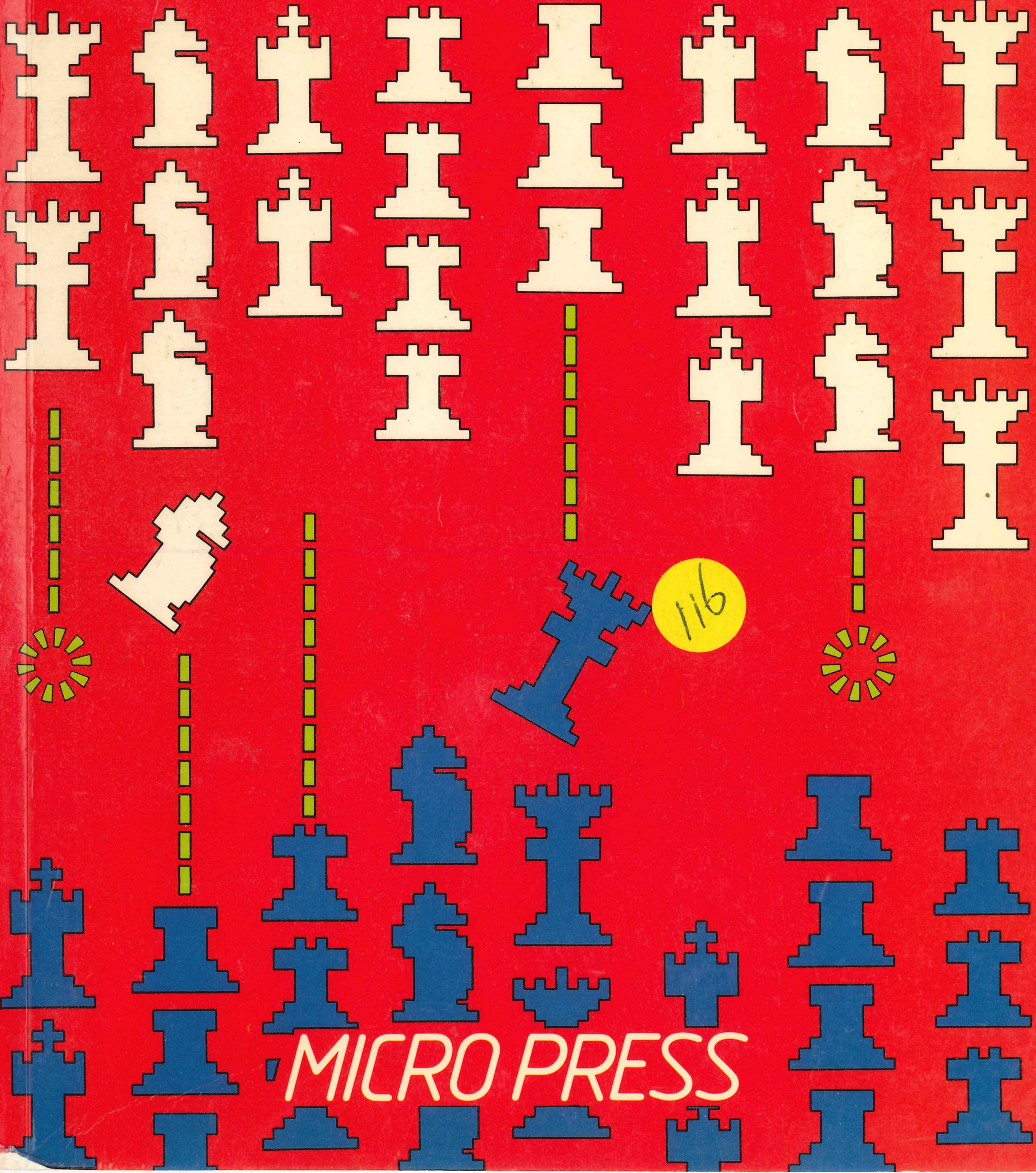
GAMES AND GRAPHICS PROGRAMMING ON THE AMSTRAD COMPUTERS CPC 464, 664 and 6128

Steve Colwill



MICRO PRESS

234 x 155

Games and Graphics Programming on the Amstrad CPC 464, 664 and 6128

Games and Graphics Programming on the Amstrad CPC 464, 664 and 6128

Steve Colwill

MICRO PRESS

First published in 1985 in the United Kingdom by
Micro Press
Castle House, 27 London Road
Tunbridge Wells, Kent

British Library Cataloguing in Publication Data

Colwill, S.W.
Games programming on the
Amstrad CPC 464, 664 and 6128.
1. Amstrad CPC 664 (Computer) — Programming
2. Amstrad CPC 464 (Computer) — Programming
3. Amstrad CPC 6128 (Computer) — Programming
I. Title
001.64'2 QA76.8.A4/

ISBN 0-7447-0032-9

Typeset by MC Typeset, Chatham, Kent
Printed by Mackays of Chatham Ltd

Contents

To Christine

Foreword

This book is intended for people who have some knowledge of BASIC and wish to become more familiar with the Amstrad CPC 664/464. Its aim is to introduce the graphics capabilities of the Amstrad CPC 664/464 and demonstrate how to use them. There are a number of shorter program listings within the book that illustrate the ideas discussed within the various chapters. Also included are programming utilities to help you design your own graphics characters and experiment with the sound capabilities. During the course of the book, in sections at the end of each chapter, a long graphics game is built up in easy-to-follow stages. As the program builds up each new part added can be verified by running the program in its current state. It is hoped that by typing in the program in this way, many of the problems associated with typing in long program listings can be overcome, typing errors being much easier to find. Throughout the book, programming tips and hints are given to help you improve your programming techniques using the facilities of Amstrad CPC range BASIC.

A problem that many people have after mastering the basics of BASIC is that of putting together a large program without getting lost in a tangle of program lines. One of the main aims of the book is to help in the design of long programs by looking at program structure and the way in which a large program can be practically constructed. Programming graphics games can be fun, it is also an excellent method of teaching programming skills as it is easy to see the effects of the program. Games programming is enjoyable but more, it can help in understanding BASIC and the computer.

Steve Colwill
March 1985

Chapter One

Basic ideas

In this chapter:

Program structure and structure diagrams

Blocking structures and the use of flags
`WHILE...WEND`, `FOR...NEXT` and `GOSUB ...RETURN`

Binary and hexadecimal
`BIN$`, `HEX$`, `STR$`, `&` and `&X`

Logical operators and their uses
`AND`, `OR` and `XOR`

An introduction to the game 'Stranded'

Programming computers, either for fun or for business, is as much an art as it is a science. To many people the proof of the pudding is whether a program does what the programmer intended it to do, and obviously a program that does not work correctly is not much use to anyone! There are other factors, however, that should be considered: How readable is the finished program text? Can somebody other than the programmer follow the program? This is quite important if the program is to be read (and hopefully understood) by another person, but equally, if you write a long and complicated piece of code, you may wish to come back to it in 6 weeks' or 6 months' time to make some changes. It can be very frustrating to come back to a program you wrote a while ago and find that it takes you several hours work to understand it again. If a program is written clearly, then these problems can be much reduced. Long programs are also easier to write and debug. For all these reasons, this first chapter is devoted to looking at methods of improving your programming style.

Program structure

Whenever a programmer is faced with a programming problem he (or she) can adopt one of two methods: he can start typing in code straight away at the keyboard, or he can sit down and try to think out the main jobs that the program has to do, before starting. One of the nicest (and the worst!) aspects of BASIC is that the structure of the language allows you to start typing in your program without having thought it out in advance. Most BASIC programmers have probably adopted this approach at one time or another. Almost inevitably, unless the program is very simple, you end up in a corner and have to use a liberal sprinkling of GOTOs to get yourself out of that corner. A program, not unlike the following extreme example may result:

```
10 PRINT"WHAT A SILLY WAY"
20 GOTO 50
30 GOTO 60
40 PRINT"DIFFICULT TO FOLLOW":END
50 PRINT"TO WRITE A PROGRAM.":GOTO 30
60 PRINT"IT MAKES IT TERRIBLY":GOTO 40
```

Planning out on paper, or in your mind, what you want to do before you start can avoid tortuous structures like the one above. This does not necessarily mean planning out the program in minute detail, but identifying the major tasks, considering the jobs that go to make up each of the major tasks and so on. This approach is essentially one of thinking out the main building blocks of the program and then assembling them to make the complete program. The following program is written in a very structured way, and is a simple graphics program to doodle on the screen.

```
1000 REM **** doodle program ****
1010 GOSUB 2000:REM set up routine
1020 REM ** main loop **
1030 WHILE INKEY$=""
1040 GOSUB 5000:REM draw a line
1050 WEND
1060 END
1070 :
```

```
2000 REM **** set up s/r ****
2010 MODE 0
2020 BORDER 0
2030 GOSUB 3000:REM clear screen
2040 GOSUB 4000:REM move to start
2050 GOSUB 6000:REM set up timers
2060 RETURN
2070 :
3000 REM **** clear screen s/r ****
3010 CLS
3020 RETURN
3030 :
4000 REM **** move to start position s/r
 ****
4010 MOVE 300,200
4020 pencol=INT(RND(1)*15)+1
4030 inkcol=INT(RND(1)*27)+1
4040 INK pencol,inkcol
4050 RETURN
4060 :
5000 REM **** random draw s/r ****
5010 x=10-INT(RND(1)*20)
5020 y=10-INT(RND(1)*20)
5030 DRAWR x,y,pencol
5040 RETURN
5050 :
6000 REM **** set up interrupt timers s/
r ****
6010 EVERY 200,0 GOSUB 4000:REM set star
t
6020 EVERY 2000,1 GOSUB 3000:REM clear s
creen
6030 RETURN
```

The actual details of how this program works needn't worry us for the moment, we are more concerned with the way in which it is written. Each main task in the program has been isolated and arranged in its own short block of code as a subroutine that can be called. The idea of the program is to doodle out from the middle of the screen randomly, using randomly selected colours. The first task of the program is to select the screen mode and border colour, clear the screen and set up the Amstrad CPC range interrupt timers (much more of all these later!). The subroutine at line 2000 deals with all these tasks. Some are done directly by this subroutine, but others are done by calling other subroutines, such as the clear screen subroutine. Having completed all the preparation then the program goes into a loop that doodles randomly away from the centre until a key is pressed. Occasionally these doodles will be interrupted to either move back to the centre of the screen, or less frequently, to clear it. These two features of the program are controlled by BASIC interrupts.

The main lesson to be drawn from looking at the way this program is written is that, having identified the main jobs that have to be done, the BASIC code needed to carry out these jobs can be held as separate blocks in the form of subroutines. Whenever a GOSUB is used, a REM statement follows it to say what the subroutine will do, making the program easier to read. REMs are also used as titles for each subroutine, describing its purpose. Coming back in 6 months' time to read and follow the flow of a program written like this has to be easier than following a series of GOTOs!

A convenient way to map out a program structure is to use a structure diagram. Such diagrams do not show the precise flow of control but show how the various building blocks of the program fit together. A structure diagram for the 'doodle' program looks like this:

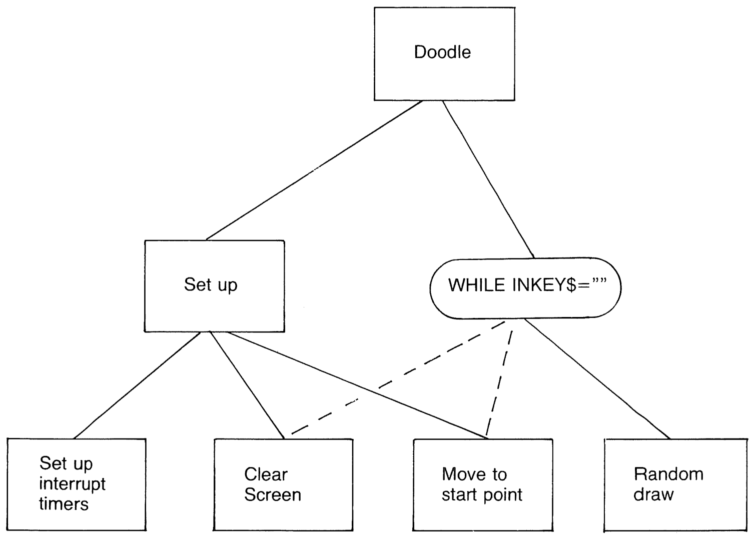


Figure 1.1

The diagram is almost self-explanatory. Each task is drawn as a box, larger tasks appearing higher in the diagram. The sausage-shaped box is used to indicate that some loop is used. An idea of my own is to use dotted lines to show subroutines that interrupt other routines, the dotted lines here showing that 'clear screen' and 'move to start point' interrupt the WHILE...WEND loop every so often.

This example program is highly structured. Slightly more initial effort is required to produce such a program and if you have not tried to write programs in this way before, the extra effort may, on first glance, not seem worth while. It does not take long, however, for this method to become second nature, and the long-term improvement in your analytical programming skills will make the extra effort well worth while. Of course, there is a large gap between the first example and the second, and there are degrees of structuring. Given the limitations of BASIC in terms of the ease of structuring, it is not always possible to avoid the use of GOTO, the guiding principle should be to minimise its use.

Block structures and the use of flags

Adopting a structured approach to BASIC programming involves the sensible use of the structures that Locomotive BASIC gives us. These are FOR...NEXT, WHILE...WEND and GOSUB...RETURN. All the advice given on structuring so far has not been obligatory, but it is important that the following rules with regard to these three structures are obeyed, if you are going to keep your Amstrad CPC 664/464 happy. All of these structures have a definite entry point (FOR, WHILE or the first line number used in the subroutine call) and a definite exit point (NEXT, WEND or RETURN). Never write program sections like these:

```
10    FOR I= 1 TO 10
20    X=RND(1)*40
30    IF X>30 THEN 100
40    NEXT 1
```

```
5     a$=""
10    WHILE a$=""
30    a$=INKEY$
30    X=RND(1)*40
40    IF X>30 THEN 100
50    WEND

10    GOSUB 1000
20    etc
900   END
1000  REM ** SUBROUTINE **
1010  X=RND(1)*40
1020  IF X>30 THEN 100
1030  RETURN
```

The fault common to all of these sections of code is that when the terminating condition is reached (the reason that the program moves out of the structure) the program does not pass through the proper exit point, but jumps out using GOTO. This is not just bad style but, if repeated often enough, will cause your computer to hang up. The reason for this is rooted in the way that the Amstrad CPC range (and many other makes of computers) operating system works. Each of these structures has the ability to 'remember' where it started from. For example, when the computer reaches NEXT, it knows which line to loop back to for the next FOR statement. It remembers because when the computer meets the FOR statement it stores away its position in the program in a special area of memory called the stack. When NEXT is reached, the operating system pulls this information back off the top of the stack, so that it can find its way back to the FOR statement. After repeating the FOR...NEXT loop a given number of times the computer must come to NEXT for the last time to pull the information off the stack and leave it clear for future use. The problem with the above example is that the program may never reach that final NEXT, because if X is greater than 30 then the loop will be jumped out of, and the positional information about FOR will not be cleared from the stack by going to the final NEXT. Do this a few times and the stack will rapidly fill up and cause the whole system to crash. WHILE...WEND and GOSUB...RETURN work in a similar way.

To get around this problem we can still test for conditions within a loop or subroutine as above, but leave the branching part until we have exited the structure correctly, by passing through `NEXT`, `WEND` or `RETURN`. To do this we use a flag to show whether the condition tested for was true. We can then test the value of the flag to make our jump after leaving the loop. The `FOR...NEXT` loop example can be handled as follows:

```
5  flag=0
10 FOR I=1 TO 10
20 X=RND(1)*40
30 IF X>30 THEN flag=1  :  I=10
40 NEXT I
50 IF flag=1 THEN 100
```

In this example the value of a variable, `flag`, is initially set to 0. If at any stage during the looping process the value of X exceeds 30 (the condition tested for) then `flag` is set to 1. In addition the loop counter I is set to the upper limit of the loop, namely 10. On meeting `NEXT` with I set to 10, the operating system will think that the looping process is finished and pass on to the next line. At this line the value of `flag` can be tested and a branch made accordingly. If we wanted to know for some reason the value of I, when X first exceeded 30 then we would need to amend line `30` as follows:

```
30 IF X>30 THEN flag=1:count=I  :  I=10
```

Here the current value of I is stored in another variable, `count`, before exiting the loop. If X>30 on the first pass through the loop then we do not waste time by continuing to execute the loop a further nine times using this method.

`WHILE...WEND` and `GOSUB...RETURN` structures can be dealt with in a similar way.

```
5  flag=0:a$=""
10 WHILE a$=""
20 a$=INKEY$
30 X=RND(1)*40
40 IF X>30 THEN flag=1  :  a$="dummy"
50 WEND
60 IF flag=1 THEN 100
```

```
10   GOSUB 1000
20   IF flag=1 THEN 100
30  etc
900  END
1000 REM SUBROUTINE
1010 flag=0
1020 X=RND(1)*40
1030 IF X>30 THEN flag=1
1040 RETURN
```

Some readers may not be familiar with the `WHILE...WEND` structure used by Amstrad CPC 664/464's Locomotive BASIC. This is an additional looping structure, not normally found in BASIC, although it does bear some relation to BBC BASIC's `REPEAT...UNTIL` structure. The `WHILE...WEND` structure allows the section of code between the `WHILE` and the `WEND` statements to be repeated until the condition stated as part of the `WHILE` command becomes false. In the above example, the `WHILE...WEND` loop will continue until `a$` becomes something other than the null string, `""`. In this case the loop will be terminated by the press of a key on the keyboard. Notice how the loop can be artificially terminated by setting `a$` to `"dummy"` within the loop. It is worth noting that if the condition stated at `WHILE` is false then the code that follows it will not be done, but control will pass directly to the code following `WEND`. If the condition is not true when `WHILE` is first met then the code inside the `WHILE...WEND` loop will never be used, as the terminating condition is tested before the code is done. For this reason it is often a good idea to ensure that the terminating condition is true before entering the loop structure. That is the purpose of setting `a$=""` in line 5 of the `WHILE...WEND` example.

The binary and hexadecimal systems

Just as humans work best using the decimal system (base 10), computers prefer to use the binary system (base 2) for their internal workings. Most of the time, the fact that the computer is fundamentally working in binary is concealed from the BASIC programmer by the computer's BASIC interpreter and operating system. These two programs, that are always present in the machine, carry out all the necessary conversions from decimal to binary, before actually using the values produced. There are, however, occasionally times when a knowledge of the binary system is useful to the BASIC programmer. For those readers who wish to progress to machine-code programming an understanding of binary is essential. A brief maths lesson is therefore in order for those who do not already know what binary is and how to carry out conversions between binary and decimal.

In the decimal system we use ten different numbers or digits, 0, 1, 2, 3, . . . , 8 and 9, and we create larger numbers by arranging these numbers in columns. For example, the number 4283 could be written with these column headings:

Th	H	T	U
4	2	8	3

In the spoken form of this word the value of each column is indicated: 'four thousand, two hundred and eighty-three'. The column headings are based on the number ten.

Units	1	=	1
Tens	1×10	=	10
Hundreds	1×10×10	=	100
Thousands	1×10×10×10	=	1000
etc.			

To obtain the value of each new column heading towards the left of the number we simply multiply the value of the preceding column by 10.

The binary system works in a similar way, but only uses two digits, 0 and 1. Here each new column heading is found by multiplying the preceding one by 2, so column headings for binary are:

Units	1	=	1
Twos	1×2	=	2
Fours	1×2×2	=	4
Eights	1×2×2×2	=	8
Sixteens	1×2×2×2×2	=	16
Thirty-twos	1×2×2×2×2×2	=	32
Sixty-fours	1×2×2×2×2×2×2	=	64
One-two-eights	1×2×2×2×2×2×2×2	=	128
etc.			

The binary number 11010001 can be written with its column headings like this:

128	64	32	16	8	4	2	U
1	1	0	1	0	0	0	1

Its decimal equivalent is easily found by adding up the column headings where a 1 appears as follows:

1×128	=	128
1×64	=	64
1×16	=	16
1×U	=	1
Total	=	209

Converting from decimal to binary can be done by repeated division by 2, using the remainders to make up the binary number.

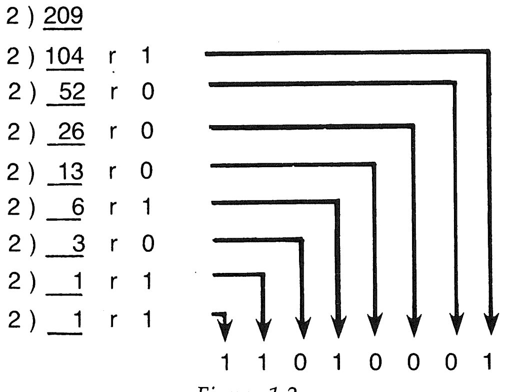


Figure 1.2

Most binary numbers that you will ever have to deal with on the Amstrad CPC range are either eight binary digits (or one byte), like the one above, or 16 binary digits; the term 'binary digit' being shortened to 'bit'. As you can imagine, a group of 16 ones and zeros is rather difficult to take in, so often binary numbers are written in as hexadecimal numbers, or in 'hex'. This system works by taking groups of four digits together and replacing them with a single digit, according to this table:

Binary pattern	Hex digit
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Notice that after nine we run out of single digits to use, the first six letters of the alphabet are therefore used instead. The number 209, or 11010001, is therefore &D1 in hex; the ampersand sign (&) being used to indicate the fact that the number is in hex.

Locomotive BASIC provides several facilities to help with conversions between decimal, binary and hexadecimal.

`BIN$` and `HEX$` allow you to let the computer do the hard work of converting from decimal to binary or hexadecimal.

`PRINT BIN$(209)` will give `11010001`
`PRINT HEX$(209)` will give `D1`

The number of digits in the final conversion can also be specified by adding a second number inside the bracket:

`PRINT BIN$(209,10)` will give `0011010001`
`PRINT HEX$(209,4)` will give `00D1`

These conversions can be held and manipulated as string variables. Conversion from hex or binary back to decimal can be done in two ways: firstly, by conversion to a decimal number,

`A=&X11010001` or `PRINT &X1101001`
`A=&D1` or `PRINT &D1`

or, secondly, as a decimal string variable,

```
A$=STR$(&X11010001)
A$=STR$(&D1)
```

The Logical operators AND, OR and XOR

The signs we use in arithmetic, +, −, ×, ÷, etc., are known as arithmetic operators because they do some kind of arithmetic operation to the numbers on either side of the operator (the numbers are known as operands). Just as there are set rules for the arithmetic operators (everyone had to learn their times tables) so there are special rules that govern a different set of operators that are peculiar to logic and computers, the logical operators. You may well have seen expressions like this in computer programs:

```
X=39 AND 52
```

Here AND does not mean 39 'plus' 54 but has a special purpose in computer applications. Try typing in PRINT 39 AND 52. The result is rather curious: 36. To understand how AND works we must look at the binary equivalent of 39 and 52. As eight-bit binary numbers 39 and 52 are as follows:

			128	64	32	16	8	4	2	U
	39	=	0	0	1	0	0	1	1	1
AND	52	=	0	0	1	1	0	1	0	0
	36	=	0	0	1	0	0	1	0	0

If you look at each column of bits in turn you may begin to understand how the value of each bit in the answer is arrived at. The only place where a bit in the answer is 1 is if *both* bits above it are 1. In other words, if the bit in the first number is 1 *and* the bit in the second number is 1 then the result will be 1.

If you think about it, there are four possible combinations of a pair of bits. If for each combination we write down the result of doing an AND, we get a table known as a truth table:

Bit pair	AND
00	0
01	0
10	0
11	1

Try typing PRINT 39 OR 52. The result is 55. Again to find out how OR works we have to look at the bit patterns:

			128	64	32	16	8	4	2	U
	39	=	0	0	1	0	0	1	1	1
OR	52	=	0	0	1	1	0	1	0	0
	55	=	0	0	1	1	0	1	1	1

Again by looking at each column in turn we can work out what the rule for OR is. Rather than both bits in the operands having to be 1 to make the answer bit 1, one, *or* the other, *or* both of the operand bits have to be 1 to make the answer bit 1. We can summarise this in a truth table for OR:

Bit pair	OR
00	0
01	1
10	1
11	1

In everyday English the word 'or' can, in fact, have two meanings. The first meaning is like the logical OR we have just looked at. 'If Jack *or* Jill can go, I will go to the match.' In this case the 'or' means if Jack *or* Jill *or* both people can go, than I will go to the match. In other words the possibility of both people being able to go is included. This is the so-called 'inclusive or'. Another use of the word 'or' exists in everyday English. You can be tall *or* short, or rich *or* poor, but you can't be both. Here the possibility of both things occurring is excluded. This is an example of the use of the 'exclusive or' often abbreviated to XOR. The result of 39 XOR 52 is 19. Try drawing out the binary patterns for these numbers again and ensure that the following truth table for XOR is true:

Bit pair	XOR
00	0
01	1
10	1
11	0

Later we shall see how these three logical operators AND, OR and XOR can be used to produce special high-resolution plotting effects, but let us now look at their uses in normal BASIC programs. The logical operators are normally used for two purposes. The first, and probably more familiar purpose, is their use within condional `IF...THEN` type statements. We can use statements like this:

```
IF X<3 AND Y>2 THEN PRINT X,Y
```

Here we have two statements `X<3` and `Y>2`. Both statements can be mathematically true or false. If we think of a statement that is *true* as being represented by 1 and a statement that is *false* as being represented by 0 then we can see how the logical `AND` can be applied. The BASIC command given above is really saying:

```
IF<statement 1 is true> AND <statement 2 is
true> THEN <action>
```

In other words, the action will take place if, and only if, both statements are true. This is obviously the same as:

`TRUE AND TRUE = TRUE` or `1 AND 1 = 1`

We can see the effect of using such a statement by using an example of two `FOR...NEXT` loops:

```
    10 FOR X= 1 TO 4
    20 FOR Y= 1 TO 4
    30 IF X<3 AND Y>2 THEN PRINT X,Y
    40 NEXT Y,X
```

The output from this program is, predictably:

```
1 3
1 4
2 3
2 4
```

Replacing line `30` by `IF X<3 OR Y>2 THEN PRINT X,Y` will produce these results:

```
1 1
1 2
1 3
1 4
2 1
2 2
2 3
```

```
2 4
3 3
3 4
4 3
4 4
```

Logical operators can be placed together on the same line such as:

```
30 IF X=1 OR X=2 AND Y=3 THEN PRINT X,Y
```

Here the result will be:

```
1 1
1 2
1 3
1 4
2 3
```

Just as in an arithmetic expression there is an order in which operations like multiply and add are done, e.g. in 3 + 4 × 7 then '×' is done before the '+', there is an order of precedence with logical operators. From the output produced above it seems as though the computer is really thinking of the statement as:

```
IF X=1 OR (X=2 AND Y=3) THEN PRINT X,Y
```

where the brackets indicate the operation to be done first. In fact running the program with these brackets in place will produce the same results. If no brackets exist, then `AND` is done before `OR` (just as '×' was done before '+' in the arithmetic example). If we want to force the computer to do `OR` before `AND` we must use brackets. Changing line `30` to `IF (X=1 OR X=2) AND Y=3` will produce the results:

```
1 3
2 3
```

Logical expressions can also be used with the `WHILE` statement as in this example:

```
5     flag1=0:flag2=0
10    WHILE flag1=0 AND flag2=0
20    GOSUB 1000:REM SELECT RANDOM NUMBER
30    WEND
```

```
40   If flag1=1 THEN PRINT"Greater than 30"
ELSE PRINT"Less than 10"
50   END
1000 REM **** SELECT RANDOM NUMBER ****
1010 flag1=0:flag2=0
1020 X=RND(1)*40
1030 IF X>30 THEN flag1=1
1040 IF X<10 THEN flag2=1
1050 RETURN
```

This short program uses a subroutine that selects a random number before `RETURN`ing to the main program. The subroutine is called from within a loop. Two flags are used to signal if the number selected is greater than 30 or if it is less than 10. The point of the example is to show the use of `AND` in the `WHILE` statement. This loop will only go on repeating until one or other of the two flags is set to 1. Translated into English line `10` means 'keep doing the loop while both flags are zero'.

The second use of `AND` and `OR` is to affect single bits within a byte, without altering the values of other bits within the byte. As the Amstrad CPC range's Locomotive BASIC does such a good job of protecting the user from the fundamental binary structure of the machine, by providing an excellent version of BASIC, the occasions on which this is necessary are rare, but there are times when an advanced BASIC programmer will want to start `POKE`ing and `PEEK`ing around in the machine's memory when this technique is useful, as well as proving indispensable to the machine-code programmer. Let us say that we want to set bit 2 of a special byte in memory to 1, without disturbing the other bits. The initial contents of this memory location might be 178:

Bit no.	7	6	5	4	3	2	1	0
178	1	0	1	1	0	0	1	0

To set bit 2 to 1 without changing the other bits we must do the following in BASIC:

```
POKE memory,PEEK(memory) OR 4
```

Looking at the bit patterns we can see what happens:

	Bit no.	7	6	5	4	3	2	1	0
	178	1	0	1	1	0	0	1	0
OR	4	0	0	0	0	0	1	0	0
	182	1	0	1	1	0	1	1	0

We can see that the final contents of the memory location are as they were before except that bit 2 has changed from 0 to 1. To reset the same bit to 0, again without affecting the others, the following BASIC statement should be used:

```
POKE memory,PEEK(memory) AND 251
```

Again, looking at the bit patterns makes it clear why the number 251 was chosen.

	Bit no.	7	6	5	4	3	2	1	0
	182	1	0	1	1	0	1	1	0
AND	251	1	1	1	1	1	0	1	1
	178	1	0	1	1	0	0	1	0

In general, to turn on a bit use this `POKE`:

```
POKE memory,PEEK(memory) OR 2↑bitnum
```

and turn it off using

```
POKE memory,PEEK(memory) AND
(255-(2↑bitnum) )
```

As many of a computer's functions are controlled by single bits within special memory locations used by the operating system, known as registers, the ability to alter the state of individual bits within a register can be very useful to those who wish to delve into the computer's inner mysteries.

'Stranded' — An introduction

'Stranded' is an arcade game designed especially for this book and written in Locomotive BASIC for the Amstrad CPC range. At the end of each chapter there is a section on the game, adding routines to the game as the various facilities for graphics and games programming available on the Amstrad

CPC range are introduced. The game uses most of the features the Amstrad CPC range such as BASIC interrupts, screen windows, high-resolution drawing and collision detection. The purpose of these sections is to reinforce the material covered in the chapters and show you how to use the Amstrad CPC range's features to design a high-quality graphics game in BASIC. The intention is to present the game in such a way that you can type in each section of program listing given as you work your way through the book, building up these sections progressively to form the entire program. Each section is designed to run correctly, provided that earlier sections have also been added, allowing you to test each short section of program after you have keyed it in. A complete listing of 'Stranded' is also given in Appendix A.

The game has a group of men stranded on rock pillars in the sea as the tide comes in. The player's task is to climb down a cliff path, row out to the men and rescue each one in turn. The cliff path is dangerous; parts of the path move and there are ladders to climb or descend. Points are scored for each hazard successfully negotiated and each man rescued, but you only have four lives to lose or for three men to get drowned by the incoming tide before the game is ended. 'Stranded' also has a number of different levels of difficulty, the path becoming trickier to walk and the tide coming in faster as you reach each succeeding level.

The program is well structured along the lines of the principles indicated to you in this chapter, subroutines and flags being used for each important program section. The structure will be explained as the various blocks of program are assembled through the book with structure diagrams to show you how to put a large program together in a structured way. I hope that, if you follow the construction of the game carefully, by the end you see why structuring makes large programs such as this so much easier to write.

The emphasis with all these sections is to take the programming ideas learned out of the abstract and show you how they can be put together to make a complex and worth-while program.

Chapter Two

Screen display

In this chapter:

Screen display modes

 Layout and uses

Selection of colours

 PEN, PAPER, INK, SPEED INK
 Logical and actual colour numbers

Positioning characters

 The LOCATE command

Windows

 Defining a window
 Controlling colours and printing
 Overlapping windows

'Stranded'

 Setting up the screen

The screen display modes

The Amstrad CPC range have three different screen modes which may be selected using the MODE command. This table shows what each mode offers in terms of character columns and colours.

Mode	Characters (columns × rows)	Number of simultaneous colours
0	20 × 25	16
1	40 × 25	4
2	80 × 25	2

Mode 0 is the multicolour mode, allowing 16 different colours to be displayed on the screen at the same time. It has, however, only 20 columns across the screen making program listings and other text difficult to read. Mode 0's large character cells and numerous colour combinations make it the ideal mode for most graphics games.

Mode 1 is the normal display mode and is automatically selected by the Amstrad CPC range when it is first switched on. The 40-column screen makes program listings and other text easy to read and is therefore mainly used for programming. The four colours make mode 1 a good choice for displaying graphical and textual information, provided that the information does not require more than 40 characters across the screen.

Mode 2 might be termed the 'business' mode. Only two colours, one foreground and one background, are allowed. The 80-column display makes mode 2 useful for spreadsheet accounting and viewing word-processor text.

The choice of mode also affects the way in which high-resolution graphics are seen but we'll look at that aspect later, in Chapter 4.

Selecting colours

The way in which your Amstrad CPC range is designed to select colours is fairly confusing at first sight. Colour selection is handled by using a combination of INK, PEN, PAPER and BORDER. Colours can be referred to in two ways; either by their 'actual ink number' (AIN), or by what I shall call their 'logical mode colour number' (LMCN). The Amstrad CPC range has 27 different actual colours, each with their own actual ink number:

Actual inks chart

Actual ink number	Colour	Actual ink number	Colour
0	Black	14	Pastel blue
1	Blue	15	Orange
2	Bright blue	16	Pink
3	Red	17	Pastel magenta
4	Magenta	18	Bright green
5	Mauve	19	Sea green
6	Bright red	20	Bright cyan
7	Purple	21	Lime green
8	Bright magenta	22	Pastel green
9	Green	23	Pastel cyan
10	Cyan	24	Bright yellow
11	Sky blue	25	Pastel yellow
12	Yellow	26	Bright white
13	White		

Each mode uses this resource of 27 actual colours to make up a selection of 16, 4 or 2 colours for use within any particular mode. Mode 0, for example, has 16 different colours. To make up this selection it takes 16 of the actual colours and numbers them from 0 to 15, like this:

Logical mode 0 colour chart

Logical mode colour number	Actual ink number	Colour normally seen
0	1	Blue
1	24	Bright yellow
2	20	Bright cyan
3	6	Bright red
4	26	Bright white
5	0	Black
6	2	Bright blue
7	8	Bright magenta
8	10	Cyan
9	12	Yellow
10	14	Pastel blue
11	16	Pink
12	18	Bright green
13	22	Pastel green
14	Flash 1, 24	Flash blue, yellow
15	Flash 16, 11	Flash pink, sky blue

Before we move on to look at each of the other modes let us see how to select a colour in mode 0. Before we start hold down the **CTRL** and **SHIFT** keys together and tap **ESC** to reset the machine.

PEN and PAPER

These two commands control the colour of the writing (the characters) and of the background screen. Both these commands use the logical mode colour numbers to select the foreground and background colours. After switching the machine on, or doing a reset like that described above, the paper colour is blue, LMCN 0, and the pen colour is bright yellow (LMCN 1). To change the pen colour to red type in:

```
MODE Ø
```

followed by:

```
PEN 3
```

The READY message and cursor are now red, but notice that any previous writing on the screen is still yellow, the original foreground colour. To change the pen colour to bright green type:

```
PEN 12
```

What would you type in to change the foreground colour to cyan? Let's reset the foreground colour back to bright yellow by typing:

```
PEN 1
```

and look at the PAPER command. Type in:

```
PAPER 3
```

Notice that the screen behind the READY message has turned bright red, but the rest of the screen remains blue. To change the whole screen to the selected colour the PAPER command must be followed by CLS to clear the screen. Type in

```
CLS
```

To change the whole screen to pink type in:

```
PAPER 11:CLS
```

What command is required to change the screen back to its normal blue colour?

Mode 1's and mode 2's colour charts are similar to that for mode 0 but reflect the fewer colour choices available in these two modes.

Logical mode 1 colour chart

Logical mode colour number	Actual ink number	Colour normally seen
0	1	Blue
1	24	Bright yellow
2	20	Bright cyan
3	6	Bright red

Logical mode 2 colour chart

Logical mode colour number	Actual ink number	Colour normally seen
0	1	Blue
1	24	Bright yellow

Selecting new colours

Although these tables show which colours you can normally expect to get when using a logical mode colour number, that is not the end of the story. It is possible to change any of the actual colour selection given by using the INK command . The way in which INK is used is best illustrated using mode 2. Press **CTRL/SHIFT** and **ESC** to reset and then type:

```
MODE 2
```

The text colour is bright yellow and the screen is blue. The logical colour chart for mode 2 shows that these are the colours that correspond to LMCN 1 and LMCN 0, respectively. As you can see from the table, LMCN 1 corresponds to the actual ink number 24, bright yellow. We can change the ink number for LMCN 1 to 26, bright white, by the following command:

```
INK 1,26
```

Notice that when we do this *all* the text currently displayed on the screen changes from yellow to white, not just the text that follows the command. Similarly we can change LMCN 0, used for the background colour, to purple by entering:

```
INK 0,7
```

Notice that a CLS command does not have to be used to change the entire screen colour.

We can think of our logical mode colour numbers as a group of 2, 4 or 16 pens that can be used to draw on the screen in modes 2, 1 and 0. When first given to you these pens contain certain colours, as shown in the logical mode colour charts. To select one of these pens to write with you use the PEN command and to select one of these colours to write on you use the PAPER command. At any time you can change the colour used by any of the pens (rather like putting a different colour refill in a ball-point pen) by using the INK command.

Note that if you change modes by issuing a MODE command the paper colour always resets to LMCN 0 (normally blue unless changed by using INK) and the text colour always resets to LMCN 1 (normally bright yellow unless changed by INK). Logical mode colour numbers that have been changed by INK do not revert to their normal actual colours after a change of mode, but only if the computer is reset by switching off and on again or by pressing **CTRL/SHIFT/ESC**.

The INK command can also be used to create flashing colours by adding a third number to the command. Type in:

```
MODE 0
```

followed by:

```
INK 2,10,15
```

Nothing appears to happen until you type in:

```
PEN 2
```

The INK command has changed LMCN 2 from its normal bright cyan to a flashing combination of cyan (AIN 10) and orange (AIN 15). The flash rate can be altered by using SPEED INK. Typing in:

```
SPEED INK 100,20
```

makes the first colour in the flashing pair stay on for 2 seconds and the second stay on for 2/5ths of a second. The units used by SPEED INK are 1/50th of a second.

The BORDER command

You probably noticed, when you changed the screen colour using `PAPER`, that an area around the screen did not change colour. Normally you cannot see this border because it is the same colour as the rest of the screen, but it is possible to change the border colour using the `BORDER` command. As the border colour is independent of the screen display mode it does not select its colour using the logical mode colour number system but directly from the 27 actual colour numbers. So:

```
BORDER Ø
```

does not set the border colour to blue (as `PAPER Ø` would set the screen colour to blue) but to black; 0 being the actual ink number for black. The border colour is not reset during mode changes, only after a full machine reset.

Positioning characters

There are several commands in Locomotive BASIC that enable us to position characters on a screen, but the most useful for graphics and games purposes is the `LOCATE` command. Imagine the screen divided up into a series of rows and columns. In mode 0 there are 20 columns and 25 rows. Try typing in this short program and running it:

```
    1Ø MODE Ø
    2Ø LOCATE 1,1
    3Ø PRINT "Q"
```

A capital Q will appear to the top left corner of the screen, in column 1, row 1. Changing line `2Ø` to:

```
2Ø LOCATE 2,1
```

will place the Q in column 2, row 1. By altering the first number only we can make the Q appear anywhere on the top line between column 1 and column 20. Similarly, by altering the second number we can make the Q appear in any row we choose, between 1 and 25. If `charx` is used for the column number and `chary` is used for the row number then we can position characters anywhere on the screen using `LOCATE charx,chary`.

We need not restrict ourselves to single characters we can just as easily position sentences and string or numeric variable values. For example:

```
10 MODE 0
20 LOCATE 5,4
30 PRINT "* LOCATION 5,4"
```

Here the star is the character that is actually at position 5,4. Using LOCATE with sentences or long strings can have its problems. You should be aware that if a sentence or string is too long to fit on a line, the whole string of characters is moved to the beginning of the next line, rather than allowing an overflow of excess characters as one might expect. This can be useful if you want this to happen, but a little annoying if you don't. To demonstrate this, change line 30 in the program above to:

```
30 PRINT "* THIS IS LOCATION 5,4"
```

and insert this line to place a red star at the real position of location 5,4

```
15 LOCATE 5,4:PEN 3:PRINT "*":PEN 1
```

The LOCATE command is of great use to us in setting up static screen displays and will be used in the next chapter to produce character animation.

Windows

Finally in this chapter on screen display let us look at one of the most interesting capabilities of the Amstrad CPC range, their ability to display up to eight simultaneous screen windows.

Defining a window is like defining your own miniature screen within a screen. The foreground and background colours for the window can be set independently from the colours used on the main screen, or in other windows, and the window can be independently printed to, or cleared. The following program shows how the screen can be divided into four windows.

```
1000 REM **** Windows Demo #1 ****
1010 MODE 1
1020 GOSUB 2000:REM define windows
1030 GOSUB 3000:REM set paper/pen for wi
ndows
1040 GOSUB 4000:REM identify each stream
1050 GOSUB 5000:REM write to each window
1060 GOSUB 6000:REM clear each window
1070 GOSUB 7000:REM change pen/paper col
ours
1080 GOSUB 8000:REM reset
1090 END
1100 :
2000 REM **** define windows ****
2010 WINDOW 1,20,1,12
2020 WINDOW #1,21,40,1,12
2030 WINDOW #2,1,20,13,25
2040 WINDOW #3,21,40,13,25
2050 RETURN
2060 :
3000 REM **** set paper/pen for each win
dow
3010 BORDER 0
3020 PAPER 0:PEN 1:CLS          :REM stre
am 0
3030 PAPER #1,1:PEN #1,2:CLS #1:REM stre
am 1
3040 PAPER #2,2:PEN #2,3:CLS #2:REM stre
am 2
3050 PAPER #3,3:PEN #3,0:CLS #3:REM stre
am 3
3060 RETURN
3070 :
4000 REM **** identify each stream ****
4010 FOR stream=0 TO 3
4020 LOCATE #stream,1,1
4030 PRINT #stream,"Stream ";stream
4040 FOR delay=1 TO 1000:NEXT delay
4050 NEXT stream
4060 RETURN
4070 :
5000 REM **** write to each window ****
5010 FOR stream=0 TO 3
```

```
5020 PRINT #stream
5030 PRINT #stream,"This is stream";stre
am;"I am totally separate to other strea
ms"
5040 FOR delay=1 TO 1000:NEXT delay
5050 NEXT stream
5060 RETURN
5070 :
6000 REM **** clear each window ****
6010 FOR stream=3 TO 0 STEP -1
6020 CLS #stream
6030 FOR delay=1 TO 1000:NEXT delay
6040 NEXT stream
6050 RETURN
6060 :
7000 REM **** change colours ****
7010 PAPER 1:PEN 2:CLS          :REM stre
am 0
7020 PAPER #1,2:PEN #1,3:CLS #1:REM stre
am 1
7030 PAPER #2,3:PEN #2,0:CLS #2:REM stre
am 2
7040 PAPER #3,0:PEN #3,1:CLS #3:REM stre
am 3
7050 FOR stream= 0 TO 3
7060 PRINT #stream,"and I can independen
tly change pen and paper colours"
7070 FOR delay=1 TO 1000:NEXT delay
7080 NEXT stream
7090 RETURN
7100 :
8000 REM **** reset ****
8010 PAPER 0:PEN 1:CLS
8020 MODE 1
8030 RETURN
```

This program splits the screen into four quarters and proceeds to demonstrate how each window can be separately controlled. The main concept behind windowing is 'streaming'. Each window area is defined by five numbers, giving its dimensions and a stream number. The commands PEN, PAPER and CLS can be directed to any particular window by following them with the relevant stream number. The window command simply defines an area in terms of character cells (as used by the LOCATE command) and gives it a stream number. The syntax is:

```
WINDOW #stream, left edge, right edge, top
edge, bottom edge
```

If no stream number is specified when using WINDOW, PEN, PAPER, PRINT and CLS then it assumed that you mean stream 0, the normal screen. In the example program the normal screen area is redefined as the top left quarter of the screen. It should be noted that all window-area definitions are destroyed after a change of mode, although pen and paper colour definitions for each stream are retained.

