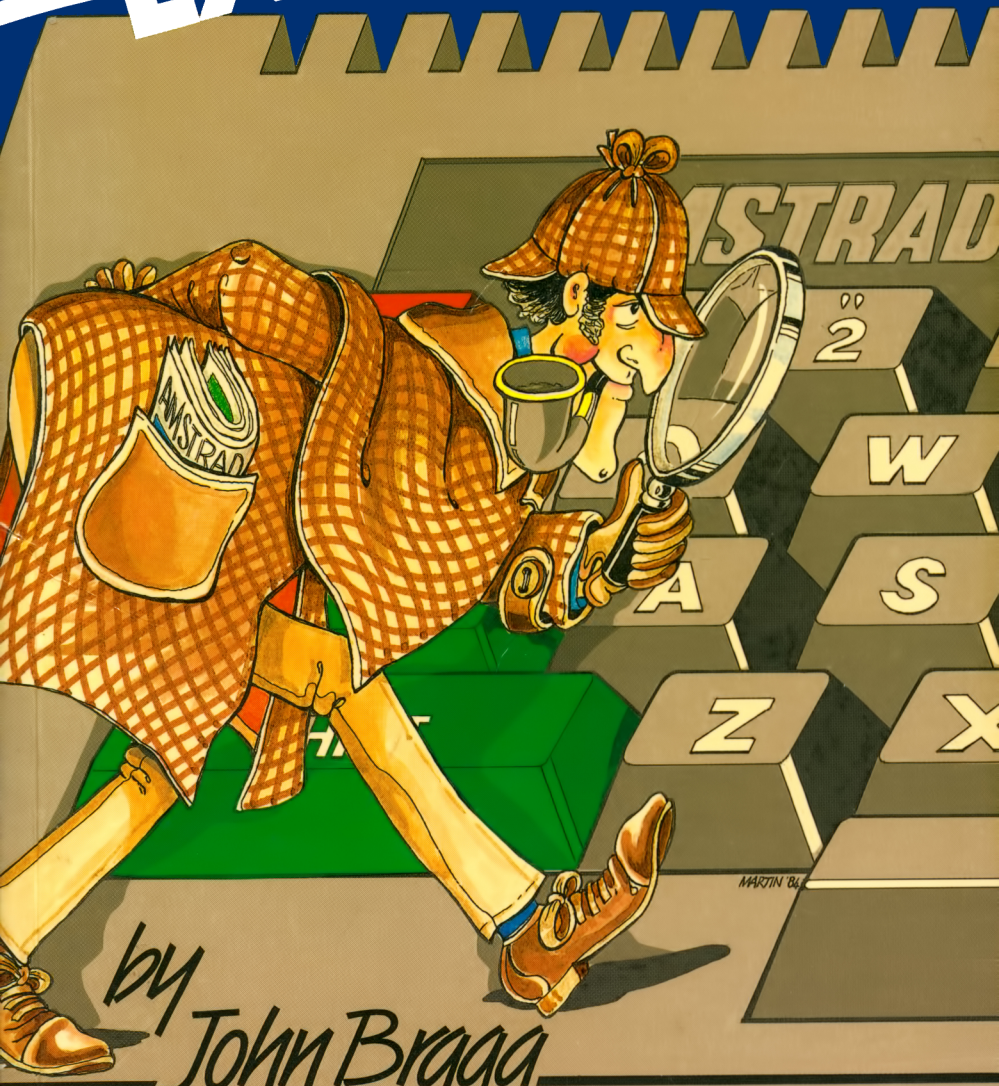
AMSTRAD C4 P6 C4 EXPLORED



by John Braga

AMSTRAD EXPLORED

by John Braga

A Guide
to the Amstrad CPC464

Published by: Kuma Computers Ltd.

ISBN 0-7457-0131-0

Published by:

Kuma Computers Ltd.,
12 Horseshoe Park,
Horseshoe Road,
Pangbourne,
Berkshire RG8 7JW.

Tel: 07357-4335
Telex: 849462

Produced on a Juki 6100 daisywheel printer, connected to a Newbrain microcomputer running under CP/M 2.2 and using Wordstar 3.0.

First Edition September 1984
First Reprint December 1984

Also by John Braga:

The Newbrain Dissected.
The Newbrain Expanded.

PREFACE

The Amstrad CPC464 is one of the most exciting home computers to be launched in 1984. It has already found wide acceptance by those who find excellent value for money coupled with a highly usable language and operating system to their taste. Micro-computers by their nature are complex and very soon enquiring minds start asking questions as to how parts of it really work. Understanding never comes easily and we at Kuma hope that this book will help answer many of the questions first raised and point the way to your own "Explorations".

Tim Moore

September 1984

Kuma Computers Ltd.

CONTENTS

INTRODUCTION

Welcome to the world of the Amstrad CPC464! This interesting and powerful system has attracted very favourable reviews since it first appeared earlier this year. Not surprising, because it offers excellent value for money, combining as it does a great number of facilities in a neat and tidy package.

This book is intended to appeal to all users and potential users of the Amstrad system, of whatever level of skill. It is designed to amplify the information supplied in the system handbook. Rather than merely reproducing, though, the facts given in that handbook, this book concentrates on some of the more unusual aspects of the system, and in particular on two of the major features, the graphics and the sound. Many program examples are given so that the reader can try out the techniques mentioned for himself.

The interest and value of home computing lies in expanding one's knowledge and skill. There is no point in copying large amounts of code without understanding what the writer was aiming at. So this book aims to explain WHY things work, not just WHAT to enter. This should enable the reader to build on the programs here, to write his own games, and to improve on the examples given. The author believes that games should not be sneered at, they can prove very educational providing that the user probes behind the jokey facade.

If your interests lie beyond games, however, Part V which covers the entry of a complete Home Accounting program will be of interest. It shows the capability of the Amstrad to handle serious applications.

2 tapes are available from Kuma Computers to accompany this book. Tape 1 consists of 3 programs, the Sound Harness program from chapter 9, and the 2 games given in chapters 12 and 13. Tape 2 consists of the Home Accounting program. Purchase of these will save you a lot of keying! The tapes are unprotected so that you can modify the source if you wish.

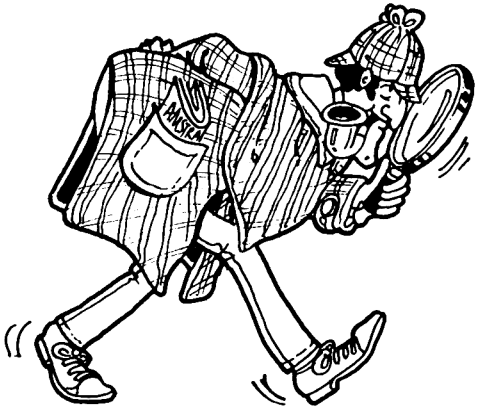
John Braga.
September 1984.

NOTE

Some of the sample programs in this book are printed using an English character-set daisy-wheel so that the ´hash´ sign prints as a £. When copying those programs on the Amstrad keyboard you should use the hash not the £ key.

PART I

GETTING ACQUAINTED



This part contains an overview of the features of the Amstrad, and should be taken as a prerequisite to the other parts.

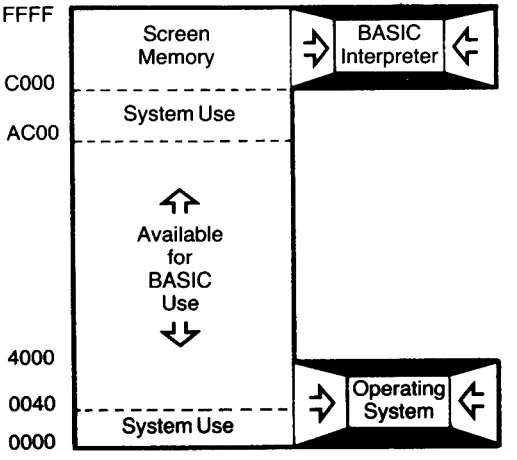
CHAPTER 1
THE AMSTRAD FACILITIES

In this chapter we give an overview of the Amstrad system and introduce the features that will be covered in depth in later chapters.

RAM Galore!

The Amstrad has a generous memory. Gone are the days when home computers had to make do with 1 or 2k of RAM. You now have a full 64k to wallow in. This statement does not perhaps mean much in itself. It is not uncommon for a so-called 64k system to offer in fact very considerably less to the user once the system has grabbed all the working storage it needs! Not so with this one, where the user has some 42k of space under BASIC, quite enough for some man-sized programs as we shall see. Some systems sneakily omit to mention that the use of high-resolution graphics means that large chunks of precious memory are hived off for use by the screen. But the Amstrad uses a constant 16k regardless of the resolution, so there is no memory penalty involved in choosing a high resolution.

In fact, you get more than a full 64k, because there are two 16k ROMS (or one 32k ROM to be quite correct) provided in addition which house the operating system and the BASIC interpreter itself. Since the Z80 can address only 64k, a technique is provided whereby the ROMS can be made to 'appear' or 'disappear' as required. The memory layout is:



The 'paging' of ROMS is normally completely invisible to the BASIC programmer. When you PEEK any address, the ROMS are made invisible so that you see only the RAM.

More than a basic BASIC...

The BASIC is a comprehensive implementation, with many interesting features. As well as the normal facilities you would expect, it is expanded to handle graphics, sound, and even to allow multi-tasking. Unlike some other systems, where programmers have to resort to PEEKs and POKEs every time they want to do anything at all interesting, Amstrad BASIC programs are remarkably readable.

The BASIC has clearly been designed to handle the hardware facilities available, rather than being added on later as an afterthought. As a result you should very rarely find it necessary to enter assembler in order to use system facilities - nearly all are available from the main language. This makes it a very suitable system for the newcomer to computing.

And it is fast. Fast enough for many games, even those involving animation, to be programmed directly in BASIC - though obviously the judicious use of assembler subroutines can always add an extra touch of speed. The Assembler environment is discussed in Part IV, and both of the games in Part III use assembler subroutines for some screen handling.

You've programmed Before?

I said above that the Amstrad is an excellent system for beginners to computing. But there is no reason why the experienced programmer need feel his style cramped. Amstrad seems to be unusual in that it is already making available full details of the system via a Technical Manual. This describes the operating system interface in some depth, and is essential reading for the serious enthusiast. It is in fact remarkably easy to program in machine code on the Amstrad since one can make constant use of the system functions provided. This will also make it easy for software houses to provide additional dedicated programs such as word-processors and spread-sheets.

Flexible Keyboard System.

The keyboard is a standard commercial layout, comfortable to use for long periods and including a full numeric pad. And if you do not like the layout provided you can change it, for every key can be 'soft-programmed' to change its value. You can even decide whether or not you like auto-repeat, and if so how fast. Nor are you restricted to the large character set supplied, for you can design your own characters with ease, ranging from a few monsters for an arcade game to an entire new alphabet if you wish. More of that later.

A single key can also be used to invoke a sequence of several characters, the so-called function key feature.

Cassettes are Fine for Hi-Fi...

Most users of cassette systems start saving up for disks as soon as they can. Cassettes are frankly less than ideal for computer storage, being very slow and unreliable in comparison to floppy disks. But the Amstrad cassette is built in, and I have had no problems with read errors. There is no tone-control to fiddle with, and the volume control acts merely as a monitor so you that you can hear the program being read or written if you wish. The lack of trailing cables and separate units is a real blessing. Most users will have only a single power-cable connected to the colour or black and white monitor. Neat!

Amstrad adds a useful feature in that the user can select to save programs at double speed if he wishes - at 2000 bps instead of 1000. It is true that there is an increased risk of read errors at the higher speed, but the use of good quality tapes (C12 or C30 preferred) should help to avoid problems. I always use the higher speed for normal use, and the lower speed when making backup copies.

How about the Screen?

Windows (on the screen, that is) are very much in vogue at present in the microcomputer world. The Amstrad user can handle up to 8 text windows which can overlap in any fashion. In addition he has at his disposal a separate graphics window which can cover the whole screen area if desired. It is good to note also that there are no restrictions on mixing text and graphics in the one area. In general the graphics are such that sophisticated games are entirely possible, and the intelligent use of colour can make a very attractive user interface for serious applications.

Talking of colours, the Amstrad offers 27, though not all can be seen on the screen at once. There is also a trade-off - colour-range versus degree of resolution. If you choose to have 80 columns on the screen (high resolution) you can have only 2 colours, if on the other hand you reduce the resolution as far as 20 columns you can have 16 colours on the screen.

Most users will find the middle mode a satisfactory compromise - 4 colours and 40 columns.

All 3 modes offer 25 rows.

A Sound Choice!

The Amstrad, perhaps conscious of its company's speciality, has impressive sound capabilities. 3 separate voices plus a noise channel are available and can be manipulated and synchronised to give very pleasing results. Once again all the features are available from BASIC, but the SOUND, ENVELOPE and ENT commands are very complex, and it is not easy to get the full effects without care and study, so Part II concentrates on showing what can be done in this area.

Some other home computers also provide good sound facilities, but let themselves down with tinny internal speakers. The Amstrad has an outlet which can be connected to a stereo hi-fi system, thereby making the sound a good deal more impressive. But you can of course use the internal speaker if you wish.

What Next?

Most users will want to expand their system. A printer is perhaps the first choice for many, and the system supports the attachment of an industry-standard Centronics-compatible unit, so you have a wide choice. Joy-sticks are an attractive acquisition for the games-player. Disk drives are a boon to the serious user, adding enormously to the speed and flexibility of the system. We must wait for these but Amstrad are likely to want to add them soon to take advantage of software available under CP/M.

An alternative to disks is to add dedicated software in ROM. The Amstrad is designed to allow up to 240(!) additional 16k ROMs to be added via ROM expansion boards. Sideways ROMs (so called because they all occupy the same address space, so the system uses only one at a time, the others being made invisible) have proved very popular in the Acorn BBC system, and it is to be hoped that spreadsheets, word-processors, and other languages such as Pascal and Forth, will all be available for the Amstrad before too long. Amstrad have already published guidance for creating ROMS for the CPC464 system, an encouraging sign.

You will see from the memory map earlier in this chapter, that there is a 16k BASIC ROM at the top of memory. It is this ROM which can be 'overlaid' by the sideways ROMS. The lower ROM containing the operating system is never overlaid.

Roll on Pascal!

Summary.

As you can tell, I am impressed by this system which shows considerable care in design. The capabilities are such that it rewards careful study, so here goes...

CHAPTER 2
A TOUR OF THE BASIC

It is not the purpose of this book to teach you BASIC. There are after all many dozens of books on the market which can do that. It is not even its intention to teach you the elements of Amstrad BASIC, for which you should study the handbook supplied. In this chapter we assume you understand at least the fundamentals of BASIC, and therefore concentrate on covering some of the more unusual or complex features.

Graphics and Sound commands are in general covered in later chapters rather than here.

The version of BASIC most commonly available is Microsoft's MBASIC, found on a variety of 8- and 16-bit systems. The Amstrad BASIC, developed by Locomotive Software, is a very comprehensive one, and has many features similar to MBASIC, so you should not encounter many difficulties should you wish to convert programs from one system to another. It also has several extensions to handle the Amstrad colour graphics and sound.

The BASIC interpreter is a large one, occupying the upper 16k ROM. Its facilities are the equal of most commercial systems; there has been no paring of features just because this is primarily a home system. It has a rich range of string functions (including unusual ones like UPPER$ and LOWER$), all the vital numeric functions, the ability to handle integers and reals, multi-dimensional arrays, and so on. It seems equally well adapted to writing serious commercial-type programs, arcade games, graph drawing, or helping with maths homework.

It is also particularly flexible in its use of the keyboard, so we devote the whole of the next chapter to covering the keyboard and the character set.

Structured Programming.

There has been much attention given to this topic in recent years. The Amstrad BASIC offers the usual control structures FOR...NEXT, GOSUB...RETURN, ON X GOSUB and also adds the less usually found (but very useful) WHILE...WEND. It offers user functions - single line user functions only - but no procedures or local variables.

While the omission of these structures is to be regretted, it could be argued that the inclusion of multitasking features more than compensates for their loss since it facilitates many structures that are difficult to program elegantly in less well-endowed BASICs. Multitasking is discussed later in this chapter.

Cassette Files.

One of the important elements any BASIC must handle is Input-Output. It is going to be interesting to see how the Amstrad BASIC is adapted for disk when that unit comes. Of course at present any talk of disk files is mere conjecture, and we must live with cassette. A cassette file is always sequential of course, and creation is simple. The following program writes 10 numbers to a tape file called NUMFILE and reads them back in again (for the hell of it):

```
10 OPENOUT "NUMFILE"
20 FOR J = 1 TO 10
30 WRITE #9, J
40 NEXT J
50 CLOSEOUT
60 PRINT "NOW TO READ BACK THE DATA"
70 OPENIN "NUMFILE"
80 IF EOF THEN 150
90 INPUT #9,J
100 PRINT J
110 GOTO 80
150 CLOSEIN
160 END
```

Simple enough maybe, but there are several interesting points to observe.

- File names can be up to 16 characters in length. (Names longer than this are truncated before being used).

- The usual way of putting data to a file is with a WRITE statement, though a PRINT could also be used. A WRITE is generally preferable since it writes data in such a way that INPUT will handle it in a straightforward fashion on return.

- The ´stream´ for cassette input and output is number 9, hence the ´WRITE #9´. You need not write this as a constant. It may be useful, while debugging a program, to write:

```
20 FILEOUT = 3
30 WINDOW #3,30,40,1,5
40 OPENOUT "NUMFILE"
 ...
100 WRITE #FILEOUT, J, CUSNAME
 ...
150 CLOSEOUT
```

  Here we will see the data written appearing on a small window at the top right of the screen. When all appears to be running satisfactorily line 20 can be changed to FILEOUT = 9, and line 30 deleted.

In the above example lines 40 and 150 are only of use if line 20 references stream 9, but they do no harm. OPENOUT (or OPENIN) does have the effect, however, of causing a 4k buffer to be allocated and this is discussed further below.

It may be reassuring to see messages on the screen while tape blocks are being read or written, but it may at times be inconvenient. Such messages can be suppressed by changing the name used in the OPENxx statement, for example:

```
OPENOUT "!NUMFILE"
```

The exclamation mark is not a part of the filename, but merely a signal to the operating system to avoid messing up your screen with tape messages.

Incidentally you might at first think there is no verify command; some systems offer this so that you can check that what you have written to tape is actually readable while you still have the possibility of writing the data or program again!

It is true that the Amstrad has no explicit verify command, but it does offer the CAT command which reads the entire tape and displays the names of the programs or files it finds there. Not quite the same thing in that it is not comparing the data on the tape with data in memory, but the most important thing is that it verifies that the tape is readable.

Programs on Tape.

Of course, it is not only data files you will be storing on tape, but also programs. A simple:

```
SAVE "PROG1"
```

will save your masterpiece. There are then many ways it can be reloaded the next day. Among others:

- LOAD "PROG1" will discard all other files found, if any, and will load PROG1.

- RUN "PROG1" The program will be loaded, if found, and executed without further intervention.

- MERGE "PROG1" will add PROG1 to the existing program, if any, overwriting lines only when it needs to. Useful for building up a library of subroutines.

- CHAIN "PROG1" The current program is erased and PROG1 loaded and executed, but the variables of the current program are left intact and are therefore available to PROG1.

- CHAIN MERGE "PROG1" works as CHAIN, except that the existing program is retained in whole or in part.

- Do not forget you can always use CTRL and ENTER which is a quick way of typing RUN "".

In summary we have all the facilities to link programs we could wish.

It is not only BASIC programs that may be LOADed and SAVEd. Assembler subroutines can be loaded into an area of memory above BASIC. SAVE and LOAD are used with extra parameters that denote where the routine starts and ends. The same commands can also be used to handle tables such as in-memory indices.

The whole subject of Assembler subroutines is covered further in Part IV.

Memory Layout.

Mention has been made of BASIC requiring 4k for a cassette file buffer, and also of space allocated for subroutines. Both these requirements cause the amount of storage available to BASIC to be reduced, and the memory is taken from the top of the partition, in one case automatically (when OPENIN or OPENOUT is encountered) or in the other case through a user command MEMORY which adjusts the upper boundary.

When first you start the system, space for use by the BASIC interpreter extends from &40 to &AB7F, so HIMEM, which shows the highest byte available to BASIC has the value &AB7F (or 43903, but it is more convenient, when discussing addresses to use hexadecimal notation). If you wish to leave a space free above BASIC for an assembler subroutine or some other purpose you use the MEMORY command to reserve what you need. Assume that you are loading a routine 128 bytes long, which has been created to run at location &AB00.

```
10 MEMORY &AAFF        : REM MOVE BASIC DOWN
                           BELOW ROUTINE.
20 LOAD "!SUB1"        : REM LOAD ROUTINE.
30 SUB1 = &AB00        : REM ADDRESS ROUTINE.
 ...
100 CALL SUB1
```

Notice that the LOAD command does not affect the existing BASIC program since the routine has been saved as a binary file. In fact the saving would have been accomplished with a statement such as:

```
SAVE "SUB1",B,&AB00,&80,&AB00
```

which causes the start, length and entry point to be stored on the tape so that the LOAD can position the file in the correct place.

If you only need the SUB1 routine temporarily you can regain the space later if you wish. If you add a statement such as:

```
5 ORIGMEMORY = HIMEM
```

and

```
200 MEMORY ORIGMEMORY
```

you will restore memory as it was. (Do not try to call SUB1 again though!)

The above showed how the top of BASIC space can be manipulated by the programmer. If the system needs 4k of memory because you open a cassette file it moves HIMEM down automatically as you will discover if you enter:

```
PRINT HIMEM
OPENOUT "TEST"
PRINT HIMEM
```

and gives the area back if it can when it has finished with it:

```
CLOSEOUT
PRINT HIMEM
```

If however, you move HIMEM down after the 4k has been allocated by the system, it cannot move it back again when CLOSEOUT is encountered, so it keeps it!

```
OPENOUT "TEST"
PRINT HIMEM
MEMORY HIMEM-1
CLOSEOUT
PRINT HIMEM
```

The 4k may not be wasted, however, since if you subsequently open a cassette file, BASIC uses the same area rather that steal a new one. Still, you may prefer to allocate your subroutine area before opening any cassette files, and in this way the tape buffer will be released on closing the file.

The MEMORY statement and the cassette buffer are not the only reasons for HIMEM being shifted, as we shall see in the next chapter, which also explains what the 128 bytes between &AB80 and &ABFF are used for.

Multitasking.

We have left the most interesting part of this BASIC until the end of the chapter - multitasking.

Normally a user program executes one routine at a time. The program starts at the first statement when you type RUN and proceeds until eventually it reaches the END statement. Obviously most programs contain loops since computers execute instructions at great speed, otherwise execution times, even of large programs, would be extremely short.

You expect the program to pause on occasion - for example to receive input from the keyboard - but you rely on it stepping in a predictable and serial fashion through the instructions.

Of course the system interrupts your program from time to time. For example we know that interrupts occur every 20 milliseconds to refresh the screen in order to keep the display generated. And there are other system housekeeping chores that have to be done, like updating the TIME counter that acts as a system clock.

The Amstrad BASIC allows the user to request that his main program be interrupted at time-intervals of his choice, for high-priority tasks to be serviced. Each task has a priority level, and a low priority task can be interrupted by a higher priority task. For example a program might be written:

```
10 EVERY 100,0 GOSUB 3000
20 CLS
30 ...
999 END
3000 REM TASK RUNNING AT PRIORITY 0
3010 ...
3999 RETURN
```

The main program runs from 10 to 999. Statement 10 causes a subtask to be created at priority level 0. This task may execute any instructions necessary, but should eventually reach statement 3999.

After statement 10 the main program will probably never refer to the subtask again. It will proceed about its own business, knowing that the operating system has started a count down, and that the subtask will be executed every 2 seconds (the count is in 50/ths of a second and we asked for 100).

When the subtask receives control at statement 3000 it knows it has total control until it relinquishes the reins of power at statement 3999, at which time the main program resumes its interrupted progress. Subtasks have higher priority than the background task.

This is an extremely useful facility in all types of application. In a game we may well use it for moving figures round a screen at regular intervals (see the games in Part III), or for displaying and updating a clock on the screen:

```
2 ON BREAK GOSUB 3000
3 CLS
5 WINDOW #0,1,30,1,25 : WINDOW #1,31,40,25,25
10 INPUT "Enter hours, minutes, seconds ",
          HOURS,MINS,SECS
20 EVERY 50 GOSUB 2000
30 FOR J = 1 TO 500
40 PRINT J
50 NEXT J
60 GOTO 30
2000 REM PRIORITY 0 TASK EVERY SECOND
2005 SECS = SECS + 1
2010 IF SECS = 60 THEN SECS = 0 : MINS = MINS
          + 1
2020  IF MINS = 60 THEN MINS = 0 : HOURS =
          HOURS + 1
2030 IF HOURS = 25 THEN HOURS = 0
2100 PRINT #3 USING "##:##:##";HOURS,MINS,
               SECS;
2999 RETURN
3000 REM BREAK ROUTINE
3010 MODE 1
3020 STOP
```

The above program shows a rather boring main task at statements 5-60, a subtask at 2000-2999 and a routine at 3000 to intercept matters if you decide you have had enough.

Of course you may decide that an interruption every second is an unnecessary overhead. In this case alter line 20 to EVERY 3000 rather than every 50, alter line 10 to input hours and minutes only, and remove seconds from the logic of the subtask. Then your clock will 'tick' once a minute.

There are two important points to note when constructing the above form of subtask:

- Make sure the subtask and the main task do not unintentionally use the same variables. For example if the subtask altered variable J the results in the main task would be unpredictable!

- If you run the above program, then press ESC once only, the program pauses. If you press ESC again, a break occurs and the program ends. But if you press SPACE the program continues. Do that after a 10 second pause, and watch the frantic activity as the clock catches up! This serves to show that the system does not forget interrupts it cannot handle, but stacks them up and executes them later when it can (up to a maximum of 127 or so).

So subtasking can be quite easy to use, but there are a few precautions to take. In the above fragment the subtask printed in a different window from the main task. How about if both are printing to the same window? Consider a main task that has a routine:

```
100 LOCATE X,Y
104 PEN 2
106 PRINT A,B,C
110 FOR J = 1 TO 25
120 ...
```

and a subtask with:

```
4020 LOCATE XPLACE,YPLACE : PEN 3 : PRINT A$
```

If the interrupt to the subtask occurs between statements 106 and 110, or 110 and 120, all may be rosy, but you can see what will happen if the interrupt occurs between 100 and 104 - we have carefully positioned ourselves at X,Y and before we have a chance to print we are snatched away and positioned at XPLACE,YPLACE. When the subtask ends (with a RETURN) the main task changes to PEN 2 and prints - but of course the location is very unlikely to be correct, and the output will therefore look strange.

The problem is even more severe if the interrupt happens to occur after statement 104; subsequent printing will be both in the wrong place and in the wrong colour!

Fortunately the problem is easily avoided once understood. Amstrad BASIC supplies DI and EI instructions (Disable and Enable Interrupts) and we can enclose sensitive pieces of code with these. In the above example,

```
99 DI
108 EI
```

should solve the problem, ensuring we are not interrupted between positioning ourselves and printing. Naturally we should keep the code between DI and EI as short as possible, since we do not want interrupts to be kept waiting long, but we already know that they will be only delayed, not forgotten, so nothing is lost.

Do not confuse 'interrupts' as discussed here with the system interrupts which occur at least every 20 milliseconds. DI has no effect on system interrupts.

The above example showed 1 subtask, but you can have a maximum of 4, which means a total of 5 tasks including the main task. The subtask number, 0 to 3, is indicated by the second parameter of EVERY, ie.

```
10 EVERY 100,1 GOSUB 1500
20 EVERY 80,2 GOSUB 3400
```

kicks off 2 subtasks, numbers 1 and 2. As usual 0 is the default task.

There is a strict priority with 3 having the highest priority and the main task the lowest. So if a priority 1 subtask is operating it may be interrupted by a priority 2 or 3 subtask, but interrupts for priority 0 subtasks will be held pending until the priority 1 subtask terminates, when control will pass to subtask 0 rather than to the main task.

Up to now I have only mentioned the EVERY statement, which causes a regular interruption. There is also the AFTER statement which causes a single interruption only.

A subtask can of course originate another subtask, including itself. So you might have:

```
10 AFTER 1000 GOSUB 3000
....
3000 REM SUBTASK 0
....
3100 AFTER 500 GOSUB 3000
3110 RETURN
```

There is also the REMAIN function which is useful if you decide to put a halt to further interrupts. Invoking REMAIN(subtaskno) causes any pending interrupt for the appropriate subtask to be cancelled. Subtasks are cancelled anyway when an END statement is encountered.

A few words of warning.

You can see that the creation of subtasks is made remarkably easy for you. The few minor pitfalls that there are are quite easily avoided.

Piftall number 1 we have already discussed. Check that you do not unintentionally use a variable in one subtask that is also used in another, or unpredictable results may occur.

Pitfall number 2. Check that you use DI and EI to surround bits of coding that cannot be interrupted.

Pitfall number 3. It is no use saying EVERY 50 GOSUB 3000 if your subtask at 3000 takes more than 1 second to execute! You will spend all your time in the subtask and the main task will never get a look in... If necessary you can check the length of a subroutine using a T=TIME statement at the start, and a PRINT TIME-T at the end. The value printed will be the time in 300/ths second.

To summarise, the Amstrad Interrupt and Multitasking commands are an extremely useful and powerful facility, and provide an easy way to handle many situations that are complex in other BASICs.

Future Expansion.

A hopeful sign for the future is that the BASIC is designed to be expanded, so future extensions should not come as an afterthought (as undoubtedly happens with many other systems!) The mechanism is that a command preceded by a vertical bar '|' will be considered by BASIC as an external command, ie. a command to be handled by another ROM.

Quite how this will be used remains to be seen, but it may be that the Amstrad disk controller, when it comes, will include a ROM containing some new BASIC commands to allow disk directories to be listed, direct-access files to be read and so on. By the use of the 'external command' convention, the new ROM can effectively extend the existing ROM, replacing existing functions or adding new ones.

Mere conjecture at present, but it would be a much cheaper and more painless method than having to prise out existing ROMS to replace them with new ones!

CHAPTER 3
KEYS AND CHARACTERS

The Amstrad keyboard is a very satisfactory design, both in terms of its comfort in use and because of the software behind it. In this chapter we consider the Amstrad character set, what is involved in changing it, and how the keyboard can be tailored to your requirements.

User Defined Characters.

In Chapter 2 we discussed how the boundary at the top of BASIC could shift up and down. There is one other system function that causes HIMEM to be moved down and that is the SYMBOL AFTER command. The Amstrad allows 256 different characters to be displayed. Normally the first 32 are not seen since they are interpreted as control codes, but you can in fact display them by preceding them with CHR$(1).

```
10 REM DISPLAY ALL CHARACTERS, 0 TO 255
20 FOR J = 0 TO 31
30 PRINT CHR$(1);
40 PRINT CHR$(J);
50 NEXT J
60 FOR J = 32 TO 255
70 PRINT CHR$(J);
80 NEXT J
90 END
```

A very useful feature of the Amstrad is that any of these 256 symbols may be replaced by another, so you can define special characters to suit yourself. When you first start the system you already have the ability to redefine up to 16 characters only, numbers 240 to 255, but you can alter matters so that all 256 can be redefined if you wish.

Since the character definitions are normally held in ROM (all but the last 16) and your definitions will obviously need to be held in RAM, the system needs some space to be reserved, so if you enter SYMBOL AFTER, HIMEM is moved down. SYMBOL AFTER 200 means that you wish to have the capability of redefining all characters from 200 onwards. SYMBOL AFTER 0 allows all 256 characters to be redefined.

Having used SYMBOL AFTER, if necessary (remember there is an assumed SYMBOL AFTER 240 when you start up), you can use SYMBOL to set up the new character definitions. Each character is defined as an 8 x 8 matrix of pixels and you denote a pixel as 'ON' by a 1 and 'OFF' by 0, dealing with the top row first. For example the character A - CHR$(65) - would be defined as:

```
SYMBOL 65,&18,&3C,&66,&66,&7E,&66,&66,0
```

```
...**... &18
..****.. &3C
.**..**. &66
.**..**. &66
.******. &7E
.**..**. &66
.**..**. &66
........ &00
```

Do not assume a pixel is the same as a single dot on the screen. In mode 2 (high resolution) this is true, but in mode 1 a pixel is 2 dots, and in mode 0 it is 4.

Note that if you enter a second SYMBOL AFTER command, any existing user-defined characters are wiped out! Also, if you have caused the top of BASIC memory to be adjusted since keying in the last SYMBOL AFTER, the system will refuse the second SYMBOL AFTER command. This explains why you cannot use MEMORY to reserve space for a subroutine, then issue a SYMBOL AFTER. The system has issued its default SYMBOL AFTER 240 command on starting BASIC, then you move the boundary with a MEMORY statement. The remedy is to issue SYMBOL AFTER before issuing any MEMORY statement or using any cassette files.

Incidentally, the system uses the 128 bytes at &AB80 to store the definitions for symbols 240 to 255, 8 bytes for each character.

Character Experiments.

