part 2.

When you are told that you have 31597 bytes left, this refers to the memory being currently accessed. If you include the M disc, you have 399K of memory available for your program on the Amstrad PCW8512 or PCW9512! If that is not enough, you can make use of as many physical discs as you like in Drives A and B. You are not short of memory -- you just have to manoeuvre it. There are sufficient commands in Mallard BASIC to make this perfectly easy without having to understand how the hidden memory works.

Let us suppose that you write a program that has a main menu with four options. Each of these options will take about 20K, so that you have a program

of nearly three times the official memory size. You write a little program called MAIN. This gives you the main and some CHAIN MERGE commands. The line numbering might be below 1000. You then write your four programs SECTION1, SECTION2, etc, each starting at line 1000. After your main menu and a "Press key 1-4", you can go to any section in a single line such as:

```
CHAIN MERGE "m:" + "SECTION" + CHR$ (z + 48), 1000, ALL, DELETE-999
```

That's all you need to do to treble your memory!

You will notice that the section files are on the M disc. This is not essential, but usually it will make for smoother operation if the extra programs and any other required files on the program disc are transferred to the M disc before the main menu appears for the first time (Section 4.3 describes how to do this). For those of you unfamiliar with the CHAIN MERGE command -- I must admit that I usually have to look it up, to check on the order of the arguments -- the first argument is the file name, the second the line number for restarting, the third says retain ALL variables, and the last gives the lines to delete from the program already in memory. An alternative to ALL is to omit this and define some variables (the date, for instance) as COMMON before the CHAIN MERGE.

A long program will probably contain sequential, random, or Jetsam files. This is another memory saving device, since only the part of the file that immediately concerns you will be in accessible memory. Such files can be used in unconventional ways, such as containing a machine code routine, as well as the more normal database material.

Some programs may need pages of instructions. While these can be produced in the form of PRINT statements in your program, you will probably find that to write them in LocoScript and convert to page image files saves both labour and memory. You can then use TYPE or DISPLAY in your program when needed. For more advanced programmers, it is also possible to use the M disc as extra memory.

31K? No problem!

2.5 Integer variables, etc

Do you use integer variables whenever you can? Probably you should do, and probably you don't -- like me. It isn't vitally important, but integer variables make your program run slightly more quickly and, debatably, use less space. You don't have to put % after them, you can DEFINT a-f for example. The former method makes for typing mistakes, the latter for occasional awkward debugging problems. It is sensible to use them in random or keyed files where you have to make a decision whether to use MKS$ or MKUK$. It is essential to use them in some machine code routines (including GSX, if you brave this territory). It is advisable to make a substantial array integer-variable, if possible. DIM a (100) commandeers 400 bytes. DIM a% (100) takes 200 bytes.

Otherwise, it is largely a matter of style. On the first computer I used which allowed integer variables, they actually made the program slower and used more space! They have also been responsible for the worst bug I think I have ever made -- no, I am too embarrassed to tell you about it! Let us just say that YOU should use them as often as possible if you want ten out of ten for style.

Double precision variables are probably not much used. The Mallard implementation is not the jewel in Locomotive Software's crown. It is not applicable to functions such as SQR or SIN, and this is probably where it is most likely to be needed -- astronomical calculations, for instance. It is

necessary to remember that double precision calculations will only be as accurate as the least accurate part of the expression. a# = 2.13765894567# * EXP (4#) will produce lots of decimal places, of which about the last eight are meaningless. It is worth mentioning that a FOR-NEXT loop will not work with double precision variables. If you do this inadvertently, you may be scratching your head for a long time!

It is possibly worth a mention that you can obtain double precision values of functions, but you have to write your own routines for them:

EXAMPLE: Return b# as the square root of a#:

```
        b# = 1# : c# = 0 : WHILE c# <> b# : c# = b# : b# = (c# + a# / c#) / 2#
: WEND : RETURN
```

(ROCHE> Indented, this gives:

```
        100 b# = 1#
        110 c# = 0
        120 WHILE c# <> b#
        130       c# = b#
        140       b# = (c# + a# / c#) / 2#
        150 WEND
        160 RETURN
)
```

2.6 Loops

I been told, you been told, all God's children have been told: never jump into a loop, never jump out of a loop. Obviously, you can never jump into a loop, because, if you encounter a NEXT without a FOR, the computer does not know where to go, and cannot do anything else but produce an error. But, if you read the manual carefully, you will see that it says that it is permissible to jump out of a FOR-NEXT or WHILE-WEND loop. I read that too, and I admit that I do occasionally jump out of a loop. But I still think that it is bad practice.

```
        10 FOR n = 1 TO 20
        ...
        50     IF x > 1000 THEN 600
        ...
        100 NEXT
```

On the Amstrad PCW, you will get away with this. On most other computers, you will get away with it for a time but, if you do it too often, you will run into an "Out of Memory" error, or something similar. Consider:

```
        50     IF x > 1000 THEN n = 30 : GOTO 100
        ...
        101 IF n > 21 THEN 600
```

This completes the loop, and will work on any computer, even if it is not too elegant. With many computers, you could write:

```
        50 IF x > 1000 THEN n = 30 : NEXT : GOTO 600
```

However, Mallard BASIC insists on only one NEXT applying to each FOR, and so this is not accepted. I suppose this is an exchange for the nearly safe jump from a loop.

Am I making an unreasonable mountain out of a molehill? Can I crash by jumping out of a loop? Look at this spaghetti horror:

```
C2P1
100 FOR a = 1 TO 10
110 FOR b = 1 TO 10
120 PRINT a, b
130 NEXT
140 GOTO 170
149 FOR c = 1 TO 5
150 NEXT
160 NEXT : END
170 GOTO 160
```

(ROCHE> Indented, this gives:

```
100 FOR a = 1 TO 10
110     FOR b = 1 TO 10
120         PRINT a, b
130     NEXT
140     GOTO 170
149     FOR c = 1 TO 5  ' Never executed
150     NEXT
160 NEXT
165 END
170 GOTO 160
```

Notice that none of the NEXTs indicates which FOR it is ending...)

Try it and it will 'work'. Now, delete line 149. You crash DESPITE THE FACT THAT LINES 149 AND 150 ARE NEVER EXECUTED. I grant that you have to be able to write exquisitely badly to produce this standard of program, but it does bring home the point that it is best to stick to the rules.

Having used BASICs without a WHILE-WEND facility, I know that it is always possible to use FOR-NEXT all the time. Some people appear to think that the WHILE-WEND is the more elegant choice when possible. My own practice is to use the type of loop that comes naturally. Some people prefer to include the variable name after a NEXT. Maybe it does make programs easier to follow, but it is not necessary, and slows down the program marginally.

2.7 Approximations

A hidden trap in any BASIC that allows Floating-Point numbers (decimals to you and me) is that some numbers that APPEAR to be equal are not EXACTLY equal. Consider:

```
FOR n = 1 TO 2 STEP 0.1
```

It may seem obvious that the loop will be executed 11 times, ending with n being 2. This may indeed happen, but it is also possible that the loop will be only executed 10 times. A computer works in binary numbers, and 0.1 will not be an exact binary decimal. When 0.1 is added to 1 ten times, the result may be exactly 2, just over 2, or just under 2. If you want to ensure that the loop does execute 11 times, you should write:

```
FOR n = 1 TO 2.00001 STEP 0.1
```

In the same way, if you test two numbers for equality, or test a number to see

whether it is exactly a whole number, you should test for approximate equality, rather than exact equality:

```
IF a = INT (a) THEN PRINT "Result is an integer."
```

should be replaced by:

```
IF ABS (a - ROUND (a)) < 0.00001 THEN PRINT "Result is an integer."
```

Perhaps these points have little to do with your BASIC style, but a program that 'nearly always works' can never be called a stylish one!

2.8 AND, OR, XOR

This section can be omitted if you dislike Maths, but to some of you it may be helpful for an understanding of the following Section (2.9). AND and OR sound like everyday words, XOR doesn't! To the computer, AND and OR mean something very different from what these words mean to you and me. Yet, when AND and OR are used in an IF-THEN, the computer's interpretation will have exactly the same result as if it had been thinking in everyday English!

You have probably been told that the computer thinks in terms of binary numbers. Binary numbers are explained in the Graphics chapter, but don't worry about them for the moment. Here are two binary numbers:

```
 73 = 01001001
200 = 11001000
```

The computer can 'operate' on two binary numbers with AND, OR, and XOR, provided that they are no more complicated than integer variables. The word 'operate' means: take the two numbers and produce a third number from them. + is an operator, so is - or *.

AND, OR, and XOR work on the numbers in their BINARY form, looking at each column of the 0s and 1s in turn. AND gives a result of 1 only if BOTH rows in the column are 1. OR gives a result of 1 if EITHER row is 1, or both rows are 1. XOR gives a result of 1 if one or other BUT NOT both the rows are 1. In all other cases, the result of the operation is 0. So:

```
 73 = 01001001
200 = 11001000

73 AND 200 = 01001000 =  72
73  OR 200 = 11001001 = 201
73 XOR 200 = 10000001 = 129
```

What do you notice about the three answers? Also, you might be persuaded to try XORing 73 and 200 with the XOR answer 129. Interesting? (Note for boffins only: 0, 73, 129, and 200 form a group with respect to XOR.)

In the next Section, when we are talking about -1, we are actually talking about the integer variable -1, which the computer represents as:

```
1111111111111111
```

2.9 IF...

In my opinion, more bad programming is caused by excessive use of IF than by the use of GOTO. I will have my beef about that in the section about ON-GOTO. This is not to say that you should always try to avoid IF statements. They are frequently the simplest and best way to do things.

Some stylish though often trivial effects can be used with IF, provided that the programmer thinks the way the computer does. The program line x=4 means that the variable x becomes 4. The line IF x=4... means something quite different. x=4... is evaluated as a NUMBER: -1 if the statement is true, and 0 if the statement is false. IF x=4 AND y=5... contains a mathematical operation on two NUMBERS, resulting in -1 or 0.

IF p<>0... can be replaced simply by IF p. While the computer always evaluates truth as -1, it accepts as truth any value other than 0. For instance, IF x=4 AND y=5... will be evaluated as -1, provided both conditions are true. IF (x=4)*(y=5)... would be evaluated as +1, but it would still have exactly the same effect. In the same way, '+' can replace OR, and usually '-' can replace XOR.

Thus, IF p<>0 AND q<>0... can be shortened to IF p*q. When you use a condition that mixes AND with OR, it is often awkward to be quite sure what will happen in all cases. Using * and + may clarify things to the programmer.

IF x=0... can sometimes be replaced by IF NOT x. You may say that this is longer. Quite! (Although it is shorter in program storage.) Yet, I cannot recall anyone who writes WHILE EOF (1) = 0. Admit it! You write WHILE NOT EOF (1).

Occasional programming tricks can be performed by using the words AND, OR, and XOR in their mathematical sense. One of the neatest is toggling a flag from 0 to 1.

```
IF x=1 THEN x=0 : ELSE x=1...
```

can be replaced by

```
x = x XOR 1
```

Chapter 3: Input techniques

INPUT techniques are probably the most boring programming tasks that you will have to do. They are also going to be the part of the program by which you stand or fall when you let somebody else use your program. This is not the most interesting chapter in the book. It may be the most important if you hope that your programs will be sold, given away, lent, or pirated.

3.1 INPUT, INKEY$, etc

The clarity of your prompts is not something that we can demonstrate here -- this is not an English grammar textbook -- and the depth of input checking will depend entirely on whether you rate the user at idiot or moron level! What we can do is show you efficient ways of programming for the input that you need.

Some programs will run with keypress responses from the user. INKEY$ (or INPUT$) will be sufficient. Others require sentences, words, or numbers to be entered. These programs will need INPUT. Many programs require a mixture of

keypresses and longer INPUTs. For these, should the programmer use INPUT for all responses, or should the program use INKEY$ when possible? It is a question that has not been entirely settled. The case for using INPUTs only is based on consistency. Up to a point, I agree. A good program is consistent. If you use "Press Space Bar to Continue" at one point, it is poor programming to use "Press RETURN to continue" at another. If you use INPUT and INKEY$ in the same program, the user can be muddled about whether a press of [RETURN] is needed.

On the other hand, it is also good programming practice to ask the user to do as little as possible. It seems to be wrong to ask for a [RETURN] press when it is not needed. My own practice is to use INPUT only when needed, and INKEY$ otherwise. I do make a point of using the verb ENTER for INPUT, and PRESS for INKEY$ (other words will do equally well, provided that the use is consistent throughout the program). Certainly, I get mildly annoyed with a program that uses INPUT when INKEY$ would do.

Another point to consider is whether you should use the facility to INPUT a numeric variable. For example, you might use:

```
60 INPUT "Enter cost price in pence: "; cp : IF cp <= 0 THEN 60
```

You could also use:

```
60 INPUT "Enter cost price in pence: "; cp$ : cp = VAL (cp$) : IF cp <= 0 THEN 60
```

The latter seems preferable, as you avoid the possibility of the message "?Redo from start" which can muddle your palooka, and an entry of 60p is accepted correctly. However, I don't have strong feelings one way or the other, as a great deal depends on the expected skill of the user. A good stylist might well add an error message and a beep in these lines, but that is not the point, here.

We have used INPUT and INKEY$, since most programmers seem to use these methods exclusively. In the following section, we shall discuss alternatives to INKEY$. There is also an alternative to INPUT, LINE INPUT. The manual does not make it very clear why anybody should ever want to use LINE INPUT, but the main point is that you can include commas (",") in the input (an address, for instance).

3.2 INKEY$ or...

All BASICs have a method for scanning the keyboard and putting the result in a variable -- usually, but not always, a string variable. The word used can be INKEY$, INPUT$, GET, or KBD. (GET means something different in Mallard BASIC, and KBD means nothing at all.) The result of such a scan will also depend on how the interpreter is programmed to respond to the scan. To know precisely what is happening, you have to know the answers to two questions:

1) "Is it a RETROSPECTIVE or INSTANT scan?"

2) "Will it wait (STATIONARY) or continue (MOVING) if no keypress is found?"

Every time that a line is executed in Mallard BASIC, the interpreter is programmed to look at the keyboard. If a key is pressed, it will be noted. There is a mechanism to prevent this happening if the last effective keypress is still being held down. (The locations for these signals will be found in

Chapter 14.)

When an INKEY$ or an INPUT$ is met later in the program, a RETROSPECTIVE keypress (one that has previously been signalled) will be taken as the value required. At this point, z$ = INKEY$ and z$ = INPUT$ (1) differ. INKEY$ takes a last look at the keyboard and carries on with the program (MOVING), leaving z$ = "" if no response is forthcoming, whereas INPUT$ (1) rescans until a positive response is achieved (STATIONARY).

Since the usual response required is STATIONARY, INKEY$ is often used when there is a better way to program. The easiest way to turn INKEY$ into a STATIONARY GET is very rarely used! The manuel does not use it, and the natural tendency of the user is to believe that, if the manual demonstrates one method, there cannot be a simpler way. Usually right, but not this time! Instead of:

```
z$ = "" : WHILE z$ = "" : z$ = INKEY$ : WEND
```

(ROCHE> Indented, this gives:

```
100 z$ = ""
110 WHILE z$ = ""
120       z$ = INKEY$
130 WEND
```

Personally, I use WHILE INKEY$ = "" : WEND...)

just use:

```
z$ = INPUT$ (1)
```

INPUT$ (1) will pick up a retrospective keypress, so that, if you don't want this to happen, you can precede it with z$ = INKEY$.

If you are satisfied with ANY keypress, an alternative method for the stationary get is to define v = 1018 and use CALL v every time you request a keypress. This uses a routine in the Mallard BASIC interpreter.

Alternatively, you can use the CP/M Plus routine, which has the difference that a restrospective keypress is not picked up. You can do this by defining v = 65062 (65074 if you are using the Amstrad PCW9512), and again CALL v.

>From this point on, I shall assume that I have converted you to INPUT$! If I have not done so, all comments about INPUT$ will also apply to INKEY$.

3.3 Filename checking

I will readily admit that checking INPUT is not one of my favourite programming jobs. A frequent example is the INPUT of a filename. You don't mind if the user calls the database JEAN or SALLY.ANN. But the computer will mind if it is called ESMERALDA or Mrs J.A.SMITH. I will admit to writing extensive checks to find out if a filename entered was acceptable, but this was largely wasted effort. It brings in an important principle. If the computer will do it for you, don't do it yourself.

With an ON ERROR check, I improve things by using FIND$, but this did mean a short delay and a user-unidentifiable whirring. I hit on the idea of altering the KILL routine by changing a CALL to a JP (ROCHE> Z-80 mnemonics...), so that the routine would error check without erasing the file. Now, I think that

the simplest way is to use FIND$, but to check on the M disc, which is much quicker, and silent. This sort of thing:

```
        C3P1
        200 INPUT "Enter filename xxxxxxxxxxx: ", f$
        210 ON ERROR GOTO 5000
        220 g$ = FIND$ ("m:" + f$) : ON ERROR GOTO 0
        230 IF f$ = "" THEN 5010
        240 REM Etc.
        ...
        5000 RESUME 5010
        5010  PRINT CHR$ (7) ; "Why don't you do what you were told?"  :  GOTO
200
```

xxxxxxxxxxx represents 3 pints of Australian beer, or however you want to prompt the user about filename choice.

3.4 Date checking

I don't know what it is about dates, but most programmers seem to have gone to great pains to devise a routine to input dates, with numerous checks for accuracy. So, I am not going to try to slay your own particular sacred cow at this point. If you have devised such a routine, I am sure that it is better than anything I can show you!

I am afraid I simply ask for day, month, and year as separate inputs. It is really easier all round than asking for input in a six figure form, say 070289 for 7 February 1989. Section 7.5 gives a routine for friendly month input.

There are a great many programs that do not need the date, but still request it and put it on every print out. Some users like this, but others will be irritated. An answer to this is to use a 'hidden menu', so that any keypress-routine checks for certain letters which temporarily interrupt the program. Using [ALT]-D for date entry will not interfere with anything else, and a user who likes to date the documents can press this at any time. Those who don't want to do so are not obliged to do so. Another friendly little trick on the hidden menu is to allow [ALT]-I to invert the screen to black characters on green background. If you use this as a toggle, with the variable fl signalling the current condition, fl = fl XOR 1... is a convenient way to operate the toggle.

3.5 Checking input

```
        C3P2
        10 INPUT "Please enter your name: ", a$
        20  PRINT : PRINT "Is your name "; a$; "? Press Y for YES, and  N  for
NO."
        30 z$ = UPPER$ (INPUT$ (1)) : IF z$ = "N" THEN 10 : ELSE IF z$ <>  "Y"
THEN 30
        40 PRINT : PRINT "Hello, "; a$; "!"
```

(ROCHE> Rearranged, this gives:

```
        10 INPUT "Please enter your name: ", a$
        20 PRINT
        30 PRINT "Is your name " a$ " (Y/N) ?"
        40 z$ = UPPER$ (INPUT$ (1))
```

```
        50 IF z$ = "N" THEN 10 : ELSE IF z$ <> "Y" THEN 50
        60 PRINT
        70 PRINT "Hello, " a$ "!"
)
```

Grrrrrh. Yes, I know that we have to do this for some programs. We have also run some programs that do this to us, and drive us to distraction! The balance between over-checking and under-checking is a delicate one, and only the programmer can decide what is right.

Let us suppose that we are writing a program which asks for entries of a share purchase. Perhaps 10 items of information will be needed. You or I could easily make a mistake in entering this much, so some element of checking is needed. Checking ech item as entered is one solution. Giving an option to go back and start again after the 10 items are entered is another solution.

Neither is very satisfactory, but what about this?

1. Clear the screen.

2. Print an instruction at the top. PRINT (not INPUT yet) all 10 questions with any prompts on alternate lines.

3. Use the CHR$ (27) "Y" sequence (or an easier replacement) to move the cursor to the answer to each question in turn, and accept the INPUT.

4. Ask whether to proceed. If the answer is no, then go back to the INPUT sequence, but this time accept a simple [RETURN] as confirmation that the entry stands.

I don't think that you will be able to write this routine in a couple of minutes, but, by using an array for the questions and another array for the answers, and sensible loops and subroutines, you should be able to develop a routine that, once written, can easily be adapted to a similar situation elsewhere in the program or, indeed, in another program. The second input sequence will probably need INPUT$ (1), INPUT, and SPACE$ to work in a presentable manner.

An interesting little program to try along these lines is to prompt for entries of Cost Price, Selling Price, and Profit %. A simple [RETURN] at the INPUT is taken as 'Don't know'. When two inputs have been made, the third one is filled in. You have made a mini-spreadsheet! Obviously, this idea could be extended to a greater number of variables.

In Appendix 7, we use the multiple input and spreadsheet idea in a program using DWBAS. In elementary mechanics, there are simple relationships between initial velocity, final velocity, acceleration, distance, and time (assuming the acceleration is constant). Any three of these determine the other two. If you study this program, you will see that it accepts any three of the five inputs you can make, and produces the result for the unknown inputs.

3.6 ON z GOTO or GOSUB

Some programmers do not appear to have heard of the word ON. This ought not to be the case. ON is a word that EVERY Mallard BASIC programmer should use often. The worst examples of not using it come in games programs, and you can generally spot a bad programmer by a set of five to ten lines all starting with the word IF. I have sometimes (out of the goodness of my heart) rewritten such programs, and the authors have been amazed at how their pigs fly at

double the speed. The reason for this is that the original program may have had to evaluate and make 10 different decisions, of which only one could be true. Using ON, all the decisions are made in a single statement. But it is not only in games programs that ON is conspicuous by its absence, so let us consider the use after a menu choice has been given.

It is simpler for the programmer, and not usually a great hardship to the user, to write a menu that ends with a prompt: "Please press key 1-7, depending on choice." In this case:

```
150 z$ = INPUT$ (1) : z = ASC (z$) - 48 : IF z < 1 or z > 7 THEN 150
```

The variable z contains 1-7, and is immediately ready for ON z.

You cannot use this if there are 10 or more options. In that case, you could use the letters A-M (say) to identify the chosen option. Here, you could calculate z by: z = ASC (UPPER$ (z$)) - 64 or the slightly fancier:

```
z = (ASC (z$) AND 223) - 64
```

I know what you are going to say next. Your accounts program is written so that the menu offers C for Cashbook, P for Payroll, I for Invoices, and Q for Quit, and you don't want to change them to 1-4 or A-D. Fair enough. There is a simple solution using INSTR:

```
150 i$ = "CPIQ" : z$ = UPPER$ (INPUT$ (1)) : i = INSTR (i$, z$) : IF i
= 0 THEN 150
```

A point worth making for the games programmers is that this technique can also be used with the cursor keys:

```
i$ = CHR$ (1) + CHR$ (6) + CHR$ (31) + CHR$ (30)
```

will give you left, right, up, and down.

Menus are far from being the only place where ON makes for neat programming but, if you make a point of using it in this fairly obvious situation, you will come to realise the many other occasions when you can use ON to advantage.

You can either use ON GOTO or ON GOSUB. Which is best depends on the general structure of your program. Don't feel that, because people say that GOSUBs are more stylish than GOTOs, you should try to use ON GOSUB rather than ON GOTO. Use the version which is more natural.

3.7 Menus

In the old days, programs were advertised as Menu Driven. The reader was meant to think: "Wow, how clever!" and reach for the cheque book. If you think about it, there are many programs which would be extremely difficult to write without the use of Menus. Consider this approach:

C3P3

```
100 DATA Rates and Rent, Food, Clothes, Drink, Car, Fuel, Holidays
101 Data Miscellaneous, Quit
110 t$ = " M A I N M E N U " : j = 9
120 RESTORE 100 : FOR n = 1 TO j : READ a$ (n) : NEXT
130 GOSUB 5500
140 ON z GOTO...
```

Since the Menu idea is really so simple, you will find that most programmers do not give it sufficient thought. What I have written above is the information for a particular menu. What the subroutine called at 5500 will do is to print and operate the menu. Have you seen the point, yet? The subroutine at 5500 will be able to operate ANY menu if you give it the information as we done above.

How you write the subroutine at 5500 is up to you. You can spend some time on it, as you will not have to rewrite it in your program, however many menus you evoke. Indeed, you will be able to pirate the code for any other programs that you write that need menus. (Appendix 6 gives a Menu subroutine in DWBAS that you may or may not wish to use.)

Perhaps you will centre the title string, call the items 1 to 9, and use a loop for printing on alternate lines. Notice how simple the use of an array makes the coding. You will prompt for your keypress, and write code for a subroutine to obtain the variable z as 1 to 9. If you like, you can make things more decorative by suitable use of inverse video and a box design. You will use the variable j at various points in the subroutine, so that the code will adapt itself to different numbers of items on the menu. And you will have a menu-sub that will last for life. OK, you say, that's such a simple idea that I could have thought of it myself, and you have been wasting good paper telling me. Yes, you could. But I didn't -- until I had written perhaps 100 menus!

Chapter 4: Files

This chapter, we look at BASIC problems that will only occur if you are using the discs to read or write files. Mallard BASIC is clearly written with such programs very much in mind, and works very efficiently. The manual covers the various types of files available in Mallard BASIC more thoroughly than it covers other aspects of programming. I have not a lot to add to this. Your efficiency with files will depend more on your general programming ability than anything I say here. However, there are one or two little points that are specific to such programs.

4.1 Sequential, random, or Jetsam?

Should you use sequential, random, or Jetsam files? I have used all of them but, generally, a simple random file seems to suit most purposes best. That's got a few Jetsam fans jumping up and down in their seats with anger! My view of Jetsam is that it is a very fine solution for some programs, especially very large databases, but that, most of the time, it is an unnecessary elaboration of what could be done with little loss of speed, and some gain of space, by random files. If I am wrong, I bow to your superior knowledge. My only excuse is that I don't enjoy writing database programs very much, although one has to do so, at times.

Are there any programming tricks that I can pass on from experience? Not many. If you follow the manual and the example programs, you should be able to write at least as good a filing program as I can.

EOF is a useful tool to signal the End Of File. In a sequential file, it works well. For example, we might use

```
PRINT "Teams in Football League Division 1:" : OPEN "I", 1, "div1.lst"
```

```
: WHILE NOT EOF (1) : INPUT# 1, a$ : PRINT a$ : WEND : CLOSE
```

(ROCHE> Indented, this gives:

```
        PRINT "Teams in Football League Division 1:"
        OPEN "I", 1, "div1.lst"
        WHILE NOT EOF (1)
              INPUT# 1, a$
              PRINT a$
        WEND
        CLOSE
)
```

However, if the list of teams had been stored in a random file, this simple program does not work (even if INPUT# is replaced by GET). EOF with random files has caused me (and I believe others) some headaches. An EOF is signalled at the start of a random file, as well as the end. This might be a good solution:

```
        WHILE LOC (1) = 0 OR EOF (1) = 0 :...: WEND
```

There is also a problem when you are GETting and PUTting in the same routine:

```
        WHILE LOC (1) = 0 OR EOF (1) = 0 : n = n + 1 : GET 1, n :...: PUT 1, n
: WEND
```

The PUT makes the EOF meaningless. One solution is to insert a dummy GET 1,n between the PUT and the WEND. Inelegant, but it works.

MKS$ and CVS (among others) are mystery words to some programmers. The manual uses them, and the manual, like the laws of the Medes and the Persians, must be obeyed. All MKS$ actually does is to code any single precision variable into 4 bytes, which will usually have the effect of compressing the code. If you prefer, you can enter a number as a string and subsequently use VAL. Incidentally, MKS$ (a) will contain the same four bytes as you would read from VARPTR (a) (See Section 7.3.). More mundane than mystical!

Another problem occurs when it is necessary to manipulate a string defined as a field. If you have written FIELD 1,128 AS a$, don't alter a$ itself, use another variable:

```
        GET 1, n : b$ = a$ :...: LSET a$ = b$ : PUT 1, n
```

You are allowed to use #1 instead of 1 as a file-reference number. Why you should want to do so (except that, at one point in the manual, this is used) is a mystery. Some people do use # in this context. I know because it does not work in an extended BASIC that I have written. Please, omit the #!

4.2 Saving memory

Most people think that a sequential file is the simplest type of file to use, and that a random file is more sophisticated. The reverse is probably true, especially in the case where there is just one field, normally 128 bytes long. Such a file can be used for saving ANYTHING in 128-byte blocks. In particular, a random file can be used for saving blocks of the computer memory, instead of the strings of DATA which are usually the building blocks of such a file. A COM file, which is a machine code program which loads at &H0100 (256), is identical to a random file of 128-character strings, each character in the string corresponding to one byte of machine code.

This use of random files is probably not generally known, and is not mentioned in the manual, so we will include a tutorial program. We are going to save Mallard BASIC but, before we save it, we could make some alterations. For example, if you prefer the different operation of REM, as suggested in Section 6.1, you could load the normal Mallard BASIC, do the relevant POKEs, and save BAS2 using this program. BAS2 can then be loaded on a future occasion, instead of Mallard BASIC.

A recommended alternative amendment is to POKE 257,150 and POKE 258,1 to save a 'warm start' Mallard BASIC. The value of this will be seen if you inadvertently crash into CP/M Plus and want to recover not only Mallard BASIC, but your unsaved program (See Section 5.5.). A normal load of Mallard BASIC does some housekeeping, which includes clearing out any program and variables in memory, and giving the start up messages. If this routine is bypassed by the POKEs above, you save any Mallard BASIC program already in memory -- this is called a warm start. You cannot use a warm start Mallard BASIC if you have just switched on the computer.

If you use the program below, load Mallard BASIC but DO NOT add any extensions. Just do the required POKEs, and load and run the program below. Obviously, you will need 28K spare on the disc.

```
        C4P1
        10 b$ = SPACE$ (128) : a = 1 : b = 0
        20 OPEN "r", 1, "bas2.com", 128 : FIELD 1, 128 AS a$
        30 FOR n = 0 TO 223
        40  v = VARPTR (b$) : POKE v + 1, b : POKE v + 2, a : LSET a$ =  b$  :
PUT 1
        50 a = a + b / 128 : b = b XOR 128 : NEXT : CLOSE
```

(ROCHE> Indented, this gives:

```
        10 b$ = SPACE$ (128)
        20 a = 1
        30 b = 0
        40 OPEN "r", 1, "bas2.com", 128
        50 FIELD 1, 128 AS a$
        60 FOR n = 0 TO 223
        70     v = VARPTR (b$)
        80     POKE v + 1, b
        90     POKE v + 2, a
        100    LSET a$ = b$
        110    PUT 1
        120    a = a + b / 128
        130    b = b XOR 128
        140 NEXT
        150 CLOSE
```

)

Line 40 is well worth a close look. What we are doing is a con trick on the computer to make it think that b$ is stored in the relevant part of the Mallard BASIC interpreter. It is then LSET from there into the respectable a$, and the computer does not even know that it has been conned. The use of XOR in line 50 is also a point to note.

A friend commented that he found this program very difficult to understand (he is being unduly modest). Il will agree that there are some unusual concepts, but the idea may well become clearer when you have read the comments on VARPTR in Chapter 7. It may also be instructive to include these two lines in the program:

```
45 PRINT a * 256 + b
46 PRINT b$
```

These give the address of the start of the record that is being processed, and the contents of the string being entered in the file. On the first loop, for example, you will see that it contains most of the start up message for Mallard BASIC.