A further point of interest is that you can overlap windows. This program defines two windows that overlap by three character widths in the middle of the screen.

```
10 REM **** windows demo #2 ****
15 MODE 1:BORDER 0
20 GOSUB 1000:REM define windows
30 GOSUB 2000:REM set up pen/paper colou
rs
40 END
50 :
1000 REM **** define windows ****
1010 WINDOW 1,21,1,25
1020 WINDOW #1,19,40,1,25
1030 RETURN
1040 :
2000 REM **** set up pen/paper colours *
***
2010 PAPER 3:PEN 1:CLS
2020 PAPER #1,2:PEN #1,0:CLS #1
2030 RETURN
2040 :
```

Run the program, then type:

```
LIST
```

The program will list into the left-hand half of the screen, but notice that any long lines will extend into the overlap area overlaying the cyan colour of window 1. Now try typing:

```
LIST #1
```

This command directs the program to be listed in the right-hand window. Any intrusions made by the first listing into the overlap area are overlayed again by the blue on cyan listing occurring in window 1. Notice also that although window 1 scrolls during the listing process, as the right-hand half of the screen fills up, the left-hand window does not scroll. Type:

```
LIST
```

again and the program will again list in yellow on red, in the left-hand window, overlaying the previous blue on cyan listing in the overlap area. The simple rule is that where windows overlap, both windows can use the overlap area, writing over anything previously there.

The `LOCATE` command can also be used with each window, in a similar way to the other commands we have mentioned. The top-left corner of any window is defined as location 1,1, and characters can be positioned in the window in the same way as previously discussed. To locate a character at the top-left corner of a window, the following form of the command should be used:

```
LOCATE #stream,1,1
```

where `stream` is the stream number of the window in question.

Windows are very useful in graphics games, where we want to define different coloured areas to represent things like sea or sky. The use of windows allows us to have different coloured backdrops to our game.

'Stranded' — Setting up the screen

This first part of 'Stranded' deals with setting up the screen. This involves defining windows and drawing in part of the cliff-walk, as well as defining each of the colours to be used in the program.

The game is designed to work in the multicolour mode 0, and therefore the screen will have 20 columns and 25 rows. It is often a good idea to sketch out a screen layout on squared paper, like this:

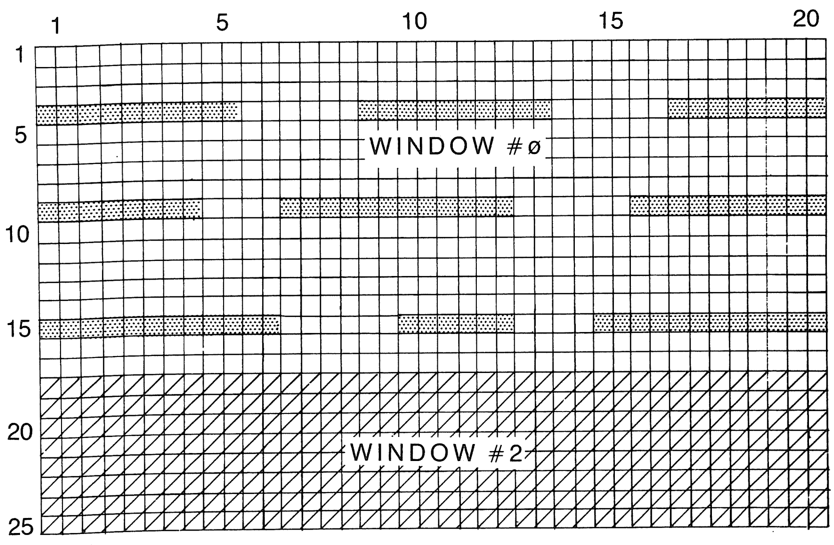


Figure 2.1

The character used to create the cliff-walkway is the chequerboard character `CHR$(207)`. Line `1850` in the initialisation routine uses the `STRING$` command to create a string, `deck$`, made up of 20 of these characters. The `LEFT$` command is then used in lines `2680–2760`, together with `LOCATE` and `PEN`, to create the walkway from portions of `deck$`. `SPACE$(n)` used in these `PRINT` statements creates a string of n space characters.

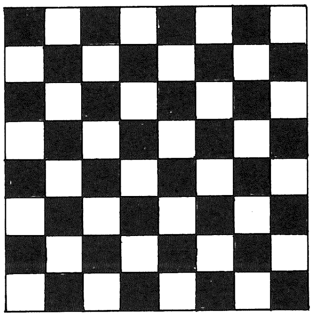


Figure 2.2

Three windows are defined by the initialisation routine. The normal window, stream 0, is defined as the whole screen area. This may seem unnecessary, but it is a failsafe to ensure that window 0 had not been defined before this program was loaded and run. Window 2 takes up the bottom part of the screen, to define the area of sea. Window 3 will serve as a short jetty out into the sea, although at this stage it will not be visiblè, as window 3 has not been cleared yet.

Lines `2620–2660` are concerned with defining the colours that will be used by the program. As a precaution all logical mode colours used are defined using `INK`. It is not assumed that the logical colour numbers correspond to the normal actual colours. Again these may have been redefined, prior to loading and running this program. You will also notice that I have defined several variables such as 'red' or 'black'. It can get quite confusing within a large program to keep track of all the various logical mode and actual ink colour numbers floating about. It is a good idea therefore to set up variable names for each number, telling you what they are. Once defined these variables allow you to write commands like `PEN red` or `PAPER sky`. This avoids having to keep referring to colour charts whilst programming. All the colour variable names like 'red' or 'sea' refer to the logical mode colour numbers, not the actual ink colours. If actual ink numbers are needed, for example for use in an `INK` command, I have used variable names like `redink` to show that these are ink numbers, and cannot be used successfully with the `PEN` or `PAPER` commands.

```
1000 REM ***************************
1010 REM ***************************
1020 REM **                       **
1030 REM **       Stranded!       **
1040 REM **                       **
1050 REM ** (c)1985 S.W. Colwill  **
1060 REM **                       **
1070 REM ***************************
1080 REM ***************************
1240 GOSUB 1800:REM initialise
1270 REM ********************
1280 REM * set up routines *
1290 REM ********************
```

```
1320 GOSUB 2600:REM set up screen
1760 REM ***************
1770 REM * subroutines *
1780 REM ***************
1790 :
1800 REM **** initialisation ****
1810 MODE 0
1840 REM ** define strings **
1850 deck$=STRING$(20,CHR$(207))
1900 REM ** define windows **
1910 WINDOW 1,20,1,25
1920 WINDOW #2,1,20,18,25
1930 WINDOW #3,19,19,18,23
2330 RETURN
2600 REM **** set up screen ****
2610 BORDER 0
2620 whiteink=26:blueink=1:redink=6
2630 sky=10:black=5:water=0:red=3:white=
4
2640 green=12:blue=6:pink=11:pastgreen=1
3:mauve=14
2645 deck(1)=red:deck(2)=green:deck(3)=b
lue
2650 INK 10,14:INK 3,6:INK 4,26:INK 5,0:
INK 14,5
2660 INK 12,18:INK 6,2:INK 11,16:INK 13,
22
2670 PAPER sky:CLS:PAPER #2,water:CLS #2
:PEN #2,white
2680 LOCATE 1,4:PEN deck(1)
2690 PRINT LEFT$(deck$,5);SPACE$(3);LEFT
$(deck$,5);
2700 PRINT SPACE$(3);LEFT$(deck$,4)
2710 LOCATE 1,9:PEN deck(2)
2720 PRINT LEFT$(deck$,4);SPACE$(2);LEFT
$(deck$,6);
2730 PRINT SPACE$(3);LEFT$(deck$,5)
2740 LOCATE 1,15:PEN deck(3)
2750 PRINT LEFT$(deck$,6);SPACE$(3);LEFT
$(deck$,3);
2760 PRINT SPACE$(2);LEFT$(deck$,6)
2770 RETURN
```

Program structure

The structure of the program so far is very simple. Two subroutines are called from the main program; one is an initialisation routine and the other sets up the screen display. The structure diagram so far is:

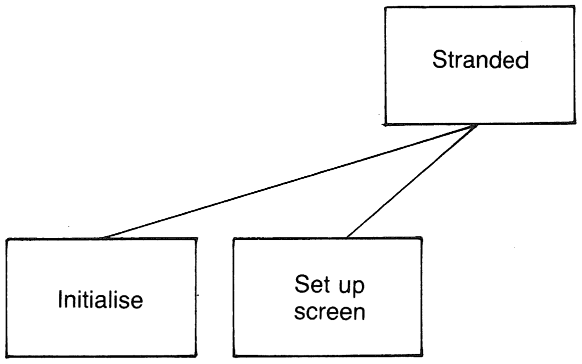


Figure 2.3

Chapter Three

Animation and control

In this chapter:

User-defined characters

In common with most other home computers the Amstrad CPC range display letters, numbers and symbols by lighting up a pattern of small dots within an 8 × 8 dot grid to form the image of the character. These dots are known as picture elements, or 'pixels' for short. Depending on the pattern of pixels lit up within this 8 × 8 grid, different characters can be made to appear. For example, a capital A and a question mark are actually made up of these patterns of dots:

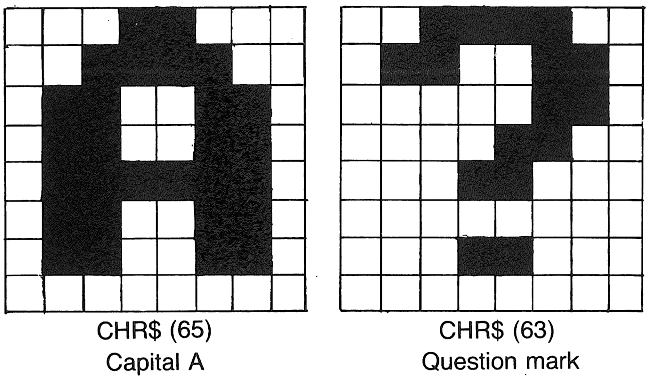


Figure 3.1

The collection of character shapes, like these, that the Amstrad CPC range have stored inside their ROM memory is called the 'character set', and includes the upper- and lower-case alphabet, the digits 0–9, punctuation and arithmetic symbols, together with a number of other special symbols, making 255 characters in total, many of which are available from the keyboard using the keys, either by themselves or together with **SHIFT** or **CTRL**. Each character in the character set can also be accessed by using its CHR$(pronounced 'character string') number. For example, capital A has a CHR$ number of 65. Try typing:

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

to make a capital A appear. Some of the characters in the character set can only be accessed by using their CHR$ number. Type:

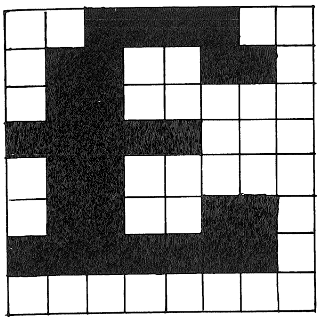
```
PRINT CHR$(164)
```

to print the copyright symbol ©.

The characters with CHR$ numbers 0 and 31 should not be used in this way as they have special control functions that are dealt with in Chapter 7. This short program displays all the characters in the Amstrad CPC range's character set, together with their CHR$ numbers.

```
10 REM **** charset ****
20 MODE 0
30 FOR charnum=32 TO 254 STEP 2
40 PRINT charnum;CHR$(charnum);
50 PRINT SPACE$(5);charnum+1;CHR$(charnu
m+1)
60 PRINT
70 FOR delay=1 TO 500:NEXT delay
80 NEXT charnum
```

Each character is actually held in memory as eight consecutive bytes, representing the eight rows that make up the character. On each row there are eight pixels which are each represented by one bit within the byte for that particular row. If the bit is set to 1 then the corresponding pixel is *on*, if it is set to 0 then the pixel is *off*. The following figure shows how any character can be held as a group of eight bytes, or their decimal equivalents.



CHR$ (163)
Pound sign

128	64	32	16	8	4	2	U	
0	0	1	1	1	1	0	0	60
0	1	1	0	0	1	1	0	102
0	1	1	0	0	0	0	0	96
1	1	1	1	1	0	0	0	248
0	1	1	0	0	0	0	0	96
0	1	1	0	0	1	1	0	96
1	1	1	1	1	1	1	0	254
0	0	0	0	0	0	0	0	0

Binary Image forming eight bytes of data

Decimal. equivalent of each byte

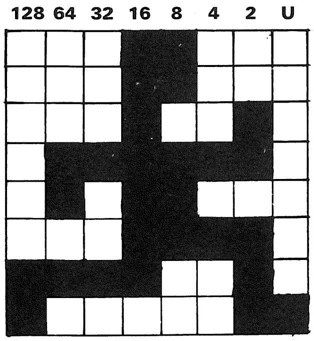
Figure 3.2

Why should we want to know this sort of information? The reason is that we can redefine the £ sign (or any of the other characters with codes 32–255) to our own design using the `SYMBOL` and `SYMBOL AFTER` commands. Type in and run this program:

```
5  MODE Ø
1Ø SYMBOL AFTER 163
```

```
20 SYMBOL 163,24,24,18,
   126,88,30,242,131
30 PRINT CHR$(163)
```

A running man should appear on the screen.



CHR$ (163) Running man	Decimal equivalent of each byte	Hexadecimal equivalent of each byte
	24	& 1 8
	24	& 1 8
	18	& 1 2
	126	& 7 E
	88	& 5 8
	30	& 1 E
	242	& F 2
	131	& 8 3

Figure 3.3

The SYMBOL command has a list of nine numbers following it. The first of these determines the CHR$ number of the character to be redefined, in this case 163, the £ sign. The following eight numbers are the eight numbers that define the eight rows of the character as shown. The character is then printed using its CHR$ number. Once you have run this program, test that our running-man character has indeed taken over from the £ sign by making it appear by pressing the **£** key on the keyboard. The SYMBOL AFTER command determines how many characters can be redefined. If SYMBOL AFTER is not used then the computer assumes a value of 240. This means that 16 characters can be redefined, those from CHR$(240) to CHR$(255). Here the SYMBOL AFTER command allows all the characters from CHR$(163) onwards to be redefined.

One point of interest is that if you redefine a normal letter character, say capital O, then list the program after having run it, all the O's appear as running men (or any other shape you define). However, the program will still work perfectly. Change the number 163 in lines 10 and 20 to 79 and add this line to see the effect:

```
40 LIST
```

If the left line 3Ø as it was then you will also see the normal £ sign printed before the program lists itself. This character has returned to its normal appearance because of the new SYMBOL AFTER command, this resets any changed definitions back to their normal state.

The character-set location in RAM

More advanced programmers may wish to know where the character set lives in the Amstrad CPC range's memories so that they can PEEK and POKE around in them. Firstly, we have to understand the function of the SYMBOL AFTER command. This command copies a portion of the character set from ROM to RAM, so that it may be redefined. The area of memory used is that at the top of the RAM area used by BASIC. The system variable HIMEM gives the address of the highest byte of this area: We can find the top of BASIC memory after a machine reset by typing:

```
PRINT HEX$(HIMEM)
```

The last address used by BASIC is &AB7F (&A67B for disk versions). We can see the effect of the SYMBOL AFTER command by typing in:

```
SYMBOL AFTER 239
PRINT HEX$(HIMEM)
```

This time we find that the top of BASIC memory has been lowered by eight bytes to &AB77 (&A673). This is to make room for the eight bytes defining CHR$(239) (remember that the 128 bytes data defining CHR$(24Ø) to CHR$(255) is automatically copied into RAM on power-up). If we make all characters from CHR$(32) onwards redefinable by copying them into RAM using:

```
SYMBOL AFTER 32
```

then HIMEM drops to &A4FF (&9FFB). The first byte of any eight-byte block defining a character with CHR$(n) can be found from this formula:

address of first byte = &A500 + 8×(n−32)
(&9FFC + 8 × (n−32) for disk systems

assuming that the relevant definition has been copied into RAM using SYMBOL AFTER. Thus the eight bytes defining the £ sign, CHR$(163), can be found from this short program:

```
10 REM **** Findchar ****
20 GOSUB 1000:REM initialise
30 GOSUB 2000:REM find start address
40 GOSUB 3000:REM print out next eight b
ytes
50 END
60 :
1000 REM **** initialise ****
1010 n=163:REM chr$(163) is pound sign
1020 SYMBOL AFTER 163
1030 RETURN
1040 :
2000 REM **** find start address of chr$
(n) ****
2010 addr=&A500+8*(n-32):REM for disk sy
stems use &9FFC
2020 RETURN
2030 :
3000 REM **** print out bytes ****
3010 FOR a=addr TO addr+7
3020 PRINT PEEK(a)
3030 NEXT a
3040 RETURN
```

Having located the RAM definition of any character then, if required, redefinition can be made simply by changing the contents of the eight defining bytes, either from machine code, or from BASIC using POKE.

A user-defined character generator program

This program is a sophisticated utility allowing you to define your own characters on a large 8×8 grid. The decimal equivalents of the eight defining bytes are displayed at all times and can be noted for future use. A further enhancement of the utility is a second display square that allows you to combine several redefined characters together to make larger shapes. Instructions are contained within the program and it is suitable for use with either the cursor keys or a joystick.

Advanced programmers will note the direct access to memory made in the subroutine at line 3000, using the techniques discussed in Chapter 1 to alter individual bits within any of the eight defining bytes.

```
1000 REM ***********************
1005 REM ***********************
1010 REM **                   **
1020 REM **   user defined    **
1030 REM **    character      **
1040 REM **    generator      **
1050 REM **                   **
1060 REM ***********************
1070 REM ***********************
1080 :
1090 GOSUB 9300:REM initialise
1100 GOSUB 9000: REM draw border/title
1110 GOSUB 9500:REM instructions
1120 CLS
1130 :
1140 REM **** main program loop ****
1150 :
1160 edflag=1:REM set edit flag
1170 GOSUB 9000: REM border/title
1180 GOSUB 8000: REM grid
1190 GOSUB 9100: REM display border
1200 GOSUB 6000: REM display
1210 nx=5:ny=6:REM start position
1220 GOSUB 5000:REM test position
1230 EVERY 10,0 GOSUB 4000:REM flash cur
sor
1240 :
1250 WHILE edflag=1
1260 a$=INKEY$:IF a$="" THEN 1260:REM aw
ait key
1270 DI:REM disable interrupt
1280 REM ** cursor keys or joystick **
1290 dx=0:dy=0
1300 IF INKEY(0)=0 OR INKEY(72)=0 THEN d
y=-1
1310 IF INKEY(2)=0 OR INKEY(73)=0 THEN d
y=1
1320 IF INKEY(8)=0 OR INKEY(74)=0 THEN d
x=-1
```

```
1330 IF INKEY(1)=0 OR INKEY(75)=0 THEN d
x=1
1340 REM ** toggle cell ? **
1350 IF INKEY(18)=0 OR INKEY(76)=0 THEN
GOSUB 3000
1360 REM ** quit ? **
1370 IF INKEY(67)=0 THEN END
1380 REM ** z or x keys ? **
1390 IF INKEY(71)=0 THEN upflag=1:GOSUB
2000
1400 IF INKEY(63)=0 THEN upflag=0:GOSUB
2000
1410 REM **** display character ? ****
1420 IF INKEY(61)=0 THEN PEN 3:LOCATE nx
+18,ny:PRINT CHR$(char)
1430 REM **** erase display ? ****
1440 IF INKEY(58)=0 THEN GOSUB 9790
1450 LOCATE nx,ny
1460 PEN charcolour:PRINT charunder$
1470 nx=nx+dx:IF nx>12 THEN nx=5
1480 IF nx<5 THEN nx=12
1490 ny=ny+dy:IF ny>13 THEN ny=6
1500 IF ny<6 THEN ny=13
1510 GOSUB 5000:REM test position
1520 EI:REM enable interrupt
1530 WEND
1540 :
1550 dummy=REMAIN(0):reset timer
1560 GOTO 1140:REM restart loop
1570 END
1580 :
2000 REM **** new character s/r ****
2010 IF upflag=1 THEN char=char+1:IF cha
r>255 THEN char=255
2020 IF upflag=0 THEN char=char-1:IF cha
r<32 THEN char=32
2030 edflag=0:REM unset edit flag
2040 RETURN
2050 :
3000 REM **** edit a cell ****
3010 charunder$=CHR$(287-ASC(charunder$)
)
3020 charcolour=1+(144-ASC(charunder$))*
2
```

```
3030 base=41984+8*char:REM for disk vers
ion use 40700+8*char
3040 addr=base+(ny-6)
3050 IF charunder$=CHR$(143) THEN POKE a
ddr,(PEEK(addr) OR 2^(12-nx))
3060 IF charunder$=CHR$(144) THEN POKE a
ddr,(PEEK(addr) AND (255-2^(12-nx)))
3070 PEN 3:REM white
3080 LOCATE 14,ny
3090 PRINT PEEK(addr);"  "
3100 RETURN
3110 :
4000 REM ** flash cursor **
4010 flash=1-flash
4020 LOCATE nx,ny
4030 IF flash=1 THEN PEN 2:PRINT CHR$(14
3)
4040 IF flash=0 THEN PEN charcolour:PRIN
T charunder$
4050 RETURN
4060 :
5000 REM **** test position s/r ****
5010 gx=16*(nx-0.5)
5020 gy=399-16*(ny-0.5)
5030 charcolour=TEST(gx,gy)
5040 IF charcolour=1 THEN charunder$=CHR
$(144) ELSE charunder$=CHR$(143)
5050 RETURN
5060 :
6000 REM **** display pattern ****
6010 char$=STR$(char)
6020 char$=MID$(char$,2,LEN(char$)-1)
6030 PEN 1:REM black letters
6040 LOCATE 9,4:PRINT SPACE$(5)
6050 LOCATE 5,4:PRINT"CHR$(";char$;")"
6060 PEN 3:REM white shape
6070 PRINT CHR$(22)+CHR$(1):REM trans on

6080 a$(0)=CHR$(32):REM space
6090 a$(1)=CHR$(143):REM square
6100 y=6:REM vertical tab
6110 base=41984+8*char:REM for disk vers
ion use 40700+8*char
6120 FOR address=base TO base+7
```

```
6130 contents=PEEK(address)
6140 GOSUB 7000:REM convert
6150 LOCATE 5,y
6160 PRINT display$;" ";contents
6170 y=y+1
6180 NEXT address
6190 PRINT CHR$(22)+CHR$(0):REM trans of
f
6200 RETURN
6210 :
7000 REM ** convert s/r **
7010 display$=""
7020 FOR dg=7 TO 0 STEP -1
7030 IF (contents AND 2^dg)=2^dg THEN r=
1 ELSE r=0
7040 display$=display$+a$(r)
7050 NEXT dg
7060 RETURN
7070 :
8000 REM **** grid ****
8010 PEN 1:REM black spots
8020 spots$=""
8030 FOR i= 1 TO 8
8040 spots$=spots$+CHR$(144)
8050 NEXT
8060 y=6:REM vertical tab
8070 FOR l = 0 TO 7
8080 LOCATE 5,y+l
8090 PRINT spots$;SPACE$(5)
8100 NEXT l
8110 RETURN
8120 :
9000 REM **** border ****
9010 ORIGIN 0,0
9020 DRAW 0,399,3
9030 DRAW 639,399
9040 DRAW 639,0
9050 DRAW 0,0
9060 PEN 1
9070 LOCATE 7,2:PRINT"CPC 464/664 CHARAC
TER GENERATOR"
9080 LOCATE 7,3:PRINT"------------------
-------------"
9090 RETURN
```

```
9100 :
9200 REM **** display border s/r ****
9210 x=22*16-2:y=399-16*13
9220 MOVE x,y
9230 DRAWR 130,0,2
9240 DRAWR 0,130
9250 DRAWR -130,0
9260 DRAWR 0,-130
9270 RETURN
9280 :
9300 REM **** initialise s/r ****
9310 MODE 1
9320 INK 0,1:REM background
9330 INK 2,2:REM bright blue
9340 INK 1,0: REM pen 1 black
9350 INK 3,26: REM pen 3 white
9360 PAPER 0: BORDER 2
9370 SPEED KEY 30,20
9380 CLS
9390 RETURN
9400 :
9500 REM **** instructions s/r ****
9510 LOCATE 2,4
9520 PRINT"Use the Z and X keys to step
through"
9530 LOCATE 2,5
9540 PRINT"the character set."
9550 LOCATE 2,7
9560 PRINT"Cursor keys or joystick 1 mov
e cursor"
9570 LOCATE 2,8
9580 PRINT"within the edit square. ENTER
 or FIRE"
9590 LOCATE 2,9
9600 PRINT"toggle a cell on or off."
9610 LOCATE 2,11
9620 PRINT"The current character can be
made to"
9630 LOCATE 2,12
9640 PRINT"appear at the corresponding c
ursor"
9650 LOCATE 2,13
9660 PRINT"position by pressing the D ke
y."
```

```
9670 LOCATE 2,15
9680 PRINT"Q quits program."
9690 LOCATE 2,17:PEN 2
9700 PRINT"Reset character set (y/n) ?"
9710 a$=INKEY$:IF a$<>"y" AND a$<>"n" TH
EN 9710
9720 IF a$="y" THEN SYMBOL AFTER 32
9730 LOCATE 2,19
9740 INPUT"Enter first CHR$";cr$
9750 char=VAL(cr$)
9760 IF char<32 OR char>255 THEN LOCATE
18,17:PRINT SPACE$(5):GOTO 9730
9770 RETURN
9780 :
9790 REM **** erase display square s/r *
***
9800 FOR y=0 TO 7
9810 LOCATE 23,y+6
9820 PRINT SPACE$(8)
9830 NEXT y
9840 RETURN
```

Animation

Now that we have the ability to redefine character shapes we can start to learn about animating the shapes we create. Probably the simplest way to animate is to use single-character shapes and interchange them. We could define our own shapes to create a simple piece of animation depicting a man exercising on the spot, but the Amstrad CPC range have prepared some standard characters within the character set that we can use. Let's introduce the figures by typing in:

```
MODE 0
```

followed by:

```
PRINT CHR$(248);CHR$(249)
```

We can create the animation effect by rapidly printing alternating characters.

```
10 REM **** animation demo #1 ****
20 GOSUB 500:REM initialise
```

```
30 WHILE INKEY$="" AND delayval>0
40 GOSUB 1000:REM toggle character
50 WEND
60 END
70 :
500 REM **** initialise ****
510 MODE 0
520 delayval=500
530 RETURN
540 :
1000 REM **** toggle character ****
1010 tog=1-tog
1020 LOCATE 10,12
1030 PRINT CHR$(248+tog)
1040 delayval=delayval-10
1050 GOSUB 2000:REM delay
1060 RETURN
1070 :
2000 REM **** delay ****
2010 FOR i=1 TO delayval:NEXT i
2020 RETURN
```

This simple program has two main parts: an initialisation routine and a WHILE...WEND loop that alternates (or 'toggles') the character printed at location 10,12. Without any delay between one character (say the 'arms up' character) and the other (say the 'arms out' character) the motion produced would be so fast as to blur. A delay loop is therefore introduced to wait around for a while after one character has been printed before printing the other. To demonstrate what different delay values look like, the delay loop limit, delayval, is decremented in tens from its initial value of 500.

Line 1010 shows a useful tip. Often the occasion arises when a value (say) within a subroutine needs to alternate between two specific values each time the subroutine is called. Here we need to alternate between printing CHR$(248), the 'arms up' character, with CHR$(249), the 'arms down' character. When the program is run the initial value of the variable tog will have the value 0. When line 1010 is met for the first time, the value of tog will therefore be changed to 1. On meeting line 1010 a second time tog will be reset to 0 (because 1−1=0) and so on. The value of tog will therefore alternate between 0 and 1 each time the subroutine at line 1000 is called. By subsequently adding 248 to tog in line 1030 we will get alternating CHR$ values of 248+0 and 248+1, i.e. 248 and 249, the CHR$ numbers of our two characters.

We can make a character move around the screen by changing the position it is located at, but we must also print a space over the old position to erase the old character. This short program shows the 'arms down' character walking around the edge of the screen.

```
10 REM **** animation demo #2 ****
20 GOSUB 500:REM initialise
30 GOSUB 1000:REM top edge
40 GOSUB 2000:REM right side
50 GOSUB 3000:REM bottom edge
60 GOSUB 4000:REM left side
70 GOSUB 5000:REM diagonally
80 GOSUB 3000:GOSUB 4000:REM and home
90 END
100 :
500 REM **** initialise ****
510 MODE 0:BORDER 0
520 charx=1:chary=1
530 man$=CHR$(248)
540 delayval=100
550 RETURN
560 :
1000 REM **** along the top edge ****
1010 WHILE charx<20
1020 LOCATE charx,chary:PRINT " ":REM ru
bout old
1030 charx=charx+1
1040 LOCATE charx,chary:PRINT man$;
1050 GOSUB 6000:REM delay
1060 WEND
1070 RETURN
1080 :
2000 REM **** down right side ****
2010 WHILE chary<25
2020 LOCATE charx,chary:PRINT " ":REM ru
bout old
2030 chary=chary+1
```

```
2040 LOCATE charx,chary:PRINT man$;
2050 GOSUB 6000:REM delay
2060 WEND
2070 RETURN
2080 :
3000 REM **** along bottom edge ****
3010 WHILE charx>1
3020 LOCATE charx,chary:PRINT " ":REM ru
bout old
3030 charx=charx-1
3040 LOCATE charx,chary:PRINT man$;
3050 GOSUB 6000:REM delay
3060 WEND
3070 RETURN
3080 :
4000 REM **** up left side ****
4010 WHILE chary>1
4020 LOCATE charx,chary:PRINT " ":REM ru
bout old
4030 chary=chary-1
4040 LOCATE charx,chary:PRINT man$;
4050 GOSUB 6000:REM delay
4060 WEND
4070 RETURN
4080 :
5000 REM **** diagonally ****
5010 WHILE chary<25 AND charx<20
5020 LOCATE charx,chary:PRINT " ":REM ru
bout old
5030 charx=charx+1:chary=chary+1
5040 LOCATE charx,chary:PRINT man$;
5050 GOSUB 6000:REM delay
5060 WEND
5070 RETURN
5080 :
6000 REM **** delay ****
6010 FOR i=1 TO delayval:NEXT i
6020 RETURN
```

The program consists of five major subroutines to deal with the five directions of motion used. The structure of each subroutine is similar: a `WHILE...WEND` loop testing for a terminating condition, and two print statements, one to erase the old position and the second to print the character's new position. The variables charx and chary are used to keep track of the position of the character, each being successively incremented or decremented by 1 to move the character left, right, up, down or diagonally. Notice the semicolon (;) added to end of the print statements where the character is printed. This surpresses the normal carriage return to avoid the screen scrolling when the character is printed at the bottom of the screen.

The above program is designed to show, as simply as possible, the various stages required for animation. You will no doubt have noticed that each movement routine is very similar to the others. Adopting a slightly more structured approach we can design a generalised move routine that blanks out the old character and prints the new. This program uses just such a routine and does exactly the same job as the last program.

```
10 REM **** animation demo #3 ****
20 GOSUB 500:REM initialise
30 GOSUB 1000:REM top edge
40 GOSUB 2000:REM right side
50 GOSUB 3000:REM bottom edge
60 GOSUB 4000:REM left side
70 GOSUB 5000:REM diagonally
80 GOSUB 3000:GOSUB 4000:REM and home
90 END
100 :
500 REM **** initialise ****
510 MODE 0:BORDER 0
520 charx=1:chary=1
530 man$=CHR$(248)
540 delayval=100
550 RETURN
560 :
1000 REM **** along the top edge ****
1010 WHILE charx<20
1020 dx=1:dy=0:GOSUB 5500:REM move
1060 WEND
1070 RETURN
1080 :
2000 REM **** down right side ****
```

```
2010 WHILE chary<25
2020 dx=0:dy=1:GOSUB 5500:REM move
2060 WEND
2070 RETURN
2080 :
3000 REM **** along bottom edge ****
3010 WHILE charx>1
3020 dx=-1:dy=0:GOSUB 5500:REM move
3060 WEND
3070 RETURN
3080 :
4000 REM **** up left side ****
4010 WHILE chary>1
4020 dx=0:dy=-1:GOSUB 5500:REM move
4060 WEND
4070 RETURN
4080 :
5000 REM **** diagonally ****
5010 WHILE chary<25 AND charx<20
5020 dx=1:dy=1:GOSUB 5500:REM move
5060 WEND
5070 RETURN
5080 :
5500 REM **** general move routine ****
5510 LOCATE charx,chary:PRINT " ":REM ru
bout old
5520 charx=charx+dx:chary=chary+dy
5530 LOCATE charx,chary:PRINT man$;
5540 GOSUB 6000:REM delay
5550 RETURN
5560 :
6000 REM **** delay ****
6010 FOR i=1 TO delayval:NEXT i
6020 RETURN
```

Here, instead of incrementing or decrementing `charx` and `chary` explicitly using numbers we do so within the general movement routine at line `5500` using two variables `dx` and `dy`. These variables represent the change in `charx` and `chary`, respectively, and are set just prior to calling the subroutine. If, say, you want to move horizontally to the right then you will want to increase `charx` by 1 each time and leave `chary` unchanged. To do this simply `dx` to 1 and `dy` to

0 before calling the general movement routine, as is done in line `1020` of this program.

Structurally, the second version of the program is an improvement over the first, as the steps that produce movement are not duplicated in the second version. It must also be pointed out that the second version will run slightly slower than the first version as an extra subroutine level has to be called and some time may be wasted in doing unnecessary calculations; for example, calculating `chary+dy` when `dy` is 0. In this simple program speed is not an important factor (in fact we have to put a delay loop in to artificially slow it down!) but in larger programs that require high-speed movement time spent doing unnecessary calculations and subroutine calls can slow movement down considerably. This structure versus speed discussion will be developed later in the book.

An alternative method for defining characters

This program shows an alternative way of defining the running-man graphic using binary number forms. These can be read straight from a set of `DATA` statements and the values used directly with `SYMBOL` to define the character. This method is quite useful as the character's shape can be clearly seen reflected in the pattern of ones used in the `DATA` statements.

```
10 REM **** alternative definition ****
20 DIM d(8)
30 MODE 0
40 GOSUB 1000:REM read data definition 1
50 SYMBOL 254,d(1),d(2),d(3),d(4),d(5),d
(6),d(7),d(8)
60 GOSUB 1000:REM read data definition 2
70 SYMBOL 255,d(1),d(2),d(3),d(4),d(5),d
(6),d(7),d(8)
80 LOCATE 9,12:PRINT CHR$(254);:PRINT CH
R$(255)
90 END
100 :
1000 REM **** read data ****
1010 FOR a=1 TO 8
1020 READ d(a)
```

```
1030 NEXT a
1040 RETURN
1050 :
1060 REM **** definition 1 ****
1070 DATA &X00011000
1080 DATA &X00011000
1090 DATA &X00010010
1100 DATA &X01111110
1110 DATA &X01011000
1120 DATA &X00011110
1130 DATA &X11110010
1140 DATA &X10000011
1150 :
1160 REM **** definition 2 ****
1170 DATA &X00011000
1180 DATA &X00011000
1190 DATA &X01010000
1200 DATA &X01111110
1210 DATA &X00011010
1220 DATA &X01111000
1230 DATA &X01010000
1240 DATA &X00011000
```

Controlling the action

The movement in all the animation examples so far has been directed by the computer; you, the user, have had to sit passively and watch the action taking place. Let us now look at allowing the user to interact with a program, directing movement themselves. There are two main ways in which the user can communicate with a running program: either by pressing keys on the keyboard or by moving the handle of a joystick. On the Amstrad CPC range these two methods are closely interrelated and we shall therefore look at both methods in this section.

Keyboard control

You may well be familiar with the `INPUT` command. This command allows you to type in variable values or an alphabetic response at some stage during a program. This command is not suitable for controlling animation for two reasons. Firstly, it copies anything typed in at the keyboard onto the screen and, secondly, the program pauses waiting for the user to type in a response and press the **ENTER** key before continuing with the program. It is seldom desirable to print keyboard characters to the screen during a game, and having the program stop each time a direction control instruction is to be typed in will not allow the program to have continuous movement. In addition to `INPUT` there are two other commands that allow you to communicate with a running program from the keyboard. These are `INKEY$` and `INKEY`. Both these commands will register a keypress on the keyboard, but do not copy the keypress to the screen or cause the program to pause if no key has been pressed. These commands do, however, work in different ways.

To understand the differences between these two commands we must understand a little about the way in which the keyboard sends signals to the computer. Electronically the keyboard is made up of a grid (or matrix) of electrical wires underneath the keys. Pressing a key causes an electronic signal to be set up in one of the wires in the matrix. The keyboard matrix is scanned every so often (around 50 times a second) by the computer's operating system, looking for signals. If there is a signal in one of the wires then the operating system knows that a key is being pressed and it also knows which key is being pressed from the particular wire in the matrix that carries the signal. (Even when there is no BASIC program running the operating system is scanning the keyboard and turning key presses into characters that it prints on the screen!) Often, if the person is typing very quickly, the keypresses happen too fast to be processed immediately. The operating system therefore places them in a 'waiting area' where they form a queue. This queue is known as the 'keyboard buffer' and it is in the use of this keyboard buffer that the differences between `INKEY$` and `INKEY` are apparent.

When the `INKEY$` command is issued, it takes the character that is at the front of the keyboard buffer queue and holds it as a string. An example of this use is a loop that looks for a yes/no answer before moving on.

```
10 ans$=""
20 WHILE ans$<>"y" AND ans$<>"n"
```

```
30 ans$=INKEY$
40 WEND
```

In line 30 the variable ans$ is assigned the character that is at the front of the keyboard buffer; the WHILE...WEND loop repeating until either a "y" or a "n" character makes its way to the front of the keyboard buffer queue. This command has many advantages over INPUT but its main disadvantage in game control applications is that we (the programmer) have no idea how long the character at the front of the buffer queue has been languishing there waiting for an INKEY$ command to come and get it. Because the Amstrad CPC range keyboards have an automatic repeat on every key (in other words if you hold a key down it will keep sending new characters into the keyboard buffer), this can cause lack of responsiveness in control; as we shall see in a moment. To get around this problem Locomotive BASIC is kind enough to give us a different way of testing the keyboard: the INKEY command. Rather than taking characters from the front of the keyboard buffer, INKEY tests the keyboard matrix wires directly to see if a particular key is being pressed as the INKEY command is issued. Each key has its own INKEY number, and when an INKEY command is sent a value is returned that indicates whether the key in question is being pressed, or held together with **SHIFT** key, or together with the **CTRL** key. Try typing in this program that returns a value for the **X** key (INKEY number 63):

```
 5 pigsfly=0
10 WHILE pigsfly=0
20 A=INKEY(63)
30 PRINT A
40 WEND
```

With the program running you will see a stream of −1 numbers. This value, returned by INKEY indicates that the **X** key is not being pressed. Pressing the **X** key will change the value to 0, whilst the key is being pressed. Now try pressing **SHIFT** and **X** together. A value of 32 is returned. **CTRL/X** returns a value of 128. A combination of **SHIFT/CTRL/X** returns a value of 160. In this way we can tell:

a) If the key in question is being pressed: −1 if not, 0 if pressed.
b) If the key is pressed with **SHIFT**: a value of 32 returned if yes.
c) If the key is pressed with **CTRL**: a value of 128 returned if yes.
d) If the key is pressed with **CTRL** and **SHIFT**: a value of 160 returned if yes.

We can test the value returned in the normal way. For example, break into the program and replace line `10` with:

```
10 WHILE A<>160
```

This time the program will terminate if, and only if, the **X** key is pressed together with **CTRL** and **SHIFT**.

A full list of `INKEY` numbers for all the keys is given in Appendix B.

The following program demonstrates keyboard control in a simple catch game. Three characters are redefined to form a cup shape, the idea being to catch a falling ball in the cup.

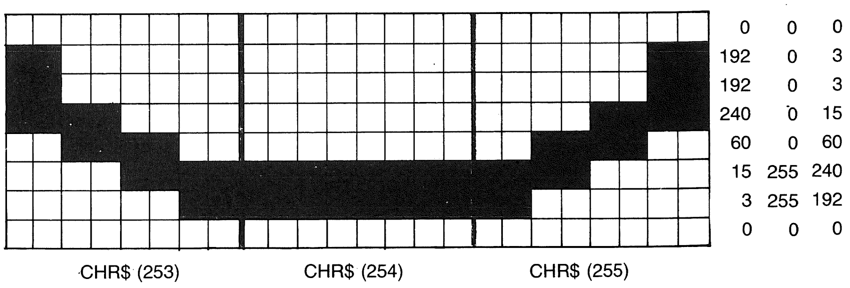


Figure 3.4

The main loop of the program selects a random starting point at the top of the screen for the ball. The ball then drops down using a `FOR...NEXT` loop to increment the ball's vertical coordinate, `bally`. Each time this loop is executed a keyboard scanning routine is called, not once but twice (try omitting line `70` and see what happens). This routine uses `INKEY` to see if the **Z** or **X** keys are pressed down and the horizontal coordinate of the cup, `cupx`, is decremented or incremented accordingly. Notice that checks are included to ensure that `cupx` does not become too small or too large. An upper limit of 38 is tested for rather than 40, as one might expect in mode 1, as the figure printed is three characters wide, occupying columns 38, 39 and 40 at its rightmost position. Notice also that the cup will 'wrap around', that is if it reaches one edge of the screen it will appear at the other edge. Having erased the old position of the cup, the new cup is printed at the updated position. Notice that even if the cup is not moved it will be erased and reprinted in the same place by this routine; a fact that accounts for the flicker in the final shape. Could you redesign the routine another way to overcome this?

```
10 REM **** Catch ****
20 GOSUB 1000:REM initialise
30 WHILE contflag=1
40 RANDOMIZE TIME
50 ballx=INT(RND(1)*40)+1
60 FOR bally=1 TO cupy-1
70 GOSUB 2000:REM scan keyboard
80 LOCATE ballx,bally:PRINT ball$
90 GOSUB 2000:REM scan keyboard
100 LOCATE  ballx,bally:PRINT " "
110 NEXT bally
120 GOSUB 3000:REM test for catch
130 WEND
140 END
150 :
1000 REM **** initialise ****
1010 MODE 1:BORDER 0
1020 cupx=20:cupy=24
1030 REM ** define characters **
1040 SYMBOL 253,0,192,192,240,60,15,3,0
1050 SYMBOL 254,0,0,0,0,0,255,255,0
1060 SYMBOL 255,0,3,3,15,60,240,192,0
1070 cup$=CHR$(253)+CHR$(254)+CHR$(255)
1080 ball$=CHR$(231)
1090 blank$=SPACE$(3)
1100 contflag=1
1110 RETURN
1120 :
2000 REM **** scan for keypress ****
2010 LOCATE cupx,cupy:PRINT blank$
2020 IF INKEY(71)=0 THEN cupx=cupx-1:IF
cupx<1 THEN cupx=38
```

```
2030 IF INKEY(63)=0 THEN cupx=cupx+1:IF
cupx>38 THEN cupx=1
2040 LOCATE cupx,cupy:PRINT cup$
2050 RETURN
2060 :
3000 REM **** test for catch ****
3010 contflag=0
3020 FOR i=0 TO 2
3030 IF ballx=cupx+i THEN contflag=1:i=2
3040 NEXT i
3050 RETURN
```

An alternative to using INKEY is, of course, to use INKEY$. Replace the original subroutine at line **2000** in the 'Catch' program with this subroutine that uses INKEY$.

```
2000 REM **** scan for keypress ****
2010 LOCATE cupx,cupy:PRINT blank$
2015 char$=INKEY$:REM get char fom buffe
r
2020 IF char$="z" THEN cupx=cupx-1:IF cu
px<1 THEN cupx=38
2030 IF char$="x" THEN cupx=cupx+1:IF cu
px>38 THEN cupx=1
2040 LOCATE cupx,cupy:PRINT cup$
2050 RETURN
```

If you run this version of the program you will notice that the cup is less responsive than in the original version of the program. In particular, when you hold down the **Z** or **X** key there is an immediate movement of one character but then a pause before the movement is repeated. As mentioned earlier, the Amstrad CPC range's keys have an auto repeat facility. The way in which a key repeats has two stages: an initial delay after responding to a press, followed by a series of shorter delays between subsequent repeats. The normal repeat speeds are around 2/5ths of a second for the first delay, followed by 1/25th-second delays subsequently. The reason for this longer first delay is to avoid registering double characters when typing. There is a command that allows us to change these delay times called SPEED KEY, its parameters being units of 1/50th of a second.

```
SPEED KEY 20,2
```

is therefore the normal repeat speed setting. Try typing in:

```
SPEED KEY 20,10
```

and then play with the cursor to see the difference altering the second delay period makes. You may wish to experiment with `SPEED KEY` to get rid of the response problem in the `INKEY$` version of 'Catch', but, as a hint, remember to reset the repeat speeds to their normal setting just prior to exiting the program or you may have trouble typing single characters afterwards!

Joystick control

There are no less than three methods of interrogating a joystick plugged into the joystick port. The joystick can be scanned using `INKEY$` or `INKEY`, just as though it were an extension of the keyboard, alternatively it can be read using the new command `JOY`. Let's look at each one of these in turn.