The following program can be used to experiment with your own characters. It uses MODE 0 for clarity. Use the arrow keys to move around the matrix. Pressing D at any location turns the pixel 'ON', pressing U causes the pixel to be turned off (to be 'undotted') and pressing Q causes the program to end and the definition for SYMBOL to be displayed. The symbol is shown in its normal size as it is being created, and the hexadecimal coding for each row of pixels is shown to assist understanding of how the character is declared in a SYMBOL statement.

```
1 REM USER CHARACTER DEFINITIONS
5 ON BREAK GOSUB 6000
10 GOSUB 1000 : REM initialise
20 GOSUB 2000 : REM receive characters
30 END
1000 MODE 0
1010 LOCATE 1,1
1020 FOR row = 1 TO 8
1030  FOR col = 1 TO 8
1040   PRINT ". ";
1050  NEXT col
1060  PRINT " &0"
1080  PRINT
1090 NEXT row
1100 LOCATE 5,22 : PRINT "Enter D (dot),
 U (Undot), Q (Quit) or an arrow key."
1200 WINDOW £1,10,10,20,20
1210 PEN £1,2
1220 SYMBOL AFTER 128
1230 DIM prow(8,8),PXROW(8)
1240 GOSUB 3000 : REM DEFINE CHAR 128
1250 dot$=CHR$(143)
1260 undot$="."
1270 dotcursor$=CHR$(127)
1280 undotcursor$="*"
1290 LOCATE 1,1 : x=1 : y=1 : PRINT undo
tcursor$;
1999 RETURN
2000 REM
2010 z$=INKEY$ : IF z$="" THEN 2010
2015 z$=UPPER$(Z$)
2020 z=INSTR("DUQ"+CHR$(240)+CHR$(241)+C
HR$(242)+CHR$(243),z$)
2030 IF z=0 THEN 2010
```

```
2035 nx=0 : ny=0
2040 ON z GOSUB 2100,2200,2300,2400,2500
,2600,2700
2045 LOCATE X*2-1,Y*2-1
2050 IF prow(x,y)=1 THEN PRINT dot$; ELS
E PRINT undot$;
2060 x=x+nx : y=y+ny : LOCATE X*2-1,Y*2-
1
2070 IF prow(x,y)=1 THEN PRINT dotcursor
$; ELSE PRINT undotcursor$;
2080 LOCATE 19,y*2-1 : PRINT "  " : LOCA
TE 19,Y*2-1 : PRINT HEX$(PXROW(Y))
2090 GOSUB 3000
2095 PRINT £1,CHR$(128);
2099 GOTO 2010
2100 REM D
2110 PROW(X,Y)=1
2120 pxrow(y)=pxrow(y) OR 2^(8-x)
2199 RETURN
2200 REM u
2210 prow(x,y)=0
2220 pxrow(y)=pxrow(y) XOR 2^(8-x)
2299 RETURN
2300 REM q
2310 LOCATE 1,22 : PRINT CHR$(20);:PRINT
 "SYMBOL x,";
2320 FOR row = 1 TO 7
2330 PRINT "&";HEX$(pxrow(row));",";
2340 NEXT row
2350 PRINT "&";HEX$(pxrow(row));
2360 END
2400 REM uparrow
2410 IF y>1 THEN ny=-1
2499 RETURN
2500 REM down arrow
2510 IF y<8 THEN ny=1
2599 RETURN
2600 REM left arrow
2610 IF x>1 THEN nx=-1
2699 RETURN
2700 REM right arrow
2710 IF x<8 THEN nx=1
2799 RETURN
3000 REM define char 128
3010 SYMBOL 128,pxrow(1),pxrow(2),pxrow(
3),pxrow(4),pxrow(5),pxrow(6),pxrow(7),p
xrow(8)
3999 RETURN
6000 REM break
6010 MODE 1
6020 STOP
```

Keying Your Characters.

If you create your own characters using the above method, you may want to be able to display them on the screen using a single key stroke, rather than saying, for example PRINT CHR$(220). This can be achieved easily because the Amstrad allows you to redefine any of the keys to return any value - or more precisely any of 3 values, because the key may be 'normal', or entered at the same time as CTRL or SHIFT.

In addition you can also choose whether you wish the key to auto-repeat, that is to repeat the character if the key is held down. You can even experiment with the repeat rate until you find a speed that suits you personally, using the SPEED KEY command.

To understand all this you should know that every key has a number. You might think that the second key from the left on the third row down is the 'A key', but it only returns A by convention, and can be altered to give any other character you wish. But one thing that never alters is the key number, or rather the key and joystick number, for the optional joysticks are included in the above scheme.

A chart showing the key numbers is shown in the Amstrad handbook, Appendix III page 16. Referring to the key number, we can alter the character returned by any key, when normal, shifted or pressed with the CTRL key. For example:

```
KEY DEF 69,1,&62,&42,2
```

tells the system to alter key 69 (which normally returns a, A, or ctrl-A) so that it returns b, B or ctrl-B. The 1 signifies that we wish it to repeat the character if held down. Try it!

The entire keyboard can be reprogrammed in this way if you wish (except the SHIFT and CTRL keys), so that you could change from a QWERTY layout to a DVORAK layout if you wish (pity about the keytops though).

Changing the A key to produce a B is usually a nonsensical thing to do, but you might well wish to change an infrequently used key to represent a user-defined character instead. Suppose you have used SYMBOL to create 3 special graphics symbols, 220, 221 and 222. You may also decide you can dispense with the square-bracket keys for the time being.

Turning to Appendix III of the Amstrad handbook you will see that the number of the left square bracket key is 17 and that of the right bracket 19. So you could enter:

```
KEY DEF 17,1,220,221,222
```

which will change the left-hand square bracket key so that it will display symbol 220 in 'normal' mode, 221 if shifted and 222 if pressed with CTRL. (The 1 means REPEAT=YES). Key 19 can be saved for other purposes.

An examination of the keyboard will reveal many 'spare' keys that can be used in this way.

Or Key Entire Phrases.

You may well think that the above has demonstrated the full flexibility of the Amstrad keyboard. But there is more! You can redefine any of the keys so that when pressed they generate whole phrases rather than single characters.

This is achieved by reserving certain values 128-159 to be 'expansion tokens'. When any key returns any of these values, the system looks up a 'phrase-table' and extracts the appropriate character or phrase. So token 128 may cause LIST to appear, token 129 RENUM and so on.

The system in fact defaults so that the numeric pad keys return the values 128 to 140, and these have been set up to create the ASCII values '0' through '9' and a decimal point. Also the small ENTER key on the numeric pad has been defined so that if the CTRL key is pressed simultaneously the phrase RUN "+CHR$(13) is generated. This explains why you have to use the small ENTER key, not the large one, in this case.

This use of tokens, and the initial state of the 'phrase-table' is shown in the handbook, Appendix III page 15.

By typing the KEY command (do not confuse this with the KEY DEF command which we have used above) you can set up a library of phrases, each one associated with a separate expansion token. If, for example, you are writing a long program that uses different colours, inks, pens etc., you may well need a phrase such as:

```
CLS:INK 0,1:INK 1,24:PAPER 0:PEN 1:LIST
```

to restore sanity to the situation. But that is a long one to key in if you find yourself typing in bright green on a lime green background!

You cannot unfortunately assign the whole string to 1 key because there is a limit of 32 characters per string. But you can split it between 2.

```
KEY 138,CHR$(13)+"CLS:INK 0,1:INK 1,24"+CHR$(13)
KEY 139,CHR$(13)+"PAPER 0:PEN 1:LIST"+CHR$(13)
```

which will cause the decimal point key and the small ENTER key to generate the required instructions whenever you wish.

There is considerable scope for ingenuity here. For example, if you define a key to produce LIST you can either surround it with CHR$(13) so that a press of the key produces an immediate list, or choose to leave off the final CHR$(13). In the latter case the command LIST will be typed, but the system will then wait for you to complete the instruction. You could then choose to press ENTER (the large key, not the small), or to select the lines to be listed.

Two potential questions may occur to you.

- What happens if you wish, say, the '7' key on the numeric pad to display CHR$(135) when pressed, rather than considering 135 as an expansion token and searching a phrase-table? The answer is simply to insert CHR$(135) in the phrase-table:

```
KEY 135,CHR$(135)
```

- What happens if you are reading the keyboard using INKEY$ and a key is pressed which returns an expansion token? The answer is that the first character of the phrase is returned, and subsequent INKEY$ requests return the other characters.

This feature gives you considerable flexibility, and is one I have found missing on many larger more expensive systems. Properly used it can be a great timesaver. The limitations are that no phrase can be longer than 32 characters, and there is a maximum of 120 characters altogether - so you cannot set up keys for every BASIC keyword for example.

Remember, there is no need to restrict yourself to the numeric pad. Any key on the keyboard can be defined to return a value in the range 128 to 159 by using KEY DEF, and you can then use KEY to assign a phrase or phrases to the newly redefined key.

Remember too that every key can potentially produce 3 characters or expansion tokens, since the key is defined for normal, shifted and CTRL-d operation.

Simple! Two words of warning, though. Remember that the CAPS LOCK key can cause a key to be shifted automatically, and if you have defined its shifted state to be a phrase you may get a surprise... And do not tamper with the ESC key. Redefining it seems to give BASIC nervous flushes, and the system can reset itself unpredictably.

The Keyboard Buffer.

To finish our perusal of the keyboard, we must consider the keyboard buffer. On many machines you can lose characters because the BASIC is too slow to keep up with your typing. This is extremely annoying. But Amstrad BASIC checks every 20 ms to see if a key has been pressed, and if so puts the character in a buffer. When you program using INKEY$, the system does not immediately wait for a key to be pressed, but merely gets the next character from the buffer, if there is one. So you can enter characters independantly of what the program is doing. To prove the point try the following:

```
10  FOR J = 1 TO 10
20  Z$ = INKEY$ : IF Z$ = "" THEN 20
30  FOR J1 = 1 TO 1000 : NEXT J1
40  PRINT Z$;" ";
50  NEXT J
```

and see the system plod manfully on without spilling a drop...

CHAPTER 4
GLORIOUS TECHNICOLOUR

In this chapter we consider the Amstrad screen, how it is handled, and how to manipulate the choice of colour. This lays the basis for more advanced graphics possibilities later.

Any discussion of colour must start with a clear understanding of the basic tools available. (I hope you have a colour monitor!)

Screen Memory.

The Amstrad screen may look peaceful at times, but in fact it is feverishly active. A monitor or TV does not retain an image long, it has to be 'refreshed' at frequent intervals. In the case of the Amstrad the system rewrites the whole screen 50 times a second, ie. every 20 ms. Since this is far too fast for the eye to follow the screen image appears stationary.

The data or images displayed on the screen are held in an area of computer main memory, normally starting at &C000 (but see chapter 5). If we write anything to this area it will be displayed on the screen within 20 ms, ie. the next time the screen is refreshed. So it will appear to be instantaneous.

Unfortunately you will not succeed in writing meaningful data to the screen by POKEing it into memory at address &C000 and above. This is because the mapping of the screen area is complex; a single byte does not match to a character, so POKEing &C000 with 65, for example, will not cause a capital A to appear in the first position of the screen. Tough!

The complexity of the screen memory area does not usually concern us since there are routines available, both in BASIC and Assembler to handle the mapping for us. If you feel you really need to know the mapping of pixels to bytes you need the Amstrad Technical Manual.

Modes of Display.

We have 3 choices concerning the type of display we will use. There are 3 MODES available, and our choice at any time will represent a trade-off between colour and resolution. Each 'pixel' (the smallest addressable point on the screen) has to be individually defined.

In mode 1, which is the default mode into which the system goes on start-up, there are 25 rows of 40, ie. 1000 characters to be defined in memory, and the screen area allocated in the Amstrad is 16k, generally from &C000 to &FFFF. So there are 16 bytes per character, with a few left over.

As you may well remember from chapter 3 (user-defined characters), a character is really a matrix of 64 pixels, therefore each pixel has 16/64 bytes per pixel, ie. 2 bits. With 2 bits you can represent 4 states, 00, 01, 10, or 11, ie. 0, 1, 2 or 3. Hence you can describe any pixel as being one of 4 colours, but no more.

You should now easily be able to work out why mode 0 allows 16 colours (because there are 4 bits per pixel) and mode 2 allows only 2 (a single bit available per pixel!) The area allocated for screen memory remains a constant 16k in all modes.

A Colour Test.

So, back in the default mode, mode 1, we are limited to 4 colours at any time (although, all colours may be set up as flashing, ie. alternating between 2 colours if you really want a headache!) Start by resetting the system. The READY prompt is portrayed in yellow on a blue background - if you have a colour monitor! Clearly only 2 of the 4 colours are in evidence so far.

Enter the following small program:

```
10 CLS
20 GOSUB 1000
30 END
1000 FOR J = 1 TO 5
1010 PRINT J;
1020 NEXT J
1030 PRINT
1040 RETURN
```

Run the program. We often compare writing characters on a screen to using pen and paper. So we may say here that the paper is blue, and that we have picked up a yellow pen.

Now enter the following changes to the above program:

```
30 PEN 2
40 GOSUB 1000
50 PEN 3
60 GOSUB 1000
70 END
```

and you should see 3 sets of figures on the screen, each set in a different colour. Together with the background of blue, that completes our 4 colours. (It is true that we could set the border to a 5th steady colour or to a 5th and 6th flashing colour, but we can do nothing with the border except change its colour and it does not form part of the 16k screen memory area. It will be ignored for the remainder of this discussion).

If we were to change to mode 0 we could display 15 sets of figures, each in a different colour by entering PEN 4, PEN 5, etc. You should visualise PEN as allowing us to select colours from a palette which is limited to a maximum capacity of 16 colours (mode 0), 4 (mode 1), or 2 (mode 2). It is true that we have 27 colours in our paintbox, but we are restricted in the way we use them, since it would require 5 bits per pixel to be able to select 27 colours (with a few spare combinations).

The Artist At Work.

The command INK allows us to change the colours in our palette. Tell the system which of the 4 colours you wish to change (first parameter) and which colour you are going to bring out of the paintbox (second parameter). Whatever colour you were using is, as it were, put back in the box. So although you are limited to a maximum of 4 (in mode 1) you can choose any 4 from the total range of 27.

For example, try:

```
INK 2,4
```

and the second set of figures, in fact anything on the screen written in pen 2, will change from bright cyan to magenta instantaneously.

Any of the 4 can be flashing colours. Try:

```
INK 1,20,8
```

and the first set of figures, which were written using PEN 1, alternate between colours 20 (bright cyan) and 8 (bright magenta).

I must admit that the analogy of the palette is now beginning to wear a bit thin, since merely changing the ink in the palette has caused the picture on the screen to change also, if any of that ink has already been used in the painting. A lightning artist indeed!

A final word of explanation on the INK command is perhaps necessary. In mode 1, as already explained, we are limited to 4 colours. These are denoted in PEN and in the first parameter of INK commands as 0, 1, 2 or 3. When you first start up, the paper you 'write' on is set to ink 0, and the pen to ink 1. What we did in the above program was to write a set of figures in the default ink for pens - ie. 1 - and then to change the pen to pen 2 and finally to pen 3. We have not altered the paper colour at all, it has stayed as ink 0. (Try changing it by all means. Merely enter INK 0,6 for example, and you will see the third set of figures vanish, since you have effectively written them with red pen on red paper).

As well as putting down one pen and taking up another (the PEN command), we can change to a different paper by using the PAPER command. PAPER 2 would cause the background to change to whatever colour INK 2 was set to for all future writing.

It is important that you understand that entering PEN 2 or PAPER 3, for example, does not alter what is already on the screen but only what is written later, whereas entering INK 2,4 or INK 0,6 certainly has an immediate (and frequently dramatic) effect.

Logical vs Physical Colours.

For reference, here is a table of colour numbers. This should also help remind you of the difference between LOGICAL COLOURS (the first parameter of INK) and PHYSICAL COLOURS (the second parameter).

PHYSICAL COLOURS		LOGICAL COLOURS (DEFAULTS)				
Colour Number	Colour Description	Mode 0	Mode 1	Mode 2	Paper	Pen
0	Black	5				
1	Blue	0	0	0	0	
2	Bright Blue	6				
3	Red					
4	Magenta					
5	Mauve					
6	Bright Red	3	3			
7	Purple					
8	Bright Magenta	7				
9	Green					
10	Cyan	8				
11	Sky Blue					
12	Yellow	9				
13	White					
14	Pastel Blue	10				
15	Orange					
16	Pink	11				
17	Pastel Magenta					
18	Bright Green	12				
19	Sea Green					
20	Bright Cyan	2	2			
21	Lime Green					
22	Pastel Green	13				
23	Pastel Cyan					
24	Bright Yellow	1	1	1		1
25	Pastel Yellow					
26	Bright White	4				
	Flashing 1,24	14				
	Flashing 16,11	15				

The above table shows that the default paper is ink 0 and colour 1 (Blue) in all modes. Similarly the default pen is ink 1 and colour 24 in all modes.

Note: Using a logical colour greater than the range allowed (for example PEN 5 while in mode 1) will not cause an error, the value will be taken as MOD 16 (mode 0), MOD 4 (mode 1) or MOD 2 (mode 2).

Special Effects

Do not assume from the above explanation that all colours have always to be different. It may sometimes pay us to make them the same. For example, enter:

```
INK 0,1
PAPER 0
INK 1,24
PEN 1
```

to restore some sanity to your flashing screen, then run the program again, and enter:

```
INK 1,1
```

The first set of figures instantly vanishes! And, if we enter:

```
INK 1,23
```

it reappears just as quickly, but in a different colour. This touches on a very important feature which will be very useful to us when we look at graphics later - the ability to make images appear and disappear again under control. The proper use of INK, and PEN, together with a few control codes will enable us to do such 'tricks' as:

- making objects appear and disappear 'instantaneously',
- giving the illusion of several planes or layers, allowing foreground objects to move in front of midground objects and to obscure background,
- animation.

More of all that later; it is time we moved on from colour to introduce the Amstrad graphics!

CHAPTER 5
GRAPHICS AND TEXT

Following the discussion in the previous chapter on Amstrad colour, we look now at text and graphic commands, and at techniques for handling displays.

Block Graphics.

Everyone understands text; text characters are those in the ASCII character set, characters 32 (space) to 127 (a chequer pattern known as 'course shading'). But graphics perhaps needs some preliminary definition. On the Amstrad there are 2 sorts of graphics, block (or mozaic) and line. Block graphics are single characters and are printed just like text. For example

```
PRINT CHR$(224)
```

prints a single block graphic, a smiling face. Because block graphics behave just like ordinary text characters you can define your own if you wish (see chapter 3) and combine them to form larger characters.

The Amstrad provides a wide range of useful block graphics, including 'sets' of characters that are useful for drawing boxes or diagrams. Study the large definitions given in the handbook, and you will notice various groupings:

characters 128-143: blocks. Each block represents 4 cells, any of which may be 'on' or 'off'.

characters 144-159: lines. Designed to connect the centre of a character to the centre of an edge, in various combinations.

characters 160-191: further text characters. You will notice various foreign characters, including Greek letters, French accents, a Spanish upside-down question mark, etc.

characters 192-255: Miscellaneous graphic symbols.

characters 0-31: Further miscellaneous symbols. Since these are also control codes you have to precede them with CHR$(1) if you wish to see them displayed on the screen, (see chapter 3), but they contain some useful symbols, so should not be forgotten.

Do not forget, when you are using normal line graphics that it may be advantageous to blend them with a judicious mixture of block graphics, for greater effect.

Line Graphics.

Line graphics are, as you might think, formed by lines. Reset the system and try the following simple program to draw a rectangle:

```
5 CLS
10 MOVE 300,200
20 DRAW 340,200
30 DRAW 340,300
40 DRAW 300,300
50 DRAW 300,200
60 END
```

Line graphics are always drawn on a **Graphics Screen**, whereas block graphics and text are drawn on a **Text Screen**. Each type of screen has its own boundaries and its own cursor - but the graphics cursor is always invisible! It is there however, and you can always find its position in XY co-ordinate terms by investigating XPOS and YPOS. After running the above program, XPOS - the horizontal position - will be at 300 and YPOS - the vertical position - will be at 200. Try:

```
PRINT XPOS, YPOS
```

to verify the above statement.

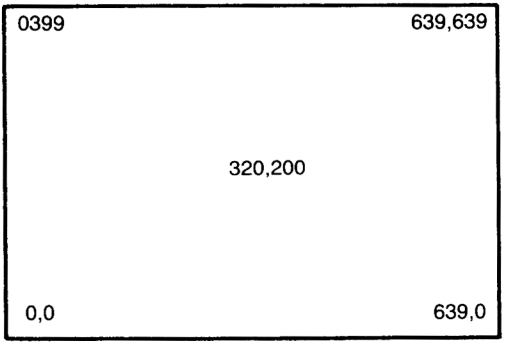
XPOS can range from 0 to 639 and YPOS from 0 to 399 because the graphics screen is 640 x 400 dots. There are actually only 200 dots vertically, but doubling the figure makes the distance between, say, 10 dots approximately the same in either direction, vertical or horizontal, so makes for easy graphics calculations.

The graphics screen is the same size whatever mode you are working in, so graphic programs written for one mode may well work in another, although the results may well look very different. In mode 0, the lowest resolution mode, each pixel will occupy 4 horizontal dots, and if you enter:

```
MODE 0
PLOT 320,200
PLOT 321,200
PLOT 322,200
PLOT 323,200
```

you will see that there is only 1 pixel plotted (or 4 plotted in the same place!)

Positions on the graphics screen are always given with the X coordinate first. Location 0,0 is at the bottom left and 639,399 at the top right.



The Graphics Screen

There is nothing wrong with this arrangement, except that it happens to be the opposite way to that in which the text screens operate. They start numbering at 1 (rather than 0) and run from top left to bottom right, and the X coordinate varies from 20 (mode 0) through 40 (mode 1) to 80 (mode 2)... Oh well, I suppose it keeps you young.

Graphics Commands.

To return to line graphics: The commands are few and very simple. DRAW is obvious; it draws a line in the current 'graphics-pen' colour from whatever the current graphics location is to a new one. The current colour, in graphics terms, is whatever colour you used last time! If this is the first time a graphics command has been used the 'current colour' is ink number 1.

You can select a new graphics colour any time you wish by adding a colour to the DRAW command itself so DRAW 500,200,2 will draw a line in INK 2 colour, whatever that happens to be at the time, and all further graphics commands will continue to use INK 2 until you specifically select another.

MOVE moves to any XY position on the screen but draws nothing, and PLOT draws a single pixel (the size of which will vary according to the current mode) at the specified graphics location. PLOT, like DRAW, can also have a colour stated as the third parameter, and if so that becomes the new graphics colour.

TEST can be used to discover what graphics colour a particular pixel is written in. This might be useful in a games program, for example, where you are manoeuvering about a screen that has been dotted with random objects.

DRAW, MOVE, PLOT and TEST all use absolute XY coordinates, but there are different command forms that use relative addresses - relative to the current location that is. So PLOTR 40,50 will plot a point at XPOS+40,YPOS+50.

In the above small program we used CLS to clear the screen, but strictly speaking that command clears text screens and we should have used CLG which clears the graphics screen. The effect in the above program was the same, assuming you reset the system first, but will not always be so. CLG will normally restore the current PAPER colour, whatever that is, (default 0), but you can clear to a specific colour, ie. CLG 2 will clear to ink number 2 which may or may not be at its default of bright cyan (colour 20), and that will remain the current background graphics colour until you elect to change it.

The only remaining line graphics command to be discussed here is ORIGIN which allows us to move the 0,0 location away from the bottom left-hand corner if we wish.

For example:

```
ORIGIN 320,200
```

moves the 0,0 point to the centre of the screen and it will remain there until we move it elsewhere. While it is there, we must use negative coordinates to move below or to the left of the new 0,0 point. For example, to address the bottom left-hand corner we could say:

```
DRAW -320,-200
```

and

```
DRAW 319,-200
```

to reach the bottom right.

Normally the graphics screen occupies the whole area of the physical screen (excluding the border which is not a part of either the text or the graphics screen). But we can limit graphics to a smaller area if we wish, again using the ORIGIN command.

```
ORIGIN 0,0,160,480,300,100
```

will define a reduced graphics window in the centre of the physical screen. Try entering

```
CLG 2
```

to see the boundaries of the new area. Then try CLS, noting that the clearing of the text screen also clears the graphics screen, since they overlap. CLG will restore it, so there is obviously a simple rule whereby, in the case of screen conflict, the last operation wins!

Drawing Circles.

Apart from TAG and TAGOFF (see below), those are all the graphics commands! No CIRCLE or FILL, which would have been nice. But with those few a lot can be achieved, as we shall see.

Here is a subroutine to plot a hollow circle.

```
1000 REM DRAW CIRCLE OF RADIUS R AT CX,CY
1010 DEG
1020 ORIGIN CX,CY
1030 FOR J = 1 TO 360
1040 PLOT R*COS(J),R*SIN(J)
1050 NEXT J
1060 RETURN
```

Alter line 1040 to:

```
1040 MOVE 0,0 : DRAW R*COS(J),R*SIN(J)
```

if a (reasonably) solid circle is wanted instead.

That subroutine has the feature(?) that the origin is altered to the centre of the circle. If this is not what is wanted, you could write:

```
1000 REM DRAW CIRCLE OF RADIUS R AT CX,CY
1010 DEG
1020 FOR J = 1 TO 360
1030 MOVE CX,CY
1040 PLOTR R*COS(J),R*SIN(J)
1050 NEXT J
1060 RETURN
```

Both subroutines would be invoked by setting CX, CY and R. For example:

```
10 CX= 320 : CY = 200 : R = 150 : GOSUB 1000
```

A Tolerant System.

When experimenting with different circles and other shapes, note that if, by accident or design, you draw lines or plot points off the screen, the system patiently calculates where the line would have gone to, had the screen been large enough, and does the best it can! Try:

```
10 ORIGIN 0,0
20 MOVE 320,200
30 DRAW 0,200
40 DRAW 320,900
50 DRAW 1000,100
60 DRAW 320,200
```

to see the effect. Much better than collapsing with an error message! The only slight disadvantage of this tolerance is that, if you get the coordinate calculations badly wrong you can plot whole figures off the screen which is wasteful of time to say the least.

In general, if the graphics command seems to produce nothing but a deathly silence, check first that you are not plotting in the same ink as the background, then check the current XPOS, YPOS locations to make sure you have not strayed far into the wilderness of outer screen!

Moving the Screen Memory.

I said in chapter 4 that the screen memory normally starts at &C000. That is the default, but you can position it on any 16k boundary if you wish and there is an operating system routine at &BC08 to set the screen memory address to a new position. Theoretically this could be at 0, &4000, or &8000 but in fact there is an excellent reason why only &4000 can be used as an alternative. Placing the screen at &0 or &8000 would overwrite operating system areas in constant use and cause a system crash.

If we do decide to create a new screen area at &4000, we could also create a design at &C000 and switch between the two. This is one way of creating special screen effects such as animation. Run the following program:

```
5 REM wheels program
10 GOSUB 1000
20 scr = scr XOR &80 : GOSUB 2000
30 GOTO 20
1000 REM initialise
1010 MEMORY 16383
1020 FOR j=0 TO 100
1030 READ x
1040 IF x=-1 THEN 1100
1050 POKE &8000+j,x
1060 NEXT j
1070 DATA &3e,&c0,&cd,&8,&bc,&c9
1080 DATA -1
1100 scr = &C0 : GOSUB 2000
1110 start = 1 : GOSUB 1900
1200 scr = &40 : GOSUB 2000
1210 start = 5 : GOSUB 1900
1299 RETURN
1900 CLG : DEG
1902 RESTORE 1995
1905 FOR wheel = 1 TO 2
1907 READ gx,gy
1910 ORIGIN gx,gy : DEG
1920 FOR j = 1 TO 360
1930 PLOT 50*COS(j),50*SIN(j),1
1940 NEXT j
1950 FOR j=start TO start + 360 STEP 8
1960 MOVE 0,0
1970 DRAW 50*COS(j),50*SIN(j)
1980 NEXT j
1985 NEXT wheel
1990 RETURN
1995 DATA 160,100,480,100
2000 REM change screen
2010 POKE &8001,scr
2020 CALL &8000
2999 RETURN
```

The DATA statements make up an assembler subroutine which loads register A with the value &C0, the high-order part of the screen address, and calls the operating system routine at &BC08. The routine at 2000 ensures that the value loaded into A is either &40 or &C0.

There are other ways to achieve the same sort of effect with a single screen area, but the use of two can give an added dimension to complex graphics. Be thankful for the large memory of the Amstrad that allows you to use 32k for screen areas and still have plenty for a large BASIC program!

Text Screens - Windows.

Leaving graphics for the moment, we return to text. We have been talking in this chapter of 'a text screen' and 'a graphics screen'. In fact there are 8 text screens, although by default they are all set to the same area, the physical screen. We have been writing to screen 0, the default text screen, but there are 7 more numbered 1 to 7. There is only 1 graphics screen.

The size and position of a text screen can be defined with the WINDOW command, and that is how we can create several screens to coexist on one physical screen. For example we might want to define a window in the top right of the physical screen. Try resetting the system, then type:

```
WINDOW #1,21,40,1,12
```

which defines screen 1 area as the rightmost 20 columns of our MODE 1 screen and the top 12 rows.

The fact that there are now 2 windows on the screen is not immediately visible, but try entering and running the following program:

```
10 CLS
20 FOR J = 1 TO 20
30 PRINT J;TAB(5);SQR(J)
40 NEXT J
50 END
```

then enter the command

```
LIST #1
```

and the program will be listed in the top right corner without affecting the figures on the left-hand side, which are of course in screen 0.

Note that CLS was taken by the system to mean CLS #0 and that screen 0 is still the full physical screen. Entering:

```
WINDOW #0,1,20,1,25
```

contains screen 0 to the left-hand side of the screen, and

```
WINDOW #2,21,40,13,25
```

stakes a claim to the bottom right.

From now on each window will mind its own business. Each remembers its current location independantly, and printing text in one window does not alter the cursor location (invisible unless the stream is waiting for input) in the other windows. A command such as PAPER #1,3 will change the background ink for one screen only, so a subsequent CLS #3 will cause an instantaneous change of colour. (In some applications that might be a quick way of colouring an entire rectangle at a stroke).

Remember though that windows are selfish and overwrite everything else in sight if they need to - the last man wins!

You already know the CLS command and the PRINT command (use PRINT #3,... to write to screen 3). LOCATE moves the text cursor to the specified XY position, but remember that the text screens work in a different way to the graphics screen, ie. top down and that the first screen position is 1 not 0.

Windows are used extensively in the Home Accounting program in Part V.

Mixing Graphics and Text.

Normally, when you print characters, the system picks the current location as the place to start writing, and updates the current position to point past the end of the characters when finished. But the Amstrad allows you to write text on the graphics screen if you wish. The command TAG #1 will cause subsequent printing to screen 1 to be positioned at the graphics cursor rather than the text cursor position. The graphics position is updated by 8 pixels after each character is written in this way.

The TAG command is very useful for labelling graphs, pie-charts, and other diagrams. To return to normal mode of printing use the TAGOFF command.

There is one other complication when intermingling graphics and text which should be mentioned here. If you have a complicated diagram and are trying to label it using TAG followed by PRINT it will be annoying to you if the text printing obliterates much of the detail. This will happen because when printing a character, the system normally clears the 8 x 8 pixel matrix to the current paper colour before writing the character in the current pen colour. The printing of the character obviously has to be done, but it is that first clearing of the background which we would sometimes like to avoid.

The answer is easy! We will switch to 'Transparent Mode'. All this means is that characters will be printed without the background being cleared first - exactly what we want. Transparency is enabled by writing control code 22 followed by parameter 1 to whichever window we wish - default 0 of course.

```
PRINT CHR$(22)+CHR$(1);
```

will turn transparency ON for screen 0 and

```
PRINT CHR$(22)+CHR$(0);
```

will switch it OFF again.

You will see the effect of this very clearly if you switch transparency on, then type:

```
LOCATE 1,1
LIST
```

assuming, of course, a program exists. The listing will appear jumbled because any characters that were already on the screen are not cleared before the new characters are written.

We return to transparency in chapter 11.

Foreign Character Sets.

As well as being important for the labelling of diagrams, transparency can be very useful if we want to write in French or some other foreign language. It was remarked above that the character set contains foreign accents, and you may have wondered at the time how they could be used without obliterating the character already there. But it should now be clear that:

```
PRINT "e";CHR$(8);CHR$(167)
```

will write e with a grave accent provided that you have switched transparency on first, and

```
PRINT "u";CHR$(8);CHR$(160)
```

will give u with a circumflex accent. CHR$(8) is of course a backspace.