If you save memory in this way, it is not essential that the locations from which it was saved are the same as the locations to which it will be reloaded. Hence, a COM file can be written in high memory, or even in strings saved to a random file, with the Mallard BASIC interpreter loaded.

4.3 File transfers

If you wish to transfer a random file from one disc to another in a Mallard BASIC program, say ADDRESS.DAT from drive A to M, this method will work:

```
100 OPEN "r", 1, "address.dat" : FIELD 1, 128 AS a$
110 OPEN "r", 2, "address.dat" : FIELD 2, 128 AS b$
120 WHILE EOF (1) = 0 OR LOC (1) = 0
130       GET 1 : LSET b$ = a$ : PUT 2
140 WEND : CLOSE
```

Oddly enough, the above method will work even if the file is a sequential one! While the program runs, the computer is fooled into thinking that it is a random file.

Chapter 5: Editing and debugging

Programming is done by a consenting adult and computer in private, so it is not always easy to find out what other programmers actually do in these sessions. Much of my time, and probably much of your time, is spent writing programs. It is obviously desirable that this time should be used as efficiently as possible. Experience is probably the greatest teacher, in this respect. We discover tricks, we file them in our own memories, and use them on future occasions. Yet, I have often thought: "Why didn't I realise that I could do that six months ago? It would have saved so much time!"

It is by no means a bad idea to sit behind another programmer and watch how he or she does it. For example, I watched somebody use the PIP command, and suddenly realised that I had been using it thoroughly inefficiently.

In one sense, this is not the easiest of chapters to write. Some things become so automatic that it is hard to realise others people may not know the tricks. You will have to forgive me when I mention the obvious -- it may not be obvious to somebody else.

Editing, debugging, and avoiding pitfalls due to oddities in the interpreter, are all part of the same process -- transferring an idea into code that works. We include all these aspects in a single chapter.

5.1 Editing

Most manuals seem to assume that the programmer is nearly perfect, and that most of the programs will run first time. Very little is written about editing, testing, and debugging programs. Most programmers spend at least as long on this stage as on writing the programs. We all have our methods of sorting out the problems, and nobody criticises us because it is all done behind closed doors. My baptism of fire was to teach a class of pupils in a room full of computers, with a dozen different programs going wrong in a dozen different places... That soon made me learn speedy techniques!

The Mallard BASIC editing system is not the friendliest that I have met, but it is not nearly as bad as I thought it was when I first used the Amstrad PCW. For instance, it is fussy about spaces, and friendly about whether you put words in capital or small letters. Previously, I had used computers that were fussy about capitals and tolerant about spaces. It is natural to prefer the system that you know, and any strange computer will seem difficult, in this respect.

One of my initial grudges was quite unfair. I do a lot of testing and discovery work in direct mode -- writing a single unnumbered lined and pressing [RETURN]. One that I have written countless times is a variation on:

```
FOR n = 1 TO LEN (b$) : PRINT ASC (MID$ (b$, n)); : NEXT
```

It is at least even money that I miss out a semicolon or a bracket and, without a full screen editor, you have to go through the whole process again... and so on. Or do you? The manual tells you that you can press [ALT]-A to recapture your line. Muggins did not read that for two months. Two months later, I realised that you DON'T have to press [ALT] and A -- cursor left is simpler! The manual does not tell you that, unless you have the correct page in Manual 1 and the correct page in Manual 2 open at the same time!

Cursor left has uses other that in direct mode. Pressing [RETURN] or [STOP] by mistake when you are writing a long line is annoying, but pressing cursor left immediately presents it again. Did you write LIST 40 when you meant EDIT 40? Press cursor left, LIST a program, and follow with cursor left. You can edit the last line. You have written SAVE"MYPROG", and it is such a good program that you want it on the backup disc. Change discs, press cursor left, and [RETURN]. A curiosity here is that this does not work with SAVE"MYPROG",A. The translation from the semi-compiled version of a Mallard BASIC program back to ASCII uses the vital buffer.

There are two other controls that I use regularly. Pressing [CUT][CUT] (a double press of the [CUT] key) will erase the rest of the line from the cursor position. Pressing [FIND][FIND] takes the cursor to the end of the line. Some people may use the [CUT] and [FIND] keys with other arguments. [FIND], for instance, would take you to the next statement. It may be obvious, but using cursor down can save time in editing a long Mallard BASIC line. Some very nimble-fingered people might find it an advantage to speed up the repeat key rates -- it is possible to do this.

It is debatable whether it is advantageous to set other function keys to assist editing. This can be done through SETKEYS, or directly in the Lightning BASIC extension (See also Chapter 16.). It may save time to set, say, [f8] to SAVE"MYPROG" and [f7] to LIST 1000-1200. Usually, I feel it is not quite worth the bother, but it depends on the individual's programming methods.

You probably know that [f5] can be used to halt a listing or a program, and any further keypress restarts. You probably know that ? can be used as an abbreviation for PRINT, and that you do not have to follow it with a space. But, maybe you didn't know!

AUTO and RENUM are useful editing tools, but we shall deal with them in Section 5.3.

5.2 Debugging

Debugging takes more hours out of programmers' lives than you would believe by hearing them talk of their successes! You may feel that you are the only one who has to write and rewrite and rewrite again. Believe me, you are not. Generally, I make more mistakes when I write in BASIC than in code. Most Mallard BASIC errors are simple to sort out, and a program crash is only a minor setback. If you crash in code, you usually have to start up all over again, and so it is worth checking things very carefully before putting it to the test by running. In Mallard BASIC, it is not usually worth the bother. The golden rule is to SAVE before you test. There are countless times I have regretted my carelessness! The second golden rule is to write short sections of a program, and test each section before you continue.

Syntax errors are usually the simplest. The report: "Syntax Error in line 567" will mean, 99 times out of 100, that there is something wrong with line 567. Look at it. Os and 0s can cause problems, as can Is and 1s. A colon (":") instead of a semicolon (";") in a PRINT statement may be the culprit. If the mistake is not obvious, count the opening brackets and closing brackets, and see if they balance. If the line is long, test to see how far it did actually execute -- is the sixth of ten statements is v=20, use ?v to see if it really is 20.

Having said that, I have recently spent some time debugging somebody else's program that crashed with "Syntax Error in 1031". The error turned out to be not even in the program itself, but in a previously chained program! The hardest errors to correct are those which occur because of a previous error, and not on the line itself. A common one is that a DEF FN is incorrectly entered, but the error occurs much later, when the function is used. DEFINT can cause problems that are tough to identify.

It is worth mentioning that, if you want to test part of a program, using GOTO (or GOSUB) instead of RUN retains the variables. The only official debugging aid that you have with Mallard is TRON. If I have a TRON, I don't often use it -- if I haven't got one, I always seem to need it! Debugging tools are often conspicuous by their absence, and Lightning BASIC does include a program search, a variable list, a variable-speed TRON, and the error line presented for editing for most errors, instead of just for Syntax Error. These things all help, but your main debugging aids will be intelligence and experience.

There is another type of debugging that is worth a mention. If you are writing a lot of PRINT lines for instructions, or designing a pretty menu or title page, don't expect to get it right first time. Write it quickly, and don't try to do more than get the syntax and most of the spelling right. Run it and it will be much clearer from the screen what alterations should be made.

We all have programs that just will not come right. The quickest answer can be to go back and start again, but I cannot say that I like doing this. If you do, you will never know what was wrong, and many programming skills are developed from one's own mistakes. Not so long ago, I had spent four hours on a section of a program that just would not work right. In disgust, I turned off the computer and went out. Half way down the third pint, I knew in a flash exactly what was wrong and, when I came home, it only took a couple of minutes to have the program working perfectly. If you are teetotal, sleeping on it is a good alternative!

I may be in line for the Masochist of the Year competition as I find debugging quite a satisfying part of programming! Often, I enjoy debugging programs written by others. Sometimes, this needs a special sort of tact, when the programs are so bad that it would be quicker to scrap everything and start from scratch! But it brings me to perhaps the most important point of all in debugging. Somebody else is much more likely to see the glaring mistakes, the muddled instructions, the input checks needed, the short cuts you have missed. If you are at all serious about a program, pass it on to a friend before letting it loose on the general population. With the Amstrad PCW, it is helpful if your friend has a different model of the machine, to check (for example) that something written using CP/M Plus Version 1.4 and Mallard BASIC Version 1.29 will actually run on CP/M Plus 2.1 and Mallard BASIC 1.39.

5.3 AUTO and RENUM

It is debatable whether AUTO is more of a help or a hindrance when you are composing a program. If you are copying from paper, of if the code is very straightforward, it can be of assistance. If the program is likely to be difficult and to need alterations as you write it, it is probably best not to use AUTO.

The real value of AUTO is in debugging. AUTO 3000,10 will present each line of your section, starting at 3000, in turn. You just press [RETURN] for the satisfactory lines, and edit those that need alteration. If you have not numbered in 10s, it may be necessary to use AUTO 3000,1 and keep pressing [RETURN] for empty lines. Even this is usually much quicker than using EDIT for each alteration.

Consider this frequent situation. You have code at 3000-3400 which prints pages of information to the screen. You want to allow a hard copy print of the same information. There will be some differences in the format of the paper printing, so the POKE suggested in Section 2.2 is not appropriate. First, SAVE"PROG" if you haven't done so. DELETE-2999 and DELETE 3401-. Now, RENUM 4000. AUTO 4000, and make the changes as the PRINT lines appear. Finally, MERGE"PROG". All very simple, but the sort of thing that programmers easily miss. An alternative way of making a 'near-copy' of a section of a program is to call it as a flagged subroutine, so that the flag gives the alternative actions when they occur. You can get back with:

```
IF sr = 1 THEN sr = 0 : RETURN
```

I am not entirely convinced about the merits of RENUMbering a program when it is complete. It may appear to be more organised, but it may also prove less readable after the RENUM. It is quite likely that you used a pattern of numbering when you wrote the program, and this pattern should become clear to the reader, and in effect be a pseudo-index. There is certainly a case for RENUMbering each subroutine that you write before incorporating it in the main design. If you are writing for a magazine that expects users to type in programs, it is usually sensible to RENUMber. This allows the typist to use AUTO.

5.4 Subroutine libraries?

Many programming tasks are repetitive, so it makes a lot of sense to create a subroutine library. It also takes more organisation than I usually display, so I cannot honestly say that I have a library of my own. However, I do have a good memory, and often lift subroutines from one program to another. I also

have programs with specialist types of subroutines, such as graphics. Yet, I have to admit that I do reinvent the wheel from time to time -- it is often quicker to rewrite a routine than to find it somewhere else. It is also possible that rewriting a subroutine improves on the original.

If you do write a subroutine library, it is a good idea to use internal variables (ones that are NOT passed to the subroutine by the main program), with names you are not likely to use elsewhere. For example, you could end variables with a 9, so that a subroutine might start with h9 = HIMEM. Library (or library-type) subroutines should be as general as possible. Variables should not be assumed to have a value of 0. Code routines should be made relocatable if possible, and should depend only on the initial value of HIMEM. If you use DATA, you should include a RESTORE.

On the whole, a subroutine library is a good idea, if you forgive the 'Do as I say, don't do as I do' attitude.

5.5 Mallard 'bugs'

One aspect of the programmers task is to make allowances for possible weaknesses in the language used. Luckily, Mallard scores very high marks in this respect -- I mean it has few bugs, not many! In fact, I don't think that I have found anything in Mallard which could actually be described as a full grown bug. At times, I think that I have found a Mallard bug, but I have always discovered later that it is my own faulty programming.

Mallard BASIC is less than perfect when it interacts with CP/M Plus. If an error occurs during a call to a CP/M Plus routine, it returns to the A> prompt after the message, instead of returning to Mallard BASIC. The simplest example of this is OPTION FILES "N" (instead of "M"). It is not too difficult to find other occurrences, mainly in disc handling routines. A strange one can occur if you press [f5] to pause a program, and the decided to [STOP]. Normally, everything is fine but, if the [f5]-press coincides with some CP/M Plus routines, the [STOP] acts as in CP/M Plus and returns to A>.

In Section 4.2, we showed you how to save an alternative Mallard BASIC, which runs from a 'warm' start. If you have this alternative, the accidental return to CP/M Plus is not nearly so troublesome, as you can usually return to Mallard BASIC with your program intact. There is one problem that can be far more disastrous than the above. I have only done it once... once is enough!

If you make an error in saving a program, you will usually receive a Mallard BASIC error message. No problem. Suppose that you have a write-protected disc -- possibly a master disc from Amstrad or some software company. You have forgotten to put in your programming disc, and leave the write-protected one in. The SAVE goes to CP/M Plus before the write protect is found. You get: "Retry, Ignore, or Cancel". It is the work of a moment to put in the correct disc and R for Retry. DON'T DO IT!!! The routine uses the directory of the previous disc, and will probably overwrite bits and pieces of several programs that you have on your own disc. Cancel returns to CP/M Plus, but this is not so bad, particularly if you have the warm-start Mallard BASIC available. Of course, this would never happen if you followed the rules, and always made back up copies of masters!

The following is a 'STOP PRESS' addition. I think that there is an answer to the CP/M Plus problem in Mallard BASIC. Like a new drug, it should be tested thoroughly to see whether there are any side effects. There has not been time to do so, so you use it at your own risk. If you include it, the CP/M Plus error system is changed, so that the message is printed but the program

continues. The result is often a bit ugly, but your program is not destroyed. After loading Mallard BASIC, POKE 64487,254 (This location is normally set to 0.). Alternatively, you could load Mallard BASIC and make the following amendments:

```
POKE 803, 143 : POKE 384, 0
DATA 14, 45, 30, 254, 205, 5, 0
FOR n = 399 TO 405 : READ a : POKE n, a : NEXT
```

Then use the method in Section 4.2 to save the new version, and the changed error trap will be installed when you use this version of Mallard BASIC.

Chapter 6: Style

The idea of this chapter is to pinpoint some of the main aspects in which two good programmers might have totally opposing views. Programming style can lead to heated arguments, but I hope that we will keep temperatures at a reasonable level, by putting both sides of most of the arguments -- even though you may just possibly detect that I am, sometimes, a little biased! I shall occasionally go off on a red herring. Some of these topics are not discussed elsewhere, and there are a few interesting little digressions to be made.

6.1 To REM or not to REM

Maybe this section is the unconverted preaching to the converted! Sometimes, I will REM a program, but it is not my normal practice. Occasionally, I write a program that I want the user to study, and the program is so simple that a few bright breezy REMs will explain all, even to the meanest intellect. Usually, it is better to document on-screen, in a separate DOC file, or on paper. Sometimes, it is better not to document at all -- you don't always want the user to understand your programs!

One argument for REMs is that, if you have written a program some time ago, and wish to amend it, you will have forgotten why and how you did this and that, and REMs will help you to remember. If you find this is true, it justifies putting in REMs and using up space. Once upon a time, I was strongly anti-REM, but now I am indifferent. I don't particularly like porridge either, but don't let me stop you eating it!

There is one rule that I do feel is worth keeping. Never GOTO or GOSUB to a REM -- particularly if you are writing for a magazine. Those who copy your program will probably leave out the REMs. I have no strong feelings whether you should use REM or ', since both work identically. If I use REMs, I prefer to number operative lines in 10s, and put the REM lines with numbers ending in 9. Incidentally, I only realised the other day that you can generally put a REM or a ' in the middle of a line WITHOUT a preceding colon (":").

One aspect of REM which differs in various dialects of BASIC is the question of whether a REM continues to the end of a line, or just to the next colon. Consider the line:

```
20 REM Degrees to radians: DEF FN r (x) = x / 45 * ATN (1)
```

Mallard BASIC would ignore the DEF FN. In some BASICs, the second statement is operative. I feel that this is preferable. To adjust REM so that it ignores only up to the next colon is not quite as easy as I thought it would be. It is necessary to adjust the editing process as well as the operation. However, it

is possible, but should be used with care. In Mallard 1.29:

```
        POKE  18720, 211 : POKE 18724, 0 : POKE 18725, 185 : POKE 18693, 37  :
POKE 18539, 117
```

In Mallard 1.39, a better way is to change RMDIR to RM, and use RM for an in-line REM. This can be done by POKEing 20197-20200 with 9, 9, 9, and 205.

A use of a REM that may not be obvious is that, if you make your first program line REM followed by a long string of rubbish, it is possible to write machine code into the rubbish without having to reserve memory elsewhere. In Mallard 1.29, the line 1 REM... enables you to POKE and CALL code from 31388 onwards. In Mallard 1.39, the relevant location is 31529.

Another obscure and perhaps more interesting use of REM is to create a self-modifying program. There is a school of thought that does not like a program that changes itself. (ROCHE> Because the program cannot be put in ROM, then.) I have found that, on one or two occasions, it is a very useful technique, and it is certainly fun, not to say spectacular! The idea is to put some REM... lines at the start of the program, which will be ignored. Later lines POKE values into the REM lines, and change the word REM into the appropriate one. Finally, the rest of the program is deleted, and the converted lines are saved. This does need some knowledge of how a program is stored, but will not usually require an understanding of machine code (See Section 9.3.).

6.2 ON ERROR GOTO

I was once told by the boss of a software firm: "If you don't put an ON ERROR trap throughout your programs, I will." He did. We had a complaint that Option 1 mysteriously ended with a prompt that should only occur in Option 4. On another occasion, the boss of another software firm refused to allow me to write in an ON ERROR trap. Who was right? I say neither was... but the variation shows up this matter of style.

ON ERROR is a useful command if, and only if, it is used with a specific purpose in mind. Once the specific hurdle is passed, the error trap should immediately be cancelled. There are two distinct uses of the trap. The first occurs when an error may happen and, if it does happen, you want the program to continue, and no action to be taken. A specific example might be in a general graph plotting program. A function such as y = SQR (x²-1) in the range x = -3 to x = 3 does not exist in the range -1 to 1. If the program has to plot points in this range, you want them to be ignored, rather than produce a built-in error message.

The other type of trap is required when you ask for an input that might be incorrectly entered in several ways, a filename for example. Put in a general error trap, with a RESUME taking you back to the original prompt. This is simpler than writing individual checks for everything that can go wrong. In this type of error trap, you can use ERR to give extra prompts for specific mistakes.

Some would say that it is obligatory for the programmer to provide an error trap for EVERYTHING that could possibly go wrong. My own view is that this, as with so many aspects of programming, should be modified according to the expected ability of the user. If I write a program to calculate standard deviations, and ask for items of data entry, I will happily put INPUT n. If the user gets a '?Redo from start' -- he, she, or it, should jolly well know better! If I write a menu which requests the computer to print a nursery rhyme with a choice of 6 rhymes, I will put:

```
        INPUT n$ : n% = VAL (n$) : IF n% < 1 OR n% > 6 THEN...
```

where something polite follows the "THEN".

6.3 Long or short lines?

It is probably easier to read programs that consist of a single statement per line. It is more efficient, in terms of compactness (and, very marginally, in speed), to use lines which include several statements. Two statements on two lines takes four bytes more than two statements on one line. The two considerations can be balanced against each other. It is not a question of correctness, but of style.

I think that even the short-line programmers would conceded that IF-THEN-ELSE lines may sometimes contain several statements. It can be avoided by IF... THEN GOSUB 6100 ELSE GOSUB 6200. The subroutine at 6100 will contain all the statements that a long-line writer would put between IF and THEN (followed by RETURN). The subroutine at 6200 will contain the statements that would follow the ELSE. As this does not make for easier reading, it is pretty pointless.

Generally, I would side with the long-line programmers, although I would not make a point of trying to cram as near to 255 bytes as possible into every line. If writing a program for publication on paper, it is sensible to consider how the program will appear. If the format is, say, 70 characters per line, it is worth taking a little trouble to see that no line exceeds this number, and that most lines do not fall too far short of it. Even in this situation, there are times when it will be sensible to exceed the 70 characters.

Line format also influence the ease or otherwise of debugging. If you are an inexperienced programmer, or one liable to many typing errors, short lines make for easier debugging. It takes experience to be able to spot the 'Syntax Error in 60' if line 60 consists of 15 to 20 statements.

Short-line programmers probably use long variable names. My main reason for disliking long variable names is that it is easy to misspell them, and hence they cause confusion. When we were about 11 years old, we were all taught to write "Let Bill's age be x", so that we could write down and manipulate the resulting equation. Perhaps the fashion now is to write: 3(BILLS.AGE+5)-4(BILLS.AGE-7)=60...!?

For those who like tidy programs with plenty of REMs, blank lines, and single statements, indented loops (see below) are probably a glimpse of heaven! A few BASICs do put in the indentations themselves, but Mallard BASIC (rightly in my view) has no time for such frivolities. This does not stop you putting them in for yourself if you value presentation highly.

An indented loop might look like this:

```
        100 FOR n = 1 TO 20
               110 FOR m = 1 TO 15
                    120 a (n, m) = 0
                130 NEXT m
        140 NEXT n
```

(ROCHE> Rubbish! Under Mallard BASIC, indented loop looks like this:

```
        100 FOR n = 1 TO 20
```

```
        110     FOR m = 1 TO 15
        120         a (n, m) = 0
        130     NEXT m
        140 NEXT n
```

Simply press the [Space Bar] to insert spaces after the line number.)

You may run up against something rather strange. If you indent in unextended Mallard BASIC, there is no problem. If you use either DWBAS or Lightning BASIC, the indentation will be edited out. Location 288 in the Mallard BASIC interpreter contains an apparently innocent 0 to signal the end of part of the welcome message. If it is POKEd to anything other than 0, the system of line editing is changed so that ANY UNNECESSARY SPACES will not be contained in the program. As my algorithm for extensions did not cater for extra spaces at the end of a line, I always put a POKE 288,1 into my extensions. You can POKE 288,0 and then put in indentations, but take care about silly spaces if you do this.

6.4 Protection and OPTION RUN

Mallard BASIC programs can, in theory, be protected from inspection by SAVE "XXX",p. Whether it is worth doing this is another matter. The only Mallard BASIC program on the master disc, RPED, is protected. There are two reasons I feel this is a mistake. There will be some Amstrad PCW owners who want to lean programming, and an inspection of RPED shows some useful and interesting tricks. It is probably more instructive than the demonstration programs in the manual. Secondly, users may want RPED to return to Mallard BASIC, or alter the keys which operate it. This is straightforward if you have the listing. I cannot see that protecting RPED gives any advantage.

Since several ways have been published of breaking a protected program -- Chapter 14 indicates the method that I use -- protection loses most of its value. The value of protection is presumably to prevent copying and, as a protected Mallard BASIC program can be copied as easily as any other, it seems rather pointless. The other value of protection is to restrict use of a program to certain users. In this respect, the Mallard BASIC protection system is far more foolproof than the CP/M Plus password protection, if you use it to best effect.

Imagine that a businessman wants to keep confidential information about his customers. This is probably illegal under the "Data Protection Act", so you or I would not do it, of course. This information is kept in a random file, and only certain senior members must have access to it, although there will be times when less senior people have access to the computers and discs. You are asked to help. You carefully code the information in the random file:

```
        C6P1  (CONF)
        500 INPUT "Enter details: ", b$ : b$ = UPPER$ (b$)
        510 c$ = "" : FOR n = 1 TO LEN (b$) : p = ASC (MID$ (b$, n))
        520 p = (n + 37) XOR p
        530 c$ = c$ + CHR$ (p) : NEXT
```

(ROCHE> Indented, this gives:

```
        500 INPUT "Enter details: ", b$
        510 b$ = UPPER$ (b$)
        520 c$ = ""
        530 FOR n = 1 TO LEN (b$)
        540     p = ASC (MID$ (b$, n))
```

```
        550     p = (n + 37) XOR p
        560     c$ = c$ + CHR$ (p)
        570 NEXT
)
```

This is an 'off the cuff' scrambling routine which should be difficult to decipher, and also reversible (interchange c$ and b$, and put a final PRINT instead of the initial INPUT).

Now, you have to save your program "CONF" from prying eyes. In Mallard 1.29, you may have heard that, somewhere in memory, comes the rather surprising message 'Acorn Computers'. This is actually part of the protection routine! First of all, we save CONF unprotected.

```
        C6P2
        1 INPUT "Enter your password: ", p$
        2 q$ = SPACE$ (16) : LSET q$ = p$
        3 FOR n = 0 TO 15 : POKE 22466 + n, ASC (MID$ (q$, n + 1)) : NEXT
```

(ROCHE> Indented, this gives:

```
        10 INPUT "Enter your password: ", p$
        20 q$ = SPACE$ (16)
        30 LSET q$ = p$
        40 FOR n = 0 TO 15
        50     POKE 22466 + n, ASC (MID$ (q$, n + 1))
        60 NEXT
)
```

We RUN this, and SAVE it as PASSWORD. LOAD CONF. Now, SAVE"CONF",P. It is now impossible to RUN"CONF" unless PASSWORD is run and the correct response is given. It is probably wise to make CONF an OPTION RUN program which either returns to SYSTEM, or re-enters "Acorn Computers" in the region 22466-22481.

It is rather sad, for Amstrad PCW9512 owners, that they cannot find this entertaining message. Perhaps the programmers at Locomotive have had to put up with too many rude comments. All is not lost, however. If you have Mallard 1.39, try this:

```
        10  FOR  n = 22644 TO 22659 : p = 256 - PEEK (n) : PRINT CHR$  (p);  :
NEXT
```

(ROCHE> Indented, this gives:

```
        10 FOR n = 22644 TO 22659
        20     p = 256 - PEEK (n)
        30     PRINT CHR$ (p);
        40 NEXT
)
```

The 1.39 coding contains a SUB (HL) for an ADD (HL), so that the message is a little more difficult to find! With 1.39, 22644 should replace 22466 when this has been used above.

OPTION RUN is another technique often favoured by programmers for no very good reason. I like to stop programs in the middle, from time to time. OPTION RUN is often a spoilsport technique! There are times when it is necessary. Occasionally, a part of a program could crash badly if [STOP] was pressed during execution. At other times, an inexperienced user might be confused -- for instance, if the cursor has been disabled.

The main reason for OPTION RUN is that it is often essential to do some housekeeping before leaving the program. An obvious example is a program that copies files to M disc, updates them on M disc, and must copy them back to A or B disc before ending. Also, if your program uses Jetsam keyed-files, a [STOP] at a crucial moment may prevent the index and data elements of the file from being marked consistent, with results too horrifying to describe.

6.5 Where does one put subroutines?

Mr A will tell you that you should always put subroutines at the beginning of your program. Mrs B will tell you to place your subroutines on higher-numbered lines (but preferably only slightly higher numbers) than the lines that call them. Let us dispose of this phantom stylistic point. Mr A and Mrs B were both right when they started programming. Mr A's version of BASIC always searched for a subroutine from the first line of the program to the last. Mrs B's version started from the current line, went to the end of the program, and then started again at the first line. In Mallard BASIC, it makes no difference (or only a negligible one). The interpreter replaces a line number in a GOSUB or a GOTO or RESTORE with an address in memory on the first occasion it executes that line. Each GOSUB, therefore, only needs a single search, and this is not going to make a significant difference in speed.

```
C6P3
10 GOSUB 40
20 GOSUB 40
30 END
40 m = 31382 : IF PEEK (5103) = 197 THEN m = 31523
50 FOR n = m TO m + 19 : PRINT PEEK (n); : NEXT : PRINT : RETURN
```

The subroutine prints out the PEEKs of the first two lines of the program. You will see that line 20 is altered after the line itself has been executed. Originally, the two bytes after the 28 refer to line 40. Later, 28 becomes 29 as the signal that the change has been made, and the next two bytes are the ADDRESS of line 40.

The answer is to place your subroutines in the most convenient spots. I favour the end of the program for general ones, and the end of a section for those that only apply to one section of the program.

6.6 GOTO taboo?

'BASIC is a bad language because a programmer can write poorly-designed programs that work by using GOTO.' This is what many computer theorists have been saying over the years. The argument is illogical, but that, in itself, does not necessarily excuse the use of GOTO.

There is little doubt that beginners in BASIC do use GOTO when it is untidy and unnecessary. Programs that jump about all over the place are difficult to read, and usually operate more slowly. A well-planned program should not need many GOTOs and, with experience, they can generally be avoided. I think that the academics would concede that it is not sensible to consider IF-THEN-GOTO or ON-GOTO as wrong. It is the unconditional GOTOs that set their blood racing! My view is that, sometimes, GOTO is the natural way to program, and there is no point in being devious to avoid it. It can usually be done by using an unnatural GOSUB or WHILE-WEND. But what's the point?

At one time, RENUM was a luxury not included in most BASICs. It was possible

to write parts of a program, and then have an afterthought that might (for example) make the presentation better. There was not room to fit in the extra coding -- so GOTO 12000 and write it there. With a RENUM, you can squeeze in the extra lines where they should logically be included. I am not sure that I would always advise this. There are disadvantages in renumbering a program which have been discussed elsewhere.

I would advise programmers to examine every unconditional GOTO that they use, and decide each case on individual merit! If this is done, your programs will gradually become more structured, as some structure actually makes them easier to write.

6.7 PRINT

PRINT is probably the first BASIC word we use. It is also, probably, the last one we master! I fear that I never won any prizes for handwriting, art, or design -- so it is more than likely that you can create prettier screens than I do.

The best computer screens, like the best referees, are the ones that are not noticed (graphic displays and high-tech games are exceptions). The user wants screen information to be clear and concise. An untidy screen is a distraction. So is an incorrect spelling, or a lack of punctuation. Your very clever programming trick may be, as well. It is irritating to see a programmer showing off. The user is generally more interested in getting on with the job that the program does, than seeing beautifully-designed title pages. If the user is likely to run the program every day, he or she does not want to stare at "Written by Joe Bloggs" for ten seconds each time the program runs. A comparison between LocoScript and WordStar is worth making. You don't notice the screen in LocoScript, whereas most people would probably feel that it is untidy in WordStar.

Having made this point, the rest of my comments are only suggestions, which you may or may not agree. When lengthy textual instructions have to be given on screen, I prefer printing on alternate lines, and usually with right justification. An alternative is to write in short sentences, and centre the text. Reverse video should be used sparingly, to emphasise important points or contrasts, and not just for decoration. PRINT USING is often helpful, but one must be aware of the effects of silly inputs. It is sometimes helpful to title each page with the option being performed. In most cases, the screen should not be seen to scroll. Blanking a screen by OUT 248,8 may give an instant and attractive effect, but it has to be weighed against a moment of panic for the user. Titles should be centred, but options on a menu should be aligned. Boxing-in text is tidy but, without a code routine, it can be annoyingly time-consuming. TABbing with full stops can often create a helpful effect. Screen flashing and underlining are occasionally useful. Consideration should be given to disabling the cursor.

Two little POKEs may be occasionally useful. POKE 24348,46 changes TAB so that it prints "..." instead of spaces. The 1.39 location is 24513 (46 is the ASCII code for a full stop). POKEing 17240 (17311 in 1.39) will change the input prompt from '?' to the ASCII character POKEd into this address.

Chapter 7: It pays to increase your word power

Up to this point, most references to Mallard BASIC reserved words have been to ones that you have used yourself. In this chapter, we discuss some words that

you probably do not use, and others that you may not use to their full extent. Mallard BASIC contains a great number of commands and functions. There are some that I have never used. Probably I never will use them. There are others that seem to be redundant, but which eventually turn out to be powerful tools.

7.1 FIND$

We start with FIND$, not because YOU ignore it, but because it took ME some time before I recognised the full uses of this function! I imagine that I am not the only one.