A joystick is essentially made up of four direction buttons and one or two fire buttons. You can't see the direction buttons but pushing the joystick handle one way or another causes the appropriate button to be pressed inside the joystick case. Just as each key on the keyboard has a `CHR$` number (sometimes known as its ASCII code) so each of the five (or six) joystick buttons has an ASCII code that can be used to control movement. If you have a joystick try running this program, moving the joystick handle and pressing the fire button(s).

```
 5 pigsfly=0
10 WHILE pigsfly=0
20 a$=INKEY$
30 IF a$<>"" THEN PRINT ASC(a$)
40 WEND
```

The program prints up the ASCII codes that correspond to each of the joystick buttons as you press them. You should be able to verify that these are the ASCII values for a single joystick plugged directly into the joystick port.

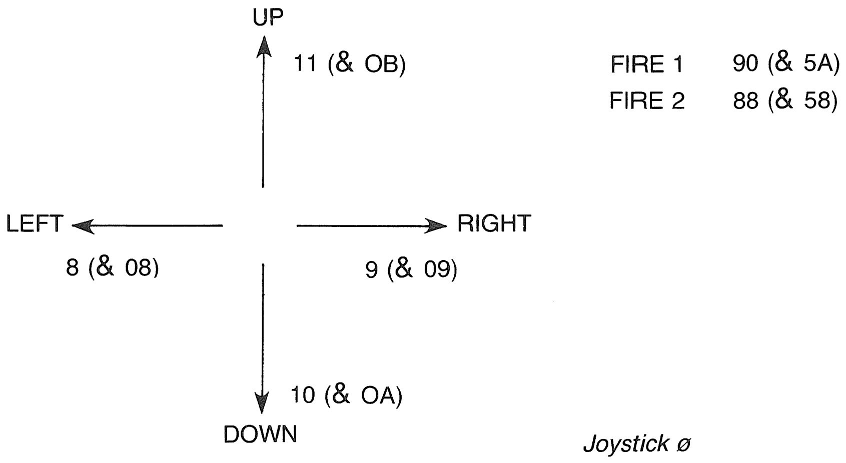


Figure 3.5

Notice that a single joystick plugging directly into the port is referred to as joystick 0 and if the joystick has only one fire button then it will probably be 'fire 2'. Using Amstrad's own make of joystick it is possible to plug a second joystick into the first. This second joystick will return different ASCII values for each button if the **SHIFT** or **CTRL** keys are pressed as well. Where three numbers are given, the lowest is the button on its own, the middle is the button/**SHIFT**, and the highest is the button/**CTRL**. Where only two numbers are given these refer to button only and button/**SHIFT** values. Each of the buttons on joystick 1 has the same ASCII codes as a key on the keyboard. This is useful in programs where two joysticks are

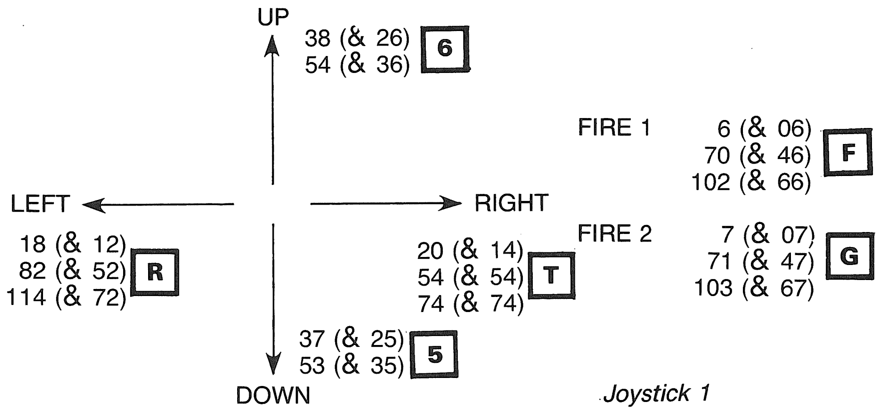


Figure 3.6

required. If the user has only got a single joystick then the action of the second joystick can be reproduced using the relevant keys as marked in the diagram, without having to alter the software.

Joysticks can be interrogated in the same way as keys using INKEY. The INKEY numbers for the joysticks are also given in Appendix B. Note that it is possible to detect diagonal directions with INKEY whereas this was not possible with INKEY$. The following short program shows how this can be done, taking the diagonal direction between 'up' and 'right' for joystick 0.

```
 5 pigsfly=0
10 WHILE pigsfly=0
20 up=INKEY(72):right=INKEY(75)
30 PRINT up,right;
40 IF up=0 AND right=0 THEN PRINT "diag";
50 PRINT
60 WEND
```

Note that the INKEY numbers for joystick 1 mirror the same keyboard keys as the ASCII codes used by INKEY$.

The third method of interrogating joysticks is to use the JOY command. This function reads the state of the joystick buttons. The number returned has so-called 'bit significance'. In other words, if the number returned is thought of as binary then each bit has a special meaning. In this case the less significant six bits correspond directly to the four direction and two fire buttons on the joystick, taking a value of 1 if the button is closed and 0 if it is open. The following short program reads joystick 0 using the JOY command and displaying the value returned as a binary number.

```
10 CLS:pigsfly=0
20 LOCATE 2,10: PRINT "JOY(0)"
30 WHILE pigsfly=0
40 joyval=JOY(0)
50 LOCATE 1,12
60 PRINT BIN$(joyval,8)
70 WEND
```

Run the program to verify that the bits correspond to the joystick buttons as follows:

Bit	Button	Decimal value
Bit 0	Up	1
Bit 1	Down	2
Bit 2	Left	4
Bit 3	Right	8
Bit 4	Fire 2	16
Bit 5	Fire 1	32
Bit 6	No function	64
Bit 7	No function	128

The second joystick can be read in a similar way using JOY(1). Both INKEY and JOY scan the joystick at the same speed (around 50 times a second) but the advantage of JOY over INKEY is that it returns a value that gives the state of all six joystick buttons, whereas each button has to be interrogated separately by INKEY (as in the INKEY example above). Using JOY a diagonal direction is indicated by two direction bits being set. For example, the diagonal direction between 'up' and 'right' would be returned as 00001001 or 9 in decimal. A further advantage of returning the joystick data as a bit-significant number is that individual bits can be isolated and tested using logical AND and OR, as outlined in Chapter 1. For example, testing to see if fire button 2 is pressed could be tested by the following BASIC statement:

```
IF JOY(Ø)=16 THEN PRINT "FIRE 2 PRESSED"
```

This would only work correctly if the joystick handle was in the central position when the fire button was pressed. If, for example, the joystick was pushed into the 'up' position and fire button 2 was simultaneously pressed then JOY(Ø) would return 00010001, i.e. 17 not 16, and the above test would fail. The alternative method is to isolate bit 4, the bit corresponding to fire button 2, and test it independently. The following statement does this:

```
IF (JOY(Ø) AND 16)=16 THEN PRINT "FIRE 2
PRESSED"
```

If you are unsure of the reason behind this try writing out the eight-bit binary equivalents and performing AND in each pair of bits, as described in Chapter 1. As a further example the following test would test both fire buttons independently of the direction buttons to see if either were being pressed.

```
IF (JOY(Ø) AND 48)<>Ø THEN PRINT "BOMBS
AWAY"
```

To introduce a joystick into the 'Catch' game, the following subroutine should be used, in place of the existing one at line 2ØØØ.

```
2000 REM **** scan for keypress ****
2010 LOCATE cupx,cupy:PRINT blank$
2020 IF JOY(0)=4 THEN cupx=cupx-1:IF cup
x<1 THEN cupx=38
2030 IF JOY(0)=8 THEN cupx=cupx+1:IF cup
x>38 THEN cupx=1
2040 LOCATE cupx,cupy:PRINT cup$
2050 RETURN
```

The provision of these three methods of joystick interrogation make it easy to write software that allows either joystick or keyboard control. By slight alteration of the keyboard scanning routine, we can make the 'Catch' program work with either a joystick or keyboard, without having to select one or the other at the start of the program. The following example shows how this can be done using INKEY to read keypressed and joystick movements.

```
2000 REM **** scan for keypress ****
2010 LOCATE cupx,cupy:PRINT blank$
2020 IF INKEY(74)=0 OR INKEY(71)=0 THEN
cupx=cupx-1:IF cupx<1 THEN cupx=38
2030 IF INKEY(75)=0 OR INKEY(63)=0 THEN
cupx=cupx+1:IF cupx>38 THEN cupx=1
2040 LOCATE cupx,cupy:PRINT cup$
2050 RETURN
```

Of course, having multiple tests within such a routine will slow down the speed of the game slightly. If speed is vital then it will be necessary to design separate routines for joystick and keyboard input, asking the user to choose at the beginning of the program.

'Stranded' — assembling the cast

'Stranded' is designed for use with either a single joystick or the cursor and 'copy' keys. Later we will design a proper title screen where the choice between cursor and joystick can be made, but for now, we shall use a temporary section of code so that we can test and use the joystick and cursor control routines given in this section.

```
1250 ans$="":LOCATE 1,12:PRINT "'c' for
cursor"
1251 PRINT "'j' for joystick"
1252 WHILE ans$<>"j" AND ans$<>"c"
1254 ans$=INKEY$
1256 WEND
1258 IF ans$="j" THEN joyflag=1 ELSE joy
flag=0
1260 CLS
```

Before we start we must define the characters that will be used in the program. Some characters, such as the standing-man character, `CHR$(248)`, and the explosion character, `CHR$(238)`, can be lifted straight from the normal character set, but other characters have to be tailor-made for the program.

```
1950 REM ** user defined chars **
1960 SYMBOL AFTER 240
1970 REM SYMBOL 255,14,8,24,30,16,16,16,
16
1980 SYMBOL 254,56,56,18,252,144,40,68,1
34
1990 SYMBOL 253,56,56,18,252,144,16,16,2
4
2000 SYMBOL 245,0,0,0,0,255,127,62,62
2010 SYMBOL 247,28,28,72,63,9,20,35,97
2020 SYMBOL 246,28,28,72,63,9,8,8,24
2030 transon$=CHR$(22)+CHR$(1)
2040 transoff$=CHR$(22)+CHR$(0)
2050 standman$=CHR$(248):boat$=CHR$(245)
2070 explode$=CHR$(238)
```

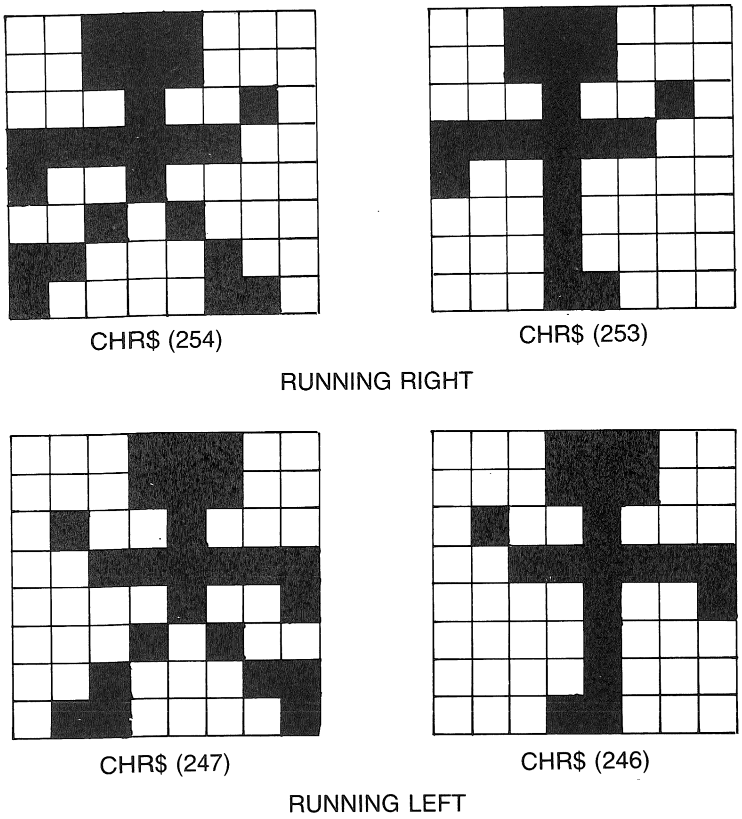


Figure 3.7

We also need to define the number of men that the player has at his disposal. In addition, to make the program more readable, the joystick directions are defined as variables.

```
1880 numbermen=4:men$=STRING$(7,CHR$(248
))
1890 :
2220 REM ** joystick directions **
2230 left=4:right=8:centre=0:fire=16
```

The player's character must have its position and colour initialised. This routine does not form part of the main initialisation routine as later we will need to reset these parameters. This reset is therefore done separately before the main loop starts. Also included in the reset procedure are routines to print the number of lives the player has left in terms of a number of standing-man characters. The variable `men$` was assigned as a string of these characters and `LEFT$` is used to print the relevant number, as defined by `number-men`.

```
2900 REM **** reset routine ****
2910 mancol=white:PEN mancol:charx=1:cha
ry=3
2920 REM ** print number men left **
2930 IF fallflag=1 THEN numbermen=number
men-1
2940 IF numbermen<1 THEN RETURN
2950 LOCATE 1,1:PRINT LEFT$(men$,numberm
en-1);SPACE$(4-numbermen)
2960 RETURN
2970 :
3000 REM **** reset flags ****
3010 boatflag=0:fallflag=0:resetflag=0
3020 fetchflag=0:carryflag=0
3030 RETURN
3040 :
```

The purpose of `fallflag` in line `2930` is to signal when a player has lost a life. We will encounter the routines that set this flag later.

The loop that calls the relevant movement control routine forms part of a main program loop. This calling loop is a simple `WHILE...WEND` structure, terminated by any of the three flags mentioned in line `1460` taking a non-zero value (i.e. being 'set'). At this stage no routines exist that will set these flags so the loop will continue indefinitely until **ESC** is hit twice. After adding all the program segments from this section, you should be able to move the player's character under cursor or joystick control.

```
1360 REM ********************
1370 REM * main action loop *
1380 REM ********************
1390 :
1460 WHILE boatflag=0 AND fallflag=0  AND
 endflag=0
1470 IF joyflag=1 THEN GOSUB 3900 ELSE G
```

```
OSUB 8400
1480 WEND
1490 :
```

Note the use of the `IF...THEN...ELSE` structure to determine whether the joystick or cursor control routines are called, the value of `joyflag` having been assigned when we chose whether we wished to use the cursor keys or the joystick.

The joystick and cursor control routines are fairly self-explanatory, the subroutines at lines `4000` and `4100` handling left and right movement of the player's character. The subroutine at line `4200` prints a standing man, if no movement is required. The use of `ladflag` in line `3930` and `8430` will be discussed in a future chapter. The movement routines each use a pair of toggled characters to simulate a running action, either to the left, or to the right.

```
3900 REM **** scan joystick ****
3910 IF JOY(0)=left THEN DI:GOSUB 4000:G
OSUB 4300:EI
3920 IF JOY(0)=right THEN DI:GOSUB 4100:
GOSUB 4300:EI
3930 IF JOY(0)=centre AND ladflag=0 THEN
 DI:GOSUB 4200:GOSUB 4300:EI
3940 RETURN
3950 :
8400 REM **** cursor control on ladders
etc ****
8410 IF INKEY(8)=0 THEN DI:GOSUB 4000:GO
SUB 4300:EI
8420 IF INKEY(1)=0 THEN DI:GOSUB 4100:GO
SUB 4300:EI
8430 IF INKEY(1)<>0 AND INKEY(8)<>0 AND
ladflag=0 THEN DI:GOSUB 4200:GOSUB 4300:
EI
8440 RETURN
8450 :
4000 REM **** move left ****
4010 chartog=1-chartog:REM character typ
e
4020 LOCATE charx,chary:PRINT" "
```

```
4030 charx=charx-1:IF charx<1 THEN charx
=1
4040 LOCATE charx,chary:PRINT CHR$(246+c
hartog)
4070 RETURN
4080 :
4100 REM **** move right ****
4110 chartog=1-chartog:REM character typ
e
4120 LOCATE charx,chary:PRINT" "
4130 charx=charx+1:IF charx>20 THEN char
x=20
4140 LOCATE charx,chary:PRINT CHR$(253+c
hartog)
4170 RETURN
4180 :
4200 REM **** standing still ****
4210 LOCATE charx,chary:PRINT standman$
4220 RETURN
4230 :
```

The subroutine call to 4300 involves testing under the player's figure to see if he is walking over a gap in the walkway. Testing for collisions (or in this case the lack of collision between the player's character and the walkway!) is one of the topics discussed in the next chapter so for now we will simply insert the subroutine's title and RETURN statement. This dummy routine will stop the program crashing.

```
4300 REM **** check under figure ****
4360 RETURN
```

2. Program structure

The structure of the program is now developing rapidly. We have four preparatory routines and a WHILE...WEND loop enabling the program to scan the joystick or cursor keys. Notice how a binary (i.e. two possible outcomes) decision can be represented on a structure diagram, the true and false branches corresponding directly to the THEN and ELSE part of the IF...THEN...ELSE construct. It is also easy to show how the 'scan joystick' and 'scan cursor keys' routines can share the same movement subroutines.

Program structure

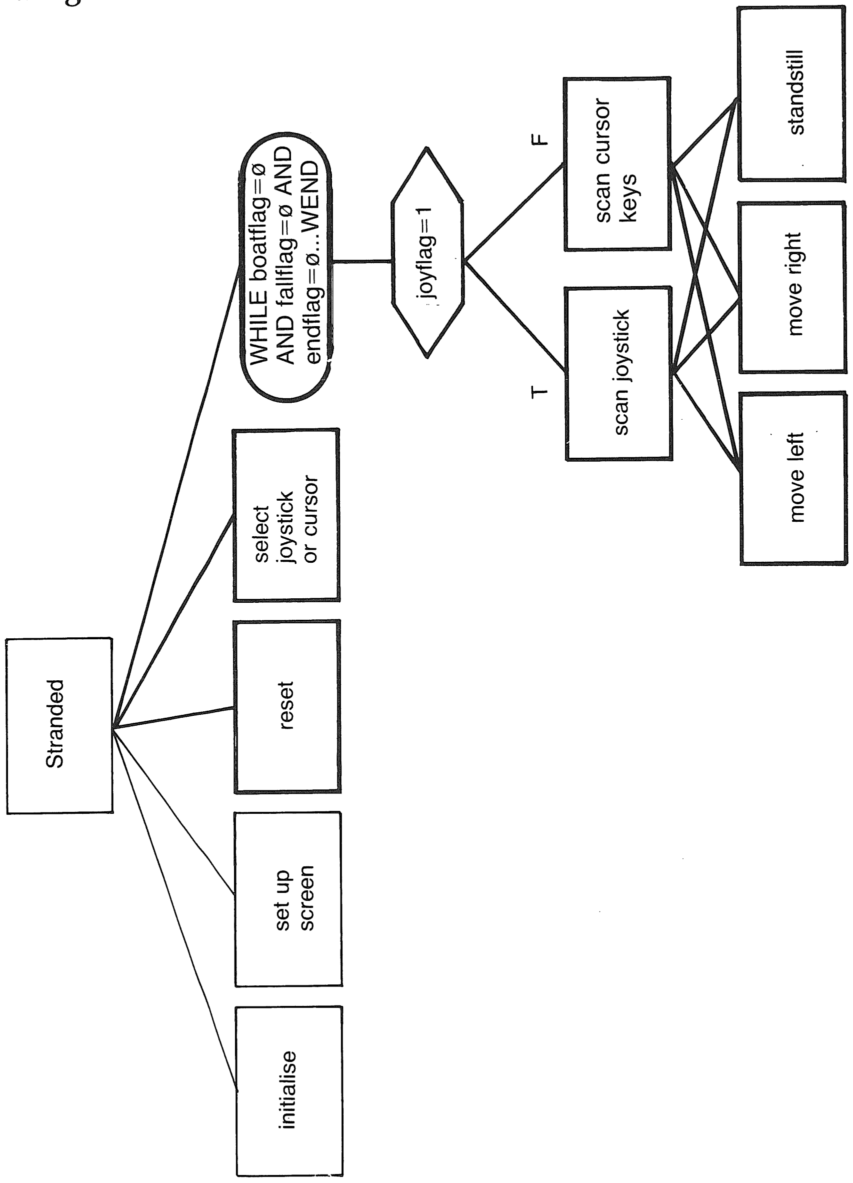


Figure 3.8

Chapter Four

High-resolution graphics

In this chapter:

The Amstrad CPC range have excellent high-resolution graphics that allow us to create detailed pictures and to mix graphics and text on the screen. High-resolution graphics work on a coordinate system that is essentially the same in all three modes. To identify a point on the screen we have to specify an x coordinate and a y coordinate. (Most of you will remember drawing graphs and plotting points like this at school.) The origin, the point where x and y are both zero is normally in the bottom-left corner of the screen and we can plot values of x up to 639 and values of y up to 399. This diagram shows the layout.

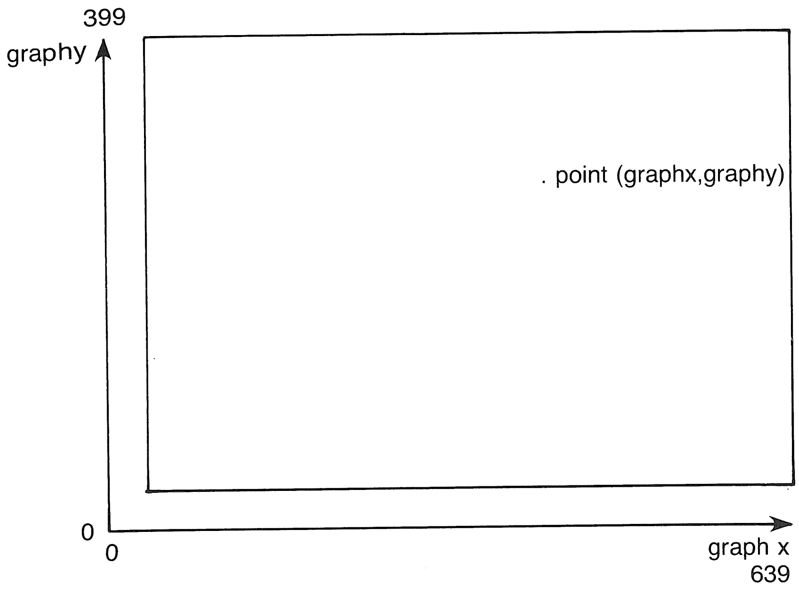


Figure 4.1

The simplest way to make a high-resolution point appear on the screen is to PLOT it. Try typing:

```
MODE 2:PLOT 0,0
```

to see a point appear at the origin. If you try the same in mode 1 and mode 0 you will see that we get larger points but still in the same place. More of this later.

We can set the logical mode colour for high resolution by adding a third number to the PLOT command. Try:

```
MODE Ø:PLOT Ø,Ø,2
```

to change the colour of the point plotted to cyan (LMCN 2). Notice that the colour of the READY printed after the PLOT command has not changed colour. We can have independent high-resolution and text colours operating at the same time. Conversly, if we change the text colour using PEN 2 and then plot a point using LMCN 1, we get a yellow point, but the READY is printed in cyan. Try:

```
PEN 2:PLOT Ø,Ø,1
```

The thing to remember about text and high-resolution colours is that both use the logical mode colour number system to identify colours, but the colours selected for each type of graphic are otherwise unconnected.

As well as PLOT we can DRAW on the screen. A DRAW command has a pair of coordinates and draws a line between the last point plotted and the point whose coordinates form part of the DRAW statement. For example, type in:

```
DRAW 1ØØ,5Ø,3
```

This draws a line between (Ø,Ø), the last point plotted and (1ØØ,5Ø) in LMCN 3, red. If we now type:

```
DRAW 2ØØ,Ø,2
```

we get another line in cyan, starting at (1ØØ,5Ø) and ending at (2ØØ,Ø). This line starts at (1ØØ,5Ø) because that was the last point plotted before the new draw command was given. Most programmers think of the way that a computer remembers the last point plotted each time as an invisible graphics cursor that moves around the screen as we enter high-resolution graphics commands. We can find the position of this invisible graphics cursor at any time. Its coordinates are held in two special variables XPOS and YPOS. Try:

```
PRINT XPOS,YPOS
```

If you have typed in all the commands given earlier then the values of XPOS and YPOS will be 200 and 0, respectively. If we wish we can move the graphics cursor around without plotting or drawing using the MOVE command. Type in:

```
MOVE 500,100:DRAW 0,0
```

to move the graphics cursor out to a new start point and draw a line back to the origin.

Press **ENTER** a few times and you will see the text scroll up the screen. Notice that any high resolution also scrolls up. You must always be careful, when creating a high-resolution picture that you do not print text at the bottom of the screen and cause the screen to scroll, or your display will be spoilt.

PLOT, DRAW and MOVE are known as 'absolute' high-resolution commands. The coordinates used to specify points are actual positions on the screen. There is another group of commands that allows you to plot or draw 'relative' to the last point plotted. These are PLOTR, DRAWR and MOVER. Type:

```
CLS:PLOT 320,200
```

to clear the screen and plot a point roughly in the middle. Now type:

```
PLOTR 10,10
```

This does not plot a point at the 'actual' point (10,10) but at a point near to the first one plotted. What the second command really does is to plot a point 10 units along and 10 units up from the first point. These are called the x and y 'offsets' from the previous position. DRAWR and MOVER work in the same way.

```
DRAWR 100,50,3
```

will draw a line from the current cursor position to a point 100 units along and 50 units up from that position.

We can use either method to plot points and draw lines on the screen. Often the relative plotting method is useful for designing short graphics routines where a shape can be drawn on the screen by, firstly, moving to a point using MOVE and then creating the shape relative to the starting point by using MOVER, PLOTR and DRAWR. This short program demonstrates the method, using a general triangle-drawing subroutine to place triangles all over the screen.

```
10 REM **** relative triangles ****
20 MODE 1:BORDER 0
```

```
30 CLS
40 WHILE INKEY$=""
50 x=INT(RND(1)*590)
60 y=INT(RND(1)*350)
70 GOSUB 1000:REM draw triangle
80 WEND
90 END
100 :
1000 REM **** general triangle ****
1010 MOVE x,y
1020 DRAWR 50,0
1030 DRAWR -25,43
1040 DRAWR -25,-43
1050 RETURN
```

A particularly useful feature of the collection of high-resolution commands is the command that tests a point on the screen to see what colour is there. `TEST(x,y)` will return the logical colour number of the point with coordinates `(x,y)`. This facility allows us to test for collisions in games. There is, of course, a relative form of the `TEST` command, `TESTR`, that uses offsets to specify the point to be tested, rather than using absolute coordinates.

Light cycles

The following program demonstrates much of the ground we have covered in the early chapters of this book. It is a game for two players, both controlling light cycles from the keyboard. Player 1 uses the **W**, **X**, **A** and **D** keys and player 2 uses the **11**, **8**, **4** and **6** keys on the numeric keypad alongside the main keyboard. The idea is to avoid running over the opponent's trail, or doubling back over your own trail.

The program works by scanning all eight of the players' keys and updating the coordinates of the front of each light cycle. Before the point that is to make the new front of the cycle is plotted the position is tested for any other colour except logical colour 0, the normal blue background. In this way collisions between cycles, or between a cycle and the border or between a cycle front and its own tail are all tested for. The explosion effect is created by drawing 100 random lines relatively out to points up to 50 units from the point of the collision. In this program we have combined key control with high-resolution plotting and collision detection for the first time.

```
10 REM **** light cycles ****
20 GOSUB 2000:REM initialise
30 WHILE pigfly=0
40 GOSUB 1000:REM scan keys etc
50 WEND
60 END
70 :
1000 REM **** scan keyboard etc ****
1010 IF INKEY(59)<>-1 THEN dx=0:dy=2
1020 IF INKEY(63)<>-1 THEN dx=0:dy=-2
1030 IF INKEY(69)<>-1 THEN dx=-2:dy=0
1040 IF INKEY(61)<>-1 THEN dx=2:dy=0
1050 :
1060 IF INKEY(11)<>-1 THEN dm=0:dn=2
1070 IF INKEY(14)<>-1 THEN dm=0:dn=-2
1080 IF INKEY(20)<>-1 THEN dm=-2:dn=0
1090 IF INKEY(4)<>-1 THEN dm=2:dn=0
1100 :
1110 REM ** add to coordinate values **
1120 x=x+dx:y=y+dy
1130 m=m+dm:n=n+dn
1140 REM ** test for collisions **
1150 IF TEST(x,y)<>0 THEN f=1:GOSUB 6000
:RETURN
1160 IF TEST(m,n)<>0 THEN f=2:GOSUB 6000
:RETURN
1170 REM ** plot new cycle fronts **
1180 PLOT x,y,1
1190 PLOT m,n,2
1200 RETURN
1210 :
2000 REM **** initialise ****
2010 MODE 1:BORDER 0
2020 GOSUB 3000:REM draw border
2030 GOSUB 4000:REM set up scores
2040 GOSUB 5000:REM reset coords
2050 INK 2,20
2060 RETURN
2070 :
```

```
3000 REM **** draw border s/r ****
3010 MOVE 10,10
3020 DRAW 10,380,3
3030 DRAW 629,380
3040 DRAW 629,10
3050 DRAW 10,10
3060 RETURN
3070 :
4000 REM **** print scores s/r ****
4010 LOCATE 3,1
4020 PRINT"player 1:";s1
4030 LOCATE 28,1
4040 PRINT"player 2:";s2
4050 RETURN
4060 :
5000 REM **** reset coords etc ****
5010 x=50:y=200:m=580:n=200
5020 dx=2:dy=0:dm=-2:dn=0
5030 RETURN
5040 :
6000 REM **** collsion s/r ****
6010 IF f=1 THEN s2=s2+1 ELSE s1=s1+1
6020 GOSUB 7000:REM explosion
6030 CLS
6040 GOSUB 3000: REM draw border
6050 GOSUB 4000: REM inc scores
6060 GOSUB 5000:REM reset coords
6070 LOCATE 10,12
6080 PRINT"press a key to restart"
6090 j$="dummy":WHILE j$<>"":j$=INKEY$:W
END
6100 a$="":WHILE a$="":a$=INKEY$:WEND
6110 LOCATE 10,12
6120 PRINT SPACE$(24):REM rubout message
6130 RETURN
6140 :
7000 REM **** explosion ****
7010 IF f=1 THEN ex=x:ey=y ELSE ex=m:ey=
n
7020 FOR i=1 TO 100
7030 MOVE ex,ey
7040 rx=25-INT(RND(1)*50+1)
7050 ry=25-INT(RND(1)*50+1)
7060 cl=INT(RND(1)*3+1)
```

```
7070 DRAWR rx,ry,cl
7080 NEXT i
7090 RETURN
```

Other points of interest, besides the control and collision aspects of the game, are the way in which a flag, `f`, is used to indicate which cycle caused the collision before calling the collision subroutine to update the score and reset the game. In addition when the players are asked to press a key to restart there is a good chance that there will already be a character in the keyboard buffer that will trigger an immediate restart if we just use a simple loop and `INKEY$`, as we have before to detect a keypress. Instead, a special loop is used first, not to detect a character in the keyboard buffer, but to clear the keyboard buffer out. Once the buffer is empty we can then employ the more usual loop looking for a keypress. Clearing the keyboard buffer, prior to testing for a keypress, is often necessary in games that are controlled from the keyboard.

Three-dimensional effects

The Amstrad CPC range's high-resolution graphics can be used to simulate three-dimensional shapes on a flat monitor screen. One method is to introduce ideas of depth and perspective by overlaying similar shapes of decreasing scale. A simple form of this technique can be easily demonstrated using a general purpose triangle-drawing routine, similar to that used in a previous program. This time, however, we should include a facility that will allow us to alter the size of the triangle each time we call the subroutine. By moving the start point of the triangle progressively up and right, and decreasing the size of the triangle each time we do so, we can produce a simulated three-dimensional shape like a prism (or 'Toblerone' box shape). The following program uses such a general triangle-drawing routine and a loop in which the triangle's start point is altered, and the triangle's size is steadily diminished.

```
10 REM **** 3D prism ****
20 GOSUB 1000:REM initialise
```

```
30 FOR dx=15 TO -15 STEP-2
40 GOSUB 4000:REM reset size,x,y
50 GOSUB 2000:REM draw prism
60 NEXT dx
70 END
80 :
1000 REM **** initialise ****
1010 MODE 2:DEG
1020 GOSUB 4000:REM reset size,x,y
1030 dy=2:ds=8
1040 RETURN
1050 :
2000 REM **** draw prism ****
2010 WHILE size>100
2020 MOVE x,y
2030 GOSUB 2070:REM draw triangle
2040 x=x+dx:y=y+dy:size=size-ds
2050 WEND
2060 RETURN
2070 :
3000 REM ***** draw tringle ****

3010 height=size*SIN(60)
3020 halfbase=size/2
3030 DRAWR size,0
3040 DRAWR -halfbase,height
3050 DRAWR -halfbase,-height
3060 RETURN
3070 :
4000 REM **** reset size,x,y ****
4010 CLS
4020 size=200:x=250:y=100
4030 RETURN
```

For those interested (this information is not essential!) an equilateral triangle can be drawn using simple rules of trigonometry. The height of the triangle is simply the side length, `size`, multiplied by `SIN(60)`.

The triangle's three sides are drawn anticlockwise from the bottom left corner using the relative offsets `halfbase` and `height`.

The perspective of the final prism shape is dependent on three things: the horizontal and vertical moves between one

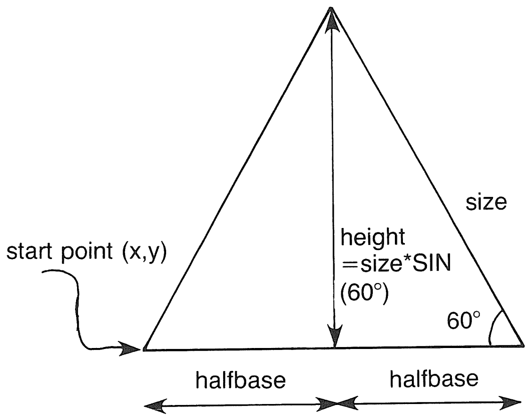


Figure 4.2

triangle's start point and the next, and the amount by which size decreases between each successive triangle. In the program these three factors are called dx, dy and ds. To show how the perspective can be changed by altering one of these factors, the program draws a number of prisms, each time decreasing dx by 2. The result is a simple horizontal shift in perspective of the shape seen. You may like to modify the program to see what effect progressively altering dy or ds has on the shape.

Circles and ellipses

Although the Amstrad CPC range's set of high-resolution commands does not include a command for drawing circles we can easily use the commands we do have available to create circles. This simple program shows how:

```
10 REM **** draw a circle ****
20 MODE 1:RAD
30 radius=150:cx=320:cy=200
40 MOVE cx+radius,cy
50 FOR angle=0 TO 2*PI STEP 0.2
60 x=cx+radius*COS(angle)
70 y=cy+radius*SIN(angle)
80 PLOT x,y
90 NEXT angle
100 PLOT  cx+radius,cy
```

The program does not actually draw a complete circle but a series of dots around its circumference. The number of dots can be altered by changing the `STEP` length in the `FOR` statement at line `5Ø`. A `STEP` of 0.1 for example gives many more dots on the circumference and we get a better-defined circle. It is actually impossible to draw an exact circle on a computer screen, but the more points we calculate around the circle, the nearer we get to that elusive perfect circle. To join up the dots to make an unbroken circle can be done by exchanging the `PLOT` statements in lines `8Ø` and `1ØØ` for `DRAW`. Try experimenting with different `STEP` lengths. You will find that there is a trade-off between speed and shape; the smaller the `STEP` used, the better the circle's shape, but, as more points are calculated and drawn, the slower the circle is drawn on the screen.

Points on the circle are found by using the `SIN` and `COS` functions to calculate the x and y offsets from the centre of the circle. The loop calculates points by increasing the angle from `Ø` to `2*PI` radians (or 0 to 360 degrees).

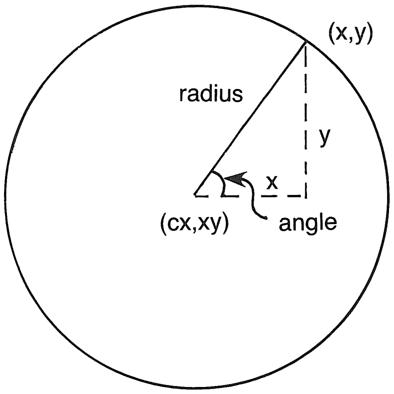


Figure 4.3

One of the advantages of using this method to plot circles is that it can be easily adapted for drawing ellipses. We can squash the shape of the circle vertically by multiplying the y value for each point on the circle by a number less than 1. Try altering line `7Ø` to:

```
7Ø y=cy+(radius*SIN(angle))*Ø.5
```

and run the program. You will see that the vertical width of the original circle has been halved, forming an ellipse.

Three-dimensional circles

An alternative way to view an ellipse like the one just drawn, is as a three-dimensional circular hoop viewed from an angle. This program uses this fact to produce some impressive three-dimensional effects. The program works by calculating a set of points on an ellipse and then using those points as the centres of smaller circles. The program allows you to enter a number of parameters. The following table shows what each parameter does and gives three example sets for you to try with the program.

Parameter	Example 1	Example 2	Example 3	Comments
Start radius	90	60	80	Initial radius of small circles
Reduction ratio	0.98	1	0.97	Reduction of small circles' radius factor
Number of circles	200	100	61	Total number of small circles drawn
Aspect ratio	0	0.4	0.6	Angle at which large hoop appears to be drawn
Number of circuits	2.5	1	10	Total number of times around large hoop

By altering the values of the five parameters different shapes can be made to appear, from simple doughnuts to more complex figures that appear to twist inside themselves. The three example parameter sets given in the table demonstrate three very different shapes. Small modifications in one or more parameters can make interesting changes to the basic types shown.

```
10 REM **** 3D circles ****
20 MODE 2
30 INPUT"starting radius";sr
40 INPUT"reduction ratio";rr
```

```
50 INPUT"number of circles";nc
60 INPUT"aspect ratio";ar
70 INPUT"number of circuits";cn
80 GOSUB 1000:REM print data
90 ORIGIN 320,200
100 r=150
110 GOSUB 2000:REM draw pattern
120 END
130 :
1000 REM **** data output s/r ****
1010 CLS
1020 LOCATE 10,1: PRINT"starting radius"
;sr
1030 LOCATE 50,1:PRINT"reduction ratio";
rr
1040 LOCATE 10,2:PRINT"number of circles
";nc
1050 LOCATE 50,2:PRINT"aspect ratio";ar
1060 LOCATE 10,3:PRINT"number of circuit
s";cn
1070 RETURN
1080 :
2000 REM **** large circle s/r ****
2010 MOVE r,0
2020 RAD
2030 sg=2*PI*cn/nc
2040 FOR i=0 TO nc
2050 ag=i*sg
2060 x=r*COS(ag)
2070 y=(r*SIN(ag))*ar
2080 GOSUB 3000: REM small circles
2090 NEXT i
2100 RETURN
2110 :
3000 REM **** small circles s/r ****
3010 MOVE x+sr,y
3020 FOR j= 0 TO 2*PI STEP 0.2
3030 m=x+sr*COS(j)
3040 n=y+sr*SIN(j)
3050 DRAW m,n
3060 NEXT j
3070 DRAW x+sr,y
3080 sr=sr*rr
3090 RETURN
```

Three-dimensional graphics

An alternative to creating three-dimensional effects by using repeating similar figures is to represent the three-dimensional shape as a net; in other words, as a set of crossing lines that define the surfaces of the figure. This technique is very popular in computer-aided engineering design and commercial computer systems allow the net figures to be rotated so that they can be viewed from different angles. The three-dimensional graphics program given here uses mathematical functions to create three-dimensional nets, calculating a series of lateral lines across the screen, and a second series of lines appearing to go into the screen from the same function. Four examples have been given in the subroutines starting at 4000. In each subroutine a mathematical formula links x, y and z. It is the relationship between these three variables that gives each net its distinctive shape. You can experiment with your own mathematical formulae by replacing the one given in the subroutine at line 4300.

The methods of calculation are rather complex but are based on a two-dimensional array, each element representing a point on the horizontal plane (the x–z plane). The initial calculation routine uses the formula given in the relevant subroutine to calculate a corresponding y value for each point on the plane. The array, therefore, holds the height of the figure above each point on the plane. The array is then scanned twice to produce the lateral and depth lines that go to make up the net.

```
10 REM **** 3D Graphics ****
20 GOSUB 1000:REM initialise
30 FOR cc=1 TO 4
40 GOSUB 2000:REM calculate
50 GOSUB 3000:REM draw shape
60 REM ** await keypress **
70 j$=INKEY$:WHILE j$<>"":j$=INKEY$:WEND
80 a$="":WHILE a$="":a$=INKEY$:WEND
90 NEXT cc
100 END
110 :
1000 REM **** initialisation s/r ****
1010 MODE 2
```

```
1020 ac=640:hi=400:up=-1:st=1:xg=12:zg=7

1030 wi=INT(ac/xg/2)
1040 de=INT(hi/zg/3)
1050 DIM p(wi,de)
1060 RETURN
1070 :
2000 REM **** calculation s/r ****
2010 CLS:INK 1,24,1
2020 LOCATE 28,1:PRINT"Please Wait - Cal
culating"
2030 FOR a=-de/2 TO de/2
2040 FOR b=-wi/2 TO wi/2
2050 x=20*a/wi:z=20*b/de
2060 y=20*(b-wi/2)/de
2070 ON cc GOSUB 4000,4100,4200,4300
2080 p(b+wi/2,a+de/2)=y*up*hi
2090 NEXT b,a
2100 RETURN
2110 :
3000 REM **** draw shape s/r ****
3010 CLS
3020 REM ** title and colour **
3030 ON cc GOSUB 5000,5100,5200,5300
3040 :
3050 REM ** lateral lines **
3060 FOR z=1 TO de
3070 xb=xg*z:zb=hi/2+z*zg+st*up
3080 xo=xb+xg:zo=zb-zg-p(1,z)
3090 FOR x=1 TO wi
3100 xn=xb+x*xg:zn=zb-x*zg-p(x,z)
3110 MOVE xo,zo:DRAW xn,zn
3120 xo=xn:zo=zn
3130 NEXT x,z
3140 :
3150 REM ** depth lines **
3160 FOR x= 1 TO wi
3170 xb=xg*x+de*xg:zb=hi/2-x*zg+de*zg+st
*up
3180 zo=zb-zg-p(x,de-1):xo=xb-xg
3190 FOR z=0 TO de-1
3200 xn=xb-z*xg:zn=zb-z*zg-p(x,de-z)
3210 zn=zb-z*zg-p(x,de-z)
3220 MOVE xo,zo:DRAW xn,zn
```

```
3230 xo=xn:zo=zn
3240 NEXT z,x
3250 RETURN
3260 :
4000 REM **** simple plane ****
4010 y=(SIN(x)+COS(z))/45
4020 RETURN
4030 :
4100 REM **** hemishere ****
4110 fc=60-x^2-z^2
4120 y=SQR(fc*(SGN(fc)+1))/30
4130 RETURN
4140 :
4200 REM **** pillar ****
4210 fc=x^2+z^2
4220 y=SGN(INT(20/fc))/3+SGN(INT(55/fc))
/15
4230 RETURN
4240 :
4300 REM **** your shape ****
4310 fc=LOG(50*ABS(z)+0.0001)
4320 y=(fc*SGN(fc+1)-5)/15
4330 RETURN
4340 :
5000 REM ***** plane title ****
5010 INK 1,20
5020 LOCATE 34,1:PRINT"simple plane"
5030 RETURN
5040 :
5100 REM **** hemisphere title ****
5110 INK 1,24
5120 LOCATE 35,1:PRINT"hemisphere"
5130 RETURN
5140 :
5200 REM **** pillar title ****
5210 INK 1,15
5220 LOCATE 37,1:PRINT"pillar"
5230 RETURN
5240 :
5300 REM **** your shape title ****
5310 INK 1,11
5320 LOCATE 35,1:PRINT"Your Shape"
5330 RETURN
```

Rotating figures

It is often useful in animated games to enable shapes to rotate, often under a player's control. The principle of the technique is straightforward, although BASIC is a little too slow to achieve a smooth rotating effect. The basis of the technique is to transform each pair of coordinates that define the shape to be rotated, by applying a pair of formulae. If (x, y) are the original coordinates of a point in the shape and (x', y') are the coordinates of the point after rotation, then the following assignments generate the new point position from the old:

```
x'=x*COS(angle)-y*SIN(angle)
y'=x*SIN(angle)+y*COS(angle)
```

An alternative way of writing these equations is as a transformation matrix:

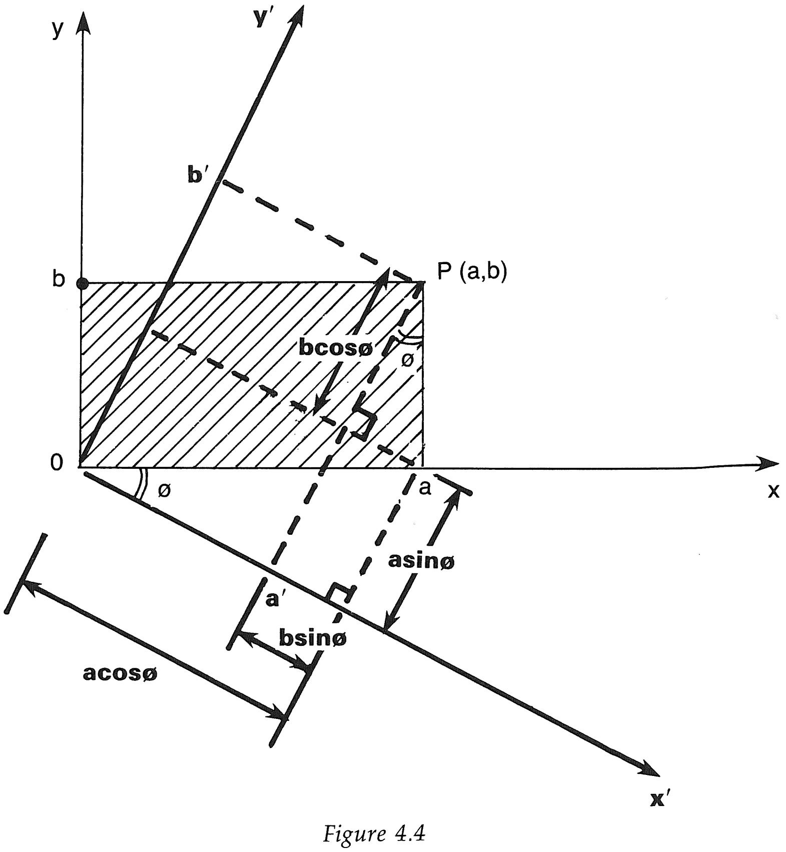


Figure 4.4

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\text{angle} & -\sin(\text{angle}) \\ \sin(\text{angle}) & \cos(\text{angle}) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

The variable 'angle' is the angle of rotation of the shape about the origin, to the horizontal. Although you do not need to understand the mathematics that underlie this pair of equations some readers may be interested to know how they are derived.