Alternatively you could define your own characters to include foreign accents, perhaps using the USER CHARACTER DEFINITION program from chapter 3, then make foreign accented characters directly usable from the keyboard, as in the following example:

```
5 REM French characters.
10 SYMBOL AFTER 220
20 GOSUB 1000
30 GOSUB 2000
40 FOR j=228 TO 235 : PRINT CHR$(j);: NE
XT
999 END
1000 REM define foreign characters
1010 SYMBOL 228,&18,&18,&3C,&66,&7E,&60,
&3C,0 : REM e grave
1020 SYMBOL 229,&C,&18,&C,&66,&7E,&60,&3
C,0 : REM e acute
1030 SYMBOL 230,&10,&18,&78,&C,&7C,&CC,&
76,0 : REM a grave
1040 SYMBOL 231,&10,&38,&7C,&C,&7C,&CC,&
76,0 : REM a circumflex
1050 SYMBOL 232,&66,&0,&7C,&C,&7C,&CC,&7
6,0 : REM a umlaut
1060 SYMBOL 233,&66,&0,&66,&66,&66,&66,&
3E,0 : REM u umlaut
1070 SYMBOL 234,&18,&3C,&0,&66,&66,&66,&
3E,0 : REM u circumflex
1080 SYMBOL 235,&0,0,&3C,&66,&60,&66,&3C
,&18 : REM c cedilla
1999 RETURN
2000 REM fix keys to print foreign chara
cters
2010 KEY 135,CHR$(228)
2020 KEY 136,CHR$(229)
2030 KEY 137,CHR$(230)
2040 KEY 132,CHR$(231)
2050 KEY 133,CHR$(232)
2060 KEY 134,CHR$(233)
2070 KEY 129,CHR$(234)
2080 KEY 130,CHR$(235)
2999 RETURN
```

Now try using the numeric pad! C'est formidable!

The only snag may come if you wish to print the results, since your printer character set may not be the same as your unique user-defined set... Tant pis!

CHAPTER 6
PROGRAMMABLE HI-FI!

The sound chip supplied with the Amstrad is a sophisticated one allowing:

- 3 channels and a noise channel
- 8 octaves
- control of volume and tone

To use these sophisticated facilities we have to learn some sophisticated commands. There are very few of them, but they are undeniably daunting at first. The commands are:

```
SOUND
ENV
ENT
ON SQ() GOSUB
RELEASE
```

and there is one function:

```
SQ()
```

The SOUND Command.

In this chapter we will look at the SOUND command, leaving a discussion of the others to Part II. With this one command we tell the Amstrad what sound to make (pitch), how long to make it (duration), what stereo channel to put it out on (channel), and how loud to play it (volume).

There are other features of the SOUND command but that will be quite enough to be going on with!

Turn to Appendix VII of the Amstrad handbook, which lists sound frequencies together with their corresponding period number, and try the following to get the feeling of the command:

```
SOUND 1,284
```

This sounds the note known as A, a pitch of 440 cps, or 'International A'. We do not specify the pitch itself, but the period, 284, which is derived from it by the formula shown in Appendix VII. For now it is satisfactory to pick out our notes from the table provided.

We chose to put this sound out on Channel 1, or A. We did not specify a duration or a volume, so defaults were assumed - 1/5th of a second and volume 12 respectively. Repeat it with a longer duration:

```
SOUND 1,284,100
```

The duration is given in 1/100ths of a second, so the sound should have lasted 1 second this time.

Let us experiment with the volume setting. This has a range from 0 (off) to 7 (full on) if a volume envelope is not being used, and 0-15 if you are using an ENV command. Try:

```
10 FOR VOL = 0 TO 7
20 SOUND 1,284,100,VOL
30 NEXT VOL
40 END
```

We can also adjust the sound by means of the wheel on the right side of the Amstrad case.

You may well wonder how the above BASIC program finishes well before the sound ceases. That will be explained in the next chapter.

In case you are tiring of International A by now, let us try a few more notes:

```
10 FOR PERIOD = 3822 TO 16 STEP -30
20 SOUND 1,PERIOD,20,4
30 NEXT PERIOD
40 END
```

Awful! A scale, though a very unsuccessful one! It seemed to spend ages in the depths before passing far too quickly through the heights. We clearly have to find a better way of moving up and down than by adding a fixed increment to a period. We could create a period-table, but let us instead use the a similar formula to that given in the handbook:

```
10 FOR OCTAVE = -3 TO 4
20  FOR NOTE = 0 TO 12
30   GOSUB 1000
40   SOUND 1,PERIOD,50,4
50  NEXT NOTE
60  SOUND 1,0,100,0
70 NEXT OCTAVE
80 END
```

```
1000 REM  CALCULATE PERIOD GIVEN OCTAVE  AND
          NOTE
1010 FREQ = 261.626*(2^(OCTAVE+NOTE/12))
1020 PERIOD = ROUND(125000/FREQ)
1030 RETURN
```

Much better! The thing sounds almost musical. We now have a formula that will return the period given the octave and the note. Far less trouble than looking it up in a period table. Progress!

Our First Tune.

Now that we have the means to calculate any note of the total range, how about a tune? We need to be able to feed the program details of the notes and durations and let it produce the sound. Of course we could look up the notes in Appendix VII, convert them to periods and enter them in data statements, but having worked out a formula, it would seem preferable to enter the notes as near to the original musical notation as possible and let the computer calculate the period for us.

For those readers who have avoided a musical upbringing, the basics are explained in Part II, so they should perhaps continue with this chapter, enter the tune as given, and return to study it later having absorbed the elements of musical notation given later.

The following program plays the tune ´God Save the Queen´. Data statements representing the notes and the duration of each are stored in statement 100-199. The routine at 2000 converts each note from the semi-musical notation in the data statement to an octave number and and note number within the octave.

The tempo is given as a single figure which is used in the calculation of the duration of each note. This makes it easy to adjust the pace of the tune by changing only one statement.

```
1 REM GOD SAVE THE QUEEN
5 TEMPO = 20
7 ENT -1,1,1,3,2,-1,3,1,1,3
10 READ note$,dur$
15 IF note$="" THEN END
20 GOSUB 2000
25 OCTAVE = OCTAVE+2
30 GOSUB 1000
40 SOUND 1,period,dur,4,0,1
42 SOUND 1,PERIOD,1,0
45 GOTO 10
100 DATA C5,C,C5,C,D5,C,B4,DC,C5,Q,D5,C
105 DATA E5,C,E5,C,F5,C,E5,DC,D5,Q,C5,C
110 DATA D5,C,C5,C,B4,C,C5,C
115 DATA C5,Q,D5,Q,E5,Q,F5,Q,G5,C,G5,C,G
5,C,G5,DC,F5,Q,E5,C
120 DATA F5,C,F5,C,F5,C,F5,DC,E5,Q,D5,C,
E5,C,F5,Q,E5,Q,D5,Q,C5,Q
125 DATA E5,DC,F5,Q,G5,C,A5,Q,F5,Q,E5,C,
D5,C,C5,DC
199 DATA "",""
1000 REM calculate period from octave an
d note
1010 freq = 261.626*(2^(octave+note/12))
1020 period = ROUND(125000/freq)
1099 RETURN
2000 REM calculate note, octave and dura
tion from note$ and dur$
2010 Z$=LEFT$(NOTE$,1)
2020 note=INSTR("CCDDEFFGGAAB",Z$)-1
2030 IF NOTE < 0 THEN OCTAVE = 5 : DUR =
 1 : GOTO 2999
2040 Z$=RIGHT$(NOTE$,1)
2050 IF Z$="£" THEN NOTE = NOTE + 1
2060 IF Z$="b" THEN NOTE = NOTE - 1
2080 OCTAVE = VAL(MID$(NOTE$,2,1))-4 : I
F OCTAVE < 0 THEN OCTAVE = 1
2090 IF NOTE < 0 THEN NOTE = 0 : OCTAVE
= OCTAVE - 1
2100 DUR=2^INSTR("HDSQCM",RIGHT$(DUR$,1)
)/16
2120 IF LEN(DUR$)=1 THEN 2200
2130 Z=INSTR("DT",LEFT$(DUR$,1))
2140 IF Z=0 THEN 2200
2150 IF Z=1 THEN DUR=1.5*DUR
2160 IF Z=2 THEN DUR=DUR/3
2200 DUR=DUR*TEMPO
2999 RETURN
```

If you feel the results resemble a violinist with the shakes, please bear with us. Improvements come later!

This program does not use the most efficient means of passing notes via the SOUND command, but it works, and serves to demonstrate the principle involved in playing a tune. More refinement will be added in Part II.

Subroutine 1000 converts the octave and note into a period. The tempo can be adjusted by experimenting with statement 5.

Different tunes could be played by changing lines 100 to 125. The notation used for notes is "XYZ" where X is the note C to B, Y is the octave number 1 to 8 and Z is the optional indicator which shows whether the note must be sharpened or flattened.

Altering the Volume.

Our tune is certainly recognisable, but it is undeniably mechanical. But so is a fairground organ, or a Victorian musical box. The real challenge is to make computer sound as musical as its inevitable limitations will allow. Do not be discouraged, we have barely started!

God Save the Queen was played at fixed volume. You have seen that we can vary the volume in the SOUND command. However we cannot vary it within one command, only between one command and the next. The ENV command allows us to alter the volume as the sound is being played and this can make a big difference to the effectiveness of the music.

The ENV command needs a full discussion to itself, and is covered later, as is the ENT command which allows us to vary the pitch (ie. the period number) as the sound is being heard. An example was given in the tune above.

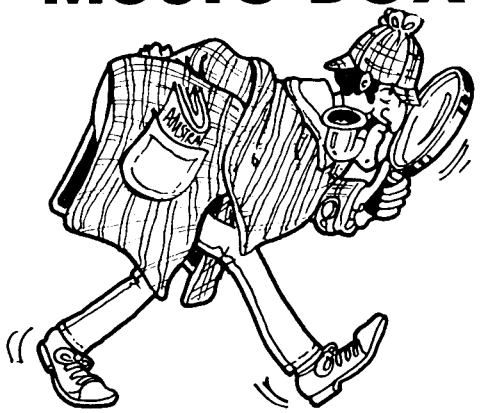
Sound Summary.

So far we have only dipped into the possibilities of the sound facilities, but I hope you can already appreciate the scope. In part II we extend the discussion to cover volume changes, pitch changes, sound queues, tunes for 2 and 3 separate voices, synchronisation and special effects.

So, 'Onwards, relentlessing pressing onwards' (with apologies to Terry Wogan)...

PART II

A VERSATILE MUSIC BOX



This part delves further into the mysteries of programming the Amstrad's Sound capabilities

CHAPTER 7
MUSICAL NOTATION

In the last chapter we introduced the basics of making music on the Amstrad and showed a program to play a simple tune. Since there are 3 music channels we can add 2 more voices to create a far more satisfying sound.

Before we do that we need to explain something of what is involved in capturing sounds on paper in a form that allows other people who have never heard the music to reproduce the composer's intentions.

The Elements of Musical Notation.

If you are worried about not knowing musical notation I suggest you need not be. If you have mastered any of BASIC, hexadecimal or Z80 assembler you can be assured that learning musical notation is trivial by comparison!

The musical scale is conventionally divided into octaves, where the frequency of the top note is exactly twice the frequency of the bottom note. The octave is divided into 12 equal parts called semitones.

In normal Western music, all the notes are selected from these 12ths of the octave, and a scale consists in moving up and down in various combinations of semitones and whole tones. A tone consists of two semitones.

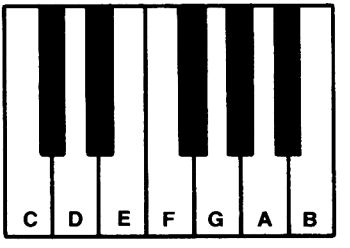
Of course there is no divine law saying that an octave has to be divided into 12 parts, it is just our convention, and music in other parts of the world uses other intervals, such as quarter-tones - which we could reproduce on the Amstrad if we wished.

A scale consisting of all semitones is called a chromatic scale:

```
1 REM CHROMATIC SCALE
10 READ OCTAVE,NOTE
20 IF OCTAVE = 99 THEN END
30 GOSUB 1000
40 SOUND 1,PERIOD,20,4
50 GOTO 10
60 DATA 1,0,1,1,1,2,1,3,1,4,1,5,1,6,1,7,1,8,
     1,9,1,10,1,11,1,12,2,0
70 DATA 99,0
1000 REM CALCULATE PERIOD FROM OCTAVE AND
     NOTE
1010 FREQ = 261.626*(2^(OCTAVE+NOTE/12))
1020 PERIOD = ROUND(125000/FREQ)
1099 RETURN
```

and if you have a piano you should see that a chromatic scale is played by starting on one note and playing all the notes, white and black, until you reach one octave above the first note.

Each note of the octave is given a name. The first 7 letters of the alphabet are used to name the 7 white keys in the octave.



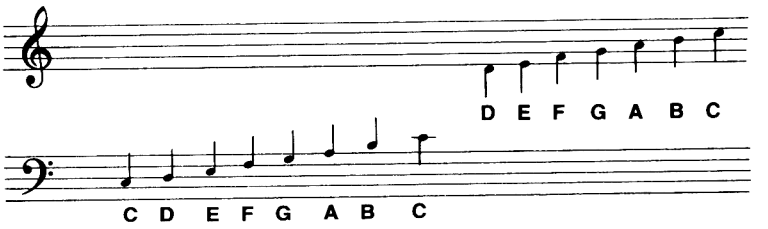
You will notice that most white keys are a whole tone apart, that is there is a black key in between which sounds a semi-tone above one white key and obviously a semi-tone below the white key above. But there is only a semi-tone between the keys E and F, and between B and C, hence no black key is required.

Our chromatic scale happened to start on the note C because that is we used the frequency of C in our formula. That makes C note 0. To play the scale of C major adjust the data statement to read:

```
60 DATA 1,0,1,2,1,4,1,5,1,7,1,9,1,11,1,12,2,0
```

That scale corresponds to the white notes on the piano keyboard, and the octave played started on middle C, which is so called because it is located roughly in the centre of a piano keyboard. The black notes play intervening semitones and the notation ♯ (sharp) and ♭ (flat) is used after the note. ♯ raises the note by a semitone and ♭ flattens it by a semitone. On a piano keyboard there is no distinction between, say G♯ and A♭.

To write our music we use paper lined with 5 lines close to each other, called music staves. Then we put a sign at the beginning of each stave to show which lines represent which note. There are two such signs in common use for keyboard music, though more for some other instruments such as violas. The G clef shows that the second line up on the stave is a G, the G above middle C. So middle C lies just below the stave, and if we need to show middle C we place the note on an extra line of its own, a so-called ledger line.



The F clef shows that the second line down from the top of the stave is F below middle C, so this stave is best at showing lower notes than the G clef.

If you look at piano music you will see that the right part mostly uses the G Clef which is most convenient for music above middle C, while the left hand plays deeper notes and uses the F clef. This is not an inviolate rule!

Now we have the basis to represent the pitch of the notes, but they also have to be assigned a duration. If we take a demi-semi-quaver as 1, a semi-quaver is 2 and so on up to 32 (semi-breve). Duration here is a relative not an absolute term, with the amount of time actually allocated to a note varying widely according to the tempo of the piece, the player's interpretation, etc.

demi-semi-quaver 1

semi-quaver 2

quaver 4

crotchet 8

minim 16

semi-breve 32

The duration of a note can be increased by half by having a dot placed after it. So a dotted minim is the same time as a minim and a crotchet and counts as 24.

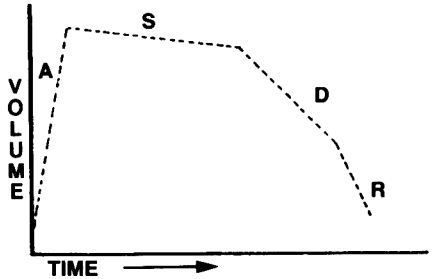
Summary.

Now we know how to represent pitch and duration values we can return to considering how to write tunes on the Amstrad. There is of course a considerable amount more that could be said about musical notation; we have not covered rhythm, tempo, phrasing, and so on, but any reader who has not previously encountered music should now be able to follow the discussion that follows and will hopefully be encouraged to delve deeper into what can be a a most enjoyable lifetime study.

CHAPTER 8
VOLUME AND PITCH

The ENV Command.

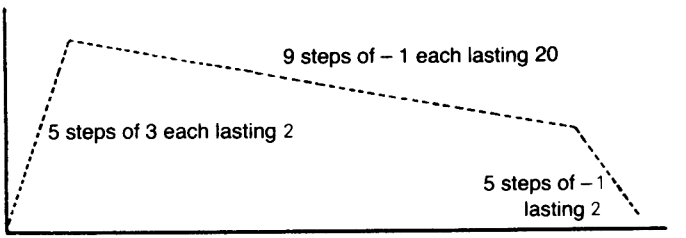
If you play a note on the piano holding the key down for a few seconds, the note starts with a quick rise in volume, followed by a period during which the volume falls only slightly. Finally, the note decays and dies away very quickly when the key is released. It may be pictured:



and is frequently described in terms of **Attack, Sustain, Decay,** and **Release,** or ASDR for short.

If we wish to produce a note electronically we can give precise values to ASDR in terms of the duration of each phase and the amount of volume 'gained' or 'lost', in other words, the length and angle of the lines in the above graph.

The ENV command allows us to simulate this action fairly closely, since we can define up to 5 sections. To simulate a piano note 3 sections might be used, and a suitable envelope for a note lasting 2 seconds might be:



and this could be represented by:

```
ENV 1,5,3,2,9,-1,20,5,-1,2
```

where the first parameter is the envelope number, 1 to 15, the first section 5,3,2 means 5 steps of 3 each 2 units (1/100ths second) in length, followed by the second section 9,-1,20 - 9 steps each of -1 lasting 20 units, and 5 steps of -1 lasting 2 units.

Of course the steps are really a set of small jagged jumps rather than a smooth line, but the effect is approximately the same in most cases.

To use the ENV command we add a fifth parameter to the SOUND command, so a command corresponding to the ENV command above might be:

```
SOUND 1,284,200,0,1
```

which selects 1 as the volume envelope number and 0 as the starting volume. Although you can start the volume at any level you like, it often makes sense to start at 0 if you are using a ENV command, and therefore put all the volume manipulation in one place.

For serious work it may well be worth constructing the following table to check your calculations:

	No. of Steps	Step size	Step Lngth	Total Lngth	End Volume
Starting Volume (from sound command)					0
Section 1:	5	3	2	10	15
Section 2:	9	-1	20	180	6
Section 3:	5	-1	2	10	1

Although the starting volume is given in the table as 0, you should know that the first step in the ENV command is added to the volume before the sound is played, so the first sound actually heard will be at volume 3 in the above example.

You can see why you need pencil and paper to set out volume envelopes with any accuracy. You need to check that the duration is correct and that the volume does not go over 15 or below 0. (If you do exceed the limits, no error will be generated, the volume merely wraps round to give a number between 0 and 15, but the result may not be what you were expecting).

Incidentally, the duration in the above note matches the 200 in the SOUND command, but what happens if there is a discrepancy between the duration in the SOUND command and that specified in the ENV command? The answer is that the duration in the SOUND command predominates, but there are one or two special cases. Consider the following table.

----Duration in----		
SOUND	ENV	
100	100	Duration is 100, ie. 1 second.
90	100	Duration is 90, ENV command will not be fully used.
100	90	Duration is 100. The final 10 100ths/second will sound at constant volume.
0	100	Duration is 100.
-5	10	The ENV command is used 5 times so the duration is 50.

When heard on its own, 'clicks' as the volume changes rather mar its effectiveness, but used together with tone changes and with several voices it can infuse life into the music.

Varying the Pitch - the ENT Command.

Just as the ENV command is used to vary the volume, so the ENT command can be used to change the pitch while the note is playing. It has a similar syntax to the ENV command and can also contain up to 5 sections. Each section is again described in terms of:

- number of steps,
- size of step,
- length of step.

A rapid change of pitch above and below the note is called vibrato. It is a technique used widely by instrumentalists and singers. A pianist, having a percussive instrument, cannot excercise control over vibrato, whereas a string, brass or woodwind player can.

So can the Amstrad, though it must be admitted that the results are somewhat mechanical.

To experiment, reload God Save the Queen and make the following changes:

```
7 ENT -1,1,1,8,1,-1,8
40 SOUND 1,PERIOD,DUR,4,0,1
```

which instruct the system to use tone envelope number 1 (the sixth parameter of SOUND). The first parameter of ENT gives the tone envelope number as 1, but makes it negative to indicate that this envelope is to be repeated as long as the note lasts. Then there are 2 sections, the first containing 1 step of 1 period, lasting 8/100ths second, the second again containing 1 step of -1, lasting the same time.

The above, to my ear, sounds a pleasing and reasonably natural vibrato. You may disagree. In any case you can experiment. Try varying the pitch by more:

```
7 ENT -1,1,2,8,1,-2,8
```

This seems to me to be too extreme on long notes, though quite acceptable on short notes. (The degree of vibrato that is acceptable is a personal judgement!)

Note that both these ENT commands are actually varying the pitch upwards only, and not below the note. To make a true vibrato, try:

```
7 ENT -1,1,1,5,2,-1,5,1,1,5
```

which brings the pitch below the note too. This is a restrained vibrato which should please most ears.

Altering the step time parameter outwards will make the vibrato slow and lazy, while making it shorter will give a 'tighter' effect until you will probably decide that the sound is taking on a neurotic tinge!

It's For You!

The ENT command is not only useful in music. Try:

```
10 ENT -1,1,1,2,1,-1,2
20 GOSUB 1000
30 GOSUB 1000
40 SOUND 1,0,200,0
50 GOTO 20
1000 REM
1010 SOUND 49,70,50,15,0,1
1020 SOUND 28,35,50,5,0,1
1030 SOUND 42,17,50,5,0,1
1040 SOUND 1,0,15,0
1999 RETURN
```

The acute observer may notice that the above sound is 'out of tune', as the period in line 1030 should ideally be 17.5. But so are most phones, so I am unrepentant!

Ample opportunity to enhance your programs with some interesting embellishments!

CHAPTER 9
2 AND 3 PART HARMONY

When we explained in chapter 6 how to program a tune, it was stated that the method shown was not the most efficient. In this chapter we explain an interesting facet of Amstrad BASIC which makes the programming of music much easier.

Sound Queues.

When experimenting with the Sound command you will have noticed that a BASIC program which uses sound frequently finishes before the tune - it appears to be able to 'run ahead' a certain way.

This is exactly what it does do. The sound command actually places instructions in a sound queue, and the operating system takes the sounds off the top of the queue and feeds them to the sound chip when it can.

The sound queue seems to hold 4 sounds so the BASIC program will very quickly fill up the queue and then have to wait until the first sound has finished playing before the queue is 'shuffled up' by the operating system as the 2nd note is played. BASIC can then place the 5th note in the queue, and hangs impatiently around waiting to be able to put the 6th one on.

This seems a wasteful way to handle things, so Amstrad BASIC has thoughtfully arranged for us not to have to worry about waiting. As long as we let the system know we wish to be informed when a slot in the sound queue becomes available, we can go away and sunbathe and BASIC will wake us up with a cup of tea when all is ready. This is effectively an interrupt, similar to the AFTER command we saw in chapter 2.

So if we are feeding notes to channel 1 we say

```
ON SQ(1) GOSUB 3000
```

and whenever there is a spare slot in the channel 1 queue our program will be interrupted and we will be diverted to 3000 where we can put another note onto the queue with the SOUND command and get back to whatever we were doing with a RETURN statement.

Before we reach the RETURN however we will normally want to issue another ON SQ(1) GOSUB 3000 command so that the interrupt mechanism is primed again. That is, assuming there are more notes to play. Simple!

This facility is particularly useful when we are adding music as the background to some other activity such as a game. It is, as far as I know, unique to Amstrad BASIC.

The following simple example will illustrate the above points:

```
10 ON SQ(1) GOSUB 3000
20 GOTO 20

3000 REM ADD TO CHANNEL 1
3010 SOUND 1,284+J*8,300,4
3020 J=J+1
3030 PRINT J
3040 ON SQ(1) GOSUB 3000
3050 RETURN
```

You will see the first few numbers being printed out very quickly as the first sounds are added to the queue, then the action is very much slower. Of course a real program would make line 20 do some useful work instead of looping.

Synchronisation.

Up to now we have only used a single channel, channel 1 or A. Let us branch out and try channel B and C as well:

```
10 SOUND 1,284,200
20 SOUND 2,338,200
30 SOUND 4,426,200
```

Note that channel C is not 3, as you might expect but 4, since this is a 'bit-sensitive' parameter, and:

```
1 turns on bit 0 channel A
2 turns on bit 1 channel B
4 turns on bit 2 channel C
8 turns on bit 3 synchronise with channel A
etc.
```

The above chord (a D-major triad if you are interested!) seemed to start all 3 notes simultaneously, but that is only because the BASIC is quick. Insert:

```
25 FOR J = 1 TO 500 : NEXT J
```

and you will see the problem. But we can avoid any trouble by using the synchronisation feature.

Normally, when a sound is added to the queue by the sound command, it plays as soon as it reaches the top of the queue. But you can choose to attach extra bits or flags, which alter matters.

If you send a sound to channel A but tell it to synchronise with channel B it will wait obediently in the queue until there is a sound at the head of the channel B queue which has been told to wait for channel A! Then, and only then, will the 2 sound queues release their notes and a glorious synchronised chord be heard.

Try:

```
SOUND 33,284,200
```

Nothing will happen. We have entered a sound onto the sound queue, but we have turned on bit 5 of the first parameter as well as bit 1, and bit 5 means 'synchronise with channel C'.

If we now type:

```
SOUND 4,426,200
```

we hear a note on channel C, but there is still nothing from channel A, because the note we sent to channel C said nothing about synchronising with A. We need to type:

```
SOUND 12,426,200
```

and this time we hear 2 notes sounded, because '12' is bits 3 and 4, and means 'send to channel C and synchronise with channel A'.

The same technique is available for channel B so we can send sounds to different channels at different times knowing they will sound together. What a relief!

This is not only vital for the first note of a piece of music, but also for subsequent chords. In most pieces there will be notes of different duration in the various voices, and channel A may be wanting to play 32 notes in the time channel B is playing 8 and channel C 6! So some notes will be synchronised and some not.

A Framework for Harmony.

Armed with fresh knowledge, we will now write a harness that will enable us to write tunes in 2 or 3 parts using synchronisation and sound queues. The tune chosen to test our harness is a Bourrée by Handel.

(This program is available on tape from Kuma Computers).

```
5 REM copyright (c) 1984 John Braga
10 REM play a tune for 1 to 3 voices
15 TRUE=-1 : FALSE=0 :DONE(1)=TRUE:DONE(
2)=TRUE:DONE(4)=TRUE
20 NUMVOICE =3
30 syncha=8 : synchb=16 : synchc=32
40 TEMPO=4
50 GOSUB 1000 : REM initialise
60 ON 4-numvoice GOTO 70,80,90
70 ON SQ(4) GOSUB 4100
80 ON SQ(2) GOSUB 4050
90 ON SQ(1) GOSUB 4000
100 IF DONE(1) AND DONE(2) AND DONE(4) T
HEN 110 ELSE 100
110 INPUT "Again";z$
120 IF UPPER$(z$)<>"Y" THEN 200
130 done(1)=0:done(2)=0:done(4)=0:note(1
)=0:note(2)=0:note(4)=0:GOTO 60
200 END
1000 REM initialise
1010 DIM period(4,200),dur(4,200),synch(
4,200)
1020 FOR j = 1 TO numvoice
1030 channel = 2^(j-1)
1040 numnotes=0
1050 DONE(CHANNEL)=FALSE
1060 IF CHANNEL=1 THEN RESTORE 5000
1070 IF CHANNEL=2 THEN RESTORE 5400
1080 IF CHANNEL=4 THEN RESTORE 5700
```

```
1100 READ note$,duration,synch$
1110 IF note$="" THEN 1500
1120 GOSUB 2000 : REM get period from no
te$
1200 numnotes = numnotes+1
1210 period(channel,numnotes)=period
1220 dur(channel,numnotes)=durATION*temp
o
1230 synch(channel,numnotes)=synch
1240 GOTO 1100
1500 PERIOD(CHANNEL,0) = NUMNOTES
1510 NEXT J
1600 ENT -1,1,1,7,1,-1,7
1999 RETURN
2000 REM calculate octave and note from
note$
2010 Z$=LEFT$(NOTE$,1)
2020 note=INSTR("C D EF G A BR",Z$)-1
2025 IF NOTE=12 THEN OCTAVE=0 : GOTO 210
0
2030 IF NOTE < 0 THEN OCTAVE = 5 : DUR =
 1 : GOTO 2999
2040 Z$=RIGHT$(NOTE$,1)
2050 IF Z$="£" THEN NOTE = NOTE + 1
2060 IF Z$="b" THEN NOTE = NOTE - 1
2080 OCTAVE = VAL(MID$(NOTE$,2,1))-3 : R
EM IF OCTAVE < 0 THEN OCTAVE = 1
2090 IF NOTE < 0 THEN NOTE = 0 : OCTAVE
= OCTAVE - 1
2100 GOSUB 3000
2110 SYNCH=0
2120 FOR J1 = 1 TO LEN(SYNCH$)
2130 IF MID$(SYNCH$,J1,1)="A" THEN SYNCH
=SYNCH+SYNCHA
2140 IF MID$(SYNCH$,J1,1)="B" THEN SYNCH
=SYNCH+SYNCHB
2150 IF MID$(SYNCH$,J1,1)="C" THEN SYNCH
=SYNCH+SYNCHC
2160 NEXT J1
2999 RETURN
3000 REM calculate period from octave an
d note
3005 IF NOTE=12 THEN PERIOD=0 : GOTO 399
9
3010 freq=261.626*(2^(octave+note/12))
3020 period = ROUND(125000/freq)
3999 RETURN
4000 channel=1
4040 GOTO 4200
```

```
4050 channel=2
4090 GOTO 4200
4100 channel=4
4200 NOTE(CHANNEL)=NOTE(CHANNEL)+1
4205 vol=5
4207 TONE=0
4210 SOUND CHANNEL+SYNCH(CHANNEL,NOTE(CH
ANNEL)),PERIOD(CHANNEL,NOTE(CHANNEL)),DU
R(CHANNEL,NOTE(CHANNEL)),VOL,0,TONE
4220 IF NOTE(CHANNEL) >= PERIOD(CHANNEL,
0) THEN DONE(CHANNEL) = TRUE : GOTO 4999
4300 J=CHANNEL
4310 IF J>3 THEN J=3
4320 ON J GOTO 4400,4420,4440
4400 ON SQ(1) GOSUB 4000
4410 GOTO 4999
4420 ON SQ(2) GOSUB 4050
4430 GOTO 4999
4440 ON SQ(4) GOSUB 4100
4999 RETURN
5000 REM data for channel a
5010 DATA D5,6,C,R,2,,D5,7,C,R,1,,B4,6,C
,R,2,,C5,4,C,B4,4,,A4,4,C,G4,3,,R,1," "
5015 DATA E5,6,C,R,2,,G5,14,C,R,2,,F5£,4
,C,E5,3,,R,1," "
5020 DATA D5,5,C,R,3,,C5,4,C,B4,3,,R,1,,
A4,4,C,B4,4,,C5,4,C,A4,3,,R,1," "
5025 DATA B4,6,C,R,2,,G4,14,C,R,2,,A4,6,
C,R,2," "
5030 DATA B4,4,C,C5£,4,,D5,4,C,B4,3,,R,1
,,C5£,4,C,D5,4,,E5,4,C,C5£,3,,R,1," "
5035 DATA D5,4,C,E5,4,,F5£,4,C,D5,3,,R,1
,,E5,4,C,F5£,4,,G5,4,C,E5,3,,R,1," "
5040 DATA F5£,4,C,G5,4,,A5,4,C,R,4,,A4,4
,C,R,4,,C5£,4,C,R,4,,D5,20;C,R,4," "
5050 DATA A5,6,C,R,2,,A5,8,C,F5£,6,C,R,2
,,G5,4,C,F5£,4,,E5,4,C,D5,3,,R,1," "
5055 DATA G5,6,C,R,2,,B5,14,C,R,2,,E5,7,
C,R,1," "
5060 DATA D5£,8,C,E5,8,C,F5£,7,C,R,1,,G5
,4,C,A5,3,,R,1," "
5065 DATA G5,6,C,R,2,,E5,14,C,R,2,,D5,14
,C,R,2," "
5070 DATA C5,4,C,B4,3,,R,1,,C5,5,C,R,3,,
C5,14,C
5075 DATA R,2,,B4,4,C,A4,3,,R,1,,B4,6,C,
R,2,,D5,6,C,R,2," "
```

```
5080 DATA E5,4,C,F5£,4,,G5,4,C,E5,3,,R,1
,,F5£,4,C,G5,4,,A5,4,C,F5£,3,,R,1," "
5085 DATA G5,4,C,A5,4,,B5,4,C,G5,3,,R,1,
,A5,4,C,B5,4,,C6,4,C,A5,3,,R,1," "
5090 DATA B5,5,C,R,3,,A5,4,C,G5,3,,R,1,,
F5£,4,C,G5,4,,A5,4,C,F5£,3,,R,1," "
5095 DATA G5,4,C,A5,4,,B5,4,C,G5,3,,R,1,
,A5,6,C,R,2,,C6,6,C,R,2," "
5100 DATA B5,6,C,R,2,,A5,4,C,G5,3,,R,1,,
F5£,4,C,G5,4,,A5,4,C,F5£,3,,R,1," "
5105 DATA G5,4,C,A5,4,,B5,4,C,G5,3,,R,1,
,A5,4,C,B5,4,,C6,4,C,A5,3,,R,1," "
5110 DATA B5,4,C,C6,4,,D6,7,C,R,1,,B5,7,
C,R,1,,A5,4,C,G5,4," "
5115 DATA R,2,BC,R,6,,G5,24," "
5399 DATA "",0," "
5400 REM DATA FOR VOICE B
5410 DATA R,2,AC,R,3,,B3,27," "
5699 DATA "",0," "
5700 REM DATA FOR VOICE C
5705 DATA G3,5,A,F3£,7,A,G3,5,A,D4,6,A,E
4,5,A
5710 DATA C3,6,A,C4,5,A,R,3,,B3,4,,C4,5,
A
5715 DATA F3£,6,A,G3,6,A,C3,6,A,D3,6,A,G
2,6,A,G3,14,A,D3,6,A
5720 DATA G3,6,A,F3£,6,A,E3,6,A,A3,5,A,F
3£,6,A,D4,5,A,C4£,6,A,A3,5,A,D4,4,A,F3£,
4,A,A3,4,A,A2,4,A,D3,8,A,D4,4,,C4£,4,,D4
,6," "
5725 DATA D3,5,A,D4,7,A,A3,5,A,D4,6,A,C4
,5,A
5730 DATA B3,5,A,G3,4,A,A3,4,,B3,8,,C4,7
,A
5735 DATA B3,8,A,E3,8,A,A3,8,A,B3,7,A
5740 DATA E3,4,A,R,4,,E4,14,A,B3,6,A,R,2
," "
5745 DATA E3,8,,G3£,8,A,A3,6,A,A2,6,A,R,
2," "
5750 DATA D3,8,,F3£,8,A,G3,6,A,B2,4,A
5755 DATA C3,4,A,A3,4,A,D4,4,A,C4,4,A
5760 DATA B3,4,A,G4,4,A,F4£,4,A,D4,4,A
5765 DATA G3,4,A,B3,4,A,D4,4,A,D3,4,A
5770 DATA E3,4,A,C4,4,A,F3£,4,A,D4,4,A
5775 DATA G3,4,A,B3,4,A,D4,4,A,C4,4,A
5780 DATA B3,4,A,G3,4,A,F3£,4,A,D3,4,A
5785 DATA G3,5,A,B3,5,A,D4,5,A,D3,5,A
5790 DATA R,2,AB,G3,30," "
5899 DATA "",0," "
```

Bourrée Commentary.

