An Introduction to Programming the ORIC-1

R. A. & J. W. PENFOLD

AN INTRODUCTION TO PROGRAMMING THE ORIC–1

AN INTRODUCTION TO PROGRAMMING THE ORIC–1

by
R. A. & J. W. PENFOLD

BERNARD BABANI (publishing) LTD
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND

First Published — September 1983

British Library Cataloguing in Publication Data
Penfold, R. A.
An introduction to programming the ORIC—1.
1. ORIC—1 (Computer) — Programming
I. Title II. Penfold, J. W.
001.64'2 QA76.8.0/

ISBN 0 85934 104 6

Printed and bound in Great Britain by Cox & Wyman Ltd, Reading

PREFACE

A modern microcomputer such as the ORIC–1 looks deceptively simple, and appears to be no more difficult to use than (say) a calculator or a typewriter. If you only use ready-made software a microcomputer is in fact as easy to use as most other types of electronic consumer goods, but if you want to write your own programs and to fully utilise the facilities of the machine there is a great deal to learn. However, learning to program with competence is not too difficult provided it is tackled in the right way, starting with the more straightforward commands and short programs, and then gradually introducing more complex instructions and developing longer programs.

In this book most aspects of programming the ORIC–1 microcomputer are covered, the omissions being where little could usefully be added to the information provided in the ORIC–1 manual. Starting with simple commands and programs the more difficult topics such as animated graphics and using the sound commands are gradually introduced together with longer and more sophisticated programs which illustrate their use. The main programs are mostly games, and these serve as an interesting way of demonstrating points and gaining programming experience even if you eventually intend to produce programs for more serious applications. Hopefully, using the games will provide a great deal of fun and amusement and they will not merely act as demonstration programs.

R.A.Penfold

ACKNOWLEDGEMENT

The Publishers would like to thank ORIC Products International Ltd for kindly allowing the use of the photograph of ORIC–1 on the cover of this book.

CONTENTS

Chapter 1

VARIABLES AND CODES

Among low-priced computers, ORIC–1 is one of the most attractive, with one of the most pleasant to use keyboards. With its serial attribute system it also allows a full-colour screen display with remarkably economic use of memory. This book is a guide to how to make the best use of its many features.

You can't go far in computing without an understanding of variables. A good analogy of a variable is a pigeonhole in an office filing system. We can store things in the pigeonhole, we can look to see what the pigeonhole contains, we can alter the contents of the pigeonhole, and we can make a copy of the contents of the pigeonhole and store it in another, leaving the original unaltered.

Our computer pigeonholes have labels, so that we can refer to them easily. These labels are somewhat prosaically called variable names. These need some discussion, as ORIC is a little fussy about what it will accept as a variable name.

For a start, all variable names must begin with a capital letter, though numbers (but not lower case letters or punctuation marks) may be used subsequently. Variable names in ORIC BASIC may be of any length (subject to the limitation that a program line cannot be longer than 78 characters), but ORIC only uses the first two characters to distinguish between one variable and another. It may seem pointless to use variable names of more than two characters, but in fact it is often very useful to use words that give some indication of what the variable contains, or what it is to be used for. This is called mnemonic naming.

It is also important that the variable name does not contain any BASIC keyword (the words which the computer recognises as specific instructions). This can be limiting when you are trying to use mnemonic names, as it means, for example, that you cannot have TOTAL as a variable name (TO is a keyword) or SCORE, as it contains the boolean operator OR. If you want to invent computer cricket, you can't have RUNS, and if you turn to football you can't have GOALS. GO is actually a keyword, though in the V1.0 BASIC on which this book is based, it doesn't actually do anything. Two other words *not* in the manual but which cannot be used as variable names are INVERSE and NORMAL. Obviously, scope for development has been incorporated in the system.