If we think of a point in the original x–y plane, P, with coordinates (a, b), and rotate the coordinate plane through ø degrees to form a new plane x'–y', then the new coordinates of P will be (a', b') as marked on the diagram. By using simple trigonometry on the right-angled triangles marked we find that:

$$a' = a\cos(ø) - b\sin(ø)$$
$$b' = a\sin(ø) + b\cos(ø)$$

The following program demonstrates the rotation principle using a simple arrowhead shape. The original coordinates of the four corners of the shape are held in the arrays x( ) and y( ) and these are used to generate rotated coordinates for any given value of `angle`. The shape looks like this:

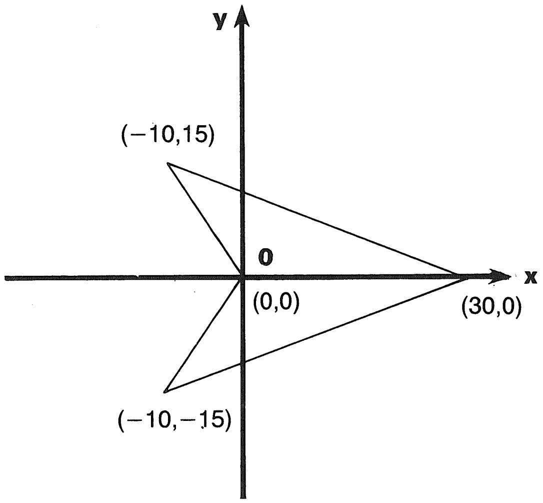


Figure 4.5

When the program is run you can rotate the figure about the point O by pressing the left and right cursor keys. When one of these keys is pressed `angle` is increased or decreased by a set amount and the routine at line `3000` is called to generate a new set of coordinates for the rotated shape. Note that the new position of point O, (0,0), is not rotated, as this will remain (0,0) no matter what the angle of rotation. The subroutine that actually draws the shape from the coordinates generated by this routine, at line `5000`, is called twice, once to erase the old position and again to draw the new shape once its coordinates have been calculated. One particular feature of the Amstrad CPC range that makes it easy to rotate shapes in BASIC is their ability to move the graphics origin. This is normally in the bottom-left corner of the screen but can be moved anywhere, using the `ORIGIN` command. The origin is moved to the middle of the screen during the initialisation routine, but we can use this feature to allow us to move the shape around the screen, whilst still allowing easy rotation of the figure. The routine at line `4000` does this. To make the arrowhead shape move in the direction it is pointing is very simple. We simply have to move the origin.

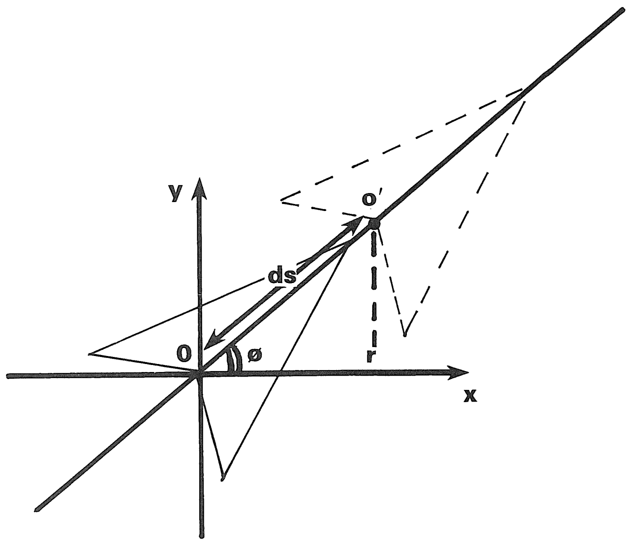


Figure 4.6

To alter the coordinates of the origin so that the arrowhead moves 'ds' units along the line it points, the coordinates of the old origin O must be increased by ds*COS(ø) and ds*SIN(ø).

```
10 REM **** rotating figures ****
20 GOSUB 1000
30 colour=cyan:GOSUB 5000:REM draw shape

40 WHILE pigsfly=0
50 GOSUB 2000:REM key scan
60 WEND
70 END
80 :
1000 REM **** initialise ****
1010 MODE 1:DEG
1020 blue=0:cyan=2
1030 cx=320:cy=200:ORIGIN cx,cy
1040 da=10:ds=10:REM angle and movement
steps
1050 FOR i=1 TO 3
1060 READ x(i),y(i)
1070 a(i)=x(i):b(i)=y(i)
1080 NEXT i
1090 DATA -10,-15,30,0,-10,15
1100 RETURN
1110 :
2000 REM **** scan key ****
2010 IF INKEY(0)=0 THEN GOSUB 4000:REM m
ove forward
2020 IF INKEY(8)=0 THEN angle=angle+da:G
OSUB 3000:REM rotate
2030 IF INKEY(1)=0 THEN angle=angle-da:G
OSUB 3000:REM rotate
2040 RETURN
2050 :
3000 REM **** rotate ****
3010 colour=blue:GOSUB 5000:REM draw sha
pe
3020 ca=COS(angle):sa=SIN(angle)
3030 FOR i=1 TO 3
3040 a(i)=x(i)*ca-y(i)*sa
3050 b(i)=x(i)*sa+y(i)*ca
3060 NEXT i
3070 colour=cyan:GOSUB 5000:REM draw sha
pe
```

```
3080 RETURN
3090 :
4000 REM **** move forward ****
4010 colour=blue:GOSUB 5000:REM draw sha
pe
4020 cx=cx+ds*COS(angle)
4030 cy=cy+ds*SIN(angle)
4040 ORIGIN cx,cy:REM new origin
4050 colour=cyan:GOSUB 5000:REM draw sha
pe
4060 RETURN
4070 :
5000 REM **** draw shape ****
5010 FOR i=1 TO 3
5020 DRAW a(i),b(i),colour
5030 NEXT i
5040 DRAW 0,0
5050 RETURN
```

Mixing text and graphics

The Amstrad CPC range has, as we have seen, two ways of displaying data on the screen; either by high-resolution plotting and drawing, or by printing characters. Most graphics applications will actually require a mixture of these two methods to produce a composite display. Adding text to a high-resolution display is fairly straightforward. Normally characters are located on the screen in a grid of character cells. However, we can free characters from this rigid positioning system by using the TAG command. TAG allows all subsequent characters to be printed at the graphics cursor rather than at the text cursor, allowing accurate positioning of labels on bar and pie charts. However, TAG can also be used to move characters around a pixel at a time. This simple program shows how TAG can be used to move the Q character smoothly across the screen:

```
    10 REM **** TAG DEMO #1 ****
    20 MODE 1
    30 TAG
    40 FOR X=0 TO 600 STEP 2
    50 MOVE X,200
```

```
60 PRINT "Q";
70 NEXT X
80 TAGOFF
```

If you run this program you will see the Q move across the screen, but you will also see one of the problems associated with single-pixel movement of characters. The Q leaves a stream of colour in its wake. As the Q is printed 1 pixel to the right of its previous position most of the old Q is erased by the new; all that is except the most left-hand column of pixels in the old Q. The stream of colour left behind the Q as it moves across the screen is in fact a number of these left-over columns that never get overprinted. One solution would be to print a blank in place of the old Q before moving and printing a new Q. Alternatively, we can erase the offending column by drawing over it in the background colour. This short program demonstrates both methods.

```
10 REM **** pixel move of chars ****
20 MODE 1
30 WHILE ans$<>"h" AND ans$<>"b"
40 INPUT"high res or blank (h/b)";ans$
50 WEND
60 TAG
70 FOR x=1 TO 600 STEP 2
80 IF ans$="h" THEN GOSUB 1000 ELSE GOSU
B 2000
90 NEXT x
100 TAGOFF
110 END
120 :
1000 REM **** move by high res rubout **
**
1010 MOVE x-2,200:DRAWR 0,-16,0
1020 PLOT x,200,1
1030 PRINT "Q";
1040 RETURN
1050 :
2000 REM **** move by char blank ****
2010 MOVE x-16,100
2020 PRINT" ";
2030 MOVE x,100
2040 PRINT"Q";
2050 RETURN
```

When designing custom-made characters for single-pixel movement, leaving a 1-pixel border around the shape eliminates this problem. Several other points are worthy of note when using TAG. Firstly, the character is positioned so that the graphics cursor is in the top-left corner of the character cell; secondly, a semicolon (;) is required after a tagged PRINT statement to suppress the carriage return and line feed control characters (try leaving off the semicolon in either of the two examples above to see these!) and, thirdly, the character or text string will be printed in the current high-resolution colour, not the current text colour.

Frame synchronisation

If you run the above example program you will see that the Q character does not move smoothly across the screen. Using the high-resolution erase method, the left-hand side of the figure appears to flicker and tear and the second method, using a space character to erase the old Q before printing the new, is jerky. Both of these problems are due to the fact that the picture we see on the monitor is not continuous, as it appears, but is a sequence of 'frames', built up by a rapidly scanning electron beam. If the alterations and movements on the screen are not synchronised to these video frames then we get the flicker effects witnessed above.

Fortunately, there is a way around this problem. By making a call to the part of the operating system that controls the video display we can make our program synchronise its drawing actions to frames produced by the video display. Inserting this line into the program above, makes this call:

```
65 CALL &BD19
```

On the Amstrad CPC 664 this operating system call has been incorporated into the BASIC in the form of the FRAME command. On the Amstrad CPC 664 the equivalent is therefore:

```
65 FRAME
```

Both these commands, when inserted into the main animation loop, cause the program to wait until the current video frame has been constructed (a very quick process) before allowing the program to continue and update the information on the screen.

Converting between graphics and character coordinates

Even though we can use TAG to combine small amounts of text with high-resolution pictures there will still be a time when it becomes useful to relate graphics coordinates to character coordinates. Indeed, if you write a program involving animated character coordinate graphics the only way to test for collisions with other objects is to convert the character coordinates into high-resolution graphics coordinates and use the TEST or TESTR commands to test the colour of a point in the character square ahead. The relationship between character coordinates and graphics coordinates depends on the screen display mode being used. Taking these modes one at a time, and starting with mode 2, the highest resolution mode, we have this situation:

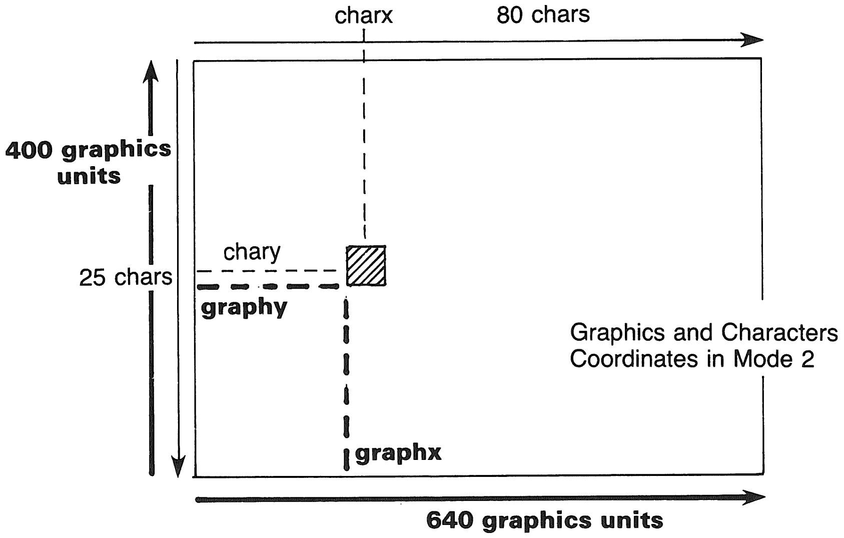


Figure 4.7

Looking at the horizontal coordinates first it is easy to see that each character cell must be 8 graphics units wide. As each cell is itself made up of an 8 × 8 grid of pixels, there is a simple relationship in the horizontal direction:

1 pixel = 1 graphics unit

Thus to find the horizontal graphics coordinate of the left-hand end of a character cell:

```
graph x = 8*(charx-1)
```

In the vertical direction things aren't quite so straightforward. From the diagram we can see that each character cell is 16 graphics units high. In other words, in the y direction:

1 pixel = 2 graphics units

To make life more complex still the two systems work in opposite directions. Whilst `chary` starts at 1, at the top of the screen and increases downwards to 25, `graphy` starts at 0 and the bottom of the screen and increases upwards to 399 at the top. To find the graphics coordinate of the bottom of a character cell:

```
graphy = 399-16*chary
```

In mode 1 and mode 0 the vertical relationships remain the same but the horizontal resolution is halved for mode 1 and halved again for mode 0. In mode 1, therefore, each character cell is 16 graphics units wide. This means that

1 pixel = 2 graphics units

and

```
graphx = 16*(charx-1)
```

In mode 0:

1 pixel = 4 graphics units

and

```
graphx = 32*(charx-1)
```

In general, we can say that to find the graphics coordinates (`graphx,graphy`) of the bottom-left corner of a character cell (`charx,chary`) in mode 'm':

```
graphy = 399-16*chary
```

and

```
graphx = 64/(2↑(m+1))*(charx-1)
```

The problem of relating graphics coordinates to actual pixels on the screen can be demonstrated by thinking of the pixel turned on by the command `PLOT Ø,Ø` in any mode.

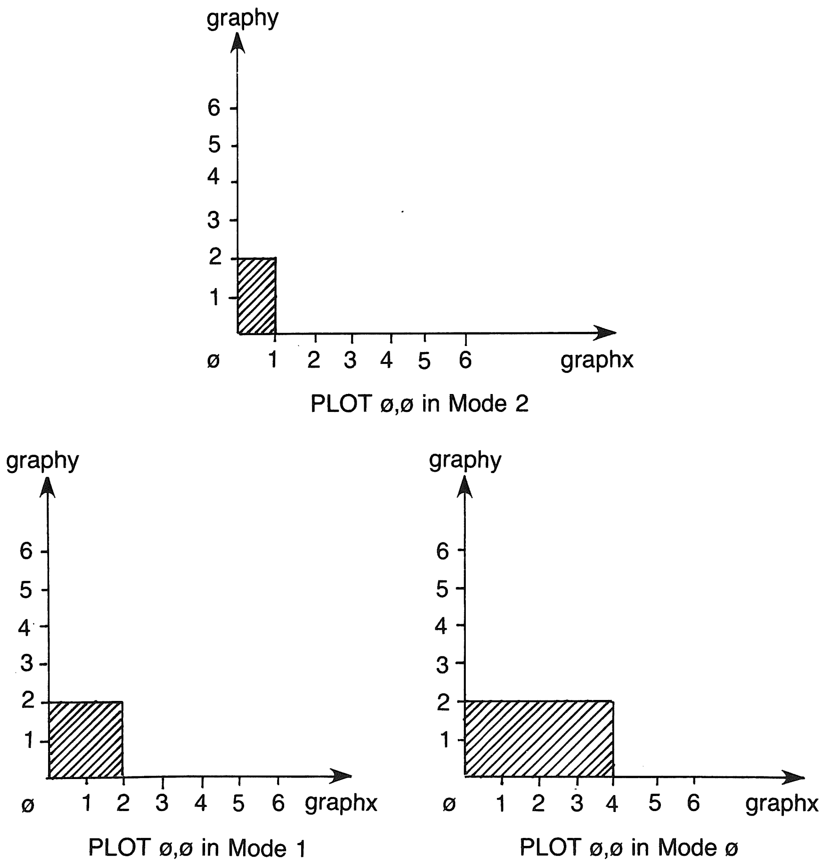


Figure 4.8

The size of the pixel plotted depends on the mode used, but the graphics coordinates system stays the same in all three modes. This means that some plotting instructions can become redundant. For example, in mode 1, `PLOT 1,1`, `PLOT Ø,1` and `PLOT 1,Ø` will all light the same pixel as `PLOT Ø,Ø`, and in mode 0, seven other plotted coordinates would light the same pixel as `PLOT Ø,Ø`. When using high-resolution graphics or printing with `TAG` it is well worth being aware of this fact, as programs can be speeded up substantially by removing redundant `PLOT` instructions. This can be illustrated by a simple program to draw a horizontal line across the screen by plotting a number of individual points in a row. Using mode 0, the program draws the line by plotting points at every possible x coordinate between 0 and 639. The process is then repeated but this time points are plotted at every fourth x coordinate. The result is not, as you might expect a dotted line. All we have done is to remove the redundant PLOT commands, but are still lighting every pixel that makes up the line. In mode 1 we would need to plot points at every second coordinate to get a continuous line. To draw a corresponding vertical line in the most efficient way we should plot points at every second y coordinate irrespective of the mode used.

```
10 REM **** redundant plot demo ****
20 MODE 0
30 start1=TIME
40 s=1:y=200:GOSUB 1000:REM plot line
50 interval1=TIME-start1
60 start2=TIME
70 s=4:y=100:GOSUB 1000:REM plot line
80 interval2=TIME-start2
90 factor=ROUND(interval1/interval2,2)
100 PRINT"second method is";factor;"time
s faster"
110 END
120 :
1000 REM **** plot line ****
1010 FOR x=0 TO 639 STEP s
1020 PLOT x,y
1030 NEXT x
1040 RETURN
```

By irradicating redundancies like this from programs we can get them to run roughly twice or even four times as fast, depending on the mode used.

High-resolution graphics can be a lot of fun to play with as well as allowing us to write good-looking programs. The Amstrad CPC range have excellent ranges of commands to make the task of programming high-resolution displays as easy as possible.

664 graphics enhancements

The version of Locomotive BASIC provided with the Amstrad CPC 664, BASIC 1.1, has several extra graphics commands to complement the excellent range of commands available on the Amstrad CPC 464. We have already met the FRAME command, used for frame flyback synchronisation, earlier in this chapter and its equivalent operating system call on the Amstrad CPC 464. Now let us look at the Amstrad CPC 664's commands that allow us to fill areas of the screen with colour and draw dotted or dashed lines.

The FILL command

This new command can prove extremely useful for colouring areas of the screen simply and quickly. The command is straightforward; we simply issue the FILL command together with a logical mode colour number to specify the colour we wish to use. The area of the screen filled depends on two things: where the graphics cursor was when you issued the FILL command and the other information present on the screen. The fill area is bounded by lines drawn in the current LMCN or the LMCN used by the FILL command. This short program demonstrates how the command can be used:

```
10   REM **** FILL DEMO ****
20   MODE 0:DEG:red=3:pink=11
30   side=250:height=side*SIN(60)
40   MOVE 200,150
50   GOSUB 1000:REM DRAW TRIANGLE
60   MOVER 0,2*height/
     3:height=-height
70   GOSUB 1000:REM SECOND TRIANGLE
80   REM **** NOW FILL OUTSIDE ****
90   savx=XPOS:savy=YPOS:REM SAVE
     CURRENT CURSOR POS
100  MOVE 0,0:FILL pink
110  WHILE INKEY$="":WEND:REM WAITFOR
     KEYPRESS
120  REM **** NOW FILL INSIDE SECTIONS
     ****
```

```
130  MOVE savx,savy
140  FOR area=1 TO 7
150  REM ** POSITION CURSOR READY FOR
     FILL **
160  READ x,y:MOVER x,y
170  FILL area
180  NEXT area
190  END
1000 REM **** DRAW TRIANGLE S/R ****
1010 DRAWR side,0,red
1020 DRAWR -side/2,height
1030 DRAWR -side/2,-height
1040 RETURN
2000 REM **** CURSOR POS DATA ****
2010 DATA 40,-30,0,-80,90,-80
2020 DATA 80,80,0,80,-90,80,0,-100
```

This demonstration draws a six-pointed star and then uses the `FILL` command to fill the outside in pink and then the seven sections that go to make up the star in various colours. The loop that accomplishes this second part firstly reads from the `DATA` statements in lines `2000–2020` in order to position the cursor within the area to be filled. When the fill is done, the area coloured is limited by the red lines that define the star. The current graphics colour in the program is red, selected at line `1010` as part of a `DRAWR` instruction and, as no other instruction in the program changes the current graphics colour, then the `FILL` operation 'recognises' the red lines as boundary markers. It is worth noting that crazy things can happen with the `FILL` command if you inadvertently change the current foreground colour between drawing a shape and filling it. Re-run the demonstration program but insert this line which changes the current graphics colour to green:

```
135 PLOT savx,savy,13
```

Subsequent filling will no longer recognise the red lines used to draw the star as boundaries and will simply fill over them. Try changing the colour used to draw the star to pink on line `1010`. Can you explain the effect this time?

Dotted lines

The addition of the `MASK` command to the Amstrad CPC 664's BASIC makes broken lines easy to draw. The `MASK` command is used to define which, out of any group of eight, pixels you wish to show in the foreground colour and which in the background colour by using a number in the range 0–255. If you imagine the number to be in binary then the bits set to 1 represent pixels in the foreground colour and those set to 0 represent pixels in the background colour. Thus `MASK 15` followed by a `DRAW` command would produce a half-broken line, as 15 is 00001111 in binary. Normally, the foreground colour is the current foreground colour and the background colour that of the screen, but these can be changed using `GRAPHICS PEN` and `GRAPHICS PAPER`. The first of these commands is an alternative way of changing the current graphics foreground colour and can be used instead of adding a colour parameter to a graphics command as described earlier. The second sets the background graphics colour in the same way that `PAPER` sets the background text colour. Thus, if `GRAPHICS PAPER 2` is selected in mode 1, the zeros in the mask will appear as cyan dashes in the broken line.

'Stranded' — ladders and pillars

In the last two chapters we have built up the part of the screen display that consists of low-resolution character graphics, windows, and introduced the joystick and cursor key control routines. In this chapter we add some of the high-resolution graphics detail to the screen display. This section is mainly concerned with drawing ladders between the various cliff-walk levels and drawing the pillars that rise out of the sea on which the stranded figures stand. The following figure shows these two features added to the screen display. In addition a jetty has been added by introducing a third window and setting its background colour to mauve. In the final version of the game the player will negotiate the cliff-walk to reach the jetty from where he will board the boat to rescue the stranded figures.

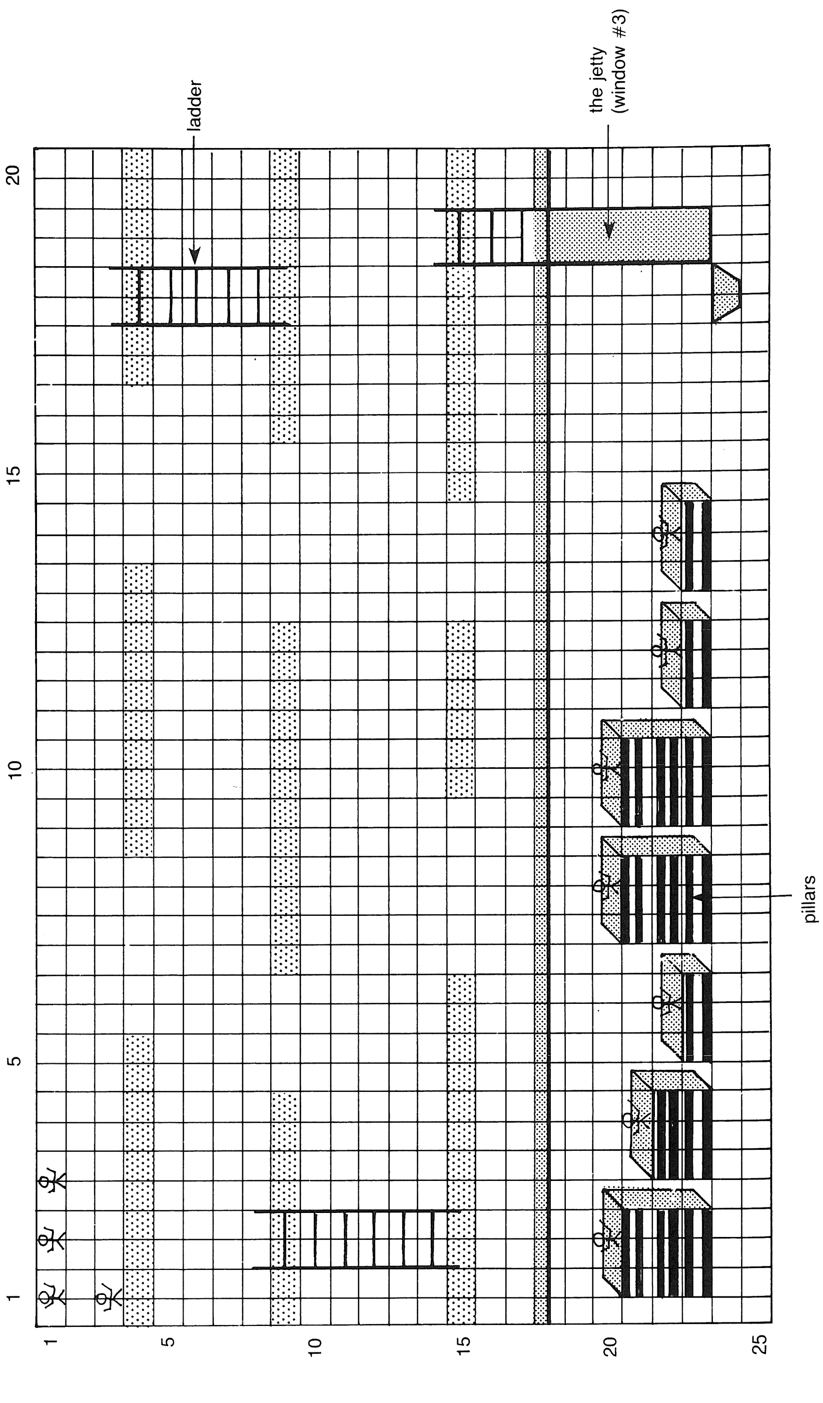


Figure 4.9

Drawing the ladders

As the game uses character-cell animation techniques (that is, the moving figure moves one character cell at a time using LOCATE) we must relate the high-resolution graphics to the low-resolution character-cell coordinate system. In this chapter we have developed simple relationships that allow us to convert character-cell coordinates to high-resolution coordinates. The ladders positions can therefore be defined using character coordinates, and a short conversion routine introduced to generate the equivalent high-resolution coordinates needed to draw the ladders. In order to draw the ladders we need three pieces of information (assuming they are all to be the same width!) — its x and y coordinates and its height. As we have three ladders to draw we can use a series of arrays to hold the information for each ladder. We should define these arrays at the beginning of the program:

```
1100 REM ********************
1110 REM * dimension arrays *
1120 REM ********************
1130 :
1190 REM ** dimension ladder arrays **
1200 DIM lx(3),ly(3),ll(3),glx(3),gly(3)
,gll(3),deck(3)
```

We can define the position and length of each ladder by READing DATA into these arrays. The arrays `lx()`, `ly()` and `ll()` hold the data in character-cell coordinate form; `glx()`, `gly()` and `gll()` hold the corresponding high-resolution graphics conversions. We can set the values in these arrays as part of our initialisation subroutine.

```
2240 REM ** ladder data **
2250 FOR i=1 TO 3
2260 READ lx(i),ly(i),ll(i)
2270 glx(i)=32*(lx(i)-1):gly(i)=399-16*(
ly(i)-1)-8
2280 gll(i)=(ll(i)+1)*16
2290 NEXT i
2300 :
2310 REM ** ladder data **
2320 DATA 18,9,5,2,15,6,19,18,3
```

This data can then be used to draw the ladders. The starting point as defined by `glx()` and `gly()` is the bottom of the left upright. The drawing routine uses a `MOVE` command to move to this start point but the rest of the ladder is drawn using the relative commands `MOVER` and `DRAWR`. The following diagram shows how the ladder can be constructed from the array data.

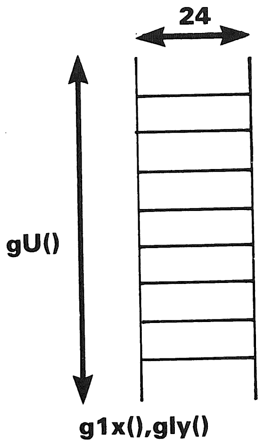


Figure 4.10

This pair of subroutines actually draw the three ladders:

```
6400 REM **** draw ladders ****
6410 FOR lad=1 TO 3:GOSUB 6500:NEXT lad
6420 RETURN
6430 :
6500 REM **** draw a ladder ****
6505 PEN deck(lad):LOCATE lx(lad),ly(lad
)-ll(lad):PRINT LEFT$(deck$,1)
6507 PEN mancol
6510 MOVE glx(lad),gly(lad)
6520 DRAWR 0,gll(lad),black
6530 MOVER 28,0
6540 DRAWR 0,-gll(lad)
6550 FOR ly=8 TO gll(lad)-8 STEP 8
6560 MOVER -28,8
6570 DRAWR 28,0
6580 NEXT ly
6590 RETURN
6600 :
```

Lines 6520–6540 draw the two ladder uprights and the FOR...NEXT loop in lines 6550–6580 puts in the appropriate number of rungs. Notice that a separate subroutine draws a ladder, as defined by the number in the variable lad whilst a second subroutine is used to call the drawing routine three times using lad values of 1, 2 and 3 to draw the three ladders. An alternative method would have been to just enclose the drawing routine inside a FOR-...NEXT loop, within a single subroutine. The reason for not doing this is that later in the game we shall need to be able to redraw any ladder independently of the other two. To do this we need only to set lad equal to 1, 2 or 3 depending on the ladder we wished to draw and call the drawing subroutine at line 6500. Initially, however, we want to draw all three ladders, so we call the subroutine starting at line 6400.

One advantage of setting up routines that allow us to enter the defining data for each ladder in the character-cell coordinates is that we can play around with our screen display very easily during the design stage of the program. Adding a fourth ladder would not be difficult, neither would shifting the position of the existing three ladders. The system is flexible enough to allow us to experiment with the arrangement of ladders in the game. Indeed, you might like to try experimenting with the data values given in line 2320 to change the existing arrangement of ladders.

Drawing pillars

The same techniques can be applied to drawing the seven pillars that rise out of the sea. Arrays to hold the x coordinate and height of each pillar in both character cell and graphics coordinates are used. As the y coordinate of each pillar is the same, we do not need to hold its value for each pillar. We must therefore also DIMension these arrays at the beginning of the program.

```
1210 REM ** dimension pillar arrays **
1220 DIM pillcx(7),pillch(7),pillarx(7),
pilarh(7)
```

The values that define the arrays are not set by a series of data values. The x coordinate is defined to be 1, 3, 5, etc., for each successive pillar and the height of the pillar is defined randomly between 1 and 4 character-cell units. The routine to set up these values is not included in the initialisation routine as there will be stages during the game when we shall want to redefine and redraw the pillars, for example, when all seven stranded men have been rescued. This routine therefore forms its own individual module. Whenever we wish to redraw the pillar data we will also wish to redraw the jetty. The commands to do this can therefore be included in this routine.

```
2400 REM **** set up pillars ****
2410 RANDOMIZE TIME
2420 cc=1:pillcy=23:pillary=405-16*pillc
y
2430 FOR n=1 TO 7
2440 pillcx(n)=cc:pillch(n)=INT(RND(1)*4
)+1
2450 pillarx(n)=32*pillcx(n)-16
2460 pillarh(n)=16*pillch(n)-8
2470 maxlev(n)=pillary+pillarh(n)+16
2480 cc=cc+2
2490 NEXT n
2500 level=pillary
2510 GOSUB 7500:REM draw walkway
2520 PAPER #3,mauve:CLS #3:PEN #3,white

2530 GOSUB 7000:REM draw pillars
2540 LOCATE #2,18,7:PEN #2,green:PRINT#2
,boat$
2550 RETURN
```

The actual drawing routine to create the pillars is located at line 7000:

```
7000 REM **** draw pillars ****
7010 PEN #2,white
7020 FOR n=1 TO 7
7030 pillcol=red
7040 FOR i=0 TO pillarh(n)/2
7050 MOVE pillarx(n),pillary+2*i
7060 DRAWR 48,0,pillcol
```

```
7070 IF pillcol=red THEN pillcol=white E
LSE pillcol=red
7080 SOUND 4,10*i,2,5
7090 NEXT i
7100 FOR j=1 TO 5
7110 MOVE pillarx(n)+48+2*j,pillary+j

7120 DRAWR 0,pillarh(n),pink
7130 DRAWR -48,0,pastgreen
7140 NEXT j
7150 PRINT#2,transon$
7160 LOCATE #2,pillcx(n)+1,pillcy-pillch
(n)-17
7170 PRINT#2,standman$
7180 PRINT#2,transoff$
7190 NEXT n
7200 RETURN
```

Each pillar is drawn so that a three-dimensional effect is achieved. The front face is drawn as a series of red and white horizontal bars. Lines 7030–7090 do this. The side and top of the pillar are then drawn in pink and pastel green, respectively, in lines 7100–7140. The final task is to place the stranded man figure on the top of the pillar. This could be done using TAG, but in keeping with the philosophy adopted by the rest of the program to date, namely to relate all graphics to the character-cell coordinate system, the positioning of the man is done using character-cell coordinates. The y coordinate of the man on top of the pillar demonstrates one difficulty in using windows. `pillcy` and `pillch()` are both character-cell coordinates based on the assumption that the cell (1,1) is in the top-left corner of the screen, but, as we are now printing in window 2, this is no longer the case. In this window, the cell (1,1) is in the top-left corner of the window, not the entire screen. In calculating the y coordinate of the man we wish to print in window 2, we must take account of this by subtracting 17 from the value arrived at from subtracting `pillch()` from `pillcy`. The use of `transon$` and `transoff$` switches on and off the transparent printing option that is described in detail in Chapter 7. Its use in this case is to stop the top of the pillar being obliterated when the man character is printed over it.

The horizontal detail of the jetty is filled in by this short routine:

```
7500 REM draw walkway ****
7510 FOR y= 118 TO 120 STEP 2
7520 MOVE 0,y:DRAW 639,y,black
7530 NEXT y
7540 FOR y=31 TO 27 STEP -2
7550 MOVE 576,y:DRAW 608,y,black
7560 NEXT y
7570 DRAW 608,118
7580 FOR y=122 TO 126 STEP 2
7590 MOVE 0,y:DRAW 639,y,mauve
7600 NEXT y
7610 RETURN
```

All that remains is to insert calling commands into the relevant parts of the program. The subroutine to draw the foreground scene is as follows:

```
2800 REM **** draw foreground etc ****
2820 GOSUB 2400:REM draw sea scene
2830 GOSUB 6400:REM draw ladders
2860 RETURN
```

This routine must, in turn, be called from the main program:

```
1330 GOSUB 2800:REM foreground etc
```

Program structure

We can see how these additional program sections fit into the program, as so far devised, from the additions made to the program structure diagram. The program is now at the stage where most of the preparatory routines have been completed. In the next chapter we shall insert the routines to create moving panels between the cliff-walk sections and control routines that allow the player to ascend or descend the ladders.

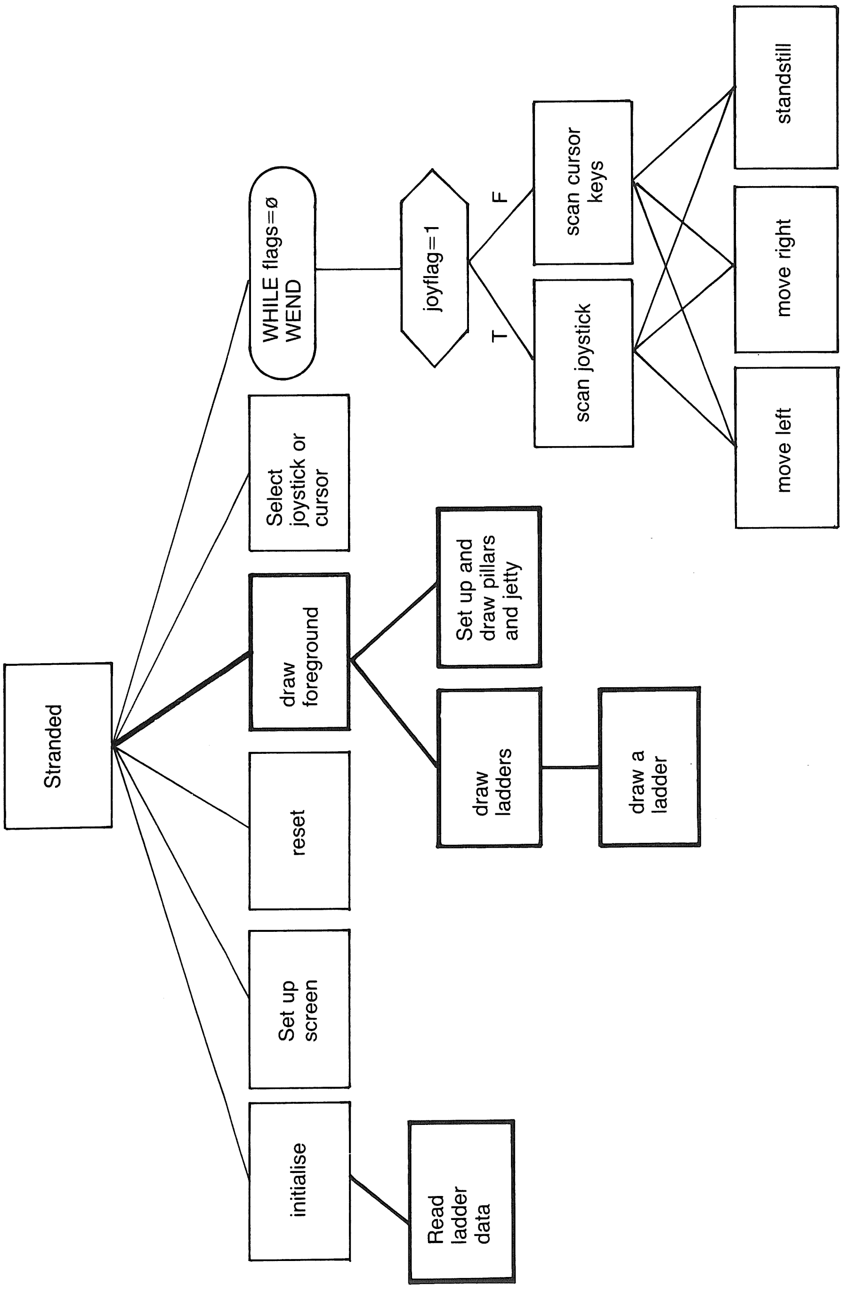

Stranded
initialise
Read ladder data
Set up screen
reset
draw foreground
draw ladders
draw a ladder
Set up and draw pillars and jetty
Select joystick or cursor
WHILE flags=ø WEND
joyflag=1
T
F
scan joystick
scan cursor keys
move left
move right
standstill

Figure 4.11

Chapter Five

Interrupts

In this chapter:

On many computers the use of interrupts as a way of controlling events within a program has been solely the domain of the machine-code programmer. The Amstrad CPC range, however, allows BASIC programmers to use interrupts. Even if you are an experienced BASIC programmer you may not be familiar with the concept of interrupts. This chapter introduces programming with interrupts and deals with some of the common pitfalls to be wary of when using them within your own programs.

What is an interrupt?

Normally, when the processor runs a BASIC program it follows the program, line by line, converting the line into machine code and then executing it; a process known as 'interpreting' the BASIC program. The lines are interpreted in the order determined by the program, the processor following the loops and branches that make up the program's structure. This process continues until the end of the program is reached or there is an error in the program. An interrupt can best be described by a simple analogy. If we think of the processor running the BASIC program as a person reading a book, then an interrupt can be thought of as a telephone call. Under normal circumstances the person will break off from reading the book to answer the telephone. When the telephone conversation is finished the person will return to carry on reading the book from the point that was reached before the interruption caused by the telephone call.

An interrupt causes the processor to break off from its main task of interpreting the main BASIC program to do another short job before carrying on with the main program from the point it was at before the interrupt occurred. Interrupts were first developed so that pieces of hardware, such as printers and disk units, could interrupt the processor when they wanted to make a data transfer, or so that the operating system could perform some immediately necessary task. However, BASIC interrupts on the Amstrad CPC range allow us to do short jobs within a program either at regular intervals or after a certain amount of time has elapsed. Because of the sequential (one line after another) way in which BASIC programs are normally interpreted it is difficult to predict exactly when any one section of program will be completed. Interrupts allow us to do certain jobs at precise times because they interrupt the normal program flow in order to break off and do the job we require.

EVERY and AFTER

There are two commands that will generate interrupts in BASIC, EVERY and AFTER. The first allows us to do a short task repeatedly at regular intervals; the second allows us to do a certain job after a certain amount of time has elapsed. The task we want to see done when an interrupt occurs must be held in a subroutine. The syntax of these two commands is very simple:

```
EVERY <time interval>,<timer> GOSUB <line
      number>
AFTER <time interval>,<timer> GOSUB <line
      number>
```

This short demonstration shows how each type of interrupt can be used. The main program being executed is the `WHILE...WEND` loop in lines `50–80`. This loop prints a counter value to the screen. The first subroutine at line `1000` randomly reselects the `PEN` colour and the command at line `30` makes this event occur every 100 time units (each time unit is equal to 1/50th of a second). The second subroutine simply sets a flag, `endflag`, that will terminate the `WHILE...WEND` loop. The `AFTER` command in line `40` sets this to occur 1000 time units after the `AFTER` command was issued.

```
10 REM **** interrupts ****
20 MODE 0
30 EVERY 100,0 GOSUB 1000
40 AFTER 1000,1 GOSUB 2000
50 WHILE endflag=0
60 cc=cc+1
70 LOCATE 10,12:PRINT cc
80 WEND
90 END
100 :
1000 REM **** interrupt #1 ****
1010 RANDOMIZE -TIME
1020 pencol=INT(RND(1)*15)
1030 PEN pencol
1040 RETURN
1050 :
2000 REM **** interrupt #2 ****
2010 endflag=1
2020 RETURN
```

Notice that the `EVERY` command and `AFTER` command in this program use different timers. The `EVERY` command uses timer 0, whereas the `AFTER` command uses timer 1. Each interrupt we wish to operate within a program must have its own timer. There are in fact four timers, numbered 0–3, for this purpose. If we alter the timer value in line `40` by entering this line:

```
40 AFTER 1000,0 GOSUB 2000
```

and run the program we can see why we require different timers. You will notice that the PEN colour is never reset during the program run. This is because the AFTER command has grabbed timer 0 to time its own interrupt, effectively stopping the EVERY command that generates the PEN colour changes from having a timer. We can therefore only ever have four interrupt routines current at any one time during a program using the timers 0, 1, 2 and 3.

We have now seen how to turn an interrupt on, but how can we turn it off again? There are two ways of doing this. One, as we have just seen, is to pinch the timer for use by another interrupt we want to bring into effect. In other words for each timer a new AFTER or EVERY command overrides any previous EVERY or AFTER that was using that timer. The second method is to use the REMAIN command. This command actually has two functions. Firstly, it returns the number of time units left until the next interrupt is due to be generated, and, secondly, it disables the timer in question, stopping any further interrupts from occurring. We can see how REMAIN can be used to disable a timer by adding a line to the last demonstration program:

```
75 IF cc>200 THEN dummy=REMAIN(0)
```

If the program is run with this line inserted then the interrupt timed by timer 0, the PEN changing subroutine, will be disabled when cc exceeds 200; the PEN colour current at this time remaining in force until the end of the program. Notice that to use REMAIN in this way we need to set it equal to a variable. In this application we are not interested in the number of time units that remain before timer 0 generates another interrupt, so the variable we use is simply a dummy variable to achieve the correct syntax.