The main body of the program is from 10 to 200. Some program variables are initialised and a jump is made to 1000 subroutine, where arrays containing the notes are set up. Then the interrupt subroutines are primed in lines 70-90 and the program loops at statement 100 until all voices have finished playing.

Subroutine 1000-1999 sets up 3 arrays, PERIOD, DUR and SYNCH, containing values for each note for each voice. The values are period, duration and synchronisation details. The input for the notes is read from data statements at 5000- and the subroutine at 2000-2999 is used to convert a note into a period value. The RESTORE statements at 1060-1080 are redundant if all 3 voices are played, but are useful if you are isolating 1 alone (see below).

Subroutine 3000-3999 is called from subroutine 2000.

Subroutine 4000-4999 contains statements that are called by BASIC whenever there is a spare space on the queue. It contains the SOUND command that puts the notes from the arrays set up by 1000-1999 onto the sound queues. Note that the music interrupt routine has to reprime itself before exiting with a RETURN.

The DATA statements start at 5000 and are split into 3, one section for each voice. Note the use of R to create a 'rest', ie. period of silence.

Interpretation.

It is fascinating how much variation can be given to the Bourrée (and of course every other tune) by small changes to the DATA statements. The following records some of the changes I experimented with; there are many other things you might care to try.

You may wonder why the above DATA statements contain so many rests, and why crotchets are given duration values of anything from 4 to 8. The answer is that some notes are intended to be played 'staccato' (short and detached) and are given smaller durations. The full value is made up by a following rest in the case of channel A, but for channel C this is not usually necessary since synchronisation causes the channel to wait for channel A anyway.

Very short rests are useful for causing the note following to sound 'articulated' rather than slurred. There is considerable scope for experimenting with note durations and rests in the above piece. You can change the interpretation markedly if you wish.

A tone envelope is used by channels B and C but not by A, since I feel the smallest vibrato achievable is excessive on the higher notes. One would ideally like to vibrate with units of less that 1 period in the upper register, and this is unfortunately not possible. But experiment with the ENT command at 1600 and with statement 4207 by all means. Try:

```
4207 TONE=0
```

for no vibrato, followed by:

```
4207 IF CHANNEL=4 THEN TONE=1 ELSE TONE=0
```

for a vibrato on the lower channel only, followed by:

```
4207 TONE=1
```

for a vibrato on all 3 - or try another ENT command altogether!

Experiment too with the volume control in statement 4205. There is no ENV command but you can try making the top line more prominent if you wish. Try:

```
4205 IF CHANNEL=1 THEN VOL=5 ELSE VOL=4
```

or something of the sort.

Note how the last bar is entered (statements 5115, 5410 and 5790). There is a short synchronised rest at the start of the bar which is, strictly speaking, out of tempo. This gives the effect of a slight slowing down. Then the notes are slightly 'splayed' from the bass up to give an arpeggio effect. This style is particularly common in harpsichord music, for which instrument this bourrée is written. Finally the last note is given a longer duration than written. This is not a mistake!

Entering a Tune.

Of course the above program can be used as the basis for creating many pieces of music. In most cases all that will be needed is to enter fresh data statements at 5000-5399 (channel A), 5400-5699 (channel B) and 5700-5999 (channel C).

Creating a tune is a laborious process and is best done in stages. First load the Bourree program and delete all the data lines except the 'terminators' at 5399, 5699 and 5999.

Enter the data statements for the channel A voice first and then play them using the following program alterations:

```
20 NUMVOICE = 1
30 REM ...
```

which effectively removes the synchronisation. Check the values of the notes and their duration by ear. Experiment with small rests to get the articulation right. You will find it helps considerably if you keep the data statements short, usually 1 per bar; this makes it easier to track mistakes. You can alter 2 statements which help 'debug' the tune:

```
1105 PRINT NOTE$;" ";
1505 PRINT
```

Then insert the bass part. It is of little use playing this on its own since it will be synchronised with channel A, and will sound very strange if you REM statement 30. But alter lines 20-30 back to their correct values and enter:

```
4205 IF CHANNEL=4 THEN VOL=4 ELSE VOL=0
```

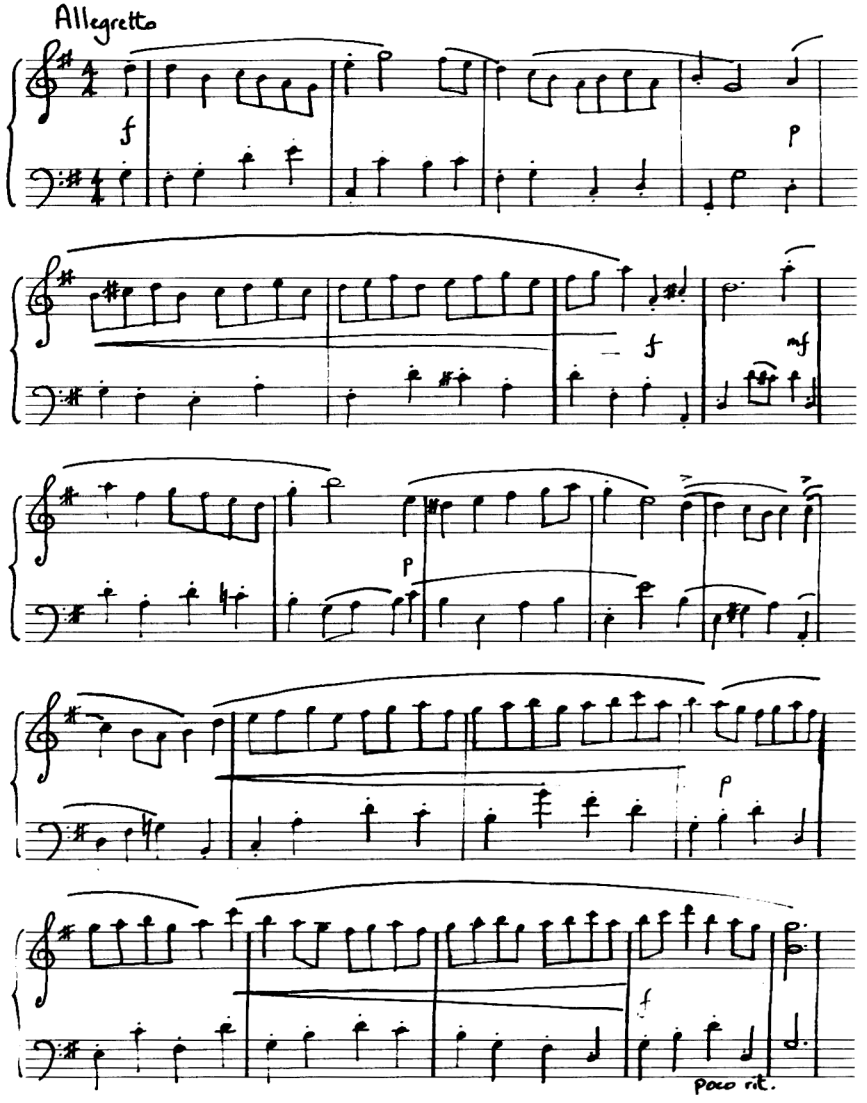
which makes it possible to hear the bass part only. Again, juggle with the articulation. Then change line 4205 so that you hear both parts together.

Add the middle part if required in the same way, playing it first separately, then with the top part, and finally with both other parts. All this can be done by use of statement 4205.

Musical Judgement.

I hope the preceding dissertation shows clearly how difficult it would be to make a computer play classical music 'musically'. The more one tries to create a pleasant effect, the more one realises how much the performer deviates from the written page in the interests of musicianship. A musician, though, has trained to a pitch whereby he does not need to think consciously about altering the duration of a note, he knows that it 'sounds right'. We, on the other hand, have to find and edit every note in a string of DATA statements!

Several people may criticise the above program on the grounds that the sound is 'electronic' and the effect mechanical despite our efforts to humanise it. It is perfectly true that the musical effect will never approach realism. But surely the value lies in our appreciating precisely that, and in struggling to recognise what the human performer adds so that we can get closer to what he achieves. Our appreciation of music and of the musician's art can only be enhanced by the exercise.


Allegretto
f
p
f
mf
p
p
poco rit.

CHAPTER 10
SPECIAL EFFECTS

Up to now we have considered the sound chip almost purely from a musical angle. How about its use for embellishing games?

The Noise Channel.

There is a further parameter to the SOUND command which we have not yet used in this book. A noise period can be added to the sound and the parameter varies from 1 to 31. Try the following to see the effect:

```
10 FOR J= 1 TO 31
20 SOUND 1,0,100,4,0,0,J
30 NEXT J
```

and you will see that the noise gets 'darker' the higher the noise period number. Try:

```
5 FOR J=1 TO 2
10 FOR J1=16 TO 31
20 SOUND 1,0,5+J1,5,0,0,J1
30 NEXT J1
40 FOR J1=31 TO 12 STEP -1
50 SOUND 1,0,7,6,0,0,J1
60 NEXT J1
70 NEXT J
```

and you will hear the gale lashing itself into a fury...

Footsteps, Bombs, Saws...

The above example used the noise channel on its own. However it can be combined with normal tones. Try:

```
10 FOR VOL=1 TO 7 STEP 0.25
20 SOUND 1,95,1,VOL,0,0,1
30 SOUND 1,400,1,VOL,0,0,31
40 SOUND 1,0,20,0
50 NEXT VOL
```

This simple example lends itself to many variations and can be used to effect in many games to resemble footsteps, ticking bombs, watches, etc. (You need to have a fairly strong imagination in some

cases).

The next sound reminds me of a circular saw starting on some tough timber:

```
10 ENT 1,10,1,8
15 ENV 1,5,1,5,1,0,200
20 SOUND 1,50,200,6,1,1,3
30 SOUND 1,48,50,5,1,1,6
35 SOUND 1,48,10,5,1,1,1
40 SOUND 1,50,150,6,1,1,3
```

I leave it to you to put names to the following hideous noises, but I expect you can find uses for them...

```
10 FOR J=1 TO 5
20 ENV 1,1,1,20
30 ENT -1,1,3,1
40 SOUND 1,284,-5,10,1,1,3
50 NEXT J
```

```
10 ENT -1,1,4,2,1,-4,2
20 SOUND 1,50,100,4,0,1
```

This is a prime area for experimentation. There are few rules, but a clear understanding of the syntax of the various sound commands will help you discover many new and fascinating sounds.

Hold and Flush.

When it comes to adding sound to games, there is a further trick to the versatile SOUND command that we have not yet discussed. You will remember that each channel has a sound queue. If you issue a sound with a 'flush' bit on, any sounds already in the queue, and even those currently sounding, are instantly terminated and the sound is heard. So if you are synchronising a bomb explosion to some action on the screen you can link the sound directly to the vital keystroke.

The flush bit is bit 7 so you might have a statement such as

```
10 FLUSH = 128
...
1010 SOUND 1+FLUSH,...
```

Bit 64 is called the hold bit, and if on causes the sound (and any in the queue behind it) to sit and wait in the queue until released. The command to release is, not surprisingly, the RELEASE command, so:

```
RELEASE 4
```

will release any sounds waiting on channel C without affecting any other channels. So you can prime the sound queues and then by a single RELEASE, let all hell loose...

If there are no sounds waiting on the channel at the time the RELEASE is issued, no harm will be done.

PART III

THE GRAPHIC POSSIBILITIES

This part returns to the delights of the Amstrad Graphics, and explores some possibilities in colour manipulation and animation, besides giving examples of the use of graphics for graphs and charts.

CHAPTER 11
ANIMATION AND ILLUSION

In the simplest form of animation we want to move a character round the screen. To move a man across the screen we will need to print him, then blank him out, and print him in the next square.

```
5  MAN$ = CHR$(250)
10 CLS
20 Y = 23 : LOCATE 1,Y : PRINT MAN$;
30 FOR X = 1 TO 39
40 LOCATE X,Y
50 PRINT " ";MAN$;
60 NEXT X
```

You may notice that the figure seems to move in a slightly jerky fashion, with gaps. If you insert:

```
45 CALL &BD19
```

the action is noticeably smoother. This is because you are now using an operating system routine which waits for 'Frame Flyback' to occur. This is a time when the system is not refreshing the screen, so it is the best time for you to write to the screen.

Frame Flyback is time critical, and therefore the results will be different according to the location you are addressing. Alter line 20 to

```
20 Y = 8 ...
```

and you will see what I mean! So some experimentation is called for.

A Bouncing Ball.

Extending the simple technique shown above we can write a program to send a ball bouncing around the screen. When it hits the border it bounces back at a suitable angle. The basic program is:

```
10  CLS
20  BALL$ = CHR$(231)
30  X = 20 : Y = 12
40  XDIR = 1 : YDIR = 1
50  LOCATE X,Y : PRINT " ";
60  X = X+XDIR : Y = Y+YDIR
70  IF X<1 THEN X =  1 : XDIR = 1
80  IF X>40 THEN X = 40 : XDIR = -1
90  IF Y<1 THEN Y = 1 : YDIR = 1
100 IF YDIR>25 THEN Y = 25 : YDIR = -1
110 LOCATE X,Y
120 PRINT BALL$;
130 GOTO 50
```

This works, but could be improved. Try adding

```
15  BORDER = 6
115 CALL &BD19
```

On some lines the ball is still being obliterated before it has been seen! Try slowing it down a little.

```
125 FOR J = 1 TO 5 : NEXT J
```

Much better! If anyone wants a game using a bouncing lemon, we have just the thing...

The line at 125 is rather wasteful. It would be better in a real game to make the above routine a separate subtask which could be kicked off at suitable intervals such as

```
EVERY 10 GOSUB ...
```

and a little experimenting would give us the most suitable speed.

How about adding some variety, some obstacles to divert our ball? At present we are changing direction only if we have reached the edge of the screen. To handle obstacles we will need some method of testing whether the screen position into which we want to move is clear before we actually make the jump. This is not provided by BASIC (unless you count the TEST command which could be pressed into service), but there is an operating system routine call TXT RD CHAR which does exactly what we want by reading and returning the character at the current location. The games in chapters 12 and 13 both use that routine and you can copy it into your own programs.

Juggling Screens and Inks.

The technique of blanking and drawing is used in many games. It is simple to program and effective in use, but it does of course have limitations, since if what you are trying to move is bigger than one character you will have to blank out several squares and redraw them, making a slower routine.

There are other ways of of making objects move around the screen. In chapter 5 it was shown that we could keep 2 screen areas in memory and flit between them. This was virtually instantaneous, so could give the impression of fast movement. Draw an object in one place using the normal screen area at &C000, and draw it at a slightly different place using the area at &4000. Then merely switch from one to the other.

That gives us one advantage - the object can be far bigger than one character. But it also has a disadvantage:- it will usually take a few seconds to draw the character in the first place so repositioning the character can be slow. Advanced programmers could expand this idea by scrolling the screen area so that the character appears to move, or by POKING directly into one screen area while the other area is being displayed. Such techniques are generally found in the better arcade games, and are certainly programmed in assembly language.

At the end of chapter 4 we touched on the judicious use of inks to create the illusion of appearing and disappearing objects. If we are in mode 1 we have 4 colours at our disposal. One is normally the background, so that leaves 3 to draw with. If we set them all to the same colour as the background, we can draw 3 different (non-overlapping) figures on the screen, one with each pen colour, but none will appear. The screen is apparently blank. Then by turning each ink on and off in succession we can make an object appear and disappear at will, ie. to move around!

If of course we use mode 0, and 2 screen areas, we can draw 15 figures in each area and show a figure in 30 different locations...

Multiple Planes.

All this may be very useful in some circumstances, but you will have noticed that the figures must not overlap, or one deletes the other. How can we create the illusion of one object appearing from behind another?

Let us imagine 3 planes, a background, midground and foreground. What we need to do is to be able to show:

- background (a single colour)
- an object in midground (obscuring the background)
- an object in foreground (obscuring the background)
- an object in foreground (obscuring both the midground and the background).

ie. 4 separate states. And in mode 1 we have 4 separate inks, which should set you thinking. How about using ink 0 for the background, ink 1 for the midground, and inks 2 and 3 for the 2 states of the foreground?

Sounds promising, but how can we decide quickly, when writing in the foreground, whether we need to use ink 2 or ink 3 in a particular place? And how about deleting an object from the foreground without also deleting the midground figure it may be hiding?

There has to be an answer to this, and of course there is (otherwise I would hardly have bothered leading up to it!) The answer lies in the 'Graphics Ink Mode'.

We have not mentioned this up to now. When you write a character or draw a line using, say, ink 1, you will perhaps have assumed that all the necessary pixels along the route were set to ink 1. And so indeed they are if Graphics Ink Mode is set to normal mode, but not necessarily if it is set to another state! It has 4 states and in 3 of them the new ink reacts in some way with the old.

State 0 - (Force Mode). The new ink overwrites the old. This is the normal state, and the one we have always used up to now.

State 1 - (XOR mode). The new ink is exclusive-ORed with the old ink and the pixel drawn in the resulting ink.

State 2 - (AND mode). The new ink is ANDed with the old ink.

State 3 - (OR mode). The new ink is ORed with the old ink.

(If you need to revise the meanings of AND, OR and XOR see the Amstrad handbook, chapter 4)

So let us imagine we have set the graphics ink mode to state 3 - the OR mode. Now if we have our background in ink 0 and our midground in ink 1 we can write our foreground in ink 2. We never actually write with ink 3 at all. The result will be:

Old Ink	New Ink	Result
0 (background)	0	0 (background)
0	1	1 (midground)
0	2	2 (foreground)
1 (midground)	1	1 (midground)
1	2	3 (foreground obscuring midground)

Inks 2 and 3 must be set to the same colour of course, so that they look the same, but they are in fact different, since we must be able to tell the difference when we come to delete them! As far as we are concerned, though, we set our pen to ink 2 and write merrily away, with the resulting pixels being set to ink 2 or 3 as dictated by the existing colour at that location.

What we have done is sacrificed a colour from our set of 4 and substituted the ability to handle 3 planes instead.

How about deleting a foreground object that partially obscures a midground object? Or deleting a midground object without leaving a hole in the foreground? No problem. We merely switch the graphics ink mode to state 2 (AND mode) and select the appropriate pen colour, 2 (for midground delete) or 1 (for foreground delete). Suppose we wish to delete a midground object. We use ink 2 in our pen and the results are:

Old Ink	New Ink	Result
1 (midground)	2	0 (background)
3 (foreground obscuring midground)	2	2 (foreground)

So foreground is apparently untouched (though in reality changed from 3 to 2). Similarly, if we stay in Graphics Ink State 2 and use ink 1 in our pen, we can delete a foreground object without disturbing the midground:

Old Ink	New Ink	Result
2 (foreground)	1	0 (background)
3 (foreground obscuring midground)	1	1 (midground)

The following program is useful to verify the above. Note that transparency is also set on.

```
10  MODE 1
20  INK 0,1 : INK 1,24 : INK 2,20 : INK 3,6
     PAPER 0 : PEN 1
25  PRINT CHR$(22)+CHR$(1); : REM SET ON
     TRANSPARENCY
30  INK 1,6 : INK 2,24
35  FOREPEN = 2 : MIDPEN = 1 : FORERUB =
     1 : MIDRUB = 2
40  PRINT CHR$(23)+CHR$(3);: REM SELECT
     GRAPHICS INK STATE 3 (OR) FOR
     DRAWING FOREGROUND OR MIDGROUND
     FIGURES OR TEXT.
50  PENCIL = FOREPEN : GOSUB 1000 : REM
     DRAW IN FOREGROUND
60  PENCIL = MIDPEN : GOSUB 2000 : REM
     DRAW IN MIDGROUND
65  LOCATE 10,25 : PRINT "Press any key ";
70  Z$=INKEY$ : IF Z$="" THEN 70
80  PRINT CHR$(23)+CHR$(2); : REM SELECT
     GRAPHICS INK STATE 2 (AND) FOR
     DELETING FOREGROUND OR MIDGROUND
     FIGURES OR TEXT.
90  PENCIL = MIDRUB : 'Gosub' 2000 : REM
     RUB OUT MIDGROUND FIGURE
100 PRINT CHR$(22)+CHR$(0); : REM SWITCH
     OFF TRANSPARENCY
110 PRINT CHR$(23)+CHR$(0); : REM SELECT
     NORMAL GRAPHICS INK STATE (FORCE)
120 END
1000 REM DRAW OR UNDRAW TEXT
1010 LOCATE 20,12
1020 PEN PENCIL
1030 PRINT "      This text is in the
      foreground";
1999 RETURN
```

```
2000 REM DRAW OR UNDRAW TRIANGLE
2010 MOVE 100,100
2020 540,100,PENCIL
2030 FOR J = 100 TO 540 STEP 2
2040 MOVE 320,360
2050 DRAW J,100,PENCIL
2060 NEXT J
2999 RETURN
```

This technique is a powerful one and well worth exploring!

Now you see it, now you don't!

The same sort of trick is used in the following program which shows how 2 figures can be drawn in the same location, then displayed alternately to give the impression of vanishing and reappearing:

```
10 MODE 1 : DEG
20 GOSUB 1000 : REM DRAW IMAGE 1
30 GOSUB 2000 : REM DRAW IMAGE 2
40 GOSUB 6000 : REM DRAW CAPTION
50 FOR SCREEN = 1 TO 10
60   GOSUB 3000 : REM DISPLAY IMAGE 1
70   GOSUB 4000 : REM DELAY
80   GOSUB 5000 : REM DISPLAY IMAGE 2
90   GOSUB 4000 : REM DELAY
100 NEXT SCREEN
110 INK 0,1
120 INK 1,24
130 INK 2,20
140 INK 3,6
150 PAPER 0
160 PEN 1
170 PRINT CHR$(23)+CHR$(0);
180 END
1000 REM DRAW IMAGE 1
1010 PRINT CHR$(23)+CHR$(3);
1020 FOR Z = 1 TO 360
1030  PLOT 320,200,1
1040  DRAW 320+90*COS(Z),200+90*SIN(Z),1
1050 NEXT Z
1999 RETURN
2000 REM DRAW IMAGE 2
2010 PRINT CHR$(23)+CHR$(3);
2020 PLOT 100,100,2
2030 DRAW 540,100,2
2040 DRAW 320,360,2
2050 DRAW 100,100,2
2060 MOVE 320,100
2070 DRAW 320,360,2
2999 RETURN
```

```
3000 REM DISPLAY IMAGE 1
3010 INK 0,2
3020 INK 1,1
3030 INK 2,2
3040 INK 3,1
3999 RETURN
4000 REM DELAY
4010 TM=TIME+300
4020 WHILE TIME < TM
4030 WEND
4999 RETURN
5000 REM DISPLAY IMAGE 2
5010 INK 0,4
5020 INK 1,4
5030 INK 2,0
5040 INK 3,0
5999 RETURN
6000 REM PRINT CAPTION
6010 LOCATE 18,24
6020 PRINT CHR$(23)+CHR$(0);
6030 PEN 3
6040 PRINT "This is a caption"
6999 RETURN
```

Note that the above program changed the background, but that is not necessary, and it will improve the illusion in many cases if you keep the background the same.

Note also that the caption appears with both images.

The above techniques are only small examples of what can be achieved. An understanding of these methods is essential if you want to experiment with 3-D drawing, rotating solid objects, proportional drawing, etc., etc.

Future Horizons.

Before we leave this subject, consider for a moment the fact that we have been using mode 1. How about experimenting with mode 0, which would allow us the use of 16 inks. Instead of 3 planes, how about 4? We would need:

Background
Rearground
Midground
Midground obscuring rearground
Foreground
Foreground obscuring rearground
Foreground obscuring midground and rearground

That is 7 of our inks. We could perhaps use the others for different foreground colours, or for more than one midground colour, or for making foreground objects appear and disappear, etc. You will need pencil and paper to experiment with the effects of ORing various inks. Draw a column for the old ink, one for the new and one for the result... You will find many combinations are possible.

CHAPTER 12
GAME NUMBER 1 – HUNGRY HEFFALUMP

Computer Games - Arcane or Arcade?

I make no apologies for presenting games! Properly viewed they can be highly amusing and educational. The programming skills needed to make an attractive game are of the highest, and many programmers have learned the skills precisely because they were driven to create better and better games.

People who denigrate arcade games generally see only the (admittedly limited) end-result and miss the effort that has gone into its creation. Most programmers spend very little time playing the game that results from their labours, they are too busy planning the next achievement.

With apologies to A.A.Milne.

The first game we have depicts a simple-minded heffalump that has its lair in the north-east corner of a wood. As I need hardly remind you, a heffalump's sight is poor and he relies heavily on sense of smell to search out his prey.

His victims in this case are a group of explorers who are trying, one by one, to cross the wood from the north-west corner to the south-east, where there is a river. The heffalump has a distinct aversion to water, so if they can plunge in, they are safe and can whistle a scornful victory theme to show their relief.

You, the player, control the direction of the men by using the arrow keys. Once set in one direction the men rush straight ahead in blind panic, rebounding off trees, and no doubt looking fearfully over their shoulder for signs of the blundering monster. They seem to be committed to taking a direct approach, rarely deviating from a straight line, and showing a strong liking for right angles.

The monster, proceeding by smell, is equally prone to blundering into trees, but possessing an even smaller brain than his potential victims, he does not seek a way round obstacles, but pushes in frustration in the direction he feels he ought to go. As the game proceeds, however, his speed over the ground improves dramatically, as a result either of rage at his victims escaping, or of an increasing taste for blood. As the last man leaves the saftey of the edge, he is probably doomed, for the beast seems possessed of a frantic energy...

By now you should have gained an impression of the intellectual depth of this game, and we can proceed to the programming.

Program Commentary.

The main program thread is a short one, from 5 to 170. Lines 10 - 40 set the scene, lines 50 to 120 form the body of the program and lines 130 to 170 are the conclusion.

```
5 REM HUNGRY HEFFA COPYRIGHT (C) 1984 JOHN
                              BRAGA
10 GOSUB 1000 : REM INITIALISE
20 GOSUB 2000 : REM DRAW BORDER
30 GOSUB 2500 : REM DRAW FOREST
40 GOSUB 3000 : REM PLACE HEFFA
50 FOR MAN = 1 TO NUMBEROFMEN
60   GOSUB 3400 : REM START MAN
70   GOSUB 3800 : REM START HEFFA
80   GOSUB 4000 : REM MOVE MAN
90   GOSUB 5000 : REM CLEAN UP
120 NEXT MAN
130 GOSUB 6000 : REM FINAL MESSAGE
140 CLS
150 PEN 1
160 SPEED KEY 20,3
170 END
```

The conclusion could restore the original colours if you wish.

```
1000 REM INITIALISE
1010 MAN$=CHR$(249) : EDGE$=CHR$(143)
1020 NUMBEROFMEN = 10
1040 TREE$ = CHR$(229) : NUMBEROFTREES = 40
1050 RIVER$=CHR$(207) : HEFFA$=CHR$(184)
1060 TOPLINE=2: BOTTOMLINE=25 : LEFTLINE = 1:
     RIGHTLINE = 40
1070 HEFFAINTERVAL = 5
1080 TRUE = -1 : FALSE = 0
1090 ESCAPED = 0 : CAPTURED = 0
1100 MODE 1
1110 INK 0,9
1120 PAPER 0
1130 INK 1,24
1140 PEN 1
1150 INK 2,20
1160 INK 3,6
1170 SPEED KEY 4,2
1200 MEMORY &AB77
1210 RESTORE 1250
1220 FOR J = 1 TO 8
1230 READ X : POKE HIMEM+J,X : NEXT J
1240 DATA &CD,&60,&BB,&32,&7F,&AB,&C9,0
1250 RDCHAR = &AB78 : CHREAD = &AB7F
1999 RETURN
```

1000 is the initialision subroutine. Since it is only executed once, it might be more correct to include it in the main line code, but it is tidier to keep all initialisation statements in one place.

It is expected that you will want to experiment with many of the variables in this game, and to add new embellishments. Numberofmen in line 1020 can be set to any suitable number.

In line 1060 the borders of the game are set, but you could experiment with mode 2 and an 80-column game by altering this line and line 1100. Of course you will have to alter the INK and PEN statements to take account of the fact you will then have only 2 colours.

Line 1070 helps to control the speed of the heffa's reactions. Experiment according to your preference and skill level. Those who hate losing should set it to 10 or more...See also line 3805.

The SPEED KEY statement takes advantage of the fact that there is a very useful Amstrad feature which governs the speed at which the keys react to your touch. The only snag of this approach is that if you are running this program and have made a typing error, you will find the keyboard is almost uncontrollable! To avoid problems, enter the statements:

```
2 ON ERROR GOTO  200
3 ON BREAK GOSUB 150
200 PRINT "ERROR";ERR;" AT";ERL
210 GOTO 150
```

The Drawborder subroutine is very straightforward and needs no comment:

```
2000 REM DRAWBORDER
2010 FOR J=LEFTLINE TO RIGHTLINE
2020   LOCATE J,TOPLINE : PRINT EDGE$;
2030   LOCATE J,BOTTOMLINE : PRINT EDGE$;
2040 NEXT J
2050 FOR J = TOPLINE TO BOTTOMLINE
2060   LOCATE LEFTLINE,J : PRINT EDGE$;
2070   LOCATE RIGHTLINE,J : PRINT EDGE$;
2080 NEXT J
2100 RIVERXPOS = RIGHTLINE : RIVERYPOS =
          BOTTOMLINE - 3
2110 LOCATE RIVERXPOS,RIVERYPOS : PRINT
          RIVER$;
2499 RETURN
```

Next comes the routine that adds the trees. You should perhaps experiment with the number of trees drawn.

```
2500 REM DRAWFOREST
2510 RANDOMIZE TIME
2520 FOR J = 1 TO NUMBEROFTREES
2530 X%=RND(1)*RIGHTLINE + LEFTLINE: IF
          X%>=RIGHTLINE OR X% <= LEFTLINE
          THEN 2530
2540 Y%=RND(1)*BOTTOMLINE+TOPLINE: IF Y% >=
          BOTTOMLINE OR Y% <= TOPLINE THEN
          2540
2550 LOCATE X%,Y% : PRINT TREE$;
2560 NEXT J
2570 LOCATE RIVERXPOS-1,RIVERYPOS : PRINT
       " ";
2999 RETURN
```

The Randomize statement serves to ensure a different forest on each occasion.