There are three types of variables. Two of these contain numbers, on which arithmetic and algebraic operations may be performed. These are collectively called numeric variables. The third type is used to contain strings of characters, usually letters or words or sentences, but numbers, and also punctuation marks may be included. These are called string variables. ORIC has an impressive collection of commands for manipulating strings (put bluntly, this means lots of ways of chopping them up), and strings can also be joined together with the + operator.

The most frequently used type of variable is the real or floating point variable. In ORIC, these may hold any number between 2.93874E–39 and 1.70141E38. These variables are given simple names like A or X or N2 or TRIES.

The other type of numeric variables are called integer variables. As the name indicates, these can be used to contain whole numbers only. If you try to store a number with a decimal point in it, the computer will truncate it to the nearest whole number *smaller* than it before storing it. It is important to remember that the computer never rounds up. Even 9.9999 will be truncated to 9. The range of numbers which can be stored in an integer variable is from 32767 to –32768. The advantage of integer variables when whole numbers are being dealt with is that they use up considerably less memory space, and that operations on integer variables usually execute faster than operations on floating-point variables. Integer variable names are indicated by a per cent sign as a suffix, for example A%, X%, ROW%.

String variables can contain up to 255 characters, and can contain any character that can be produced from the keyboard. As the longest line that can be typed into ORIC is 78 characters, a string 255 characters long obviously cannot be entered in one go, but it is possible to produce long strings by joining several short ones together, a process known as concatenation. Strings *can* contain BASIC keywords. These will not have any effect on operations, and cannot be made to have any effect. This also holds for punctuation marks, such as commas, which are used for particular purposes in BASIC.

Special control characters, such as the serial attributes used in ORIC to control the displayed colour on the screen, can be incorporated in strings, and these *will* have their effect when the string is PRINTED or PLOTTED. (More of this in the next chapter.) String variables are indicated by a dollar sign as a suffix to the name, e.g. A$ X$, NAME$, LTTR$.

It is important to understand that when we perform an arithmetic operation on numeric variables, or a string manipulation on string variables, in creating a new variable we do not alter or destroy the old one(s). For example, if we have a program line "COST=PRICE+TAX" we create a new variable COST which is equal to the sum of the variables PRICE and TAX, which keep their values. To go back to our original analogy, this line could be explained as "make copies of the contents of the pigeonholes labelled PRICE and TAX, add the copies together, and store them in a new pigeonhole, and label it COST. It could be, of course, that when the computer comes to this line in a program, there is already a previously-used variable called COST. In this case, the value of COST would, of course, be altered.

There is thus a clear distinction between the ways in which variables are used in BASIC and the ways in which they are used in algebra. For example, X=X+1 would be a nonsense in algebra, but lines like it are frequently used in BASIC, where it means "make a copy of the contents of pigeonhole X add 1 to it, and replace the current contents of X with it". Remember that when we refer to a variable, we are referring to the contents of a memory location, and that the variable name is the label of that location. Thus if we say X=10 we do not, as in algebra, mean X represents the number 10, but rather that "the contents of memory location labelled X is 10". This distinction is not always important in BASIC, but it can be very important in other computer languages, and especially in assembly language and machine code, so it is as well to understand it from the start.

Sometimes it is very useful to group together variables which are used for the same purpose, for example, in a data file of names and telephone numbers. In BASIC, such a grouping is called an array. An array can be thought of as a row of pigeonholes, all identified by the same name, and distinguished from each other by a number. In the case of our data file, we could call one row of pigeonholes NAMES, and identify the individual pigeonholes as NAMES 1, NAMES 2, and so on.

This is very similar to what is done in BASIC, but the number must be enclosed in brackets, and the numbering starts from 0, not 1. Thus our variables in the array would be called NAMES$(0), NAMES$(1), and so on, according to how many names and numbers we want to store.