Interrupt priorities

Just as an interrupt can cause the processor to break off from processing the main program to do another task, so an interrupt can itself be interrupted by another interrupt. If we go back to our book and telephone call analogy, we can think of the four interrupt timers as being four telephones on a desk. The telephones are such that a person sitting at the desk knows that calls on one telephone are more important than calls on another telephone; indeed, that there is a set order of priorities between the four telephones. The person (the processor) is reading the book (interpreting the main program) when one of the telephones rings (an interrupt occurs). He puts the book down and answers the telephone. While he is still on the telephone another, more important, telephone rings (a second interrupt occurs). The person asks the caller on the first telephone to hold on while he answers the second telephone. Eventually the call on the second telephone will end (second interrupt task completed), the person will then go back to his previous telephone conversation and finish that (return to first interrupt task and finish that), finally returning to the book he was reading before the telephones started ringing (back to processing the main program).

It is important to realise that if the second telephone had been less important than the first then the person would not have broken off his first conversation to answer it. In the Amstrad CPC range's BASIC interrupt system the four telephones are the four interrupt timers. Timer 3 has the highest priority and timer 0 has the lowest priority, although all these timers have priority over the normal BASIC program being run. We can see the effect of priorities using a simple program to generate several interrupts to occur simultaneously. Notice that in this example there is no real activity in the main program. The main program consists of an empty loop cycling around at line 6Ø. In this case we have a completely interrupt-driven program. The interrupt timings are set to generate interrupts at intervals of 50, 100 and 200 time units. After 200 time units have elapsed, all three interrupts will be generated simultaneously. In this case the priority is made clear. The interrupt on timer 2 is handled first, then the interrupt on timer 1 and, finally, the interrupt on timer 0.

```
10 REM **** interrupt priorities ****
20 MODE 1
30 EVERY 50,0 GOSUB 1000
40 EVERY 100,1 GOSUB 2000
```

```
50 EVERY 200,2 GOSUB 3000
60 GOTO 60
70 :
1000 REM ***** interrupt #0 ****
1010 PEN 1:PRINT TAB(10)"timer 0"
1020 PRINT
1030 RETURN
1040 :
2000 REM ***** interrupt #1 ****
2010 PEN 2:PRINT TAB(5)"timer 1"
2020 RETURN
2030 :
3000 REM ***** interrupt #2 ****
3010 PEN 3:PRINT"timer 2"
3020 RETURN
```

Programming pitfalls

Although having BASIC interrupts can be of great use to the programmer, they can also cause a lot of headaches when things do not quite go according to plan. It is sometimes difficult to be sure whether the fault is in the main program or in a subroutine that is interrupting the main program, or in the interaction of the main program and the interrupt. There are, however, a number of tips that you can follow to avoid some common errors.

The main source of errors to the unexperienced interrupt programmer is when the interrupt alters variable values or moves the text or graphics cursors that are used by the main program. The first of these problems is easily avoided (once you are aware that the problem exists) by ensuring that you use variable names in your interrupts that are not used in any other place in the program. As Locomotive BASIC allows you to have extended variable names this should be easy! Keeping track of the graphic and text cursors in a program that uses interrupts can be a little more difficult. Here is an example of the sort of (apparently) crazy things that can happen:

```
10 REM **** interrupt problems ****
20 GOSUB 500:REM initialise
30 REM **** plot lines ****
```

```
40 FOR y=199 TO 399 STEP 10
50 MOVE 0,y
60 FOR x=0 TO 639 STEP 10
70 DRAW x,y
80 NEXT x,y
90 END
100 :
500 REM **** initialise ****
510 MODE 0
520 cx=30:cy=50:r=20
530 EVERY 85,0 GOSUB 1000:REM draw a cir
cle
540 RETURN
550 :
1000 REM **** circle interrupt ****
1010 RANDOMIZE -TIME
1020 colour=INT(RND(1)*14+1)
1030 MOVE cx+r,cy
1040 FOR angle=0 TO 2*PI STEP 0.2
1050 a=cx+r*COS(angle)
1060 b=cy+r*SIN(angle)
1070 DRAW a,b,colour
1080 NEXT angle
1090 DRAW cx+r,cy
1100 cx=cx+50
1110 RETURN
```

If we surpress the interrupt by inserting a REM at the beginning of line 530 we can see the main action of the program, to draw a series of parallel lines in the top half of the screen. The interrupt subroutine draws a circle in the bottom half of the screen in a randomly selected colour. The idea is to draw a series of circles whilst drawing the parallel lines pattern. Remove the REM from line 530 to enable the interrupt to occur and run the program to see what goes wrong. The first problem is that when the program returns from a circle-drawing interrupt the graphic cursor is not in the correct position to continue drawing the parallel lines and so spurious lines connecting the circle with the lines pattern are drawn. The second problem is that the circle-drawing interrupt subroutine alters the graphics colour each time it is called, this means that when the program returns to its line-pattern task it continues drawing in the wrong colour. What steps can we take to eliminate these problems? The problem with the graphics cursor can be rectified by storing away its current position on entering the interrupt and restoring it on exit, using XPOS and YPOS (see Chapter 4). Insert these lines to do this:

```
1005 savex=XPOS:savey=YPOS
1105 MOVE savex,savey
```

Text-cursor problems encountered when printing text in a program that uses interrupts can be overcome similarly, using POS and VPOS. These variables carry the current horizontal and vertical text-cursor coordinates.

The colour problem can be overcome by adding a third parameter to the DRAW command used to create the parallel lines pattern, to specify the graphics colour to be used every time the DRAW command is executed. To draw the lines pattern in cyan, we need simply to change line 70:

```
70 DRAW x,y,2
```

One interesting effect that can be noticed from making this change is that a DRAW command with the third colour parameter added takes slightly longer to execute. This can be seen in the fact that the program has time to generate two extra circles when line 70 is amended as above. However, we have achieved our aim, to draw a series of parallel lines in one colour and generate a number of multi-coloured circles on an interrupt basis.

Interrupts and timing

Another source of problems when dealing with interrupts is that of timing. It is difficult to predict how long a certain interrupt subroutine will take to execute. Often relative timings can only be worked out by trial and error, experimenting with different interrupt intervals until the program 'feels' right. The golden rule with interrupts is to try and keep them short in comparison with the number of times they occur. If we shorten the interval between interrupts in the last example program to 50 time units, we see one consequence of long interrupts — the processor can never get back to deal with the main program. The time taken to draw a circle is relatively long, and by the time the task is finished it is time for another interrupt to occur. The program keeps drawing circles, but never gets back to its main task of drawing the lines pattern. If an interrupt routine takes so long to execute that another interrupt occurs then one of two things will happen: if the new interrupt is of a higher priority then the processor will switch to the new interrupt routine; if the new interrupt has a lower priority then it will 'stack up' waiting for the higher priority interrupt routines to be completed. If several interrupts are running at the same time, then their relative timings can be difficult to work out. Again the rule is *keep interrupts short!*

Stopping interrupts happening

It can often be useful to suspend temporarily all interrupts whilst doing a particular section of program. This may be to avoid some of the problems discussed earlier, or because an interrupt is inappropriate at that time, but you do not wish to disable any interrupt permanently. To disable all interrupts temporarily we use the DI command. Interrupts are re-enabled by the EI command, the section of code we wish to be processed uninterrupted being sandwiched between DI and EI. These commands can also be used to stop a low-priority interrupt routine being interrupted by a higher-priority interrupt. In this case interrupts are automatically re-enabled, without the need for an EI, when the interrupt routine reaches the RETURN command. As far as possible, sections of program that are protected from interruption by DI should be short, as any interrupts that occur during the time when interrupts are disabled are not ignored but stack up awaiting service when the EI (or RETURN) command is reached. If the length of time when interrupts are disabled is too long then the program will have a queue of other interrupts to process when they are eventually re-enabled.

Uses of BASIC interrupts

Interrupts, provided they are used carefully, are of great use in programming arcade games. Any process that requires regular attention, or that must occur after a certain amount of time, is a prime candidate for use as an interrupt. Scanning the joystick or keyboard could be done under interrupts as could the update of shape positions in a 'shoot-em-up' space-invaders type game. The AFTER command might be used as count-down timer to the end of the game, or for switching from one phase of the game to the next. With a little fine-tuning interrupts can make movement and control within games smoother and more effective. This chapter has attempted to cover some of the difficulties of real-time interrupt programming, but there is no universal panacea. The techniques required to get interrupt-driven programs debugged and running smoothly must be learned through experience.

'Stranded' — moving panels and rising tides

In Chapter 4 we added most of the foreground detail to 'Stranded'. However, one important feature was not included — the moving panels between each section of the cliff-walk. These form the major hazard to the player, as he must successfully jump on and off these panels in order to make his way down the cliff to the men stranded on pillars in the sea. Because these panels move back and forth across the gaps between cliff-walk sections it is an ideal application for the BASIC interrupt facilities offered by the Amstrad CPC range. The movement of each panel can be controlled by a regular interrupt using EVERY and one of the four interrupt timers. The program design means that the main program loop repeatedly scans the joystick or cursor keys and updates the player character's position and is interrupted regularly to move the panels. The alternative to this structure is to place a panel-moving routine within the main program loop. One of the advantages of using an interrupt in this instance is that it makes the game more difficult to play. If the panels were moved regularly within the same loop as the player-movement routine then the two motions would be, to some extent, synchronised, allowing the player to hop on and off panels more easily. As the timing of an interrupt routine is independent of the current position within the loop, the two types of motion are not synchronised, making the game less predictable.

The second feature of the game that is a good candidate for use as an interrupt is the routine that raises the sea level around the pillars as the tide comes in. In this section we look at both of these routines, see how they can be driven by interrupts, and how they fit into the game's structure as so far developed.

Drawing the panels

The panels move in the gaps between the cliff-walk sections and will be drawn using high-resolution graphics. As with the other high-resolution routines detailed so far, the position and width of each panel and the gap between the cliff-walk sections will be defined in terms of character-cell coordinates and held in a series of DATA statements. Conversion routines will convert each panel parameter to its graphics coordinates equivalent. We therefore need to dimension a number of arrays to hold these parameters:

```
1140 REM ** dimension panel arrays **
1150 np=6:REM set number of panels
1160 DIM scx(np),scy(np),gcp(np),pcl(np)

1170 DIM pl(np),gp(np),stx(np),sty(np)
1180 DIM ex(np),dx(np),c(np),mc(np)
```

The four parameters that define each panel are read in by this part of the initialisation routine:

```
2090 REM ** panel data **
2100 FOR n=1 TO 6
2110 READ pcl(n),gcp(n),scx(n),scy(n)
2120 pl(n)=32*pcl(n):gp(n)=32*gcp(n)-1
2130 stx(n)=32*(scx(n)-1):sty(n)=399-16*
scy(n)
2140 ex(n)=stx(n)+gp(n)
2150 mc(n)=gp(n)-pl(n)
2160 dx(n)=32
2170 NEXT n
2180 DATA 2,3,6,3,2,3,14,3
```

```
2190 DATA 1,2,5,8,2,3,13,8
2200 DATA 2,3,7,14,1,2,13,14
2210 :
```

The parameters are used to generate several other parameters that will be of use in moving each panel. The first task is to convert the four data values read in for each panel into graphics coordinates. The following diagram shows what data the other arrays that are calculated from the four initial parameters hold:

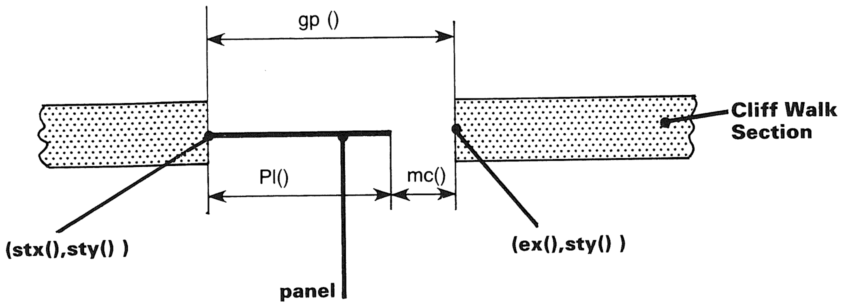


Figure 5.1

`pl()` holds the length of the panel and `gp()` the size of the gap between the cliff-walk sections. `stx()` and `sty()` hold the starting coordinate for the panel and `ex()` holds the x coordinate of the other side of the gap. The following subroutine will therefore move a panel, as designated by the value of the variable `n`.

```
3600 REM **** move panel n ****
3610 DI
3630 MOVE stx(n),sty(n)
3640 c(n)=c(n)+dx(n)
3650 IF c(n)>=mc(n) THEN dx(n)=-dx(n):c(
n)=mc(n)
3660 IF c(n)<=0 THEN dx(n)=-dx(n):c(n)=0
3670 DRAWR c(n),0,sky
3680 DRAWR pl(n),0,black
3690 DRAW ex(n),sty(n),sky
3700 EI
3710 RETURN
```

This routine uses a counter `c()` to keep track of the left-hand end of the panel. Essentially what this routine does is to draw a horizontal line across the gap. The two end sections of the line are the same colour as the sky-blue background but the middle section is black, denoting the panel. Each time the routine is called for panel `n` the counter is increased by the value of `dx(n)`. The three sections of the line are drawn as follows, starting at the left-hand end of the gap:

a) A sky-blue horizontal line is drawn from the start point coordinates, 'c(n)' coordinates long.
b) A horizontal black line is drawn from this point, 'pl(n)' units long.
c) A horizontal sky-blue line is then drawn from the end of the black line to the right-hand end of the gap, as held in `ex(n)`.

As the counter `c(n)` increases so the starting point of the black section of the line will move further and further to the right. To make the panel move back to the left we simply have to change `dx(n)` from a positive to a negative value, so that `c(n)` is decreased each time rather than increased. The checks for the right- and leftmost values of `c(n)` are made in lines 3650 and 3660 before the drawing of the three line sections takes place. As the whole of the gap is drawn in either sky-blue or black, old positions of the panel are automatically erased when the new position is drawn.

This routine only works for panel `n`, we need to repeat the process for each of the six panels in the game, we therefore need a further routine to call this routine with `n` set to 1, 2, 3, 4, 5 and 6:

```
3500 REM **** moving panels ****
3510 REM ** disable and re-enable timers
 **
3520 IF resetflag=1 THEN GOSUB 7300:GOSU
B 7400
3530 n=n+1:IF n>6 THEN n=1
3540 GOSUB 3600
3550 RETURN
3560 :
```

Notice that a `FOR...NEXT` loop is not used here. Instead the routine cycles round from n=1 to n=6, before resetting to 1. This routine is the actual routine that will be called by the interrupt timer. To move all six panels on a single interrupt would take too long, so, instead, this arrangement ensures that every time an interrupt is generated only one panel is moved, but that the moves are made in a sequence, from panel 1 to panel 6.

Raising the water level

Making the sea level appear to rise around the pillars is straightforward. This routine simply draws a blue line around the current base of each pillar, as defined by `level`.

```
6100 REM **** raise water level ****
6110 DI
6120 FOR pn= 1 TO 7
6130 MOVE pillarx(pn),level
6140 DRAWR 48,0,water
6150 FOR px=1 TO 5
6160 PLOTR 2,1
6170 NEXT px
6190 NEXT pn
6200 level=level+2
6240 EI
6250 RETURN
```

Setting the interrupt timers

We shall need two interrupt timers to control these two events. As the intervals between each successive event might be changed later in the program (for example, to make the panels move faster or make the tide come in quicker) the program uses two variables, `moveint` and `riseint`, to set these intervals. These variables are set at the beginning of the program, during the initialisation routine.

```
1820 REM ** set interval timer values **
1830 moveint=12:riseint=300
```

The following routine can then be used to start the interrupt timers going:

```
7400 REM **** enable timers ****
7410 EVERY moveint,0 GOSUB 3500:REM move
 panels
7420 EVERY riseint,1 GOSUB 6100:REM rais
e water level
7430 RETURN
```

As we may wish, at some stage, to switch these interrupts off (this can be useful if there is an unacceptably large number of interrupt routines queueing up — switching off and restarting the interrupt timers makes the queue disappear) the following routine may prove useful, making use of `REMAIN` to permanently disable the two interrupt timers used:

```
7300 REM **** disable timers ****
7310 resetflag=0
7320 FOR i=0 TO 2:r=REMAIN(i):NEXT i
7330 RETURN
```

The final task to make both of these interrupt routines work within the program as we now have it, is to call the 'enable' routine from the main program as follows:

```
1440 GOSUB 7400:REM enable timers
```

Program structure

These two new interrupt routines work in conjunction with the `WHILE...WEND` loop that scans the joystick or cursor keys and updates the man character's position. As described in the first chapter, dotted lines are used within the structure diagram to show where one routine interrupts another. The structure diagram for the program so far, now looks like this:

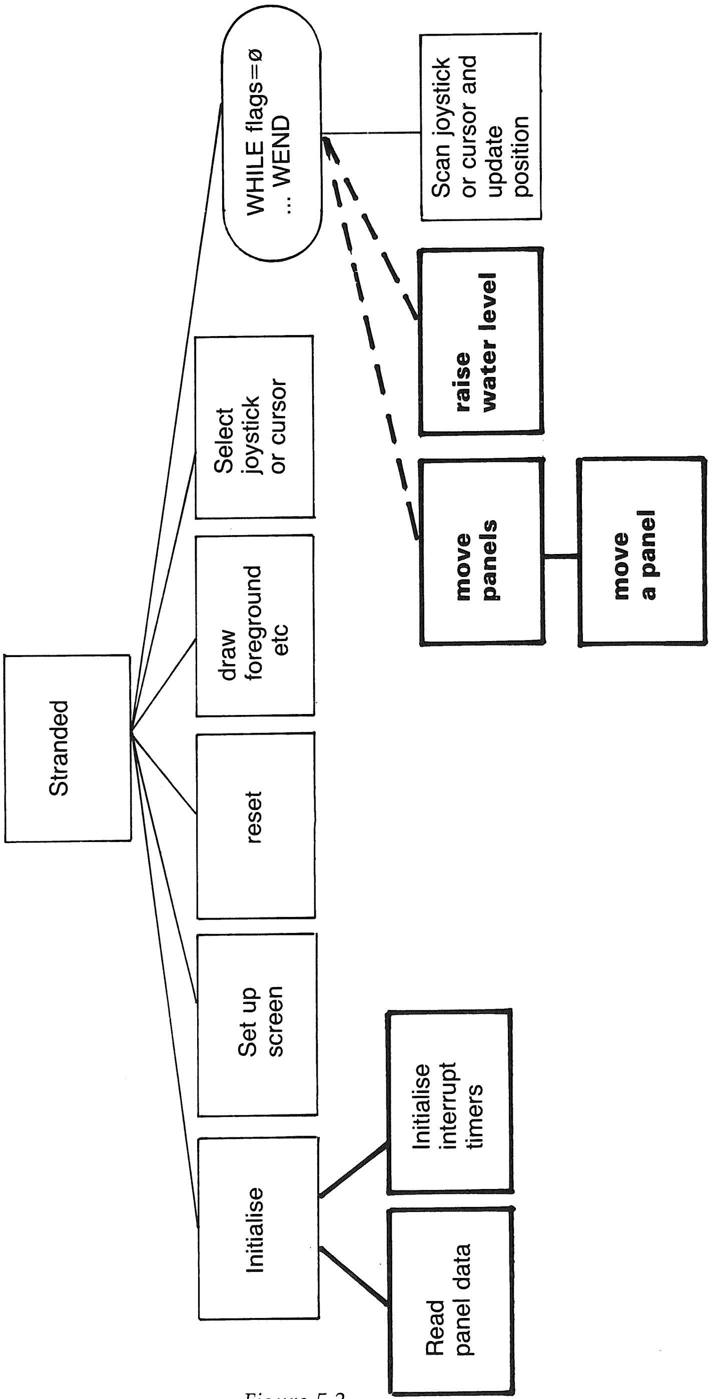

Stranded
Initialise
Set up screen
reset
draw foreground etc
Select joystick or cursor
WHILE flags=ø ... WEND
Read panel data
Initialise interrupt timers
move panels
move a panel
raise water level
Scan joystick or cursor and update position

Figure 5.2

Chapter Six

Sound and sound effects

In this chapter:

The addition of sound to a game adds excitement and makes the game more absorbing and appealing to the player. The Amstrad CPC range has excellent sound facilities that can be used from BASIC. The range of sound commands in Locomotive BASIC are extensive, allowing programs to play three simultaneous parts of a tune. Because of the power and range of the sound commands, much of their use falls outside the scope of this book. In this chapter we shall look at how basic sounds are generated and how more sophisticated sounds can be generated by modifying the sound by defining its 'shape'. Other features of sound, such as coordinating the three sound channels and generating sound interrupts will also be discussed briefly.

Sound parameters

When we hear a note, for example from a key struck on a piano, there are many things that go to define the sound. The most simple of these are the pitch of the note (how high or low the note is), duration (how long the note is) and volume (how loud the note is). Other, more complex, parts of the sound alter the quality of the sound we hear, but the three basic parameters alone can be used to produce sound on the Amstrad CPC range.

The pitch of the note is defined by setting a period number. Pitch is usually related to the frequency of the note, that is how many times per second the basic waveform repeats, also known as the number of cycles. As the note gets higher, so frequency increases. The period of a waveform is the length of one waveform cycle. Longer cycles produce lower notes or, to put it another way, as the note gets higher the period decreases. The period number can be any whole number in the range 0–4095. For those who wish to use their Amstrad CPC range to play music the following program produces a look-up table of notes to period numbers. Eight octaves are available. Octave 0 starts on middle C.

```
10 REM **** Note Table ****
20 DIM note$(12)
30 FOR i=1 TO 12:READ note$(i):NEXT i
40 DATA C,C#,D,D#,E,F,F#,G,G#,A,A#,B
50 MODE 1
60 FOR octave=-3 TO 4
70 CLS
80 PRINT"Octave ";octave:PRINT
90 PRINT"Note","Period":PRINT
100 FOR n=1 TO 12
110 freq=440*(2^(octave+(n-10)/12))
120 period=ROUND(125000/freq)
130 PRINT note$(n),period
140 NEXT n
150 a$="":WHILE a$="":a$=INKEY$:WEND
160 NEXT octave
```

The duration of the note is measured in 1/100th of a second and can be any positive number in the range 1–32,767. Zero and negative values of this parameter have a meaning in relation to sound envelopes which we shall consider later.

The volume of a note is defined by a whole number in the range 0–7, the note getting louder as this value increases.

The final task, before we can get a note to play, is to define which of the sound channels A, B or C we wish to play it on. We do this by setting a channel status number. Think of the bits within the binary equivalent of the channel status number as controlling some function of the sound. Bits 0, 1 and 2 define the sound channel used. So, channel A is selected by setting bit 0 to one, i.e. channel status=1, setting bit 1 to one selects channel B, i.e. channel status=2, and setting bit 2 to one selects channel C, i.e. channel status=4.

Other bits in the channel status number have other functions that will be covered briefly later in the chapter. The syntax of the **SOUND** command is this:

```
SOUND C,P,D,V
```

where C is the channel status, P is the period, D is the duration and V is the volume.

The following **SOUND** command plays a middle C, period=478, at maximum volume, for half a second on channel A:

```
SOUND 1,478,50,7
```

Although the sound produced is rather crude, we have enough control over the sound to play a tune. This short program plays 'Ba-Ba Black Sheep':

```
10 REM **** Ba-Ba Black Sheep ****
20 :
30 REM ** read note data **
40 DIM period(13),duration(13)
50 FOR note=1 TO 13
60 READ period(note),duration(note)
70 NEXT note
80 tempo=30
90 :
100 REM ** play tune **
110 FOR note =1 TO 13
120 SOUND 1,period(note),duration(note)*
tempo,5
```

```
130 NEXT note
140 :
150 DATA 239,0.5,0,0.5,239,0.5,0,0.5
160 DATA 159,0.5,0,0.5,159,0.5,0,0.5
170 DATA 142,0.5,127,0.5,119,0.5,142,0.5
180 DATA 159,2
```

The program reads in the period and duration data for each of the 13 notes or rests in the phrase. Notice that where a rest is needed, this can be achieved by setting the period number to 0. In the data statements, the duration of each note was defined in fractions of a note. By defining a variable, `tempo`, which we use to multiply the note duration in the `SOUND` command, the speed of the entire piece can be changed simply by altering its value. Try changing `tempo` to 50 or 75 or 15 to see the effect on the tune.

Shaping the sound

The main reason that the human ear can detect the difference between two instruments that play exactly the same tune is the difference in the quality of the sound produced by different instruments. A piano produces a percussive sound as its hammers hit the strings, whereas a harpsichord, which plucks its strings, makes a much sharper sound. When attempting to synthesise the characteristics of an instrument's sound, using electronics, this quality is called the sound envelope. It is possible on the Amstrad CPC range to improve upon the coarse tones produced by the `SOUND` command by defining an envelope that shapes the sound heard and gives it a different quality. The envelope of a note normally has four main sections as shown in the diagram overleaf:

The first section, called the attack, is characteristic of the note as it first starts; a sharply struck or plucked string will have a short attack, the note rising to maximum volume rapidly. The second section is the decay, the speed at which the note reduces in volume after the peak has been reached. The third section is the sustain, a period during which the note volume may remain more or less constant, and, finally, comes the release section as the note dies away completely.

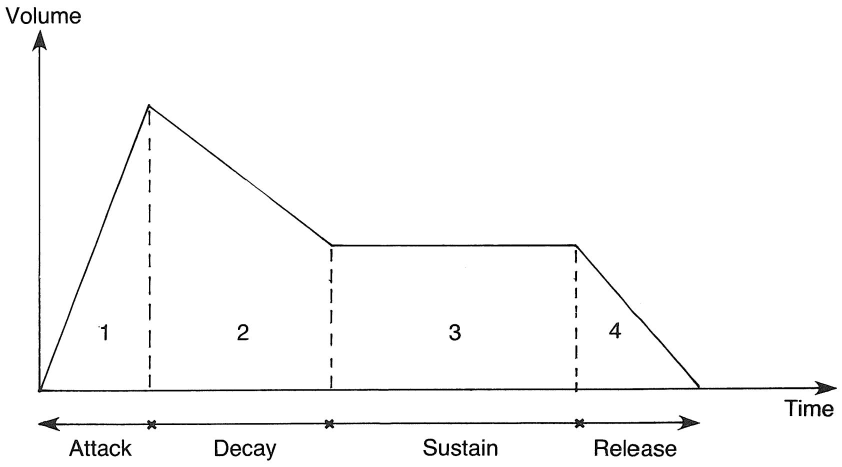


Figure 6.1

A sound envelope's shape is defined on the Amstrad CPC range, using the ENV command, the syntax of which looks frightening, but is, in fact easy to use:

```
ENV C,P1,Q1,R1,P2,Q2,R2,P3,Q3,R3,P4,
Q4,R4,P5,Q5,R5
```

where C is the envelope number, 0–15, P is the number of steps in each section, 0–127, Q is the vertical step size, −128 to 127, and R is the horizontal step size, 0–255.

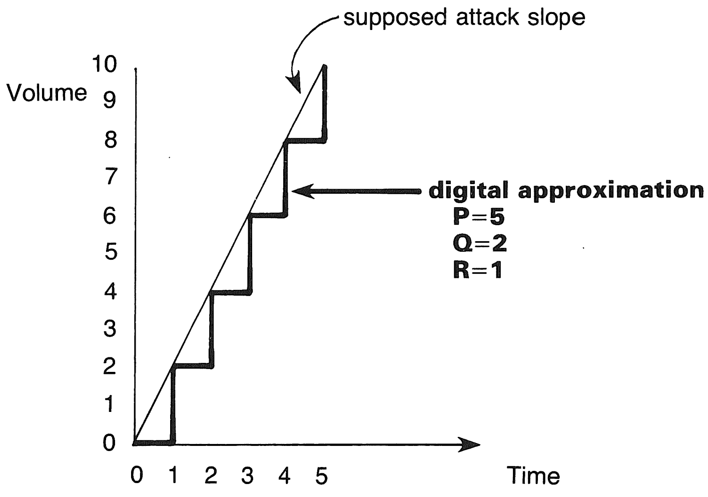


Figure 6.2

Up to five sections can be defined, each having three parameters, P, Q and R, to define the slope and size of each section. If, say, we wished the attack section to cause a rise in volume of 10 units in 5 time units then P, Q and R would approximate this digitally as a series of steps (see Figure 6.2):

Five steps are required, so P=5. Each step is 2 units vertically and 1 unit horizontally, so Q=2 and R=1, respectively. A complete envelope can be approximated digitally in this way, as in the example shown in Figure 6.3:

The 16 numbers that go to make up the ENV command consist of one envelope number and five groups of three numbers that define each of the five sections in this envelope. Envelopes do not need to have five sections, they can have four, three, two or even one section. As long as there are three numbers present for each section you want then the ENV command will work correctly.

The envelope given in the example above has a duration of 30/100ths of a second. We can adapt the SOUND command in several ways to include envelope shaping by the ENV command. A fifth number must be added to SOUND to the four we have already looked at. This fifth number identifies the envelope we wish to use with that sound. Up to 15 envelopes can be defined, each identified by the first number in the ENV command's list of parameters. We use this number with SOUND to select the correct envelope. If we wished to use a previously defined envelope, 1, with our SOUND, then the SOUND command would look like this:

```
SOUND C,P,D,V,1
```

It is worth noting that once an envelope has been defined using ENV it does not need to be redefined each time we wish to play a note using SOUND. Redefinition of an envelope is only necessary when you want to change the envelope's shape.

Envelopes and duration

When using sound envelopes the duration of the note and its volume are set by the envelope shape. The fourth SOUND parameter, controlling volume, is therefore usually set to 0, allowing all the note's volume to come from the envelope. The

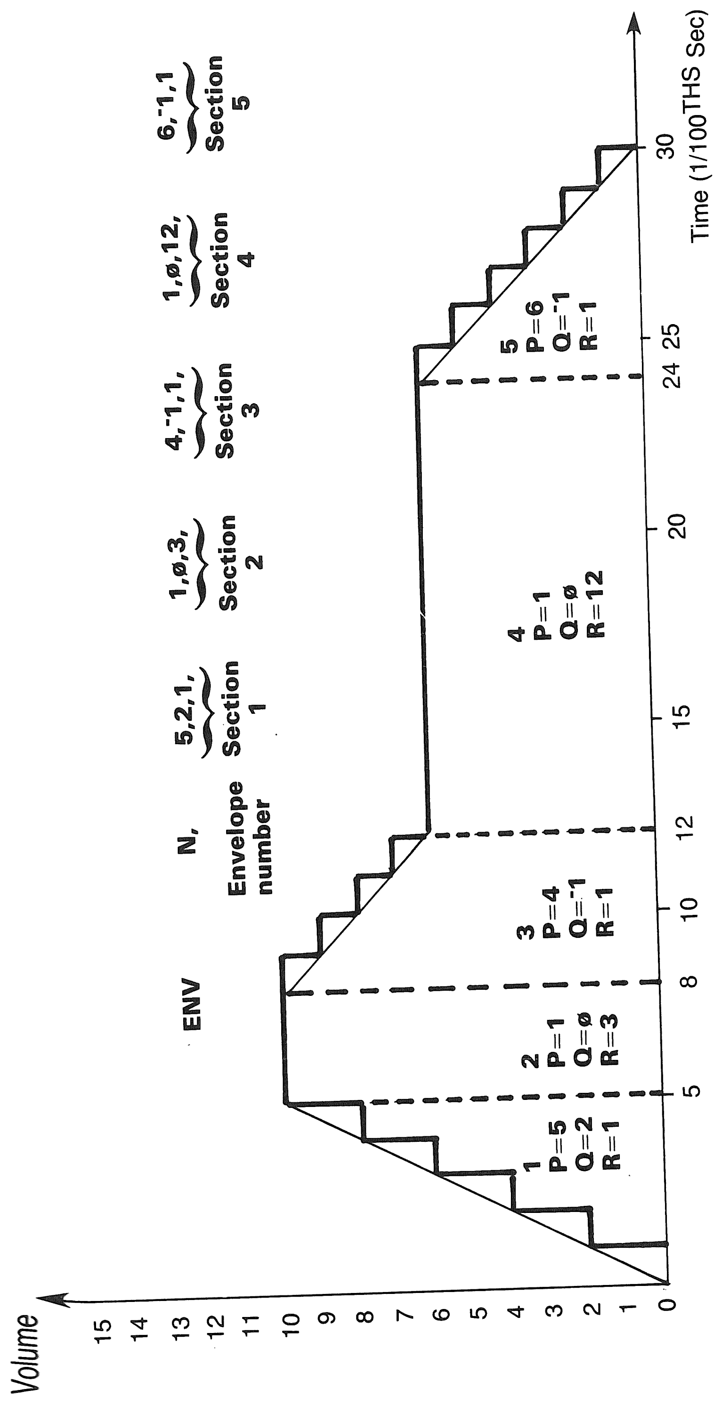

ENV
N,
Envelope number
5,2,1,
Section 1
1,ø,3,
Section 2
4,¯1,1,
Section 3
1,ø,12,
Section 4
6,¯1,1
Section 5
1 P=5 Q=2 R=1
2 P=1 Q=ø R=3
3 P=4 Q=¯1 R=1
4 P=1 Q=ø R=12
5 P=6 Q=¯1 R=1
Volume
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 5 8 10 12 15 20 24 25 30
Time (1/100THS Sec)

Figure 6.3

third parameter, controlling duration, can, however, act in three ways: firstly, if D is positive then it controls the duration of the note, rather than the envelope. If the envelope duration was 30 time units (as in our example) and D was set to 45 time units, then the note would simply be silent for the last 15 time units. The most sensible thing to do might be to work out the duration of the envelope you had designed and set D to that value. To save you the bother of having to work this out, the process is done automatically if, instead, D is set to 0. The duration of the note is then completely controlled by the envelope. The third way in which D can be used is as a negative value. If D is negative then it defines the number of times the envelope is repeated. So, for example, if D were −3, the sound would consist of three sounds, one after another, shaped by the envelope. The following short program demonstrates this:

```
10 REM **** duration demo ****
20 ENV 1,5,2,1,1,0,3,4,-1,1,1,0,12,
6,-1,1
30 CLS
40 PRINT"duration 0, ie envelope con-
trol"
50 SOUND 1,478,0,0,1
60 FOR delay=1 TO 1000:NEXT delay
70 CLS
80 PRINT"duration -3 ie three repeats"
90 SOUND 1,478,-3,0,1
```

The envelope generator program

To allow you to experiment with volume envelopes easily this program can be used. The cursor keys control a small cross on the screen that can be moved to define the end of each of the five possible stages of the envelope shape. Simply move the cross to the required position and press the **COPY** key. The envelope stage will then be drawn automatically from the last section, or from the origin if you are creating the first section. Two envelopes are drawn by the program, in fact. One is the ideal envelope shape you are trying to create; the second is the best digital approximation to the shape, which is automatically calculated and displayed by the program. Any stage of the envelope can be deleted by pressing **DEL** so that incorrect entries can be easily corrected. When you have finished designing your envelope shape, press the **ENTER** key to see the corresponding list of ENV parameters. The program also allows you to select the note duration, if you wish it to be different from that defined by the envelope. Twelve keys on the keyboard are defined as notes so that you can hear the effect of your envelope over an octave. The octave number (−3 at the lowest to +4 at the highest) can also be selected. The keys are:

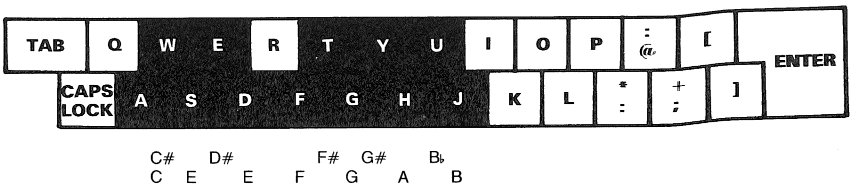


Figure 6.4

To design a new envelope press the **ENTER** key to return to edit mode.

```
10 REM ************************************
20 REM ************************************

30 REM **                                **
40 REM **  CPC 464 Envelope Generator    **
50 REM **                                **

60 REM **       by Steve Colwill         **
70 REM **           (c)1985              **
80 REM **                                **

90 REM ************************************

100 REM ***********************************
*
110 :
120 GOSUB 1000:REM initialise
130 GOSUB 1060:REM draw axes
140 x=10:y=10
150 FOR stage=1 TO 5
160 LOCATE 70,3:PRINT"stage :";stage
```

```
170 GOSUB 2000:REM move cursor
180 IF a$=CHR$(224) THEN GOSUB 4000:REM
calc stage
190 IF a$=CHR$(127) THEN GOSUB 5000:REM
rubout stage
200 IF a$=CHR$(13) THEN stage=5:REM exit
210 NEXT stage
220 GOSUB 6000:REM set other sound param
eters
230 GOSUB 7000:REM play
240 RUN
250 END
260 :
1000 REM **** initialise ****
1010 MODE 2:ORIGIN 40,200
1020 DIM a(16),period(20),press(20)
1030 DIM rx(5),ry(5),sx(5),sy(5)
1040 s=10:lx=0:ly=0
1050 RETURN
1060 REM **** draw axes ****
1070 MOVE -2,-2
1080 DRAWR 0,150,1
1090 MOVE -2,-2
1100 DRAWR 623,0
1110 REM ** label x axis **
1120 FOR x=0 TO 620 STEP 10
1130 PLOT x,-4
1140 NEXT x
1150 TAG:FOR x=0 TO 620 STEP 50
1160 MOVE x-8,-8:PRINT x/5;
1170 NEXT x
1180 MOVE 424,-24:PRINT"time in 1/100th
sec";
1190 REM ** label y axis **
1200 FOR y=0 TO 150 STEP 10
1210 PLOT -4,y
1220 NEXT y
1230 TAG:FOR y=0 TO 150 STEP 50
1240 MOVE -48,y+8:PRINT y/10;
1250 NEXT y
1260 MOVE -40,176:PRINT"volume";
1270 TAGOFF
1280 REM ** title **
1290 LOCATE 24,1:PRINT"CPC 464 Volume En
```

```
velope Generator"
1300 REM ** keypress data **
1310 FOR i=1 TO 12:READ press(i):NEXT i

1320 DATA &61,&77,&73,&65,&64,&66,&74
1330 DATA &67,&79,&68,&75,&6A
1340 REM ** period data **
1350 FOR i=1 TO 12:READ period(i):NEXT i

1360 DATA 478,451,426,402,379
1370 DATA 358,338,319,301,284,268,253
1380 RETURN
1390 :
2000 REM **** move cursor ****
2010 SPEED KEY 10,2
2020 a$="":col=1:GOSUB 3000:REM position
 cursor
2030 WHILE a$<>CHR$(224) AND a$<>CHR$(13
) AND a$<>CHR$(127)
2040 a$="":WHILE a$="":a$=INKEY$:WEND
2050 col=0: GOSUB 3000:REM rubout old cu
rsor
2060 IF INKEY(0)=0 THEN y=y+s:IF y>150 T
HEN y=150
2070 IF INKEY(2)=0 THEN y=y-s:IF y<10 TH
EN y=10
2080 IF INKEY(1)=0 THEN x=x+s:IF x>620 T
HEN x=620
2090 IF INKEY(8)=0 THEN x=x-s:IF x<snx+1
0 THEN x=snx+10
2100 col=1:GOSUB 3000:REM draw new curso
r
2110 WEND
2120 RETURN
2130 :
3000 REM **** draw cursor ****
3010 MOVE x-5,y:DRAWR 10,0,col
3020 MOVER -5,5:DRAWR 0,-10
3030 RETURN
3040 :
4000 REM **** calculate stage ****
4010 col=0:GOSUB 3000:REM rubout cursor

4020 a=INT((y-sy(stage-1))/10):b=INT((x-
sx(stage-1))/5)
4030 IF ABS(a)<ABS(b) THEN p=ABS(a) ELSE
 p=ABS(b)
4040 IF p<>0 THEN q=INT(a/p):r=INT(b/p)
ELSE p=1:q=a:r=b
4050 a(3*stage-1)=p:a(3*stage)=q:a(3*sta
ge+1)=r
4060 REM ** draw stage **
4070 MOVE sx(stage-1),sy(stage-1)
4080 DRAW x,y,1
4090 REM ** draw approx shape **
4100 MOVE rx(stage-1),ry(stage-1)
4110 FOR i=1 TO a(3*stage-1):DRAWR 5*a(3
*stage+1),0
4120 DRAWR 0,a(3*stage)*10
4130 NEXT
4140 rx(stage)=XPOS:ry(stage)=YPOS
4150 sx(stage)=x:sy(stage)=y
4160 x=x+10
4170 RETURN
4180 :
5000 REM **** rubout a stage ****
5010 stage=stage-1
5020 IF stage<1 THEN stage=0:RETURN
5030 REM ** rubout approx env **
5040 FOR ey=0 TO 150 STEP 2
5050 MOVE sx(stage),ey:DRAW sx(stage-1),
ey,0
5060 MOVE rx(stage),ey:DRAW rx(stage-1),
ey,0
5070 NEXT ey
5080 REM ** remove env entries **
5090 a(3*stage-1)=0:a(3*stage+1)=0:a(3*s
tage)=0
5100 stage=stage-1
5110 RETURN
5120 :
6000 REM **** other sound parameters ***
*
6010 ENV 1,a(2),a(3),a(4),a(5),a(6),a(7)
,a(8),a(9),a(10),a(11),a(12),a(13),a(14)
,a(15),a(16)
6020 envdur=0:FOR i=2 TO 16 STEP 3:envdu
r=envdur+a(i)*a(i+2):NEXT i
```

```
6030 LOCATE 1,16:PRINT"ENV command is: 1
,";
6040 FOR i=2 TO 15:PRINT a(i);",";:NEXT:
PRINT a(16)
6050 LOCATE 1,18:PRINT"envelope duration
 is:";envdur
6060 LOCATE 1,20:INPUT"enter note durati
on or (enter) for env duration";dur
6070 IF dur=0 THEN dur=envdur
6080 LOCATE 1,22:INPUT"enter octave numb
er";oct
6090 IF oct<-3 OR oct>4 THEN 6080:REM ch
eck valid octave
6100 LOCATE 1,24:PRINT"SOUND command is:
 1, period,";dur;", 0, 1"
6110 RETURN
6120 :
7000 REM **** play monosynth ****
7010 SPEED KEY dur,dur
7020 a$="":WHILE a$<>CHR$(13)
7030 a$="":WHILE a$="":a$=INKEY$:WEND:RE
M get note
7040 FOR note=1 TO 12
7050 IF a$=CHR$(press(note)) OR a$=CHR$(
press(note)-32) THEN SOUND 1,period(note
)/(2^oct),dur,0,1:note=20
7060 NEXT note
7070 WEND
7080 RETURN
```

Tone envelopes

To create really sophisticated musical sounds any synthesiser needs a method of adding vibrato to a note. When violinists or guitarists hold a string down whilst playing a note, they often rock their finger back and forth gently to raise and lower the pitch of the note slightly. The effect is often used to make long notes sound more interesting. For those of you with musical aspirations, vibrato can be added to notes played using the ENT command. This command is similar to ENV, allowing you to shape a tone envelope for your sound. The pitch of the note is set by the second parameter in the SOUND command, but we can raise and lower the pitch slightly using ENT. Again, up to five sections can be defined; the example below has only two sections, causing the pitch of the note to dip (i.e. increasing the period) before returning to normal. The tone period set in the **SOUND** command can be varied by up to 5 units, up or down, using **ENT**:

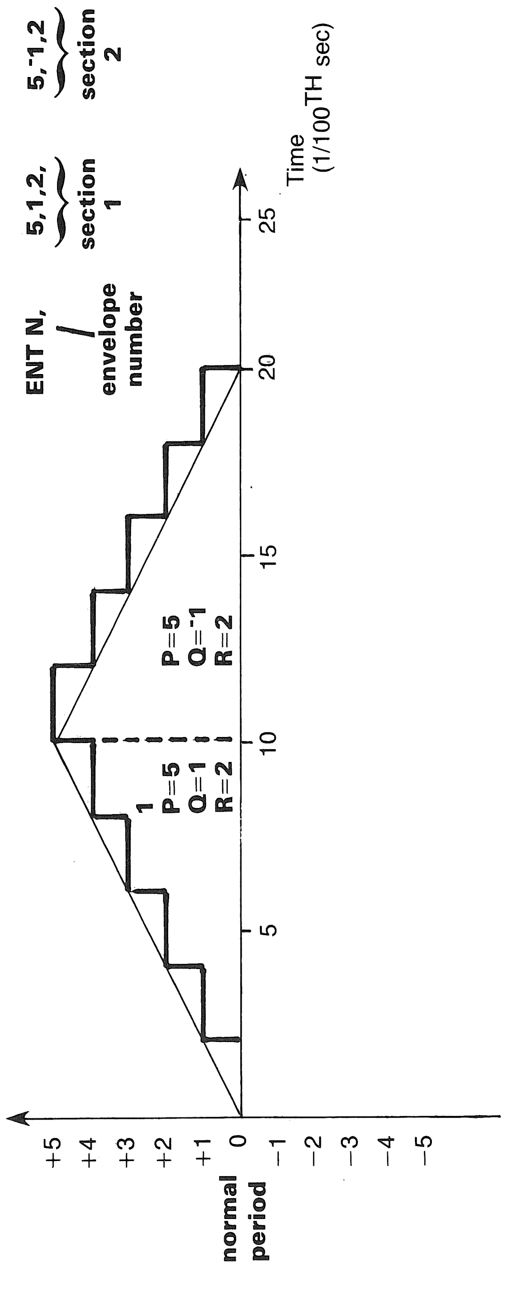


Figure 6.5

If the duration of the tone envelope is greater than the duration of the note (as defined by the volume envelope or D in the SOUND command) then the remainder is ignored when the note is played. If the tone envelope is shorter, then the normal period is restored for the remainder of the note.