Note statement 2570 that attempts to ensure that victims have a clear escape route!

```
3000 REM PLACE HEFFA
3010 HEFFXPOS = RIGHTLINE-1 : HEFFYPOS =
          TOPLINE + 1
3015 PEN 3
3020 LOCATE HEFFXPOS, HEFFYPOS : PRINT
          HEFFA$;
3399 RETURN
```

A heffa is drawn in bright red in the top righthand corner.

```
3400 REM START MAN
3405 PEN 2
3410 LOCATE LEFTLINE + 1,TOPLINE + 1 : PRINT
     MAN$;
3420 MANXPOS = LEFTLINE + 1 : MANYPOS =
     TOPLINE + 1
3430 SEARCHING = TRUE : CAUGHT = FALSE
3799 RETURN
```

Note that Searching and Caught are used as boolean variables, in a way familiar to any Pascal programmer.

```
3800 REM START HEFFA
3805 J = HEFFINTERVAL+(NUMBEROFMEN-MAN)*2
3810 EVERY J GOSUB 7000
3999 RETURN

7000 REM SEPARATE TASK TO MOVE HEFFA
     TOWARDS MAN
7005 X=HEFFXPOS:Y=HEFFYPOS
7010 IF HEFFXPOS>MANXPOS THEN X=HEFFXPOS-1
     ELSE IF HEFFXPOS<MANXPOS THEN X=
     HEFFXPOS+1
7020 IF HEFFYPOS>MANYPOS THEN Y=HEFFYPOS-1
     ELSE IF HEFFYPOS<MANYPOS THEN Y=
     HEFFYPOS+1
7025 LOCATE X,Y:CALL RDCHAR
7027 IF PEEK(CHREAD)=32 THEN 7030
7028 IF PEEK(CHREAD)=ASC(MAN$) THEN SEARCHING
     = FALSE:CAUGHT=TRUE:CAPTURED=
     CAPTURED+1 ELSE 7999
7030 LOCATE HEFFXPOS,HEFFYPOS:PRINT " ";:
     PEN 3:LOCATE X,Y:PRINT HEFFA$;:
     HEFFXPOS=X:HEFFYPOS=Y
7032 CALL &BD19
7999 RETURN
```

The movement of the heffa is a separate subtask which is initiated by the 3800 subroutine. Line 3805 is crucial to the reaction speed of the heffa, so feel free to slow him down by unfair means.

```
4000 REM MOVE MAN
4005 IF NOT SEARCHING THEN 4999
4007 DI
4010 Z$ = INKEY$ : IF Z$ = "" THEN 4800
4020 Z = ASC(Z$)
4030 IF Z<240 OR Z>243 THEN 4010
4040 Z = Z-239
4050 ON Z GOSUB 4100,4200,4300,4400
4100 REM UP
```

```
4110 XDIR = 0 : YDIR = -1
4120 GOTO 4800
4200 REM DOWN
4210 XDIR = 0 : YDIR = 1
4220 GOTO 4800
4300 REM LEFT
4310 YDIR = 0 : XDIR = -1
4320 GOTO 4800
4400 REM RIGHT
4410 YDIR = 0 : XDIR = 1
4800 X = MANXPOS+XDIR : Y=MANYPOS+YDIR
4805 LOCATE X,Y : CALL RDCHAR : CH$=CHR$(
          PEEK(CHREAD)) : IF CH$ = RIVER$
          THEN SEARCHING = FALSE : GOTO
          4900
4810 IF CH$ = HEFFA$ THEN SEARCHING =
          FALSE : CAUGHT = TRUE : CAPTURED
          = CAPTURED + 1 : LOCATE MANXPOS,
          MANYPOS : PRINT " " ;: GOTO 4930
4820 IF CH$ <> " " THEN XDIR = 0-XDIR : YDIR
          = 0 - YDIR : GOTO 4930
4900 LOCATE MANXPOS,MANYPOS : PRINT " ";:
          LOCATE X,Y : PEN 2 : PRINT MAN$;:
          MANXPOS = X : MANYPOS = Y
4930 EI
4940 GOTO 4005
4999 RETURN
```

Subroutine 4000 is the inner loop of the main task. The program stays in here until the man is caught or escapes, but of course is repeatedly interrupted by the subtask which moves the heffa.

If a key is pressed it is analysed; Only the arrow keys are valid so all others are rejected. It would therefore be very easy to adapt this game to use a joystick.

Lines 4100 to 4410 set the direction. Once set it remains until altered. You could experiment by making the man able to turn at 45 degrees instead of 90. It might improve his agility under pressure.

Line 4800 sets the proposed position to x,y. The current position is maintained at manxpos,manypos. Then an assembler language subroutine is called which was loaded during the initialisation routine. This reads the character at the proposed location to avoid collision. If the character is river$ or heffa$ then the quest is over - at least for this player. If the character is not a space it must be an edge or a tree, and the direction is reversed.

Note the DI and EI instructions which are vital unless you want occasional red men appearing in odd locations. (This can happen if the interrupt to move the heffa occurs after you have located at manxpos, manypos but before you have had a chance to print the man in yellow; by the time you are returned, the location has been altered and the pen is red!)

Subroutine 5000 clears up the mess.

```
5000 REM CLEAN UP
5005 DI
5007 PEN 1
5010 LOCATE RIVERXPOS,RIVERYPOS : PRINT
        RIVER$;
5015 MANXPOS = RIGHTLINE - 1 : MANYPOS =
        TOPLINE + 1
5017 LOCATE 1,TOPLINE-1 : PRINT "Escaped";
        MAN-CAPTURED;". Caught";
        CAPTURED
5020 EI
5030 IF CAUGHT THEN GOSUB 10000 ELSE GOSUB
        9000
5999 RETURN
```

The river is replaced, just to make sure. A new position is prepared for the next man, if any. Then a score is printed on the top line. Finally the correct song is chosen.

```
9000 REM VICTORY SONG!
9010 RESTORE 9800
9020 READ T
9030 IF T=-1 THEN 9900
9040 SOUND 1,T,20,5
9050 GOTO 9020
9800 DATA 80,60,47,80,60,47,80,60,47
9810 DATA -1
9900 WHILE SQ(1)>127 : WEND
9999 RETURN

10000 REM FUNERAL MUSIC!
10010 RESTORE 10800
10020 READ T,L
10030 IF T=-1 THEN 10900
10040 SOUND 1,T,L*40,5
10045 SOUND 1,0,5,0
10050 GOTO 10020
10800 DATA 478,2,478,1.5,478,.5,478,2,
        402,1.5,426,.5,426,1.5,478,.5,478,
        1.5,536,.5,478,2
10810 DATA -1,-1
10900 WHILE SQ(1)>127:WEND
10999 RETURN
```

```
6000 REM FINAL MESSAGE
6010 J = REMAIN(0)
6999 RETURN
```

This subroutine halts the heffalump's activities by cancelling the subtask. There is no current final message, over to you...

CHAPTER 13
GAME NUMBER 2 – GHOSTS IN A BOTTLE!

I cannot claim any credit for the design of this game; it was thought up by my 10 year-old son. Where HE got it from I do not like to enquire.

The source and commentary is given in this chapter so that you can follow the logic of the game and (who knows?) improve on it. The game is also available on a tape from Kuma Computers, together with the game of the preceding chapter, and the 3-part tune from chapter 9.

Program Plot.

The ´plot´ is simple. A hero (you, of course) starts life at the top right of a sparsely-populated screen. The only objects to be seen are a bottle (lower left-hand side) and a tomb (lower right).

Suddenly a bilious-looking ghost emerges from the bottle with a moan, followed at intervals by several more. They seem set on making your acquaintance and direct their way unnervingly towards you in hideous convoy.

Your only chance of survival is:

- to put the stopper on the bottle before too many escape for you to deal with

- To lead them into the tomb from the top, acting as a decoy, to escape from the bottom, jamming the exit with a stone as you leave, then to sneak round to the top to block that entrance also with another stone.

If you fail you are doomed to flee through space for ever pursued by these ghouls. Alternatively, and equally unpleasantly, you lock yourself in the tomb by mistake, leaving the ghosts clamouring on the outside.

Please do not write to me pointing out that self-respecting ghosts should not be deterred by solid stone. I did not invent the idea, only struggled with the programming...

Instructions for Use.

When you have loaded the program in the normal way, use the 4 arrow keys to move round the screen and the COPY key to insert the block in the exit and entrance to the tomb. This latter key must be pressed when you are exactly in the exit or entrance, it is ineffectual if used elsewhere.

Note that it is vital that you lead the ghosts in from the top, and therefore block the bottom first! Try it the other way round and you will see what I mean...

You put the stopper on the bottle (your first task if you have any sense) merely by positioning yourself on the mouth of the bottle. No key pressing is necessary.

Program Commentary

The program uses several techniques similar to those in the last game, but there are 2 independant subtasks and a much longer assembler subroutine. This is there for speed. I found that having to cope with the movements of 10 spectres put something of a damper on the game in BASIC alone. The judicious use of assembler makes for a satisfactory response.

The source for the assembler subroutine is discussed in full in chapter 16.

Lines 7-50 deal with the set-up. GOSUBS are made to 1000 (initialisation), 2000 (drawing of the bottle and tomb), 3000 (the hero is positioned), and a separate task (4000) is actioned every 2500 ms which will launch the ghosts regularly until a maximum of 10, or until you manage to stopper the bottle.

```
7 REM COPYRIGHT (C) 1984 JOHN BRAGA
10 REM ghosts in a bottle!
20 GOSUB 1000
30 GOSUB 2000
40 GOSUB 3000
50 EVERY 50 GOSUB 4000
```

The main game is played during a single call to subroutine 5000 in statement 60. All this does is move the hero, since the launching of the ghosts and their movement is taken care of by separate subtasks, a simple technique (in this version of BASIC!) that relieves you of timing problems.

```
60 GOSUB 5000
```

The final cleanup routines are in lines 70-300. Any further launching or movements of the spectres is halted and their positions are calculated to see if they fall inside or outside the tomb. If all are inside you get a thankful message of gladness, but if any have remained outside you see a message of doom. If you have been trodden on by a ghost you see a gloating 'Gotcha!' message.

```
70 X=REMAIN(0):Y=REMAIN(1)
80 IF NOT FREE THEN 200
100 FOR J=1 TO NUMSPECTRE
110 GX=PEEK(GXSTORE+2*J-2):GY=PEEK(GXSTO
RE+2*J-1)
120 IF GX<31 OR GX>33 OR GY<16 OR GY>20
THEN DOOMED=TRUE
130 NEXT J
140 IF DOOMED THEN LOCATE 15,1:PRINT "DO
OMED TO FLEE FOR EVER!!!":GOTO 210
150 LOCATE 20,1 : PRINT "YOU EARN HUMANI
TY'S BLESSING!"
160 GOTO 210
200 LOCATE 20,1 : PRINT "GOTCHA!!!!!";
210 FOR J=1 TO 4000 : NEXT J
220 CLS
240 SPEED KEY 10,10
250 PEN 1
300 END
```

The initialisation subroutine at 1000 first reserves memory for the assembler language subroutine, then later loads it in at 7000h. The loading is done by reading data statements and poking in preference to loading a binary file from tape. This is merely a slight added convenience.

As an aside, rather than type all the data statements by hand, a time-consuming and error-prone task, I used ZEN to display the contents of the area containing the object code at the top of the screen, exited to BASIC, and used the COPY key to create data statements one by one, each containing 8 bytes. Once the data was 'captured' in this way I edited the lines at my leisure to add commas and ampersands instead of spaces between the hex bytes.

```
1000 REM initialise
1005 TXTRD = &7000 : RDCHAR = &7007 : MV
GHOST = &7008 : DONEFLAG = &70FE : NUMGH
OSTS = &70FF : MXSTORE = &7100 : GXSTORE
 = &7102
1010 CLS
1015 MEMORY &6FFF
1017 SPEED KEY 8,3
1020 bottlecolour=1:mancolour=2:ghostcol
our=3
1030 man$=CHR$(249):ghost$=CHR$(225):edg
e$=CHR$(143)
1040 true=-1 : false=0
1050 stoppered = false
1060 INK 3,6,5
1070 free = true : PLAYING = TRUE : TOMB
STONES=0:DOOMED=FALSE
1080 STOPPER$=CHR$(210)
1085 ENT -1,10,10,3
1090 POKE NUMGHOSTS,0 : POKE DONEFLAG,FA
LSE
1100 GOSUB 7000:CLS
1500 RESTORE 1900
1510 FOR J=0 TO 2000
1520 READ X : IF X=-1 THEN 1590
1530 POKE &7000+J,X
1580 NEXT J
1590 REM
1900 DATA &CD,&60,&BB,&32,&07,&70,&C9,0
1910 DATA &AF,&32,&FE,&70,&3A,&FF,&70,&3
2
1911 DATA &FD,&70,&2A,&00,&71,&E5,&C1,&2
1
1912 DATA &2,&71,&C5,&56,&23,&5E,&23,&E5

1913 DATA &D5,&79,&BA,&28,&06,&38,&3,&14
1914 DATA &18,1,&15,&78,&BB,&28,6,&38
1915 DATA 3,&1C,&18,1,&1D,&EB,&E5,&CD
1916 DATA &75,&BB,&CD,&60,&BB,&D1,&E1,&D
5
1917 DATA &FE,&F9,&20,&A,&3E,1,&32,&FE
1918 DATA &70,&32,&FD,&70,&18,4,&FE,&20
1919 DATA &20,&1C,&CD,&75,&BB,&3E,&20,&C
3
1920 DATA &5D,&BB,&E1,&E5,&CD,&70,&BB,&7
5
```

```
1921 DATA &E1,&CD,&5D,&BB,&D1,&E1,&2B,&2
B
1922 DATA &72,&23,&73,&23,&E5,&D5,&3A,&F
D
1923 DATA &70,&3D,&32,&FD,&70,&D1,&E1,&C
1
1924 DATA &20,&A0,&C9,-1
1999 RETURN
```

Instructions are displayed by 7000.

```
7000 REM INSTRUCTIONS
7010 LOCATE 10,1:PRINT "Ghosts in a Bott
le!"
7020 PRINT
7030 PRINT "In this game you are pursued
 by "
7040 PRINT "miserable ghosts which are e
scaping "
7050 PRINT "from a bottle.  First you mu
st rush to"
7060 PRINT "the bottle (use the arrow ke
ys), and "
7070 PRINT "put on the stopper by touchi
ng the "
7080 PRINT "mouth.  This stops further g
hosts"
7090 PRINT "appearing.  Then you can try
 to lead "
7100 PRINT "the wailing band into the TO
MB and "
7110 PRINT "shut them in for ever!  Be c
areful to"
7120 PRINT "close the LOWER door first (
and when "
7130 PRINT "the ghosts are inside).  If
you succeed"
7140 PRINT "creep round to the TOP door
and plug"
7150 PRINT "that one behind them.
7160 PRINT
7170 PRINT "You close a door by pressing
 the COPY "
7180 PRINT "key when exactly in the entr
ance."
7190 PRINT
7200 INPUT "Press ENTER when ready to st
art...",z$
7999 RETURN
```

The subroutine to draw the bottle and tomb needs little commentary, except to say that if you decide to locate the objects in a different place you will have to make changes elsewhere since the routine which checks whether the ghouls are all inside the tomb will have to be altered also, and the check made on whether the hero has stoppered the bottle will also have to be adjusted.

```
2000 REM draw bottle & tomb
2010 LOCATE 1,21
2015 PEN bottlecolour
2020 PRINT CHR$(209);edge$;CHR$(211)
2030 PRINT CHR$(214);edge$;CHR$(215)
2040 PRINT STRING$(3,edge$)
2050 PRINT STRING$(3,edge$)
2100 LOCATE 30,15
2110 PRINT edge$;edge$;" ";edge$;edge$
2120 FOR j=1 TO 5
2130 LOCATE 30,j+15:PRINT edge$;"   ";ed
ge$
2140 NEXT j
2150 LOCATE 30,21:PRINT STRING$(3,edge$)
;" ";edge$
2999 RETURN
```

The start-man subroutine at 3000 records the position at mx,my and also at mxstore and mystore for use by the assembler language subroutine. It then disables interrupts so that it can draw the man on the screen in peace and quiet (rather superfluous as written since the subtasks have not yet been initiated. Call it a precaution against future change!)

```
3000 REM start man
3010 mx=40 : my=1
3020 nmx=mx:nmy=my
3030 POKE MXSTORE,mx:POKE MXSTORE+1,my
3040 DI:LOCATE mx,my:PEN mancolour:PRINT
 man$;:EI
3999 RETURN
```

At 4000, a subtask which is called every 2500 ms, a check is made to see if the bottle is stoppered or whether 10 spectres have already been launched on a deserving world. If so the subtask obligingly cancels future executions of itself via a REMAIN function and exits with no action.

If there is work to do however, a ghost is launched. Note the simple technique to draw a line then to make it disappear by redrawing it in the background colour.

```
4000 REM start ghost
4005 NUMSPECTRE=PEEK(NUMGHOSTS)
4010 IF stoppered OR (NUMSPECTRE=10) THE
N j=REMAIN(0) : GOTO 4999
4020 DI:PEN ghostcolour : PLOT 20,82,gho
stcolour:DRAW 40,360,ghostcolour:LOCATE
5,1:PRINT ghost$;:PLOT 20,82,0:DRAW 40,3
60,0:EI
4030 SOUND 1,0,20,7
4040 POKE NUMGHOSTS,NUMSPECTRE+1:POKE GX
STORE+2*NUMSPECTRE,5:POKE GXSTORE+1+2*NU
MSPECTRE,1
4050 EVERY 30,1 GOSUB 6000
4999 RETURN
```

The game is played by iterating round subroutine 5000 while the 2 subtasks act asynchronously and independantly. Only the 4 arrow keys and the COPY key are looked for; others are discarded. The subroutine is terminated when 2 tombstones have been put in place, or when the free flag is set to false (externally).

```
5000 REM moveman
5010 IF NOT (FREE AND PLAYING) THEN 5999

5015 z$=INKEY$:IF z$="" THEN 5010
5020 z=ASC(Z$):IF z=224 THEN z=244
5030 IF z<240 OR z>244 THEN 5010 ELSE Z=
Z-239
5040 nmx=mx:nmy=my
5050 ON z GOTO 5100,5200,5300,5400,5500
5100 REM up
5110 IF my>1 THEN nmy=my-1
5120 GOTO 5600
5200 REM down
5210 IF my<25 THEN nmy=my+1
5220 GOTO 5600
5300 REM left
5310 IF mx>1 THEN nmx=mx-1
5320 GOTO 5600
5400 REM right
5410 IF mx<40 THEN nmx=mx+1
5420 GOTO 5600
```

```
5500 REM COPY KEY
5510 IF MX=32 AND MY=15 THEN DI:LOCATE M
X,MY:PEN BOTTLECOLOUR : PRINT EDGE$;:NMX
=32:NMY=14:GOTO 5530
5520 IF MX=33 AND MY=21 THEN DI:LOCATE M
X,MY:PEN BOTTLECOLOUR : PRINT EDGE$;:NMX
=33:NMY=22:GOTO 5530
5525 GOTO 5540
5530 TOMBSTONES=TOMBSTONES+1:IF TOMBSTON
ES=2 THEN PLAYING=FALSE
5540 GOTO 5705
5600 REM test
5610 DI:LOCATE NMX,NMY : CALL TXTRD : IF
 PEEK(&DCHAR) <> 32 THEN EI:GOTO 5720
5700 DI:LOCATE mx,my:PRINT " ";
5705 LOCATE nmx,nmy:PEN mancolour:PRINT
man$;: mx=nmx: my=nmy :EI
5710 POKE MXSTORE,mx:POKE MXSTORE+1,my
5712 IF MX=2 AND MY=20 THEN DI:STOPPERED
=TRUE:LOCATE 1,20:PEN BOTTLECOLOUR:PRINT
 STRING$(3,STOPPER$);:NMX=4:NMY=20:GOTO
5705
5720 GOTO 5610
5999 RETURN
```

The subroutine at 6000 is an independant subtask called by the launch ghost routine in 4000. It operates at the highest priority and therefore the DI/EI instructions are strictly superfluous.

Its main function is to call the assembler language subroutine which moves the ghosts (see chapter 16).

It also issues the ghostly moan, using a tone envelope issued during the initialisation routine.

```
6000 REM move ghosts
6010 DI
6020 PEN GHOSTCOLOUR
6030 CALL MVGHOST
6040 IF PEEK(DONEFLAG)=1 THEN FREE = FAL
SE : X=REMAIN(0): Y=REMAIN(1)
6050 SOUND 1,350,40,4,0,1
6990 EI
6999 RETURN
```

Happy Spectre-bashing!

PART IV

ASSEMBLY LANGUAGE PROGRAMMING

This part contains details of programming the Amstrad in Assembly Language, showing how routines can be written to interface with BASIC programs and Enhance them.

CHAPTER 14
THE ZEN ENVIRONMENT

Useful as BASIC is, there comes a time when we want something extra, and turn to Assembler. There are several possible reasons for this:

- a need for extra speed. Assembler programs can operate many times faster than BASIC.

- a need to conserve memory. Assembler programs are several times more concise than BASIC equivalents.

- A need to access system facilities which are not available in BASIC. All machine features are available to the Assembler programmer, it is only knowledge which holds him back!

Of course the Amstrad, as supplied, is a BASIC machine in the sense that BASIC is supplied in ROM, and the machine is set to power up into BASIC on switch-on. But it does not have to stay there!

ZEN.

Kuma Computers supply an Assembler called simply ZEN for the Amstrad which enables the the user:

- to write complete Assembler programs that can be saved on tape and reloaded as desired, to run independantly of BASIC

- to write programs that are designed to be called as subroutines from BASIC, and will therefore live alongside BASIC programs.

In addition ZEN contains a disassembler, so that it can be used to fathom the mysteries of the Amstrad operating system, if desired.

Versions of ZEN have been supplied for other systems, for example Sharp, Newbrain and Epson, so that the product is a well-proven one and can be recommended to any Amstrad owner who is familiar with Assembler, or who wishes to learn. The handbook supplied contains a full listing of the ZEN source code, so that the expert can make changes to the product to suit himself, and the learner can study a large well-written piece of Z80 Assembler code.

Loading ZEN.

ZEN is supplied on tape. It resides at 4000h which is slap in the middle of what is normally BASIC´s area, so a MEMORY statement must first be entered to lower the top of BASIC´s partition to 3FFFh or 16383. ZEN can then loaded by a simple LOAD "" command, after which it can be entered by:

```
CALL 16384
```

and displays its sign-on prompt

```
ZEN>
```

The loading process, at the slower speed, takes about 1 minute.

Memory Layout.

At this point the user will normally wish to enter source code, either typed in or loaded from tape, to be modified or extended and assembled. The default location for the source code is 6000h, but this can easily be altered.

ZEN is an in-memory assembler, so if it is desired, as it normally is, to run the program produced, space must be allocated for the object code to be stored once ZEN assembles the source. This would normally be above the source code, at - say - 8000h upwards.

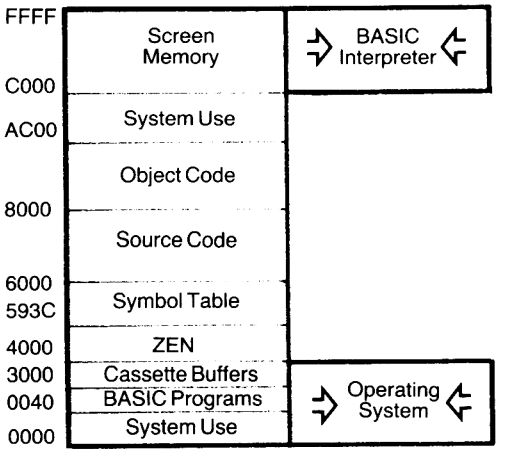
A work area is needed by ZEN to put the symbol table containing all the variable names in the source; the space for this starts immediately after ZEN itself, at 593Ch and extends to the beginning of the source code at 6000h. If this proves insufficient the start of source code can be moved.

The only other area of memory that needs to be mentioned is the 4k needed for reading and writing to cassette. When you issue the instruction to load ZEN, BASIC performs its normal trick of lowering HIMEM by 4096 bytes, in this case to 2FFFh. So ZEN expects them to be at the same place, a read buffer at 3000h and a write buffer at 3800h. If you decide to put them elsewhere, ZEN will be just as happy provided it is kept informed by your modifying 2 pointers.

The pointer allocation is under user control, so it is perfectly possible to have more than one source program in memory, each terminated by an END statement, and to assemble one or the other alternately.

To summarise, the normal memory map while ZEN is loaded is as follows. Compare this with the map shown in chapter 1.

Amstrad Memory Map with ZEN Loaded.



It must be stressed that this is only the suggested layout, for normal use. Larger BASIC programs can be accomodated by moving the cassette buffers above the ZEN area, and the boundaries of the Assembler Source and Object Areas are entirely at the user's discretion.

In fact the size of the memory allocated for BASIC is not usually a problem. There are many other systems on the market who would be glad to have as much! If you do find that your BASIC program is too large to co-exist with ZEN, remember that once you have tested your assembler subroutine using a skeleton BASIC program, the object program can be loaded without either ZEN or the source code being present and will be called directly from your BASIC program, so HIMEM will very probably be at 7FFFh or higher.

ZEN Facilities.

Having loaded ZEN into memory as shown above, what commands are available? All are a single letter and some have a parameter.

A	Assemble	O	Out
B	Bye (return to BASIC)	P	Print
C	Copy	Q	Query
D	Down	R	Read
E	Enter	S	Sort
F	Fill	T	Target
G	Goto	U	Up
H	Howbig	W	Write
I	In	X	Xamine
K	Kill	Z	Zap
L	Locate	c	Catalogue
M	Modify	d	Dissassemble
N	New	u	Unscramble

You can see that ZEN provides the basics of a complete development environment. All assembling is done direct to memory, so the object code, if requested, is available at once.

Source programs are entered using the E command, assembled via the A command, and may be deleted with the Kill (K) command. They can be executed, when the assembly is 'clean' with a Goto command (G). A breakpoint can be set so that execution can proceed in stages with registers and storage being displayed at each halt.

ZEN does not provide a full screen editor for entering source, but commands are provided to display a line or lines (P), locate a specific character string (L), goto a specific line (T), move up or down a relative amount (U, D), or delete one or more lines (Z).

Data can be created by Fill (F), Copy (C) or Modify (M) and examined by Query (Q) or X (which displays the registers).

You can save source or object files by the W command, and read them back in via the R command. The tape can be scanned by the catalogue (c) command.

The disassembler is invoked with the d command, and there is also a useful 'unscrambler' which disassembles 8 commands at a time.

Output from the Assemble command (A), the Sort and display symbol table (S) or the disassembler (d) can be sent either to the screen or to an external device, ie. a printer.

The user can use the B command to move from ZEN to BASIC, and CALL 16384 to move back again. Programs in either area will be just as you left them.

ZEN Summary.

ZEN is a useful product, providing a comprehensive and well documented environment for the Assembler programmer. A knowledge of Assembler language is an enormous asset, and opens a large number of doors, so I do urge those who are hesitating to plunge in. The water´s lovely!

CHAPTER 15
INTERFACING TO BASIC

Most users, approaching Assembler for the first time will be doing so in order to write subroutines performed from BASIC, rather than writing stand-alone Assembler programs. In most cases parameters will need to be passed from one language to the other. This chapter covers the techniques needed to move happily from BASIC to Assembler and back again.

The examples shown apply particularly to the ZEN Assembler environment, but the differences, if another Assembler is used, should be minimal, since what we are studying are the conventions used by Amstrad BASIC.

The Subroutine Development Sequence.

BASIC programmers are used to finding and fixing faults in their programs; few of us write a significant program without bugs creeping in. When we run the program and a fault such as an endless loop occurs we break in, stop the program, fix the problem and restart. If a divide by zero error occurs, we insert an extra test to make sure the condition cannot reoccur.

In Assembler a program fault can wipe out an entire evening's work in microseconds!

It is for this reason that users are urged to save copies of their source programs frequently. The object code can be replaced in seconds if the source is intact. To replace the source may take hours.

The following is a suggested sequence for developing Assembler subroutines under ZEN. Following it will save you growing old before your time...

1. Type a MEMORY 16383 command and load the ZEN Assembler above BASIC.

2. Enter ZEN and type in the source of the program.

3. Create a subsidiary program, which will eventually be thrown away, alongside the main one, to set up the registers and data as they will appear when the subroutine is called from BASIC.

4. Assemble and get rid of assembly-time errors such as spelling mistakes, undeclared symbols, etc.

5. Save the source code (twice) to tape. LABEL THE TAPE!

6. Start the subsidiary program and stop it with a breakpoint at the start of the main subroutine, verifying it sets up the environment as expected.

7. Restart the subsidiary program, moving the breakpoint further into the program. Verify the registers and data areas are being correctly manipulated.

8. If (or when!) faults occur, note the corrections by whatever means is suitable, whether it be a pencil note on the back of an envelope, or a new assembly directed to the printer and marked with the date and time.

9. Change the version number (which should be in a comment statement at the start of the program) and resave to tape whenever you have changed more than a few statements. Do NOT save on top of the last source copy saved, use another tape so that you circulate at least 2. Include the time in the title being saved, ie.
```
SAVE "ABCIII 4.30 23/8",B,...
```

10. When the program appears to work completely correctly, exit to BASIC with the BYE command, write a BASIC program to call the subroutine and repeat the test using all combinations of data.

11. Return to ZEN and delete the subsidiary program which sets up data. Then reassemble.

12. Save the object code (for the first time) onto tape.

13. Reset the system completely. (Jumping to location 0 will do this very well!)

14. Type in the BASIC program to set memory and load the assembler object module. Then call the module to verify correct operation.

If the above sounds to you harder work than programming BASIC, you're right! But the rewards are significant too, as you will find when your first program (eventually) works as expected...

Passing Information the Simple Way.

Let us consider a simple subroutine as an example. The Amstrad Operating System is full of useful routines that we can call directly from BASIC or use from Assembler subroutines. A list of the most useful is given in the next chapter, and a full list is found in the Amstrad Technical Manual.

If we are projecting a missile such as a ball around a screen we may well wish to know if we are about to bump into anything.

This appears to be a problem! In BASIC there is no command that tells us what character is at a specific screen location; we could use TEST or TESTR to test for the presence of a specific colour, but that is not sufficient for many purposes. We could PEEK the screen display, but the layout is very complex, and would need considerable deciphering. However there is an operating system routine called TXT RD CHAR which is just what we want.

To use this routine we must position ourselves at the square in question using the LOCATE command, then call the routine which will return the character value in register A - 32 for a space, 65 for a capital 'A', 250 for a man-shaped figure, etc. If there is no valid character at the location (which can happen if one character is overwriting another for example) the Carry flag is cleared and zero is returned in A.

So we need a short assembler subroutine which we can call from BASIC that will call the TXT RD CHAR routine and then place the contents of register A in a place where BASIC can find it.

This passing of information can be handled in two ways:

- by passing the name of a BASIC variable into which the character is inserted,

- by writing the assembler subroutine so that it inserts the character in a fixed place in memory such that BASIC can look at it via a PEEK instruction.

This second method is not very elegant but is remarkably easy, and is the method we show first.

Normally HIMEM is set to AB7Fh (or 43903 if you prefer). Our routine is likely to be short so we will plan to run it at AB60h and we will make it store the character at AB7Fh.

The TXT RD CHAR routine is situated at BB60h. It does not require any input parameters, but exits in one of 2 states:

- Carry set and a character in A.
- Carry clear and A set to zero (error).

For the purposes of our example we will ignore the error case and assume that a correct character is always returned. The routine as entered into ZEN is:

```
ORG  0AB60H          ;Starting position.
LOAD $               ;Generate object code.
CALL 0BB60H          ;Call TXT RD CHAR
LD   (0AB7FH),A      ;Store result
RET                  ;Return to BASIC
END
```

Assembling it produces a measly total of 7 bytes (I said Assembler was concise!) at 0AB60H.

If we follow our own advice, given earlier, we should create an Assembler test harness to test our routine, but every rule is meant to be broken on occasion, and we will test this from BASIC, as it is easier to move around the screen using LOCATE.

Return to BASIC and enter the following program:

```
10 CALL &AB60
20 PRINT PEEK(&AB7F)
```

Clear the screen and run. You should see 32 printed out which is the value of a space. Since RUN, followed by ENTER, was typed on line 2, the system was located at column 1 of row 3, which was a space.

All well so far! Now, without clearing the screen, enter:

```
5 LOCATE 2,3
```

and RUN again. This time you should see the value 51 printed out since the character at column 2 of row 3 where we positioned ourselves was a '3' and as you can see from the character chart in Appendix III of the Amstrad handbook, the character 3 is ASCII 51.

Please feel free to devise other tests, all of which should prove that as long as you locate at a correct position the subroutine works as we would like. Success!

Using Parameters.

Now we have our first Assembler routine, we will get ambitious, and return the result in a variable rather than in a fixed location. What we want to do is to test for a space by writing:

```
CALL &AB60,A$
IF A$ <> " " THEN...
```

But this will not work, no matter how we alter our Assembler subroutine! The reason is that Amstrad BASIC parameters are always passed **by value.** That means that, in the above example, BASIC will create a copy of A$ in storage somewhere and pass it to the subroutine. Because it is a copy, you can alter it until you are blue in the face, it makes no difference to the original contents of A$ which are left snugly undisturbed.

Happily there is a simple solution. Amstrad BASIC supplies a little-documented operator called '@' which says 'instead of passing the value of the variable as a copy, pass the address of the variable instead.' So we can use the address to access the original variable and change it. Our BASIC program must set up a constant with a length of 1 (since the Assembler cannot alter the length of a BASIC variable) before calling the routine. The test program will be:

```
10 A$ = "Z"
20 CALL &AB60,@A$
30 PRINT A$
```

and we will expect to see A$ changed from 'Z' to some other value.

Parameter Types.