FIND$ is more flexible than most users may realise. For example, it can carry out a search on any disc by prefacing the filename with the disc name and a colon (":").

```
        n = 0 : WHILE n = 0 OR g$ = <> "" : n = n + 1 : g$ = FIND$ ("m:*.bas",
n) : PRINT g$ : WEND
```

(ROCHE> Indented, this gives:

```
        n = 0
        WHILE n = 0 OR g$ = <> ""
              n = n + 1
              g$ = FIND$ ("m:*.bas", n)
              PRINT g$
        WEND
)
```

This illustrates the point, giving a directory of all the BAS files currently on the M disc. It also shows the use of the parameter n, and illustrates that FIND$ can deal with wildcards. Maybe you knew this already, but I don't think that I picked it all up from my first read through the Mallard BASIC manual. You can use DIR M:*.BAS instead of this routine.

There is one other point worth mentioning, regarding FIND$. It is only for the dedicated hackers! If you have had a successful FIND$, the details of the file found will lie in the area 128-255 decimal. It is in the lap of the gods whether it will start at 128, 160, 192, or 224. The 32 bytes include the filename, where it resides on the disc, and the number of records. Try it, if you are curious!

7.2 DEF FN

Most Mallard BASIC programmers DO use DEF FN in escape sequences. Most other programmers do not use it at all! It is a powerful tool, and the Mallard implementation is very flexible. For non-mathematicians, the idea may be difficult at first sight, and this is probably why it is generally avoided, unless it can be copied from some other program.

A useful example is a routine to print a string in the centre of the screen.

```
        10 DEF FN t$ (t$) = SPACE$ (45 - LEN (t$) / 2) + t$
```

Make t$ the title that you require, and PRINT FN t$ (t$) will print it centrally. Incidentally, you don't have to use the same variable. a$ = "ACCOUNTS." : PRINT FN t$ (a$) will work, and so would PRINT FN t$

("ACCOUNTS.").

Earlier BASIC usually included DEF FN, but restricted it to numerical variables. Possibly this is the main use. Obvious examples come from mathematical functions not included in the BASIC. For instance, most users will not want the COSH function (except on dark nights in inner cities), but mathematicians may. Simple:

```
DEF FN cosh (x) = (EXP (x) + EXP (-x)) / 2
```

Finding the gradient of a graph is a standard mathematical procedure in Calculus. All right, go on to the next section if you are shuddering with horror! The program below shows how a DEF FN can be used to calculate the gradient of the graph 'y = x * x * x' at any point. FN a# (x) calculates the value of y for any value of x that the user chooses to input. The function is also used to calculate the gradient of a line between two points close to each other on the curve. This gives a close approximation to the gradient of the tangent at the actual point chosen by the input of x. Double precision variables are used, since the values of 'y' at the two points will only differ by a small amount.

```
C7P1
10 DEFDBL x
20 DEF FN a# (x) = x * x * x
30 INPUT "Enter value of x "; x
40 g# = (FN a# (x + 0.0001#) - FN a# (x)) / 0.0001#
50 g = ROUND (g#, 2)
60 PRINT "Gradient at (" x "," ROUND (FN a# (x), 2) ") is" g
```

We shall return to this program in Chapter 9. It has a fundamental fault -- the fact that the program has to be changed for each curve -- that is not easy to circumvent. For the moment, it is just an example of how DEF FN makes a difficult process simple.

7.3 MID$ and VARPTR

Most programmers use MID$ quite frequently, even if they do not use it to its full potential. MID$ is one of the few words which can be used as both a command and a function and, once you have registered mentally that it can be a command, you will find that you can use it very neatly, on occasions. However, the main reason for including MID$ in this section is that there are many occasions when MID$ is used where VARPTR would be neater and quicker (sometimes very, very much quicker). There is a powerful example of this in Section 4.2. ASC and MID$ could have been used for the same purpose, but it would have been much slower.

The average Mallard BASIC programmer does not use VARPTR, because it needs some understanding of how the computer organises variables into memory. You don't HAVE to understand this to write good BASIC programs. Even so, most Mallard BASIC programmers are quite interested in this sort of exercise. As some understanding may aid programming, it is worth a look.

VARPTR (a) or VARPTR (a$) will return a number which is the start of information about the relevant variable. If the variable is a string, VARPTR will point to three bytes of information. The first gives the length of the string. The second and third give the location of the start of the string (Second + 256*third). An integer variable is also fairly straightforward.

```
v = VARPTR (a%) : h = PEEK (v) + 256 * PEEK (v + 1)
```

This will return the value of a% in h. (Strictly speaking, UNT (h) will always equal a%.)

The storage of single (or double) precision variables is more complicated. Each single precision variable is stored in 4 bytes. If you want to find out how this works, you may be able to do this for yourself. For those who want some clues, try to understand this Mallard BASIC program, which obtains the value of a variable from the memory indicated by VARPTR.

```
C7P2
10 INPUT "Enter a number: ", a
20 v = VARPTR (a) : b = PEEK (v + 3) : c = PEEK (v + 2)
30 d = PEEK (v + 1) : e = PEEK (v)
40 f = 2^(b - 128) : IF c > 127 THEN f = -f
50 g = ((c AND 127) / 128 + d / 128 / 256 + e / 128 / 256 / 256)
60 PRINT f / 2 * (1 + g)
```

The idea of VARPTR is important if you use CALL with parameters. CALL v (a%, b$, c) will put VARPTR (a%) in register HL, VARPR (b$) in register DE, and VARPTR (c) in register BC.

7.4 LSET and RSET

A recent magazine article (which was otherwise a good one) described LSET and RSET as words which could only be used to precede a PUT in random filing programs. This is far from being the case. There is no reason why LSET (or RSET) should not be used to format the output from a sequential file, or indeed any other type of program. To use LSET outside a random file, it is necessary to define a string first -- e.g. a$=SPACE$(20). Any future string LSET into a$ will be truncated to the first 20 characters if it is over 20 characters in length, and padded with spaces to 20 characters if it originally has less.

Similar effects can usually be achieved by the use of TAB or PRINT USING, but LSET or RSET are often convenient and neater. There is an example of this in Section 6.4 but, once you have used LSET outside random filing, you will find it is often useful. Another example is given in the next section.

7.5 INSTR

BASICs that I used before Mallard did not include INSTR. I don't say that I dismissed INSTR as an unneccesary extra, but I certainly did not recognize it initially as an important programming function. It has already been discussed in INPUT routines, but I include it in this chapter as I feel that many Amstrad PCW programmers are not alive to its full use. It can be used as a search function in files or programs, but I feel that a simpler example may serve to awaken you to its power. The routine below is a dual-purpose illustration, as it also contains a powerful example of using LSET outside a random file!

```
C7P3
200 a$ = "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
210 INPUT "Enter month: ", m$ : m = VAL (m$)
220 n$ = SPACE$ (3) : LSET n$ = UPPER$ (m$)
230 IF m = 0 THEN m = (INSTR (a$, n$) + 2) / 3
240 IF m <> ROUND (m) OR m < 1 OR m > 12 THEN 210
```

```
        250 PRINT m
```

This is the hardest part of the date entry. It accepts a month, whether it is entered as 3, MARCH, Mar, march or (as far as I can see) any legitimate entry. It will NOT accept MA, which is ambiguous. As I have said elsewhere, everybody but everybody writes date routines. Can YOU do it more neatly?

7.6 CHR$ and ASC

You know about CHR$ (27). You probably know that PRINT CHR$ (7) makes the computer beep. Make the computer print this: She said "I love you."

```
        PRINT "She said "I love you."
```

Won't work! You need:

```
        PRINT "She said " CHR$ (34) "I love you." CHR$ (34)
```

This is not the main point I want to make. CHR$ is not just something you use if the computer is having a sulking fit and making you do things the hard way!

It is sometimes easier for the programmer to work in symbols. At other times, it is easier to work in numbers. CHR$, and the reverse function ASC, translate from one to the other. The ASCII code gives a correspondence between every symbol printed on the screen and a number between 0 and 255. Does it horrify you if I say that it is worthwhile to try to remember what ASCII codes 32-90 represent? If you do a considerable amount of programming, it will pay off, eventually.

Most programmers fight shy of ASC and CHR$ in anything but the most obvious situations. A good programmer takes advantage of them.

Some programmers may not realise that Greek and mathematical characters are available from Mallard BASIC using ASCII codes under 32. To produce these, they have to be preceded by CHR$ (27). PRINT CHR$ (27) CHR$ (7), for example, will give the 'therefore' sign. Here is a little trick question: How do you obtain the DOWN arrow sign? The answer appears at the end of the chapter.

7.7 PEEK, POKE, CALL, and USR

I was once told that the only reserved words I used were PEEK, POKE, and CALL -- because they were the only ones that I could spell correctly. Not very nice! Undoubtedly, your programs will be more powerful if and when you learn to incorporate machine code routines, but we are not including any advice on this in the first part of this book.

This should not stop you PEEKing about in memory, a perfectly safe procedure, although it may not mean very much. POKEing randomly is not so safe -- you will not damage the machine, but you may crash and have to switch off and on again. If you don't try, you won't learn. We have used POKEs quite freely in places and, naturally, all these are intended to be safe.

If you don't understand code, don't use CALL unless you are pirating it from somebody else. USR is generally a more complicated way to achieve the same result as a CALL. Mallard includes it on the grounds of compatibility with earlier BASICs, rather than for its own sake.

7.8 LIST

LIST? Yes, I have included this since few people use it in a PROGRAM. It is useful in instruction or demo programs. It is not used because it cannot be included successfully in a Mallard BASIC program, as LIST ends the program. You can change all this by a simple amendment that returns to the program, instead of the warm start address after a LIST. It does not even use a single extra byte. I put this change in any extension that I make, but it is easy to do yourself if you are using unextended Mallard. In 1.29:

```
DATA 229, 205, 17, 71, 205, 136, 71, 205, 3, 75, 225, 201
FOR n = 19191 TO 19202 : READ a : POKE n, A : NEXT
```

In Version 1.39:

```
DATA 229, 205, 65, 71, 205, 200, 71, 205, 104, 75, 225, 201
FOR n = 19292 TO 19303 : READ a : POKE n, A : NEXT
```

Answer to question in Section 7.6

PRINT CHR$ (27) CHR$ (9) gives the down arrow, BUT you have to use the command OPTION NOT TAB first, otherwise the above will be interpreted as a TAB.

Chapter 8: Writing for all PCWs

Many programmers write simply for their own use and pleasure. As you progress with programming, it is inevitable that you will want other users to be able to share your programs. The other users may be friends, fellow readers of an Amstrad PCW magazine, or unknown customers who will want to buy your programs. There are differences between the Amstrad PCW8256, PCW8512, and PCW9512 (ROCHE> And the PcW9256, PcW9512+, and PcW10!) that have to be considered if your version is going to run on each machine, and use the extra facilities that the bigger machines provide. This chapter discusses some of the points that you will have to think about.

8.1 Mallard 1.29 and 1.39

When the Amstrad PCW9512 was produced, assurances were given that it would run programs written for the other machines in Mallard BASIC and CP/M Plus. In general, this is true. The machine runs a different version of CP/M Plus, but most of the standard addresses are the same and, provided a CP/M program does not try to communicate directly with the BIOS, it will probably run on both machines. Our first concern is the different version of Mallard BASIC.

You can use Mallard BASIC Version 1.39 on an Amstrad PCW8256 or PCW8512 without any apparent difficulty. As far as I know, you can use Mallard BASIC Version 1.29 on the Amstrad PCW9512. For instance, the routine for protected programs has been tightened in Mallard BASIC Version 1.39. There may be other similar modifications. Mallard BASIC Version 1.39 does use 141 extra bytes.

If you never PEEK, POKE, or CALL, there is only one point to worry about, which is that Version 1.39 has various extra reserved words. This might be useful if they actually did something -- but they don't! Possibly, they

prepare for an update, since the words are similar to those used by a 16-bit machine running MS-DOS. If you use RMDIR in a 1.39 program, it will act in nearly the same way as REM. There is a small problem this does create. The new words cannot be used as variables. Three of them are CD, MD, and RD. It is not stretching imagination too far to say that you are quite likely to have used these in a program. CD might be used for the Current Date, for example. You may get an error, e.g. PRINT CD. You may get ignored, e.g. CD = VAL (date$). The other words are RMDIR, CHDIR, MKDIR, FINDDIR$, and CHDIR$.

C8P1

```
10 x = 19820 : IF PEEK (5103) = 197 THEN x = 19966
20 PRINT " "; : FOR n = 90 TO 65 STEP -1 : IF PEEK (x) THEN PRINT CHR$
(8) CHR$ (n);
30 p = 1 : WHILE p : p = PEEK (x) : x = x + 1
40 IF p = 9 THEN p = 46
50 IF p > 127 THEN p = p - 128 : PRINT CHR$ (p), CHR$ (n); : x = x + 1
: GOTO 70
60 PRINT CHR$ (p);
70 WEND
80 NEXT : PRINT CHR$ (8) " " : END
```

(ROCHE> Indented, this gives:

```
10 x = 19820
20 IF PEEK (5103) = 197 THEN x = 19966
30 PRINT " ";
40 FOR n = 90 TO 65 STEP -1
50     IF PEEK (x) THEN PRINT CHR$ (8) CHR$ (n);
60     p = 1
70     WHILE p
80           p = PEEK (x)
90           x = x + 1
100          IF p = 9 THEN p = 46
110          IF p > 127 THEN p = p - 128 : PRINT CHR$ (p), CHR$ (n); :
x = x + 1 : GOTO 130
120          PRINT CHR$ (p);
130    WEND
140 NEXT
150 PRINT CHR$ (8) " "
160 END
```

)

This little program will give you a list of all the reserved words in either version of Mallard BASIC. If you want hard copy (ROCHE> Difficult, since this program generates ASCII BackSpace (08H) characters!), you could change all the PRINTs to LPRINT, or use the POKE we told you about to do this. A full stop (".") will be printed in those words where a space is optional. (For example, GO TO and GOTO are accepted.) The test in line 10 is to check which version is used, and the start address of the word table is altered for 1.39 (ROCHE> There is a "word table" before the abbreviated keywords, but it is not used, here. Those 26 addresses correspond to Z-A (not A-Z).). As far as I can see, the tables are identical, apart from the words mentioned above.

The major problems with the 1.39 version will not affect you, unless you use PEEKs and POKEs to the interpreter, or machine code that makes use of the routines resident in the interpreter. If you can find a routine to do your work in 1.29, there will be a similar one in 1.39, but the start may be at a slightly (or very) different address in 1.29. For instance, v=3757 : CALL v... gives you "Improper Argument" in 1.29. You need v=3756 to do this in 1.39. Addresses of routines up to about 3000 appear to be identical in both Mallard BASICs but above that, normally, there is a difference. Usually, the routines

are identical, but you cannot absolutely rely on this. One favourite of mine -- v=4931 : CALL v -- evaluates from HL, and puts the answer in register A (0-255). In 1.29, the answer is echoed in register E but, in 1.39, the original value of register E is replaced. Hence, everything needs checking if a part-code program is to work in both versions.

It is also just possible that you have written programs that use PEEKs and POKEs into the program to modify it. If you do this, you will have to remember that the program start is normally 31382 in 1.29, and 31523 in 1.39. You need not worry about any POKEs that we give you in this book. If they are different in 1.39, it has been mentioned.

Obviously, the different printer is another problem with the Amstrad PCW9512. The answer is to keep printing as simple as you can, by keeping escape codes to the minimum. Some codes are different on the Amstrad PCW9512, unless you tell the daisy-wheel printer that it is a dot-matrix by LPRINT CHR$ (27) "@". Full graphic screen dumps are impossible, unless the Amstrad PCW9512 has a dot-matrix printer attached -- this is possible, but rather eccentric!

8.2 Using the discs available

If you are programming for yourself, it is easy to know how you will use the disc system that you have. If you are writing more generally, your program should adapt itself to use what discs are available. It looks as though it should be an easy problem to find out how many drives are fitted to the computer running the program. It is easy enough if you ask for an input by the user -- but this suggests a lack of professionalism.

If the program will need to chain other programs, or make use of existing files, it is sensible to use the M disc for anything at all complex. Section 4.3 shows a method of doing this. The housekeeping is done at the start of the program, and all the necessary discs are transferred to M. On quitting the program, any altered files are transferred to the disc on which they were found.

One solution is to use ON ERROR traps. Look for a file on the A drive. If an error occurs, trap it and look on B. This might cause untidiness with an Amstrad PCW8256, as a request to search the B drive usually brings a rather meaningless prompt.

C8P2

```
10 h = HIMEM : j = h - 15 : MEMORY j - 1
20 DATA 229, 205, 90, 252, 230, 0, 230, 1, 60, 225, 119, 201
30 RESTORE 20 : FOR n = j TO j + 11 : READ a : POKE n, a : NEXT
40 d% = 0 : CALL j (d%) : MEMORY h
```

This routine may be the answer. The variable d% will be set to 1 or 2, depending on the number of drives. Different routines and prompts can be written for the two cases. An alternative approach is to use the fact that, on the Amstrad PCW8256/8512, PEEKing 65359/65360 will give you the size of the A disc, and PEEKing 65413/65414 gives the size of the M disc. On the Amstrad PCW9512, the addresses are 65309/65310 and 65492/65493, respectively.

One disadvantage of using the M disc for all the program files is that it will often be necessary to use OPTION RUN for at least part of the program. This will ensure that any files that need to be saved to a physical disc will be saved when the Quit option is requested.

8.3 Machine code and compilers

If you are writing for other users, you will have to face the question of whether Mallard BASIC is an adequate language for the purpose. To some extent, this must depend on the application. For example, I do not think that you could write a commercially viable arcade game using Mallard BASIC alone. At the same time, I dislike the comments of the superior sniffers... "It is only written in BASIC..." "Pascal is my language. So structured, compared to BASIC..." "I have never used a high-level language. Assembler just comes naturally..."

Mallard BASIC is perfectly adequate for the majority of Amstrad PCW applications that you are likely to write. That is not to say that someone else could not write a superior version in a different language. However, it is generally better to write a good program in a language that you know well than to use a language that may be more suitable, but that you cannot exploit to the same extent as Mallard BASIC. There is not much that you cannot do in Mallard BASIC, with a little ingenuity.

The main drawback of Mallard BASIC is its speed. It is not as great a problem as many experts would have us believe. Computers work at high speed in any language, and many applications will work faster than the user can appreciate. In many programs, the only delays will occur when loading from disc, and a change of language is not going to make any difference. Besides, Mallard is a very fast version of BASIC. While Mallard is fundamentally an interpreter, it does act as a partial compiler. Some compiling is done when a line is edited into a program. More happens after "RUN", before the program executes, and some is done on the first execution of a program line.

If more speed is needed, there are two broad solutions. The first is to use machine code for those parts of the program which run slowly. The second is to use a compiled language.

Personally, I prefer the first solution. Mallard BASIC interacts well with machine code. We shall show various examples of this in the second part of the book.

Using a compiled language does not always mean that code routines are irrelevant. Just because Mallard BASIC does not access high-resolution graphics, it does not mean that your compiled language will do so! Indeed, you may well find that the compiled language has not been specifically designed for the Amstrad PCW, and may not interact so well with the machine.

A compiler should be quicker than an interpreter -- in theory. In practice, this will usually be so, but the compiled code may not be very efficient, and I have heard that some compiled language (CBASIC Compiler, for example) for the Amstrad PCW are, actually, slower than Mallard BASIC in places. The compiler is unlikely to execute as quickly as a custom-built machine code routine. This is not the place to discuss the relative merits of Forth, Logo, Pacal, C, etc..., some of which are compiled languages. I have heard of several Mallard BASIC programmers who have 'progressed' to new languages, only to return to Mallard BASIC in the end.

Is there a way of compiling a Mallard BASIC program? As far as I know, there is no direct Mallard BASIC compiler, and certainly no compiler that can deal with Jetsam. However, MBASIC can be compiled, and is very similar to Mallard BASIC. Trying to write and edit in MBASIC makes the Mallard BASIC editor look 100 years ahead of its time, but it is possible to write a program in Mallard BASIC, save it in ASCII, and run it in MBASIC. To do this, you must avoid certain words, such as FIND$, that do not operate in MBASIC. You have to use

USR instead of CALL, the compiler cannot deal happily with reserving space for code, and INPUT is user-megahostile. After all this, the program takes more space and does not run that much faster!

Chapter 9: Intermission!

This chapter does not quite fit into Part 1. Nor is it part of the guided tour of memory that we shall give you in Part 2. Hence the title! The first two sections discuss problems that you might expect Mallard BASIC to solve, problems that, on some computers, BASIC CAN solve. We find solutions, but a little code is needed in both sections. In contrast, the third section shows you some ideas that you probably did not imagine were possible in 'straight' BASIC.

9.1 Can you INPUT a function?

To start this chapter, I want to discuss a very old problem that I had with a different BASIC. It also happens in Mallard BASIC. You may remember this program in the DEF FN section:

```
10 DEFDBL x
20 DEF FN a# (x) = x * x * x
30 INPUT "Enter value of x "; x
40 g# = (FN a# (x + 0.0001#) - FN a# (x)) / 0.0001#
50 g = ROUND (g#, 2)
60 PRINT "Gradient at (" x "," ROUND (FN a# (x), 2) ") is" g
```

The unsatisfactory feature of this progrm is that, while the program will give the gradient at any point on $y = x^3$, it will not operate for any other curve, unless line 20 is changed. To expect the user to rewrite the program is definitely 'user unfriendly'. What we require is:

```
20 INPUT "Enter your function in terms of x: ", FN a# (x)
```

Mallard BASIC does not allow this. Nor is it any use trying to input an ordinary string and using VAL to evaluate it. This is not a problem that will worry every programmer, but it is a serious shortcoming if you wish to write programs that work in a general mathematical situation. It is not an incorrect answer to say that what we are trying to do is impossible in Mallard BASIC, and that, if we want to be able to input a function, we will have to use another implementation of BASIC, or another language. (ROCHE> Like Dr. Logo, running on the Amstrad PCWs.) However, mountains are there to be climbed, and there is a lot of satisfaction in making a computer do what 'it can't do'!

I am going to use two different approaches to the problem. The first is to do a few conjuring tricks, and come to a solution with a program that scores high marks for eccentricity and ingenuity. However, it could only be said to be user friendly if compared to the original. It does contain some machine code to set an expansion key but, in practice, you could use SETKEYS to do this. We put the INPUT string, together with necessary additions, into the buffer which is normally used by Mallard BASIC in the EDIT routine. The EDIT routine is then intercepted, and the program has written a line of its own into itself. That stops it! This is why we need the slightly unfriendly method of PRESS [COPY] KEY to continue.

C9P1

```
        5 e$ = CHR$ (27) : c$ = e$ + "E" + e$ + "H" : DEFDBL x
        10  h  = HIMEM : v = 51200! : MEMORY v - 1 : DATA 78, 26, 71,  22,  3,
205, 90, 252, 215, 0, 201
        20 DATA 6, 156, 14, 10, 33, 40, 200, 205, 90, 252, 212, 0, 201
        30 DATA 13, 71, 79, 84, 79, 32, 49, 48, 48, 13
        35 FOR n = 0 TO 10 : READ a : POKE 51200! + n, a : NEXT
        36 FOR n = 0 TO 12 : READ a : POKE 51220! + n, A : NEXT
        37 FOR n = 0 TO  9 : READ a : POKE 51240! + n, a : NEXT
        40 u = 51220! : CALL u
        50 a% = 11 : b% = 156 : CALL v (a%, b%) : MEMORY h
        51 PRINT c$ : INPUT "Enter your function: ", f$
        52 f$ = "100 DEF FN a# (x) = " + f$ + CHR$ (0) : bu = PEEK (511) + 256
* PEEK (512) - 1
        53 FOR n = 1 TO LEN (f$) : POKE bu + n, ASC (MID$ (f$, n)) : NEXT
        54 PRINT : PRINT "Now, press the [COPY] key."
        55 v = 510 : CALL v
        100 DEF FN a# (x) = x * x
        110 PRINT c$ : INPUT "Enter value of x "; x
        120 g# = (FN a# (x + 0.0001#) - FN a# (x)) / 0.0001# : g = ROUND  (g#,
3)
        130 PRINT "Gradient at (" x "," ROUND (FN a# (x), 3) ") is" g
        140 PRINT "Press X for new value of x, E to edit function, Q to quit."
        150 z$ = UPPER$ (INPUT$ (1)) : i = INSTR ("XEQ", z$) : ON i GOTO  110,
51, 160 : GOTO 150
        160 END
```

(ROCHE> $$$$)

Up to line 50, we are setting the [COPY] key to RETURN, followed by GOTO 100 and a further RETURN. The technique will be discussed in greater detail in Chapter 16. Lines 51-54 write the input function into line 100, and then the [COPY] key does a GOTO 100 to restart.

Whatever you may think of that program, it does not entirely satisfy criteria of user friendliness, and the only entirely satisfactory answer is to rewrite the BASIC. (ROCHE> Dr. Logo for the Amstrad PCW is a functional language, so interpreting a function is childplay for it. BASIC is a procedural language, so is a little bit lower level than Dr. Logo. In this case, it is clearly better to leave Mallard BASIC and use Dr. Logo. Each programming language has advantages and drawbacks. When you know several programming language, you select the one better suited for the job.) I am going to include a program that just uses one of the extras in Lightning BASIC -- I am not cheating, I wrote the code for it myself (ROCHE> Yes, you are cheating, since you are the only one who has it...) -- namely, an extension to EXP so that, if a string variable is entered, it will evaluate it as though it were a function. The EXP is very similar to EVAL in BBC or XTAL BASICs. I feel that it should have been included in Mallard BASIC. It is obviously a waste of time entering this program if you do not have Lightning BASIC. (ROCHE> Then, why publish it? Vanity?)

C9P2

```
        10 DEFDBL x
        20 INPUT "Enter function in terms of x: ", a$
        30 INPUT "Enter value of x: "; x : x1 = x
        40 b# = EXP (a$) : x = x + 0.0001# : c# = EXP (a$) : g# = (c# - b#)  /
0.0001# : g = ROUND (g#, 3)
        50 PRINT "Gradient at (" x1 "," ROUND (b#, 3) ") is " g
        60 PRINT "Press X for new value of x, E to edit function, Q to quit."
        70  z$ = UPPER$ (INPUT$ (1)) : i = INSTR ("XEQ", z$) : ON i  GOTO  30,
20, 80 : GOTO 70
        80 END
```

(ROCHE> Re-arranged, this gives:

```
        10 DEFDBL x
        20 INPUT "Enter function in terms of x: ", a$
        30 INPUT "Enter value of x: "; x
        40 x1 = x
        50 b# = EXP (a$)
        60 x = x + 0.0001#
        70 c# = EXP (a$)
        80 g# = (c# - b#) / 0.0001#
        90 g = ROUND (g#, 3)
        100 PRINT "Gradient at (" x1 "," ROUND (b#, 3) ") is " g
        110 PRINT "Press X for new value of x, E to edit function, Q to quit."
        120 z$ = UPPER$ (INPUT$ (1))
        130 i = INSTR ("XEQ", z$)
        140 ON i GOTO 30, 20, 150 : GOTO 120
        150 END
```

)

9.2 UDG program

The program below is more substantial than those included so far. It is an example of a techique that we mentioned in passing -- a self-modifying program. It shows in practice some of the ideas that have been given to you in theory. It is not possible to access the screen character set without a bit of code. (This will be discussed in greater detail in Part 2, so don't worry about it now.) We shall study the program in the order in which it was written -- at least you will see how madness has its methods!

The idea is to create a new program from an old one. Using the original program, UDGs (User-Defined Graphics) are created in the range CHR$ (192) - CHR$ (207). On exit from the original, you are told to SAVE, and the result can be the start of your own program using the new graphics.