ORIC is unusual in that it automatically pre-dimensions arrays to 11 elements (numbered 0 to 10). Most other BASICS require that arrays be declared before use. It is necessary to pre-declare arrays if we want more than eleven elements. The keyword DIM (abbreviated from DIMension) is used for this purpose. Thus DIM NAMES$(100) would create an array to store up to 101 names.

You may have noticed the dollar symbol after NAMES in our BASIC example, and concluded from this that string, integer, and real arrays are identified like string, integer, and real variables, and you would be right. NAMES$(4) would be an element in a string array, LINE%(67) an element in an integer array, and X(0) the first element in a floating-point array.

In our example of a file of telephone numbers and names, we could use two arrays, one called NAMES$(N), and the other called NUMBERS$(N). (When we do not intend performing arithmetic on numbers, as would be the case with telephone numbers, it is often more convenient to store them as strings.)

As an alternative, it is possible to dimension an array with two dimensions. We could set up an array with two rows of 101 elements, storing the names in one row, and the numbers in the corresponding positions in the other row. The statement to set up such an array would take the form DIM BUSBY$(100,1). If required, we could have a third row to hold the addresses as well. This could be declared as BUSBY$(100,2). Note that this is still a two-dimensional array. It has 101 elements in one dimension and 3 in the other.

In theory, the number of elements you can have is limited only by the available memory space in the computer. In practice, large arrays, and especially string arrays, gobble up memory at an alarming rate, so it is not good practice to make arrays larger than you really need.

It is possible to have arrays with more than two dimensions. In fact, the maximum number of dimensions is limited to 255, but with a maximum line length of 78 characters, such an array would be difficult to declare! A three-dimensional array would be dimensioned by the statement DIM ARRAY(5,5,5). The total number of elements in a multi-dimensional array is given by multiplying together the number of elements in each dimension, so even this small example would have (6*6*6)=216 elements, so beware! If you find arrays with more than two dimensions difficult to comprehend, don't worry. They are not particularly useful and are not often used in programs.

The great value of arrays is that they fit easily into loops, and this makes it relatively easy to design a program which can sort through a list of data and pick out a particular item. For instance, with our telephone numbers file, to pick out a name and print out the number. In fact, we will do just this in a later chapter.

To conclude this chapter, let us consider how a computer stores letters in memory. As you no doubt know, a computer stores things as numbers. In fact, at the lowest level it stores things as 1s and 0s, or more pedantically as ons and offs. A code is therefore used which identifies each letter, numeral, and punctuation mark by a number. The code used by ORIC and almost all other computers is the American Standard Code for Information Exchange, usually abbreviated to ASCII and pronounced "askey" Fortunately, this code is fairly well standardised, though different codes are used by different computers for some symbols, for example the pound and hash (#) symbols. These code numbers are also used to send characters to the printer.

One of the appendices in the ORIC manual gives the codes of all ORIC's available characters, but listings 1 and 2 are a more *fun* way of exploring the ASCII codes. Listing 1 prints the ASCII code of any letter or symbol you care to enter. You do not need to press RETURN after the key – the program responds immediately. In fact, if you do press RETURN you will discover that this has an ASCII code, as do the arrowed cursor keys, and the space bar! In fact, all the keys except the SHIFTs and CTRL generate codes. You will discover that holding down SHIFT or CTRL while pressing another key will (in most cases) alter the code the key generates. It should be obvious why this is so.

```
   10 REM *Listing 1*
   20 REPEAT
   30 GET LTR$
   40 PRINT ASC(LTR$)
   50 UNTIL FALSE
>
```

One small warning about this program. If you press the single inverted comma (') key, the program will stop with an out of range error message. There is no real reason for this, it is just a small fault in the operating system, or what is normally termed a bug. Even the best computers have them.

```
   10 REM *Listing 2*
   20 REPEAT
   30 INPUT CODE
   40 PRINT CHR$(CODE)
   50 UNTIL FALSE
>
```

Listing 2 will print the character corresponding to any code number you enter. Note that if you enter numbers below 32 odd things may happen, and some codes will jam up the computer completely. This is because numbers below those used for characters are used as control codes. Numbers above the character set are also used as control codes in ORIC, in particular to control colour, as we shall see in subsequent chapters.