Before we can show the changes to the Assembler routine we need to differentiate between different types of parameter, and to show how they are stored by BASIC.

There are 3 types of variable: string, integer and real.

Character strings, as you well know, are usually represented by names which end with '$' (though not always - see the DEFSTR command). They may be up to 255 bytes long. A string variable is in fact stored in 2 parts, the **String Descriptor** and the **String Body.** The descriptor contains the length (as 1 byte, hence the maximum of 255) followed by the address of the body. The body contains the variable data.

Integers have a range of -32768 to +32767. This will tell you, if you know Z80 Assembler language, that they are stored as 2 bytes, and you would be right.

Reals are used to hold a larger range than integers can manage and to hold fractions. They have a fairly complex internal representation using a mantissa and an exponent.

String Parameters.

Knowing the above, we would expect A$ in this example to be stored (after line 10) as:

```
Descriptor      -    DB   1     ;length
                -    DW   body  ;address of body.

Body            -    DB   'Z'
```

Whether the parameter is a value or an address it is pointed to by register IX on entry to the subroutine. So our Assembler routine might be changed to:

```
        ORG  0AB60H
        LOAD $
        CALL 0BB60H          ;CALL TXT RD CHAR
        LD   L,(IX+0)        ;ADDRESS OF A$ DESCRIPTOR
        LD   H,(IX+1)
        INC  HL              ;IGNORE LENGTH
        LD   E,(HL)          ;PICK UP ADDRESS OF BODY
        INC  HL
        LD   D,(HL)          ;INTO DE
        LD   (DE),A          ;STORE VALUE OF A IN BODY
        RET
```

and the BASIC routine would be changed as above to include the @ operator.

You should be able to position yourself in various parts of the screen to test that the routine works satisfactorily.

Integer Parameters.

The format of an integer parameter is simpler. The value is held as a 16-bit signed binary number in the range -32768 to 32767, (using two's-complement notation in the case of negative values). Again, if we want to alter the value of the variable passed during the subroutine we need to pass its address using the @ operator. If we merely want to read the value without altering it we need not bother with the @. So our routine could use an integer instead of a string variable:

```
10 Z%=0
20 CALL &AB60,@Z%
30 PRINT Z%
```

and the Assembler routine would be altered to:

```
ORG  0AB60H
LOAD $
CALL 0BB60H
LD   L,(IX+0)
LD   H,(IX+1)
LD   (HL),A
RET
```

Test as before.

Real Numbers as Parameters.

Having conquered strings and integers we are left with reals. Unfortunately their format is complex! A 4-byte exponent is followed by a 1-byte mantissa which is 'biased by 128'. This format is explained in more detail in the Amstrad Concise BASIC Specific Specification (SOFT 157) which covers the Amstrad BASIC in greater detail than the standard handbook. If you want to study the format of various real numbers I suggest you write a routine that moves the 5 bytes into specific areas of memory, then prints out their value on return to BASIC. The following will suffice:

```
10 Z = 32
20 CALL &AB60,@Z
30 FOR J = &AB7B TO &AB7F
40 PRINT HEX$(PEEK(J));" ";
50 NEXT J
60 PRINT
```

together with an assembler subroutine of:

```
ORG  0AB60H
LOAD $
CALL 0BB60H
LD   L,(IX+0)
LD   H,(IX+1)
LD   DE,0AB7BH
LD   BC,5
LDIR
RET
```

and as line 10 is altered, this will produce results like:

```
0 0 0 0 0   Z = 0

0 0 0 0 81  Z = 1

0 0 0 0 82  Z = 2

0 0 0 80 81 Z = -1, etc.
```

In the above example we were passing the address of the real number, not the value itself, using the @ operator. It should be noted that if you attempt to pass the value itself, BASIC converts the real number to unsigned integer format, and therefore you are passed a 2-byte number and not the complicated 5-byte one just shown. Of course the conversion from real to integer will usually cause loss of data, since fractions are rounded to the nearest whole number.

Unless you are writing your own Assembler maths routines to handle these complex numbers, I think it is unlikely that you will wish to do much parameter passing of reals!

More than One Parameter.

So far we have talked only of passing one parameter. But we can pass several, and they can be a mixture of **value parameters** or **address parameters**, as you wish. Whichever type you choose the parameter will always be a 2-byte number.

When you enter the subroutine, BASIC has set register A to contain the number of parameters, from 0 upwards. As discussed already, register IX is set up to point to the parameters themselves, but note that it points to the last one, not the first. To illustrate this, consider the fragment:

```
5 A%=5 : ABC$="HELLO" : Z = 3.5
10 CALL &8000,@A%,@ABC$,Z,-1
```

There are 4 parameters, 2 addresses and 2 values. So A will be 4 on entry and register IX will be set up as follows:

```
Register IX -->  DW   -1
                 DW   value of Z
                 DW   address of ABC$ descriptor
                 DW   address of A%
```

which is the opposite way round to what you might at first expect.

When I tested the above, I recorded the following values; if you try the same you may get small differences if your version of BASIC or ZEN differs from mine, or if you type with extra spaces, but the results ought to be the same.

Register IX contained BFF6h. At BFF6h I found:

```
BFF6:     FF FF
BFF8:     04 00 CA 01 C2 01 ...
```

and you can see that the parameters are all stored as 2 byte numbers, with IX indeed pointing to -1 (FFFFh), followed by 4 (Z converted to the nearest integer), followed by 2 addresses 01CAh and 01C2h. At 01CAh we find the string descriptor containing a length 05h, followed by an address, 0185h, which points correctly to the string body of ABC$:

```
0185      48 45 4C 4C 4F      HELLO
```

Finally at 01C2h we find 05 00 which is the value of A%.

In this example we could alter the values of A% and ABC$, but not those of Z or (obviously) of -1.

All unravelled, I hope?

CHAPTER 16
OPERATING SYSTEM ROUTINES

Now that you can interface assembler routines to BASIC you may be anxious to try out a few ideas.

Jump Blocks.

The Amstrad Operating System has been designed to be accessible to users. It contains a large number of routines which it needs primarily for its own purposes, but which are documented and available for user programs as well.

One problem always faced by developers is that routines may alter position in different releases of the operating system, due to corrections or improvements being made. If the user writes his program assuming the TXT RD CHAR routine is at BB60h and finds it is at BB68h in some later version of the Amstrad he will not be pleased, nor will other users of his program.

In the Amstrad this is solved by 'Jump Blocks'. As the name implies jump blocks are merely sequences of JUMP instructions to various operating system routines. These blocks are contained in the ROMs and moved to RAM at system initialisation. They are always moved to the same place, so that the user can rely on their always being a JP TXT.RD.CHAR instruction at location BB60h, while the developer is free to move TXT RD CHAR anywhere he likes, as long as he alters the Jump Block in step. Simple!

The moral is: rely on the jump blocks, and do not try to bypass them. Remember that in 2 years' time, an Amstrad expanded with disk and extra ROMs may look very different internally, but the existing jump blocks are likely to be maintained.

There are 4 jump blocks in all. The Amstrad Technical Manual gives full details of them and pays particular attention to the entry and exit conditions of each routine. If you plan to program extensively in Assembler, the Technical Manual is essential reading. Here we do not attempt to replace it, only to give an outline of the more important routines.

Advanced users should note that they can alter a jump instruction in a jumpblock to point to their own routine in RAM rather than the system-provided routine in ROM. Provided that the new routine observes the documented entry and exit conditions, other programs using the routine should continue to work as expected. But you have to know what you are doing...

The Main Jump Block.

The main jump block, the one users will normally wish to access, is situated at BB00h - BD37h. It contains 190 3-byte instructions, but most of them are not actually jumps! You will remember that the Amstrad ROMs overlay the RAM at top and bottom of memory. Many of the entries in the main jump block actually consist of an RST 8 instruction followed by the address of a routine, with flags set in the high-order bits. This causes the required jump to low memory, but also sets the ROMs as enabled or disabled according to the flags. It is in effect an extra Z80 instruction 'Jump and set ROM state'.

Some of the most useful entries in this jump block are:

BB00 Keyboard Manager Initialise. This is used to restore the state of the keyboard as it would be after a system reset. Any expansions are reset to their defaults, the buffer is cleared, repeat speeds are returned to their default etc.

BB06 Keyboard Wait Char. Wait for the next character from the keyboard. This is normally the next character typed, but may be, for example, the next character of an expansion string, or a character from a key translation table. A contains the character.

BB09 Keyboard Read Char. Unlike the above routine, this one returns immediately, whether there is a character ready or not. If a character is found it is returned in A with carry on. If not carry is clear.

BB24 Get Joystick. Investigate the status of the joysticks. A byte of information is returned for each joystick (whether or not one is actually attached). Register H shows joystick 0 and register L joystick 1. The bits are:

0 - up.
1 - down.
2 - left.
3 - right.
4 - fire 2.
5 - fire 1.

BB4E Txt Initialise. A full initialisation of the text VDU takes place, as for system reset. This means that all streams are set to their default values, the cursor is moved to the top left and the screen is cleared.

BB5A Txt Output. Output a character to the currently selected stream. The character, contained in A, may be a control character, in which case it will not be printed.

BB5D Txt Wr Char. Write a character to the currently selected stream. Unlike the Txt Output routine, control characters are printed just like any other character.

BB60 Txt Rd Char. Read a character from the screen. If you have used the examples in this book, you are well used to this routine by now! If there is a valid character at the currently select position it is returned in register A, with Carry set. If not, carry is cleared.

BB63 Txt Set Graphic. This routine is used both to enable and to disable graphic character writing. A is non-zero to enable and zero to disable. This equates to using TAG and TAGOFF in BASIC.

BB6C Txt Clear Window. Clear the current window. It is reset to the paper ink of the currently selected stream.

BB6F Txt Set Column. A contains the column on entry. This sets the cursor horizontal position.

BB72 Txt Set Row. A contains the row.

BB75 Txt Set Cursor. H contains the required column and L the required row. So this performs a function equivalent to the above two.

BB90 Txt Set Pen. A contains the ink to be used. The system masks the value to ensure it is brought within the required range.

BB96 Txt Set Paper. A contains the ink for setting the paper.

BB9F Txt Set Background. If A is zero, transparency is disabled. If non-zero it is enabled. This affects the way text characters are written to the screen; if transparency is enabled, the background is not cleared first, so the character is written on top of whatever is there already. Mainly intended for labelling charts, etc. Note: In graphics mode, the background is ALWAYS cleared first, regardless of the transparency setting.

BBB4 Txt Stream Select. A contains the stream number 0 - 7 to be selected for future writing.

BBBA Gra Initialise. The Graphics VDU stream is initialised as for system reset.

BBCO Gra Move Absolute. DE contains the required X address (horizontal position) and HL the required Y address. These positions can be outside the window if required.

BBDB Gra Clear Window. The graphics window is cleared to the current graphics paper ink.

BBDE Gra Set Pen. The Graphics plotting ink is set to the ink contained in register A.

BBE4 Gra Set Paper. The background ink is set to the colour contained in A.

BBEA Gra Plot Absolute. A point is plotted at the X coordinate (contained in DE) and the Y coordinate (contained in HL).

BBF6 Gra Line Absolute. A line is drawn to the X,Y coordinates supplied in DE, HL.

BBFC Gra Wr Char. A Character, contained in A, is written to the screen at the current graphics position.

BC14 Scr Clear. The screen is cleared to ink 0.

There are of course many more, and there are many more details that could be written on the above routines. For a much fuller explanation see the Technical Manual. The above is intended to give you a flavour of what is available.

A Subroutine Example.

The following subroutine is taken from the Ghosts in a Bottle Program in chapter 13. It illustrates 3 of the above routines being called, and shows a possible way of interfacing to BASIC programs.

The routine actually contains 2 subroutines. The first, at 7000h reads the character situated at the current location, stores the character in SAVECHAR (7007h) and returns to BASIC. BASIC can retrieve the character by PEEKing 7007h.

The second routine, much longer, takes care of moving the ghosts round the screen. The function required is to look at each ghost in turn, and to move it closer to where the man is situated (indicated by the MX,MY coordinates which have been POKEd by BASIC).

Each ghost's coordinates are adjusted up or down according to whether the man is above or below and to left or right. This gives a new set of coordinates which are saved on the stack at instruction G4. The cursor is positioned at the new position and the character already there is checked. If it is the man, the search is over and the DONEFLAG is set on as an indicator to BASIC. If it is not a space no action is taken, but if it is clear to write, a space is written onto the ghost's current location and the ghost is rewritten at the new location. Then the new coordinates are stored for the next iteration.

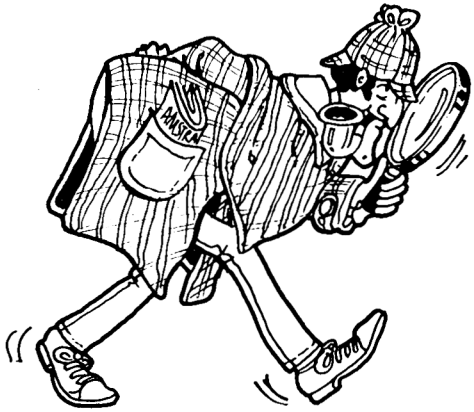
```
1                       MAN:        EQU  249
2                       GHOST:      EQU  225
3                       SPACE:      EQU  32
4                                   ORG  7000H
5                                   LOAD $
6                       TXTSETCUR:  EQU  0BB75H
7                       TXTWRCHAR:  EQU  0BB5DH
8                       TXTRDCHAR:  EQU  0BB60H
9  7000 CD60BB          RD:         CALL TXTRDCHAR
10 7003 320770                      LD   (SAVECHAR),A
11 7006 C9                          RET
12 7007 00              SAVECHAR:   DB   0
13                      MVGHOSTS:   EQU  $
14 7008 AF                          XOR  A
15 7009 32FE70                      LD   (DONEFLAG),A       ;show not done
16 700C 3AFF70                      LD   A,(NUMGHOSTS)
17 700F 32FD70                      LD   (TEMPGHOSTS),A
18 7012 2A0071                      LD   HL,(MX)            ;Get Man's coords in HL
19 7015 E5                          PUSH HL
20 7016 C1                          POP  BC                 ;Change to BC
21 7017 210271                      LD   HL,G1X
22 701A C5              G0:         PUSH BC
23 701B 56                          LD   D,(HL)             ;D= GX
24 701C 23                          INC  HL
25 701D 5E                          LD   E,(HL)             ;E= GY
26 701E 23                          INC  HL
27 701F E5                          PUSH HL                 ; Ptr to next coords.
28 7020 D5                          PUSH DE                 ;Current Ghost coords.
29 7021 79                          LD   A,C                ;MX
30 7022 BA                          CP   D
31 7023 2806                        JR   Z,G2
32 7025 3803                        JR   C,G1
33 7027 14                          INC  D                  ;GX+1
34 7028 1801                        JR   G2
35 702A 15              G1:         DEC  D                  ;GX-1
36 702B 78              G2:         LD   A,B                ;MY
37 702C BB                          CP   E
38 702D 2806                        JR   Z,G4
39 702F 3803                        JR   C,G3
40 7031 1C                          INC  E                  ;GY+1
41 7032 1801                        JR   G4
42 7034 1D              G3:         DEC  E                  ;GY-1
43 7035 EB              G4:         EX   DE,HL
44 7036 E5                          PUSH HL                 ;Save NGX,NGY
45 7037 CD75BB                      CALL TXTSETCUR
```

```
46 703A CD60BB              CALL TXTRDCHAR
47 703D D1                  POP  DE
48 703E E1                  POP  HL
49 703F D5                  PUSH DE
50 7040 FEF9                CP   MAN
51 7042 200A                JR   NZ,NOTDONE
52              DONE:       EQU  $
53 7044 3E01                LD   A,1
54 7046 32FE70              LD   (DONEFLAG),A
55 7049 32FD70              LD   (TEMPGHOSTS),A
56 704C 1804                JR   WRITE
57              NOTDONE:    EQU  $
58 704E FE20                CP   SPACE
59 7050 201C                JR   NZ,NOWRITE
60              WRITE:      EQU  $
61 7052 CD75BB              CALL TXTSETCUR
62 7055 3E20                LD   A,SPACE
63 7057 CD5DBB              CALL TXTWRCHAR
64 705A E1                  POP  HL
65 705B E5                  PUSH HL
66 705C CD75BB              CALL TXTSETCUR
67 705F 3EE1                LD   A,GHOST
68 7061 CD5DBB              CALL TXTWRCHAR
69 7064 D1                  POP  DE
70 7065 E1                  POP  HL
71 7066 2B                  DEC  HL
72 7067 2B                  DEC  HL
73 7068 72                  LD   (HL),D
74 7069 23                  INC  HL
75 706A 73                  LD   (HL),E
76 706B 23                  INC  HL
77 706C E5                  PUSH HL                 ;Save ptr to coords
78 706D D5                  PUSH DE
79              NOWRITE:    EQU  $
80 706E 3AFD70              LD   A,(TEMPGHOSTS)
81 7071 3D                  DEC  A
82 7072 32FD70              LD   (TEMPGHOSTS),A
83 7075 D1                  POP  DE
84 7076 E1                  POP  HL
85 7077 C1                  POP  BC
86 7078 20A0                JR   NZ,GO
87 707A C9                  RET                     ;Back to BASIC.
88                          ORG  70FDH
89                          LOAD $
90 70FD 00      TEMPGHOSTS: DB   0
91 70FE 00      DONEFLAG:   DB   0
92 70FF 00      NUMGHOSTS:  DB   0
```

```
 93 7100 0000    MX:         DW  0
 94 7102 0000    G1X:        DW  0
 95 7104 0000    G2X:        DW  0
 96 7106 0000    G3X:        DW  0
 97 7108 0000    G4X:        DW  0
 98 710A 0000    G5X:        DW  0
 99 710C 0000    G6X:        DW  0
100 710E 0000    G7X:        DW  0
101 7110 0000    G8X:        DW  0
102 7112 0000    G9X:        DW  0
103 7114 0000    G10X:       DW  0
104                          END
```

PART V

SERIOUS APPLICATIONS



This part contains an overview of what needs to be considered when programming business-like applications for the Amstrad. Considerations of screen design, option selection and user input are discussed, and the section includes a complete Home Accounting Program.

CHAPTER 17
The Design of a Home Accounting Program

The programs we have written so far have been fairly light-hearted ventures, written for our own amusement. It is time to take a look at a program with a more serious intent, a program to sort out our home accounts.

The Amstrad should not be thought of purely as a games systems. The BASIC is fast enough for serious applications, and has several powerful facilities which enable us to manipulate character strings, format printing in columns, and so on.

Of course, not having disk drives (yet!) imposes some limitations; we shall not want to be reading from or writing to tape very often, it would be too slow. But, to compensate, there is a more than ample amount of RAM available, so we shall read all we need into memory at the start, manipulate it there, and write it back only when we have finished.

What Functions?

What do we want a home accounting program to do? Tell us 'where we stand' at any time (if we are brave enough!), help check our bank statements, analyse where the money has gone so that we may be able to plan better tomorrow, keep things in balance so that if we have one account in the black while another - say a credit cart account - is in the red, we can transfer funds and so avoid interest payments.

That will do for a start. So much for the functions. How about the design?

Design Guidelines.

We want it to be easy to use, and fast to display information. We are not good typists, so we want the minimum number of keystrokes, which implies the program should be able to guess at default values on occasion. We want a certain amount of checking to avoid errors where we are keying a date into an amount field, but we are not looking for the sort of rigorous checking we would insert in a commercial program. We do not need to protect ourselves against fraud, after all!

How about the display? Should it be 40 or 80 columns? There seems no reason why we should not use both at different times. Should we use colour, or is that just a gimmick in this type of program?

Of course my decisions in the above matters may not be the same as yours, which is why I have included the source code, so that things can be changed if you wish.

Menu Selection.

One of the early design questions we face is how the user will select a routine. We can already see there are going to be many routines such as 'Display current balances', 'List all payments this month', 'Show me what I have spent on the car since September'. We want a way of selecting with the minimum of fuss and keystrokes.

There are several options. The simplest, and one you will see frequently in commercial programs, is the numbered menu:

1. Show Current Balances.
2. List all Payments.
3. Analyse Payments by category.
4. ...

Select number and press ENTER.

This is simple to set up, and it works quite well when there are a maximum of (say) 20 options. Beyond that the screen gets very cluttered, and you have to descend into submenus. Once you are forced to take that step the numbers become difficult to remember and you tend to make mistakes.

An alternative is the menu where you select by entering the first letter or letters of the option:

List Balances
List Payments
Analyse Payments

Enter starting letters to select option: [ ]

and the user can type LB, LP or AP.

This is much easier for the user to remember providing the options are well phrased, and providing there are not some that clash. It becomes more difficult to choose suitable option names after 20 or so, and the scheme loses some of its attractiveness.

A third scheme, much in vogue at the present, is to 'point to' an option by means of an arrow, or a cursor, and to select it by means of the ENTER key. Once you have chosen, a list of further choices may appear, and selection is made in the same way.

Several schemes of this sort exist, centred round the use of a mouse, although the computer software industry as a whole seems to be taking a 'wait and see' attitude towards their use. Is it just a gimmick?

It must be admitted that a mouse is useful for menu selection. The act of pointing is a natural one, and it does not matter at all if the option you want is next door or in the other corner of the screen, distance is no object.

Well, the Amstrad has no mouse (as yet!) but there is no reason why we cannot use the 'point-and-select' technique for our home accounting menu. There is no definitive answer as to which technique is the best, but I have made the undemocratic choice for you in this case.

Screen Design.

A book could be written on the principles of interfacing with the screen (and probably has!). Suffice it to say that we will divide our screen into 3 logical areas - heading, body and prompt. The heading will take up the top 7 lines, the body (where most of the detail will appear) lines 8 to 23, and there will be a single line for prompts and error messages on line 25. A consistent screen design is reassuring to the user.

Coming back to the question of 40 vs 80 columns, I decided to use 40 wherever possible, for clarity, but to change to 80 when forced to by the nature of the data being displayed. Most of the data entry can be done on a 40 column display.

I also decided to select black as the background colour and white as the pen colour. This is recommended for maximum clarity in mode 2 on the colour display, and so I have adopted it throughout. Light blue is used to show the user where his input is expected (not on the 80-column display of course).

Handling Input.

So much for the look of the screen, how about the inputting side? It may seem obvious that the best statement is the INPUT or LINE INPUT statement, but that is actually not true at all, and it is a statement best avoided. If you program

```
INPUT "Enter Amount ",z
```

and the user mistakenly types

```
current account
```

BASIC snaps back with a 'REDO from START' message which untidies your attractive screen display as it is most unlikely to come out on line 25 where all other error-messages occur.

Further disadvantages to the use of INPUT are:

- you, the programmer, can do nothing while the data is being entered; you do not receive control until the user presses ENTER. If he has entered alpha data into what is meant to be a numeric field, you do not know until he has finished.

- If he continues without pressing ENTER he could type 256 characters before BASIC gets tired, and your screen display will be destroyed.

You may wonder what sort of idiotic user I have in mind. But I believe that in writing this sort of program you should assume the worst. You cannot prevent against deliberate sabotage, but you can and should protect your program against misuse by technically naive users.

So the first task is to write an input subroutine centred round the use of INKEY$. This means:

- we can set a maximum length for each field entered, and not let the user go past it.

- we can choose the type of data to be entered, and verify it (to a point) as each keystroke is made, refusing to allow graphic symbols in names, alpha strings in numbers, etc.

- we can indicate visually the boundaries of the field to be entered.

- we can use the same routine for cases where we do not want to wait for the ENTER key to be pressed - for example when we are moving an arrow round the screen during menu selection.

In the home accounting program this subroutine is at 8000-8999 and I hope you will find it useful in other programs. Note that it is not a definitive solution, you could decide to make the error checking more stringent, but it should give you ideas for improvement. There is an explanation of the coding used in this subroutine in chapter 19.

CHAPTER 18
INSTRUCTIONS FOR USE

The Home Accounting Program is listed in the next chapter. It is a long program and will take you a long time to key in, with every chance of introducing a few errors along the way. It has therefore been made available on tape from Kuma Computers. The tape has not been protected, so you can study the source listing and amend it if you wish.

Reset the system and execute the program by inserting the program cassette and holding down CTRL and pressing the small ENTER key. The program is a large one and will take several minutes to load if it has been recorded at the slow speed. Therefore it may well be worth your while saving it at the higher speed.

You will be asked for the date. Enter it as a 6-digit number in the form day-month-year.

A prompt asks you for a file name. The program can hold several hundred payments and receipts spread over up to 10 accounts. For most users it will be convenient to keep 3 months' data on one tape, so you may decide to call the files 1Q84, 2Q84, etc. It is worth including the current date, ie. the date you created the data in the filename also, so a full name might be '1Q84 4/5/84'. You are allowed a maximum of 16 characters.

You will then be asked whether the file already exists. If this is the first time of use it will obviously not exist, so answer N (for No). Of course you will need a blank tape or two ready.

Option Selection.

The screen then changes to a menu selection, with an arrow indicating the current choice, of which there are 6. A list of suboptions, on the right of the screen relates to the choice to which the arrow is pointing, 'Accounts'.

The up and down arrow keys are used to position the screen arrow where you want it. Experiment by pressing the keys and you will find that only 5 keys have any effect:

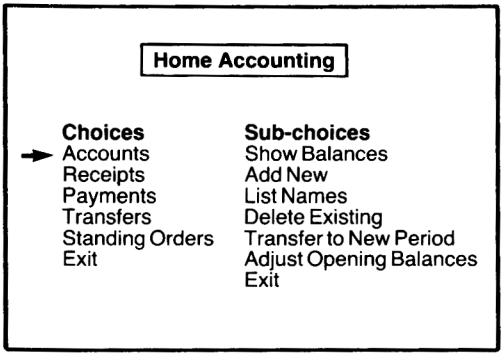
- The ESC key (which you should not use since there is no reason why you should want to break into the program).

- The up and down arrow keys. Pressing the up-arrow key when you are already on the top line will take you to the bottom line, and vice versa.

- The ENTER key selects your choice by 'freezing' the left-hand arrow and by offering you a second arrow opposite the list of subchoices.

- The CLR key is a 'quick-escape' key, which takes you back to the list of choices if you were among the subchoices, and positions you at the exit if you were already on the left-hand side.

The screen handling is meant to be easy! Make sure you are well acquainted with it.



Get Familiar!

Before trying to put on 'real' data you are strongly advised to get familiar with all the options of the program. Enter a few account names, a few payment and receipt categories and methods, then enter 20 or 30 transactions, and display the current balances, the analysis of payments, etc. This will help you decide on which categories you wish to adopt. Far better to get them right at the beginning than to change them later, even though the facilities are there for you to do exactly that if you wish.

Setting up Initial Data.

Assuming this is the first time of use, you will have to enter some constant data, and you will need to think carefully of how you wish to hold the data before you enter it. The program allows you to change your mind later and add extra payment categories or whatever, but the advantage of choosing a scheme and staying with it is that you can compare two periods easily.

The decisions to be taken at this stage are:

- the number of accounts.
- the payment categories.
- the payment methods.
- the receipt categories.
- the receipt methods.

In addition you will want to enter details of standing orders and direct debits.

Program Capacities.

Because this is a tape based system we accumulate all the data in memory and write it out at the end. This means there is a limit to how much data we can hold at any time. The limits are:

10 accounts
30 payment categories
15 receipt categories
20 payment methods
10 receipt methods
150 payment transactions
100 receipt transactions
20 standing orders or direct debits
20 transfers between accounts.

You may find that the program slows up when you are approaching the limits. This is because housekeeping is taking place frequently to allow space for variables to be manipulated - the so-called 'garbage collection'. It is therefore adviseable to keep comfortably below the limits. A routine exists to transfer the constant data and the closing balances of one period to become the opening balances of the next.

When you get used to the system you may find - for example - that you need more payment categories, but never use more than 3 receipt categories. By all means experiment and alter the limits to suit yourself. The program is designed so that such change is easy. See the program commentary in the next chapter which will tell you what statements to change.

Choice 1 - Accounts.

In the context of this program an account can be a current bank account, a deposit account, a Giro account, a credit-card account, etc. If you use a credit card you may prefer merely to keep records of your current account, and to record cheques sent to the credit card company as payments. I find it more convenient, although a little more work, to keep a record of the credit card balance and transactions, so I enter the credit card as an account, and a cheque drawn on the current account made out to the credit card account is then a transfer, not a payment.

The purpose of a program like this is to enable you to manage your affairs the way you would like, not to enforce a totally alien way of working on you. So experiment to find out what way suits you!

Choose ACCOUNTS and you will see 7 subchoices:

```
SHOW BALANCES
ADD NEW
LIST NAMES
DELETE EXISTING
TRANSFER TO NEW PERIOD
ADJUST OPENING BALANCES
EXIT
```

Accounts Subchoice 1 - Show Balances.

Displays all accounts with the opening balances, payments, receipts, transfers and closing balances. Needs 80 columns. A frequently used option!

Account Subchoice 2 - Add New.

Enables a new account to be added. All that is necessary is the name which may be up to 20 characters long.

Account Subchoice 3 - List Names.

The names of all the accounts are listed.

Account Subchoice 4 - Delete Existing.

Any account can be deleted. However, if you delete an account with existing transactions or balance it will make a nonsense of your figures, so this option should very rarely be used.

Account Subchoice 5 - Transfer to New Period.

When you have completed a file for one period and wish to move forward to the next because you are approaching the data limits, use this option which will:

- calculate the closing balance for each account to become the opening balance for the new period.

- clear out all details of payments made, amounts received, and transfers made.

Constant data, namely account names, payment categories and methods, receipt categories and methods, and standing orders are maintained intact so do not need to be entered in the new period.

Note that this option does NOT create a file, you still need to save the new period data before exiting from the program. Of course you must also have saved the old period data before running this option.

Account Subchoice 6 - Adjust Opening Balance.

With most accounts you will need to enter an opening balance when you first use the program, since it is unlikely to be zero. Of course in subsequent periods the opening balance is created not by using this option but by transferring the balances from the previous period using subchoice 5.

If you have run subchoice 5, transferring the balances from a previous period, then subsequently find an error in the previous period which obliges you to amend the data, you can use this option to adjust the opening balance to take care of the error.

Account Subchoice 7 - Exit.

This deletes the right-hand arrow and returns you to the menu of choices on the left-hand side.

Choice 2 - Receipts.

There are 11 subchoices:

ADD NEW
DELETE
LIST
SUMMARISE
LIST CATEGORIES
ADD CATEGORY
DELETE CATEGORY
LIST METHODS
ADD METHOD
DELETE METHOD
EXIT

Receipts Subchoice 1 - Add New.

Up to 100 receipts can be stored in total. You will normally be entering from a paying-in counterfoil or book of counterfoils. Receipts for one account are dealt with at one time.

You are prompted for the name of the account. Enter the first letter of the account, and if valid, the account name is displayed. Then enter up to 16 receipts, the data for each one being:

- The Date, as DDMM. After the first line, the program remembers the date you used last time, so you can press ENTER to accept it as the default. This saves keying. Overwrite it with a new date if you need to.

- A transaction identifier. Any 3 characters will do, or you can press ENTER to omit if you wish.

- The amount. Up to #32767 plus 2 decimal places are allowed, which should cater for most of us.

- Category. This is used to allow you to analyse your receipts over a period. You may decide on a breakdown such as SALARY, INTEREST RECEIVED, GIFT (if you are so lucky!), TAX REBATE (we can all dream), etc. Of course you cannot use this option until you have selected and entered categories using subchoice 6. Categories are identified by the first 2 characters of their name.

16 entries should be ample for most people, but if you wish to enter more, the screen will be cleared and you can continue. If you fill up the 100 allocation, you will be informed that there is no more space, and you can use the option to transfer all data to a new file before continuing.

Note that if you make an error while entering a field you can use the left arrow or the DELete key to move back and correct. If you notice the error after you have pressed ENTER but before you have finished the line, you can press ENTER when asked for a category or an amount and the system will 'back-pedal' and move you back to a preceding field. If you only notice an error after you have completed the entire line you will have to use the DELETE EXISTING option to delete the entry, then return to this option to enter it correctly.

You will normally wish to enter in date order, but this is not obligatory.

Receipts Subchoice 2 - Delete.

Use this to delete any receipt entered in error. You can scan through the list of receipts, using the up and down arrow to move up and down the page, and using the left arrow and the ENTER key to move forward and back pages. When the arrow is opposite the entry you wish to delete, press D or d and the entry is at once deleted. So make sure you have the right one!

Press the CLR key to return to the menu.

Receipts Subchoice 3 - List.

Use this to scan all receipts. See the instructions under subchoice 2 for details of how to scan forward or back.

Receipts Subchoice 4 - Summarise.

This option analyses all your receipts for the period by category and lists them.

Receipts Subchoice 5 - List Categories.

Use this to remind you what categories you have selected. Upto 15 are allowed.

Receipts Subchoice 6 - Add a Category.

Use this to add a new receipt category. The first 2 characters must be unique, so your option will be rejected if they have already been used for another category. This is sometimes restrictive, but it is a distinct advantage when entering a category to be able to remember the code without an artificial system such as 1 for SALARY, 2 for BANK INTEREST, etc. which you are forever forgetting.

Receipts Subchoice 7 - Delete a Category.