C9P3

```
        1 DATA 00,00,00,00,00,00,00,00
        2 DATA 00,00,00,00,00,00,00,00
        3 DATA 00,00,00,00,00,00,00,00
        4 DATA 00,00,00,00,00,00,00,00
        5 DATA 00,00,00,00,00,00,00,00
        6 DATA 00,00,00,00,00,00,00,00
        7 DATA 00,00,00,00,00,00,00,00
        8 DATA 00,00,00,00,00,00,00,00
        9 DATA 00,00,00,00,00,00,00,00
        10 DATA 00,00,00,00,00,00,00,00
        11 DATA 00,00,00,00,00,00,00,00
        12 DATA 00,00,00,00,00,00,00,00
        13 DATA 00,00,00,00,00,00,00,00
        14 DATA 00,00,00,00,00,00,00,00
        15 DATA 00,00,00,00,00,00,00,00
        16 DATA 00,00,00,00,00,00,00,00
        49 GOTO 100
        50 h = HIMEM : j = INT ((h + 1) / 256) - 1 : k = j * 256 : udg = k
        60  FOR n = k + 128 TO k + 255 : READ a$ : a = VAL ("&H" + a$) :  POKE
n, a : NEXT
        70  DATA 1, 9, 192, 205, 90, 252, 233, 0, 201, 6, 128, 33,  128,  192,
17, 0, 190
        71 DATA 26, 78, 119, 121, 18, 19, 35, 16, 247, 201
```

```
        80  FOR n = k TO k + 26 : READ a : POKE n, a : NEXT : POKE k + 2, j  :
POKE k + 13, j
        90 CALL udg
        100 e$ = CHR$ (27) : c$ = e$ + "E" + e$ + "H"
        101 DEF FN a$ (r, c) = e$ + "Y" + CHR$ (32 + r) + CHR$ (32 + c)
        110  d = 191 : MEMORY ,,3 : p = 31388 : IF PEEK (5103) = 197 THEN p  =
31529
        150 k$ = CHR$ (1) + CHR$ (6) + CHR$ (31) + CHR$ (30)
        160 DIM k (7) : GOSUB 300
        170 PRINT c$; "The UDGs are ready to be saved."
        171 PRINT : PRINT "DON'T FORGET TO DO SO!" CHR$ (7)
        180 DELETE 49,190
        190 DELETE 100-1000,65000
        200 z = 0 : r = 0 : c = 0 : a = 127 : b = 128 : WHILE z <> 27 :  PRINT
FN a$ (13 + r, 41 + c);
        205 z$ = INPUT$ (1) : z = ASC (z$)
        210 i = INSTR (k$, z$) : ON i GOSUB 700, 710, 720, 730
        220 IF z = 32 THEN GOSUB 740 : ELSE IF z > 32 THEN GOSUB 750
        230 b = 2^(7 - c) : a = 255 - b : WEND
        240 RETURN
        290 :
        300 WHILE d < 207 : d = d + 1
        305 PRINT c$ : PRINT "CHARACTER NUMBER"; d
        310  PRINT  :  PRINT  "Press Y to design it, X to  end,  C  to  change
number."
        320 z$ = UPPER$ (INPUT$ (1)) : i = INSTR ("YXC", z$)
        321 ON i GOTO 340, 370, 330 : GOTO 320
        330 PRINT : INPUT "Enter number between 192 and 207: "; d
        331 IF d < 192 OR d > 207 THEN 330
        340 GOSUB 800 : GOSUB 200
        345 FOR n = O TO 7 : s$ = "00" + HEX$ (k (n)) : s$ = RIGHT$ (s$, 2)
        350 u = ASC (s$) : v = ASC (RIGHT$ (s$, 1)) : q = p + (d - 192) * 30 +
n * 3
        360 POKE q, u : POKE q + 1, v : NEXT : WEND
        370 RETURN
        690 :
        700 c = c - 1 - (c = 0) : RETURN
        710 c = c + 1 + (c = 7) : RETURN
        720 r = r - 1 - (r = 0) : RETURN
        730 r = r + 1 + (r = 7) : RETURN
        739 :
        740 k (r) = k (r) AND a : PRINT " "; : GOTO 760
        750 k (r) = k (r) OR b : PRINT CHR$ (188);
        760 c = c + 1 : IF c = 8 THEN c = 0 : r = r + 1 : IF r = 8 THEN r = 0
        770 RETURN
        790 :
        800 PRINT FN a$ (12, 40); CHR$ (134) STRING$ (8, 138) CHR$ (140)
        810  FOR n = 0 TO 7: PRINT FN a$ (13 + n, 40); CHR$ (133)  SPACE$  (8)
CHR$ (133) : NEXT
        820 PRINT FN a$ (21, 40); CHR$ (131) STRING$ (8, 138) CHR$ (137)
        821 PRINT FN a$ (25, 0); "Use cursor keys to move."
        822 PRINT "Spacebar for unlit pixel."
        823  PRINT "Any LETTER key to light pixel." : PRINT "Use  [EXIT]  when
finished."
        830 FOR n = 0 TO 7 : k (n) = 0 : NEXT : RETURN
        890:
        65000 END

(ROCHE> Indented, this gives:

        10 DATA 00,00,00,00,00,00,00,00
```

```
20 DATA 00,00,00,00,00,00,00,00
30 DATA 00,00,00,00,00,00,00,00
40 DATA 00,00,00,00,00,00,00,00
50 DATA 00,00,00,00,00,00,00,00
60 DATA 00,00,00,00,00,00,00,00
70 DATA 00,00,00,00,00,00,00,00
80 DATA 00,00,00,00,00,00,00,00
90 DATA 00,00,00,00,00,00,00,00
100 DATA 00,00,00,00,00,00,00,00
110 DATA 00,00,00,00,00,00,00,00
120 DATA 00,00,00,00,00,00,00,00
130 DATA 00,00,00,00,00,00,00,00
140 DATA 00,00,00,00,00,00,00,00
150 DATA 00,00,00,00,00,00,00,00
160 DATA 00,00,00,00,00,00,00,00
170 GOTO 360
180 h = HIMEM
190 j = INT ((h + 1) / 256 - 1
200 k = j * 256
210 udg = k
220 FOR n = k + 128 TO k + 255
230     READ a$
240     a = VAL ("&H" + a$)
250     POKE n, a
260 NEXT
270 DATA 1, 9, 192, 205, 90, 252, 233, 0, 201, 6, 128, 33,  128,  192
280 DATA 17, 0, 190, 26, 78, 119, 121, 18, 19, 35, 16, 247, 201
290 FOR n = k TO k + 26
300     READ a
310     POKE n, a
320 NEXT
330 POKE k + 2, j
340 POKE k + 13, j
350 CALL udg
360 e$ = CHR$ (27)
370 c$ = e$ + "E" + e$ + "H"
380 DEF FN a$ (r, c) = e$ + "Y" + CHR$ (32 + r) + CHR$ (32 + c)
390 d = 191
400 MEMORY ,,3
410 p = 31388
420 IF PEEK (5103) = 197 THEN p  = 31529
430 k$ = CHR$ (1) + CHR$ (6) + CHR$ (31) + CHR$ (30)
440 DIM k (7)
450 GOSUB 680
460 PRINT c$; "The UDGs are ready to be saved."
470 PRINT
480 PRINT "DON'T FORGET TO DO SO!" CHR$ (7)
490 DELETE 170,500
500 DELETE 360-1170,1180
510 z = 0
520 r = 0
530 c = 0
540 a = 127
550 b = 128
560 WHILE z <> 27
570       PRINT FN a$ (13 + r, 41 + c);
580       z$ = INPUT$ (1)
590       z = ASC (z$)
600       i = INSTR (k$, z$)
610       ON i GOSUB 940, 950, 960, 970
620       IF z = 32 THEN GOSUB 990 : ELSE IF z > 32 THEN GOSUB 1000
```

```
        630       b = 2^(7 - c)
        640       a = 255 - b
        650 WEND
        660 RETURN
        670 :
        680 WHILE d < 207
        690       d = d + 1
        700       PRINT c$
        710       PRINT "CHARACTER NUMBER"; d
        720       PRINT
        730       PRINT "Press Y to design it, X to end, C to change number."
        740       z$ = UPPER$ (INPUT$ (1))
        750       i = INSTR ("YXC", z$)
        760       ON i GOTO 340, 370, 330 : GOTO 740
        770       PRINT
        780       INPUT "Enter number between 192 and 207: "; d
        790       IF d < 192 OR d > 207 THEN 770
        800       GOSUB 1040
        810       GOSUB 510
        820       FOR n = O TO 7
        830           s$ = "00" + HEX$ (k (n))
        840           s$ = RIGHT$ (s$, 2)
        850           u = ASC (s$)
        860           v = ASC (RIGHT$ (s$, 1))
        870           q = p + (d - 192) * 30 + n * 3
        880           POKE q, u
        890           POKE q + 1, v
        900       NEXT
        910 WEND
        920 RETURN
        930 :
        940 c = c - 1 - (c = 0) : RETURN
        950 c = c + 1 + (c = 7) : RETURN
        960 r = r - 1 - (r = 0) : RETURN
        970 r = r + 1 + (r = 7) : RETURN
        980 :
        990 k (r) = k (r) AND a : PRINT " "; : GOTO 1010
        1000 k (r) = k (r) OR b : PRINT CHR$ (188);
        1010 c = c + 1 : IF c = 8 THEN c = 0 : r = r + 1 : IF r = 8 THEN r = 0
        1020 RETURN
        1030 :
        1040 PRINT FN a$ (12, 40); CHR$ (134) STRING$ (8, 138) CHR$ (140)
        1050 FOR n = 0 TO 7
        1060     PRINT FN a$ (13 + n, 40); CHR$ (133)  SPACE$  (8) CHR$ (133)
        1070 NEXT
        1080 PRINT FN a$ (21, 40); CHR$ (131) STRING$ (8, 138) CHR$ (137)
        1090 PRINT FN a$ (25, 0); "Use cursor keys to move."
        1100 PRINT "Spacebar for unlit pixel."
        1110 PRINT "Any LETTER key to light pixel."
        1120 PRINT "Use  [EXIT]  when finished."
        1130 FOR n = 0 TO 7
        1140     k (n) = 0
        1150 NEXT
        1160 RETURN
        1170 :
        1180 END
)
```

Lines 1-16 were written first. These provide the part of the program that will be altered. The reason for writing them first is that they don't take much copying with clever use of AUTO and RENUM. Work it out for yourself!

I then wrote the subroutine at 800, to put a grid on the page. Some of the lines here were afterthoughts. Now comes the main subroutine at 200. This allows designing on the grid, by use of the cursor keys and others. A point to note is that the input key is recognized by z$ or by z, its ASCII code. The array K(7) contains the binary representation of each row at the current state of play. You will notice the use of INSTR and the overuse (?) of subroutines in the 700-799 range.

The 700+ subroutines are neat (although I say it myself)! IF is avoided in the checks whether the cursor goes off the grid by using logical expressions. The AND and OR in 740 and 750 are worth noting. It is also a point that the UNCONDITIONAL GOTO in line 740 makes for tidy and structured coding!!!

At this point, it was time for a checking (the grid had already been tested) and so a few lines in the 100-section were needed. Nothing too disastrous -- the cursor went up when it should have gone down, and a few inevitable syntax errors. On to the routine at 300, which is the nub of the program.

300-340 are routine stuff. 345-360 do the modification. The k() array contains the vital numbers but, first, they have to be turned into hex, and each two digits poked into the correct place in the first 16 lines.

All that is left -- after testing this -- is to finish off the program, so that it ends tidily, deletes what is unwanted, and prompts to save. That, plus the machine code routine in lines 50-99, which will be required in the saved amemdment. This code first reads the data into an appropriate part of the memory, and then includes a machine code routine to change the relevant characters. It calls it by CALL udg. This code is a toggle, so that, when you have finished your own program, you can CALL udg again to restore the character set to normal.

A friend rightly pointed out to me that there is a simpler way to write line 345. FOR n=0 TO 7 : s$ = HEX$ (k (n), 2). I did not know that you could format a HEX$ like that, although it IS in the manual... Did you?

9.3 Sound and OUT

Compared with the computers of the 1970s, Mallard BASIC allows you tremendous sound effects. By PRINT CHR$(7), you can actually produce a BEEP! For most Amstrad PCW users and programmers, that is quite enough, thank you! The Amstrad PCW can, in fact, do a little more than this, although it does fall somewhat short of the effects that a well-run symphony orchestra can produce. You can produce different notes with differing tempos, but you will have to use machine code to do anything effective.

Mallard BASIC is capable of turning on sound and turning it off, but it is not fast enough to produce the frequencies by which you can vary a note to taste. The program below uses OUT 248,11 (Beeper on) and OUT 248,12 (Beeper off) to produce a slightly different note, but I don't think that you will get much more than this without using code.

C9P4

```
10 PRINT CHR$ (7)
20 FOR n = 1 TO 100 : NEXT
30 FOR m = 1 TO 20 : OUT 248, 11 : OUT 248, 12 : NEXT
```

Although OUT and INP are Mallard BASIC words, most applications will be of little use without code. INP is generally used with peripherals not supplied

with the computer (e.g. mice), and so is beyond the scope of this book. You will find further use of OUT in the secont part of the book. However, there are one or two effects of OUT that can be safely used with Mallard BASIC programs.

OUT (like POKE), if used indiscriminately, will often cause a crash into limbo from which the only remedy is to switch off. However, there are some OUTs which can be used safely, and may impress your friends.

OUT 246,n will reset the screen by n pixels vertically. Hence, a 'circular' scroll can be achieved. If a line disappears from the top, it will appear at the bottom, and vice versa. For example:

```
FOR n = 255 TO 0 STEP -1 : OUT 246, n : NEXT
```

OUT 247,n produces a flash. If n is 0-63, it is an off, on flash. If n is greater than 127, then you get an inverse flash. OUT 247,64-127 seems to operate only if the inverse screen is installed.

OUT 248 does various things. The most useful are the sound values as above and the screen off and screen on. Screen off OUT 248,8. Screen on OUT 248,7. The one to watch is OUT 248,1. This has the same effect as [SHIFT] [EXTRA] [EXIT]. So, we end Part 1 of the book with:

```
OUT 248, 1
```

Part 2

Chapter 10: A magical mystery tour

You have just booked your ticket for our magical mystery tour of the Amstrad PCW. You probably bought the Amstrad PCW as a word processor. You have realised that it is much more than a word processor. It is a computer that can be programmed to do those dull and necessary tasks of life, such as creating databases and invoicing. Probably, you have not yet realised just how powerful a computer the Amstrad PCW actually is. It has a lot of secret worlds that we intend to explore.

Even on a mystery tour, you do have to make some preparations before you start out. In this chapter, we shall discuss our mode of transport, the essential equipment, and an outline map of the countries to which we may be travelling.

10.1 The transport

Computers work with sequences of numbers but, for all practical purposes, mere humans have to work with a language that the computer can translate into these sequences that it adores. Languages fall into two categories. High-level languages are those that, to a greater or lesser extent, can be understood as readable English. Low-level languages are series of instructions (sometimes mnemonics) that are much closer to the language the computer understands. The lower the level of the language, the faster the computer will operate. It is also true that, in low-level languages, it is possible to make the computer stretch itself to the limit of its ability.

We shall continue to use Mallard BASIC as our high-level language -- our main form of 'transport'. Mallard BASIC is the bus which will be taking us from

hotel to hotel but, just as would happen on a real tour, we shall have to use more exotic transport to traverse the deserts, or scale the mountains. In this case, it does mean that we shall have to use assembler or machine code, from time to time. I don't think that you will find that you have to understand Z-80 code to follow where we are going, but you must not be frightened about it. Otherwise, you will miss some of our best day trips! There is nothing to stop you skipping any sections that you find too difficult, perhaps to return to them later. The code needed is all contained in the programs on your disc. Naturally, if you want to develop our ideas, you will need some knowledge of what is happening.

Even when we are using code, we will usually avoid hexadecimal. There are two reasons for this. The first is purely selfish -- unlike most people who use code, I prefer to think in decimal, as I can remember numbers better that way. The second is that readers who prefer hexadecimal will easily be able to convert from decimal, whereas the reverse may not be true.

10.2 Equipment

There are just two items of equipment that I would advise readers to take on this tour: a disassembler, and a multiple poke. Some people will prefer to use an assembler as well, although it is not essential. (ROCHE> It depends on the size of your code. One subroutine can be hand-coded, but anything more complex needs to be written in a file, so be processed by an assembler.) You can just about write in Assembler by using SID and various other rather horrific sounding facilities, all of which are on side 3 of the master discs. These have the disadvantage that they were written for an earlier processor, the Intel 8080, and do not operate with all the Zilog Z-80 commands. (ROCHE> A Z-80 version of SID was made by Digital Research: ZSID.) If you want to use an assembler frequently, it is better to buy a more modern system of your own. The Hisoft discs are probably the ones most used by the professionals. (ROCHE> The standard Z-80 assembler is Microsoft's M80 Version 3.44.) At DW Computronics, we have produced an assembler toolkit and tutorial that may well be easier for beginners. (ROCHE> I did not manage to find it.)

Assembler kits are an optional extra. All the examples in this book will work from Mallard BASIC. Provided that you can copy in strings of figures, you can work without an assembler. (ROCHE> Only if the code is relocatable...) We shall include assembler code examples (written with the DW Computronics assembler) in the chapter on Graphics, but we shall always include an alternative coding which can be typed straight into Mallard BASIC.

Personally, I do not usually use an assembler, but this is an individual choice. Most people feel that it is essential for writing their own code. (ROCHE> No: the most important tool is a good debugger, like ZSID.) However, I make a lot of use of a disassembler, and I think that most readers will learn more from these chapters if they sometimes use a disassembler. (ROCHE> This only shows the high-level mnemonics. A real debugger will show the contents of the flags and the registers as each instruction is executed. This is the only way to know what is going on, inside the Z-80 processor.) SID has an option that will give you a disassembly but, as this only recognises Intel 8080 codes, it is rather limited for our purposes. (ROCHE> Simply use the Z-80 version of SID: ZSID.) Program A5 is a disassembler (ROCHE> Sorry, but this is just showing the high-level mnemonics: this is a subset of any debugger, not a real disassembler, producing complete source code files ready to assemble.) that is only slightly modified from the first program I ever wrote on the Amstrad PCW!

I am a strong advocate of the MULTIPLE POKE, although Mallard BASIC, rather

surprisingly, does not include this facility. A multiple poke allows successive bytes to be POKEd into memory with one command.

```
POKE 128, 3, 4, 5
```

is equivalent to

```
POKE 128, 3 : POKE 129, 4 : POKE 130, 5
```

This makes code routines shorter to write. It is over 12 times as fast as the DATA-READ-POKE method. It also makes writing relocatable code easier. Anyway, I am going to use it in this part of the book, whether you like it or not! If you use Lightning BASIC or DWBAS, there is no problem. Appendix 4 contains routines for 1.29 and 1.39 which will give you this facility, although nothing else. This may occasionally be useful for construction of your own programs.

10.3 The map

We are going to assume that you have an Amstrad PCW8512 or PCW9512 for the moment. If you have a PCW8256, all will be made plain at the end. Some people find it something of a mystery that, while you have a 512K computer, loading Mallard BASIC gives you the message that you only have about 31K left!

The reason is that the Z-80, at any instant of time, can only work with 64K of RAM. (RAM stands for Random Access Memory, bytes of memory that can be read or written). The other 448K of memory is available, ready present and waiting for an invitation to come into the accessible memory. As a programmer, you can call any other part of the 448K into accessible memory but, while it is there, some of the normal memory will be waiting outside.

You can do this by a Mallard BASIC command: OUT x,y. This is not usually much use, as interrupts will very soon change the memory back to normal. Let us assume that interrupts don't exist, as it is easier to explain in terms of a BASIC command. We shall use two terms, BANKS and BLOCKS, that are probably not in the official computer-speak language!

The 64K of accessible memory is divided into four BLOCKS of size 16K. The 512K of physical memory consists of thirty-two 16K BLOCKS. Each BANK is like a socket, and each BLOCK is like a pin. It is possible to put any pin into any socket.

We shall refer to the BANKS by the first parameter that we would use in an OUT command. I/O port 240 refers to memory bytes 0-16383, port 241 to 16384-32767, port 242 to 32768-49151, and port 243 to 49152-65535.

BLOCKS will be numbered 128 to 159. The usual numbering is 0-31, but the second parameter of OUT for these purposes is 128 to 159. When you are working in Mallard BASIC, blocks 132-135 will be the accessible ones, and they will be in the I/O ports 240-243, respectively. Even in Mallard BASIC, there will be time when other blocks go into the accessible banks, but their visits are so short that you will not be able to detect this.

We talked about an outline map. This is a description of what is contained in each of the blocks. Without going into much detail here, CP/M Plus refers to the general Operating System program that applies to the Amstrad PCW, and to many other computers using the Z-80 processor. The BIOS is the program that CP/M Plus addresses to achieve the specific results for the Amstrad PCW. The Operating System looks after the fundamental tasks, such as accessing the screen, discs, and printer.

BLOCK 128 contains BIOS routines. BLOCK 129 contains some more, and also part of the screen memory. BLOCK 130 contains the rest of the screen memory. BLOCK 131 contains other BIOS routines that are not screen-oriented.

In CP/M Plus, blocks 132-135 are known as the TPA ("Transient Program Area"), where COM programs are loaded and run. It is probably more interesting to see how they operate when BASIC.COM (which is a transient program) is run.

The first 256 bytes of BLOCK 132 contain the CP/M Plus work area, known as "Page Zero". (ROCHE> Under CP/M, a "page" is 256 bytes long, which correspond to 00H to 0FFH in hexadecimal. When displaying memory locations in hexadecimal, each time the location reaches 00H, you have reached another page. So, 0000H = Page Zero. 0100H = Page One (Start of the TPA). So, there can be 00H to 0FFH, or 256, pages.) Mallard BASIC will sometimes refer here (ROCHE> When asking the BDOS of CP/M Plus to do some system calls.), but this is not part of the interpreter. The rest of the block contains some of the interpreter.

The rest of the Mallard BASIC interpreter fits into BLOCK 133. This also contains the start of the Mallard BASIC program area.

BLOCK 134 contains more Mallard BASIC program area.

BLOCK 135 is sometimes known as COMMON MEMORY. This block is almost always resident in I/O port 243. 62982-65535 contains the CP/M Plus program (ROCHE> Well, the "resident" portion, since CP/M Plus puts most of the BDOS in another bank. So, the resident portion of CP/M contains the minimum to communicate with the other banks, where everything else is done.) The rest of the block can be used by Mallard BASIC programs (ROCHE> The bottom of this BLOCK, since the resident portion of CP/M Plus goes up to the top of the memory (65535 = 0FFFFH).), although some of them will use part of it for other purposes, by the MEMORY command (ROCHE> With MEMORY, you can allocate some of the memory below the resident CP/M Plus, for example to POKE long routines.).

BLOCK 136 contains the CCP program and the printer tables. When you see an A> prompt in CP/M Plus, the program in memory is the CCP program. It is just a program to do enough to load and run the COM program that you need.

BLOCKS 137-159 contain the M disc.

BANK 1 is the configuration normally present. The BLOCKS in the I/O ports 240-243 are 132, 133, 134, and 135.

The other common configuration (during interrupts, for example) is called BNAK 0. This consists of blocks 128, 129, 131, and 135. For screen operations, block 130 replaces block 131.

On the Amstrad PCW8256, the M disc will only extend from block 137-143. However, if a block in the range 144-159 is requested, the result will be the same as if the block 16 lower was called. This is a convenience as you can write programs with the code equivalent of OUT 242,159, which will put the last block of M disc into I/O port 242 on ANY Amstrad PCW computer. OUT 242,144 needs greater caution. On the Amstrad PCW8512, this would insert a block of memory disc but, on the Amstrad PCW8256, you would insert the main BIOS block... **Handle this with extreme care!**

Chapter 11: Graphics

In this chapter, we explore the world of high-resolution graphics. We have now reached a point where we cannot do everything from Mallard BASIC alone. Some machine code is needed in our programs. In this chapter, we shall sometimes show it in assembler but, even when we do this, don't try to understand if you don't want to do so. Just use the programs that follow, and see what happens.

There are many graphics routines that could be written, but they will all usually stem from one of a few fundamental ideas. Plotting a pixel, making new screen characters, saving to memory, dumping to a printer. Some of the coding is not difficult, once you know what is happening in blocks 129 and 130. We start by having a look at how this part of the memory is organised.

11.1 Binary patterns

Skip this section if you understand what is meant by a binary pattern.

The screen is organised into 23040 sets of 8 pixels. Each pixel can be lit or unlit. If a pixel is lit, it will form a tiny dot on the screen. From these dots, familiar shapes can be constructed. Each of the 23040 screen bytes contains a number 0-255, which describes a pattern of a horizontal row of 8 pixels. Representing pixels by * if lit and . if unlit, let us consider a typical pattern:

```
**..*..*
```

The left-hand "*" represents 128 if lit, and 0 if unlit. This is sometimes known as BIT 7, since 128 = 2^7. (ROCHE> Computers, and programmers, count from 0, not 1. So, Bits 0 to 7 (from right to left...) for a byte (8 bits).) The next "*" has the value 64, the third from the left has value 32, and so on until the eighth "*", which has a value of 1.

(ROCHE> So,

```
********                          and       ........
|||||||└-> Bit 0 =   1                      |||||||└-> Bit 0 =   0
||||||└--> Bit 1 =   2                      ||||||└--> Bit 1 =   0
|||||└---> Bit 2 =   4                      |||||└---> Bit 2 =   0
||||└----> Bit 3 =   8                      ||||└----> Bit 3 =   0
|||└-----> Bit 4 =  16                      |||└-----> Bit 4 =   0
||└------> Bit 5 =  32                      ||└------> Bit 5 =   0
|└-------> Bit 6 =  64                      |└-------> Bit 6 =   0
└--------> Bit 7 = 128                      └--------> Bit 7 =   0
                    ---                                          ---
                    255                                            0
```

255 + 0 = 256 possible values for a byte. QED.)

The number that would represent this particular pattern will be:

```
**..*..*
|||└-->   1
||└--->   8
|└---->  64
└-----> 128
          ---
          201
```

If 201 is expressed as a binary number, it will be 11001001 which, as you see, is exactly the same pattern as the pixels above.

11.2 The visible screen

When blocks numbers 129 and 130 are sent to I/O ports 241 and 242, bytes 22832-45871 contain the representation of all the pixels on the screen. Assuming that the computer has just been switched on and the screen has not scrolled, bytes 22832-22839 give the pattern of the character at the top left of the screen. (ROCHE> So, characters are 8 pixels tall.) Each byte contains a binary pattern representing a row of the character. (ROCHE> So, characters are 8 pixels wide.) For example, if the letter A was in the top corner, these bytes would contain 24, 60, 102, 102, 126, 102, 102, 0, respectively. 22840-22847 represent the second column of the top row, and this continues until the row is filled and bytes 22832-23551 have been used. 23552 would be the start of the second row down. If you do a few sums, you will see that, by the time all 32 rows are filled, we have used bytes up to 45871.

Even though this may seem too simple, it is in fact likely to be more complicated! We shall see how this happens in Section 11.4.

(ROCHE> So, letter A is made of the following binary patterns:

```
 24        ...**...
 60        ..****..
102        .**..**.
102        .**..**.
126        .******.
102        .**..**.
102        .**..**.
  0        ........
)
```

11.3 The screen characters

Bytes 47104-49151 hold the character set that will be familiar to all of you. Each screen character has an ASCII number, and each character will be held in 8 bytes in this table. For instance, A has the ASCII code 65, and 47104+65*8= 47624. So, 47624-47631 hold the same bytes as 22832-22839 did when we imagined an A at the top of the screen.

11.4 Roller RAM

When the screen rolls, every character on the screen moves into a new position. You would imagine that each of the bytes 22832-45871 would need to be altered. This is not the case. The top line disappears and the new bottom line is entered in its place, but the rest of this memory stays unaltered. The change takes place in the roller RAM table, which goes from 46592-47103. This means that about 1000 bytes may have to be changed, instead of over 20.000. It makes sense, and gives smoother and quicker scrolling.

The 512 bytes in roller RAM contain 256 two-byte numbers which indicate the position in memory of the start of each of the 256 rows of pixels in screen memory. I have used the word INDICATE instead of GIVE since, for some reason

beyond my comprehension, the start of each line has still to be calculated from this indication.

Using a BASIC formula, if the two bytes in roller RAM are i, low byte, and h, high byte, the line start is given by the formula:

```
2 * (i + 256 * h) - (i AND 7)
```

NO, NO, NO. Please, save me. Honest... I did not design this computer!

As an unimportant but mildly puzzling point, you may well be asking about the missing bytes 45872-46591. I don't know! They are filled with zeros, and the length is exactly a row of pixels. Could it be the status line when disabled? Apparently not. Could it have something to do with scrolling? This does not appear to affect it, although it may be used as a temporary dump. Another faint possibility is that it might be used in the [PTR][EXTRA] screen dump. Wouldn't it be dull, if we knew ALL the answers?

11.5 SCR_RUN

While it is possible to access screen memory by using OUT, it is generally easier and shorter to use a BIOS routine named SCR_RUN. (ROCHE> Actually, a subroutine of the Banked BIOS, not the Resident BIOS.) The result of calling this routine is that I/O ports 240-242 are occupied by block numbers 128-130, which include the screen environment. This call also has the advantage that interrupts are disabled and enabled where necessary, and the stack is moved to a safe position.

To use SCR_RUN, it is essential that it is fed with a routine of your own to run. This routine must end with a RET, AND IT MUST BE ENTIRELY IN COMMON MEMORY (48152-62981). (ROCHE> In hexadecimal: C000 to F605.) The coding looks a trifle odd -- to say the least! First, BC must be loaded with the start of the routine that you have written. You then CALL 64602 (ROCHE> FC5A, that is to say: USERF, a Resident BIOS routine that Amstrad uses to access the Banked BDOS and BIOS.), which is the general call for any BIOS routine (ROCHE> No! There is a generic "Call BIOS" routine, function 50, available under CP/M Plus. This one was customised by Amstrad to call the Banked BIOS. Only the Amstrad PCW uses it.) This is followed by TWO bytes which are the address in (ROCHE> banked...) BIOS where the routine starts. For SCR_RUN, the address is 233 (decimal) (ROCHE> E9: that is to say: the SCR_RUN routine in the banked BIOS.) Hence, after CALL 64602, the next byte must be 233, and the following one 0 (ROCHE> That is to say: 00E9. It is an INTEGER value.) If there is no further work to be done after returning from the CALL, you finish with another RET (after the 233 and 0). When you return from SCR_RUN, you are back in the normal environment, with the original stack. This means that it is possible to do further operations in code before returning to Mallard BASIC or any other main routine that you are using. (ROCHE> The chapter "The Implementation of CP/M Plus on the Amstrad CPC6128 and PCW8256", written by Locomotive Software, and published as Appendix C of "The Amstrad CP/M Plus", 1986, is much clearer.)

11.6 A new screen clear

As the authors of '1066 and All That' would put it -- roller RAM is a Good Thing. It is also a confounded nuisance when you are working with high-resolution graphics. In such a program, you will not want to scroll the screen. Sometimes, the answer is to restore roller RAM to normal before

working in high-resolution graphics. This makes the coding quicker and shorter. This routine clears the screen, and resets roller RAM at the same time. If it is included in a Mallard BASIC program, CALL j will do both these jobs. Here, the routine is written in Assembler (ROCHE> Well, obviously the famous DW Computronics assembler, must be written in Mallard BASIC, since it uses an ASC file produced by Mallard BASIC. Note also the non-standard mnemonics. And how do you insert new lines?):

```
1 'LD A 27
2 'CALL 1168
3 'LD A 69
4 'CALL 1168
5 'LD BC START
6 'CALL 64602
7 'DW 233
8 'LD A 27
9 'CALL 1168
10 'LD A 72
11 'JP 1168
12 '! START
13 'LD HL 46592
14 'LD DE 11416
15 'LD B 32
16 '! LOOP
17 'PUSH BC
18 'LD B 8
19 'LD (HL) E
20 'INC HL
21 'LD (HL) D
22 'INC DE
23 'INC HL
24 'DJNZ 249
25 'EX DE HL
26 'LD BC 352
27 'ADD HL BC
28 'EX DE HL
29 'POP BC
30 'DJNZ (LOOP)
31 'RET
```

(ROCHE> In standard Z-80 code, this gives:

```
$$$$
)
```

First of all, we empty the screen by the equivalent of PRINT CHR$(27)+"E". The Mallard BASIC interpreter contains a routine to print the value in register A by a call at 1168. (ROCHE> That is to say: this code is not portable.) If you are not using Mallard BASIC, an equivalent is LD C 2, LD E value, CALL 5, which uses a BDOS routine (ROCHE> BDOS function 2: CONOUT.). The ! in line 8 is a convention to signal a label at this point in the code which is interpreted in line 5. DW is followed by a number which is entered in two bytes into the code. Two bytes are often called a word, so DW stands for Define Word. (ROCHE> A common assembler pseudo-op, since 1972...) We call SCR_RUN, and the roller RAM is reset by some fairly standard code. After the return from SCR_RUN, the cursor is sent to the top left. As the screen is empty during the reset, nothing odd appears to happen! (ROCHE> Again, this explanation is not simple.)

C11P1

```
100 h = HIMEM : j = h - 56 : k = INT (j / 256) : i = j - k * 256 :  IF
```

```
i > 227 THEN i = 227 : j = i + k * 256
        105 MEMORY j - 1
        110  POKE i, 62, 27, 205, 144, 4, 62, 69, 205, 144, 4, 1, i +  28,  k,
205, 90, 252
        120 POKE j + 16, 233, 0, 62, 27, 205, 144, 4, 62, 72, 195, 144, 4, 33,
0, 182, 17
        130 POKE j + 32, 152, 44, 6, 32, 197, 6, 8, 115, 35, 114, 19, 35,  16,
249, 235, 1
        140 POKE j + 48, 96, 1, 9, 235, 193, 16, 237, 201
```

(ROCHE> In standard Z-80 code, this gives:

```
$$$$
)
```

This is the same code in a Mallard BASIC routine that will place itself in the most economical position. The rest of your program can follow. It can include CALL j, whenever you want a screen clear.