As with the volume envelope, up to 15 tone envelope's can be defined, each identified by a number. To select, say, volume envelope 1 and tone envelope 3 to shape a sound, we would use this SOUND command:

```
SOUND 1,P,D,V,1,3
```

Noise

A seventh and final parameter can be added to the SOUND command to blend in noise with the note being generated. This parameter must be in the range 0–15. If it is set to 0 then no noise is added; if it is set to 15 then most noise is added. Adding noise 'dirties' the quality of the note produced, and can be useful in sound effects when we are trying to simulate shooting or explosions.

More about channel status

We have seen that the first parameter in the SOUND command, the channel status number, can be used to select one of the three sound channels. These are controlled by the first three bits of the channel status number as described earlier. The other five bits also have their functions. For the sake of completeness these are:

Bit	Function
0	Send to channel A
1	Send to channel B
2	Send to channel C
3	Rendezvous with channel A
4	Rendezvous with channel B
5	Rendezvous with channel C
6	Hold
7	Flush

The rendezvous facility is for use in pieces of music with more than one part and allow two channels to synchronise their actions and so produce notes together.

To understand how the hold function works it is first necessary to realise that each channel can hold up to five SOUND commands in a queue. This allows new SOUND commands to be issued whilst an old command is still playing on the same sound channel. The new command is not lost but is added to the queue to wait for the previous command to finish. Normally, as soon as a sound is finished the next one in the queue is started, but this process can be stopped by using the hold function. For example, a channel status number of 65 (i.e. 01000001) would direct the sound to channel A and cause a hold — effectively stopping sound processing on channel A. A hold is released by the RELEASE command followed by 1, 2, or 4 to denote sound channel A, B or C. Using the hold and release all five spaces in a sound channel queue can be filled with a SOUND command before starting to play the tune.

Bit 7 of the channel status flushes the designated sound channel queue, causing all commands waiting in the queue to be lost. This may be useful if you wish to bring all sounds on a particular channel to an abrupt halt.

The SQ command

SQ(x) allows you to test the state of any of the three sound channels where x is 1, 2 or 4. The value returned is an eight-bit number where the state of each bit in the number relates to the state of various features of the channel selected.

Bit	Function
0, 1, 2	Number of free spaces in queue tested (0...4)
3	Rendezvous with channel A (1=yes, 0=no)
4	Rendezvous with channel B (1=yes, 0=no)
5	Rendezvous with channel C (1=yes, 0=no)
6	Hold at head of queue (1=yes, 0=no)
7	This channel now playing (1=yes, 0=no)

The first three bits combine to tell you how many free spaces are available in the queue for the sound channel looked at. Bits 3–6 will tell you whether the **SOUND** command at the front of the queue is set to rendezvous with one of the other channels, or is a hold command, and bit 7 flags whether the nominated channel is now playing or not.

Because channel status and **SQ** are bit-significant numbers (i.e. the individual bits have a meaning) it is a good idea to use the logical operators **AND** and **OR** (see Chapter 1) to isolate bits within the number for testing. As a simple example of this, the following line of BASIC tests sound channel A to see if it is currently being held. By **AND**ing **SQ** with 64, bit 6 is isolated for testing, the other bits being masked out.

```
IF (SQ(1) AND 64)=1 THEN RELEASE 1
```

ON SQ(x)...GOSUB

In addition to testing the status of a channel **SQ** can be used to trigger an interrupt when a space becomes available in the sound queue nominated by x. The interrupt's priority is equal to that of interval timer 2 (see Chapter 5) but **ON SQ...GO-SUB** should be used carefully. The command is rather like a gun with only one bullet. Once an interrupt has been triggered, the command is disarmed. If you want to keep generating interrupts using **ON SQ...GOSUB** then it is probably best to re-issue the command (i.e. load another bullet) at the end of the subroutine called by the original **ON SQ...GOSUB** command. To make matters more confusing, the interrupt is also disarmed if the program meets a **SOUND** or **SQ** keyword.

Sound effects

The sound and music-making capabilities of the Amstrad CPC range are very wide-ranging but much practice is needed to achieve the sounds you want. Here are a selection of sound effects that you might wish to try.

```
10 REM **** gear change ****
20 FOR gear=1 TO 4
```

```
30 FOR period=1500 TO 350 STEP -10+gear
40 SOUND 1,period,5,7,0,0,2
50 NEXT period,gear
```

```
10 REM **** lift off ****
20 FOR period=200 TO 50 STEP -1
30 SOUND 1,period,10,4,0,0,2
40 SOUND 2,period+500,10,4
50 NEXT period
```

```
20 REM **** space jazz ****
30 RANDOMIZE -TIME
35 REM ** set up drone **
40 SOUND 2,900,2000,3
50 SOUND 1,1100,2000,2
55 REM ** test for end of drone **
60 WHILE (SQ(1)AND 128)=128
70 period=INT(RND(1)*1000)
80 SOUND 4,period,10,3
90 WEND
100 SOUND 132,0,0,0:REM flush channel C
```

```
10 REM ****      Super Shot       ****
20 REM **** press SPACE to fire ****
30 WHILE pigsfly=0
40 IF INKEY$=" " THEN GOSUB 70
50 WEND
60 END
70 FOR period=1 TO 30
80  SOUND 1,period,1,7,0,0,1
90 SOUND 2,0,1,3,0,0,10
100 NEXT period
110 RETURN
```

```
10 REM **** siren ****
20 FOR i=1 TO 10
30 FOR period=400 TO 250 STEP -5
40 SOUND 1,period,2,5
50 SOUND 2,period+100,2,5
60 NEXT period,i
```

'Stranded' — sound and movement

In the last chapter we added an interrupt routine that made the tide rise at regular intervals during the game. In this section we shall add a sound effect to simulate the sound of surf breaking against the pillars. We also add to the movement routines already devised by adding subroutines to detect when the player's character has fallen off the cliff-walk and when it is standing on one of the moving panels, allowing the character to be moved back and forth with the panel.

Surfing sounds

The effect of surf crashing against the pillars can be achieved by using a `SOUND` command that generates noise only and shaping the noise produced using a volume envelope. Noise can be generated without an accompanying tone by simply setting the period parameter in the `SOUND` command to 0. If the duration and volume are also set to 0 then these functions are controlled by the envelope. The envelope used to shape the surf sound appears as in figure 6.6.

The envelope is shaped to give a sharp attack followed by a short period of sustain at maximum volume, before undergoing a sharp decay to one-third of full volume. To gain the 'afterwash' effect the last two sections allow the noise to die slowly away. The total duration of the note is around 3.5 seconds. Add these two lines to the interrupt routine that causes the tide to rise:

```
6210 ENV 1,5,3,3,1,0,30,5,-2,10,4,-1,60,
1,0,20
6220 SOUND 2,0,0,0,1,0,15
```

Sound and movement

To give the game aural as well as visual appeal we can add a simple sound effect to the movement routines already developed. The subroutine that generates this sound effect is:

```
7800 REM **** move sound effect ****
7810 tone=100+20*chary
7820 SOUND 4,tone,5,5
7830 RETURN
```

Notice that the period of the short beep produced is calculated from the y coordinate of the player's character, `chary`. This means that as the character makes its way down

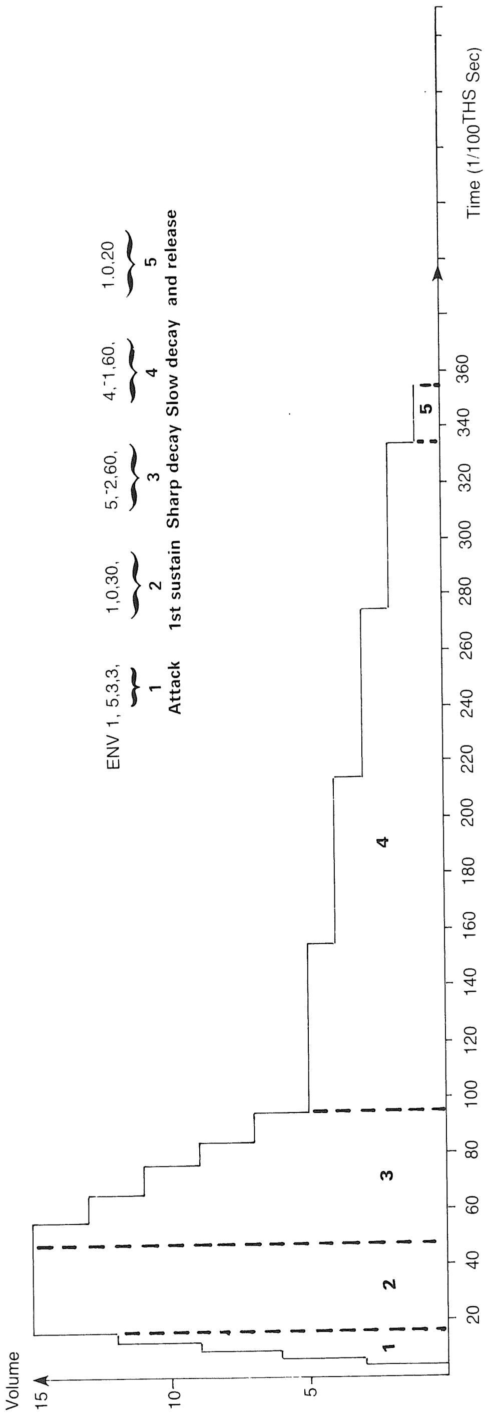

ENV 1, 5,3,3,
1,0,30,
5,-2,60,
4,-1,60,
1,0,20
1
Attack
2
1st sustain
3
Sharp decay
4
Slow decay
5
and release
Volume
15
10
5
Time (1/100THS Sec)
20
40
60
80
100
120
140
160
180
200
220
240
260
280
300
320
340
360
1
2
3
4
5

Figure 6.6

the cliff the beeps will become lower in pitch, and rise again as the cliff-walk is ascended again.

We must call this sound routine from both of the subroutines, designed earlier, to control left and right movement. Add these two lines:

```
4060 GOSUB 7800:REM sound fx
4160 GOSUB 7800:REM sound fx
```

Falling and jumping onto panels

When the player attempts to move, the program must ensure that there is part of the cliff-walk beneath him, be it the main sections or the moving panels. The simplest way to determine whether or not the player is standing on fresh air is to test the colour of a point in the square beneath the player's character using TEST. Unfortunately there is a snag to using TEST — it requires graphics coordinates, while the program holds the player's character position as character-cell coordinates. We therefore need a short subroutine to convert for us. See Chapter 4 for more details:

```
6700 REM **** convert char/graph ****
6710 graphx=32*charx-16
6720 graphy=399-16*chary
6730 RETURN
```

This routine generates a pair of graphics coordinates that correspond to a point directly below the player character's feet. We can now use these coordinates with TEST. You should already have a dummy routine, consisting of the title line and a RETURN line, for testing under the figure. We can now fill in the rest of this routine:

```
4320 GOSUB 6700:REM convert coords
4330 t=TEST(graphx,graphy)
4340 IF t=sky THEN fallflag=1:GOSUB 4400
:RETURN
4350 IF t=black THEN GOSUB 4700:RETURN E
LSE panel=0
```

This routine, firstly, performs the coordinate conversion and then tests the colour of a point directly under the player figure. If the result of that test is the colour 'sky' then a fall routine at line `4400` is called and a flag, `fallflag`, is set to indicate that this has happened. The use of this flag will be explained later.

The fall routine causes the player figure to move down the screen, by increasing its y coordinate `chary` within a loop and erasing its old position. The `WHILE...WEND` loop terminates when the figure reaches solid ground, either a section of the cliff-walk or a panel, and the routine moves into its next phase to produce an explosion effect.

```
4400 REM **** fall ****
4410 chary=chary+1:GOSUB 6700:REM conver
t
4420 DI
4430 WHILE TEST(graphx,graphy)=sky
4440 LOCATE charx,chary-1:PRINT" "
4450 chary=chary+1:GOSUB 6700:REM conver
t
4460 LOCATE charx,chary:PRINT standman$
4470 GOSUB 7800:REM sound fx
4480 WEND
4490 LOCATE charx,chary-1:PRINT" "
4500 GOSUB 6800:REM explode
4510 GOSUB 2900:REM rest coords etc
4520 EI
4530 RETURN
```

The explosion routine causes a special character to be printed and flashes different colours at the position where the player figure touched down.

```
6800 REM **** explode ****
6810 DI
6820 FOR i=1 TO 5
6830 FOR j=15 TO 0 STEP -1
6840 LOCATE charx,chary
6850 PEN j:PRINT explode$
6860 SOUND 4,0,2,5,0,0,j
6870 NEXT j,i
```

```
6880 LOCATE charx,chary:PRINT" "
6890 DI
6900 RETURN
```

If the player is not standing on thin air then the 'check under figure' routine goes on to see if the colour underneath the figure is black. This colour indicates that the player is either standing on a panel, or is standing at the top or bottom of a ladder. Before the program can take any action it needs to determine if the player is on a panel, and, if so, which panel, or, if he is on a ladder, which ladder. This routine finds this information.

```
4700 REM **** on a panel/ladder ? ****
4710 IF panel<>0 THEN RETURN
4720 FOR i=1 TO np
4730 IF graphy=sty(i) AND graphx>=stx(i)
 AND graphx<=ex(i) THEN panel=i:i=np
4740 NEXT i
4750 IF panel<>0 THEN incscore=50:GOSUB
7700
4760 REM ** test for ladder **
4770 IF ladflag=1 THEN RETURN
4780 lad=0
4790 FOR i=1 TO 3
4800 IF charx=lx(i) THEN lad=i:GOSUB 490
0:i=3
4810 NEXT i
4820 RETURN
```

The routine, firstly, checks for a panel by comparing the coordinates of the point under the figure with the coordinates of each panel until a match is found. The variable `panel` is then set to the corresponding panel number and the `FOR...NEXT` loop is terminated by setting the loop counter `i` equal to `np`, the upper limit of the loop. One of the rules of the game is that every time a player hops onto a moving panel successfully the score is increased by 50. This is done in line `4750`.

The second half of the routine goes on to test for a ladder by again comparing the player's coordinates with each of the ladder's coordinates in turn. If a match is found then the `ladder` number is stored in `lad` and a subroutine generates movement up or down the relevant ladder. As this routine is the subject of the topics covered in the next chapter we will simply insert the title and `RETURN` lines:

```
4900 REM **** up or down a ladder ****
5070 RETURN
```

The next question to be answered is: What happens if the program decides that the player is on a panel? The answer is that the player figure must be made to move backwards and forwards in synchronisation with the panel being stood upon. Remember that the variable `panel` is now set to a number that corresponds to the panel the player is on. We can therefore add a small piece of code to the interrupt that moves the panels; firstly, to see if the panel currently being moved is the one stood on, and, secondly, if it is, moving the player figure accordingly. The following lines accomplish these two tasks:

```
3620 IF panel=n THEN GOSUB 3800
3800 REM **** move man on panel ****
3810 LOCATE charx,chary:PRINT" "
3820 charx=charx+dx(n)/32:IF charx<1 THE
N charx=1
3830 LOCATE charx,chary:PRINT standman$
3840 RETURN
```

Scoring

As has already been mentioned, every time a player steps onto a panel the score is increased by 50 points. Other events in the program also cause the score to be increased. A general purpose scoring routine can be used to increase and display the score at any stage, setting the variable `incscore` to the amount by which we want the score increased. To ensure that the score is always shown as a six-digit number, leading zeros, that is zeros on the left-hand end of the number are added if necessary. This is done by string manipulation. After converting the current score to a string using `STR$` the left-hand character is removed. When numeric variables are converted to strings the left-hand character is used to carry the sign of the number, if it is negative. If the number is positive then the leftmost character is a space. To avoid unwanted spaces appearing in our score display we must therefore remove this character. Having removed it line 7750 adds the required numbers of zeros to the front of score$ and the score is printed. The variable zero$ used in this process is defined during the initialisation routine as a string of five zeros.

```
1870 zero$=STRING$(5,"0")
7700 REM **** increase score ****
7710 PEN red
7720 score=score+incscore:score$=STR$(sc
ore)
7730 lgth=LEN(score$):score$=RIGHT$(scor
e$,lgth-1)
7740 lgth=LEN(score$)
7750 score$=LEFT$(zero$,6-lgth)+score$
7760 LOCATE 14,1:PRINT score$
7770 PEN mancol
7780 RETURN
```

We can use this to display the original, zero, score at the start of the game.

```
2840 LOCATE 8,1:PRINT"Score:"
2850 incscore=0:GOSUB 7700:REM print sco
re
```

The main loop structure

Until now the program loop that repeatedly calls the movement routines (lines 1460–1480) has never been able to terminate itself. Now that the player can fall we need to be able to exit this loop and add a second outer loop to count through the number of lives that the player has. This is where the use of flags comes in. If a player falls then fallflag is set during the movement routine that checks under the player figure. On returning to the main loop the loop will be terminated as the WHILE...WEND condition for continuation of the loop depends on fallflag *and* boatflag *and* endflag being 0. If any of these flags are set then the loop will terminate. Conditions that set boatflag and endflag will be dealt with later, but for now we can add an outer loop to count through the number of players' lives and a line to reset the flags and trigger the next cycle of this outer loop if `fallflag` is set.

```
1310 numbermen=4:menhome=0
1400 WHILE numbermen>0 AND endflag=0
1410 LOCATE charx,chary:PRINT standman$
1500 IF fallflag=1 THEN GOSUB 3000:GOTO
1730:REM end loop
1730 WEND
```

To stop the program falling through to the subroutines once all four lives have been used an `END` statement must be added:

```
1750 END
```

Program structure

We have now added several routines to those that scan the joystick or cursor to update the position. In addition the whole loop structure inherited from the last chapter is now enclosed within an outer loop that will terminate when the number of player's lives is reduced to zero. We can show these additions on the program structure diagram as follows in Figure 6.7.

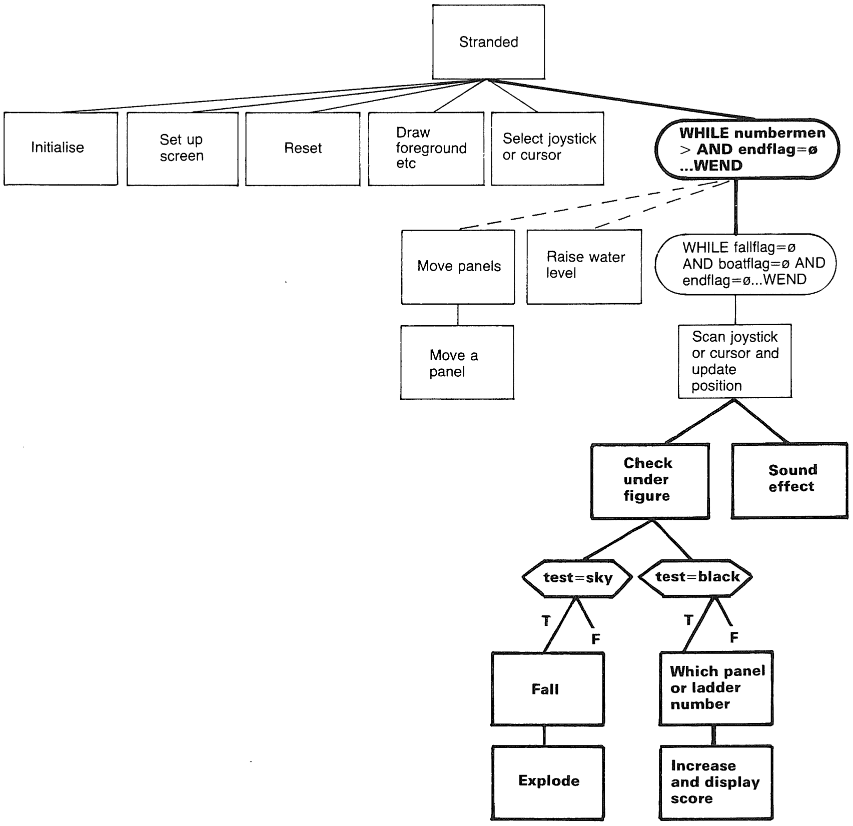

Stranded
Initialise
Set up screen
Reset
Draw foreground etc
Select joystick or cursor
WHILE numbermen > AND endflag=ø ...WEND
Move panels
Raise water level
WHILE fallflag=ø AND boatflag=ø AND endflag=ø...WEND
Move a panel
Scan joystick or cursor and update position
Check under figure
Sound effect
test=sky
test=black
T
F
T
F
Fall
Which panel or ladder number
Explode
Increase and display score

Figure 6.7

Chapter Seven

Control characters

In this chapter:

The Amstrad CPC range's character sets have ASCII codes from 32 to 255 and can be printed by the following command:

```
PRINT CHR$(x)
```

where x is the ASCII code of the character. The ASCII codes from 0 to 31 are not part of the character set and do not produce characters on the screen but can be used to control special features of the Amstrad CPC range's graphics capabilities. We start this chapter by taking a brief look at the function of each code.

Amstrad control characters

Value		Name	Parameters	Effect
Dec	Hex			
0	&00	NUL	None	Does nothing
1	&01	SOH	0–255	Prints character to screen
2	&02	STX	None	Turns text cursor off
3	&03	ETX	None	Turns text cursor on in immediate mode
4	&04	EOT	0–2	Sets screen mode
5	&05	ENQ	0–255	Prints character at graphics cursor
6	&06	ACK	None	Turns on text screen: see `CHR$(21)`
7	&07	BEL	None	Makes short beep: same as **CTRL/P**
8	&08	BS	None	Moves text cursor back one cell
9	&09	TAB	None	Moves text cursor forward one cell
10	&0A	LF	None	Moves text cursor down one cell
11	&0B	VT	None	Moves text cursor up one cell
12	&0C	FF	None	Clears screen or window: same as `CLS`
13	&0D	CR	None	Moves cursor to left edge of window on current line
14	&0E	SO	0–15	Sets `PAPER`colour
15	&0F	SI	0–15	Sets `PEN` colour
16	&10	DLE	None	Deletes character under text cursor
17	&11	DC1	None	Clears line from left of window up to and including current character
18	&12	DC2	None	Clears from current character to right edge of the window
19	&13	DC3	None	Clears from start of window up to and including current character
20	&14	DC4	None	Clears from current character to end of window
21	&15	NAK	None	Turns off text screen: see `CHR$(6)`
22	&16	SYN	0 and 1	Turns transparent option off/on
23	&17	ETB	0–3	Sets graphics plotting mode: see later
24	&18	CAN	None	Change to inverse video
25	&19	EM	Nine parameters, each 0–255	Defines a character: same as `SYMBOL`
26	&1A	SUB	Four parameters; 1–80, 1–80, 1–25, 1–25	Defines a text window. Four parameters define left, right, top and bottom of window: same as `WINDOW`
27	&1B	ESC	None	Does nothing

Value		Name	Parameters	Effect
Dec	Hex			
28	&1C	FS	Three parameters: 0–15, 0–31, 0–31	Sets INK colour to flash between two specified colours
29	&1D	GS	Two parameters: 0–31, 0–31	Sets border to a flash between a pair of specified colours
30	&1E	RS	None	Homes cursor to top left of screen
31	&1F	US	Two parameters: 1–80, 1–25	Moves cursor to cell specified by two parameters

The general use of control characters

The 32 control characters will perform their function if printed to the screen using `PRINT CHR$(x)`, where x is the control character's ASCII code between 0 and 31. Most of the control characters can be made to affect only the text window to which they are printed. For example, this short program demonstrates how window 1 can be changed to print in inverse video using `CHR$(24)` without affecting window 0, the normal screen:

```
10 WINDOW #1,1,40,1,12
20 PAPER #1,2:PEN #1,3:CLS #1
30 PRINT #1, "HELLO"
40 PRINT #1,CHR$(24):PRINT #1,"NOW IN
INVERSE VIDEO IN WINDOW 1"
50 LOCATE 1,13:PRINT "BUT NOT IN WINDOW
0"
```

Passing of a list of parameters with a control character can be demonstrated by a simple example. `CHR$(4)` can be used to set the screen mode. The parameter we must pass with this control character will determine which mode is selected. So to switch to mode 2 we would type this command:

```
PRINT CHR$(4)+CHR$(2)
```

Where more than one parameter must be passed we simply chain them into a single `PRINT` statement as `CHR$` numbers separated by plus signs.

Many of the control-characters functions are obvious from their description in the table. But some control characters have particularly interesting effects that are worth looking at in more detail.

The transparent option

When a character is printed into a particular cell the original contents of that cell are obliterated. By selecting the transparent option with

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

we stop this happening. With the option *on*, an old character in a cell will show through any part of the new character that is not solid. This effect can be used to build up multiple character images within a single square or stop any background data being erased when a character is printed on top. The characters can even be different colours! No doubt this facility was designed for adding accents and umlauts to foreign text, but the effect can be used in graphics games that use text characters. This option is used in several places in 'stranded'.

The transparent option can be turned off by

```
PRINT CHR$(22)+CHR$(Ø)
```

Screen scrolling

The Amstrad CPC range screens can be scrolled up simply by moving the cursor to the bottom of the screen and generating a line feed, but downward scrolling can also be achieved by using `CHR$(11)`.

The principle can be domonstrated by this simple program which moves the cursor to the top of the screen and causes the screen to scroll down 10 lines:

```
    1Ø LOCATE 1,1
    2Ø FOR I=1 TO 1Ø
    3Ø PRINT CHR$(11);
    4Ø NEXT I
```

The semicolon (;) which follows CHR$(11) is important as it surpresses the line feed that would normally occur. The text cursor must remain on the top line whilst printing a series of CHR$(11) characters if the screen is to move downwards.

By directing the CHR$(11) characters to a window, individual windows can be scrolled independently. This program demonstrates the effect:

```
1000 REM **** vertical scroll demo ****
1010 MODE 0:BORDER 0
1020 WINDOW 1,20,19,25
1030 WINDOW #1, 1,20,1,18
1040 PAPER 0:CLS:PEN 1
1050 PAPER #1,3:CLS #1:PEN #1,5
1060 LOCATE 5,3:PRINT"main window"
1070 LOCATE #1,7,1:PRINT#1,"window 1"
1080 :
1090 WHILE INKEY$=""
1100 REM **** scroll window 1 down ****
1110 LOCATE #1,1,1
1120 FOR i=1 TO 18:PRINT#1,CHR$(11);
1130 PRINT i;
1140 NEXT i
1150 REM **** and back up ****
1160 LOCATE #1,1,18
1170 FOR i=1 TO 18:PRINT#1,CHR$(10);
1180 PRINT i;
1190 NEXT i
1200 WEND
1210 END
```

Upward and downward scrolling of the screen can be a useful effect in certain types of arcade game as this next program shows. Scrolling offers a different way of moving characters vertically on the screen. In 'Snake Stomp' two windows are defined. The upper window contains a boot character which will move down in response to a keypress in an attempt to stand on the snake that squirms below. Needless to say the boot's downward, and subsequent upward, motion is caused by scrolling the screen using control characters.

```
1000 REM ******************
1010 REM ******************
1020 REM **                **
1030 REM **  Snake Stomp   **
1040 REM **                **
1050 REM ******************
1060 REM ******************
1070 :
1080 MODE 0:BORDER 0
1090 GOSUB 2000:REM initialise
1100 :
1110 REM **** main program loop ****
1120 FOR count=1 TO 20
1125 LOCATE #2,2,1:PRINT #2,count
1130 WHILE INKEY$=""
1140 GOSUB 6000:REM rub out old boot
1150 x=x+dx
1160 IF x>18 THEN x=17:dx=-1
1170 IF x<1 THEN x=1:dx=1
1180 GOSUB 5000:REM print new boot
1190 GOSUB 4000:REM move snake
1200 WEND
1210 :
1220 REM **** scroll down ****
1230 LOCATE #1,1,1
1240 FOR i=1 TO 16
1250 GOSUB 4000:REM move snake
1260 PRINT #1,up$;
1270 NEXT i
1280 :
1290 GOSUB 3000:REM test for hit/score
1300 :
1310 REM **** scroll back up ****
1320 LOCATE #1,1,17
1330 FOR i=1 TO 16
1340 GOSUB 4000:REM move snake
1350 PRINT #1,down$;
1360 NEXT i
1370 :
1380 NEXT count
1390 :
1400 CLS #1:CLS
1410 LOCATE #1,6,6:PRINT #1,"Game Over"
1420 END
```

```
1430 :
2000 REM **** initialisation ****
2010 blue=0:red=3:black=5:green=12:white
=4
2020 zero$=STRING$(5,"0")
2030 snake$=STRING$(2,CHR$(231))
2040 SYMBOL AFTER 250
2050 SYMBOL 250,0,126,120,126,120,126,12
0,126
2060 SYMBOL 251,127,127,127,255,255,255,
255,240
2070 SYMBOL 252,128,124,114,255,255,255,
255,254
2080 boot1$=CHR$(250)
2090 boot2$=CHR$(251)
2100 boot3$=CHR$(252)
2110 bell$=CHR$(7)
2120 up$=CHR$(11)
2130 down$=CHR$(10)
2140 sx=10:sy=7:x=1:y=1:dx=1
2150 WINDOW 1,20,19,25
2160 PAPER blue:CLS:PEN green
2170 WINDOW #1,1,20,2,18
2180 PAPER #1,red:CLS #1:PEN #1,black
2190 WINDOW #2,1,20,1,1
2200 PAPER #2,blue:CLS #2:PEN #2,white
2210 LOCATE #2,8,1:PRINT #2,"Score: 0000
0"
2220 RETURN
2230 :
3000 REM **** test for hit ****
3010 gx=32*x-16:gy=104:t1=TEST(gx,gy):t2
=TESTR(32,0)
3020 IF t1=green OR t2=green THEN sc=sc+
50:PRINT bell$
3030 IF t1=green AND t2=green THEN sc=sc
+100:PRINT bell$
3040 score$=STR$(sc)
3050 lgth=LEN(score$):score$=RIGHT$(scor
e$,lgth-1)
3060 lgth=LEN(score$)
3070 score$=LEFT$(zero$,5-lgth)+score$
3080 LOCATE #2,15,1:PRINT #2,score$
3090 FOR i=1 TO 500:NEXT i:REM delay
```

```
3100 RETURN
3110 :
4000 REM **** move snake ****
4010 sx=sx+1-INT(RND(1)*3)
4020 IF sx<1 THEN sx=1
4030 IF sx>18 THEN sx=18
4040 LOCATE sx,sy:PRINT snake$
4050 PRINT CHR$(10);
4060 RETURN
4070 :
5000 REM **** move boot ****
5010 LOCATE #1,x,y:PRINT #1,boot1$
5020 LOCATE #1,x,y+1:PRINT #1,boot2$;boo
t3$
5030 INK black,0
5040 RETURN
5050 :
6000 REM **** rub out boot ****
6010 INK black,3
6020 LOCATE #1,x,y:PRINT #1," "
6030 LOCATE #1,x,y+1:PRINT #1,"   "
6040 RETURN
```

Logical plotting

Perhaps the most interesting addition to the Amstrad CPC range's graphics by the control characters is the ability to perform logical operations when plotting high-resolution graphics. We select the operation we require by passing a parameter between 0 and 3 with `CHR$(23)`. The parameter value sets the logical operation type as follows:

Parameter	Logical operation
0	Normal plotting mode
1	Perform XOR with colour already there
2	Perform AND with colour already there
3	Perform OR with colour already there

When one of the logical plotting modes is selected (by setting the parameter to 1, 2 or 3) a logical operation is performed between the colour already present on the screen at the point to be plotted and the new graphics colour specified.

Note that these colours are represented by their logical colour mode numbers.

To see how a logical plot works press **CTRL/SHIFT/ESC** and type in the following short program:

```
10 REM **** LOGICAL 'AND' DEMO ****
20 MODE 0
30 PAPER 7:CLS
40 PRINT CHR$(23)+CHR$(2):REM SET 'AND'
MODE
50 MOVE 0,0:DRAW 300,300,14
```

In mode 0 there are 16 colours available, each colour code, 0–15, being held as a four-bit number. The program starts by setting the screen colour to LMCN 7, normally bright magenta, in line 30. After setting the plot mode to AND the program draws a line in LMCN 14, normally flashing blue/yellow. However, we do not see a flashing blue/yellow line. Instead, the line is bright blue, i.e. LMCN 6. If we look at the bit patterns for the LMCNs used we can see why the line is the 'wrong' colour.

Comments	LMCN	Bit pattern	Colour seen
Paper colour	7	0111	Bright magenta
Graphics foreground colour	14	1110	Flashing blue/yellow
Resulting colour after AND	6	0110	Bright blue

Note that the logical functions work differently in different modes. For example, in mode 1 only four colours are available. These can be represented by two bits only. We can predict what will happen in mode 1 by ignoring the two most-significant columns in the four-bit pattern. This time the paper colour will be bright red (LMCN 3, i.e. 11 in binary) and the line colour will be bright cyan (LMCN 2, i.e. 10 in binary).

Similarly, in mode 2 there are only two colours available, so we need only consider the least-significant column in the four-bit pattern. The paper colour in mode 2 will be bright yellow (LMCN 1) and the line will be blue (LMCN 0).

Performing an OR operation, instead of AND, by setting the parameter to 3, would yield different results. In mode 0 the colour of the line would be flashing pink/sky blue (LMCN 15, i.e. 1111 in binary).

XOR plotting

The three logical plotting modes can be used to change the graphics foreground colour as it falls over different backgrounds. The most useful of the three logical operations possible is the XOR function as it can be used to erase previously plotted lines without disturbing the colours behind the lines. We can see why if we look at the bit patterns. Let's say we are drawing a white line on a red background in mode 0:

Comments	LMCN	Bit pattern	Colour seen
Paper colour	3	0011	Red
Graphics foreground colour	4	0100	White
Resulting colour after XOR	7	0111	Bright magenta
Draw again in same colour	4	0100	
Result after second XOR plot	3	0011	Red, i.e. paper colour restored

The result of performing two XOR operations on the same set of pixels (i.e. redrawing the same line) is to restore the background colour. The only problem is that we don't get the line colour we wanted. Instead of a white line we get a magenta line. We must, therefore, choose our foreground graphics colour carefully. If we really wanted a white line (LMCN 4) on a red background (LMCN 3) then we would plot the line using LMCN 7, as 3 XOR 7 is 4.

This little demonstration program shows how XOR can be used to remove a foreground line, no matter what colour the background. Bands of colour are put on the screen by defining a series of text windows. The line is then drawn in XOR mode, and then redrawn after a short delay, erasing the original line and restoring the background colours.

```
10  REM **** XOR Demo ****
20  MODE 0:PAPER 3:CLS
30  PRINT CHR$(23)+CHR$(1):REM set XOR
mode
40  FOR wind=1 TO 7
50  WINDOW #wind,1,20,3*wind,3*wind+2
60  PAPER #wind,wind:CLS #wind
70  NEXT wind
```

```
80  FOR i=1 TO 2
90  FOR delay=1 TO 2000:NEXT delay
100 MOVE 0,0:DRAW 300,300,4
110 NEXT i
```

An alternative method of selecting the appropriate logical plotting mode on the Amstrad CPC 664 is to specify a fourth parameter in a graphics command, using a number in the range 0–3. Thus on the Amstrad CPC 664:

```
DRAW 100,200,2,1
```

draws a line to point `(100,200)` in LMCN 2 using the `XOR` plotting mode. Amstrad CPC 664 owners should note that the `CHR$` methods outlined above will work correctly on their machines.

'Amstradioids'

Much of what we have learned in this chapter and previous chapters can be illustrated by this short game program. Based on the classic space invaders theme 'Amstradioids' uses downward screen scrolling to make the invaders move down towards the laser base. The beams from the laser base are plotted using the `XOR` plotting mode, allowing them to be easily erased without disturbing the background. So that the laser base does not get scrolled along with the alien characters use is made of windows. In fact four windows are used each covering the width of the screen but occupying different vertical sections. Only window 1 is scrolled; thus, making the alien characters appear and move down is simple. The characters are simply printed on the top line of window 1 at random horizontal positions at regular intervals using one interrupt routine; a second interrupt routine scrolls window 1, again at regular intervals. If characters reach the bottom of the window without being destroyed by the laser base, they simply disappear from view. At present, the game only ends if an alien lands on the red laser base area, but you might wish to add to this, perhaps introducing other areas on the ground area that need defending against alien attack.

The cross-hair cursor movements are controlled using `INKEY` numbers to detect cursor key or joystick control. As all high-resolution graphics are plotted in XOR mode, making the cursor move is extremely easy. To erase an old cursor position before plotting a new one, we simply need to draw the cursor again, in its old position, before changing the coordinates and replotting. The routine that scans the cursor keys and joystick port calls the cursor-drawing routine, starting at line 7000, twice for this reason. The other time we might wish to erase the cursor is during the scrolling-interrupt routine, as, if the cursor is visible during the scroll routine, it will move down one character cell, and, therefore, will not be erased by a subsequent XOR plot to its original position. As the interrupt can occur in the middle of the key-scanning routine, we are not sure whether the cursor is currently visible or not. For example, if the scroll routine interrupted the main program between lines 4010 and 4110 then the cursor would not be visible, but if the interrupt occurred in any other portion of the main program then the cursor would be on. Because of the plot/replot method of erasing the cursor, it is important for the scroll routine to know whether the cursor is visible or not, when it interrupts. To provide this information the program makes use of a flag, curflag, which toggles between 0 and 1, each time the cursor-drawing routine is called. The scroll routine, therefore, will only erase the cursor prior to scrolling if it is currently on, indicated by curflag having the value 1. Line 2090 of the scroll routine checks curflag and erases the cursor if curflag =1, performs the scroll and then redraws the cursor again. If curflag is not 1, then the cursor is already off and all that is required is to scroll the screen.

This explanation serves to illustrate some of the difficulties of using interrupt routines. We must always be aware of what changes the interrupt will make to the main program and, if necessary, provide the interrupt routine with sufficient information to restore the original conditions after it has done its job. In the example described above, the cursor will have the same status after the interrupt as before it, i.e. if the cursor was visible it will still be visible after returning from the interrupt. Similarly, if it was off when the scroll routine interrupted, then it will be off on return to the main program. Because the interrupt will always restore the cursor status we do not need to check curflag when drawing or erasing the cursor in our key/joystick-scanning routine.

```
10 REM ******************************
20 REM ******************************
30 REM **                          **
40 REM **       Amstradioids       **

50 REM **                          **

60 REM ******************************

70 REM ******************************
80 :
110 GOSUB 1000:REM initialise
120 WHILE endflag=0
130 GOSUB 4000:REM scan keys/joystick
140 WEND
150 GOSUB 8000:REM game over
160 END
170 :
1000 REM **** initialisation ****
1010 MODE 0
1020 PRINT CHR$(23)+CHR$(1):REM XOR plot
 mode
1030 SPEED KEY 5,5
1040 zero$=STRING$(6,"0")
1050 green=12:black=5:red=3:white=4
1060 block$=STRING$(3,CHR$(143))
1070 base$=CHR$(244):up$=CHR$(11)
1080 explode$=CHR$(238)
1090 score$="000000"
1100 ds=8:curx=320:cury=200
1110 REM ** set up envelopes **
1120 ENV 1,3,5,1,1,0,20,5,3,1
1130 REM ** alien array **
1140 aliennum=6:DIM aliencode(aliennum)
1150 FOR i=1 TO aliennum
1160 READ aliencode(i):NEXT i
1170 DATA 171,226,224,225,252,253
1180 REM ** define windows **
1190 WINDOW 1,20,23,23:PAPER black:PEN r
ed:CLS
1200 WINDOW #1,1,20,2,22:PAPER #1,black:
CLS #1
1210 WINDOW #2,1,20,1,1:PAPER #2,black:P
EN #2,white:CLS #2
```

```
1220 LOCATE #2,4,1:PRINT#2,"Score 000000
"
1230 WINDOW #3,1,20,24,25:PAPER #3,green
:CLS #3
1240 basex=10:basey=1:REM start coords
1250 firex=32*basex-16:firey=48
1260 LOCATE basex,basey:PRINT base$
1270 PEN #3,red:LOCATE #3,basex-1,1:PRIN
T #3,block$
1280 REM ** set up interrupts **
1290 scrollrate=80:alienrate=90
1300 EVERY scrollrate,1 GOSUB 2000:EVERY
 alienrate,2 GOSUB 3000
1310 GOSUB 7000:REM initial cursor posit
ion plot
1320 RETURN
1330 :
2000 REM **** scroll interrupt ****
2010 alienrate=alienrate-2:IF alienrate<
30 THEN alienrate=30
2020 SOUND 2,4000-c,50,5:c=2000-c/2:IF c
>3900 THEN c=3900
2030 REM ** check for overrun **
2040 LOCATE #2,1,1
2050 FOR scan=-32 TO 32 STEP 32
2060 t=TEST(firex+scan,firey+6):IF t<>bl
ack AND t<>red THEN endflag=1
2070 NEXT scan
2080 LOCATE #1,1,1
2090 IF curflag=1 THEN GOSUB 7000:PRINT
#1,up$;up$:GOSUB 7000 ELSE PRINT #1,up$;
up$
2100 RETURN
2110 :
3000 REM **** place a character interrup
t ****
3010 EVERY alienrate,2 GOSUB 3000
3020 SOUND 4,INT(RND(1)*300),30,5
3030 alienx=INT(RND(1)*20+1)
3040 code=INT(RND(1)*aliennum+1)
3050 LOCATE #1,alienx,1:PEN #1,code*2:PR
INT#1, CHR$(aliencode(code));
3060 RETURN
3070 :
```

```
4000 REM **** move cursor ****
4010 GOSUB 7000:REM rubout cursor
4020 dx=0:dy=0
4030 IF INKEY(1)=0 OR INKEY(75)=0 THEN d
x=ds:dy=0
4040 IF INKEY(8)=0 OR INKEY(74)=0 THEN d
x=-ds:dy=0
4050 IF INKEY(0)=0 OR INKEY(72)=0 THEN d
x=0:dy=ds
4060 IF INKEY(2)=0 OR INKEY(73)=0 THEN d
x=0:dy=-ds
4070 IF INKEY(9)=0 OR INKEY(76)=0 THEN G
OSUB 5000:REM fire
4080 curx=curx+dx:cury=cury+dy
4090 IF curx<0 THEN curx=639
4100 IF curx>639 THEN curx=0
4110 IF cury<firey OR cury>383 THEN cury
=firey
4120 GOSUB 7000:REM draw cursor
4130 RETURN
4140 :
5000 REM **** fire ****
5010 DI
5020 REM ** shoot **
5030 SOUND 1,20,0,0,1,0,1
5040 MOVE firex,firey:DRAW curx,cury,1

5050 FOR delay=1 TO 100:NEXT delay
5060 MOVE firex,firey:DRAW curx,cury,1
5070 REM ** hit ? **
5080 IF TEST(curx,cury)<>black THEN GOSU
B 6000:REM hit
5090 EI
5100 RETURN
5110 :
6000 REM **** hit ****
6010 incscore=200*TEST(curx,cury):score=
score+incscore
6020 score$=STR$(score):lgth=LEN(score$)
-1
6030 score$=LEFT$(zero$,6-lgth)+RIGHT$(s
core$,lgth)
6040 LOCATE #2,10,1:PRINT#2,score$
6050 REM ** erase alien **
```

```
6060 charx=INT(curx/32)+1:chary=INT((383
-cury)/16)+1
6070 PEN #1,red:LOCATE #1,charx,chary:PR
INT#1,explode$
6080 FOR delay=1 TO 50:NEXT delay
6090 LOCATE #1,charx,chary:PRINT#1," "
6100 RETURN
6105 :
7000 REM **** plot cursor ****
7010 DI
7020 curflag=1-curflag
7030 MOVE curx-8,cury:DRAWR 16,0,6
7040 MOVER -8,8:DRAWR 0,-16
7050 EI
7060 RETURN
7070 :
8000 REM **** game over ****
8010 dummy=REMAIN(1):dummy=REMAIN(2):REM
 interrupts off
8020 SOUND 135,0:REM flush all sound que
ues
8030 MODE 0:PAPER 0:PEN 1
8040 LOCATE 6,12:PRINT"Game Over"
8050 LOCATE 2,14:PRINT"Your Score ";scor
e$
8060 FOR delay=1 TO 2000:NEXT delay
8070 LOCATE 2,16:PRINT"Another game (y/n
)"
8080 junk$=INKEY$:WHILE junk$<>""
8090 junk$=INKEY$:WEND
8100 a$="":WHILE a$<>"y" AND a$<>"n"
8110 a$=INKEY$:WEND
8120 IF a$="y" THEN RUN ELSE SPEED KEY 1
0,2:END
8130 :
```

Structure diagram

The structure of the 'Amstradioids' program is simple. The game comprises of three parts: an initialisation routine, where windows and characters are defined and interrupt timings selected; a main loop, where the keys/joystick are scanned, the cursor moved and the alien characters are scrolled onto the screen using interrupt routines; and a game-over section, where interrupts are disabled and the player is given the option of another game.