Don't worry about how these programs work for the moment, all will be explained in due course.

Chapter 2

INS AND OUTS

The ORIC has a powerful extended BASIC, and the sign of a powerful language is that there are often several ways of achieving the same end. One of the skills of programming is deciding which is best to use in a particular circumstance. The methods of displaying information on the screen and inputting information into the computer are good examples of this.

There are three ways of sending information to the screen. Firstly, by the PRINT statement; secondly by using PLOT; and thirdly, by POKEing directly into the memory area used for the screen display. These all have their particular advantages and limitations.

PRINT is very flexible in what it can print on the screen but, in ORIC BASIC, it is limited in the amount of control it allows in positioning text. Numbers can be printed on the screen either directly (e.g. PRINT 1) or by naming a variable (e.g. PRINT A). The same holds for strings (e.g. PRINT "Hello" or PRINT H$). PRINT can also be used to put control characters on the screen, for example to change the foreground or background colour, and these various items can be mixed together in any order. For example, the following is a perfectly legal PRINT statement (though actually a bit too long for ORIC!).

```
PRINT CHR$(129);"The Total is ";CHR$(131);A;CHR$(129);"points,
which is";CHR$(131);D$
```

As well as being used to print control characters on the screen, CHR$(N) can also be used to produce all the ASCII letters and symbols, i.e. PRINT "A" and PRINT CHR$(65) will have exactly the same effect. This is normally only of academic interest, but it can be used to print on the screen the one symbol which is not available from the keyboard, the copyright character, ASCII code 96. It can also be used to print random characters from random numbers in the correct range, and this is useful for some games, as we shall see at the end of this chapter.

When a PRINT statement is executed, the text appears on the screen at the position of the text cursor. This is normally at the extreme left of the next available line (excluding the two left-hand columns reserved for the INK and PAPER attributes). In ORIC BASIC, there is no way of controlling the vertical position of the cursor. There is a TAB command that can be used to move the cursor in from the left-hand edge of the screen by a specified number of spaces, but in the current version (V1.0 BASIC) this does not work correctly. PRINT TAB(10) or any number less than ten has no effect, and numbers larger than ten tend to have unpredictable results. This command can be useful, but one normally has to use trial-and-error to produce the desired result.

An alternative is the function SPC(N), which prints N spaces on the screen. This can be used at the beginning of the line of text, and between sections of it, and is an easy-to-use and effective way of controlling the screen layout. The maximum number of spaces that can be printed is 255, which of course would result in several blank lines on the screen.

The comma can be used in a similar way to SPC(N), but it moves the text cursor to pre-set positions on the screen, rather like the tab settings on a typewriter. These settings are, according to the manual, at positions five character-spaces apart, but in fact it seems to be more complicated than that, and the effect of commas is different depending on whether you are printing numbers or strings.

It is permissible to use the comma (or several commas) immediately after the word PRINT if required, as in this example.

```
90 PRINT,,A;" points,";,,T;" tries"
```

Note that the comma within the inverted commas after points does not have any effect. Also note the semi-colons after A, "points," and T. These instruct the computer not to go on to a new line. They are not always necessary, but it is best to be on the safe side and include them. A further point is the spaces between the inverted commas and the first letters of points and tries. If these are not included the numbers (the variables A and T) will be printed right next to the words.

As single-line spacing can have a crowded look, and be difficult to read on the screen, it is often desirable to put extra spaces between lines. This can be done by putting PRINT in a program line with nothing after it. As ORIC allows multi-statement lines, it is possible, for example, to put in three line spaces by a line such as:

```
180 PRINT:PRINT:PRINT
```

This is clumsy, but effective. Remember, considerable typing can be saved by using ? instead of PRINT, but the word will appear in full on subsequent listings.