11.7 A little frivolity!

When we were at school, we all considered the possibility of making a multiple pen that would write many of our lines at the same time! I never saw one that actually worked. This one does! Roller RAM is set so that each line on the screen has the same values in the roller. This means that, if you write it once, you write it 32 times.

```
        C11P2
        500 s = j : e$ = CHR$ (27)
        510 CALL s : h = HIMEM : MEMORY 51199!
        520 j = 51200! : POKE j, 1, 9, 200, 205, 90, 252, 233, 0, 201, 33,  0,
182, 17, 152, 44, 6, 8, 115, 35, 114, 19, 35, 16, 249, 62, 184, 188, 200, 195,
12, 200
        530 DATA I must, not, rite, wrude, words, about, my, teacher, on, the,
lavatory, walls
        540  m = 400 : PRINT e$ "Y $"; : FOR n = 1 TO 12: READ a$ : PRINT  a$;
" ";  :  CALL j : GOSUB 550 : NEXT : PRINT e$ "f"; : m = 4000 :  GOSUB  550  :
PRINT e$ "Y "; : PRINT SPACE$ (90) : PRINT e$ "e"; : CALL s : MEMORY h : END
        550 FOR nn = 1 TO m : NEXT : RETURN
```

(ROCHE> This gives:

```
$$$$
)
```

This program must be appended to the previous one. (If you have DWBAS or Lightning BASIC, you can avoid this by replacing CALL s with #c, as #c resets roller RAM.) It is almost worth making a syntax error, as it will be quite spectacular! You can get out of the mess and back to vague normality by pressing [RETURN] and then entering CALL s.

11.8 Plotting a pixel

The fundamental graphics routine is to plot a single pixel. Many readers will have seen routines that do this, although probably not such a short one. This one does assume roller RAM to be reset, and so the calculations are slightly less severe. CALL j(x%,y%) will plot a single pixel x% pixels from the left,

and y% pixels from the top.

```
        1 'LD A (DE)
        2 'LD E (HL)
        3 'INC HL
        4 'LD D (HL)
        5 'LD HL 64816
        6 'ADD HL DE
        7 'RET C
        8 'LD BC START
        9 'CALL 64602
        10 'DW 233
        11 'RET
        12 '! START
        13 'LD C A
        14 'LD HL 22112
        15 'ADD HL DE
        16 'LD A L
        17 'AND 248
        18 'LD B A
        19 'LD A C
        20 'AND 7
        21 'OR B
        22 'LD L A
        23 'LD A C
        24 'RRA
        25 'RRA
        26 'RRA
        27 'AND 31
        28 'LD B A
        29 'INC B
        30 'LD A E
        31 'AND 7
        32 'LD DE 720
        33 'ADD HL DE
        34 'DJNZ 253
        35 'LD B A
        36 'INC B
        37 'XOR A
        38 'SCF
        39 'RRA
        40 'DJNZ 253
        41 'OR (HL)
        42 'LD (HL) A
        43 'RET
```

(ROCHE> In standard Z-80 code, this gives:

```
$$$$
)
```

Lines 1-4 put y% into register A, and x% into register pair DE. The value of x% is then checked but, as only the low byte of y% is needed, this will always be valid. SCR_RUN is then called.

The initial value of HL is the top of the screen - 720. This slight oddity is so that register B must always contain a value greater than 0 in the first DJNZ loop. In lines 15-22, x% is added to register pair HL, to adjust for the column. The lowest three bits are discarded, and replaced by the lowest three bits of y%.

Lines 23-34 find the row number, and adjust register pair HL for this. As register pair DE is used in this routine, the lowest three bits of register E are saved in register A.

Lines 35-40 perform the operation A=2^A, so that A contains the binary pattern of the pixels to be set. The OR (HL) in line 41 could be replaced by XOR (HL), which would toggle the pixel. More usefully, line 41 could be replaced by two lines: CPL, AND (HL), which would turn off the pixel.

C11P3

```
100 h = HIMEM : j = h - 60 : k = INT (j / 256) : i = j - k * 256 : IF i > 236 THEN i = 236 : j = k * 256 + i
110 MEMORY j - 1
120 POKE j, 26, 94, 35, 86, 33, 48, 253, 25, 216, 1, i + 18, k, 205, 90, 252, 233, 0, 201
130 POKE j + 18, 79, 33, 96, 86, 25, 125, 230, 248, 71, 121, 230, 7, 176, 111, 121
140 POKE j + 33, 31, 31, 31, 230, 31, 71, 4, 123, 230, 7, 17, 208, 2, 25, 16
150 POKE j + 48, 253, 71, 4, 175, 55, 31, 16, 253, 182, 119, 201
```

(ROCHE> In standard Z-80 code, this gives:

```
$$$$
)
```

This is the conversion of the code into Mallard BASIC. As you can see, it normally uses just 60 bytes, although you will have to include the roller RAM reset if your version of BASIC does not contain it.

11.9 Saving screens

Dr. Logo for the Amstrad PCW allows you to save a screen as a file. Mallard BASIC does not. However, it contains a random file facility. Random files are far more useful that most people realise. All we need to do is to code the screen bytes into strings of length 128 bytes, then read them into a random file using PUT. Later, this file can be read using GET, and the process reversed to load the screen that we have saved. The program below contains the two subroutines that we need.

C11P4

```
1000 f$ = "m:screen"
1010 h = HIMEM : k = INT ((h + 1) / 256 - 1) : j = k * 256 : MEMORY j - 1 : j = j + 128
1020 POKE j, 94, 35, 86, 1, 140, k, 205, 90, 252, 233, 0, 201
1030 POKE j + 12, 33, 0, k, 235, 1, 128, 0, 237, 176, 201
1040 OPEN "r", 1, f$, 128 : FIELD 1, 128 AS a$
1050 c$ = SPACE$ (128) : u = 22832 : FOR n = 1 TO 180
1060 u% = UNT (u) : u = u + 128 : CALL j (u%) : v = VARPTR (c$) : POKE v + 1, 0, k
1070 LSET a$ = c$ : PUT 1 : NEXT : CLOSE : MEMORY h : RETURN
1090 :
2000 f$ = "m:screen"
2010 h = HIMEM : k = INT ((h + 1) / 256 - 1) : j = k * 256 : MEMORY j - 1 : j = j + 128
2020 POKE j, 94, 35, 86, 1, 140, k, 205, 90, 252, 233, 0, 201
2030 POKE j + 12, 33, 0, k, 1, 128, 0, 237, 176, 201
2040 m = j + 30 : POKE m, 35, 94, 35, 86, 33, 0, k, 235, 1, 128, 0, 237, 176, 201
```

```
        2050 OPEN "r", 1, f$, 128 : FIELD 1, 128 AS a$
        2060 u = 22832 : FOR n = 1 TO 180 : GET 1
        2070 u% = UNT (u) : u = u + 128 : c$ = a$ : CALL m (c$) : CALL j (u%)
        2080 NEXT : CLOSE : MEMORY h : RETURN
```

(ROCHE> This gives:

$$$$
)

The filename and disc chosen at the start of the subroutines at 1000 and 2000 is obviously your own choice. The routine at 1020 takes 128 screen bytes and copies them into reserved memory. By using VARPTR, c$ is made to point to this string, which is then LSET to the file. The reverse process is similar. We use an extra piece of code to place the GET string where we want it. This is quicker than using ASC(MID$) and POKEs.

11.10 Altering the character set

This is another fundamental screen routine. The program below is a framework that can be used to generate new screen characters. The routine is a toggle, so that, when the program ends, the character set can be restored. The variables n and s are your choice. You can change up to 28 characters, but n+s must not exceed 255.

C11P5

```
        9 REM n = Number of characters, s = Start of altered characters
        10 n = 2 : s = 200
        99 REM There should be n lines of DATA, consisting of 8 binary
patterns
        100 DATA 102, 102, 126, 102, 102, 60, 24, 0
        110 DATA 252, 204, 204, 124, 204, 204, 252, 0
        1000 GOSUB 50000 : CALL j
        1001 REM Rest of program
        1002 PRINT CHR$ (200), CHR$ (201)
        49999 CALL j : MEMORY h : END
        50000 r = n * 8 : v = 47104! + s * 8 : a = INT (v / 256) : b = v - a *
256
        50010 h = HIMEM : k = INT ((h + 1) / 256 - 1) : j = k * 256 : MEMORY j
- 1
        50020 FOR m = 1 TO r : READ x : POKE j + 31 + m, x : NEXT
        50030 POKE j, 1, 9, k, 205, 90, 252, 233, 0, 201
        50040 POKE j + 9, 33, b, a, 17, 32, k, 6, r, 26, 78, 119, 121, 18, 35,
19, 16, 247, 201
        50050 RETURN
```

(ROCHE> This gives:

$$$$
)

The characters in the DATA lines 100 and 110 are eccentric versions of A and B, as you will see when you run the program.

11.11 Further graphics

This completes what I consider the fundamental routines. It leaves you many

challenges. Let us just consider one or two. Some attractive effects can be made by using a background of dark green by setting half the pixels. A binary pattern of 85 or 170 in the part of the screen used will set alternate pixels.

Constructing a routine to draw a line can be done from Mallard BASIC with our pixel plotting routine. An all-code routine will be much quicker. The problem is that you have to consider which coordinate will be incremented the most quickly, which direction the line goes, and find coding which can be used with 3-byte numbers, as you will need this amount of accuracy.

Circles are another challenge. I have found that the best method is to consider them as multi-sided polygons, and copy a rudimentaty form of the SINE tables into memory. Fills are never easy if they are to cater for irregular shapes. It may even be better to recourse to a program that is partially in high-level language.

You now have the tools -- use them well!

Chapter 12: Sound

At the end of this short chapter, you will doubtless come to one of three conclusions:

1. The Amstrad PCW is not very musical.
2. You author is not very musical.
3. Both.

There was a time, many moons ago, when I considered that I had almost perfect pitch. I could play the first two movements of the "Moonlight Sonata" passably horribly. (Even the cat ran out when I tried the third.) I cannnot claim this now -- perhaps it is the combination of background music and "Space Invaders" that makes me appreciate that no music is infinitely preferable to bad music!

The principle of producing notes from the beeper is quite simple in theory. You turn it on and off many times. The pitch of the note depends on the time interval taken for the switch to operate. If you double this time, the note should be an octave lower. In practice, it is not quite so easy. If you try a loop increasing the interval with each pass, successive notes should go lower and lower. This does not always happen. I don't know why -- I am not a physicist. Probably, it is something to do with the natural frequency of the beeper. What we need to find is a series of intervals throughout which the pitch does appear to be working in practice reasonably in accord with theory. I spent most of a morning thrying this, and that is quite enough! If you are sufficiently interested to spend longer, you will probably do better.

As we showed in Chapter 9, it is possible to produce a slightly different note by OUT 248,11 : OUT 248,12 in Mallard BASIC, but the problem with BASIC is that the time interval to execute even this line is much longer than the sort of intervals that we need. (ROCHE> BASIC, including Mallard, is 10 to 50 times slower than assembly.) Hence, it is necessary to invoke machine code for the delay loops between the 'ons' and 'offs'. The program that we will use is a good example of the power of Mallard BASIC to harness code where needed, and yet do most of the hard work itself.

12.1 If music be the food of love...

C12P1

```
        1 GOSUB 200 : GOSUB 400 : t = 1
        10 m$ = "1h3a2f0r1h3a2f1rc2f0r1a2h1a2a#1r" : GOSUB 100
        11 m$ = "1g3a#2g0r1h3a2f0r1a2g1d2e1c4f" : GOSUB 100
        99 MEMORY h : END
        100 l = LEN (m$) : FOR n = 1 TO l : s = ASC (MID$ (m$, n)) : fl = 0
        105 IF n <> l THEN IF ASC (MID$ (m$, n + 1)) = 35 THEN fl = 1
        110 IF s = 48 THEN t = 0.5 : GOTO 160
        120 IF s > 48 AND s < 58 THEN t = s - 48 : GOTO 160
        130 s = s AND 223 : IF s = 82 THEN s = 67 : POKE i + 6, 12
        140 IF s < 64 OR s > 72 THEN 160
        150 k = c (s - 64) - fl : GOSUB 500 : POKE i + 28, a (k) : GOSUB 300
        160 POKE i + 6, 11 : NEXT : RETURN
        200 h = HIMEM : j = INT ((h + 1) / 256) - 1 : i = j * 256 : MEMORY i -
1
        210 POKE i, 118, 243, 17, 100, 0, 62, 11, 205, 25, j
        211 POKE i + 10, 60, 205, 25, j, 27, 123, 178, 32, 242, 251, 201
        220 POKE i + 25, 211, 248, 6, 30, 14, 3, 13, 32, 253, 16, 249, 201
        230 RETURN
        300 CALL i : FOR m = 1 TO 50 : NEXT : RETURN
        400 DIM a (13), b (13), c (8)
        410 b = 2^(1 / 12) : a = 60 / b
        411 FOR n = 0 TO 13 : a (n) = ROUND (a) : b (n) = 3000 / a : a = a * b
: NEXT
        420 DATA 4, 2, 13, 11, 9, 8, 6, 1
        430 FOR n = 1 TO 8 : READ c (n) : NEXT : RETURN
        500  f = ROUND (b (k) * t) : g = INT (f / 256) : POKE i + 3, f -  g  *
256, g : RETURN
```

(ROCHE> This gives:

```
$$$$
)
```

The code is contained in the subroutine at 200. The delay loop (line 220) is made by counting down B and C, which are set to the right values by POKEing elsewhere. Lines 210 and 211 form the main loop, which switches on and off. DE is set as the repeat counter. Clearly, the shorter the interval, the more times the loop must be executed to produce the same length of code.

The subroutine at 400 enters the values for the registers corresponding to the notes to be played. Intervals change on a logarithmic scale, and there are 12 semitones to an octave. The interval must be increased by a factor of the twelfth root of 2 for every semitone lower. The data in line 420 corresponds to the letter positions of the musical notation, in the scale of C Major. The subroutine at 500 is used for a double POKE of the correct values into DE.

The subroutine at 100 plays a musical string of notes passed by m$. The notation is that the letters A-G (a-g) stand for the pitch of the note to be played. The numbers are a signal for the length of the note. 1 is the standard, and 3 is three times as long. Using 0 will give a note half as long as the standard length. If no number appears before a note, the tempo will be unchanged from the previous note. If no numbers are entered at all, the tempo will be 1. H or h is used for high C, so that a complete octave is within range. R or r is used for a rest, the length is equal to the current tempo setting. # is used, following a note, to indicate that the sharp is required. There are no flats. However, you don't need them. B flat is exactly the same as A sharp.

The lines before 100 contain the program. The two strings play what (in an optimist moment) I think you may recognise as the theme tune from Beethoven's Pastoral symphony. The more musical readers will doubtless be able to improve

this!

You don't, of course, have to play notes on a musical scale to achieve sound effects. Try some yourselves -- if you live deep in the country, and the resident owls are tone deaf!

WARNING: I have been told by a physicist that the electronic vibrator (or whatever it is called) may not be the same in all machines. Hence, it is possible that, if you try this program, the result will be totally different on your computer!

Chapter 13: BASIC and CP/M

Unless you are interested in learning machine code, we suggest that you ignore this chapter at a first reading.

In the introduction, readers were told that, if they did not want to write their own programs in code, but just copy any routines given to achieve the same effect, that was fine. If that is your view, leave this chapter alone -- come back later, if you like! CP/M Plus is certainly part of our tour to places where few Mallard BASIC programmers tread, and it has treasures for the low-level programmer, but -- to be honest -- not many of these treasures will be appreciated by the programmer who wants to operate solely in high-level language.

When you have loaded Mallard BASIC, you have two libraries of code subroutines at your disposal, if you know where to find them. There are many routines in the Mallard BASIC interpreter that can be called by your own program. (ROCHE> Your programs become then version and CPU-dependent. Mallard BASIC has the VERSION command, allowing programs to know under which CPU they are running. But you then need to keep addresses valid for all the CPU versions Mallard BASIC was produced... It can be done but, fundamentally, it is simpler to stick to high-level commands.) There are also the BDOS functions in CP/M Plus. Indeed, the Mallard BASIC interpreter itself makes use of many of these functions. In this chapter, we shall first of all look at some of the more useful BDOS functions. There are about 70 of them, and it is beyond the scope of this book to detail all of them. If you want to program seriously in CP/M Plus, fuller documentation is available. (ROCHE> "The Amstrad CP/M Plus", MML Systems, 1989.) We shall, then, look at ways of making CP/M Plus and Mallard BASIC interact.

To avoid having to tell you elsewhere, the following abbreviations are commonly used -- so commonly that nobody actually tells you what they mean! (ROCHE> Simply open the "CP/M Plus Programmer's Guide".) I think that these definitions for abbreviations will give the idea -- even if they are not the ones the originators initially intended! (ROCHE> Why not follow the official documentation?)

```
CP/M -- Control Program for Microcomputers
BDOS -- Basic Disc Operating System
BIOS -- Basic Input/Output System
DMA  -- Disc Management Area
        (ROCHE> ??? It means "Direct Memory Access"!)
FCB  -- File Control Block
PFCB -- Parameters for File Control Block
DPB  -- Disc Parameter Block
X*   -- Extended something or other!
```

13.1 Input and output

To use a BDOS function, it is essential to load register C with the number of the function, and follow it by CALL 5. (ROCHE> Why not simply explain that, in high-level languages, you write CALL BDOS (Pstring, "Hello, World!") (or PRINT "Hello, World!"), while in assembler you fill the registers with the parameters, then call BDOS via its entry point at 0005H? So, LD DE,address of string, LD C,Pstring, CALL BDOS. Notice that the order of the elements of an assembler call is the reverse of a high-level language.) You may need to set other registers in certain cases. (ROCHE> Simply read the "CP/M Plus Programmer's Guide".) Usually, A and HL will contain some information about the operation when it is completed -- maybe simply error information, with which we will not concern ourselves, here. Most registers will not be preserved, so the coding must do the necessary PUSHing and POPping. In this section, we shall look at BDOS system calls 0, 1, 2, 5, 9, and 10, which include the main input and output functions.

Function 0 (ROCHE> "System Reset".) is the way a CP/M Plus program returns to the A> prompt. It is the system reset. There are several ways of ending a CP/M Plus program, but all of them, eventually, call this function. You can LD C 0 and JP or CALL 5 (ROCHE> 0 is the number of the "System Reset" function of the BDOS, and 5 (0005H) is the entry point in memory of the BDOS.). JP 0 takes you to the same address (ROCHE> 0 (0000H) is the entry point of the BIOS, but the first BIOS function happens to do the "System Reset" for the BDOS...). So does a RET, if the stack is managed correctly (ROCHE> It is safer to use LD C,SysReset; CALL BDOS.). A RST 0 also works (ROCHE> Since Geoffrey Childs did not explain the 916 opcodes of the Z-80 CPU in general, and the RST instructions in particular, only experienced assembly language programmers will understand this remark. RST ("restart") instructions are one-byte long calls, but to special addresses in the "Page Zero".)

Function 1 (ROCHE> "Console Input".) inputs a key (ROCHE> More precisely, the character outputted by a key of the keyboard.). It waits until a key is pressed. If you want the ASCII value, it is returned in register A. Function 2 (ROCHE> "Console Output".) is a single key output. Register E is loaded with the ASCII value, and the character is printed on the screen. Note that this function (and function 9) do not produce an automatic Carriage Return (ROCHE> And a Linefeed, that is to say: starting a new line after outputting the character/string.). If you want this, you must output ASCII 10 and 13 as well (ROCHE> 10 = Carriage Return, 13 = Linefeed.). Function 5 (ROCHE> "List Output". Here, "list" means "printer".) is similar to function 2, except that the output goes to the printer.

While any string of characters can be printed on the screen by successive use of function 2, it is easier to use function 9 (ROCHE> "Print String".). In this function, DE is set to the address in memory where the string begins. The string can contain normal ASCII characters, including control codes, but must not contain ASCII 36 (the $ sign), except at the end. This is the signal that the string has terminated. Function 10 (ROCHE> "Read Console Buffer".) inputs a string. We shall return to this function in a moment.

13.2 DMA and FCB

When using CP/M Plus to operate on files, there are two main data areas to consider. The DMA is a buffer of 128 bytes, which is a general workspace for the routines. The FCB is an area of (up to) 36 bytes, which contains the information about the file being used. It is usually convenient to have the DMA at 128-255 and the FCB at 92-127. If you are working from Mallard BASIC,

the relevant addresses may be different, so it is always wise to set the DMA first. This is done by setting DE to the start of the DMA, and using BDOS function 26 (ROCHE> "Set DMA Address".).

BDOS 10, the string input function, will normally use the DMA. DE is set to 0 to signal this (ROCHE> No: 0 means "use the DMA buffer", no matter where is is located in memory.). Before the function is called, it is necessary to set the first byte of DMA to the maximum length of the intput string, which should NOT exceed 126. The third byte should be set to 0. This is where the string will start when it has been stored in memory. The second byte of the DMA will contain the length of the string after the operation is complete. The string input works like INPUT in Mallard BASIC. Press [RETURN] when you have finished. As this function is considerably more complicated than the other input and output functions, we give an illustrative program. (ROCHE> Well, everything is complicated, if you don't quote the "Programmer's Guide"!)

C13P1

```
10 v = 51200! : MEMORY v - 1
20 POKE v, 17, 128, 0, 213, 14, 26, 205, 5, 0, 225
30 POKE v + 10, 54, 126, 35, 35, 54, 0, 17, 0, 0, 14, 10, 195, 5, 0
40 CALL v
50  PRINT : m = 130 : FOR n = 1 TO PEEK (129) : p = PEEK (m)  :  PRINT
CHR$ (p); : m = m + 1 : NEXT
```

(ROCHE> This gives:

```
$$$$
)
```

The code works like this:

Set DMA address, and save: LD DE 128, PUSH DE, LD C 26, CALL 5
Put 126 and 0 in first and third bytes of DMA: POP HL, LD (HL) 126, INC HL, INC HL, LD (HL) 0
Call the input string function, and return: LD DE 0, LD C 10, JP 5

I think that the best way to deal with FCBs is not to bother with details, but to show how an FCB is easily created by BDOS 152 (ROCHE> "Parse Filename".). This takes a filename string, and parses it into the correct syntax. The filename can included disc name followed by a colon (":"), it can include wildcars ("?" and "*"), and you can end it with a semicolon (";") followed by a password. The filename string should end with a zero when it is put into memory.

For this BDOS call, we need to create an extra 4-byte area called the PFCB. This contains the address of the filename string, followed by the address of the FCB. DE is loaded with the PFCB address before the function is called. The program below illustrates this.

C13P2

```
10 v = 51200! : MEMORY v - 1
20 INPUT "Enter filename: ", f$ : f$ = f$ + CHR$ (0) : l = LEN (f$)
30  FOR  n = 1 TO l : POKE 51299! + n, ASC (MID$ (f$, n))  :  NEXT   '
String at 51300
40 POKE 51250!, 100, 200, 92, 0  ' This is the PFCB
50 POKE v, 17, 50, 200, 14, 152, 195, 5, 0 : CALL v
60 FOR n = 92 TO 127 : PRINT PEEK (n); : NEXT
```

(ROCHE> This gives:

```
$$$$
```

)

13.3 Reading and writing files

Once the two ideas (FCB and DMA) are understood, reading and writing files are not difficult. All the new functions introduced in this section need DE to point to the start of the FCB, and it is wise to ensure that the DMA is set to start at 128. To read a file into memory, first make the FCB using BDOS function 152. You must not use wildcards in this application.

Now, open the file (BDOS function 15 (ROCHE> "Open File".)), and read sequential (BDOS function 20 (ROCHE> "Read Sequential".). The first record will be in the DMA, so use an LDIR to put it into memory where you require. The FCB will point to the next record, so the process can be repeated until the file is complete. Close the file by using BDOS function 16 (ROCHE> "Close File".).

To write a file, set up the FCB and start with function 22, "Make File". This will open the file (ROCHE> No: it will put the filename in the directory, but not a single byte in the file.). To write to the file, you must load the data into the DMA before calling "Write Sequential" (BDOS function 21). Once again, remember to close the file (BDOS function 16).

13.4 The file directory

An FCB can also be used to obtain directory details of a disc. DE should point to the FCB for both the functions below. BDOS function 17 (ROCHE> "Search For First".) searches the disc for the first matching file. If this is followed by BDOS function 18 (ROCHE> "Search For Next".), the next match will be found. BDOS function 18 can be repeated. In this case, the return in A from the call is of importance. A result of 255 means that no file (or further file) has been found. If A contains 0-3, this tells us where the details are stored in the DMA area. If the DMA start if 128, 0 means 128-159, 1 means 160-191, etc.

The program below gives a full directory if a filename *.* is entered. It can also give a partial directory by putting only some of the filename as a wildcard. Entering *.BAS will list only Mallard BASIC files. The record count is only accurate as LOF in Mallard BASIC. It would need a more complex program to give a correct reading for files over 128 records long (16K). The lines up to 50 are the same as the previous program, apart from the omitted REMs.

C13P3

```
10 v = 51200! : MEMORY v - 1
20 INPUT "Enter filename: ", f$ : f$ = f$ + CHR$ (0) : l = LEN (f$)
30  FOR  n = 1 TO l : POKE 51299! + n, ASC (MID$ (f$, n))  :  NEXT   '
String at 51300
40 POKE 51250!, 100, 200, 92, 0  ' This is the PFCB
50 POKE v, 17, 50, 200, 14, 152, 195, 5, 0 : CALL v
60 POKE v, 17, 128, 0, 14, 26, 195, 5, 0 : CALL v
70 POKE v, 17, 92, 0, 14, 17, 205, 5, 0, 50, 40, 200, 201
80 CALL v : GOSUB 100 : POKE 51204!, 18
90 CALL v : GOSUB 100 : GOTO 90
100 x = PEEK (51240!) : IF x = 255 THEN PRINT : END
110  y = 129 + x * 32 : FOR n = y TO y + 7 : PRINT CHR$ (PEEK (n));  :
NEXT : PRINT ".";
120 FOR n = y + 8 TO y + 10 : PRINT CHR$ (PEEK (n)); : NEXT
130 PRINT USING "###"; PEEK (y + 14); : PRINT " Records.", : RETURN
```

(ROCHE> This gives:

$$$$
)

13.5 Other BDOS calls

There are just three other BDOS functions that I want to mention here, coincidentally numbered 45, 46, and 47. Function 45 (ROCHE> "Set BDOS Error Mode".) sets the error mode. If you are a Mallard BASIC programmer, you will doubtless have suffered the irritation of being dumped unceremoniously in CP/M Plus after some disc error. Murphy's law makes it virtually certain that you had not saved your program for at least an hour! There are 3 different error modes that can be set by loading E with 0, 254, or 255, and calling this function. If you set E to 254 and call it from Mallard BASIC, your disc errors produce the usual gobble-de-gook, but then return you to Mallard BASIC (with luck)! This appears to be a considerable improvement. It may be simpler just to POKE (see Section 14.2).

BDOS function 46 (ROCHE> "Get Disk Free Space".) gives the disc free space. Set register E to the relevant drive: 0 for A, 1 for B, and 12 for M. Call the BDOS function, and then PEEK the first three bytes of the DMA. (PEEK (128) + 256 * PEEK (129 + 65536 * PEEK (130)) / 8 should give the free space in KiloBytes.

Function 47 (ROCHE> "Chain to Program".) chains a program. Register E should be set to 255. The new program name should be POKEd from 128 onwards, just as you would write it in, after the A> prompt. You must finish the string with a zero byte.

13.6 Writing CP/M files

Mallard BASIC can be used to produce a COM program, one that will run from CP/M Plus without Mallard BASIC being loaded. The technique is reasonably simple. The code is POKEd into memory, which is then transferred to string variables, and saved in a random file, which must be named with COM as the extension.

The file must be written so that it will start at location 256 (ROCHE> 0100H, the start of the TPA.) when it is eventually run and, naturally, all CALLs and JPs must refer to the eventual location, rather than the location where the code is actually written. A convenient way that I usually use is to set MEMORY 40255, and start POKEing at 40256. This means that all address references have a displacement of 40.000. A simple example will show the technique.

C13P4

```
10 MEMORY 40255! : FOR n = 40256! TO 40383! : POKE n, 0 : NEXT
20 a$ = "This is my first CP/M program." + "$" : l = LEN (a$)
30 FOR n = 1 TO l : POKE 40299! + n, ASC (MID$ (a$, n)) : NEXT
40 POKE 40256!, 17, 44, 1, 14, 9, 205, 5, 0, 195, 0, 0
50 FOR n = 0 TO 127 : b$ = b$ + CHR$ (PEEK (40256! + n)) : NEXT
60 OPEN "r", 1, "myprog.com", 128 : FIELD 1, 128 AS c$
70 LSET c$ = b$ : PUT 1 : CLOSE
```

13.7 Combining BASIC and CP/M Plus

Let us list the combinations of programs that you might want:

1. Run a CP/M Plus program, then a Mallard BASIC one.

2. Run a Mallard BASIC program, and chain a CP/M Plus program.

3. CP/M Plus, then Mallard BASIC, then CP/M Plus.

4. Mallard BASIC, then CP/M Plus, and return to the Mallard BASIC program.

1. One method for this is well known. Use a PROFILE.SUB file, and include the CP/M Plus program, followed by BASIC BASPROG. An alternative method is to end the CP/M Plus program with BDOS function 47, the chain function. The CP/M Plus program must put BASIC BASPROG (followed by a zero byte) starting at location 128.

2. A similar method can be used, here. This time, the command line is POKEd in to the correct locations, and then BDOS function 47 ("Chain to Program") is called from Mallard BASIC.

3. This needs a combination of the methods in 1 and 2.

4. And this is the $64.000 question! It certainly can be done in some circumstances. In the latest version of Lightning BASIC Plus, this facility is offered, but I have refused to guarantee it in all cases. My advice to you is not to bother with the end of this section -- unless you want to share my problems!

The technique is to save blocks 132-133-135 on the M disc, save the Stack Pointer in block 135, reset the stack in this block, and alter the JP address at 0 to jump to the new routine. (On the first use, it must do the system reset which is part of BDOS function 47, "Chain to Program". On the return from the CP/M Plus program, it must do something quite different.) The CP/M Plus program is then chained in the normal way. On return, blocks 132 and 133 are retrieved from the M disc, the original Stack Pointer is recalled, the old block 135 is restored, and a simple RET restores everything back to the original position in the Mallard BASIC program.

The problems arise with the CP/M Plus program. If this uses the same part of the M disc, we are in trouble. DISCKIT is an example. If it uses memory between 32768 and the end of our routine, something will probably be corrupted. However, the most usual problem is repairable.

In this book, I have used CALL 64602 for SCR_RUN and other BIOS calls. This works with all Amstrad PCWs. However, to cater for different versions of CP/M Plus, there is a 'safer' method. The address is calculated from what is contained at location 1 added to 87 (ROCHE> This is the method recommended by Locomotive Software.). The coding LD (HL) 1, LD DE 87, ADD HL DE,... will often be seen. There are two ways round the problem. The first is to patch every CP/M Plus program containing this. Replace LD (HL) 1 with LD HL 64515. The other method is to ensure that, 87 above the new start address in location 1, there is a JP 64602.

This does not quite solve the problem, as one or two CP/M Plus programs calculate other addresses from the system reset start. I have not found any alternative to patching the CP/M Plus program. That is as far as I have reached with this question. Perhaps you can think of something simpler or more effective! (ROCHE> Obvious: Read the CP/M Plus manuals! Use an RSX. ("Resident

System Extension"). See the manuals for more information about RSXs.)

Chapter 14: Useful addresses...

(and probably one or two useless ones, as well!)

In this chapter, we look at the Mallard BASIC interpreter, and discuss addresses of routines or tables that may be of use or of interest. Obviously, this is far from a complete disassembly. Readers may well have found other useful addresses. If a number is followed by square brackets -- e.g. 3753 [3752] for Syntax Error -- the bracketed address refers to Mallard BASIC Version 1.39. Where no brackets follow, the addresses are the same in 1.29 and 1.39. We shall list addresses in numerical order, not in order of importance. The second section of this chapter will contain some additional addresses in CP/M Plus that can be used by Mallard BASIC.