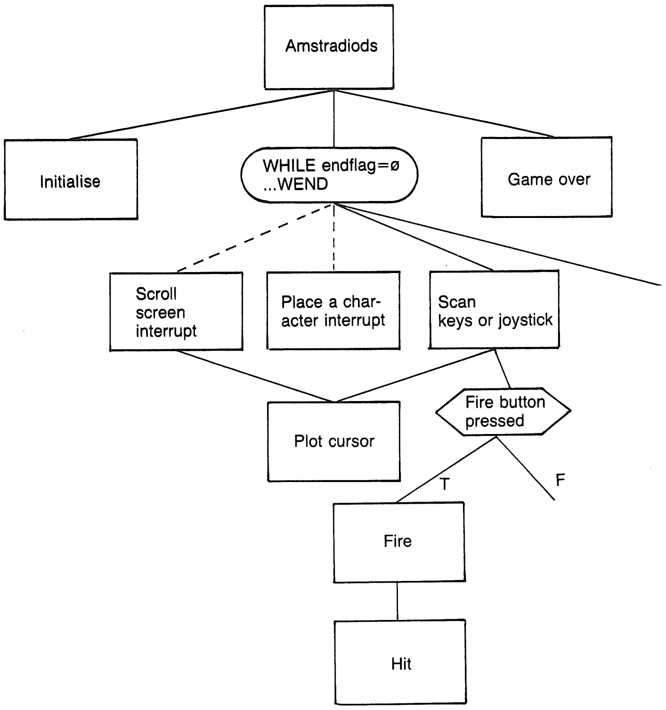


Figure 7.1

'Stranded' — climbing ladders and rescuing

In the last section of 'Stranded' we dealt with jumping on and off the moving panels. This included a routine to test underneath the player figure to see what it was standing on. We inserted dummy title and `RETURN` lines for the routine which deals with the case of the player being on a ladder. In this section we look at this routine, which completed the cliff-walk section of the program. We also look at the routines that allow the player to rescue men stranded on the pillars.

Moving up or down a ladder

The subroutine that checks under the figure (lines `47ØØ–482Ø`) sets a variable `lad` to 1, 2 or 3 if the player figure is standing on a ladder. The value of this variable relates to the ladder number (there are three ladders!) and can therefore be used to access the correct elements of the ladder arrays, so that the player can be made to move down or up the ladder and the ladder redrawn. Before the player can be moved we must determine whether it is at the top or the bottom of the ladder indicated by `lad`. This can be easily done by comparing the y coordinate of the player figure, `chary`, with that of the start point of the ladder, `ly(lad)`. A variable `dy` is set to 1 or −1 to indicate the direction in which the player figure must move. The value of `dy` is determined by lines `4910` and `4920`. The routine then enters a loop to move the player figure. When the figure is printed on the ladder the routine selects the transparent option so that only the parts of the ladder that lie directly behind the player will be erased. As each successive character position is erased the ladder is quickly redrawn by a simple `GOSUB` call to the individual ladder-drawing routine. As `lad` is also used by the ladder-drawing routine the correct ladder is automatically selected. To give the impression of climbing, the player figure is toggled between `CHR$(250)` and `CHR$(251)` from the standard Amstrad CPC range character set.

Having completed the loop, the player figure will now be standing at the top or bottom of the ladder in question. One problem of an arrangement where the figure is automatically moved down a ladder if found to be standing at the top, is that if the figure has just climbed the ladder then it will be detected as standing at the top of the ladder and automatically sent down again. The player figure will in fact climb up and down the ladder endlessly unless we do something to avoid this undesirable feature. The simplest way is to set a flag to indicate that the ladder has just been negotiated and insert a test in the 'check under figure' routine to `RETURN` if the flag is set. Line `5040` sets `ladflag` to 1, before increasing the score by 100. If the ladder just scaled was the third, i.e. the one down to the jetty, then we wish to exit our main loop and enter a new routine that deals with that section of the game. Again a flag is used for this purpose: `boatflag` is set to 1 in line `5050`.

During the 'check under figure' and 'up or down a ladder' routines interrupts have been disabled. If the player has to go up or down a ladder then the interrupts are disabled for so long that a queue will build up. To flush this queue (see interrupts in Chapter 5 for more details) we reset the timers at line 5060.

```
4900 REM **** up or down a ladder ****
4910 IF chary=ly(lad)-1 THEN dy=-1
4920 IF chary=ly(lad)-ll(lad)-1 THEN dy=
1
4930 FOR y=1 TO ll(lad)
4940 ladchar=1-ladchar
4950 LOCATE charx,chary:PRINT" "
4960 GOSUB 6500:REM redraw ladder
4970 chary=chary+dy
4980 PRINT transon$
4990 LOCATE charx,chary:PRINT CHR$(250+l
adchar)
5000 PRINT transoff$
5010 FOR delay=1 TO 100:NEXT
5020 GOSUB 7800:REM sound fx
5030 NEXT y
5040 ladflag=1:incscore=100:GOSUB 7700:R
EM inc score
5050 IF lad=3 AND boatflag=0 THEN boatfl
ag=1
5060 resetflag=1:REM reset timers
5070 RETURN
```

ladflag is only cleared when the player figure moves from the top or bottom of the ladder under cursor key or joystick control. These two lines must be inserted in the 'move left' and 'move right' routines to do this:

```
4050 IF ladflag=1 THEN ladflag=0:GOSUB 6
500
4150 IF ladflag=1 THEN ladflag=0:GOSUB 6
500
```

Rescuing the men

If the third ladder was descended then, as described above, boatflag will be set to 1 and the main loop between lines

1460 and 1480 will be exited. The next stage is to walk down the jetty and get in the boat. This section of the game is handled by a subroutine at line 5100 which is called by this line from the main program:

```
1520 GOSUB 5100:REM down gangway
```

The jetty is formed from text window 3, to move the man on the jetty we must therefore switch to print to stream 3. As the character coordinates within any window are related to the top-left corner of the window we must also change the values of charx and chary to fit in with this new system. In addition we must change the current PEN colour in window 3 to be that of the player figure. We shall see later that although this is normally white, it can be red, signifying that the player figure is carrying a rescued man.

Once we have made the necessary changes to relate to window 3, we can move the player figure down the jetty. Again, CHR$(250) and CHR$(251) are toggled to produce a walking effect. At the end of this routine the player is made to get into the boat. The sea area is defined as window 2, so, again, we must change charx and chary to fit in with the coordinate system for this window. Finally, the standman$ character and the boat$ character are printed in different colours within the same character square, using the transparent option, by a further subroutine at line 6000.

```
5100 REM **** down gangway ****
5110 LOCATE charx,chary:PRINT" "
5120 GOSUB 6500:REM redraw ladder
5130 charx=1:PEN #3,mancol:REM change to
 stream 3
5140 FOR chary=2 TO 5
5150 tog=1-tog
5160 LOCATE #3,charx,chary:PRINT#3, CHR$
(250+tog)
5170 FOR i=1 TO 100:NEXT:REM delay
5180 LOCATE #3,charx,chary:PRINT#3," "
5190 GOSUB 7800:REM sound fx
5200 NEXT chary
5210 REM **** onto boat ****
```

```
5220 charx=18:chary=7:REM change to stre
am 2
5230 manink=whiteink:GOSUB 6000:REM prin
t man/boat
5240 RETURN
6000 REM **** print man and boat ****
6010 PRINT#2,transon$
6020 INK 15,blueink
6030 LOCATE #2,charx,chary:PEN#2,15:PRIN
T#2,standman$
6040 LOCATE #2,charx,chary:PEN#2,green:P
RINT#2,boat$
6050 INK 15,manink
6060 PRINT#2,transoff$;
6070 RETURN
6080 :
```

Steering the boat

Once the player figure is in the boat then the program enters a new loop within the main program, allowing the boat to be steered left or right across the screen. Two routines exist to scan the cursor keys or joystick, and are similar to those used to control the player figure on the cliff-walk:

```
5500 REM **** scan boat joystick ****
5510 DI
5520 IF JOY(0)=left THEN GOSUB 5600
5530 IF JOY(0)=right THEN GOSUB 5700
5540 IF JOY(0)=fire THEN GOSUB 5800
5550 EI
5560 RETURN
8500 REM **** steer by cursor ****
8510 DI
8520 IF INKEY(8)=0 THEN GOSUB 5600
8530 IF INKEY(1)=0 THEN GOSUB 5700
8540 IF INKEY(47)=0 THEN GOSUB 5800
8550 EI
8560 RETURN
```

As before, the correct routine is selected by testing the valu of `joyflag` within the main program loop. These line should be added to the main program section:

```
1540 REM **** steer boat ****
1550 :
1560 WHILE fetchflag=0 AND endflag=0
1570 IF joyflag=1 THEN GOSUB 5500 ELSE G
OSUB 8500
1580 WEND
```

Two routines are used to move the boat and man to the left or right under cursor key or joystick control. Notice that `fetchflag`, one of the flags that can terminate the `WHILE...WEND` loop above, is set to 1 if the boat is moved right to make `charx` =18. This horizontal coordinate corresponds to the boat being next to the jetty. In this instance the program must move on to the next program section: to get out of the boat and move back along the jetty.

```
5600 REM **** move boat left ****
5610 LOCATE #2,charx,chary:PRINT#2," "
5620 charx=charx-1:IF charx<1 THEN charx
=1
5630 GOSUB 6000:REM print man/boat
5640 GOSUB 7800:REM sound fx
5650 RETURN
5700 REM **** move boat right ****
5710 LOCATE #2,charx,chary:PRINT#2," "
5720 charx=charx+1:IF charx>18 THEN char
x=18
5730 GOSUB 6000:REM print man/boat
5740 GOSUB 7800:REM sound fx
5750 IF charx=18 THEN fetchflag=1
5760 RETURN
```

At this stage it is worth noting the rather special way in which boat and man are moved by the subroutine at line `6000`. During the programming of this section it was found that moving the boat and man together caused the man character to flicker badly, caused by the delay between erasing it and reprinting it in its new position. To get around this problem the subroutine initially sets the `INK` colour, 15, that will be used to print the man to the background colour, blue. Only after both characters have been printed is `INK 15` redefined to be the correct colour for the man character. In this way the flicker is reduced.

Rescuing a man

A man can be rescued by manoeuvring the boat next to the pillar on which he is standing and pressing the joystick fire button, or the space bar. In either case a rescue routine will be called. Up to now we have not dealt with the seven men on the top of the pillars. One of two fates await them: they can be rescued by the player, or they can be drowned as the tide washes over the top of the pillar. For the purposes of the game the program needs to know the status of each pillar man: is he safe, drowned or still standing on the pillar? To keep track of the status of each man two arrays are used, each with seven elements. If the man is rescued then the correct element in `safeflag()` is set to 1. If, however, the pillar man is drowned then the corresponding element of `drownflag()` is set to 1. These arrays, along with an array `maxlev()`, used to indicate the maximum safe water level for each pillar, are dimensioned at line `1230`:

```
1230 DIM drownflag(7),safeflag(7),maxlev(7)
```

The following routine is called if the fire button or space bar is pressed whilst the boat is next to a pillar:

```
5800 REM **** rescue man ****
5810 pilln=charx/2
5820 IF pilln<>INT(pilln) OR charx>14 TH
EN RETURN
5830 IF carryflag=1 THEN RETURN
5840 IF safeflag(pilln)=1 THEN RETURN
5850 carryflag=1:safeflag(pilln)=1
5860 remht=5+pillarh(pilln)-level+pillar
y
5870 LOCATE #2,pillcx(pilln)+1,pillcy-pi
llch(pilln)-17
5880 IF remht<0 THEN PRINT#2," " ELSE PE
N #2,pastgreen:PRINT#2,bar$
5890 manink=redink:GOSUB 6000
```

```
5900 mancol=red
5910 incscore=1000-10*remht:GOSUB 7700:R
EM inc score
5920 GOSUB 7900:REM sound fx
5930 RETURN
```

The pillar number can be easily found by dividing the boat's horizontal coordinate by 2. If this is not an integer, or the value of charx exceeds 14, then the boat is not directly next to a pillar and no further action is taken. Another possibility is that the player has already rescued one man and has not yet returned this man to the top of the cliff-walk. This condition is detected by testing the status of carryflag used to indicate that the player is already carrying one rescued man. If so then, again, no further action is taken. If no RETURN is made during these preliminary tests then carryflag and the relevant element of the safeflag() array are set to 1, to indicate that the man on this pillar has been saved. The man on the top of the pillar must then be erased. Normally this will be done by printing, not a space, but a character where the lower part is filled, in the same position as the man was. This character is referred to as bar$ in the program and must be defined during the initialisation section. Insert this line:

```
1860 bar$=CHR$(210)
```

There is a chance that the man might be rescued just as the tide is washing around his ankles, in which case we would not want to print bar$ but merely a space. To determine the water level when the rescue takes place, a variable remht is calculated from the height of the pillar, the level of the water and a correction factor to account for the fact that the green top of the pillar is not included in its height, pillarh().

Having erased the man on the pillar, the player figure's colour is changed to red, to indicate that he is carrying a rescued man. Note that the player figure's INK colour must be changed along with its mancol PEN colour, so that the subroutine that prints man and boat together, at line 6000, will work correctly, as previously described. The score is then incremented by a factor calculated from the closeness of the rescued man to drowning — the later a man is rescued, the higher the bonus! Finally, a ship's horn sound effect signals that the man has been successfully rescued before a return is made to the main program. The siren sound-effect subroutine is as follows:

```
7900 REM **** siren sound effect ****
7910 DI:FOR i=1 TO 2
7920 FOR per=400 TO 250 STEP -5
7930 SOUND 1,per,2,5
7940 SOUND 2,per+100,2,5
7950 NEXT per,i
7960 SOUND 1,100,40,2
7970 EI:RETURN
```

Back up the gangway

When the boat is steered back to the jetty then `fetchflag` is set to 1, terminating the `WHILE...WEND` loop that controls the boat-steering section of the game. The next event is for the player figure to go back up the gangway. This is accomplished by calling a subroutine from line `1610` of the main program:

```
1610 GOSUB 5300:REM back up gangway
5300 REM **** back up gangway ****
5310 LOCATE #2,charx,chary:PEN #2,green:
PRINT#2,boat$
5320 charx=1:PEN #3,mancol:REM change to
 stream 3
5330 FOR chary=5 TO 2 STEP -1
5340 tog=1-tog
5350 LOCATE #3,charx,chary:PRINT#3, CHR$
(250+tog)
5360 FOR i=1 TO 100:NEXT:REM delay
5370 LOCATE #3,charx,chary:PRINT#3," "
5380 GOSUB 7800:REM sound fx
5390 NEXT chary
5400 charx=19:chary=17:REM change to str
em 1
5410 PEN mancol:LOCATE charx,chary:PRINT
 standman$
5420 lad=3:DI:GOSUB 4900:EI:REM up ladder
5430 RETURN
```

This subroutine effectively does the reverse of the subroutine at 51ØØ, discussed earlier, taking the player figure up the jetty and up the ladder before returning. Because the main program is not yet complete, the game will only be fully playable up to this point.

Drowning

Now that we have discussed the existence of two flag arrays to keep track of each pillar man's status. it seems a good time to insert some code into the 'raise water level' interrupt routine that causes the tide to rise. Line 618Ø calls a subroutine at 63ØØ if the water level has risen sufficiently to cover a pillar man.

```
6180 IF level>maxlev(pn) AND drownflag(p
n)=0 AND safeflag(pn)=0 THEN GOSUB 6300
```

Notice that the relevant elements of drownflag() and safeflag() must both be 0 before the 'drown' subroutine is called. This small subroutine then sets the correct element of drownflag() and calls a sound effect to indicate to the player that another man has drowned.

```
6300 REM **** drowned man ****
6310 IF safeflag(pn)=1 THEN RETURN
6320 drownflag(pn)=1
6330 GOSUB 8000:REM sound fx
6340 RETURN
8000 REM **** drown sound efect ****
8010 DI:FOR per= 250 TO 400 STEP 5
8020 SOUND 1,per,3,5
8030 NEXT per
8040 EI:RETURN
```

Structure diagram

The routines covered in this chapter make a substantial addition to the structure diagram of the program. We have added routines to record drowning men and move the player figure on the ladders, but have also introduced three new elements to the main program: a down-jetty routine, a move-boat loop and a back-up-jetty routine. Notice how the interrupt routines that raise the water level and move the panels also interrupt our boat-moving loop.

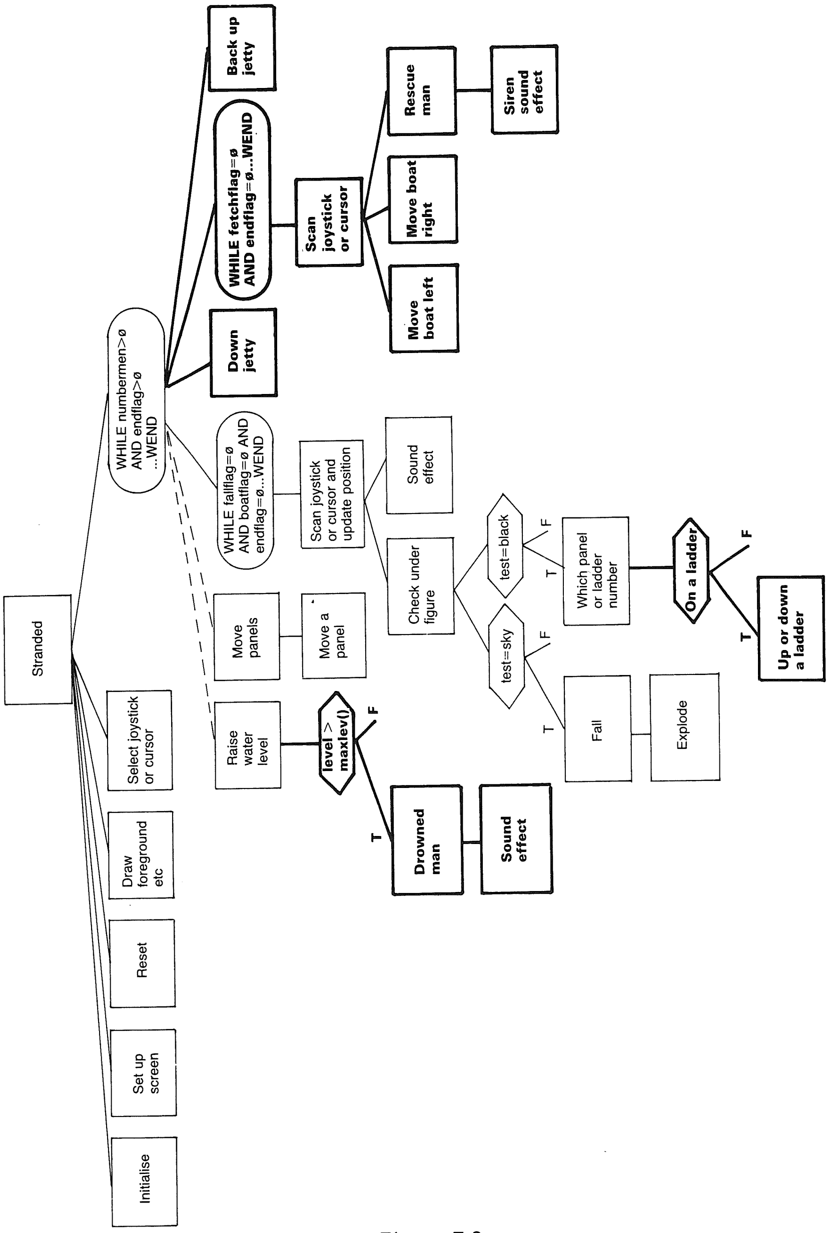


Figure 7.2

Chapter Eight

Programming tips and hints

In this chapter:

Structure versus speed

Throughout this book the approach has been to teach good programming skills through learning to program graphics games. One of the spinoffs of learning to program through graphics is that it is easy to see and understand the interplay and relationships of different sections of the program. It is becoming generally recognised that programming in structured units is, in most circumstances, the best style of programming, although there are exceptions to this rule. Although BASIC does not make it easy to structure programs, as opposed to say Pascal, I have tried to demonstrate some of the techniques that can be used in BASIC to get around its difficiencies. As we shall see by the end of this chapter, the major program 'Stranded' uses only three GOTO statements in all. Although the aim of structuring is not simply to avoid using GOTO, the fact that it is possible to write such a long and complex program without resorting to their use does indicate a reasonable program structure. In Chapter 1 examples of BASIC structuring techniques were given along with some examples of dealing with the same programming problem in an unstructured way. The development of 'Stranded' over the course of the book in simple stages has only been possible because of the structured manner of the program — indeed presenting the same program written in an unstructured way would probably have not even been possible — the reader becoming lost in an ever more complex web as program control darted around the listing. I hope by this stage in the book you will appreciate that a well-structured program is also a more readable and easily understood program; each section having a clearly defined and labelled purpose within the program as a whole.

As mentioned earlier there are cases where structuring is not desirable, most notably in the quest for speed. Structured versions of programs will not usually run as fast as an unstructured version: the reasons for this are more to do with the difficulty of structuring BASIC programs than the efficiency of unstructured ones. To illustrate why structured programs tend to run slightly slower let us take the example of leaving a loop under a certain terminating condition. In an unstructured format the loop may be controlled by an IF...THEN statement that tests for the terminating condition, looping back while the condition remains false. In a structured program we might use a WHILE...WEND loop to do much the same thing, but let us imagine that the terminating condition is set within a subroutine that is called from within the loop. To terminate the loop in this case we would probably need to set a flag variable within the subroutine if the terminating condition was found to be true. The value of this flag would then be tested again by the WHILE...WEND loop after return from the subroutine had been made. In the structured format we are in fact doing approximately twice the work that needs to be done in the unstructured version — we perform a test and set a flag, and then perform a test on the flag to actually terminate the loop. In most applications the slight reduction in speed that results is not significant. The only time where it is likely to be important is when programming games that require high-speed movement on the screen. In this case it is likely that to produce the speed needed that BASIC would be too slow anyway and the programmer would have to use machine code. If, however, maximum speed in BASIC is your aim then there are ways that speed can be improved whilst still retaining a good structure:

1) Place the subroutines used most often within the program at the beginning, arranging the other subroutines in order of use, the one used least frequently being at the end of the program. When BASIC obeys a `GOSUB` call it searches through the program from the beginning for the line number given in the `GOSUB` call. Obviously, if the subroutine is nearer the start of the program then it will be found quicker.
2) Use variable names and constants rather than actual numbers so that BASIC does not have to go through the process of reconverting into binary format every time the number is encountered within the program.
3) Try to ensure that you are not performing unnecessary tasks within looping structures — Could the task be done just once before entering the loop? Are you testing for conditions at the right place within a program block — do you then go on to do jobs that are not required if the condition you are testing for is true/false?
4) Make use of the user-defined function `DEF FN` to do often-repeated calculations. For example:

```
10 DEF FN hypotenuse=SQR(opp 2+adj 2)
20 opp=10:adj=2:PRINT FN hypotenuse
```

Note that the `DEF FN` must come before the `FN` statement that uses it.

Programmers aids

Definable keys

Locomotive BASIC offers a number of added facilities to make the programmer's task less of a chore. When typing in a program that uses the same phrase or keyword it can be annoying to have to type in, or use the cursor keys to copy, the same things time after time. For this reason the Amstrad CPC range allow you to refine the numeric keypad to the right of the main keyboard so that when you press a key it will automatically print a phrase or keyword. Typing in this, for example:

```
KEY 139,"MODE 1:PEN 1:PAPER
Ø:LIST"+CHR$(13)
```

redefines the **ENTER** key on the numeric keypad to list a program in mode 1, whenever the key is pressed. The `CHR$(13)` at the end is the equivalent of pressing the main **ENTER** key, as you would after typing in a command. This can be useful when writing and debugging a game that uses mode 0, as a program listed to the screen in mode 0 is difficult to read. By omiting the `CHR$(13)` on the end of the key-defining expression, we can define the key as a way of entering a regularly used keyword. For example:

```
KEY 139,"CHR$("
```

defines the numeric keypad **ENTER** key to print `CHR$(`. We can then add the code number and bracket in the usual way. All 12 keypad keys can be defined in this way. Regular programmers may even wish to write a short program that they load in and run to program these keys to their own specifications before starting a programming session. The keypad **ENTER** key has a code of 139 (known as its expansion code) as used in the examples above. The complete list of codes is shown on the following page.

Renumbering programs

Locomotive BASIC provides the programmer with a fairly powerful renumbering facility. You may need to renumber your program to fit in extra code, or simply to make the listing more attractive once the program is completed. The `RENUM` command can have up to three parameters:

```
RENUM old ln,new ln,step size
```

Key	Expansion code
0	128
1	129
2	130
3	131
4	132
5	133
6	134
7	135
8	136
9	137
.	138
ENTER	139
RUN" (CTRL/ENTER)	140

All GOSUBs and GOTOs are automatically changed to take account of renumbered sections of the program. If all three parameters are left out then the program is renumbered from line 10 in steps of tens. Just using the first parameter declares the starting line number, and the program is incremented in tens. Using the first and second parameters together allows you to renumber sections of the program, with the optional third parameter allowing you to specify step sizes other than tens if required.

The trace option

This debugging aid prints the line number being executed in square brackets on the screen if enabled by the command TRON. Because it interferes with the screen display it is often only of use to graphics programmers for debugging small sections of code that do not output to the screen. However, in other programming areas the trace option can be useful in tracking down the line number where things are going wrong. It is disabled by the TROFF command.

Trapping program breaks

When a program is running there are basically three ways in which it can end: it can finish successfully according to the correct flow of the program; it can stop due to an error; or it can stop because the **ESC** key has been pressed twice. It is often useful to be able to trap these last two sources of interruption to the program and Locomotive BASIC makes this very easy.

Errors can be trapped using the ON ERROR GOTO command. This command, unlike most BASIC commands, needs to be issued only once at the beginning of the program to remain in force until the program ends normally, or another ON ERROR GOTO command is issued. If an error occurs then the program jumps to the line number specified by this command, rather than stopping with an error message. We can write an error-handling routine that prints out the error number and the line at which it occurred, before resuming with the program. ERR and ERL are reserved variable names that give the error number and line number, respectively, the RESUME command allows the program to continue from where it left off, from the *next* line or from another specified line number. It may be that we want to ignore, say, the 'division by zero' error (that has error number 11) but halt the program if any other error occurs. The following example demonstrates this:

```
10   REM **** Error handling demo ****
20   FOR x=5 TO -5 STEP -1
30   y=25/x:PRINT y
40   NEXT x
50   SPLURDGE
60   GOTO 20
1000 REM ** Error handler **
1010 IF ERR=11 THEN PRINT"Div by zero but
what the heck!":RESUME NEXT
1020 PEN 2:PRINT"Error";ERR;"at
line;"ERL:PEN 1
1030 WHILE INKEY$="":WEND
1040 PRINT:LIST:END
```

The program registers that a 'division by zero' error has occurred but does not let that stop the program. However, the syntax error at line 50 is picked up and the program lists itself after a keypress.

A program break caused by the **ESC** key can be trapped in a similar way using the ON BREAK GOSUB command. The following lines cause the program to restart if an attempt is made to break into it using the **ESC** key.

```
10   ON BREAK GOSUB 1000
20   WHILE pigsfly=0:PRINT"hello ";
30   WEND
40   :
1000 REM **** Break handler ****
1010  PRINT:PRINT"Don't press my ESC
key, please!"
1020 RUN
```

The `ON BREAK GOSUB` trap can be disabled by `ON BREAK STOP`, which restores the normal function of the **ESC** key. Trapping the **ESC** key break should be used with caution. If used as above, the only way to break out is to reset the machine and lose your program! It is therefore a good idea to save your program first.

'Stranded' — completing the game

In this, the final section of 'Stranded', we look at the remainder of the routines that go to make up the game, including routines to check for an end-of-game situation and a routine to take the game to higher levels of difficulty, should the player successfully complete a section. We also look at the creation of a more professional-looking title screen.

Back up the ladders

The game is now at the stage where the player can negotiate the cliff-walk sections and ladders to reach the jetty and row out to the men stranded on the pillars. One of these men can be rescued and brought back to the jetty. The next stage is, therefore, to allow the player to go back up the ladders with the rescued man (signified by the player figure becoming red in colour) to deliver him to the start point at the top of the cliff. If the player manages to carry the man all the way to the top of the cliff then a substantial bonus is awarded.

We can obviously use the same routines for going up the cliff-face as we used to come down; we merely need to construct a `WHILE...WEND` loop, as before within the main program loop:

```
1630 REM **** back up ladders etc ****
1640 WHILE (charx<>1 OR chary<>3) AND fa
llflag=0 AND endflag=0
1650 IF joyflag=1 THEN GOSUB 3900 ELSE G
OSUB 8400
1660 WEND
```

Notice that there are three conditions that can terminate this loop. The first of these is if `charx` =1 *and* `chary` =3 (don't be fooled by the `OR` in the `WHILE` statement: remember that those are conditions that, while true, will not terminate the loop!). This position is the home position at the top of the cliff and signifies that the player figure has reached the top of the cliff. The second possible condition that terminates the loop is for `fallflag` to be 1. Obviously the player can fall off the cliff-walk on the way up as well as on the way down; `fallflag` is set by the 'check under figure' routine mentioned earlier in Chapter 6. The third way that the loop can end is if `endflag` is set to 1. More of this later. The first two conditions can be tested for once the loop has finished by these two lines:

```
1690 IF fallflag=1 THEN GOSUB 3000:GOTO
1730:REM end loop
1700 IF carryflag=1 THEN GOSUB 4600:REM
one home
```

If `fallflag` is set the flag reset routine at line `3000` is called and a jump is made to the `WEND` at line `1730`. Unless the number of lives left to the player, `numbermen`, is 0 the main program loop will then restart. If the player did not fall then we assume that the reason that the 'back up the ladders' controlling loop was terminated was because the player figure reached the top of the cliff. To avoid unscrupulous players gaining a bonus without having rescued a man, we test `carryflag`. This flag will be set if the player carries a rescued man. The subroutine that deals with the 'got one home' condition, called by line `1700`, is:

```
4600 REM **** got one home ****
4610 incscore=2000:GOSUB 7700:REM score
```

```
4620 menhome=menhome+1:GOSUB 7900:REM so
und fx
4630 PEN red:LOCATE 14,2:PRINT LEFT$(men
$,menhome)
4640 RETURN
```

This routine gives the player a bonus of 2000 points, increments the number of men home by one and shows this visually by displaying the number of men rescued and brought to the top of the cliff in red standing-man characters.

The final task within the main program loop is to perform the rest routines at lines 2900 and 3000, given earlier in the book, and check to see if there is an end-of-game situation.

```
1710 GOSUB 2900:GOSUB 3000:REM reset
1720 DI:GOSUB 3200:EI:REM no more pillar
 men?
```

Checking for an end-of-game situation

The game must either end or go on to the next level if all seven men on the pillars have either been rescued or drowned. There are several rules in the game that deal with this situation. If more than three men out of the seven are drowned then the game ends. If all seven men are successfully rescued then a 5000 bonus is added to the player's score. In any case other than three men drowning the game will go to the next level. The routine that checks for these situations is:

```
3200 REM **** any more pillarmen ****
3210 savcount=0:drowncount=0
3220 FOR pilln=1 TO 7
3230 savcount=savcount+safeflag(pilln)
3240 drowncount=drowncount+drownflag(pil
ln)
3250 NEXT pilln
3260 IF drowncount>3 THEN endflag=1:RETU
RN:REM end game
3270 IF savcount=7 THEN incscore=5000:GO
SUB 7700:REM bonus
3280 mancount=savcount+drowncount
```

```
3290 IF mancount<7 OR mancol=red THEN RE
TURN:REM same level
3300 :
3310 REM **** next level ****
3320 GOSUB 7300:REM disable timers
3330 REM ** reset flags **
3340 FOR pilln=1 TO 7
3350 safeflag(pilln)=0:drownflag(pilln)=
0
3360 NEXT pilln
3370 REM ** reset pillars **
3380 REM ** alter interrupts **
3390 CLS #2:GOSUB 2400:REM redraw sea sc
ene
3400 menhome=0:LOCATE 14,2:PRINT SPACE$(
7)
3410 moveint=moveint-2
3420 riseint=riseint-50
3430 IF moveint<4 THEN moveint=12:risein
t=300:REM reset to initial values
3440 GOSUB 7400:REM enable timers
3450 RETURN
```

Creating a new level in the game consists of the following steps that can be traced in the routine above:

1) Reset the `drownflag()` and `safeflag()` arrays.
2) Randomly redraw the pillars and the rest of the sea scene.
3) Zero the `menhome` counter.
4) Make the panel-moving and water-level-rising interrupt intervals shorter.

If you look at the main program loop you will see that each of its subloops can be terminated by setting `endflag` to 1. Why is this terminating condition included in each subloop if the only routine that can set `endflag` is not called until line `1720`? The answer lies in the fact that getting a man home (or falling off the cliff-walk in the attempt) is only one of the times when we need to check to see if we have an end-of-game or next level condition. The other time we need to call the routine above is when one of the pillar men is drowned. This can obviously only occur when the water level is raised. We should, therefore, call the above routine from the interrupt that causes the water level to rise. Insert this line into that routine:

```
6230 GOSUB 3200:REM more pillarman ?
```

Perhaps you can see why we need to be able to terminate any of the subloops within the main loop when `endflag` is set: because `endflag` can be set at any time in the main loop, whenever the 'raise water level' routine interrupts.

We now need to insert lines beneath each subloop so that if `endflag` is set we can jump straight to the main loop `WEND`, and fall through to an end-of-game routine beyond it. Insert these lines to do this:

```
1510 IF endflag=1 THEN 1730:REM end loop
1590 IF endflag=1 THEN 1730:REM end loop
1670 IF endflag=1 THEN 1730:REM end loop
```

The end-of-game routine is called by line `1740`:

```
1740 GOSUB 8100:REM end routine
```

and the actual routine starts at line `8100`:

```
8100 REM **** end routine ****
8110 GOSUB 7300:REM disable timers
8120 INK 14,3,24
8130 CLS:PEN 14:REM flash
8140 LOCATE 6,8:PRINT"Game Over"
8150 LOCATE 6,9:PRINT"========="
8160 IF score$>hiscore$ THEN hiscore$=sc
ore$
8170 PEN black:LOCATE 1,12:PRINT"Hi Scor
e:"
8180 PEN red:LOCATE 13,12:PRINT hiscore$

8190 PEN black:LOCATE 1,14:PRINT"Your Sc
ore:"
8200 PEN red:LOCATE 13,14:PRINT score$
8210 PEN green:LOCATE 2,18:PRINT"Another
 Game (y/n)"
8211 FOR p=600 TO 100 STEP -10
8212 SOUND 1,p,2,5
8213 NEXT p
8220 ans$=""
```

```
8230 WHILE ans$<>"y" AND ans$<>"n"
8240 ans$=INKEY$
8250 WEND
8260 score=0:score$="":REM zero score
8270 moveint=12:riseint=350:REM reset in
t timers
8280 GOSUB 3100:REM reset flags
8290 IF ans$="n" THEN endflag=1:CLS
8300 RETURN
```

The routine prints an end-of-game message and the highest score since the program was first run. If the new score is more than the previous highest then the new score will be adopted as the highest score. The player is then asked if another game is wanted. If the answer is `y` then the score is zeroed, the interrupt intervals are reset to their original values and the `safeflag()` and `drownflag()` arrays, together with `lad`, `panel` and `endflag` are reset to zero. If the answer is `n` then `endflag` is reset to 1. In either case a return to the main program is made. The flags are reset in their own routine at line `3100`:

```
3100 REM **** end of game flag reset ***
*
3110 FOR i=1 TO 7
3120 safeflag(i)=0:drownflag(i)=0
3130 NEXT i
3140 lad=0:panel=0:endflag=0
3150 RETURN
```

To deal with the possibility of a repeated game (signalled by `endflag` =0) we must finally enclose the set up routines and the main program loop within one further `WHILE...WEND` loop that terminates when `endflag` remains at 1, on return from the end-of-game routine above:

```
1300 WHILE endflag=0:REM game replay loop
1750 WEND:END
```

We can now see the complete program structure. The entire game loop looks like this:

```
1270 REM *******************
1280 REM * set up routines *
1290 REM *******************
1300 WHILE endflag=0:REM game replay loo
p
1310 numbermen=4:menhome=0
1320 GOSUB 2600:REM set up screen
1330 GOSUB 2800:REM foreground etc
1340 GOSUB 2900:GOSUB 3000:REM reset
1350 :
1360 REM ********************
1370 REM * main action loop *
1380 REM ********************
1390 :
1400 WHILE numbermen>0 AND endflag=0
1410 LOCATE charx,chary:PRINT standman$
1420 :
1430 REM **** negotiate ladders etc ****
1440 GOSUB 7400:REM enable timers
1450 :
1460 WHILE boatflag=0 AND fallflag=0 AND
 endflag=0
1470 IF joyflag=1 THEN GOSUB 3900 ELSE G
OSUB 8400
1480 WEND
1490 :
1500 IF fallflag=1 THEN GOSUB 3000:GOTO
1730:REM end loop
1510 IF endflag=1 THEN 1730:REM end loop

1520 GOSUB 5100:REM down gangway
1530 :
1540 REM **** steer boat ****
1550 :
1560 WHILE fetchflag=0 AND endflag=0
1570 IF joyflag=1 THEN GOSUB 5500 ELSE G
OSUB 8500
1580 WEND
1590 IF endflag=1 THEN 1730:REM end loop

1600 :
1610 GOSUB 5300:REM back up gangway
1620 :
1630 REM **** back up ladders etc ****
```

```
1640 WHILE (charx<>1 OR chary<>3) AND fa
llflag=0 AND endflag=0
1650 IF joyflag=1 THEN GOSUB 3900 ELSE G
OSUB 8400
1660 WEND
1670 IF endflag=1 THEN 1730:REM end loop

1680 :
1690 IF fallflag=1 THEN GOSUB 3000:GOTO
1730:REM end loop
1700 IF carryflag=1 THEN GOSUB 4600:REM
one home
1710 GOSUB 2900:GOSUB 3000:REM reset
1720 DI:GOSUB 3200:EI:REM no more pillar
 men?
1730 WEND
1740 GOSUB 8100:REM end routine
1750 WEND:END
```

A proper title screen

Our program should now be in full working order, but, as a finishing touch, this routine uses some of the graphics techniques covered in the book to produce an attractive title screen. Delete lines 1250—1260 and type in these lines to add the new screen display:

```
1250 GOSUB 8600:REM title screen
8600 REM **** title board ****
8610 PAPER 0:CLS
8620 FOR x=0 TO 8 STEP 4
8630 MOVE 34,256:MOVER x,x:GOSUB 8710:RE
M plot STRANDED
8640 NEXT x
8650 GOSUB 9300:REM display choices
8660 RETURN
8670 :
8700 REM **** draw STRANDED ****
8710 REM ** letter s **
8720 DRAWR 0,-16,3
8730 DRAWR 56,0:DRAWR 0,40:DRAWR -48,16
8740 DRAWR 0,16:DRAWR 40,0:DRAWR 0,-8
8750 DRAWR 8,0:DRAWR 0,16:DRAWR -56,0
```

```
8760 DRAWR 0,-32:DRAWR 48,-16:DRAWR 0,-2
4
8770 DRAWR -40,0:DRAWR 0,8:DRAWR -8,0
8780 REM ** letter t **
8790 MOVER 96,-16:DRAWR 0,72
8800 DRAWR -16,0:DRAWR 0,-8:DRAWR -8,0
8810 DRAWR 0,16:DRAWR 56,0:DRAWR 0,-16
8820 DRAWR -8,0:DRAWR 0,8:DRAWR -16,0
8830 DRAWR 0,-72:DRAWR -8,0
8840 REM ** letter r **
8850 MOVER 48,0:DRAWR 0,80
8860 DRAWR 56,0:DRAWR 0,-32:DRAWR -40,-1
6
8870 DRAWR 40,-16:DRAWR 0,-16:DRAWR -8,0
8880 DRAWR 0,8:DRAWR -40,16:DRAWR 0,-24
8890 DRAWR -8,0
8900 MOVER 8,40:DRAWR 0,32:DRAWR 40,0
8910 DRAWR 0,-16:DRAWR -40,-16
8920 REM ** letter a **
8930 MOVER 64,-40:DRAWR 0,52:DRAWR 28,28

8940 DRAWR 28,-28:DRAWR 0,-52:DRAWR -8,0
8950 DRAWR 0,32:DRAWR -40,0:DRAWR 0,-32
8960 DRAWR -8,0
8970 MOVER 8,40:DRAWR 0,8:DRAWR 20,20
8980 DRAWR 20,-20:DRAWR 0,-8:DRAWR -40,0

8990 REM ** letter n **
9000 MOVER 64,-40:DRAWR 0,80:DRAWR 16,0
9010 DRAWR 32,-72:DRAWR 0,72:DRAWR 8,0
9020 DRAWR 0,-80:DRAWR -16,0:DRAWR -32,7
2
9030 DRAWR 0,-72:DRAWR -8,0
9040 REM **** letter d ****
9050 MOVER 72,0:DRAWR 0,80:DRAWR 40,0
9060 DRAWR 16,-16:DRAWR 0,-48:DRAWR -16,
-16
9070 DRAWR -40,0
9080 MOVER 8,8:DRAWR 0,64:DRAWR 28,0
9090 DRAWR 12,-12:DRAWR 0,-40:DRAWR -12,
-12
9100 DRAWR -28,0
9110 REM ** letter e **
9120 MOVER 64,-8:DRAWR 0,80:DRAWR 56,0
```

```
9130 DRAWR 0,-16:DRAWR -8,0:DRAWR 0,8
9140 DRAWR -40,0:DRAWR 0,-32:DRAWR 24,0
9150 DRAWR 0,8:DRAWR 8,0:DRAWR 0,-24
9160 DRAWR -8,0:DRAWR 0,8:DRAWR -24,0:DR
AWR 0,-24
9170 DRAWR 40,0:DRAWR 0,8:DRAWR 8,0
9180 DRAWR 0,-16:DRAWR -56,0
9190 REM  ** letter d **
9200 MOVER 72,0:DRAWR 0,80:DRAWR 40,0
9210 DRAWR 16,-16:DRAWR 0,-48:DRAWR -16,
-16
9220 DRAWR -40,0
9230 MOVER 8,8:DRAWR 0,64:DRAWR 28,0
9240 DRAWR 12,-12:DRAWR 0,-40:DRAWR -12,
-12
9250 DRAWR -28,0
9260 RETURN
9270 :
9300 REM  **** diplay choices ****
9310 mess$=CHR$(164)+" 1985 by S.W. Colw
ill"+SPACE$(8)
9320 mess$=mess$+mess$
9330 PEN 7:LOCATE 8,17:PRINT"PRESS"
9340 LOCATE 7,20:PRINT"CONTROL"
9350 PEN 12:LOCATE 3,18:PRINT"j for joys
tick"
9360 LOCATE 4,19:PRINT"c for cursor"
9370 ans$="":c=1
9380 WHILE ans$<>"j" AND ans$<>"c"
9390 INK 3,c:REM change STRANDED ink
9400 SOUND 1,INT(RND(1)*1000),5,5
9410 LOCATE 1,13:PRINT MID$(mess$,c,20)
9420 c=c+1:IF c>30 THEN c=1
9430 FOR i=1 TO 50:NEXT:REM delay
9440 ans$=INKEY$
9450 WEND
9460 INK 3,6:REM reset ink colour
9470 IF ans$="j" THEN joyflag=1 ELSE joy
flag=0
9480 RETURN
```

The routine at line 8700 draws STRANDED in high-resolution graphics, using relative plotting. This routine is called three times, with slightly different starting points, to produce a three-dimensional effect. The 'display choices' routine at line `9300` repeatedly changes the ink that was used to create the STRANDED lettering whilst scrolling a copyright message across the screen and looking for the player to press **j** or **c** to select joystick or cursor control.

Program structure

'Stranded' is now completed and we can see the final structure of the program. Comparing this diagram with the one given at the end of Chapter 7, the addition of the final `WHILE...WEND` loop to the program changes the structure considerably. We can see that the only routines that lie outside this loop are the initialisation routine and the title-screen routine. As this loop allows the player to opt for a further game, all other routines lie within it. So within the main loop we have routines to set up the screen and foreground displays, reset the various flags and variables, an inner `WHILE...WEND` loop to control the game action and an end-of-game routine. The action routines have been discussed in earlier chapters but a flag reset routine and a routine to check on the status of the pillar men are added to the end of the action loop.