A further advantage of PRINT is that, when printing numbers or numeric variables, it is possible to have a calculation in the print line. This can be as simple as "PRINT2+2" or it can be a complex function or equation. However, it must be noted that the result of such a calculation is not put into a variable, and so is not available to the rest of the program.

PLOT allows more control of the positioning of items on the screen, as both X (horizontal) and Y (vertical) co-ordinates must be specified, but it is in some ways more limited in what it will print on the screen.

PLOT can be used to print a string of characters on the screen, and the string may either be written after the PLOT co-ordinates, enclosed in inverted commas (e.g. PLOT 5,10, "Hello") or a string variable (e.g. PLOT 5,10,H$). PLOT can also be used to put control characters on the screen, for example, PLOT 1,5,1 will make anything else on line 5 appear in foreground red (1 is the code for the red foreground serial attribute). As PLOT interprets any number as a code, it cannot be used to directly print numeric variables on the screen. As an example, the following lines would all have the same effect in a program

```
30 PLOT 10,10,"A"
30 A$=A:PLOT 10,10,A$
30 PLOT 10,10,65
30 A=65:PLOT 10,10,A
30 PLOT 10,10,CHR$(65)
```

The number 65 in these examples is of course the ASCII code for a capital A. Numbers in the range of the ASCII character set will produce these characters on the screen. Higher and lower numbers will be interpreted as control characters. The highest legal number is 255 (the largest number that can be stored in a single byte). Anything above this will produce an illegal quantity error message.

It should be obvious that in the last example, the CHR$( ) is completely superfluous. However, "PLOT X,Y,N" does not always have the same effect as "PRINT CHR$(N)". For example, "PLOT X,Y,4" will be interpreted as the colour attribute for foreground blue (as will PLOT X,Y,CHR$(4)) but "PRINT CHR$(4)" will be interpreted as CTRL D, the auto double-height toggle control. To access the colour control codes through a PRINT statement you have to use the ASCII code numbers above 128. Foreground blue is obtained by code (128+4), i.e. PRINT CHR$(132).

PLOT can only be used to put a single string on the screen. Complex strings cannot be built up as they can in a PRINT statement. However, a complex string can be built up as a string variable, and then PLOTted, as in this example.

```
110 LNG$=NAME$+" lives at "+ADDRESS$
120 PLOT 1,5,LNG$
```

It is often desirable to print a numeric variable on the screen at a fixed position, for example to display the score in a game, and although this cannot be done directly using PLOT, there is a way of achieving the desired result by first turning the numeric variable into a string, using the function STR$(N), where N is the name of the numeric variable we wish to plot. Unfortunately, in V1.0 BASIC, there is a bug in this function which inserts a colour control code in front of the number, so the number is printed in green. This is a considerable problem if you want to PLOT the number on a green background!

The only way round this is to strip off the unwanted first character, by means of the following somewhat convoluted method. There is no need to understand how this works at this stage. In this example N is the numeric variable we wish to display on the screen at a fixed position, co-ordinates 5,10

```
100 N$=STR$(N)
110 NP$=RIGHT$(N$(LEN(N$)-1) )
120 PLOT 5,10,NP$
```

One of the most important uses of PLOT is in animated graphics, and this will be dealt with in detail in the next chapter.

The BASIC function POKE directly inserts a given number into a specified memory location. If the memory location is in the area of memory used for the screen display, then the character corresponding to the number will appear on the screen (or its effect, if the number is in the control-code range). Generally, POKE is not very useful as it can only send one character to the screen, and cannot do anything that cannot be done equally well by PLOT. It is generally much more difficult to work out the memory addresses used by POKE than to work out the co-ordinates used by PLOT.

POKE is useful, however, to put control codes into the far-left screen column used in TEXT mode to control the PAPER colour. This is not normally accessible to PLOT. In this way, it is possible to create a two-(or more) colour background without losing further screen space.