14.1 The Mallard interpreter

256. The start of the Mallard BASIC interpreter. Jumps to a housekeeping routine at 724. If this jump is changed to 406, a warm start can be obtained in certain circumstances.

259-262. Addresses of routines used with USR. (ROCHE> "Get_Integer" and "Return_Integer".)

263-405. Start up messages. See also 22466. From about 300 onwards, this area can be used safely to give space for code routines that can be written without requiring a reset of HIMEM.

288. Normally set to 0 as the end of part of the message. It is also used as a signal in editing. If set to a non-zero value, all unnecessary spaces will be edited out of the listing of a program.

406-. Warm start. This can be called to end a program without any error messages. It resets the stack, so it can be used to jump out of a code routine. A slight amendment of the code will produce a line for editing in errors other than Syntax error.

450. Produces a line for EDITing. HL should be set with the line number. If the line does not exist in the program in memory, an error will occur. This routine combines with AUTO (one interception is at 471), and it is possible to amend it for variations to the AUTO operation.

724. This sets up the default parameters for the system. The various routines called could be amended, so that (for example) the default LPRINT width could be changed.

964. If register A is set to 3 and this subroutine is called in a printing routine, the output goes to the printer and not to the screen. The default value is 3 for screen printing, and this is reset into 28446 [28612] at the end of each Mallard BASIC statement.

1018. Waits for a keypress, and returns the result in register A. See 1024.

1024. Returns the latest keypress in register A. If the routine has not been called since a keypress was last made, this value is returned, even if a key is not actually being pressed at the time of the call. 0 is returned if there

is no relevant keypress.

1125. Prints characters from HL to the screen. HL is set to point to a string which is terminated by 0 as a signal to end. This can be used with the routined at 964 to divert to the printer.

1168 (or 1144). Prints a single character contained in register A.

Following the PRINT routine, there are a few routines for setting special situations (e.g. WIDTH). This is followed by the program control routines: GOSUB-RETURN, FOR-NEXT, and WHILE-WEND. We shall not detail where all Mallard BASIC word routines are to be found, but we will show you that it is not too difficult to find a particular routine that you want to examine.

At 3753 [3752] comes the first of the main addresses to call if you want to produce an error situation. There is no reason why a machine code routine, called from Mallard BASIC, should not use the error codes already present. 3753 [3752] gives Syntax Error, 3757 [3756] Improper Argument, 3763 [3762] gives the error corresponding to the value in register A when the routine is called.

4022 [4021]. "Break in XX" routine.

4306. This is rather a fascinating routine giving error messages. A new 'ASCII type' code is built up for values over 128. These codes contain combinations of letters that build up part words and complete words used in the messages. If you wish to write programs that condense files by hashing, it is well worth a study of how Mallard BASIC does this. A CALL to 4306 with DE pointing to the string to be printed demonstrates this. 0 is the terminator of the string, and this little program shows the build up of the words from characters in the 128-223 range:

```
        C14P1
        10 MEMORY 51199! : POKE 51200!, 17, 0, 201, 205, 210, 16, 201
        20 m = 51456! : FOR n = 33 TO 223 : POKE m, n, 95 : m = m + 2 : NEXT :
POKE m, 0
        30 v = 51200! : CALL v : END
```

(ROCHE> Indented, this gives:

```
        10 MEMORY 51199!
        20 POKE 51200!, 17, 0, 201, 205, 210, 16, 201
        30 m = 51456!
        40 FOR n = 33 TO 223
        50     POKE m, n, 95
        60     m = m + 2
        70 NEXT
        80 POKE m, 0
        90 v = 51200! : CALL v
        100 END
```

)

(ROCHE> In standard Z-80 code, this gives:

```
$$$$
```

)

We now come to the expression calculating routines. For all of these, HL points to the start of the expression AS IT IS IN A MALLARD BASIC PROGRAM, and not in ASCII form. You can experiment to see how numbers are contained in memory by writing a first line of a program, and PEEKing from 31382 [31523] to

see the changes from the ASCII that appears on the screen. For instance, in the line: 1 n = 2.5 + 3.4, the "2.5" would be represented by 30 0 0 64 131, the 30 being a signal for a single precision Floating-Point variable, and the following 4 bytes will evaluate to 2.5.

4931. Evaluates to a whole number 0-255, and leaves the answer in register A. A decimal result will be rounded, but a number outside the range gives an error. This is one of the few routines that differs in Mallard BASIC 1.39. In 1.29, the result is echoed in register E, which is sometimes useful. In 1.39, the register DE is preserved if the routine is called.

4944. As above, but a 0 answer gives an error.

4977 [4974]. This time, the value can be in the range 0-65535, and the result will be contained in DE.

5106 [5103]. This is the most general routine, and the result will be contained at 29778 [29919] and the bytes following. The location 29777 [29918] contains a number which gives the expexted type of variable. This routine will achieve string arithmetic, as well as numeric. In this case, 29778 [29919] and 29779 [29920] will give a pointer to the 'VARPTR' of the result.

5724 [5721]. The function address table. This is rather more complex than the command address table, which we will encounter later. A command is generally coded in Mallard BASIC into a single number above 128. For example, PRINT is coded as 179. A function is coded into two numbers: the first is 255, and the second is the number of the function. The functions are numbered 1-46, and 99-127. The second group comes first in the table. So, the address given by PEEK (5724 [5721] + 256 * PEEK (5725 [5722]) would give you the address of the routine for function 99 (CONSOLIDATE). Function 1 (ABS) will have its address in 5784-5785 [5781-5782]. Many functions will already have their arguments contained in register HL by the time they arrive at their address.

8279 [8377] onwards. Investigation in this region shows the communication between Mallard BASIC and CP/M Plus. When calls are made here, error control passes to the CP/M Plus system. This is why you can suddenly receive a nearly incomprehensible message and a return to the A> prompt. These routines all use a BDOS call.

8356-8357 [8454-8455]. In CP/M Plus, USER allows you to change the default group on the disc. If you then enter Mallard BASIC, the group is changed back to group 0 (ROCHE> ???). You can do most disc operations in another group. SAVE"2:MYPROG" would save in group 2, for example. These two locations can be changed to alter the DEFAULT group, and subsequent operations will all go to the new default group, just like USER in CP/M Plus. They contain 25, 119. Replace by 62 and the number of the new default group you require. Recently, I discovered that an easier method is: OPTION FILES "2" (ROCHE> This paragraph is very confusing, because there is no explanation of the difference between the default user number under which Mallard BASIC was loaded, and the user number under which its files are saved/loaded. OPTION FILES change the later, not the user number where Mallard BASIC was loaded, which will still be same after exiting Mallard BASIC. See the "CP/M Plus User's Guide" for more information about the USER command.)

8783-8797 [29155-29166]. This is a jump table for direct entry to CP/M Plus. The jumps are made to SYSTEM RESET, SCAN KEYBOARD, SCAN AND WAIT FOR KEYPRESS, PRINT CHARACTER IN REGISTER C, LPRINT DITTO. The 1.39 table only contains the last four jumps. System Reset is dealt with slightly differently.

We now make a big jump. Much of the intermediate area is devoted to the function routines and, if your idea of a happy afternoon is finding out the

algorithm for calculating COS, I am sure that you will satisfy your deepest desires by finding where COS resides from the function table!

15750 [15816]. HL points to a variable in a Mallard BASIC program. If the routine is called, then on exit DE will contain VARPTR for that variable. This may not seem terribly useful, but it is a vital routine for adding new keywords to Mallard BASIC.

17240 [17311]. Input prompt character.

18165 [18213]. A useful routine, which checks whether HL does point to the character expected. If so, HL is incremented; otherwise, an error occurs. The format is to make the call, and the following byte is the ASCII code of the expected character. For those of you who are learning Z-80 code, the routine is an interesting (although fairly standard) use of the Stack Pointer.

18425 [18489] onwards. The address table for the commands. The first two bytes give the address of AUTO, which is represented by 128. The next two give DATA (129), etc.

18599 [18679]. This routine is used to convert a series of ASCII characters into the form normally used in Mallard BASIC program storage. The string will usually be in the buffer, starting at 28466 [28632].

19188 [19289]. This is the start of the LIST routine, which consists of four CALLs and a JP 406. If PUSH HL is inserted between the first and second calls and the JP 406 is replaced by POP HL and RET, it is permissible to use LIST as a program line. This obviously has a use in demonstration programs. Since it uses no more space, it is rather surprising that the code was not originally written in this way.

19820 [19966]. This forms the table of the reserved words allowed by Mallard BASIC. In the first Part of the book (Section 8.1), there is a program from which the construction of the table and the contents can be studied.

21020 [21198]. This will search a Mallard BASIC program for a particular line number. DE contains the number, and HL contains the address at which it resides in memory. The Carry flag is reset if no such line exists.

22466 [22644]. This contains the "Acorn Computers" message (in less obvious form in 1.39). Together with the beginning of the start up message, it forms an encrypting technique for protected Mallard BASIC programs. Each byte of the program is XORed with one byte from each message, and the result is saved. The XOR method of encryption is a useful trick, as the routine works as a toggle to restore to the original characters.

22540 [22711]. The start of the routine which checks whether a program is protected, and signals an error if various Mallard BASIC words are used in the protection racket. By POKEing 22540 with 201 before loading a protected Mallard BASIC program, the routine can be bypassed. Subsequently, it is a simple matter to reset the protect flag at 29628 [29769].

23087 [23258]. A useful routine that prints the value of HL on the screen. It can be used quite safely to demonstrate CALL. This routine uses this and the CALL at 21020 to find where a Mallard BASIC program line begins in memory.

C14P2

```
1000 INPUT "Enter line number: ", i : i% = UNT (i)
1010 h = HIMEM : v = h - 10 : MEMORy v - 1
1020 a = 28 : b = 47 : IF PEEK (5103) = 197 THEN a = 206 : b = 128
1030 POKE v, 94, 35, 86, 205, a, 82, 195, b, 90 : CALL v (i%) : MEMORY
```

h

(ROCHE> This gives:

$$$$
)

24164 [24330]. This is the PRINT routine. It has uses in code routines as Mallard BASIC variables can be printed by pointing HL to the variable in memory and CALLing this address.

BASIC.COM is a 28K program. Yet, the interpreter takes up 31K. The last 3K (roughly) is used for a stack, flags, workspace, and buffers. We will have a look at some of the more interesting parts of this area.

28117 [28224]. This is the AUTO information. There is a flag followed by the increment and the current line number.

28446 [28612] is the location below which the Mallard BASIC stack is made. Any error will restart the stack at this point. A curiosity is that, if you carelessly load in the screen RAM into area 241, you may see little dots forming on the screen while a code routine does the stacking!

28446 [28612] also starts a parameter area of 20 bytes which may be of interest.

28446 [28612]. Current output device: 3=screen, 2=printer.

28447 [28613]. Flag to some keyboard routines. Wait for keypress?

28448 [28614]. Hold last unused keypress (ASCII).

28449 [28615]. OPTION RUN/STOP flag.

28450 [28616]. OPTION TAB/NO TAB flag.

28451-28453 [28617-28619]. Width of screen (wordwrap, and ZONE limit included).

28454 [28620]. Current PRINT position on line.

28455 [28621]. Current LPRINT position on line.

28456 [28622]. WIDTH LPRINT.

28457 [28623]. FOR-NEXT nesting flag.

28458-28461 [28624-28627]. Single precision variable at start of FOR-NEXT.

28462-28463 [28628-28629]. Address of start of FOR-NEXT loop.

28464-28465 [28630-28631]. Address of start of WHILE-WEND loop.

28466 [28632] is the buffer into which you write when you type a Mallard BASIC line on the screen. The contents are held in ASCII until the [RETURN] key is pressed. The line is partially compiled into a workspace area (see below), and then entered into the current program.

29707-29708 [29848-29849] contains the current value of HIMEM.

29717 [29858] begins a list of addresses regarding the current program. The

addresses are: the workspace mentioned above, the start of the Mallard BASIC program, the end of the Mallard BASIC program, the start of variable storage, the start of the array storage and, lastly, the end of the array storage.

Much of the subsequent area is devoted to details of the disc, and includes buffers (each of 260 bytes) into which files can be read by the OPEN command.

14.2 CP/M Plus addresses

In this section, the square brackets signify that a change of address is needed if you are using J21CPM3.EMS, the version of CP/M Plus for the Amstrad PCW9512.

CP/M Plus uses the area 0-255 as a workspace (ROCHE> No: this is the "Page Zero" of memory.) and, normally, the area 62982-65535 as a system area (ROCHE> Isn't it the Resident BIOS? Where is the Resident BDOS?). The locations 64412-64511 are known as the SCB ("System Control Block"), although only of few of these have any practical application to a Mallard BASIC program (ROCHE> Hahaha! "System" stands for "CP/M Plus Operating System", not Mallard BASIC!). We start with three work area jumps:

0-2. This is a jump to the CP/M Plus warm boot, normally 64515 (gives the A> prompt, if you prefer).

5-7. A jump to the current start of CP/M Plus (ROCHE> The BDOS.), normally 62982. This address is used by all the BDOS calls. If locations 6 and 7 do NOT contain 6 and 246 respectively, there is a RSX ("Resident System Extension") loaded, and the Mallard BASIC area will be reduced. In this case, be clever with extreme caution! (ROCHE> Simple: *NEVER* POKE any routine in "Page Zero".) A point to be wary about is that, if you SUBMIT a PROFILE.SUB which ends with < run "prog", CP/M Plus will be extended downwards to cope with this (ROCHE> The SUBMIT RSX will be loaded in memory, below the BDOS.), and complications can arise. It is better to end your PROFILE.SUB file with the line: BASIC PROG, and let PROG chain any further programs.

56-58. This is the jump to interrupts. It appears that CP/M Plus does not, itself, use this jump much, but it is used more frequently in Mallard BASIC programs. Normally, the jump goes to 64929. (ROCHE> CP/M Plus for the Amstrad PCW uses "Z-80 Mode 1 interrupts". See the Zilog doc for more info. The Z/SID debugger for the Amstrad PCW uses RST 30/RST 6.)

(ROCHE> Some fields inside the "System Control Block" follow.)

64449. This normally contains 128. If this is changed to 64, all printing will go to the printer, instead of the screen. 192 will make printing go to screen and printer.

64455. Normally, contains 64. Changing to 128 would send LPRINTs to the screen, but the main use of this is for alternative printers. POKE it with 16 for a Centronics printer and, if you have an Amstrad PCW9512, use 8 to send the printing to a parallel printer.

64474. Can be read to find the current default disc drive. 0 for A, 1 for B, and 12 for M. POKE 64474,12 might be a alternative to OPTION FILES "M", but it could have dangers as the 'wrong' directory might be in the 'right' place, if you see what I mean!

64487. Error mode. Normally, 0. If POKEd with 254, CP/M Plus will produce an error message, but continue. In Mallard BASIC, the effect is that it returns

to BASIC, rather than the A> prompt. POKEing with 255 disables the error message as well, a dubious procedure, except in very special circumstances.

64500-64504. Date (2 bytes) and Time (Hours, Minutes, and Seconds).

64515. Start of (ROCHE> Resident BIOS.) jump table. Some programmers take this address from 1 and 2, and calculate jumps or calls from it (ROCHE> This is the standard CP/M way of doing, since it is portable.) I dislike this, although it may be necessary for a program that is intended to work on other machines, as well as the Amstrad PCW (ROCHE> There were at least a dozen of microcomputers running under CP/M Plus. Personally, I was using Mallard BASIC 1.39 on the Epson QX-10, when Amstrad PCW ruled.)

(ROCHE> Some fields inside the "Resident BIOS" follow.)

64602. USERF. Well, you knew that, didn't you? (ROCHE> No, since you did not give its name...)

64801 [64813]. Can be called to bring in a memory bank. Bank 0 consists of blocks 128, 129, 131, and 135. Bank 1 is 132, 133, 134, and 135, the normal TPA ("Transient Program Area"). Bank 2 is 128, 136, 131, and 135. Before calling, register A must be loaded with the bank number.

65002 [65014]. LPRINTs ASCII character in register C.

65007 [65019]. PRINTs ASCII character in register C.

65090 [65360]. Start of table of external devices.

65354 [65304] is start of XDPB for drive A, 65381 [65331] for drive B.

65413-65414 [65492-65943) will give the size of the M disc.

Chapter 15: Going places

In the previous chapter, we have had a close look at some of the locations that are accessible in the normal environment when Mallard BASIC is loaded. We have, of course, made one foray beyond this limitation when we discussed graphics. It is time, now, to increase the boundaries of our working area in a more general way. In this chapter, we shall consider how to PEEK and POKE the unpeekable and unpokeable! We shall look at USERF, the commonest route into the forbidden areas. We have already used one application of USERF, namely SCR_RUN, but there are many others.

This chapter will include two little but useful examples that do not quite fit anywhere else: a screen dump without using [PTR] and [EXTRA], and a routine to find the true position of the cursor on the screen -- POS often gives strange answers!

15.1 Accessing the unaccessible

>From this point on, we shall be using blocks that are not normally in accessible memory. This, however, is not really as big a problem as it seems. Mallard BASIC has 31K free space, which is plenty to load a disassembler such as the one in the Appendix, copy a complete 16K block, and still have room for most Mallard BASIC extensions.

I find it convenient to copy the block to locations 40000-56383. The little program below hardly stretches the ingenuity of the code programmer to the limit, but it will serve us well.

C15P1 (BLOCPROG)

```
1 MEMORY 39977!
2 POKE 39978!, 243, 126, 211, 240, 33, 0, 0, 1, 0, 64, 17, 64, 156
3 POKE 39991!, 0, 237, 176, 62, 132, 211, 240, 251, 201
4 INPUT "Block: "; a% : a% = a% AND 31 : a% = a% OR 128
5 v = 39978! : CALL v (a%)
```

(ROCHE> In standard Z-80 code, this gives:

```
$$$$
)
```

The input is entered with the block numbers we have used. For instance, to have a look at the main BIOS block, 128 is input. The rest of the input ensures that a number between 128 and 159 is entered. The 0 at the start of line 3 is useful, as we can make it into a toggle program, by replacing it with 235 (EX DE HL).

If we wish, we can run the program in the original form, do an odd POKE or two above 40000, change line 3 by replacing 0 with 235, run again, and the block will subsequently contain the amendment.

15.2 CALLing to BIOS

In Chapter 12, we used SCR_RUN, which is a BIOS call (ROCHE> In this section, each time Geoffrey Childs uses "BIOS", it means "Banked BIOS", not the "Resident BIOS"...). While this is probably the most used of what are termed the USERF calls, it is only one of many (ROCHE> Normally, BIOS functions only do one thing, but Amstrad added a set of BIOS functions in the Banked BIOS (See the Locomotive doc.), which are accessed through USERF, instead of a jump table.). The BIOS in the different versions of CP/M Plus available for the Amstrad PCW varies, and there are only a limited number of calls which are certain to work in ALL versions, even if they do work in the one that you are using. The main BIOS calls reside in a jump table going from addresses 128 to 233 in block 128. ALL THE CALLS IN THIS JUMP BLOCK WORK IN EVERY VERSION OF CP/M PLUS FOR THE AMSTRAD PCW. This does no stop you CALLing to other points in BIOS (block 128). Any call to BIOS can be made in a similar way to SCR_RUN -- CALL 64602, followed by a two-byte address (ROCHE> The address of the extended BIOS function called.).

15.3 Screen dump

This little idea uses a CALL to a non-standard (ROCHE> Banked BIOS.) address. It will work with CP/M Plus Version 1.4 (probably the commonest version for the Amstrad PCW), but may well prove disastrous with any other version, unless the address is changed. Suppose that you have written a program which plots graphs of functions. One of the options that you would like is a screen dump when the graph is completed. The snag is that it will be necessary to prompt a press of [PTR] and [EXTRA], and this will spoil the screen. Instead of this, we intercept BIOS once we have erased the prompt line. (Presumably, you would have disabled the cursor and the status line earlier in the program.)

C15P2

```
400 PRINT CHR$ (27) "Y=A"; "Press S for Screen dump.";
409 REM Cursor to Row 31, Column 33
410 z$ = UPPER$ (INPUT$ (1))
420 IF z$ = "S" THEN PRINT CHR$ (27) "Y=A"; SPACE$ (24); : GOSUB 900
430 REM Etc...
900 h = HIMEM : MEMORY h - 6 : POKE h - 5, 205, 90, 252, 114, 20, 201
910 v = h - 5 : CALL v : MEMORY h : RETURN
```

(ROCHE> In standard Z-80 code, this gives:

$$$$
)

15.4 Where is the cursor?

Many BASICs have a command or a function to give the cursor position on the screen. Mallard BASIC does not, but details of this must be contained at some point. This time, we can approach the problem in two ways. Either we find the relevant location in BIOS, or we use a standard call which returns the row and column in HL. The first method uses SCR_RUN as we need some code of our own, the second uses a call from the jump table which gives the coordinates in HL. The first method uses a non-standard address in BIOS. This means that it will ONLY work if CP/M Plus Version 1.4 is operating. The second method is preferable, since it uses a call to the standard (ROCHE> Banked.) BIOS jumpblock, and will thus work with any version of CP/M Plus for the Amstrad PCW.

C15P3

```
600 h = HIMEM : j = INT (h / 256) : k = 256 * j : MEMORY k - 1
610 POKE k, 1, 12, j, 205, 90, 252, 233, 0, 121, 18, 112, 201
620 POKE k + 12, 237, 75, 38, 40, 201
630 CALL k (r%, c%) : PRINT "Row" r% "Column" c% : MEMORY h : RETURN
```

(ROCHE> In standard Z-80 code, this gives:

$$$$
)

C15P4

```
600 h = HIMEM : j = INT (h / 256) : k = 256 * j : MEMORY k - 1
610 POKE k, 229, 213, 205, 90, 252, 191, 0
620 POKE k + 7, 68, 125, 209, 225, 18, 112, 201
630 CALL k (r%, c%) : PRINT "Row" r% "Column" c% : MEMORY h : RETURN
```

(ROCHE> In standard Z-80 code, this gives:

$$$$
)

Chapter 16: The keyboard

Computers are not like wives (or husbands). They do what you tell them. If you press the key next to [STOP], the Amstrad PCW will dutifully produce a 1. However, you could program it so that, if you pressed that key, you would get:

What a load of rubbish!

printed on the screen! Of course, you can do this by using SETKEYS.COM, but have you ever felt that you would like to change a key temporarily during a Mallard BASIC program?

Mallard BASIC does not have the facility to set a function key. This is, perhaps, an omission on Locomotive's part (ROCHE> No: historically, functions keys simply did not exist when BASIC (and CP/M) was created.), but it does not matter! In this chapter, we look at ways to alter the operation of the keys during a program or possibly for a programming session.

16.1 Keyboard information

The last sixteen bytes of block 131 contain the most recent information about the state of the keyboard: namely, which keys were depressed at the last keyboard scan. A scan of the keyboard is done 50 times a second (ROCHE> This happens to be the frequency of 220 Volt current, the electricity used in Europe.); in other words, once in every six interrupts. This may not appear to have great uses for the Mallard BASIC or CP/M Plus programmer, since there are easier methods to obtain details of a simple keypress.

However, the knowledge of this source of information may be useful if you wish for a combination of keypresses to mean something special. A familiar example of this is [SHIFT][EXTRA][EXIT], a useful facility, but not one that you want to activate accidentally!

The interrupt scan is not an ASCII scan, but uses the numbers of keys 00-80 as given in the manual (book 1). The first 11 bytes have bits set or reset, according to whether that key is being pressed. For example, [SHIFT] and Q (keys 21 and 67) would set the 11 bytes: 0 0 32 0 0 0 0 0 8 0 0 0. The bit set is the key number MOD 8. Key number 72 sets bit 7 of the 10th byte, and none of the other bits are relevant. The 11th byte contains information about keys 73-80. The last five bytes in this block of 16 refer to specific combinations, and include what appears to be a timer.

Block 128 contains two important sets of tables. Unlike the previous information, the addresses of the tables depend on the version. The 81 bytes 6016-6096 (in CP/M Plus Version 1.4 for the Amstrad PCW) contain the ASCII characters assigned to the 81 keys in their normal state (without [SHIFT], [EXTRA], or [ALT]). There are four more tables of 81 bytes that follow this one. They represent [SHIFT], [ALT], [SHIFT][ALT], and [EXTRA].

The table at 10358 in CP/M Plus Version 1.4 is the expansion key table; There are 32 possible expansion keys, given quasi-ASCII codes of 128-159. The table contains the length of the expansion string, followed by the ASCII codes of the string for each of the 32 possible keys. It is conventional not to set 159 to an expansion string, so that it can be used for a dead key.

16.2 Setting a key

Both these tables are more easily accessed through calls to the BIOS jump table. The code below might be part of some instructional program. The programmer wants the user to try out what has been learnt, and then choose to return to the same or the next section of the program. If the user is not an experienced computer user, it is preferable to do this by a single keypress. This is achieved by first setting two unused expansion keys to restart the program, and then activating [f1] and [f7] to use these strings.

```
        C16P1
        1500 h = HIMEM : j = INT ((h + 1) / 256) - 1 : k = 256 * j : MEMORY  k
- 1
        1510 POKE k, 6, 154, 14, 10, 33, 128, j, 205, 90, 252, 212, 0, 201
        1520 POKE k + 128, 71, 79, 84, 79, 32, 49, 48, 48, 48, 13
        1530 m = k + 64 : POKE m, 6, 154, 14, 2, 22, 3, 205, 90, 252, 215,  0,
201
        1540 CALL k : CALL m
        1550  POKE k + 1, 155 : POKE k + 133, 50 : POKE m + 1, 155 : POKE m  +
3, 77
        1560 CALL k : CALL m
        1570 PRINT "To return to program:: Press [f1] for previous section."
        1580 PRINT "Press [f7] for nect section."
        1590 END
```

(ROCHE> This gives:

```
$$$$
)
```

The code in 1510 has register B set to the expansion string number, register C to the length of string, and register pair HL to the address of the string in memory. 1520 has the string GOTO 1000 [RETURN]. 1530 sets a particular key. Register B has the ASCII code or the expansion string number. Register C contains the key number. Register D is set to which states apply. (1 for normal, 2 for shift, 4 for alt, 8 for shift-alt, and 16 for extra.) In this case, setting register D to 3 will activate both [f1] and [f2]. The variations set in 1550 will set the second key in a similar way, so that [f7] produces:

```
        GOTO 2000 [RETURN]
```

16.3 Grandpa

Grandpa is a dear old boy with all his wits about him. Unfortunately, his fingers are not as nimble as they were, and when he tries to type "Granny", it tends to go on the screen as "GGgrraaaannyyy". As you know, if you depress a key, and hold it down after the letter is typed, there is a relatively long pause before the letter repeats, and subsequently short pauses only between further repeats. The first pause is called the "start up delay", and is 30 keyboard scans (or just over half a second). The subsequent pauses are "repeat delays", and are timed at 2 keyboard scans. For Grandpa, we would like to time them at 80 and 10 scans, respectively.

```
        C16P2
        10 h = HIMEM : MEMORY h - 10 : v = h - 9
        20 POKE v, 38, 80, 46, 10, 205, 90, 252, 224, 0, 201 : CALL v
```

(ROCHE> This gives:

```
$$$$
)
```

Register H is set to the start up delay, and register L to the repeat delay.

There are other (ROCHE> Banked.) BIOS calls connected with the keyboard, but I think that those are the three that you may need with Mallard BASIC. Summarising them:

```
        Job                BIOS address        Registers set
```

| --- | ---------- | ---------- |
Expansion key	212 0	B, C, HL
Set ASCII key	215 0	B, C, D
Set keyspeed	224 0	H, L

Chapter 17: The 'M' disc

On the Amstrad PCW8256, there is 112K of memory that is not used by the system, on the Amstrad PCW8512 and PCW9512, there is 368K. This is usually known as the 'M' disc, as CP/M Plus and BIOS are designed so that this memory can be used in virtually the same way as a physical disc. Since the M disc is memory, access to it is normally quicker (and quieter) than to a physical disc. Hence, a good Mallard BASIC programmer will often transfer files to the M disc which need to be read or written during a program.

This means that many programmers simply think of this memory as another disc. A better approach is to think of it as memory which is USUALLY used for disc storage. It can be used for any other storage you like, with the proviso that the storage will be lost every time the computer is switched off.

If you wish to use part of this memory as a disc, and part of it for someone else, it is advisable to use the blocks near the end of this memory for your own nefarious purposes!

Calling in an M-disc block cannot really be done from Mallard BASIC alone. You could enter OUT 242,137 to put the first block of it into 32768-49151 but, before you could use it, an interrupt would occur, and you would be back where you started. It is necessary to disable interrupts, and this can only be done in code. It is a convenient feature that the last block of the Amstrad PCW8512 memory is numbered 159. If you use this number on the Amstrad PCW8256, the block does not exist, but the computer (ROCHE> Well, the gate array.) subtracts 16 and works as though you have called for block 143. BUT: be careful with blocks 144 to 152. You could write successfully to this on an Amstrad PCW8512, but the PCW8256 would overwrite a vital part of memory -- often with disastrous consequences!

17.1 Screen saving

I said that you could save ANYTHING to the M disc, but you are probably wondering exactly what! I could be infuriating and repeat... anything! Instead, we will take one example. We will save and recall the screen. This is quite a powerful technique. Screens can be saved as random (or sequential) files to disc, and this includes the M disc. Saving and recalling is quicker on the M disc, but it is snail's pace, compared to a direct screen save. When you run this program, you will hardly realise that the screen is turned off and on during the recall routine, as it is so quick!

C17P1

```
10 h = HIMEM : j = INT ((h + 1) / 256) - 1 : k = 256 * j : MEMORY k -
1
20 POKE k, 243, 62, 129, 211, 241, 60, 211, 242, 62, 159
30 POKE k + 10, 211, 240, 1, 0, 56, 17, 0, 0, 33, 0, 128, 237, 176
40 POKE k + 23, 61, 211, 240, 1, 208, 38, 17, 48, 21, 33, 48, 89
50 POKE k + 35, 237, 176, 62, 132, 211, 240, 60, 211, 241
60 POKE k + 44, 60, 211, 242, 251, 201
70 l = k + 64
80 POKE l, 243, 62, 8, 211, 248, 62, 129, 211, 241, 60, 211, 242,  62,
```

```
159
        90 POKE l + 14, 211, 240, 1, 0, 56, 33, 0, 0, 17, 0, 128, 237, 176
        100 POKE l + 27, 61, 211, 240, 1, 208, 38, 33, 48, 21, 17, 48, 89
        110 POKE l + 39, 237, 176, 62, 132, 211, 240, 60, 211, 241
        120 POKE l + 48, 60, 211, 242, 62, 7, 211, 248, 251, 201
```

(ROCHE> In standard Z-80 code, this gives:

```
$$$$
)
```

This code is put at the start of the program. A screen is then built up by any method you like, and CALL k is used when it is complete. The program continues and, at a later stage, you use CALL l. The first screen is recalled. You MUST NOT use CALL l before a CALL k, as roller RAM is also saved by the routine, and reading a fictitious roller table from the M disc would be disastrous.