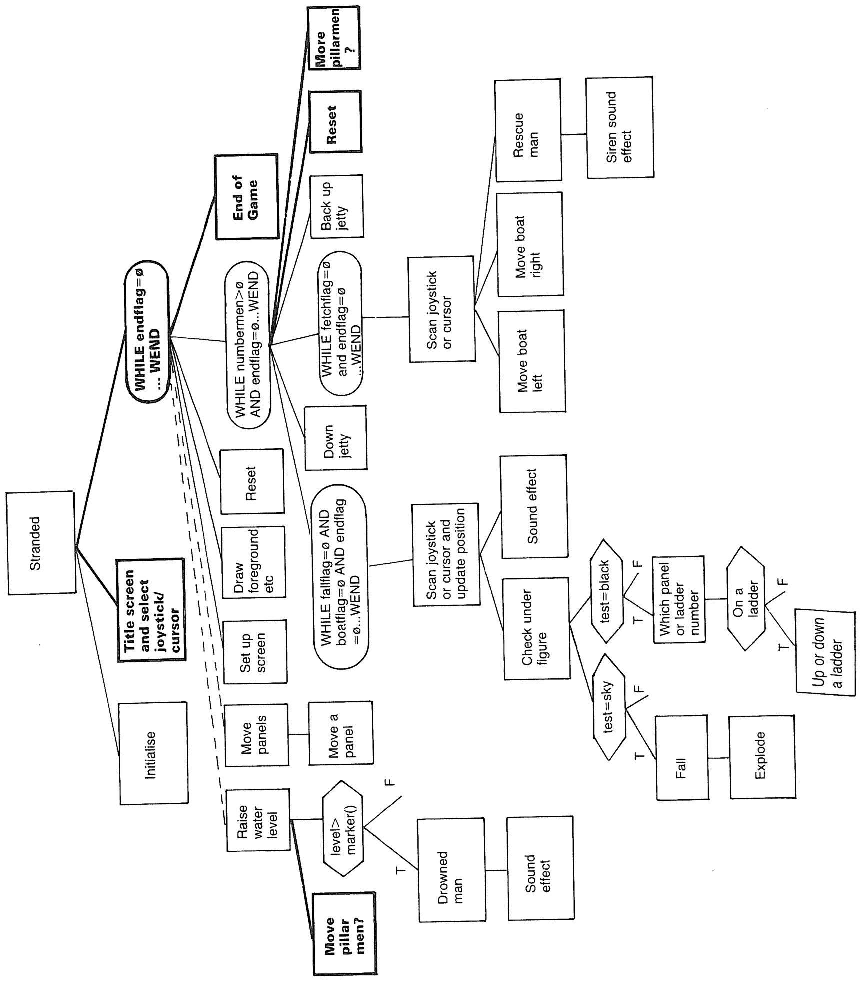

Stranded
Initialise
Title screen and select joystick/ cursor
WHILE endflag=ø ... WEND
End of Game
Reset
Draw foreground etc
Set up screen
Move panels
Move a panel
Raise water level
level> marker()
T
F
Drowned man
Sound effect
Move pillar men?
WHILE numbermen>ø AND endflag=ø...WEND
Reset
More pillarmen ?
Back up jetty
WHILE fetchflag=ø and endflag=ø ...WEND
Scan joystick or cursor
Move boat left
Move boat right
Rescue man
Siren sound effect
Down jetty
WHILE fallflag=ø AND boatflag=ø AND endflag =ø...WEND
Scan joystick or cursor and update position
Sound effect
Check under figure
test=black
T
F
Which panel or ladder number
On a ladder
T
F
Up or down a ladder
test=sky
T
F
Fall
Explode

Figure 8.1

Afterword

In this book we have looked at many of the techniques needed to write graphics games in BASIC on the Amstrad CPC range. The approach throughout has been to introduce the Amstrad CPC range's facilities by way of practical demonstration and, if you have worked your way steadily through the sections at the end of each chapter, you should have a completed version of the game 'Stranded' to play on your Amstrad CPC range. You will also find a complete listing of the game in Appendix A. Little has been said of non-graphics-based games such as adventures or strategy games. These areas would take a further book to cover. However, it is hoped that the programming techniques covered in this book will be useful in many other areas of programming. Amstrad offer an excellent dialect of BASIC which, combined with well thought out firmware and hardware, makes their computers extremely good value for money and easy to use.

Appendix A

The 'Stranded' program

```
1000 REM ***************************
1010 REM ***************************
1020 REM **                       **
1030 REM **       Stranded!       **
1040 REM **                       **
1050 REM ** (c)1985 S.W. Colwill **
1060 REM **                       **
1070 REM ***************************
1080 REM ***************************
1090 :
1100 REM ********************
1110 REM * dimension arrays *
1120 REM ********************
1130 :
1140 REM ** dimension panel arrays **
1150 np=6:REM set number of panels
1160 DIM scx(np),scy(np),gcp(np),pcl(np)

1170 DIM pl(np),gp(np),stx(np),sty(np)
1180 DIM ex(np),dx(np),c(np),mc(np)
1190 REM ** dimension ladder arrays **
1200 DIM lx(3),ly(3),ll(3),glx(3),gly(3)
,gll(3),deck(3)
1210 REM ** dimension pillar arrays **
1220 DIM pillcx(7),pillch(7),pillarx(7),
pilarh(7)
1230 DIM drownflag(7),safeflag(7),maxlev
(7)
1240 GOSUB 1800:REM initialise
1250 GOSUB 8600:REM title screen
1260 :
1270 REM *******************
1280 REM * set up routines *
1290 REM *******************
1300 WHILE endflag=0:REM game replay loop
```

```
1310 numbermen=4:menhome=0
1320 GOSUB 2600:REM set up screen
1330 GOSUB 2800:REM foreground etc
1340 GOSUB 2900:GOSUB 3000:REM reset
1350 :
1360 REM ********************
1370 REM * main action loop *
1380 REM ********************
1390 :
1400 WHILE numbermen>0 AND endflag=0
1410 LOCATE charx,chary:PRINT standman$
1420 :
1430 REM **** negotiate ladders etc ****
1440 GOSUB 7400:REM enable timers
1450 :
1460 WHILE boatflag=0 AND fallflag=0 AND
 endflag=0
1470 IF joyflag=1 THEN GOSUB 3900 ELSE G
OSUB 8400
1480 WEND
1490 :
1500 IF fallflag=1 THEN GOSUB 3000:GOTO
1730:REM end loop
1510 IF endflag=1 THEN 1730:REM end loop

1520 GOSUB 5100:REM down gangway
1530 :
1540 REM **** steer boat ****
1550 :
1560 WHILE fetchflag=0 AND endflag=0
1570 IF joyflag=1 THEN GOSUB 5500 ELSE G
OSUB 8500
1580 WEND
1590 IF endflag=1 THEN 1730:REM end loop

1600 :
1610 GOSUB 5300:REM back up gangway
1620 :
1630 REM **** back up ladders etc ****
1640 WHILE (charx<>1 OR chary<>3) AND fa
llflag=0 AND endflag=0
1650 IF joyflag=1 THEN GOSUB 3900 ELSE G
OSUB 8400
1660 WEND
```

```
1670 IF endflag=1 THEN 1730:REM end loop

1680 :
1690 IF fallflag=1 THEN GOSUB 3000:GOTO
1730:REM end loop
1700 IF carryflag=1 THEN GOSUB 4600:REM
one home
1710 GOSUB 2900:GOSUB 3000:REM reset
1720 DI:GOSUB 3200:EI:REM no more pillar
 men?
1730 WEND
1740 GOSUB 8100:REM end routine
1750 WEND:END
1755 :
1760 REM ***************
1770 REM * subroutines *
1780 REM ***************
1790 :
1800 REM **** initialisation ****
1810 MODE 0
1820 REM ** set interval timer values **
1830 moveint=12:riseint=300
1840 REM ** define strings **
1850 deck$=STRING$(20,CHR$(207))
1860 bar$=CHR$(210)
1870 zero$=STRING$(5,"0")
1880 numbermen=4:men$=STRING$(7,CHR$(248
))
1890 :
1900 REM ** define windows **
1910 WINDOW 1,20,1,25
1920 WINDOW #2,1,20,18,25
1930 WINDOW #3,19,19,18,23
1940 :
1950 REM ** user defined chars **
1960 SYMBOL AFTER 240
1970 REM SYMBOL 255,14,8,24,30,16,16,16,
16
1980 SYMBOL 254,56,56,18,252,144,40,68,1
34
1990 SYMBOL 253,56,56,18,252,144,16,16,2
4
2000 SYMBOL 245,0,0,0,0,255,127,62,62
2010 SYMBOL 247,28,28,72,63,9,20,35,97
```

```
2020 SYMBOL 246,28,28,72,63,9,8,8,24
2030 transon$=CHR$(22)+CHR$(1)
2040 transoff$=CHR$(22)+CHR$(0)
2050 standman$=CHR$(248):boat$=CHR$(245)
2070 explode$=CHR$(238)
2080 :
2090 REM ** panel data **
2100 FOR n=1 TO 6
2110 READ pcl(n),gcp(n),scx(n),scy(n)
2120 pl(n)=32*pcl(n):gp(n)=32*gcp(n)-1
2130 stx(n)=32*(scx(n)-1):sty(n)=399-16*
scy(n)
2140 ex(n)=stx(n)+gp(n)
2150 mc(n)=gp(n)-pl(n)
2160 dx(n)=32
2170 NEXT n
2180 DATA 2,3,6,3,2,3,14,3
2190 DATA 1,2,5,8,2,3,13,8
2200 DATA 2,3,7,14,1,2,13,14
2210 :
2220 REM ** joystick directions **
2230 left=4:right=8:centre=0:fire=16
2240 REM ** ladder data **
2250 FOR i=1 TO 3
2260 READ lx(i),ly(i),ll(i)
2270 glx(i)=32*(lx(i)-1):gly(i)=399-16*(
ly(i)-1)-8
2280 gll(i)=(ll(i)+1)*16
2290 NEXT i
2300 :
2310 REM ** ladder data **
2320 DATA 18,9,5,2,15,6,19,18,3
2330 RETURN
2340 :
2400 REM **** set up pillars ****
2410 RANDOMIZE TIME
2420 cc=1:pillcy=23:pillary=405-16*pillc
y
2430 FOR n=1 TO 7
2440 pillcx(n)=cc:pillch(n)=INT(RND(1)*4
)+1
2450 pillarx(n)=32*pillcx(n)-16
2460 pillarh(n)=16*pillch(n)-8
2470 maxlev(n)=pillary+pillarh(n)+16
```

```
2480 cc=cc+2
2490 NEXT n
2500 level=pillary
2510 GOSUB 7500:REM draw walkway
2520 PAPER #3,mauve:CLS #3:PEN #3,white

2530 GOSUB 7000:REM draw pillars
2540 LOCATE #2,18,7:PEN #2,green:PRINT#2
,boat$
2550 RETURN
2560 :
2600 REM **** set up screen ****
2610 BORDER 0
2620 whiteink=26:blueink=1:redink=6
2630 sky=10:black=5:water=0:red=3:white=
4
2640 green=12:blue=6:pink=11:pastgreen=1
3:mauve=14
2645 deck(1)=red:deck(2)=green:deck(3)=b
lue
2650 INK 10,14:INK 3,6:INK 4,26:INK 5,0:
INK 14,5
2660 INK 12,18:INK 6,2:INK 11,16:INK 13,
22
2670 PAPER sky:CLS:PAPER #2,water:CLS #2
:PEN #2,white
2680 LOCATE 1,4:PEN deck(1)
2690 PRINT LEFT$(deck$,5);SPACE$(3);LEFT
$(deck$,5);
2700 PRINT SPACE$(3);LEFT$(deck$,4)
2710 LOCATE 1,9:PEN deck(2)
2720 PRINT LEFT$(deck$,4);SPACE$(2);LEFT
$(deck$,6);
2730 PRINT SPACE$(3);LEFT$(deck$,5)
2740 LOCATE 1,15:PEN deck(3)
2750 PRINT LEFT$(deck$,6);SPACE$(3);LEFT
$(deck$,3);
2760 PRINT SPACE$(2);LEFT$(deck$,6)
2770 RETURN
2780 :
2800 REM **** draw foreground etc ****
2820 GOSUB 2400:REM draw sea scene
2830 GOSUB 6400:REM draw ladders
2840 LOCATE 8,1:PRINT"Score:"
```

```
2850 incscore=0:GOSUB 7700:REM print sco
re
2860 RETURN
2870 :
2900 REM **** reset routine ****
2910 mancol=white:PEN mancol:charx=1:cha
ry=3
2920 REM ** print number men left **
2930 IF fallflag=1 THEN numbermen=number
men-1
2940 IF numbermen<1 THEN RETURN
2950 LOCATE 1,1:PRINT LEFT$(men$,numberm
en-1);SPACE$(4-numbermen)
2960 RETURN
2970 :
3000 REM **** reset flags ****
3010 boatflag=0:fallflag=0:resetflag=0
3020 fetchflag=0:carryflag=0
3030 RETURN
3040 :
3100 REM **** end of game flag reset ***
*
3110 FOR i=1 TO 7
3120 safeflag(i)=0:drownflag(i)=0
3130 NEXT i
3140 lad=0:panel=0:endflag=0
3150 RETURN
3160 :
3200 REM **** any more pillarmen ****
3210 savcount=0:drowncount=0
3220 FOR pilln=1 TO 7
3230 savcount=savcount+safeflag(pilln)
3240 drowncount=drowncount+drownflag(pil
ln)
3250 NEXT pilln
3260 IF drowncount>3 THEN endflag=1:RETU
RN:REM end game
3270 IF savcount=7 THEN incscore=5000:GO
SUB 7700:REM bonus
3280 mancount=savcount+drowncount
3290 IF mancount<7 OR mancol=red THEN RE
TURN:REM same level
3300 :
3310 REM **** next level ****
```

```
3320 GOSUB 7300:REM disable timers
3330 REM ** reset flags **
3340 FOR pilln=1 TO 7
3350 safeflag(pilln)=0:drownflag(pilln)=
0
3360 NEXT pilln
3370 REM ** reset pillars **
3380 REM ** alter interrupts **
3390 CLS #2:GOSUB 2400:REM redraw sea sc
ene
3400 menhome=0:LOCATE 14,2:PRINT SPACE$(
7)
3410 moveint=moveint-2
3420 riseint=riseint-50
3430 IF moveint<4 THEN moveint=12:risein
t=300:REM reset to initial values
3440 GOSUB 7400:REM enable timers
3450 RETURN
3460 :
3500 REM **** moving panels ****
3510 REM ** disable and re-enable timers
 **
3520 IF resetflag=1 THEN GOSUB 7300:GOSU
B 7400
3530 n=n+1:IF n>6 THEN n=1
3540 GOSUB 3600
3550 RETURN
3560 :
3600 REM **** move panel n ****
3610 DI
3620 IF panel=n THEN GOSUB 3800
3630 MOVE stx(n),sty(n)
3640 c(n)=c(n)+dx(n)
3650 IF c(n)>=mc(n) THEN dx(n)=-dx(n):c(
n)=mc(n)
3660 IF c(n)<=0 THEN dx(n)=-dx(n):c(n)=0
3670 DRAWR c(n),0,sky
3680 DRAWR pl(n),0,black
3690 DRAW ex(n),sty(n),sky
3700 EI
3710 RETURN
3720 :
3800 REM **** move man on panel ****
3810 LOCATE charx,chary:PRINT" "
```

```
3820 charx=charx+dx(n)/32:IF charx<1 THE
N charx=1
3830 LOCATE charx,chary:PRINT standman$
3840 RETURN
3850 :
3900 REM **** scan joystick ****
3910 IF JOY(0)=left THEN DI:GOSUB 4000:G
OSUB 4300:EI
3920 IF JOY(0)=right THEN DI:GOSUB 4100:
GOSUB 4300:EI
3930 IF JOY(0)=centre AND ladflag=0 THEN
 DI:GOSUB 4200:GOSUB 4300:EI
3940 RETURN
3950 :
4000 REM **** move left ****
4010 chartog=1-chartog:REM character typ
e
4020 LOCATE charx,chary:PRINT" "
4030 charx=charx-1:IF charx<1 THEN charx
=1
4040 LOCATE charx,chary:PRINT CHR$(246+c
hartog)
4050 IF ladflag=1 THEN ladflag=0:GOSUB 6
500
4060 GOSUB 7800:REM sound fx
4070 RETURN
4080 :
4100 REM **** move right ****
4110 chartog=1-chartog:REM character typ
e
4120 LOCATE charx,chary:PRINT" "
4130 charx=charx+1:IF charx>20 THEN char
x=20
4140 LOCATE charx,chary:PRINT CHR$(253+c
hartog)
4150 IF ladflag=1 THEN ladflag=0:GOSUB 6
500
4160 GOSUB 7800:REM sound fx
4170 RETURN
4180 :
4200 REM **** standing still ****
4210 LOCATE charx,chary:PRINT standman$
4220 RETURN
4230 :
```

```
4300 REM **** check under figure ****
4310 :
4320 GOSUB 6700:REM convert coords
4330 t=TEST(graphx,graphy)
4340 IF t=sky THEN fallflag=1:GOSUB 4400
:RETURN
4350 IF t=black THEN GOSUB 4700:RETURN E
LSE panel=0
4360 RETURN
4370 :
4400 REM **** fall ****
4410 chary=chary+1:GOSUB 6700:REM conver
t
4420 DI
4430 WHILE TEST(graphx,graphy)=sky
4440 LOCATE charx,chary-1:PRINT" "
4450 chary=chary+1:GOSUB 6700:REM conver
t
4460 LOCATE charx,chary:PRINT standman$
4470 GOSUB 7800:REM sound fx
4480 WEND
4490 LOCATE charx,chary-1:PRINT" "
4500 GOSUB 6800:REM explode
4510 GOSUB 2900:REM rest coords etc
4520 EI
4530 RETURN
4540 :
4600 REM **** got one home ****
4610 incscore=2000:GOSUB 7700:REM score
4620 menhome=menhome+1:GOSUB 7900:REM so
und fx
4630 PEN red:LOCATE 14,2:PRINT LEFT$(men
$,menhome)
4640 RETURN
4650 :
4700 REM **** on a panel/ladder ? ****
4710 IF panel<>0 THEN RETURN
4720 FOR i=1 TO np
4730 IF graphy=sty(i) AND graphx>=stx(i)
 AND graphx<=ex(i) THEN panel=i:i=np
4740 NEXT i
4750 IF panel<>0 THEN incscore=50:GOSUB
7700
4760 REM ** test for ladder **
```

```
4770 IF ladflag=1 THEN RETURN
4780 lad=0
4790 FOR i=1 TO 3
4800 IF charx=lx(i) THEN lad=i:GOSUB 490
0:i=3
4810 NEXT i
4820 RETURN
4830 :
4900 REM **** up or down a ladder ****
4910 IF chary=ly(lad)-1 THEN dy=-1
4920 IF chary=ly(lad)-ll(lad)-1 THEN dy=
1
4930 FOR y=1 TO ll(lad)
4940 ladchar=1-ladchar
4950 LOCATE charx,chary:PRINT" "
4960 GOSUB 6500:REM redraw ladder
4970 chary=chary+dy
4980 PRINT transon$
4990 LOCATE charx,chary:PRINT CHR$(250+l
adchar)
5000 PRINT transoff$
5010 FOR delay=1 TO 100:NEXT
5020 GOSUB 7800:REM sound fx
5030 NEXT y
5040 ladflag=1:incscore=100:GOSUB 7700:R
EM inc score
5050 IF lad=3 AND boatflag=0 THEN boatfl
ag=1
5060 resetflag=1:REM reset timers
5070 RETURN
5080 :
5100 REM **** down gangway ****
5110 LOCATE charx,chary:PRINT" "
5120 GOSUB 6500:REM redraw ladder
5130 charx=1:PEN #3,mancol:REM change to
 stream 3
5140 FOR chary=2 TO 5
5150 tog=1-tog
5160 LOCATE #3,charx,chary:PRINT#3, CHR$
(250+tog)
5170 FOR i=1 TO 100:NEXT:REM delay
5180 LOCATE #3,charx,chary:PRINT#3," "
5190 GOSUB 7800:REM sound fx
5200 NEXT chary
```

```
5210 REM **** onto boat ****
5220 charx=18:chary=7:REM change to stre
am 2
5230 manink=whiteink:GOSUB 6000:REM prin
t man/boat
5240 RETURN
5250 :
5300 REM **** back up gangway ****
5310 LOCATE #2,charx,chary:PEN #2,green:
PRINT#2,boat$
5320 charx=1:PEN #3,mancol:REM change to
 stream 3
5330 FOR chary=5 TO 2 STEP -1
5340 tog=1-tog
5350 LOCATE #3,charx,chary:PRINT#3, CHR$
(250+tog)
5360 FOR i=1 TO 100:NEXT:REM delay
5370 LOCATE #3,charx,chary:PRINT#3," "
5380 GOSUB 7800:REM sound fx
5390 NEXT chary
5400 charx=19:chary=17:REM change to str
em 1
5410 PEN mancol:LOCATE charx,chary:PRINT
 standman$
5420 lad=3:DI:GOSUB 4900:EI:REM up ladde
r
5430 RETURN
5440 :
5500 REM **** scan boat joystick ****
5510 DI
5520 IF JOY(0)=left THEN GOSUB 5600
5530 IF JOY(0)=right THEN GOSUB 5700
5540 IF JOY(0)=fire THEN GOSUB 5800
5550 EI
5560 RETURN
5570 :
5600 REM **** move boat left ****
5610 LOCATE #2,charx,chary:PRINT#2," "
5620 charx=charx-1:IF charx<1 THEN charx
=1
5630 GOSUB 6000:REM print man/boat
5640 GOSUB 7800:REM sound fx
5650 RETURN
5660 :
```

```
5700 REM **** move boat right ****
5710 LOCATE #2,charx,chary:PRINT#2," "
5720 charx=charx+1:IF charx>18 THEN char
x=18
5730 GOSUB 6000:REM print man/boat
5740 GOSUB 7800:REM sound fx
5750 IF charx=18 THEN fetchflag=1
5760 RETURN
5770 :
5800 REM **** rescue man ****
5810 pilln=charx/2
5820 IF pilln<>INT(pilln) OR charx>14 TH
EN RETURN
5830 IF carryflag=1 THEN RETURN
5840 IF safeflag(pilln)=1 THEN RETURN
5850 carryflag=1:safeflag(pilln)=1
5860 remht=5+pillarh(pilln)-level+pillar
y
5870 LOCATE #2,pillcx(pilln)+1,pillcy-pi
llch(pilln)-17
5880 IF remht<0 THEN PRINT#2," " ELSE PE
N #2,pastgreen:PRINT#2,bar$
5890 manink=redink:GOSUB 6000
5900 mancol=red
5910 incscore=1000-10*remht:GOSUB 7700:R
EM inc score
5920 GOSUB 7900:REM sound fx
5930 RETURN
5940 :
6000 REM **** print man and boat ****
6010 PRINT#2,transon$
6020 INK 15,blueink
6030 LOCATE #2,charx,chary:PEN#2,15:PRIN
T#2,standman$
6040 LOCATE #2,charx,chary:PEN#2,green:P
RINT#2,boat$
6050 INK 15,manink
6060 PRINT#2,transoff$;
6070 RETURN
6080 :
6100 REM **** raise water level ****
6110 DI
6120 FOR pn= 1 TO 7
6130 MOVE pillarx(pn),level
```

```
6140 DRAWR 48,0,water
6150 FOR px=1 TO 5
6160 PLOTR 2,1
6170 NEXT px
6180 IF level>maxlev(pn) AND drownflag(p
n)=0 AND safeflag(pn)=0 THEN GOSUB 6300
6190 NEXT pn
6200 level=level+2
6210 ENV 1,5,3,3,1,0,30,5,-2,10,4,-1,60,
1,0,20
6220 SOUND 2,0,0,0,1,0,15
6230 GOSUB 3200:REM more pillarman ?
6240 EI
6250 RETURN
6260 :
6300 REM **** drowned man ****
6310 IF safeflag(pn)=1 THEN RETURN
6320 drownflag(pn)=1
6330 GOSUB 8000:REM sound fx
6340 RETURN
6350 :
6400 REM **** draw ladders ****
6410 FOR lad=1 TO 3:GOSUB 6500:NEXT lad
6420 RETURN
6430 :
6500 REM **** draw a ladder ****
6505 PEN deck(lad):LOCATE lx(lad),ly(lad
)-ll(lad):PRINT LEFT$(deck$,1)
6507 PEN mancol
6510 MOVE glx(lad),gly(lad)
6520 DRAWR 0,gll(lad),black
6530 MOVER 28,0
6540 DRAWR 0,-gll(lad)
6550 FOR ly=8 TO gll(lad)-8 STEP 8
6560 MOVER -28,8
6570 DRAWR 28,0
6580 NEXT ly
6590 RETURN
6600 :
6700 REM **** convert char/graph ****
6710 graphx=32*charx-16
6720 graphy=399-16*chary
6730 RETURN
6740 :
```

```
6800 REM **** explode ****
6810 DI
6820 FOR i=1 TO 5
6830 FOR j=15 TO 0 STEP -1
6840 LOCATE charx,chary
6850 PEN j:PRINT explode$
6860 SOUND 4,0,2,5,0,0,j
6870 NEXT j,i
6880 LOCATE charx,chary:PRINT" "
6890 DI
6900 RETURN
6910 :
7000 REM **** draw pillars ****
7010 PEN #2,white
7020 FOR n=1 TO 7
7030 pillcol=red
7040 FOR i=0 TO pillarh(n)/2
7050 MOVE pillarx(n),pillary+2*i
7060 DRAWR 48,0,pillcol
7070 IF pillcol=red THEN pillcol=white E
LSE pillcol=red
7080 SOUND 4,10*i,2,5
7090 NEXT i
7100 FOR j=1 TO 5
7110 MOVE pillarx(n)+48+2*j,pillary+j

7120 DRAWR 0,pillarh(n),pink
7130 DRAWR -48,0,pastgreen
7140 NEXT j
7150 PRINT#2,transon$
7160 LOCATE #2,pillcx(n)+1,pillcy-pillch
(n)-17
7170 PRINT#2,standman$
7180 PRINT#2,transoff$
7190 NEXT n
7200 RETURN
7210 :
7300 REM **** disable timers ****
7310 resetflag=0
7320 FOR i=0 TO 2:r=REMAIN(i):NEXT i
7330 RETURN
7340 :
7400 REM **** enable timers ****
7410 EVERY moveint,0 GOSUB 3500:REM move
```

```
 panels
7420 EVERY riseint,1 GOSUB 6100:REM rais
e water level
7430 RETURN
7440 :
7500 REM draw walkway ****
7510 FOR y= 118 TO 120 STEP 2
7520 MOVE 0,y:DRAW 639,y,black
7530 NEXT y
7540 FOR y=31 TO 27 STEP -2
7550 MOVE 576,y:DRAW 608,y,black
7560 NEXT y
7570 DRAW 608,118
7580 FOR y=122 TO 126 STEP 2
7590 MOVE 0,y:DRAW 639,y,mauve
7600 NEXT y
7610 RETURN
7620 :
7700 REM **** increase score ****
7710 PEN red
7720 score=score+incscore:score$=STR$(sc
ore)
7730 lgth=LEN(score$):score$=RIGHT$(scor
e$,lgth-1)
7740 lgth=LEN(score$)
7750 score$=LEFT$(zero$,6-lgth)+score$
7760 LOCATE 14,1:PRINT score$
7770 PEN mancol
7780 RETURN
7790 :
7800 REM **** move sound effect ****
7810 tone=100+20*chary
7820 SOUND 4,tone,5,5
7830 RETURN
7840 :
7900 REM **** siren sound effect ****
7910 DI:FOR i=1 TO 2
7920 FOR per=400 TO 250 STEP -5
7930 SOUND 1,per,2,5
7940 SOUND 2,per+100,2,5
7950 NEXT per,i
7960 SOUND 1,100,40,2
7970 EI:RETURN
7980 :
```

```
8000 REM **** drown sound efect ****
8010 DI:FOR per= 250 TO 400 STEP 5
8020 SOUND 1,per,3,5
8030 NEXT per
8040 EI:RETURN
8050 :
8100 REM **** end routine ****
8110 GOSUB 7300:REM disable timers
8120 INK 14,3,24
8130 CLS:PEN 14:REM flash
8140 LOCATE 6,8:PRINT"Game Over"
8150 LOCATE 6,9:PRINT"========="
8160 IF score$>hiscore$ THEN hiscore$=sc
ore$
8170 PEN black:LOCATE 1,12:PRINT"Hi Scor
e:"
8180 PEN red:LOCATE 13,12:PRINT hiscore$

8190 PEN black:LOCATE 1,14:PRINT"Your Sc
ore:"
8200 PEN red:LOCATE 13,14:PRINT score$
8210 PEN green:LOCATE 2,18:PRINT"Another
 Game (y/n)"
8211 FOR p=600 TO 100 STEP -10
8212 SOUND 1,p,2,5
8213 NEXT p
8220 ans$=""
8230 WHILE ans$<>"y" AND ans$<>"n"
8240 ans$=INKEY$
8250 WEND
8260 score=0:score$="":REM zero score
8270 moveint=12:riseint=350:REM reset in
t timers
8280 GOSUB 3100:REM reset flags
8290 IF ans$="n" THEN endflag=1:CLS
8300 RETURN
8310 :
8400 REM **** cursor control on ladders
etc ****
8410 IF INKEY(8)=0 THEN DI:GOSUB 4000:GO
SUB 4300:EI
8420 IF INKEY(1)=0 THEN DI:GOSUB 4100:GO
SUB 4300:EI
8430 IF INKEY(1)<>0 AND INKEY(8)<>0 AND
```

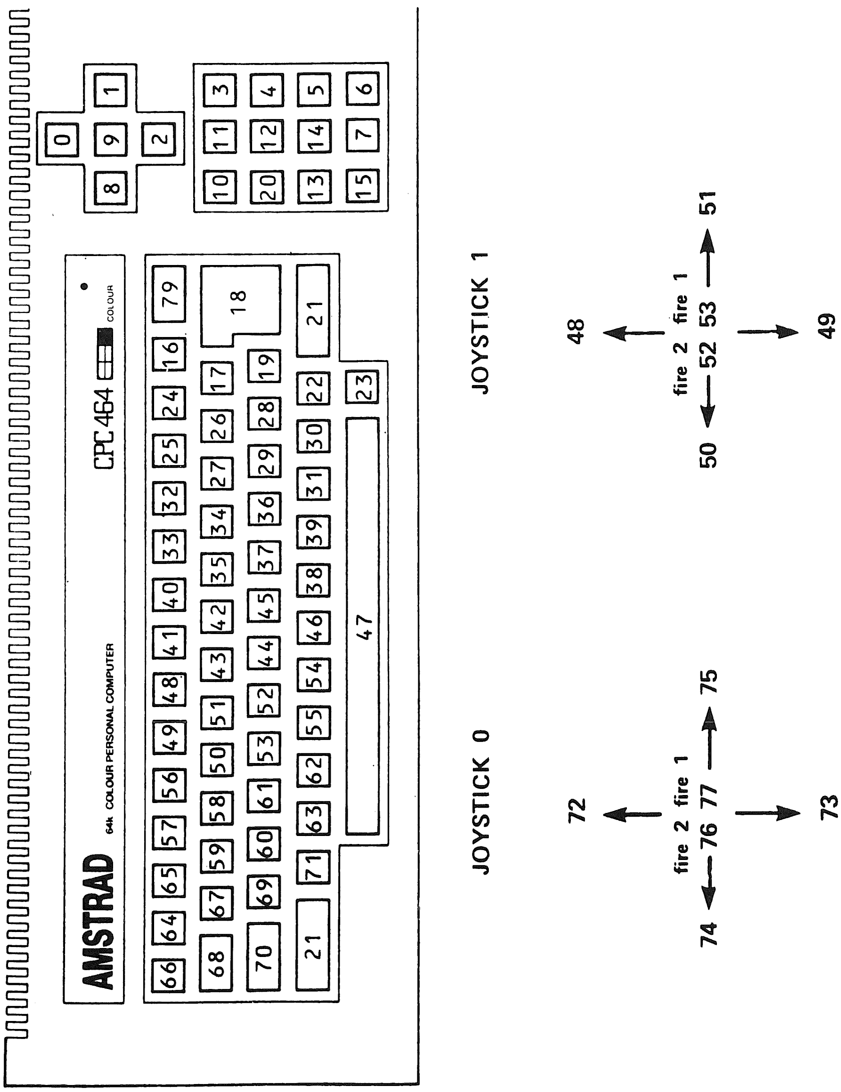
```
ladflag=0 THEN DI:GOSUB 4200:GOSUB 4300:
EI
8440 RETURN
8450 :
8500 REM **** steer by cursor ****
8510 DI
8520 IF INKEY(8)=0 THEN GOSUB 5600
8530 IF INKEY(1)=0 THEN GOSUB 5700
8540 IF INKEY(47)=0 THEN GOSUB 5800
8550 EI
8560 RETURN
8600 REM **** title board ****
8610 PAPER 0:CLS
8620 FOR x=0 TO 8 STEP 4
8630 MOVE 34,256:MOVER x,x:GOSUB 8710:RE
M plot STRANDED
8640 NEXT x
8650 GOSUB 9300:REM display choices
8660 RETURN
8670 :
8700 REM **** draw STRANDED ****
8710 REM ** letter s **
8720 DRAWR 0,-16,3
8730 DRAWR 56,0:DRAWR 0,40:DRAWR -48,16
8740 DRAWR 0,16:DRAWR 40,0:DRAWR 0,-8
8750 DRAWR 8,0:DRAWR 0,16:DRAWR -56,0
8760 DRAWR 0,-32:DRAWR 48,-16:DRAWR 0,-2
4
8770 DRAWR -40,0:DRAWR 0,8:DRAWR -8,0
8780 REM ** letter t **
8790 MOVER 96,-16:DRAWR 0,72
8800 DRAWR -16,0:DRAWR 0,-8:DRAWR -8,0
8810 DRAWR 0,16:DRAWR 56,0:DRAWR 0,-16
8820 DRAWR -8,0:DRAWR 0,8:DRAWR -16,0
8830 DRAWR 0,-72:DRAWR -8,0
8840 REM ** letter r **
8850 MOVER 48,0:DRAWR 0,80
8860 DRAWR 56,0:DRAWR 0,-32:DRAWR -40,-1
6
8870 DRAWR 40,-16:DRAWR 0,-16:DRAWR -8,0
8880 DRAWR 0,8:DRAWR -40,16:DRAWR 0,-24
8890 DRAWR -8,0
8900 MOVER 8,40:DRAWR 0,32:DRAWR 40,0
8910 DRAWR 0,-16:DRAWR -40,-16
```

```
8920 REM ** letter a **
8930 MOVER 64,-40:DRAWR 0,52:DRAWR 28,28

8940 DRAWR 28,-28:DRAWR 0,-52:DRAWR -8,0
8950 DRAWR 0,32:DRAWR -40,0:DRAWR 0,-32
8960 DRAWR -8,0
8970 MOVER 8,40:DRAWR 0,8:DRAWR 20,20
8980 DRAWR 20,-20:DRAWR 0,-8:DRAWR -40,0

8990 REM ** letter n **
9000 MOVER 64,-40:DRAWR 0,80:DRAWR 16,0
9010 DRAWR 32,-72:DRAWR 0,72:DRAWR 8,0
9020 DRAWR 0,-80:DRAWR -16,0:DRAWR -32,7
2
9030 DRAWR 0,-72:DRAWR -8,0
9040 REM **** letter d ****
9050 MOVER 72,0:DRAWR 0,80:DRAWR 40,0
9060 DRAWR 16,-16:DRAWR 0,-48:DRAWR -16,
-16
9070 DRAWR -40,0
9080 MOVER 8,8:DRAWR 0,64:DRAWR 28,0
9090 DRAWR 12,-12:DRAWR 0,-40:DRAWR -12,
-12
9100 DRAWR -28,0
9110 REM ** letter e **
9120 MOVER 64,-8:DRAWR 0,80:DRAWR 56,0
9130 DRAWR 0,-16:DRAWR -8,0:DRAWR 0,8
9140 DRAWR -40,0:DRAWR 0,-32:DRAWR 24,0
9150 DRAWR 0,8:DRAWR 8,0:DRAWR 0,-24
9160 DRAWR -8,0:DRAWR 0,8:DRAWR -24,0:DR
AWR 0,-24
9170 DRAWR 40,0:DRAWR 0,8:DRAWR 8,0
9180 DRAWR 0,-16:DRAWR -56,0
9190 REM ** letter d **
9200 MOVER 72,0:DRAWR 0,80:DRAWR 40,0
9210 DRAWR 16,-16:DRAWR 0,-48:DRAWR -16,
-16
9220 DRAWR -40,0
9230 MOVER 8,8:DRAWR 0,64:DRAWR 28,0
9240 DRAWR 12,-12:DRAWR 0,-40:DRAWR -12,
-12
9250 DRAWR -28,0
9260 RETURN
9270 :
```

```
9300 REM **** diplay choices ****
9310 mess$=CHR$(164)+" 1985 by S.W. Colw
ill"+SPACE$(8)
9320 mess$=mess$+mess$
9330 PEN 7:LOCATE 8,17:PRINT"PRESS"
9340 LOCATE 7,20:PRINT"CONTROL"
9350 PEN 12:LOCATE 3,18:PRINT"j for joys
tick"
9360 LOCATE 4,19:PRINT"c for cursor"
9370 ans$="":c=1
9380 WHILE ans$<>"j" AND ans$<>"c"
9390 INK 3,c:REM change STRANDED ink
9400 SOUND 1,INT(RND(1)*1000),5,5
9410 LOCATE 1,13:PRINT MID$(mess$,c,20)
9420 c=c+1:IF c>30 THEN c=1
9430 FOR i=1 TO 50:NEXT:REM delay
9440 ans$=INKEY$
9450 WEND
9460 INK 3,6:REM reset ink colour
9470 IF ans$="j" THEN joyflag=1 ELSE joy
flag=0
9480 RETURN
```

INKEY key numbers

"Reproduced with kind permission of Amsoft"



Appendix C

BASIC error messages

"Reproduced with kind permission of Amsoft"

Error numbers and error messages

When BASIC encounters a program statement, word or variable that it cannot understand or process, it will stop and display an error message. The form of the message will generally indicate what went wrong – and sometimes, if the error is a typographical error during program entry, BASIC will prompt in edit mode, with the line where the incorrect entry was made.

The most popular error to greet the inaccurate typist is the `Syntax Error` (number 2), and BASIC prompts with the line to edit if encountered in program mode. In direct mode, it simply states that an error occured, and assumes that the last line typed is visible to spot the problem.

If the `ON ERROR GOTO` command is included at the beginning of a program, it may refer the computer to a given line number when detecting an error. In the following example, the computer is referred to line 1000 when detecting an error:

```
10 ON ERROR GOTO 1000
```

program

```
1000 PRINT CHR$(7):MODE 2:INK 1,0: INK 0,9:
CLS :LIST
```

Whereupon the CPC464 will beep, clear the current screen, change to a suitable colour combination for the 80 column display, and list the program ready for examination. If the error is a `Syntax error` it will appear at the foot of the listing, awaiting correction in the line edit mode, although the `Syntax error` message is suppressed.

Remember to `END` the program on the last line before 1000 if you wish to save the results on the screen.

BASIC will not produce error messages for valid input – so it must be assumed that whenever an error does occur, it can be traced back to an error in the form of the program, usually guided by the message produced to help in the process of de-bugging. As with most things, you will learn most readily from your mistakes, so make the most of the fact that the CPC464 is the most tolerant of tutors: you will tire of trying long before the CPC464 loses its patience!

All errors generated by BASIC are listed here, in error number order. The messages produced by BASIC are given, as well as a brief description of possible causes.

1 Unexpected NEXT

A NEXT command has been encountered while not in a FOR loop, or the control variable in the NEXT command does not match that in the FOR.

2 Syntax Error

BASIC cannot understand the given line because a construct within it is not legal.

3 Unexpected **RETURN**

A **RETURN** command has been encountered when not in a subroutine.

4 DATA exhausted

A READ command has attempted to read beyond the end of the last DATA.

5 Improper argument

This is a general purpose error. The value of a function's argument, or a command parameter is invalid in some way.

6 Overflow

The result of an arithmetic operation has overflowed. This may be a floating point overflow, in which case some operation has yielded a value greater than 1.7E–38 (approx.). Alternatively, this may be the result of a failed attempt to change a floating point number to a 16 bit signed integer.

7 `Memory full`

The current program or its variables may be simply too big, or the control structure is too deeply nested (nested `GOSUB`s, `WHILE`s or `FOR`s).

A `MEMORY` command will give this error if an attempt is made to set the top of BASIC's memory too low, or to an impossibly high value. Note that an open cassette file has a buffer allocated to it, and that may restrict the values that `MEMORY` may use.

8 `Line does not exist`

The line referenced cannot be found.

9 `Subscript out of range`

One of the subscripts in an array reference is too big or too small.

1Ø `Array already dimensioned`

One of the arrays in a `DIM` statement has already been declared.

11 `Division by zero`

May occur in Real division, integer division, integer modulus or in exponentiation.

12 `Invalid direct command`

The last command attempted is not valid in Direct Mode.

13 `Type mismatch`

A numeric value has been presented where a string value is required, and vice versa, or an invalidly formed number has been found in `READ` or `INPUT`.

14 `String space full`

So many strings have been created that there is no further room available, even after 'garbage collection'.

15 `String too long`

String exceeds 255 characters in length. May be generated by adding a number of strings together.

16 String expression too complex

String expressions may generate a number of intermediate string values. When the number of these values exceeds a reasonable limit, BASIC gives up, and this error results.

17 Cannot CONTinue

For one reason or another the current program cannot be restarted using CONT. Note that CONT is intended for restarting after a STOP command, **[ESC][ESC]** or error, and that any alteration of the program in the meantime makes a restart impossible.

18 Unknown user function

No DEF FN has been executed for the FN just invoked.

19 RESUME missing

The end of the program has been encountered while in Error Processing Mode (ie in an ON ERROR GOTO routine).

20 Unexpected RESUME

RESUME is only valid while in Error Processing Mode (ie in an ON ERROR GOTO routine).

21 Direct command found

When loading a program from cassette a line without a line number has been found.

22 Operand missing

BASIC has encountered an incomplete expression.

23 Line too long

A line when converted to BASIC internal form becomes too big.

24 EOF met

An attempt has been made to read past end of file on the cassette input stream.

25 File type error

The cassette file being read is not of a suitable type. OPENIN is only prepared to open ASCII text files. LOAD, RUN etc, are only prepared to deal with the file types produced by `SAVE`.

`26 NEXT missing`

Cannot find a `NEXT` to match a `FOR` command.

`27 File already open`

An `OPENIN` or `OPENOUT` command has been executed before the previously opened file has been closed.

`28 Unknown command`

BASIC cannot find a taker for an external command.

`29 WEND missing`

Cannot find a `WEND` to match a `WHILE` command.

`3Ø Unexpected WEND`

Encountered a `WEND` when not in a `WHILE` loop, or a `WEND` that does not match the current `WHILE` loop.

BASIC Keywords

The following are the BASIC keywords, they are reserved and cannot be used as variable names.

`ABS, AFTER, AND, ASC, ATN, AUTO`

`BIN$, BORDER`

`CALL, CAT, CHAIN, CHR$, CINT, CLEAR, CLG, CLOSEIN, CLOSEOUT, CLS, CONT, COS, CREAL`

`DATA, DEF, DEFINT, DEFREAL, DEFSTR, DEG, DELETE, DI, DIM, DRAW, DRAWR`

`EDIT, EI, ELSE, END, ENT, ENV, EOF, ERASE, ERL, ERR, ERROR, EVERY, EXP`

`FIX, FN, FOR, FRE`

`GOSUB, GOTO`

`HEX$, HIMEM`

`IF, INK, INKEY, INKEY$, INP, INPUT, INSTR, INT`

`JOY`

Index

GAMES AND GRAPHICS PROGRAMMING ON THE AMSTRAD COMPUTERS CPC 464, 664 and 6128

This book assumes some knowledge of BASIC and is written for those who wish to become more familiar with Amstrad's CPC 464, 664 and 6128.

The author introduces the Amstrad's graphics capabilities and demonstrates how to use them to produce animated games and static graphics displays. One of the main aims of the book is to help in the design of long programs by looking at program structure and the way in which a large program can be constructed practically from small program segments.

The book includes a number of program listings illustrating the ideas discussed in the various chapters. Further, at the end of each chapter there are sections of a longer graphics game which is built up in the course of the book in easy-to-type-in stages. Programming utilities, to help you design your own graphics characters and experiments with the sound capabilities, are also included.

The Author
Steve Colwill was head of computer sciences at Sandhurst School in Berkshire before leaving to take up full-time writing. He joined the Home Computer Advanced Course Magazine where he is now technical editor.

ISBN 0-7447-0032-9

9 780744 700329

£9.95