You can delete a receipts category at any time, but your figures will be invalid if you delete a category in use. So use option 5 first, to check the status of the category in question.

Receipts Subchoice 8 - List Methods.

You can choose to differentiate between different methods of receiving money into the accounts. This can be useful if you are checking later. Examples of different methods are: PAYING IN BOOK 1, ADJUSTMENT, BANK GIRO. Of course if you do not want to use this option, create one method called MISCELLANEOUS and use only that!

Receipts Subchoice 9 - Add New Method.

You can have upto 10 methods of receiving amounts.

Receipts Subchoice 10 - Delete Existing Method.

Be careful not to delete a method already in use.

Receipts Subchoice 11 - Exit.

Leave Receipts subchoices and return to left-hand choices.

Choice 3 - Payments.

There are exactly the same 11 subchoices as for receipts:

ADD NEW
DELETE
LIST
SUMMARISE
LIST CATEGORIES
ADD CATEGORY
DELETE CATEGORY
LIST METHODS
ADD METHOD
DELETE METHOD
EXIT

but you will probably be entering far more payments than receipts, and maintaining more category and method information.

Payments Subchoice 1 - Add New.

Up to 150 payments can be entered in total. The headings are as for receipts. The optional transaction number will very probably be used for the last 3 digits of the cheque number and will be a great help in checking your bank balance.

Payments Subchoice 2 - Delete.

Use this to delete any payment entered in error. See under receipts.

Payments Subchoice 3 - List.

As for receipts.

Payments Subchoice 4 - Summarise.

As for receipts.

Payments Subchoice 5 - List Categories.

Use this to remind you what categories you have selected. Up to 30 are allowed. You will probably have more payment categories than receipt categories. Entries such as MOTORING, COMPUTERS, ELECTRICITY, GAS, WATER RATES, RATES, MORTGAGE, HOUSEKEEPING, HOLIDAYS, etc. will be typical. Notice however that the first two letters must be unique, so that MORTGAGE and MOTORING are not allowed together, nor are HOUSEKEEPING and HOLIDAYS. It is usually not difficult to select an alternative name.

Payments Subchoice 6 - Add a Category.

As for receipts.

Payments Subchoice 7 - Delete a Category.

As for receipts.

Payments Subchoice 8 - List Methods.

Typical methods will be CHEQUE, INTEREST CHARGED, DEDUCTION AT SOURCE, STANDING ORDER.

Note that if you choose to use the standing order option (see below) you must ensure you create a payment method STANDING ORDER, because that option inserts payment entries with a category code of 'ST'.

Upto 20 payment methods are allowed.

Payments Subchoice 9 - Add New Method.

As for receipts.

Payments Subchoice 10 - Delete Existing Method.

As for receipts. Once again, be careful not to delete a method already in use.

Payments Subchoice 11 - Exit.

Leave Payments subchoices and return to left-hand choices.

Choice 4 - Transfers.

The transfer facility is used to record transfers between your accounts, which are not payments or receipts in that they do not affect your net position, only where the money is kept. You can record up to 20 transfers in any period.

The subchoices are:

LIST
ADD
DELETE
EXIT.

Transfers Subchoice 1 - List.

Lists all transfers for this period using the 80-column screen.

Transfers Subchoice 2 - Add.

When adding a new transfer you will be asked for:

- The 'From Account', ie. the account giving up the money. For example you may well be paying some money to your credit card accounts to settle up in full or in part, in which case the From Account is probably the Current Account.

- Date (DDMM). A default may be chosen after the first entry.

- A Transaction number (optional). May be the last 3 digits of a cheque number, or anything else you wish.

- The amount.

- The 'To Account'. The account receiving the money.

Once again, error-correction facilites are provided, so that you can back-track if you discover errors before the end of the line.

Transfers Subchoice 3 - Delete.

Use this to take care of any error made while adding transfers.

Transfers Subchoice 4 - Exit.

Return to the list of choices.

Choice 5 - Standing Orders.

This option is intended to be used both for standing orders and for direct debits. It covers only the payment side, there is no equivalent on the receipt side, as most people receive only 1 regular receipt - the salary cheque.

The standing order file is a memorandum file only, nothing is activated until you use the CONFIRM option which uses the standing order information to make a payment transaction.

Entries in this file do not need to have a month entered. For example if you have a payment which is made on the 16th of each month, enter the date as 1600, ie. the month being 0. This is printed as a blank, but reminds you that the order is a monthly one. For an annual payment you will enter the full date.

The subchoices are:

```
LIST
ADD
DELETE
CONFIRM
EXIT
```

Standing Order Subchoice 1 - List.

There are a maximum of 20 standing orders, or direct debits. Remember that the existence of an order in the list does not mean that payment is made automatically. It is only made when you confirm it, using the CONFIRM option.

Standing Order Subchoice 2 - Add.

Use this to add a new standing order. See above for instructions on how to enter the date.

Standing Order Subchoice 3 - Delete.

Use this to delete a standing order. This will not affect any payments already made under this order.

Standing Order Subchoice 1 - Confirm.

You should review the standing orders once a month and confirm them as required. It is only when confirming that a payment transaction is actually recorded. While using this option you have the option to adjust the payment actually made, since this occasionally changes by 1p or 2p due to rounding etc.

Standing Order Subchoice 1 - Exit.

Use this to leave the standing order subchoice and to return to the left-hand choices.

Choice 6 - Exit.

You select this when you have finished for the day and wish to leave the program. If you have been merely reviewing the data and have not changed anything you are allowed to exit at once, but if you have updated anything the program will not be happy until you have inserted a data tape and have saved the file.

Do not of course overwrite your most recent file! If anything happens to the tape you have then lost the lot. Keep several tapes in circulation, they are cheap enough if you use C12s.

CHAPTER 19
PROGRAM COMMENTARY

The home accounting program is written in a fashion that should allow it to be understood and modified. Because it is a large program and it is desired to keep a considerable amount of data in memory in addition, it means that the program has to be written in a condensed style with few comments and frequently with several statements on one line. So this chapter serves as the comments.

The main thread of the program is short, lines 5 to 999. BREAK is intercepted so that no data is lost. You can always pull the plug out...

Initialisation is at 1000-1999. Experiment with colour by all means by altering 1010-1020. Feel free to change the maximum quantities by altering lines 1100-1180.

If you wish to send an initialisation string to a printer, do it here.

The file is loaded in 2000-2499 and saved in 2500-2999. Any changes to the data format must of course be mirrored in the other routine.

The date is entered as DDMMYY. Note that it is not verified, you could choose the 31st February if you insist. In a commercial program the validation would be much tighter.

The menu is at 3000-3999. Note how use is made of the window facility of Amstrad BASIC to facilitate the movement of arrows and the printing of lists of choices.

4000 is the clear screen routine. This is made into a subroutine so that LN which is used in places as a line count can be reset.

4100 prints a 'Press ENTER to continue' message on the bottom line of the screen and waits for any key to be pressed.

4200 changes the display to mode 2 (80 columns). SCRWIDTH, which is used by the routines that centre titles, is set to 80.

4300 changes the display back to 40 columns (mode 1).

4400 asks the user for a 'screen or printer' decision, and sets appropriate flags.

4500 checks for the end of page on screen or printer.

4600 ejects if output is to printer.

4700 sets up windows.

4800 sets up screen parameters for listing.

4900 sets up printer parameters for listing.

5000 routes to the appropriate function according to the choice (y2) and the subchoice (y3).

6000 lists the account names. It will need altering if you increase the number of accounts from 10 beyond what will fit on a single screen (16 or so).

6500 enables a new account to be added. Note that a variable UPDATED is set on once a new account has been added. All routines that alter (ADD, DELETE, CONFIRM) data set this flag, so that the system can tell whether an output file needs to be written when you come to exit.

7000 deletes an account. Statements could be added to calculate the closing balance, and to reject the deletion request if it was non-zero, but I do not feel this is necessary in a home accounts program. We are after all, not looking over our shoulders for the auditors!

7500 calculates and displays the balance for each account. This is quite quick even with a large amount of data.

8000 is the general-purpose input subroutine and I believe you will find it useful in many circumstances. It expects the following parameters on input:

T - Type of data expected. May be 1 (A-Z, a-z, space, quote and hyphen only), 2 (numeric between the limits of 32767 and -32768), 3 (any printable characters), 4 (must be a character contained within the string x2$), or 5 (numeric outside the above range and principally used for dates).

L - Maximum length allowed. Not used for T=4, which always expects a single character.

X3$ - Default (optional). If x3$ is set, it is displayed and if the user just presses ENTER, the contents of x3$ are taken as the default, but if the user presses any key other than ENTER, the default is lost and data must be entered.

The subroutine clears x3$, so it must be reset each time it is called.

X2$ For T=4 only. A string of valid characters. Entering any one of these will cause the subroutine to exit.

Data entered is returned as follows:

If T=1 Data in x1$, length in z0.
If T=2 Data in z0 and in x1$
If T=3 Data in x1$, length in z0.
If T=4 Position of character in x2$ returned in z0.
If T=5 Data in x1$, length in z0.

If the user presses the CLR button at any time during input, (CLR is used instead of ESC, since ESC is intercepted by BASIC), the following action occurs:

- any data already inputted is cleared
- the screen data is cleared
- T is set to -5

so CLR is a general escape route.

Note that the routine uses, and may destroy x,y x1$, z0, x3$, and z$.

9000-9499 adds a payment or a receipt. Payments and receipts share common routines, since most of the coding is the same. Note that transactions are held economically as 21-byte strings in the format:

Ammddddnnnaaaaaaaaacc

where:
A = account code
mm = Payment method code
dddd = date (ddmm)
nnn = transaction code
aaaaaaaaa = amount
cc = payment category code.

9500-9999 is the delete payment or receipt transaction routine.

10000-10999 is the handling routine for errors and breaks.

11000-11299 covers the listing of payment or receipt transactions. There is a subroutine at 11300 which handles the actual printing, which is also called by the delete routine. The printing routine is designed to allow paging in either direction.

11500-11999 allows payments or receipts to be summarised.

12000-12499 lists payment categories, 2 to a line.

12500-12999 adds a new payment or receipt category and 13000-13499 deletes a category.

13500-13999 lists methods, again 2 to a line, 14000-14499 adds a new method and 14500-14999 deletes one.

15000-15499 lists standing orders, 15500-15999 adds a new one and 16000-16499 deletes one. 16500-16999 confirms a standing order and adds a payment transaction.

Standing orders are kept in the array so$() in the format:

Addddaaaaaaaacc

where:

A = Account code
dddd = date (DDMM). MM can be blank.
aaaaaaaaa = amount
cc = category code

Transfers are catered for by 17000-17499 which lists them, 17500-17999 which adds a new one, and 18000-18499 which deletes one.

The routine at 18500 covers the transfer to a new period, and the routine at 19000 handles the adjustment of opening balances.

Transfers are held in the array xfer$() in the format:

Fddddxxxaaaaaaaaa T

where

F = From Account code
dddd = date (DDMM)
xxx = optional transaction number.
aaaaaaaaa = amount
T = To Account code.

```
5 ON BREAK GOSUB 10020
10 REM housekeeping program
20 REM copyright (c) 1984 John Braga
30 ON ERROR GOTO 10000
40 GOSUB 1000
50 GOSUB 2000
60 GOSUB 3000
65 IF Y2=CHOICES THEN 100
70 GOSUB 5000
80 GOTO 60
100 IF updated THEN GOSUB 2500
110 GOSUB 4000
999 END
1000 REM init
1005 DEFINT l,s,x
1010 bkcol=0:incol=2:pencol=26:titlecol=
20
1020 MODE 1:BORDER bkcol:INK 0,bkcol:INK
 1,pencol:INK 2,incol:INK 3,titlecol:PAP
ER 0:PEN 1
1025 scrwidth=40:ESC=16
1040 bk$=CHR$(8):sign$="-->":lft$=CHR$(2
42):up$=CHR$(240):down$=CHR$(241):enter$
=CHR$(13):cleosc$=CHR$(20):cleol$=CHR$(1
8):bel$=CHR$(7)
1050 true=-1 : false=0
1060 prlength=66:prlines=60:scrlines=16
1070 GOSUB 4700:screen=true
1100 accounts=0:maxac=10:DIM balo(maxac)
,balc(maxac),acname$(maxac)
1110 reccats=0:maxreccats=15:DIM reccat$
(maxreccats)
1120 paycats=0:maxpaycats=30:DIM paycat$
(maxpaycats)
1130 paymeths=0:maxpaymeths=20:DIM payme
th$(maxpaymeths)
1140 recmeths=0:maxrecmeths=10:DIM recme
th$(maxrecmeths)
1150 pays=0:maxpays=150:DIM pay$(maxpays
)
1160 recs=0:maxrecs=100:DIM rec$(maxrecs
)
1170 sos=0:maxsos=20:DIM so$(maxsos)
1180 xfers=0:maxxfers=20:DIM xfer$(maxxf
ers)
```

```
1999 RETURN
2000 REM load
2010 x$="Personal Accounting Program":y=
3:GOSUB 4050
2025 LOCATE 9,12:PRINT "Date (ddmmyy) ";
:t=5:l=6:GOSUB 8000:IF z0<>6 THEN 2025 E
LSE date$=LEFT$(x1$,2)+"/"+MID$(x1$,3,2)
+"/"+RIGHT$(x1$,2)
2030 LOCATE 9,14:PRINT "File name ";:t=3
:l=16:GOSUB 8000:IF x1$="" THEN 2030 ELS
E f$=UPPER$(x1$)
2040 LOCATE 5,16:PRINT "Is this an exist
ing file? (Y/N) ";:t=1:l=1:GOSUB 8000
2050 x1$=UPPER$(x1$)
2060 IF x1$="N" THEN 2499 ELSE IF x1$<>"
Y" THEN 2040
2100 LOCATE 1,18 : PRINT "Load tape"
2110 OPENIN f$
2120 INPUT #9,x$,f$,accounts,pays,recs,s
os,xfers,paycats,reccats,paymeths,recmet
hs
2130 FOR z0=1 TO accounts:INPUT#9,acname
$(z0),balo(z0):NEXT z0
2140 FOR z0=1 TO pays:INPUT #9,pay$(z0):
NEXT z0
2150 FOR z0=1 TO recs:INPUT #9,rec$(z0):
NEXT z0
2160 FOR z0=1 TO sos:INPUT #9,so$(z0):NE
XT z0
2170 FOR z0=1 TO xfers:INPUT #9,xfer$(z0
):NEXT z0
2180 FOR z0=1 TO paycats:INPUT #9,paycat
$(z0):NEXT z0
2190 FOR z0=1 TO reccats:INPUT #9,reccat
$(z0):NEXT z0
2200 FOR z0=1 TO paymeths:INPUT #9,payme
th$(z0):NEXT z0
2210 FOR z0=1 TO recmeths:INPUT #9,recme
th$(z0):NEXT z0
2290 CLOSEIN
2499 RETURN
2500 REM write
2510 GOSUB 4000:x$="Home Accounting":y=1
:GOSUB 4050
```

```
2520 LOCATE 1,7:PRINT "Filename for new
file ";:x3$=f$:t=3:l=16:GOSUB 8000
2525 IF x1$="" THEN 2520 ELSE f$=x1$
2530 OPENOUT f$
2540 WRITE #9,date$,f$,accounts,pays,rec
s,sos,xfers,paycats,reccats,paymeths,rec
meths
2550 FOR z0=1 TO accounts:WRITE#9,acname
$(z0),balo(z0):NEXT z0
2560 FOR z0=1 TO pays:WRITE #9,pay$(z0):
NEXT z0
2570 FOR z0=1 TO recs:WRITE #9,rec$(z0):
NEXT z0
2580 FOR z0=1 TO sos:WRITE #9,so$(z0):NE
XT z0
2590 FOR z0=1 TO xfers:WRITE #9,xfer$(z0
):NEXT z0
2600 FOR z0=1 TO paycats:WRITE #9,paycat
$(z0):NEXT z0
2610 FOR z0=1 TO reccats:WRITE #9,reccat
$(z0):NEXT z0
2620 FOR z0=1 TO paymeths:WRITE #9,payme
th$(z0):NEXT z0
2630 FOR z0=1 TO recmeths:WRITE #9,recme
th$(z0):NEXT z0
2640 updated = false
2700 CLOSEOUT
2999 RETURN
3000 REM menu
3010 GOSUB 4000:x$="Home Accounting":y=1
:GOSUB 4050
3020 choices=6
3030 RESTORE 3055
3040 LOCATE 5,7:PRINT "Choices";TAB(25);
"Sub-choices"
3050 x2$=up$+down$+enter$+CHR$(esc):REST
ORE 3055
3055 DATA Accounts,Receipts,Payments,Tra
nsfers,Standing Orders,Exit
3060 CLS #1:IF y2=0 THEN y2=1
3070 FOR z0=1 TO choices:READ x$:PRINT #
1,x$:NEXT z0
3080 CLS #2:LOCATE #2,1,y2:PRINT #2,sign
$;
```

```
3090 CLS #4:ON VPOS(#2) GOSUB 3300,3350,
3400,3450,3500,3550
3100 IF y3>0 THEN 3170
3105 t=4:l=0:GOSUB 8000
3110 ON z0 GOTO 3120,3140,3160,3545
3120 IF y2=1 THEN y2=choices ELSE y2=y2-
1
3130 GOTO 3080
3140 IF y2=choices THEN y2=1 ELSE y2=y2+
1
3150 GOTO 3080
3160 y3=1
3165 IF subchoices=0 THEN 3900
3170 CLS #3:LOCATE #3,1,y3:PRINT #3,sign
$;
3175 t=4:l=0:GOSUB 8000
3180 ON z0 GOTO 3190,3200,3230,3220
3190 IF y3=1 THEN y3=subchoices ELSE y3=
y3-1
3195 GOTO 3165
3200 IF y3=subchoices THEN y3=1 ELSE y3=
y3+1
3210 GOTO 3165
3220 y3=subchoices
3230 GOTO 3900
3300 RESTORE 3340:subchoices=7
3310 GOTO 3600
3340 DATA Show Balances,Add New,List Nam
es,Delete Existing,New Period,Adjust O/B
al's,Exit
3350 RESTORE 3390:subchoices=11
3360 GOTO 3600
3390 DATA Add New,Delete,List,Summarise,
List Categories,Add Category,Delete Cate
gory,List Methods,Add Method,Delete Meth
od,Exit
3400 RESTORE 3390:subchoices=11
3410 GOTO 3600
3450 RESTORE 3490:subchoices=4
3460 GOTO 3600
3490 DATA List,Add,Delete,Exit
3500 RESTORE 3540:subchoices=5
3510 GOTO 3600
3540 DATA List,Add,Delete,Confirm,Exit
```

```
3545 y2=choices
3550 subchoices=0
3600 FOR z0=1 TO subchoices
3610 READ x$:PRINT #4,x$
3620 NEXT z0
3699 RETURN
3900 REM
3910 IF y3=subchoices THEN CLS #3:y3=0:G
OTO 3050
3999 RETURN
4000 CLS : ln=0 : RETURN
4010 REM centre
4020 x=(scrwidth-LEN(x1$))/2:x1=LEN(x1$)
4030 LOCATE x,y:PRINT x1$;
4049 RETURN
4050 REM hding
4060 x=(scrwidth-2-LEN(x$))/2:x1=LEN(x$)
4062 IF screen THEN 4070
4064 PRINT #8,TAB(x+1);x$;TAB(scrwidth-8
);date$
4066 PRINT #8,TAB(x+1);STRING$(x1,"=")
4068 FOR x1=3 TO 6:PRINT #8:NEXT x1:ln=6
4069 GOTO 4099
4070 LOCATE x,y
4080 PEN 3:PRINT CHR$(135);STRING$(x1,CH
R$(131));CHR$(139)
4085 LOCATE x,y+1:PRINT CHR$(133);x$;CHR
$(138);TAB(scrwidth-8);date$
4090 LOCATE x,y+2:PRINT CHR$(141);STRING
$(x1,CHR$(140));CHR$(142);
4099 PEN 1 : RETURN
4100 REM pause
4110 IF NOT screen THEN 4199
4120 PRINT #6,"Press ENTER  ";:t=1:l=0:G
OSUB 8000:CLS #6
4199 RETURN
4200 REM mode 2
4210 MODE 2
4220 scrwidth=80
4230 BORDER 0: INK 0,0 : INK 1,26
4240 GOSUB 4700
4299 RETURN
4300 REM mode 1
4310 MODE 1
```

```
4320 scrwidth=40
4330 BORDER bkcol:INK 0,bkcol:INK 1,penc
ol
4340 GOSUB 4700
4399 RETURN
4400 REM s/p
4410 PRINT #6,"Screen or printer (S/P)?
";
4420 t=4:l=0:x2$=enter$+"SsPp":GOSUB 800
0
4430 ON z0 GOSUB 4800,4800,4800,4900,490
0
4499 CLS#6 : ln=0:RETURN
4500 REM page
4510 IF ln=lines THEN GOSUB 4600
4520 ln=ln+1
4599 RETURN
4600 REM eject
4610 IF screen THEN GOSUB 4100:CLS #5:GO
TO 4699
4620 FOR ln=ln+1 TO prlength:PRINT #8:NE
XT ln
4699 ln=0:RETURN
4700 REM windows
4710 WINDOW #1,5,20,10,25:WINDOW #2,1,4,
10,25:WINDOW #3,21,24,10,25:WINDOW #4,25
,40,10,25:WINDOW #5,1,80,8,23:WINDOW #6,
1,80,25,25
4799 RETURN
4800 REM screen
4810 prdev1=0:prdev2=5:lines=scrlines:CL
S #5:screen=true:RETURN
4900 REM printer
4910 prdev1=8:prdev2=8:lines=prlines:scr
een=false:RETURN
5000 REM route
5010 ON y2 GOTO 5020,5100,5200,5300,5400
,5500
5020 REM choice 1 - accounts
5030 ON y3 GOSUB 7500,6500,6000,7000,185
00,19000
5099 RETURN
5100 REM choice 2 - receipts
5110 pay=false
```

```
5120 GOTO 5210
5200 REM choice 3
5205 pay=true
5210 ON y3 GOSUB 9000,9500,11000,11500,1
2000,12500,13000,13500,14000,14500
5299 RETURN
5300 REM choice 4
5310 ON y3 GOSUB 17000,17500,18000
5399 RETURN
5400 REM choice 5
5410 ON y3 GOSUB 15000,15500,16000,16500
5499 RETURN
5500 REM choice 6
6000 REM list ac
6010 GOSUB 4400
6020 GOSUB 4000:x$="Account Names":Y=1:G
OSUB 4050
6030 FOR z9=1 TO accounts:GOSUB 4500:PRI
NT #prdev2,acname$(z9)
6050 NEXT z9
6060 GOSUB 4600
6070 GOSUB 4800
6499 RETURN
6500 REM add ac
6510 GOSUB 4000:x$="New Account Addition
":y=1:GOSUB 4050
6515 IF accounts=maxac THEN PRINT #6,"Al
l accounts allocated! Press ENTER ";:t=1
:l=0:GOSUB 8000:GOTO 6999
6520 CLS #5
6530 FOR z0=1 TO accounts
6540 PRINT #5,acname$(z0) : NEXT z0
6600 CLS #6:PRINT #6,"New Account ";:LOC
ATE 13,25:t=1:l=20:GOSUB 8000:CLS#6
6610 IF x1$="" THEN 6999 ELSE x1$=UPPER$
(x1$)
6620 FOR z0=1 TO accounts
6630 IF LEFT$(x1$,1)=LEFT$(acname$(z0),1
) THEN 6600
6640 NEXT z0
6650 accounts=accounts+1:updated=true:ac
name$(accounts)=x1$
6660 PRINT #5,x1$
6670 IF accounts<maxac THEN 6600
```

```
6680 CLS #6:PRINT #6,"All accounts alloc
ated! Press ENTER";:t=1:l=0:GOSUB 8000:C
LS#6
6999 RETURN
7000 REM del ac
7010 GOSUB 4000:x$="Account Deletion":y=
1:GOSUB 4050
7020 CLS #5
7030 FOR z0=1 TO accounts
7040 PRINT #5,acname$(z0)
7050 NEXT z0
7100 CLS#6:PRINT #6,"First letter of del
eted account ";:LOCATE 33,25:t=1:l=1:GOS
UB 8000
7110 IF x1$="" THEN 7499
7120 FOR z0=1 TO accounts : IF UPPER$(x1
$)=LEFT$(acname$(z0),1) THEN 7200
7130 NEXT z0
7140 CLS #6:PRINT #6,"No such account!
Press ENTER ";:t=1:l=0:GOSUB 8000:CLS #6
:GOTO 7100
7200 IF balo(z0)=0 THEN 7300
7210 CLS#6:PRINT #6,"Unable to delete no
n-zero account. ";:t=1:l=0:GOSUB 8000:CL
S#6:GOTO 7100
7300 accounts=accounts-1:updated=true
7305 IF z>accounts THEN 7330
7310 FOR z=z0 TO accounts:acname$(z)=acn
ame$(z+1):balo(z)=balo(z+1):NEXT z
7330 GOTO 7020
7499 RETURN
7500 REM bal
7505 IF accounts=0 THEN 7990 ELSE GOSUB
4400
7507 GOSUB 4200
7510 x$="Current Balances":y=1:GOSUB 405
0
7520 LOCATE 1,7:GOSUB 4500:PRINT #prdev1
,TAB(20);"   Opening    Receipts   Trans
fers        Spent      Closing"
7530 wo=0:tg=0:ts=0:tc=0
7532 FOR z0=1 TO accounts
7534 FOR z1=1 TO xfers:IF LEFT$(xfer$(z1
),1)=LEFT$(acname$(z0),1) THEN ax=ax-VAL
```

```
(MID$(xfer$(z1),9,9))
7536 IF RIGHT$(xfer$(z1),1)=LEFT$(acname
$(z0),1) THEN ax=ax+VAL(MID$(xfer$(z1),9
,9))
7538 NEXT z1
7542 FOR Z1=1 TO PAYS:IF LEFT$(PAY$(Z1),
1)=LEFT$(ACNAME$(Z0),1) THEN AS=AS+VAL(M
ID$(PAY$(Z1),11,9))
7544 NEXT Z1
7546 FOR Z1=1 TO recs:IF LEFT$(rec$(Z1),
1)=LEFT$(ACNAME$(Z0),1) THEN Ag=Ag+VAL(M
ID$(rec$(Z1),11,9))
7548 NEXT z1
7550 ao=balo(z0):balc(z0)=ao-as+ax+ag:ts
=ts+as:tg=tg+ag:tc=tc+balc(z0):wo=wo+ao
7560 GOSUB 4500:PRINT #prdev2,acname$(z0
);TAB(21);
7570 PRINT #prdev2,USING "######.##";ABS
(ao);:IF ao<0 THEN PRINT #prdev2,"cr ";
ELSE PRINT #prdev2,"    ";
7580 IF ag>0 THEN PRINT #prdev2,USING "#
#####.##    ";ag;:AG=0 ELSE PRINT #prdev2
,SPACE$(12);
7590 IF ax<>0 THEN PRINT #prdev2,USING "
######.##    ";ax;:AX=0 ELSE PRINT #prdev
2,SPACE$(12);
7600 IF as>0 THEN PRINT #prdev2,USING "#
#####.##    ";as;:AS=0 ELSE PRINT #prdev2
,SPACE$(12);
7610 PRINT #prdev2,USING "######.##";ABS
(balc(z0));:IF balc(z0)<0 THEN PRINT #pr
dev2,"cr" ELSE PRINT #prdev2
7620 NEXT z0
7630 GOSUB 4500:PRINT #prdev2
7640 GOSUB 4500:PRINT #prdev2,"Total";TA
B(21);
7650 PRINT #prdev2,USING "######.##";ABS
(wo);:IF wo<0 THEN PRINT #prdev2,"cr ";
ELSE PRINT #prdev2,"    ";
7660 PRINT #prdev2,USING "######.##    ";
tg,0,ts;
7670 PRINT #prdev2,USING "######.##";ABS
(tc);:IF tc<0 THEN PRINT #prdev2,"cr" EL
```

```
SE PRINT #prdev2
7990 GOSUB 4600
7999 GOSUB 4300:GOSUB 4800:RETURN
8000 REM get
8005 REM t=1 (alpha),2(numeric),3(any),4
(x2$),5(num>32767)
8010 y=VPOS(#0) : x=POS(#0)
8012 IF scrwidth=80 AND l>0 THEN PRINT b
k$;"[";
8020 PAPER 2:PRINT STRING$(l," ");
8022 IF scrwidth=80 AND l>0 THEN PRINT "
]";
8023 LOCATE x,y
8025 IF x3$<>"" THEN PRINT x3$;:LOCATE x
,y
8030 x1$=""
8040 z$=INKEY$:IF z$="" THEN 8040
8045 IF t=4 THEN z$=UPPER$(z$):GOTO 8095
8050 z0=ASC(z$)
8060 IF z0=13 THEN 8992
8070 IF z0=ESC THEN 8990
8080 IF LEN(x1$)>0 THEN IF z0=242 OR z0=
127 THEN 8800
8090 IF LEN(x1$)=L THEN 8040
8095 ON t GOTO 8100,8200,8300,8400,8200
8100 REM alpha
8105 IF z0=32 OR z0=39 OR z0=45 THEN 850
0
8110 IF z0<48 OR z0>122 THEN 8040
8120 IF z0>57 AND z0<65 THEN 8040
8130 IF z0>90 AND z0<97 THEN 8040 ELSE 8
500
8200 REM num
8210 IF (z0>=45 AND z0<58) AND z0<>47 TH
EN 8500 ELSE 8040
8300 REM any
8310 IF z0<32 THEN 8040 ELSE 8500
8400 REM x2$
8410 z0=INSTR(1,x2$,z$):IF z0=0 THEN 804
0 ELSE 8999
8500 PRINT z$;:x1$=x1$+z$
8510 IF LEN(x1$)=1 THEN PRINT SPC(1-1);:
LOCATE x+1,y
8520 GOTO 8040
```

```
8800 REM bk
8810 PRINT bk$+" "+bk$;:x1$=LEFT$(x1$,LE
N(x1$)-1)
8820 GOTO 8040
8990 t=-5:x1$="":X3$="":z0=0:LOCATE X,Y:
PRINT SPC(L);:GOTO 8999
8992 IF x1$="" AND x3$<>"" THEN x1$=x3$
8993 x3$=""
8995 IF t=2 THEN IF LEN(x1$)>0 AND x1$<>
"." THEN z0=VAL(x1$) ELSE z0=0
8996 IF t<>2 THEN z0=LEN(x1$)
8999 PAPER 0:RETURN
9000 REM add trans
9005 IF pay THEN trans=pays:maxtrans=max
pays:meths=paymeths:cats=paycats:x$="New
 Payments" ELSE trans=recs:maxtrans=maxr
ecs:meths=recmeths:cats=reccats:x$="New
Receipts"
9007 IF meths=0 OR cats=0 OR accounts=0
THEN 9499
9010 GOSUB 4000:y=1:GOSUB 4050
9012 IF trans=maxtrans THEN 9450
9020 LOCATE 1,6:PRINT cleol$+"Account ";
:t=3:l=1:GOSUB 8000:IF x1$="" THEN 9499
ELSE x1$=UPPER$(x1$)
9030 FOR z0=1 TO accounts:IF LEFT$(x1$,1
)=LEFT$(acname$(z0),1) THEN 9045
9040 NEXT z0 : GOTO 9020
9045 x0$=x1$+SPACE$(20)
9050 LOCATE 1,6:PRINT acname$(z0);TAB(25
);"Method ";
9055 IF meths>1 THEN 9060
9057 IF pay THEN x1$=LEFT$(paymeth$(1),2
) ELSE x1$=LEFT$(recmeth$(1),2)
9058 GOTO 9070
9060 LOCATE 32,6:t=3:l=2:GOSUB 8000:IF x
1$="" THEN 9020 ELSE x1$=UPPER$(x1$)
9070 FOR z0=1 TO meths:IF pay THEN x$=pa
ymeth$(z0) ELSE x$=recmeth$(z0)
9072 IF LEFT$(x1$,2)=LEFT$(x$,2) THEN 90
90
9080 NEXT z0 : GOTO 9060
9090 MID$(x0$,2,2)=x1$:LOCATE 25,6:PRINT
 cleol$+x$
```

```
9100 LOCATE 1,7:PRINT "DDMM No. Amount
  Category":ln=8:x3$=LEFT$(date$,2)+MID$
(date$,4,2)
9110 CLS#6:PRINT #6,"Enter date (DDMM) o
r press CLR to exit";:LOCATE 1,ln:PRINT
cleol$;:t=2:l=4:GOSUB 8000:CLS#6:IF LEN(
x1$)=0 OR x1$="0000" THEN 9000
9120 MID$(x0$,4,4)=x1$
9150 PRINT #6,"Enter transaction id. or
press ENTER";:LOCATE 6,ln:PRINT clels$;:
t=2:l=3:GOSUB 8000:CLS#6
9160 MID$(x0$,8,3)=LEFT$(x1$+"    ",3)
9200 PRINT #6,"Enter amount, or CLR to c
orrect";:LOCATE 10,ln:PRINT cleol$;:t=2:
l=9:GOSUB 8000:CLS#6:IF z0=0 THEN 9110
9210 LOCATE 10,ln:PRINT cleol$;USING "##
####.##";z0;
9220 MID$(x0$,11,9)=MID$(STR$(z0)+"
    ",2,9)
9230 LOCATE 20,ln:PRINT cleol$;
9232 IF cats>1 THEN 9238
9234 IF pay THEN x1$=LEFT$(paycat$(1),2)
 ELSE x1$=LEFT$(reccat$(1),2)
9236 GOTO 9250
9238 PRINT #6,"Enter category code, or C
LR to correct.";:t=3:l=2:GOSUB 8000:CLS#
6
9240 IF x1$="" THEN 9200 ELSE x1$=UPPER$
(x1$)
9250 FOR z0=1 TO cats:IF pay THEN x$=pay
cat$(z0) ELSE x$=reccat$(z0)
9255 IF LEFT$(x1$,2)=LEFT$(x$,2) THEN 93
00
9260 NEXT z0 : GOTO 9230
9300 LOCATE 20,ln:PRINT x$
9310 MID$(x0$,20,2)=x1$
9320 trans=trans+1:updated=true:IF pay T
HEN pays=trans:pay$(pays)=x0$ ELSE recs=
trans:rec$(recs)=x0$
9330 IF TRANS=MAXTRANS THEN 9450
9340 IF ln=23 THEN GOSUB 4100:GOTO 9000
9350 ln=ln+1:x3$=MID$(x0$,4,4):GOTO 9110
9450 IF pay THEN x$="All payments alloca
```

```
ted! " ELSE x$="All receipts allocated!
"
9460 CLS#6:PRINT #6,x$;:t=1:l=0:GOSUB 80
00:CLS#6
9499 RETURN
9500 REM del trans
9505 GOSUB 4200
9510 IF pay THEN x$="Payment Deletion" E
LSE x$="Receipt Deletion"
9515 y=1:GOSUB 4050
9520 LOCATE 1,7:x2$=CHR$(242)+CHR$(16)+C
HR$(13)+CHR$(240)+CHR$(241)+"Dd"
9530 PRINT "Account                Date
Method          No.       Amount    Catego
ry"
9540 IF pay THEN trans=pays:meths=paymet
hs:cats=paycats ELSE trans=recs:meths=re
cmeths:cats=reccats
9545 z8=1:x4$="":x5$="":GOSUB 4800
9550 GOSUB 11300:ln=0:ptr=8
9555 LOCATE 77,ptr:PRINT " <=";
9560 CLS #6:PRINT #6,"Use ENTER, CLR or
";LFT$;" to move, ";up$;" or ";down$;" t
o select, D to delete. ";:t=4:l=0:GOSUB
8000:CLS#6
9570 ON z0 GOTO 9580,9999,9600,9700,9700
,9800,9800
9580 z8=z8-16:IF z8<1 THEN z8=1
9585 GOTO 9550
9600 IF z9<= trans THEN z8=z9+1:GOTO 955
0 ELSE 9999
9700 LOCATE 77,ptr:PRINT "   ";
9710 IF z0=4 THEN ptr=ptr-1 ELSE ptr=ptr
+1
9720 IF ptr<8 THEN ptr=8
9730 IF ptr>23 THEN ptr=23
9735 IF ptr-7>(trans-z8+1) THEN ptr=ptr-
1
9740 GOTO 9555
9800 z1=z8+ptr-8
9810 FOR z9=z1 TO trans-1
9820 IF pay THEN 9840
9830 rec$(z9)=rec$(z9+1) : GOTO 9850
```

```
9840 pay$(z9)=pay$(z9+1)
9850 NEXT z9
9860 updated=true:IF pay THEN pays=pays-
1 ELSE recs=recs-1
9870 trans=trans-1
9880 GOTO 9550
9999 GOSUB 4300: RETURN
10000 REM error
10010 PRINT "Error";ERR;"at";ERL
10020 PAPER 0:PEN 1
10030 MODE 1
10999 STOP
11000 REM list trans
11005 GOSUB 4400
11007 IF screen THEN GOSUB 4200
11010 IF pay THEN x$="List of Payments"
ELSE x$="List of Receipts"
11015 GOSUB 4000:y=1:GOSUB 4050
11020 LOCATE 1,7:x2$=CHR$(242)+CHR$(16)+
CHR$(13)
11022 PRINT "Account (ENTER for all) ";:
t=3:l=1:GOSUB 8000
11023 x4$=UPPER$(x1$)
11024 LOCATE 1,9:PRINT "Category (ENTER
for all) ";:t=3:l=2:GOSUB 8000
11025 x5$=UPPER$(x1$)
11027 LOCATE 1,7:PRINT cleosc$;
11030 GOSUB 4500:PRINT #prdev1,"Account
              Date    Method          No.
     Amount    Category"
11040 IF pay THEN trans=pays:meths=payme
ths:cats=paycats ELSE trans=recs:meths=r
ecmeths:cats=reccats
11045 z8=1
11050 GOSUB 11300
11055 IF NOT screen THEN 11100 ELSE ln=0
11060 CLS #6:PRINT #6,"Use ENTER for mor
e, ";lft$;" to go back, or CLR to quit "
;:t=4:l=0:GOSUB 8000:CLS#6
11070 ON z0 GOTO 11080,11250,11100
11080 REM back
11090 z8=z8-16:IF z8<1 THEN z8=1
11095 GOTO 11050
11100 REM forward
```

```
11110 IF z9<= trans THEN z8=z9+1:GOTO 11
050
11250 REM quit
11255 IF NOT screen THEN GOSUB 4600
11260 GOSUB 4300
11299 GOSUB 4800:RETURN
11300 REM list pay or rec
11302 CLS #5
11305 FOR z9=z8 TO trans
11310 IF pay THEN x$=pay$(z9) ELSE x$=re
c$(z9)
11312 IF x4$<>"" THEN IF x4$<>LEFT$(x$,1
) THEN 11420
11313 IF x5$<>"" THEN IF x5$<>RIGHT$(x$,
2) THEN 11420
11315 x0$=LEFT$(x$,1)
11320 FOR z1=1 TO accounts:IF x0$=LEFT$(
acname$(z1),1) THEN 11330
11325 NEXT z1 : GOTO 11335
11330 x0$=acname$(z1)
11335 GOSUB 4500:PRINT #prdev2,x0$;TAB(2
2);MID$(x$,4,2);"/";MID$(x$,6,2);"  ";
11336 x0$=MID$(x$,2,2)
11337 FOR z1=1 TO meths
11340 IF pay THEN meth$=paymeth$(z1) ELS
E meth$=recmeth$(z1)
11345 IF x0$=LEFT$(meth$,2) THEN 11355
11350 NEXT z1 : GOTO 11360
11355 x0$=meth$
11360 PRINT #prdev2,x0$;TAB(45);MID$(x$,
8,3);"   ";
11365 PRINT #prdev2,USING "######.##   "
;VAL(MID$(x$,11,9));
11370 x0$=RIGHT$(x$,2)
11375 FOR z1=1 TO cats
11380 IF pay THEN CT$=paycat$(z1) ELSE C
T$=reccat$(z1)
11385 IF x0$=LEFT$(CT$,2) THEN 11395
11390 NEXT z1:GOTO 11400
11395 x0$=CT$
11400 PRINT #prdev2,x0$
11410 IF ln=lines THEN 11499
11420 NEXT z9
11499 RETURN
11500 REM summ trans
```

```
11502 GOSUB 4400
11510 IF pay THEN x$="Summary of Payment
s" ELSE x$="Summary of Receipts"
11515 GOSUB 4000:y=1:GOSUB 4050
11520 t2=0
11525 IF pay THEN trans=pays:cats=paycat
s ELSE trans=recs:cats=reccats
11530 FOR z9=1 TO cats : z2=0
11535 IF pay THEN x$=paycat$(z9) ELSE x$
=reccat$(z9)
11540 FOR z1=1 TO trans
11545 IF pay THEN x0$=pay$(z1) ELSE x0$=
rec$(z1)
11550 IF RIGHT$(x0$,2)=LEFT$(x$,2) THEN
z2=z2+VAL(MID$(x0$,11,9))
11560 NEXT z1
11565 IF z2=0 THEN 11590
11570 GOSUB 4500:PRINT #prdev2,x$;TAB(20
);
11580 PRINT #prdev2,USING "######.##";z2
 : t2=t2+z2
11590 NEXT z9
11600 GOSUB 4500:PRINT #prdev2,"Total";T
AB(20);
11610 PRINT #prdev2,USING "######.##";t2
11620 GOSUB 4600
11999 GOSUB 4800:RETURN
12000 REM list cat
12005 GOSUB 4400
12010 IF pay THEN x$="Payment Categories
" ELSE x$="Receipt Categories"
12015 GOSUB 4000:y=1:GOSUB 4050
12025 IF pay THEN cats=paycats ELSE cats
=reccats
12030 FOR z0=1 TO cats
12040 IF z0/2=INT(z0/2) THEN x=21 ELSE x
=1:GOSUB 4500
12060 PRINT #prdev2,TAB(x);:IF pay THEN
PRINT #prdev2,paycat$(z0); ELSE PRINT #p
rdev2,reccat$(z0);
12070 NEXT z0
12080 GOSUB 4600
12499 GOSUB 4800:RETURN
12500 REM add cat
12510 IF pay THEN cats=paycats:maxcats=m
```

```
axpaycats:x$="New Payment Categories" EL
SE cats=reccats:maxcats=maxreccats:x$="N
ew Receipt Categories"
12515 GOSUB 4000:y=1:GOSUB 4050
12517 PRINT cleosc$;:ln=6
12520 FOR z0=1 TO cats
12530 IF z0/2=INT(z0/2) THEN x=21 ELSE x
=1:ln=ln+1
12540 LOCATE x,ln : IF pay THEN PRINT pa
ycat$(z0); ELSE PRINT reccat$(z0);
12560 NEXT z0
12570 IF cats=maxcats THEN 12900
12575 IF pay THEN x$="New payment catego
ry " ELSE x$="New receipt category "
12580 LOCATE 1,25:PRINT cleol$+x$;:t=3:l
=15:GOSUB 8000
12590 IF x1$="" THEN 12950
12595 IF LEN(x1$)<2 THEN PRINT bel$;:GOT
O 12580 ELSE x1$=UPPER$(x1$)
12600 FOR z0=1 TO cats
12605 IF pay THEN x0$=paycat$(z0) ELSE x
0$=reccat$(z0)
12607 IF LEFT$(x1$,2)=LEFT$(x0$,2) THEN
12580
12610 NEXT z0
12620 cats=cats+1:updated=true:IF pay TH
EN paycat$(cats)=x1$ ELSE reccat$(cats)=
x1$
12630 GOTO 12517
12900 IF pay THEN x$="Payment categories
 all allocated!" ELSE x$="Receipt catego
ries all allocated!"
12910 LOCATE 1,25:PRINT cleol$+x$;:t=1:l
=0:GOSUB 8000
12950 IF pay THEN paycats=cats ELSE recc
ats=cats
12999 RETURN
13000 REM del cat
13010 IF pay THEN cats=paycats:x$="Payme
nt Category Deletion" ELSE cats=reccats:
x$="Receipt Category Deletion"
13015 GOSUB 4000:y=1:GOSUB 4050
13025 IF cats=0 THEN 13499
13030 FOR z0=1 TO cats
13040 IF z0/2=INT(z0/2) THEN x=21 ELSE x
```

```
=1:GOSUB 4500
13060 PRINT #5,TAB(x);IF pay THEN PRINT
#5,paycat$(z0); ELSE PRINT #5,reccat$(z0
);
13070 NEXT z0
13080 LOCATE 1,25:PRINT "Deleted categor
y ";:t=3:l=2:GOSUB 8000
13090 IF x1$="" THEN 13499 ELSE x1$=UPPE
R$(x1$)
13100 IF LEN(x1$)<>2 THEN 13080
13110 FOR z0=1 TO cats:IF pay THEN x$=pa
ycat$(z0) ELSE x$=reccat$(z0)
13115 IF x1$=LEFT$(x$,2) THEN 13150
13120 NEXT z0
13130 GOTO 13080
13150 FOR z1=z0 TO cats-1
13160 IF pay THEN paycat$(z1)=paycat$(z1
+1) ELSE reccat$(z1)=reccat$(z1+1)
13170 NEXT z1
13180 cats=cats-1:updated=true:IF pay TH
EN paycats=cats ELSE reccats=cats
13190 CLS #5
13200 GOTO 13025
13499 RETURN
13500 REM list meth
13505 GOSUB 4400
13510 IF pay THEN meths=paymeths:x$="Pay
ment Methods" ELSE meths=recmeths:x$="Re
ceipt Methods"
13515 GOSUB 4000:y=1:GOSUB 4050
13530 FOR z0=1 TO meths
13540 IF z0/2=INT(z0/2) THEN x=21 ELSE x
=1:GOSUB 4500
13560 PRINT #prdev2,TAB(x);:IF pay THEN
PRINT #prdev2,paymeth$(z0); ELSE PRINT #
prdev2,recmeth$(z0);
13570 NEXT z0
13580 GOSUB 4600
13999 GOSUB 4800:RETURN
14000 REM add meth
14010 IF PAY THEN meths=paymeths:maxmeth
s=maxpaymeths:X$="Payments" ELSE meths=r
ecmeths:maxmeths=maxrecmeths:x$="Receipt
s"
14012 GOSUB 4000:y=1:GOSUB 4050
```

```
14015 PRINT cleosc$;: ln=6
14020 FOR z0=1 TO meths
14030 IF z0/2=INT(z0/2) THEN x=21 ELSE x
=1:ln=ln+1
14040 LOCATE x,ln : IF pay THEN PRINT pa
ymeth$(z0); ELSE PRINT recmeth$(z0);
14050 NEXT z0
14060 IF meths=maxmeths THEN 14400
14062 IF pay THEN x$="New payment method
 " ELSE x$="New receipt method "
14065 LOCATE 1,25:PRINT cleol$+x$;:t=3:l
=15:GOSUB 8000
14070 IF x1$="" THEN 14499
14075 IF LEN(x1$)<2 THEN PRINT bel$;:GOT
O 14065 ELSE x1$=UPPER$(x1$)
14080 FOR z0=1 TO meths:IF LEFT$(x1$,2)=
LEFT$(x$,2) THEN 14065
14090 NEXT z0
14100 meths=meths+1:updated=true:IF pay
THEN paymeths=meths:paymeth$(paymeths)=x
1$ ELSE recmeths=meths:recmeth$(recmeths
)=x1$
14110 GOTO 14015
14400 CLS#6:PRINT #6,"Methods all alloca
ted!";:t=1:l=0:GOSUB 8000:CLS#6
14499 RETURN
14500 REM del method
14510 IF pay THEN meths=paymeths:x$="Pay
ment Methods" ELSE meths=recmeths:x$="Re
ceipt Methods"
14515 GOSUB 4000:y=1:GOSUB 4050:ln=0:GOS
UB 4800
14525 IF meths=0 THEN 14999
14530 FOR z0=1 TO meths
14540 IF z0/2=INT(z0/2) THEN x=21 ELSE x
=1:GOSUB 4500
14560 PRINT #5,TAB(x);:IF pay THEN PRINT
 #5,paymeth$(z0); ELSE PRINT #5,recmeth$
(z0);
14570 NEXT z0
14580 LOCATE 1,25:PRINT "Deleted method
";:t=3:l=2:GOSUB 8000
14590 IF x1$="" THEN 14999 ELSE x1$=UPPE
R$(x1$)
```

```
14600 IF LEN(x1$)<>2 THEN 14580
14610 FOR z0=1 TO meths:IF pay THEN x$=p
aymeth$(z0) ELSE x$=recmeth$(z0)
14615 IF x1$=LEFT$(x$,2) THEN 14650
14620 NEXT z0
14630 GOTO 14580
14650 FOR z1=z0 TO meths-1
14660 IF pay THEN paymeth$(z1)=paymeth$(
z1+1) ELSE recmeth$(z1)=recmeth$(z1+1)
14670 NEXT z1
14680 meths=meths-1:updated=true:IF pay
THEN paymeths=meths ELSE recmeths=meths
14690 CLS#5
14700 GOTO 14525
14999 RETURN
15000 REM list so
15005 GOSUB 4400
15007 IF screen THEN GOSUB 4200
15010 x$="Standing Orders":y=1:GOSUB 405
0
15020 LOCATE 1,7:GOSUB 4500:PRINT #prdev
1,"Account                DDMM     Amount
   Category"
15030 x2$=CHR$(242)+CHR$(16)+CHR$(13)
15040 z8=1
15050 GOSUB 15300
15055 IF NOT screen THEN 15100 ELSE ln=0
15060 CLS#6:PRINT #6,"Use ENTER for more
, ";lft$;" to go back, or CLR to quit ";
:t=4:i=0:GOSUB 8000:CLS#6
15070 ON z0 GOTO 15080,15250,15100
15080 REM back
15090 z8=z8-16:IF z8<1 THEN z8=1
15095 GOTO 15050
15100 REM forward
15110 IF z9<= sos THEN z8=z9+1:GOTO 1505
0
15250 REM quit
15255 IF NOT screen THEN GOSUB 4600
15260 GOSUB 4300
15299 GOSUB 4800:RETURN
15300 REM list so
15305 CLS #5
15310 FOR z9=z8 TO sos
15315 FOR z1=1 TO accounts:IF LEFT$(so$(
```

```
z9),1)=LEFT$(acname$(z1),1) THEN 15330
15320 NEXT z1 : x0$=LEFT$(so$(z9),1):GOT
O 15335
15330 x0$=acname$(z1)
15335 GOSUB 4500:PRINT #prdev2,x0$;TAB(2
2);MID$(so$(z9),2,2);"/";MID$(so$(z9),4,
2);"  ";
15340 PRINT #prdev2,USING "######.##   "
;VAL(MID$(so$(z9),6,9));
15350 FOR z1=1 TO paycats
15360 IF RIGHT$(so$(z9),2)=LEFT$(paycat$
(z1),2) THEN 15380
15370 NEXT z1:x0$=LEFT$(paycat$(z1),2):G
OTO 15400
15380 x0$=paycat$(z1)
15400 PRINT #prdev2,x0$
15410 IF ln=lines THEN 15499
15420 NEXT z9
15499 RETURN
15500 REM add so
15510 GOSUB 4000:x$="Standing Orders":y=
1:GOSUB 4050
15520 IF sos=maxsos THEN 15950
15530 LOCATE 1,6:PRINT cleol$+"Account "
;:t=3:l=1:GOSUB 8000:IF x1$="" THEN 1599
9 ELSE x1$=UPPER$(x1$)
15540 FOR z0=1 TO accounts:IF LEFT$(x1$,
1)=LEFT$(acname$(z0),1) THEN 15580
15545 NEXT z0: GOTO 15530
15580 x0$=x1$+SPACE$(15)
15590 LOCATE 1,6:PRINT cleol$+acname$(z0
)
15600 LOCATE 1,7:PRINT "DDMM Amount    C
ategory":ln=8
15610 CLS#6:PRINT #6,"Enter date. (Month
 can be blank)";
15612 LOCATE 1,ln:t=2:l=4:GOSUB 8000:CLS
#6:IF LEN(x1$)=0 OR x1$="0000" THEN 1550
0
15615 IF RIGHT$(x1$,2)="00" THEN x1$=LEF
T$(x1$,2)+"  "
15620 MID$(x0$,2,4)=x1$
15630 LOCATE 6,ln:PRINT cleol$;:t=2:l=9:
GOSUB 8000:IF z0=0 THEN 15610
```

```
15640 LOCATE 6,ln:PRINT cleol$;USING "##
####.##";z0;
15645 MID$(x0$,6,9)=MID$(STR$(z0)+"
    ",2,9)
15650 LOCATE 16,ln:t=3:l=2:GOSUB 8000
15660 IF x1$="" THEN 15630 ELSE x1$=UPPE
R$(x1$)
15670 FOR z0=1 TO paycats
15680 IF LEFT$(x1$,2)=LEFT$(paycat$(z0),
2) THEN 15700
15690 NEXT z0 : GOTO 15650
15700 LOCATE 16,ln:PRINT paycat$(z0)
15710 MID$(x0$,15,2)=x1$
15720 sos=sos+1:so$(sos)=x0$:updated=tru
e
15730 IF sos=maxsos THEN 15950
15740 IF ln=23 THEN GOSUB 4100:GOTO 1550
0
15750 ln=ln+1:GOTO 15610
15950 CLS#6:PRINT #6,"All standing order
s allocated! ";:t=1:l=0:GOSUB 8000:CLS#6
15999 RETURN
16000 REM del so
16010 GOSUB 4200
16020 x$="Standing Order Deletion":y=1:G
OSUB 4050
16030 LOCATE 1,7:x2$=CHR$(242)+CHR$(16)+
CHR$(13)+CHR$(240)+CHR$(241)+"Dd"
16040 PRINT "Account                Date
Amount   Category"
16050 z8=1:GOSUB 4800
16060 GOSUB 15300:ln=0:ptr=8
16070 LOCATE 57,ptr:PRINT " <=";
16080 CLS#6:PRINT #6,"Use ENTER, CLR or
";LFT$;" to move, ";up$;" or ";down$;" t
o select, D to delete. ";:t=4:l=0:GOSUB
8000:CLS#6
16090 ON z0 GOTO 16100,16499,16200,16300
,16300,16400,16400
16100 z8=z8-16:IF z8<1 THEN z8=1
16110 GOTO 16060
16200 IF z9< sos THEN z8=z9+1:GOTO 16060
 ELSE 16499
16300 LOCATE 57,ptr:PRINT "   ";
16310 IF z0=4 THEN ptr=ptr-1 ELSE ptr=pt
```

```
r+1
16320 IF ptr<8 THEN ptr=8
16330 IF ptr>23 THEN ptr=23
16340 IF ptr-7>(sos-z8+1) THEN ptr=ptr-1
16350 GOTO 16070
16400 z1=z8+ptr-8
16410 FOR z9=z1 TO sos-1
16420 so$(z9)=so$(z9+1)
16430 NEXT z9
16440 sos=sos-1:updated=true
16450 GOTO 16060
16499 GOSUB 4300 : RETURN
16500 REM con so
16510 GOSUB 4200
16520 GOSUB 4000:x$="Standing Order Conf
irmation":y=1:GOSUB 4050
16530 LOCATE 1,7:x2$=CHR$(242)+CHR$(16)+
CHR$(13)+CHR$(240)+CHR$(241)+"Cc"
16540 PRINT "Account                Date
Amount   Category              Month  Con
firmed"
16545 IF pays=maxpays THEN 16974
16550 z8=1:GOSUB 4800
16560 GOSUB 15300:ln=0:ptr=8
16570 LOCATE 57,ptr:PRINT cleol$;" <=";
16580 CLS#6:PRINT #6,"Use ENTER, CLR or
";LFT$;" to move, ";up$;" or ";down$;" t
o select, C to confirm. ";:t=4:l=0:GOSUB
 8000:CLS#6
16590 ON z0 GOTO 16600,16999,16700,16800
,16800,16900,16900
16600 z8=z8-16:IF z8<1 THEN z8=1
16610 GOTO 16560
16700 IF z9< sos THEN z8=z9+1:GOTO 16560
 ELSE 16999
16800 LOCATE 57,ptr:PRINT "   ";
16810 IF z0=4 THEN ptr=ptr-1 ELSE ptr=pt
r+1
16820 IF ptr<8 THEN ptr=8
16830 IF ptr>23 THEN ptr=23
16840 IF ptr-7>(sos-z8+1) THEN ptr=ptr-1
16850 GOTO 16570
16900 z1=z8+ptr-8:LOCATE 58,ptr:PRINT cl
eol$;
16910 x0$=so$(z1):x$=STRING$(21," "):MID
```

```
$(x$,1,3)=LEFT$(x0$,1)+"ST"
16915 x4$=MID$(x0$,2,4):IF RIGHT$(x4$,2)
<>"  " THEN 16930
16917 PRINT #6,"Enter month ";
16920 LOCATE 58,ptr:t=2:l=2:GOSUB 8000:C
LS#6:IF x1$="" THEN 16570
16925 MID$(x4$,3,2)=LEFT$(x1$+"00",2)
16930 MID$(x$,4,4)=x4$
16932 PRINT #6,"Confirm amount to be pai
d ";
16935 LOCATE 65,ptr:t=2:l=9:x3$=MID$(x0$
,6,9):GOSUB 8000:CLS#6
16940 IF z0=0 THEN LOCATE 58,ptr:PRINT c
leol$;:GOTO 16570
16950 MID$(x$,11,9)=MID$(STR$(z0)+"
   ",2,9)
16960 MID$(x$,20,2)=RIGHT$(x0$,2)
16970 pays=pays+1:updated=true:pay$(pays
)=x$
16972 IF pays<maxpays THEN 16570
16974 CLS#6:PRINT #6,"All payments alloc
ated!";:t=1:l=0:GOSUB 8000:CLS#6
16999 GOSUB 4300:RETURN
17000 REM list xfers
17010 GOSUB 4400
17015 IF screen THEN GOSUB 4200
17020 x$="Transfers between Accounts":y=
1:GOSUB 4050
17030 LOCATE 1,7:GOSUB 4500:PRINT #prdev
1,"From Account          Date  No.      A
mount     To Account"
17040 x2$=CHR$(242)+CHR$(16)+CHR$(13)
17050 z8=1
17055 IF NOT screen THEN 17100
17060 GOSUB 17300
17070 CLS#6:PRINT #6,"Use ENTER for more
, ";1ft$;" to go back, or CLR to quit ";
:t=4:l=0:GOSUB 8000:CLS#6
17080 ON z0 GOTO 17085,17250,17100
17085 REM back
17090 z8=z8-16:IF z8<1 THEN z8=1
17095 GOTO 17060
17100 REM forward
17110 IF z9<= xfers THEN z8=z9+1:GOTO 17
060
```

```
17120 IF NOT screen THEN GOSUB 4600
17250 REM quit
17299 GOSUB 4300:GOSUB 4800:RETURN
17300 REM list xfers
17305 CLS#5
17310 FOR z9=z8 TO xfers
17312 x$=xfer$(z9)
17315 FOR z1=1 TO accounts:IF LEFT$(x$,1
)=LEFT$(acname$(z1),1) THEN 17330
17320 NEXT z1 : xo$=LEFT$(x$,1)+"??":GOT
O 17335
17330 x0$=acname$(z1)
17335 GOSUB 4500:PRINT #prdev2,x0$;TAB(2
2);MID$(x$,2,2);"/";MID$(x$,4,2);"  ";
17340 PRINT #prdev2,MID$(x$,6,3);"  " US
ING "######.##     ";VAL(MID$(x$,9,9));
17345 FOR z1 =1 TO accounts
17350 IF RIGHT$(x$,1)=LEFT$(acname$(z1),
1) THEN 17365
17360 NEXT z1:x0$=RIGHT$(xfer$(z9),1)+"?
?":GOTO 17370
17365 x0$=acname$(z1)
17370 PRINT #prdev2,x0$
17400 IF ln=lines THEN 17499
17410 NEXT z9
17499 RETURN
17500 REM add xfer
17510 GOSUB 4000:x$="Transfer Addition":
y=1:GOSUB 4050
17520 IF xfers=maxxfers THEN 17950
17530 LOCATE 1,6:PRINT cleosc$+"From Acc
ount ";:t=3:l=1:GOSUB 8000:IF x1$="" THE
N 17999 ELSE x1$=UPPER$(x1$)
17540 FOR z0=1 TO accounts:IF x1$=LEFT$(
acname$(z0),1) THEN 17580
17545 NEXT z0: GOTO 17530
17580 x0$=x1$+SPACE$(17)
17590 LOCATE 1,6:PRINT cleol$;"From Acco
unt: ";acname$(z0)
17600 PRINT "DDMM No. Amount     To Accou
nt":ln=8
17610 LOCATE 1,25:PRINT cleol$+"Enter da
te. (DDMM) ";
```

```
17612 LOCATE 1,ln:PRINT cleol$;:t=2:l=4:
GOSUB 8000:IF LEN(x1$)=0 THEN 17530
17620 MID$(x0$,2,4)=x1$
17625 LOCATE 1,25:PRINT cleol$;"Enter op
tional transaction no.";
17630 LOCATE 6,ln:t=3:l=3:GOSUB 8000
17635 MID$(x0$,6,3)=x1$
17640 LOCATE 1,25:PRINT cleol$;"Enter am
ount";
17645 LOCATE 10,ln:PRINT cleol$;:t=2:l=9
:GOSUB 8000
17650 IF z0=0 THEN 17610
17660 MID$(x0$,9,9)=x1$
17670 LOCATE 1,25:PRINT "Enter Account c
ode";
17680 LOCATE 21,ln:t=3:l=1:GOSUB 8000
17690 IF x1$="" THEN 17640 ELSE x1$=UPPE
R$(x1$)
17695 IF x1$=LEFT$(x0$,1) THEN 17680
17700 FOR z1=1 TO accounts
17710 IF x1$=LEFT$(acname$(z1),1) THEN 1
7730
17720 NEXT z1:GOTO 17680
17730 MID$(x0$,18,1)=x1$
17740 LOCATE 21,ln:PRINT acname$(z1)
17750 xfers=xfers+1:updated=true:xfer$(x
fers)=x0$
17760 IF xfers=maxxfers THEN 17950
17770 IF ln=23 THEN GOSUB 4100:GOTO 1753
0
17780 ln=ln+1:GOTO 17610
17950 CLS#6:PRINT #6,"All transfers allo
cated! ";:t=1:l=0:GOSUB 8000:CLS#6
17999 RETURN
18000 REM del xfer
18010 GOSUB 4200
18020 x$="Transfer Deletion":y=1:GOSUB 4
050
18030 LOCATE 1,7:x2$=CHR$(242)+CHR$(16)+
CHR$(13)+CHR$(240)+CHR$(241)+"Dd"
18040 PRINT "From Account           Date
 No.      Amount     To Account"
18050 z8=1:GOSUB 4800
18060 GOSUB 17300:ln=0:ptr=8
18070 LOCATE 67,ptr:PRINT " <=";
```

```
18080 CLS#6:PRINT #6,"Use ENTER, CLR or
";LFT$;" to move, ";up$;" or ";down$;" t
o select, D to delete. ";:t=4:l=0:GOSUB
8000:CLS#6
18090 ON z0 GOTO 18100,18499,18200,18300
,18300,18400,18400
18100 z8=z8-16:IF z8<1 THEN z8=1
18110 GOTO 18060
18200 IF z9< xfers THEN z8=z9+1:GOTO 180
60 ELSE 18499
18300 LOCATE 67,ptr:PRINT "   ";
18310 IF z0=4 THEN ptr=ptr-1 ELSE ptr=pt
r+1
18320 IF ptr<8 THEN ptr=8
18330 IF ptr>23 THEN ptr=23
18340 IF ptr-7>(xfers-z8+1) THEN ptr=ptr
-1
18350 GOTO 18070
18400 z1=z8+ptr-8
18410 FOR z9=z1 TO xfers-1
18420 xfer$(z9)=xfer$(z9+1)
18430 NEXT z9
18440 xfers=xfers-1:updated=true
18450 GOTO 18060
18499 GOSUB 4300 : RETURN
18500 REM new period
18510 IF NOT updated THEN 18600
18520 CLS#6:PRINT #6,"Save old file befo
re transfering.";:t=1:l=0:GOSUB 8000:CLS
#6:GOTO 18999
18600 GOSUB 7500
18610 FOR z0=1 TO accounts:balo(z0)=balc
(z0):xfers=0:pays=0:recs=0:balc(z0)=0:NE
XT z0
18620 f$="new file":GOSUB 2500
18999 RETURN
19000 REM adj o/b
19010 GOSUB 4000
19020 LOCATE 1,6:PRINT cleosc$;:CLS#6:PR
INT #6,"Account Code ";:t=3:l=1:GOSUB 80
00:CLS#6
19030 IF x1$="" THEN 19499 ELSE x1$=UPPE
R$(x1$)
19040 FOR z9=1 TO accounts
19050 IF x1$=LEFT$(acname$(z9),1) THEN 1
```

```
9070
19060 NEXT z9:GOTO 19020
19070 LOCATE 1,6:PRINT cleosc$+acname$(z
9);TAB(22); USING "######.##";balo(z9)
19080 CLS#6:PRINT #6,"New Opening Balanc
e";:LOCATE 22,7:t=2:l=9:GOSUB 8000:CLS#6
19090 IF t=-5 THEN 19499 ELSE balo(z9)=z
0:updated=true:GOTO 19020
19499 RETURN
```

INDEX

INDEX

This superb book is designed to let every CPC 464 user, at what ever level, get the most from his computer. After an introductory Section on the special Basic features, the book looks in depth at the excellent sound and graphic facilities including:

- Animation
- Windows
- Character sets
- Multitasking
- 3 Voice Tunes
- M/C routines from Basic
- Use of Zen
- Use of O/S
- Sample programs

7.95net

Published by

Kuma AMSTRAD CPC464 software

Kuma Computers Ltd., Pangbourne, Berkshire, England
Telephone 07357-4335 Telex 849462 TELFAC

Document numérisé avec amour par

AMSTRAD CPC

MÉMOIRE ÉCRITE

https://acpc.me/