As you can see, the two routines are very similar. Some 33s and 17s are swapped to reverse the direction of block loads. In the second routine, we also blank the screen during the operation, by the equivalent of OUT 248,8, and recall it with OUT 248,7.

Chapter 18: Direct disc access

I start with an admission. I am wearing L plates for this part of the journey. An admission, not an apology, not a confession. All programmers should be looking for new fields to conquer. This book has shown how Mallard BASIC can be adapted to operate in areas that used to be the sole province of CP/M Plus. Like most of us, for a long time I took the attitude: "I know how to use DISCKIT, there are CP/M Plus disc editors available: I will leave the discs to them!"

BIOS, however, does contain some disc access functions in the main (ROCHE> Banked.) jump table. If they exist, they are there to be explored. This chapter contains the results of two experiments. Firstly, there is a FORMAT program that runs from Mallard BASIC. Secondly, our very own Mallard BASIC disc editor. I have not, previously, seen any BASIC program that covers these areas.

18.1 Format

Why should you want a new FORMAT program? Two reasons. The first is that, if you are a bit muddle-headed like me, there will be times when you want to back up a Mallard BASIC program, and you find you have no empty disc formatted. Of course, you can go back to CP/M Plus and use DISCKIT, but it would be easier if you didn't have to leave Mallard BASIC.

The second reason is that, believe it or not, the Mallard BASIC program appears to be quicker! I am not quite sure why, but I put forward a plus and a minus. DISCKIT is a general purpose CP/M Plus program and, as such, will be checking for all sorts of versions and format types before it hits on the right thing to do. The second is that my program is probably much cruder in that it assumes that, if it can work, the disc is in good physical shape -- oh, well, they usually are!

We are going to attempt to write a program that will format an A disc on the Amstrad PCW8256 or PCW 8512. I have to apologise to Amstrad PCW9512 owners.

You may be able to modify it for your machine but, with L plates on, I refuse to go into streets that I do not know well. As a bonus, it appears that we can get an extra 9K on the disc! I am working (unsuccessfully, so far) on the idea that we might get even more. As this is an experimental program, don't rely on discs formatted in the new way if the material is vital, until you have given it a thorough trial. It seems to work, but you can't be sure that it will do so in every circumstance.

As you know, if an A disc contains a J??CPM3.EMS program, it will boot CP/M Plus from switch on. This doesn't happen by magic! The first SECTOR on the first TRACK on the disc contains the code to do this. Our first problem is to have this code available, to put on the formatted disc. While it can be obtained from DISCKIT, it is simpler just to use an already formatted disc and write it into a random file that we can use later. This is the idea of the program C18P1.

C18P1

```
10 h = HIMEM : MEMORY 51199! : OPTION FILES "A"
20 POKE 51200!, 1, 0, 0, 17, 0, 0, 33, 0, 201, 221, 33, 74, 255, 205,
90, 252, 134, 0, 201
30 v = 51200! : CALL v
40 OPEN "r", 1, "formhead", 128 : FIELD 1, 128 AS a$ : u = 51456!
50 FOR n = 0 TO 3 : b$ = "" : v = u + 128 * n : FOR m = v TO v + 127 :
b$ = b$ + CHR$ (PEEK (m))
60 NEXT : LSET a$ = b$ : PUT 1 : NEXT : CLOSE : MEMORY h
```

(ROCHE> Indented, this gives:

```
10 h = HIMEM
20 MEMORY 51199!
30 OPTION FILES "A"
40 POKE 51200!, 1, 0, 0, 17, 0, 0, 33, 0, 201, 221, 33, 74, 255, 205,
90, 252, 134, 0, 201
50 v = 51200! : CALL v
60 OPEN "r", 1, "formhead", 128
70 FIELD 1, 128 AS a$
80 u = 51456!
90 FOR n = 0 TO 3
100     b$ = ""
110     v = u + 128 * n
120     FOR m = v TO v + 127
130         b$ = b$ + CHR$ (PEEK (m))
140     NEXT
150     LSET a$ = b$
160     PUT 1
170 NEXT
180 CLOSE
190 MEMORY h
```

In standard Z-80 code, this gives:

```
$$$$
)
```

The coding in line 20, while fairly simple, contains some definitions that we shall need to use in this chapter. The call is made to the (ROCHE> Banked.) BIOS function DD_READ, which needs the following parameters: Register B is set to the BANK, which is normally 0 for a (ROCHE> Banked.) BIOS call. Register C is set to the UNIT, which effectively means the drive: O for drive A, and 1 for drive B. Register D is set to the TRACK (0), and register E to the LOGICAL SECTOR (0). HL is an address in common memory to which the 512 bytes in the

sector will be written. Register pair IX is the start of what is known as the XDPB ("Extended Disc Parameter Block"). The XDPB is a section of 27 bytes which gives various details about the disc drive. The XDPBs for drives A, B, and M start at 65354, 65381, and 65408, respectively. (On the Amstrd PCW9512, the locations are 65304, 65331, and 65487.)

Once the disc has been read into memory, the rest of the coding simply puts it into strings, and saves it as a normal random file, which we call FORMHEAD. Once the program has run once, and the file is saved, there is no further use for it.

Let us go into the idea of tracks and sectors more carefully, as this is a source of confusion. So much so that it took me a day to debug the next program, although it only meant changing one number! A disc is divided into TRACKS, which are further subdivided into SECTORS. In the case of the A disc, there are usually 40 tracks, each containing 9 sectors. Each sector contains 512 bytes (half a kilobyte). Tracks and sectors can be PHYSICAL or LOGICAL. In the case of tracks, it makes no difference, the A disc tracks are counted from 0-39. Sectors are a different matter. Logical sectors are counted from 0-8, while physical sectors are counted from 1-9. Just to make matters worse, (ROCHE> The Banked.) BIOS sometimes uses logical sectors, and sometimes physical... It is not very logical, but it made me physically angry!

This is the main program, which we shall call FORMAT. It should be on the same disc as the file FORMHEAD, which you have just created.

C18P2

```
10 DIM a$ (3) : OPEN "r", 1, "formhead", 128 : FIELD 1, 128 AS a$
20 FOR n = 0 TO 3 : GET 1 : a$ (n) = a$ : NEXT
30 CLOSE : PRINT "Press A when disc to format is in drive A."
40 p = PEEK (64474!) : z$ = UPPER$ (INPUT$ (1)) : z = ASC (z$) : IF z
<> 65 THEN 30
50 OPTION FILES "a" : e = 74 : w = 65372! : t = PEEK (w) - 1 : s =
PEEK (w + 1) - 1
60 h = HIMEM : j = 192 : k = j * 256 : MEMORY k - 1
70 FOR n = 1 TO s + 1 : POKE k + 72 + n * 4, 0, 0, n, 2 : NEXT
80 GOSUB 260
90 POKE k, 243, 62, 128, 211, 240, 58, 1, 0, 50, 255, 192, 254, 58,
32, 5
100 POKE k + 15, 62, 42, 50, 22, 13, 62, 132, 211, 240, 251, 201 :
CALL k
110 cv = (PEEK (49407!) = 58)
120 IF cv THEN POKE 65372!, 42 : POKE 65359!, 183 : t = 41
130 POKE k, 221, 33, 74, 255, 62, 0, 205, 90, 252, 149, 0, 201 : CALL
k
140 POKE k, 86, 30, 229, 1, 0, 1, 33, 76, j, 221, 33, e, 255, 205, 90,
252, 143, 0, 201
150 POKE k + 20, 1, 0, 1, 17, 0, 0, 33, 0, j + 1, 221, 33, e, 255,
205, 90, 252, 137, 0, 201
160 POKE k + 39, 213, 94, 35, 86, 225, 35, 78, 35, 102, 105, 1, 128,
0, 237, 176, 201
170 PRINT CHR$ (27) "E" CHR$ (27) "H" CHR$ (27) "f" : FOR a% = 0 TO t
: FOR n = 0 TO s
180 POKE k + 76 + n * 4, a% : NEXT : CALL k (a%) : PRINT CHR$ (27) "Y
"; a% : NEXT
190 v1 = k + 20 : v2 = k + 39 : b% = k - 65280! : FOR n = 0 TO 3 : c$
= a$ (n) : CALL v2 (b%, c$)
200 b% = b% + 128 : NEXT : POKE k + 258, 40 - 2 * cv : CALL v1
210 PRINT "Format completed." : MEMORY h
220 OPTION FILES CHR$ (65 + p) : PRINT "Drive is "; CHR$ (65 + p) CHR$
(27) "e" : END
```

```
        230  PRINT f$ "on disc. Already formatted. Press C to carry on,  S  to
stop."
        240 x$ = UPPER (INPUT$ (1))
        250 IF x$ = "C" THEN RETURN : ELSE IF x$ = "S" THEN END : ELSE 240
        260 POKE k, 1, 0, 1, 17, 2, 2, 33, 0, 193, 221
        270 POKE k + 10, 33, 74, 255, 205, 90, 252, 134, 0, 210, 173, 14, 201
        280 ON ERROR GOTO 310 : CALL k : ON ERROR GOTO 0 : f$ = FIND$ ("*.*")
        290 IF f$ <> "" THEN GOSUB 230
        300 RETURN
        310 RESUME 300
```

(ROCHE> This gives:

```
$$$$
)
```

We start the program with the disc containing FORMHEAD in the drive. On the Amstrad PCW8512, this could be drive B, if you like. The program pauses until you signal that the disc to be formatted is in drive A. The next process is the subroutine at 260, which is omitted in DISCKIT (possibly to some users' deep regreat), is to check whether the disc is already formatted. It does this by attempting to read a sector and, if there is an error, everything is OK! If there is no error, there is trouble, and the program prompts the user with instructions to carry on or stop.

Lines 90-120 access (ROCHE> Banked.) BIOS block 128. If it finds it is reading CP/M Plus Version 1.4, it makes a change to allow 42 tracks, instead of 42. A location in common memory is POKEd with 58 if, and only if, the version is 1.4. This is tested on return to Mallard BASIC, and other amendments to the XDPB are made if we can have 42 tracks.

Line 130 is probably usually unnecessary. It just tells the machine that we want format type 0, the usual A format. Line 140 is the (ROCHE> Banked). BIOS format routine. Register pair BC is bank and unit, again. Register D contains the track number. Register E the filler byte, 229. (Goodness knows why this number was chosen (ROCHE> As far as I know, it was IBM who chose, circa 1972, this value as an "empty" flag. Legend has it that this value was chosen because, in binary, it has an easy (?) value (11100101B) to note.) but, if you study disc information, 229 is the signal for an erased file (ROCHE> E5H.).) Register pair HL points to the 36-byte information section that we fill in line 180. Register pair IX points to the XDPB.

Line 150 is (ROCHE> Banked BIOS routine.) DD_WRITE. The parameters are the same as DD_READ (ROCHE> How interesting!). Line 160 is code to move the array we made with FORMHEAD into memory, from which it can be read.

The information that we use in line 180 is track, head, sector, size, for each of the nine sectors on the track. Don't ask me what head (ROCHE> Open a dead hard disk, and you will understand.) means, 0 seems to work for it! The 2 for the size means 512 bytes.

And that's it. We have made the screen look just like DISCKIT does. Plagiarism!

18.2 Disc editor

In a sense, the program that follows illustrates better than words what the second part of the book is all about. Mallard BASIC is used as the working medium, with just two exceptions. The first is to provide the code to read and

write to the disc, using (ROCHE> Banked BIOS routines.) DD_READ and DD_WRITE, routines that we have already met. The second is to achieve the screen information from the bytes read, a process that would be perfectly practical in Mallard BASIC, but very much faster in code. The other feature is that this program would be impossible to write for most computers! The Amstrad PCW's 90 by 32 screen is used (and needed) to the full.

There is no urgent need to use memory sparingly, so we have not bothered with a relocatable program. The block 51200-51455 is used for the (ROCHE> Banked.) BIOS routines. 51456-51711 has the screen write routines. 51712-52223 is the 512-byte block to which a sector is read. Above this, we leave space for two strings to be created, into which the screen information will be written.

C18P3

```
10 GOSUB 1000
20 d$ = c$ + "f" : e$ = c$ + "e" : PRINT
21 PRINT d$; " Which drive? (A/B). Or press H for help."
25 GOSUB 3000 : IF z = 72 THEN GOSUB 2500 : PRINT : GOTO 20
30 uu = ASC (z$) - 65 : IF uu < 0 OR uu > 1 THEN 25
40  PRINT e$ : INPUT " Enter track number and press [RETURN]: ", t$  :
t% = VAL (t$)
50  INPUT " Enter sector number and press [RETURN]: ", s$ : s%  =  VAL
(s$)
55  PRINT d$; : tm = PEEK (65372! + uu * 27) * (uu + 1) : sm  =  (PEEK
(65373! + uu * 27) - 1)
60 k = 51200! : POKE k, 1, uu, 1, 17, s%, t%, 33, 0, 202, 221, 33,  74
+ uu * 27, 255
61 POKE k + 13, 205, 90, 252, 134, 0, 210, 173, 14, 201
70 g = k + 30 : POKE g, 1, uu, 1, 17, s%, t%, 33, 0, 202, 221, 33,  74
+ uu * 27, 255
71 POKE g + 13, 205, 90, 252, 137, 0, 210, 173, 14, 201
80 ON ERROR GOTO 3010 : CALL k : ON ERROR GOTO 0
90 PRINT c$ "E" c$ "H" : GOSUB 1500
100 GOSUB 2000
110 i$ = "TKAUHQ+-" + CHR$ (22) + CHR$ (28) + CHR$ (4)
120 GOSUB 3000 : i = INSTR (i$, z$) : IF i = 0 THEN 120
125 IF (i = 1 OR i > 6) AND uf THEN GOSUB 3040 : GOTO 120
126 IF i = 2 OR i = 3 THEN uf = 1
130 ON i GOSUB 150, 200, 250, 300, 350, 400, 450, 500, 550 : GOTO 120
140 ON ERROR GOTO 0 : GOTO 120
150 GOSUB 1604 : GOSUB 1610
152 PRINT FN a$ (3, 76) e$; : INPUT t$ : t% = VAL (t$) : PRINT
154  PRINT FN a$ (6, 76); : INPUT s$ : PRINT : s% = VAL (s$)  :  PRINT
d$;
156 GOSUB 800 : RETURN
200  GOSUB  700 : GOSUB 720 : GOSUB 1602 : z$ = INPUT$ (1) : q  =  ASC
(z$)
205 GOSUB 1600 : GOSUB 900 : GOSUB 1602 : RETURN
250 GOSUB 700 : GOSUB 710 : GOSUB 720 : GOSUB 1600 : GOSUB 900 : GOSUB
1602 : RETURN
300 GOSUB 1600 : uf = 0 : CALL g : GOSUB 1602 : RETURN
350  GOSUB  2500  : PRINT : PRINT TAB (30) "Press a  key  to  run  the
program."
351 GOSUB 3000 : RUN
400 PRINT c$ "e" c$ "E" c$ "H" : MEMORY hm : END
450 s% = s% + 1 : IF s% > sm THEN s% = 0 : t% = t% + 1
452 GOTO 800
500 s% = s% - 1 : IF s% < 0 THEN s% = sm : t% = t% - 1
502 GOTO 800
550 GOSUB 700 : GOSUB 720 : GOSUB 1602 : z = 0
552 WHILE z <> 13 : q$ = "&H" : FOR n = 1 TO 2
```

```
        554 GOSUB 3000 : IF z = 13 THEN 560
        555 IF z < 48 OR (z > 57 AND z < 65) OR z > 70 THEN 554
        556 q$ = q$ + z$ : NEXT : GOSUB 1600 : q = VAL (q$) : GOSUB 900 : b  =
b + 1
        558 GOSUB 1602 : WEND
        560 RETURN
        700 GOSUB 1604 : PRINT FN a$ (26, 74); "BYTE NUMBER";
        702 PRINT FN a$ (27, 76); : INPUT b$ : b = VAL ("&H" + b$) : PRINT
        704 IF b < 0 OR b > 511 THEN GOSUB 730 : GOTO 702
        706 RETURN
        710 PRINT FN a$ (28, 74); "VALUE";
        712 PRINT FN a$ (29, 76); : INPUT q$ : q = VAL (q$) : PRINT
        714 IF q < 0 OR q > 255 THEN GOSUB 730 : GOTO 712
        716 GOSUB 900 : RETURN
        720  FOR  n = 26 TO 29 : PRINT FN a$ (n, 74); SPACE$ (14);  :  NEXT  :
RETURN
        730 GOSUB 720 : PRINT FN a$ (26, 74), "RE-ENTER"; : RETURN
        800  ON ERROR GOTO 3020 : IF t% < 0 OR t% > tm THEN GOTO 3019  :  ELSE
GOSUB 1610
        801 GOSUB 1620 : POKE 51204!, s%, t% : POKE 51234!, s%, t% : CALL k
        802 PRINT FN a$ (3, 76) t% FN a$ (6, 76) s% : GOSUB 2000 : RETURN
        900 hb = INT (b / 16) : lb = b MOD 16 : q$ = r$ + HEX$ (q, 2) + o$
        902 PRINT FN a$ (hb, 6 + lb * 3); q$;
        905 qq = q : IF q < 33 THEN qq = 46
        910  POKE  51712!  + b, q : PRINT FN a$ (hb, 56 + lb);  CHR$  (qq);  :
RETURN
        999 END
        1000 PRINT CHR$ (27) "0" CHR$ (27) "E" CHR$ (27) "H" : hm = HIMEM
        1001 MEMORY 51199! : v = 51456! : u = 51440!
        1005 POKE 52224!, 27, 89, 32, 88 : POKE 52244!, 255
        1010 POKE 52248!, 27, 89, 32, 38 : POKE 52300!, 255
        1011 POKE u, 17, 255, 0, 14, 110, 195, 5, 0
        1012 c$ = CHR$ (27) : r$ = c$ + "p" : o$ = c$ + "q"
        1013 pa$ = r$ + " BUSY " + o$ + " "
        1014 pb$ = r$ + " KEYPRESS " + o$
        1015 pc$ = r$ + " ENTER " + o$ + " "
        1016 DEF FN a$ (r, c) = c$ + "Y" + CHR$ (32 + r) + CHR$ (32 + c)
        1020  POKE  51456!, 221, 33, 4, 204, 17, 26, 204, 126, 198,  32,  221,
119, 254, 18, 126
        1030 POKE 51471!, 183, 23, 23, 23, 38, 101, 111, 41, 19, 19, 6, 16
        1040 POKE 51483!, 126, 254, 33, 48, 2, 62, 46, 221, 119, 0, 221, 35
        1050 POKE 51495!, 126, 31, 31, 31, 31, 230, 15, 205, 128, 201
        1060 POKE 51505!, 126, 230, 15, 205, 128, 201, 62, 32, 18, 19, 35, 16,
221
        1070  POKE 51518!, 17, 0, 204, 14, 9, 205, 5, 0, 17, 24, 204,  14,  9,
195, 5, 0
        1080 POKE 51584!, 198, 48, 254, 58, 56, 2, 198, 7, 18, 19, 201
        1090 RETURN
        1500 FOR n = 0 TO 31 : h$ = HEX$ (n * 16, 3) : PRINT FN a$ (n, 0); h$;
: NEXT
        1501 PRINT FN a$ (0, 0);
        1502 GOSUB 1600
        1504 PRINT FN a$ (2, 74) "TRACK";
        1505 PRINT FN a$ (3, 76) t%
        1506 PRINT FN a$ (5, 74) "SECTOR";
        1507 PRINT FN a$ (6, 76) s%
        1508 PRINT FN a$ (8, 74) "Press:";
        1510 PRINT FN a$ (10, 74) "T Tr/Sec";
        1512 PRINT FN a$ (12, 74) "K Amend by";
        1514 PRINT FN a$ (13, 74) " Keypress";
        1516 PRINT FN a$ (14, 74) "A Amend by";
```

```
        1518 PRINT FN a$ (15, 74) " ASCII";
        1524 PRINT FN a$ (16, 74) " U Update";
        1526 PRINT FN a$ (17, 74) " disc.";
        1528 PRINT FN a$ (18, 74) "H Help";
        1530 PRINT FN a$ (20, 74) "+ Next";
        1532 PRINT FN a$ (21, 74) " Sector";
        1534 PRINT FN a$ (22, 74) ". Previous";
        1536 PRINT FN a$ (23, 74) " Sector";
        1538 PRINT FN a$ (24, 74) "Q Quit";
        1540 RETURN
        1600 PRINT : PRINT FN a$ (0, 74); pa$; : RETURN
        1602 PRINT : PRINT FN a$ (0, 74); pb$; : RETURN
        1604 PRINT : PRINT FN a$ (0, 74); pc$; : RETURN
        1610 PRINT FN a$ (3, 76) SPACE$ (11);
        1611 PRINT FN a$ (6, 76) SPACE$ (11); : RETURN
        1620 PRINT FN a$ (3, 76) t%
        1621 PRINT FN a$ 6, 76) s% : RETURN
        2000 GOSUB 3050 : GOSUB 1600 : POKE u + 1, 255 : CALL u
        2001  FOR a% = 0 TO 31 : CALL v (a%) : NEXT : CALL u : LEB a, 0, 0   '
<---- Lightning BASIC command...
        2002 POKE u + 1, 36 : CALL u : GOSUB 1602 : RETURN
        2500 PRINT c$ "E" c$ "H"
        2502  PRINT "To run the program, select your disc by pressing A or  B,
and choosing a sector and track."
        2504  PRINT  "0  and 0 will do, if you  are  just  experimenting!  The
program produces a full sector listing"
        2506 PRINT "in hex. At the top right of the screen, you will see:"
        2508  PRINT  :  PRINT "BUSY. Don't do  anything  until  this  changes!
Computer at work! Or:"
        2510  PRINT : PRINT "KEYPRESS. One of T K A U H Q + or -  is  awaited.
Or:"
        2512 PRINT : PRINT "ENTER. At the ? prompt, an input is awaited. Enter
and press [RETURN]."
        2514 PRINT : PRINT "T: You have asked to enter a new track and sector.
Enter each and press [RETURN]."
        2516 PRINT "K: You amend your chosen byte by entering the number of it
in HEX, and the pressing the key for the alteration."
        2518 PRINT "A: The same, but you enter the value of the code you  wish
to enter in decimal. If you want to use HEX, you may enter with &H. To  change
X to A, enter 65 or &H41."
        2520 PRINT "U: When you have done amendments, they will be entered  on
screen but NOT on the disc, as yet. Press U to update the DISC."
        2522 PRINT "Q: Quit, that's easy!"
        2524 PRINT "+: Go to the next sector."
        2526 PRINT "-: Go back a sector. (Either of the [+] and [-] keys  will
operate.)"
        2528 PRINT : PRINT "ERROR MESSAGES: There is not room on the screen to
give them conventionally:"
        2530  PRINT "Three beeps and blank screens: Sector or  track  illegal.
Re-enter."
        2532  PRINT "Flash: You have amended without updating. Press U if  you
meant to update. Repeat the keypress if it was intentional not to update."
        2534 RETURN
        3000 z$ = UPPER$ (INPUT$ (1)) : z = ASC (z$) : RETURN
        3010 PRINT "Re-enter track & sector." : RESUME 40
        3019 SCRUNCH  ' <---- Voluntary error (see text)
        3020 FOR n = 1 TO 3 : PRINT CHR$ (7); : OUT 248, 8 : GOSUB 3060
        3021 OUT 248, 7 : GOSUB 3060 : NEXT
        3030 GOSUB 1610 : GOSUB 1604 : RESUME 150
        3040 uf = 0 : FOR n = 1 TO 5 : PRINT CHR$ (7) : NEXT : OUT 247, 128  :
RETURN
```

```
        3050 IF s% > sm OR t% > tm THEN 3019 : ELSE RETURN
        3060 FOR nn = 1 TO 500 : NEXT : RETURN
```

(ROCHE> This gives:

```
$$$$
)
```

The subroutine at 1000 initializes various things, and installs the screen writing code. This is called and, up to about line 100, the coding is taken from an earlier development of the program. The line numbers for the sector on the left, and the menu on the right, are then installed, using the routine at 1500. They remain in place throughout the program. The middle of the screen is then filled from the sector of your initial choice.

The menu operates mainly by keypress, and you will see the various options. These are described in greater detail by the Help page at 2500. If you are typing in this program, it is sensible to ignore these lines. 2500 RETURN is all you need.

The menu is fairly self-explanatory, with the usual INSTR and ON-GOTO lines. You will see that each option is very short. This is because we have made much use of subroutines. There was not room to include the Direct printing option on the menu (use [ALT] and [D]). Direct printing asks for a prompt number for the start. Subsequently, bytes can be typed in as hex numbers. You must use '00' for 0, and '0F' for decimal 15. A [RETURN] signals that enough numbers have been changed. You should then press 'U' for Update if your amendment is satisfactory.

The program has been slightly modified on the disc, so that it will work on the Amstrad PCW9512, as well as the Amstrad PCW8256/8512.

The subroutines from 700-900 are the workhorses for most of the options on the menu. Those at 1600 are used for the minor printing changes in the menu. The routine at 2000 does the screen print out for a sector.

At 3000, we have a keypress routine, followed by various error routines. With less space than usual for the menu, the error routines take on an additional importance. While it would be possible to update the disc after every amendment, it is quicker and less error prone to ask for an update at the user's discretion. As this can be forgotten, a flag (named uf) keeps track of any amendments, and warns if an update is not done before going on to the next sector. The other error is an illegal track or sector. While the program finds this out eventually, it can cause horrible grinding noises. It is better to send it to an error routine while it is prefectly legal. GOTO 3019 and the line 3019 itself ("SCRUNCH") may be a novel way of producing an error! Both errors produce variations on the "son et lumière" theme.

For the less dedicated hackers, there might be a question of what use a disc editor can be. An obvious and simple use is an erased file. Track 1 Sector 0 is the start of the directory. Suppose you have accidentally erased MYPROG.BAS from the A disc. When you edit this sector (or perhaps the next), you will see a line containing MYPROG.BAS. It will start with an E5 code, the erase signal. Replace with 00, by using the ASCII option. Update and quit the program. MYPROG will be back in the directory when you DIR. (ROCHE> Geoffrey Childs does not seem to use "user numbers" because, if you patch '00', the file will appear in the directory of user 0. But you could just as well change this user number to any one legal under CP/M Plus. See the "CP/M Plus User's Guide".)

Chapter 19: Interrupts

If you tell most of your friends that have written an interrupt-driven program, they will probably tell you that their missus is like that, too! However, you may deeply impress the more knowledgeable -- and rightly, too!

Using this technique is not easy. If you try to write such programs, there will be a lot of trial and error, despair and crashes. At least, there will be, unless you are a very much better programmer than I am. The only interrupt program that I have ever written that worked first time was for the sharp MZ-80K. It consisted of two bytes -- 118, 201 (HALT, RET). It is probably the shortest machine code program on record, and just about the longest in execution! The clock was designed, on that machine, so that the only interrupt that would restart after HALT was after the clock reached a time of 11.59.59! I think that it worked O.K -- for obvious reasons, I did not test it from switch on!

The idea of interrupt programming is to create our own diversions, so that the Amstrad PCW does what we want it to do, as well as what it wants to do for itself. When it has done our job and its own interrupts, it carries on with the normal program.

19.1 Excuse me, I'll only be a microsecond!

The Amstrad PCW is interrupted 300 times in every second. An interrupt is a routine that is carried out as a diversion from the program being executed. For example, it might be a keyboard scan. If you press the [STOP] key, the program stops. Your Mallard BASIC program will not contain lines telling the computer to check for [STOP]. An interrupt will have noted the keypress, and a Mallard BASIC routine will check whether this has happened as every new Mallard BASIC command occurs.

In a Mallard BASIC program, a jump to location 56 will cause a further jump to the interrupt routine (at 64929). For certain ideas, it may be possible to relocate the jump at 56 to go via our own destination. This can be effective in simple cases, but there is no guarantee of any regularity, as most of the interrupts do not go through this route. (As we mentioned, the Sharp only used it once in 12 hours, though there will always be many more occasions on the Amstrad PCW.)

There are three main difficulties on the Amstrad PCW:

1. The only reliable way to intercept the interrupts is to use code in block 128, to trap them before they reach the relevant routine in (ROCHE> Banked.) BIOS.

2. An interrupt to an interrupt must be quick. It is all too easy to give it so much to do that, by the time it returns, it is catching up with its own tail -- put it in cruder terms, if you wish!

3. An interrupt to an interrupt must disable other interrupts. This means that it is impossible to use routines, such as many BDOS calls, which will disable interrupts, but then enable them again.

Don't think that I am trying to discourage you from trying! It is just that, when your first attempt goes wrong, it might be due to one of the reasons above!

Locations 65191 and 65192 (65143-65144 on the Amstrad PCW9512) are of great

importance. They contain the address in Block 128 where the interrupt routines start. The contents differ, depending on the version of CP/M Plus used. Block 128 has a gap of 32 bytes at location 64. If you use these bytes to contain your own code, followed by a jump to the original contents of 65191-65192, and in effect POKE 65191,64,0, all interrupts will go by your route.

32 bytes may not seem very much to play with. However, it is not as bad as that! Block 135 (common memory) is still with us when we go into (ROCHE> Banked.) BIOS, and the 32 bytes can contain a call to anywhere in this region. If you include such a call, it is advisable to make it conditional. It is very unlikely that you will need your code carried out 300 times a second!

That sounds tough going -- it is! When you actually see a program listing using this technique, it will LOOK a little easier.

19.2 The routine timer

C19P1

```
10 x9 = PEEK (65191!) : y9 = PEEK (65192!): z9 = 167
15  IF PEEK (64644) <> 3 THEN x9 = PEEK (65143) : y9 = PEEK (65144)  :
z9 = 119
20 h9 = HIMEM : k9 = INT ((h9 + 1) / 256 - 1) : j9 = k9 * 256 : MEMORY
j9 - 1
30 POKE j9, 1, 9, k9, 205, 90, 252, 233, 0, 201
40 POKE j9 + 9, 33, 21, k9, 1, 12, 0, 17, 64, 0, 237, 176, 201
50 POKE j9 + 21, 229, 42, 33, k9, 35, 34, 33, k9, 225, 195, x9, y9
60 POKE j9 + 35, 33, 64, 0, 243, 34, z9, 254, 251, 201
70 CALL j9 : m9 = j9 + 35 : CALL m9 : POKE j9 + 36, x9, y9
80 POKE 64502!, 0, 0, 0 : POKE j9 + 33, 0, 0
100 REM Put the routine here.
110 FOR n = 1 TO 10000! : NEXT
60000 CALL m9 : DEF FN a (x) = 10 * (INT (PEEK (x) / 16)) + (PEEK  (x)
MOD 16)
60010 t = FN a (64502!) * 3600 + FN a (64503!) * 60 + FN a (64504!)
60020 i = PEEK (j9 + 33) + 256 * PEEK (j9 + 34)
60030 IF ABS (t - i / 300) > 1 THEN i = i + 65536! : GOTO 60030
60040 PRINT "Time was" ROUND (i / 300, 2) "seconds." : MEMORY h9
```

(ROCHE> This gives:

```
$$$$
)
```

It is not too difficult to use the internal timer to keep track of time to the nearest second. For greater accuracy than this, we will need to use the interrupt system. This is about as simple a program as one could write as a useful example of interrupts. There are 300 interrupts per second and, as each of them is recorded by the program, we can measure the timing of a Mallard BASIC routine to 1/300th of a second. We actually print just to the nearest hundredth of a second.