It is also possible to POKE into the status line at the top of the screen (where CAPS and loading/saving messages appear), but this is only of limited interest.

INPUTS

In ORIC BASIC there are three statements to allow for entering information into the computer while the program is running. These are INPUT, GET, and KEY$. INPUT and GET can be used to enter either numbers or strings. KEY$ as the name suggests, can only be used for strings. INPUT can be used to enter any number of characters, subject to the maximum allowable line length of 78 characters and spaces, GET and KEY$ allow only a single character to be entered.

INPUT and GET cause the computer to interrupt the program and wait for input, whereas KEY$ does not wait, and input will only occur if a key is pressed at or before the instant that the computer reaches the KEY$ statement. KEY$ is often preceded by a pause (using the WAIT statement) to allow time for a key to be pressed. KEY$ is much used in animated games to allow control of the action without interrupting it.

With GET, program execution recommences as soon as a key is pressed. With INPUT, you have to press RETURN to let the computer know you have finished the input.

With all three methods, the input is placed in a variable. With INPUT and GET, the variable name used will determine whether a numeric or string input is required. INPUT NAME$ or GET A$ will specify a string input, INPUT CREDIT or GET X% will specify a numeric input. If a string is input when the computer is expecting a number, it will give a ?REDO FROM START message for an INPUT statement, and go back to the input line, and will stop program execution with a SYNTAX ERROR message for a GET statement. To prevent interruption of the program if a letter key is pressed, it is often better to use GET for string input, and convert the string into a number using VAL It is of course, perfectly legal to put a number into a string. If the first (or only) character in a string is anything other than a number (or +, –, or decimal point), VAL will return the value 0.

The INPUT statement is quite flexible, in that it is possible to put a line of text after it, which will be printed on the screen, so that it can ask for the required input, e.g.

```
100 INPUT "What's your name";NAME$
```

Note that there is no question mark. The computer provides this automatically, whether or not there is any text after INPUT. The semi-colon after the text is compulsory.

It is also possible to have more than one input per input statement, as in this example

```
130 INPUT "Enter three numbers";A,B,C
```

When entering the three numbers, they may be either separated by commas, or RETURN may be pressed after each number. String and numeric inputs may be mixed on one line, but this can lead to practical problems in giving instructions to the program user, and is probably best avoided.

It is *not* possible to put text between the input variable names.

Now for the first real program in this book, a Simple Simon game where the computer prints a string of random letters. You have a few seconds to memorise them, then they disappear, and you have to type them in. GET is used here, so you don't have to press return. The original string then re-appears and the computer gives you your score.

```
10 REM Listing 3
20 REM "Simple Simon" game.
30 REM J.W.P. 5/83
40 TEXT:CLS
50 IF (PEEK(#26A) AND 1)=1 THEN PRINT CHR$(17):REM turns off cursor
60 DIM A%(9,1):REM array to store character ASCII codes
70 COUNT=4:REM controls no. of letters
80 PRINT:PRINT:PRINT:REM 3 line spaces
90 INPUT "What's your Name";NAME$
100 TITLE$=CHR$(20)+CHR$(10)+CHR$(3)+"Simple Simon says..."+CHR$(23
)
110 YOUR$=CHR$(17)+CHR$(10)+CHR$(3)+NAME$+" says..."+CHR$(23)
120 REPEAT
130 CLS
140 PLOT 3,3,TITLE$
145 PLOT 3,4,TITLE$
150 WAIT 200
160 FOR LTRS=0 TO COUNT:REM start of random letter generation
170 A%(LTRS,0)=RND(1)*26+65:REM stores random ASCII codes in array
180 PLOT (6+LTRS),7,A%(LTRS,0):REM plots letters on screen
190 WAIT 200:REM pause between letters
200 NEXT LTRS:REM loop back for next random letter
210 WAIT 500:REM time to memorise letters
220 PLOT 2,7,7:REM makes letters disappear
230 PLOT 3,11,YOUR$
235 PLOT 3,12,YOUR$
240 FOR YRTRN=0 TO COUNT
```

```
250 GET ANS$:REM player enters letters

260 PLOT (6+YRTRN),15,ANS$:REM prints player's entries

270 A%(YRTRN,1)=ASC(ANS$):REM stores player's entries in array

280 NEXT YRTRN

290 WAIT 100

300 PLOT 2,7,0:REM makes letters reappear

310 FOR S=0 TO COUNT:REM start of scoring

320 IF A%(S,0)=A%(S,1) THEN PTS=PTS+1:REM PTS variable is points sc
ored

330 NEXT S

340 PRINT "You scored ";PTS;" out of ";COUNT+1

350 IF PTS=COUNT+1 THEN PING:REM reward for full score!

360 WAIT 500

370 COUNT=COUNT+1:REM increases no. of letters by 1 for next round
380 PTS=0:REM resets points counter

390 UNTIL COUNT=10:REM game ends at 10 letters

400 CLS:PRINT CHR$(17):REM turns cursor on

410 STOP:REM end of program
```