Line 10 to 15 finds the address of start of interrupts in block 128. This DOES vary with different versions of CP/M Plus for the Amstrad PCW. The diversion is coded in line 50. This increments a counter at j9+33 and j9+34, and then reverts to the old interrupt address. Lines 30 and 40 install it at address 64 in block 128, using our old favourite SCR_RUN. Line 60 POKEs 65191-65192 (or 65143-65144) with this new address. It is advisable to disable interrupts while this happens, so we use code for it. In line 70, we call both routines and amend the one in line 60, so that we can restore the old interrupt address

at the end. Line 80 zeroes both the normal clock and our new one.

Between 100 and 60000, you can merge a program of your own that you wish to time. The idea of using both clocks is that the normal clock acts as a crude timer which will show if the interrupts have gone 'round the clock', which they will do in about three minutes. If this has happened, t and i/300 will be sufficiently different to be noticed by the test in line 60030.

On this note, we end our tour. We hope that you have enjoyed it. If so, don't forger to tip the driver!

Appendices

A1: Program disc

The disc provided with this book contains all the programs from the text, so that it is possible to test them without the labour of typing in the text. The names of the programs are taken from the text. C18P3, for example, is the third program listed in Chapter 18. With the exception of C9P2 (which needs Lightning BASIC, as explained in the text), all the programs in the first part of the book (Chapters 1-9) only need Mallard BASIC.

The programs in the second part of the book may require a multiple POKE. The easiest way to deal with this is to RUN"DWBAS" immediately after loading Mallard BASIC. An alternative is to use the coding in Appendix 4 to install a multiple POKE, but make no other changes to Mallard BASIC.

The programs in the Appendices A5 to A8 do require DWBAS. They will also work if you have used Lightning BASIC, instead of DWBAS.

It is appropriate to mention the question of copyright. Ideas are not copyright, and the main intention of the book is to give you ideas! It is also true that those who buy this book are entitled to incorporate the programs into those that they will use for themselves. The question comes when a reader has used one of our programs as part of a bigger program that is to be sold. In general, we would encourage this on the assumption that a suitable acknowledgement was made. If in doubt, consult us -- we are friendly people, really! (ROCHE> Too bad: I have been unable to find you!)

We reserve the right to make amendments or enhancements to the program disc, and your disc will probably contain a README.DOC file. Read it! It is possible that a few disc programs may contain slight variations from the text of the book, for this reason.

A2: Turnkey discs

You should always make a copy of a master -- so they say. Let us assume that you have done so, or will do so! Many people prefer to make a turnkey disc, one that will load from switching on the computer. You will find that there is a PROFILE.SUB file on the program disc. If you don't like it, you can easily edit it.

First, use DISCKIT to copy our program disc onto a new disc. Put in the Asmstrad Master CP/M Plus disc, and type:

```
PIP [RETURN]
```

At PIP's '*' prompt, type in succession these three lines, and wait for the next '*' prompt:

```
m:=j*.ems [RETURN]
m:=basic.com [RETURN]
m:=submit.com [RETURN]
```

Now, put into drive A a new formatted disc and, at the '*' prompt, type:

```
a:=m:*.* [RETURN]
```

When the copying is complete, your turnkey disc is ready for use.

OK. You know a better method. Fair enough, but I thought I had better tell you, just in case.

A3: Documentation for DWBAS

There are so many differences between Mallard BASIC Version 1.29 and 1.39 that affect DWBAS that it is better to have a new program, if you have installed 1.39 (the Mallard BASIC for the Amstrad PCW9512). This is called DW139. So, if you have a Amstrad PCW9512, read DW139 for all our references to DWBAS. If you do type RUN"DWBAS" after you have loaded 1.39, it won't matter, as the correct program will be chained after a check. It will just take a little longer!

To load DWBAS, load Mallard BASIC in the normal way from CP/M Plus, and then put in the disc with DWBAS. Type RUN"DWBAS", press [RETURN], and that is all there is to it. If you have Mallard BASIC and DWBAS on the same disc, you can simply type BASIC DWBAS from CP/M plus to get the same result.

DWBAS is not designed to contain any very complex new commands but, generally, to make life simpler for you. There is access to graphics by one command, but you would need a much larger extension for all the bells and whistles that other extended BASICs could contain.

The extra command in DWBAS can all be obtained by entering LDW, followed by a single letter. For example, LDW c will clear the screen. However, as we know that typing THREE whole letters is hard work, all the DWBAS commands can be obtained by # followed by the letter, with no space in between. Thus, #c is all that is actually needed for a screen clear. One or two DWBAS commands need some numbers (parameters) to follow, and these must be separated from the ORIGINAL command and each other by commas (",").

Parameters can either be numbers or variables that evaluate to the numbers that you want. If you wish to print the word HELLO on row 10, column 8, you could use: #a,10,8:?"HELLO".

(It is much simpler than using CHR$ (27) + "Y" + CHR$ (42) + CHR$ (40), isn't it?) Let's look at the commands without any more waffle. If x1,x2,x3... are used, they stand for the parameters that you have to add after the command.

DWBAS commands

#a,x1,x2: Move the cursor so that printing starts at row x1, column x2.
#b: Beep.
#c: Clear the screen. (See note below.)

#d: Disable the cursor.
#e: re-Enable the cursor.
#f,x1: Flash the screen x1 times. (0-255, but 0 flashes 256 times. No comment.)
#g,x1,x2: Print a high-resolution Graphics point. x1 takes value 0-359. Left screen = 0. Right screen = 359. x2 takes values 0-255. Top = 0. Bottom = 255.
#h: Home. The cursor goes to top left. Any windows are also cancelled.
#i: Inverse video. Whole screen.
#j: Save the cursor position as in ?CHR$(27)"J".
#k: Restore the cursor position saved by above, as in ?CHR$(27)"K".
#i: This is not used!
#m: Message line enabled. (Makes screen 31 rows.)
#n: No message. (Makes screen 32 rows.)
#o: Off with reverse video and/or underlining.
#p: reset the Printer.
#q: near letter-Quality print.
#r: subsequent PRINT statements in Reverse video.
#s: Small print from printer (condensed).
#t,x1: Changes TAB to TAB with CHR$ (x1), instead of spaces. Example:

```
#t,46 changes tabbing to full stops,
#t,32 reverts to normal.
```

#u: Underline subsequent PRINT statements.
#v: cancel inverse Video.
#w,x1,x2,x3,x4: Window start at row x1, colum x2, x3 deep, and x4 wide.

Multiple POKEing

An additional facility in DWBAS is to allow a multiple POKE. The meaning of this is that a row of figures can be POKEd in at the same time. POKE 40000,11,12,13 is the same as POKE 40000,11 : POKE 40001,12 : POKE 40002,13. This makes code writing much easier and, incidentally, much quicker in operation than use of DATA statements.

You can also use LIST as a program line, which is useful in some demonstration programs. If you have disabled the cursor, it will be re-enabled when a program ends or is broken. Using #d in direct mode will, therefore, normally only disable the cursor for an instant.

The screen clear is more complex than the usual one, in that it resets roller RAM. This may be a technical point but, very simply, some graphics and screen dumping routines work much quicker if one can rely on a roller reset every time the screen is cleared. The roller will remain reset, unless any scrolling is done.

DW139 is used if, and only if, you are using Mallard BASIC Version 1.39. The operation is virtually the same, except that #p, #q, and #s, are disabled, as they will not be much use with the Amstrad PCW9512 printer.

A4: Multiple POKE

If you don't want to use DWBAS, you can use one of these short programs to install the multiple POKE that we use in the second half of the book. (For instance, you may want to write a program of your own that does not require any of the other facilities of DWBAS.) Having saved the program, it should be RUN immediately after loading Mallard BASIC in the normal way. The second

program should be used instead of the first if you use Mallard BASIC Version 1.39. If you install DWBAS, don't run these programs as well.

A4

```
10 DATA 209, 18, 62, 44, 190, 192, 35, 19, 213, 195, 69, 93
20 DATA 33, 72, 93, 54, 195, 35, 54, 104, 35, 54, 1, 201
30 FOR n = 360 TO 383 : READ m : POKE n, m : NEXT : v = 372 : CALL v
```

A4139

```
10  DATA 18, 62, 44, 190, 192, 35, 19, 195, 236, 93, 33, 233, 93,  17,
129, 1
20 DATA 6, 9, 26, 119, 19, 35, 16, 250, 201, 205, 29, 71, 205, 67, 19,
195, 104, 1
30 FOR n = 360 TO 393 : READ m : POKE n, m : NEXT : v = 370 : CALL v
```

(ROCHE> This gives:

$$$$
)

A5: Disassembler

A5 is a disassembler, written in Mallard BASIC. RUN"A5" from Mallard BASIC. That's really about all you have to know about running it! Enter the decimal (or hex number) at the prompt, and you will see the code. (ROCHE> That means that A5 is not a real disassembler, but just a routine listing the code, as found inside each debugger.) If you enter hex, use the usual conventions. For example, if you want to start at 49152 (decimal) which is C000 hex, either enter 49152 or &HC000, which is what Mallard BASIC accepts. When you have a page of code, you get various self-evident prompts.

BUT, at this point, there are one or two prompts that the progrm does not give. If you like, we can call it a hidden menu. Three of these five facilities are TOGGLES (meaning: if it is on, it goes off, and it it is off, it goes on!). All five are called by the [ALT] key and a letter pressed at the same time. If you press [ALT] and the relevant letter, the computer beeps for the toggles, and you carry on from the original prompt.

[ALT]-V changes from normal video to inverse, or vive versa. [ALT]-Z gives zero-suppress (A series of 0s in the code is not disassembled, and signalled by GAP.) (Toggle.). [ALT]-P echoes the code to the printer (Toggle.).

[ALT]-D allows disassembly of a program placed in memory as if it was somewhere else in memory. This is not a beginner's tool, but an example may suffice to give some idea of why it is there. If you would like to examine SETKEYS.COM (on the Master Disc 2), you could put it in memory at 55000. (ROCHE> So, A5 can only list the mnemonics of a code in memory, like a debugger. It cannot disassemble a file, like a disassembler. QED!) You could then use [ALT]-D, giving 55000 and 256 to the two prompts, and the disassembly would show the program, as if it was placed starting at 256 (&H0100) where it would start, if loaded as SETKEYS.COM from CP/M Plus.

[ALT]-T is used for transfer of memory. Just enter the original location of the start, the new location of the start, and the number of bytes to be moved.

This program is meant for experienced programmers. If you get a Mallard BASIC error message, it is your fault! If you want to study the program, there is nothing to stop you listing it.

6: Menu subroutine

In Chapter 4, we discussed the benefits of having a general-purpose menu subroutine. The one below, starting at 5500, is written in DWBAS, so that the boxing in can be conveniently done. This takes a little time, so may not be desirable in all circumstances. (A purpose-built routine, or the use of Lightning BASIC, would make it much quicker.) The first part of the program is the same as the one in Chapter 3.

```
        A6
        100 DATA Rates and Rent, Food, Clothes, Drink, Car, Fuel, Holidays
        101 Data Miscellaneous, Quit
        110 t$ = " M A I N M E N U " : j = 9
        120 RESTORE 100 : FOR n = 1 TO j : READ a$ (n) : NEXT
        130 GOSUB 5500
        140 LDW c : PRINT : PRINT "You chose "; a$ (z); "." : END
        5500 LDW c : LDW d : PRINT : l = LEN (t$) : PRINT TAB ((90 - l) /  2);
t$
        5510 la = 0 : FOR n = 1 TO j : lb = LEN (a$ (n)) : IF lb > la THEN  la
= lb
        5520 NEXT
        5530  a  = 14 - j : FOR n = 1 TO j : LDW a, a + 2 * n, 38 - la /  2  :
PRINT n ". " a$ (n) : NEXT
        5540 a = a * 8 : b = a + (2 + j) * 16 : c = 32 : d = 328
        5550 FOR n = c TO d : LDW g, n, a : LDW g, n, b : NEXT
        5560 FOR n = a TO b : LDW g, c, n : LDW g, d, n : NEXT
        5570  LDW a, 30, 33 : LDW r : PRINT " Press a key 1 to" j; : LDW  o  :
LDW e
        5580 z$ = INPUT$ (1) : z = ASC (z$) - 48 : IF z < 1 OR z > j THEN 5580
        5590 RETURN
```

(ROCHE> This gives:

```
$$$$
)
```

A7: Multiple input

Program A7 demonstrates the principle of allowing input to be entered in the order that the user, rather than the computer, chooses. It also illustrates the idea of a simple spreadsheet. When sufficient input has been given, the computer takes over and calculates what remains. If the user has made an incorrect entry, use of the cursor key, followed by [<-DEL] will delete an entry. I agree that this is slightly clumsy, but I couldn't see anything better in the context of the intentions of this program.

The formulas that are used are well known to mathematicians and physicists but, for those of you who are 'ignorant artists', for want of a better phrase, they represent the motion of a particle moving under constant acceleration. For example: dropping something off a tower, or stopping a car with a constant braking force. Any three of the five quantities determines the other two.

There are some interesting minor points in the program. Note the double use of the ON-GOTO to cover all 10 possible situations. b$ is used as an 'indicator', to give the current state of what has been input. It may also be instructive, from the angle of error trapping. There was more to do in this way than I had envisaged when I started to write the program. The entry can give an

impossible answer, it can produce division by zero in some cases, and some entries give two possible solutions.

The program needs DWBAS loaded before running.

A7

```
10 LDW d : LDW n
20 DATA Initial velocity, Final velocity, Acceleration, Distance, Time
30  DIM  a$ (5), d (5) : FOR n = 1 TO 5 : READ a$ (n) : NEXT  :  b$  =
SPACE$ (5) : c = 0
40 LDW c : PRINT "Enter in FIGURES only. Use cursor up and down keys."
50  FOR n = 1 TO 5 : m = 4 * n + 5 : LDW a, m, 0 : PRINT a$ (n) :  LDW
a, m, 38 : PRINT ":" : NEXT
60 n = 1 : LDW a, 9, 40 : WHILE c < 3 : z$ = INPUT$ (1) : z = ASC (z$)
70 IF z = 127 AND d (n) <> 0 THEN GOSUB 350
80 IF z = 30 OR z = 13 THEN GOSUB 360 : ELSE IF z = 31 THEN GOSUB 370
90 IF z > 47 AND z < 58 THEN GOSUB 380
100 m = 4 * n + 5 : LDW a, m, 40 : WEND
110 u = d (1) : v = d (2) : f = d (3) : s = d (4) : t = d (5)
120 ON ERROR GOTO 410
130 i = INSTR (b$, " ") : j = INSTR (i + 1, b$, " ") : ON i GOSUB 200,
260, 310, 340
140 IF d1 THEN LDW a, 9, 60 : PRINT "or "; ROUND (d1, 4)
150 IF d2 THEN LDW a, 13, 60 : PRINT "or "; ROUND (d2, 4)
160 IF d5 THEN LDW a, 25, 60 : PRINT "or "; ROUND (d5, 4)
170 LDW a, 5 + i * 4, 50 : PRINT ROUND (d (i), 4) : LDW a, 5 + j *  4,
50 : PRINT ROUND (d (j), 4)
180 ON ERROR GOTO 0 : LDW a, 29, 30 : PRINT "Another one (Y/N)?"
190  z$ = UPPER$ (INPUT$ (1)) : IF z$ = "Y" THEN RUN : ELSE IF  z$  <>
"N" THEN 190 : ELSE END
200 ON j - 1 GOTO 210, 220, 230, 240
210 IF t THEN d (1) = s / t - f * t / 2 : d (2) = s / t + f * t / 2  :
RETURN : ELSE  ' <---- Missing 420 ?
220 IF t THEN d (1) = s / t * 2 - v : d (3) = 2 * v / t - 2 * s / t  /
t : RETURN : ELSE 420
230 d (1) = v - f * t : d (4) = v * t - f * t * t / 2 : RETURN
240 d (1) = SQR (v * v - 2 * f * s) : d1 = -d (1) : d(5) = (v - d (1))
/ f
250 d5 = (v - d1) / f : RETURN
260 ON j - 2 GOTO 270, 280, 290
270 IF t THEN d (2) = 2 * s / t - u : d (3) = 2 * s / t / t - 2 * u  /
t : RETURN : ELSE 420
280 d (2) = u + f * t : d (4) = u * t + f * t * t / 2 : RETURN
290 d (2) = SQR (u * u + 2 * f * s) : d2 = -d (2)
300 d (5) = (d (2) - u) / f : d5 = (d2 - u) / f : RETURN
310 ON j - 3 GOTO 320, 330
320 IF t THEN d (3) = (v - u) / t : d (4) = (v + u) * t / 2 : RETURN :
ELSE 420
330  IF s THEN d (3) = (v * v - u * u) / 2 / s : d (5) = 2 * s / (u  +
v) : RETURN : ELSE 420
340 IF f THEN d (4) = (v * v - u * u) / 2 / f : d (5) = (v - u) / f  :
RETURN : ELSE 420
350  d  (n) = 0 : c = c - 1 : MID$ (b$, n, 1) = " " : LDW a, m,  40  :
PRINT SPACE$ (40) : RETURN
360 n = n + 1 + (n = 5) : RETURN
370 n = n - 1 - (n = 1) : RETURN
380 PRINT z$; : IF MID$ (b$, n, 1) = "!" THEN c = c - 1 : PRINT SPACE$
(40); : LDW a, m, 41
390 INPUT "", x$ : d (n) = VAL (z$ + x$) : MID$ (b$, n, 1) = "!" : c =
c + 1
400 n = (n + 1) + 5 * (n = 5) : RETURN
```

```
        410 RESUME 420
        420 LDW a, 27, 0 : PRINT "This cannot be answered." : GOTO 180
```

(ROCHE> This gives:

```
$$$$
)
```

A8: Bibliography and discotheque

Probably the wrong word, but it will do! There are three areas in which readers may wish to augment the ideas put forward in this book.

A8.1 BASIC

If you found the early chapters of this book difficult, it may well be advisable to go back to some more elementary material on BASIC. If you have not obtained a copy of the manual from Locomotive, you should do so. Reports of this manual have been mixed -- they usually are -- but the manual does what it intends. It gives faithful detail of the commands and functions that are available in Mallard BASIC, although it probably is not ideal from the point of view of the programmer without any previous experience of computers.

I have not read more than reviews on Ian Sinclair's book. He is an author of great experience, who writes for the less experienced users. His book has been well reviewed, and contains a section on GSX. We have studiously avoided GSX in this book, I am afraid -- it frightens me!

There is also a "BASIC Tutorial" disc available from DW Computronics, which interacts with the Amstrad PCW, so that you learn at the machine. It is designed for anyone between complete beginners and moderately competent programmers.

A8.2 BASIC extensions

The first Mallard BASIC extension produced commercially was EXBASIC. This was produced by Nabitchi, but the company no longer trades. I understand that there is an updated version, but I do not know where it is available. EXBASIC was well received, and very good value for money, although slightly ponderous to operate. Lightning BASIC is sold by CP Software.

A8.3 Machine code

"PCW: Machine Code" (Spa Associates) is the only book on the market that is designed for beginners in this subject, and is also Amstrad PCW specific. This would appear to be the obvious way to start to tackle this area of programming. DW Computronics has produced an "Assembler Toolkit" and "Tutorial Disc". You may well find that a combination of the two provides everything you want for a year or two!

For those who wish to delve deeper into the subject, I can recommend "The Amstrad CP/M Plus" (MML Systems). This is not bedtime reading, but it contains so much information that the biggest problem is finding what you want -- but,

99% of the time, it is there! (ROCHE> This 540-pages book was commissioned by Amstrad, to document the version of CP/M Plus for the Amstrad PCW. When Amstrad saw its size, they refused to print it... So, the authors decided to self-publish it, and it was a best-seller!)

Further details can be obtained from: DW Computronics, Freepost, Chathill, Northumberland. CP Software, Stonefield, The Hill, Burford, Oxon. Spa Associates, Spa Croft, Clifford Rd., Boston Spa, W.Yorks. MML Systems, 11 Sun St. London. (ROCHE> MML Systems is the only one still alive, but they no longer answer messages about CP/M Plus.)

A9: Printer fonts

This is a big subject, and did not slot in with the other chapters of the book, but it merits a brief introduction. We shall only consider the draft tables for the Amstrad PCW8256 and PCW8512. (NLQ tables and locoscript tables work on a similar system, but we shall not consider them, here. The Amstrad PCW9512 uses a different system for the daisy-wheel printer.) The hash tables for these fonts can be obtained from block 136. They are copied there when CP/M Plus is loaded, so they can also be obtained from J1?CPM3.EMS. The program below searches the various versions of CP/M Plus programs, and loads the tables at 40000, from where they can be inspected.

```
A9
10 MEMORY 39872!
20 a$ = CHR$ (87) + CHR$ (1) + CHR$ (93) + CHR$ (1)
30 f$ = FIND$ ("j*.ems") : OPEN "r", 1, f$ : FIELD 1, 128 AS b$
40 FOR n = 1 TO 320 : GET 1, n : c$ = b$
50 GET 1, n + 1 : d$ = b$ : c$ = LEFT$ (d$, 3)
60 i = INSTR (c$, a$)
70 IF i THEN 90
80 NEXT : CLOSE : PRINT "Not found" : END
90 b = 40001! - i : FOR m = 1 TO 10 : GET 1, n
100 FOR k = 1 TO 128 : e$ = b$ : a = ASC (MID$ (e$, k))
110 POKE b, a : b = b + 1 : NEXT : n = n + 1 : NEXT
120 CLOSE : END
```

(ROCHE> Indented, this gives:

```
$$$$
)
```

(No coding tricks, so it is a bit slow -- I warned you about ASC(MID$)!)

There are three tables: an address table, a binary column pattern table, and a table of details for each individual character. After the program is run, the first table starts at 40000, the second at 40258, and the third at 40343. The last table finishes just above 41000, and is followed by a series of unused bytes filled with 255s. These bytes could be overwritten if you wished to make amendments to the tables.

Suppose that you wish to investigate the character that you print using CHR$(0) (which would have to be preceded by CHR$(27), in practice). The table at 40000 starts 87, 1, 93, 1. Each two bytes give the relative displacement (from the first table start, 40000) of the start of each character. 40000+87+1*256=40343. We can see that 6 bytes are needed for the character, so we PEEK from 40343 to 40348.

For the sake of argument, let us imagine we get 15, 16, 17, 18, 19, and 20. We

now use the second table, and read off the binary patterns corresponding to these numbers. Add 15 to 40258, and PEEK (40273) shows the binary pattern of the first column of CHR$(0).

All this may seem very complex, when compared with the storage of characters which appear on the screen. The point of it becomes apparent when you find the refinements to the general system, and the space that is saved thereby.

In the first table, the second byte of each displacement is only a small number. Hence, the largest four bits can be used, and are used for other purposes. If bit 7 is set (the number will exceed 128), take 128 away to give the 'correct' number. You have been signalled that the printed character will be a descended one. For instance, the printed g goes below the line. If the number is still greater than 16, divide by 16, and take the remainder. The result of the division signals that the character needs this number of blank columns before it starts.

In the third table, the entries would only reach 85 (unless you add a few more). Setting bit 7 can be used as a signal. It means print a blank column, then deal with the remainder. The numbers 123-127 also have special meanings, and are used to signal repeat printings of the column pattern that follows.

Imagine that you would like to change the @ character to a square root sign. First, design your character. Turn it into binary column patterns, and see if they exist in the list from 40258-40342. If they don't, you will have to add them. Suppose that you need two extra patterns. You will have to move the third table up two bytes, to make room. Then, include the two new patterns at 40343-40344. Every relative address in table 1 will have to be increased by two bytes. You now replace the coding for @ by your new coding. Hopefully, it is not longer -- if so, more problems. If it is shorter, you can pad it with 0s. All that remains is to write a reverse of program A9 to rewrite your CP/M Plus. At this stage, you are probably feeling that this is far too difficult an operation to try -- I don't blame you in the least! All I wanted to do was to show that, like most things, it COULD be done from Mallard BASIC.

10: LNER 'Mallard' listing

Our Mallard locomotive engine picture appearing on the cover of this book was drawn using Locomotive Software's Mallard BASIC with CP Software's 'Lightning BASIC' extension.

A10

```
10 LEB xm : LEB d : LEB gh : LEB c : LEB l, 20, 175, 180, 175
20 LEB l, 180, 175, 200, 170 : LEB l, 200, 170, 670, 170
30 LEB l, 140, 99, 660, 99
40 FOR n = 1 TO 3 : LEB i, 580 + 4 * n, 99 - n, 660, 99 - n : NEXT
50 LEB l, 120, 175, 100, 109 : LEB l, 100, 109, 112, 104 : LEB l, 112,
104, 140, 99 : LEB l, 660, 99, 660, 175
60 LEB f, 146, 170
70  LEB l, 100, 109, 100, 96 : LEB l, 100, 96, 140, 96 : LEB  l,  140,
96, 140, 99
80 FOR n = 98 TO 108 STEP 2 : LEB l, 100, n, 140, n : NEXT
90 LEB gm
100  DATA 40, 180, 10, 80, 180, 10, 140, 165, 25, 192, 165,  25,  244,
165, 25, 300, 180, 10
110 FOR nn = 1 TO 6 : GOSUB 290 : NEXT
120 LEB gh : LEB xl, 580, 99, 580, 160 : LEB xl, 580, 160, 660, 170
130  LEB xl, 580, 160, 440, 145 : LEB xl, 440, 145, 70, 145 : LEB  xl,
70, 145, 40, 160
```

```
        140 LEB a, 17, 73 : LEB r : PRINT "4 4 6 8"
        150  LEB  o : LEB gh : FOR x = 590 TO 620 : LEB xl, x, 109, x,  127  :
NEXT
        160 FOR x = 624 TO 640 : LEB xl, x, 109, x, 127 : NEXT
        170 LEB xl, 135, 110, 205, 110 : LEB xl, 135, 121, 135, 110
        180 LEB xl, 135, 121, 205, 121 : LEB xl, 205, 110, 205, 121
        190 LEB r : LEB a, 14, 18 : PRINT "MALLARD" : LEB o
        200 LEB xl, 120, 105, 580, 105
        210  LEB  l, 15, 175, 20, 175 : LEB l, 15, 170, 15, 177 : LEB  l,  16,
170, 16, 177 : LEB l, 17, 173, 17, 177
        220 FOR m = 175 TO 182 : FOR n = 20 TO 660 STEP 4 : LEB p, n, m : NEXT
: NEXT
        230  FOR  m = 170 TO 175 : FOR n = 180 TO 660 STEP 4 : LEB p, n,  m  :
NEXT : NEXT
        240  FOR  n = 1 TO 1000 : r = ROUND * 900 : s = SQR (r) : r2 =  RND  *
14400!
        245 s2 = SQR (r2) : LEB p, 225 - s2, 64 + s : NEXT
        250 PRINT CHR$ (7)
        260 z$ = "" : WHILE z$ = "" : z$ = INKEY$ : WEND
        270 IF UPPER$ (z$) = "S" THEN LEB sd
        280 END
        290 p = 3.141593 : READ x, y, r
        300  LEB gm : LEB zc, x, y, r : LEB zc, x, y, r + 1 : IF r >  20  THEN
LEB zc, x, y, r - 1
        310 st = p / 16 : IF r < 20 THEN st = p / 8
        320 FOR n = 0 TO 2 * p + 0.001 STEP st
        330 LEB l, x - r * COS (n), y - r * SIN (n) * 0.9, x + r * COS (n),  y
+ r * SIN (n) * 0.9
        340 NEXT : RETURN
```

A11: Notes to the second edition

A second edition of a book should be a good sign. Both the author and the publisher must remain keen about it. *Streamlined* has produced a fair amount of comment, most of it favourable, and the extras we put into the new edition are mainly a result of readers' suggestions. Among many others, I am particularly grateful to George Bridge, who has given me various ideas for improvement of the disc editor, and Chris Shipp, whose reaction to the book has been similar to Oliver Twist's reaction to food.

There are a few minor amendments to the text, and an index -- my thanks are due to readers who provided some excellent ones of their own. The major changes, however, consist of extra programs which are provided on the B side of the disc. All of them need DWBAS (or DW139).

Of all the new programs that I wrote for *Streamlined,* the one I have used most is C18P3, the disc editor. I came to love it on a morning, when disaster struck. I had spent about an hour working on a long program, and dutifully SAVEd the update. The Midlands Electricity Board, with precise timing, chose the instant for an electricity cut when the directory of the old version had been erased and the directory for the new version had not been written. The chances of this must be pretty small, but that's the sort of luck I have! C18P3 passed this test with flying colours, and all was recovered.

The original intention of the program was simply to show that it was possible to write a disc editor in Mallard BASIC, rather than to suggest that it was comparable in quality to other commercial editors. As I used it, I came to realise that, while it did some things well, other aspects were clumsy and more options would be useful. BCDE, on the B side of the disc, is the result.

The additions are described on the extra HELP pages, but readers will see that it is developed from C18P3 which is, of course, fully documented in the text of Chapter 18.

A criticism that was made of the first edition concerned the lack of diagrams and screen dumps in the graphics section. I am dubious whether this was altogether fair, since there were a number of programs in that section. Readers could run these and create their own pictures. However, I would accept that the attitude of the chapter was: "Here is the theory, now go on and develop it yourself."

DRAWDEMO (and DRAWSUB.DOC) are an attempt to answer this. DRAWDEMO can simply be run as a mild form of entertainment, to demonstrate a wide range of graphics effects that can be obtained on the Amstrad PCW. Some of these are very quick, others are more detailed. The more adventurous readers will be able to go further, and extract the routines from this program to write into their own files.

We conclude with three more programs, of a lighter nature. These are simply meant to be fun and, if you don't agree, well, don't agree! They do also contain some programming tricks which may be of interest to some readers.

Chris Shipp showed me a neat technique for creating a sequence of random numbers that does not repeat itself. This is obviously a necessary part of a programmer's armoury. For example, when you write a Quiz program, you don't want a question to repeat itself. The program SNOW was developed from this algorithm. It also illustrates the use of the character set and OUT 246 to produce interesting effects.

Liverpool (who are top of the Footbal League table) would probably beat Milwall (who are bottom) if they played them to-day. But, once in a while, the 'wrong' result happens. To simulate this using Poisson variables -- and, if you don't know and don't care what these are, it doesn't matter in the least -- is the only serious part of the program SOCCER. Maybe you can use the program to keep the children out of trouble on a wet Sunday afternoon. If you use it for pools prediction, don't blame me.

Lastly, CRAZYWP. Appendix 9 discusses altering printer characters on the Amstrad PCW8256/PCW8512. You may have read it, and come to the conclusion that this was far too difficult to achieve in Mallard BASIC (or any other language, for that matter). CRAZYWP demonstrates how this can be done, but I don't think that it will make anybody abandon Locoscript!

The latest version of BCDE is contained in two programs, as the numerous additions made it too long when a B drive was being examined. The second file is called MAP.BAS, which does not run on its own. The best method to operate BCDE is to PIP both files to the M disc, before running Mallard BASIC and DWBAS. Then, RUN"M:BCDE". If the files have not been copied already, BCDE will do this when it is run. Subsequently, changes can be made smoothly between the mapping and the other operations.

Index

(To be done by WS4.)

(Listing of un-protected RPED ?)

EOF