At the beginning of the game, the computer asks for your name. This is an INPUT line, so you do have to press return after this.

This program mostly uses ideas we have already discussed, and with the explanatory REMs included in it, you should be able to follow how it works. (There is no need to type the REMs in if you don't want to – these are just REMarks which you can include in programs to remind yourself how it works – very useful when you list it a few months later. The computer ignores anything after a REM.)

The FOR. . .NEXT loops will be discussed in a later chapter. For now, it is enough to know that the computer will repeat everything between FOR and NEXT a specified number of times.

Chapter 3

ANIMATION AND LOOPS

Animation and Loop structures in programming go together naturally, as most, if not all, animation is based on loops, and animation is a good way to demonstrate loops.

A loop is, simply, a way of making a computer repeat a set of instructions a number of times. An entire program may consist of a single loop, and in fact the first two programs in this book are like that. Loops are a vital part of programming, and there are very few computer programs that don't contain some sort of loop somewhere.

ORIC BASIC has two loop structures. The first, the FOR. . .NEXT loop, is used to repeat a set of instructions a set number of times. The second, the REPEAT . . .UNTIL loop is used to repeat a set of instructions until a specified condition is met.

The following is a simple example of a FOR. . .NEXT loop.

```
10 FOR C=1 TO 10
20 PRINT C
30 NEXT C
```

This loop will print out all the whole numbers between 1 and 10. A step size of +1 is assumed, but a different size can be specified. If the first line was changed to 10 FOR C=1 TO 10 STEP 2 the program would print out all the odd numbers from 1 to 9. The variable C (usually called the control variable) will never actually take the value 10. The loop will terminate when the control variable is greater than the specified value. On termination, C will have the value 11, and it is important to remember this if the control variable is used outside the loop for any purpose.

It is important to know that FOR. . .NEXT loops always execute at least once. If the first line was changed to 10 FOR C=11 TO 10 the program would print out 11. This is because the terminating condition of the loop is not checked until it reaches the NEXT line.

It is perfectly possible to have a negative step size, e.g. FOR C=10 TO 1 STEP–1. In this case the first number after the control variable must be larger than the second, if the loop is to execute more than once.

It is quite legitimate to use the control variable in calculations, or for other purposes (in the Simple Simon game in the last chapter, for instance, control variables were used to place the ASCII code values sequentially in the array, and to position the letters on the screen). However, care must be taken if the value of the control variable is altered. It is permissible to do this if necessary, but if the control variable is prevented from reaching the terminating value, the loop will continue forever, or at least until you press CTRL C, or pull the plug!

The following is a simple REPEAT. . .UNTIL loop.

```
10 REPEAT
20 R=R+1
30 PRINT R
40 UNTIL R=10
```

This, in fact, does exactly the same as the simple FOR. . . NEXT loop. If we were to change line 20 to 20 R=R+2 it would print out the odd numbers like our second FOR. . .NEXT example, except that it would just keep on going! In a REPEAT. . .UNTIL loop the condition specified in the UNTIL line must be met exactly if the loop is to terminate. Like FOR. . .NEXT loops, REPEAT. . .UNTIL loops always execute at least once.

Of course, this example is much better done as a FOR. . . NEXT loop. REPEAT. . .UNTIL loops are really for when we cannot determine in advance how many loops will be necessary before the condition to end the loop is met.

If you have tried the first two programs in this book, you will have found that they go on forever, or until an error occurs. You will see that the last line in these programs is UNTIL FALSE. The exact meaning of FALSE will be discussed in a later chapter, but in fact it sets a condition which will never be met! This is a standard way of producing an infinite loop, and is very useful in simple exploratory programs.

Animation

All moving pictures, be they television, cine, computer games, or what-the-butler saw, work by presenting a series of slightly different images to the eye at such speed that the brain interprets the changes as movement. In computer animation, a character can be made to appear to move across the screen by PLOTting it in successive positions on the screen in quick succession. This can be easily achieved by using a FOR. . . NEXT loop, and using the control variable as one of the co-ordinates in the PLOT statement. Of course, once a character is PLOTted, it stays PLOTted, so it is necessary to rub it out again. This is most easily done by PLOTting a space over the character. This is a very simple example:

```
5 CLS
10 FOR Y=0 TO 24
20 PLOT 20,Y,"*"
30 WAIT 10
40 PLOT 20,Y," "
50 NEXT Y
```

The WAIT 10 line is to give the image time to register on the retina.

Of course, to be of any use in an animated game, we need to be able to control the movement of our character on the screen. Since we have keys with appropriate arrows on them, (the cursor control keys), it would be nice to use them. As we have discovered in the first chapter, these keys generate ASCII codes, and so we can use KEY$ to get an input from these keys, and the function ASC( ) to determine which has been pressed.

This brings us to our first animated game program. In this, an arrow is fired at a target consisting of three stars. The player has to steer the arrow onto the target using the cursor keys. Note that this program is essentially a FOR. . .NEXT loop, giving the movement, inside a REPEAT. . .UNTIL loop, which keeps firing arrows until a hit is scored.

```
1 REM Listing 4
2 REM Arrow Game
5 REPEAT
10 CLS
20 PLOT RND(1)*20+10,0,"***"
30 X=25-RND(1)*10
40 FOR Y=25 TO 2 STEP-1
50 X$=KEY$
60 IF X$="" GOTO 100
70 IF ASC(X$)=8 THEN M=-1
80 IF ASC(X$)=9 THEN M=1
90 IF ASC(X$)=32 THEN M=0
100 X=X+M
110 IF X<1 THEN X=1
120 IF X>38 THEN X=38
130 PLOT X,Y,"^"
140 WAIT 4
150 PLOT X,Y," "
160 NEXT
170 M=0
180 UNTIL SCRN(X,Y-1)=42
190 EXPLODE
200 GET RE$:GOTO 5
```

Line 20 PLOTs the target at random positions at the top of the screen. Line 30 sets the horizontal starting point of the arrow, also at random. Line 40 is the start of the FOR. . .NEXT loop. Note that it counts backwards, to give movement up the screen. Line 50 is the input line for key presses. Lines 70–90 determine which key has been pressed and set the value of the variable M (for Movement) accordingly. A slight problem with ORIC is the delay before the auto-repeat action of the keys starts. This makes it impractical to get continuous movement by holding the appropriate key down, hence this part of the program is written so that once an appropriate key has been pressed, the sideways movement continues until the other key is pressed to move in the opposite direction. Line 90 allows the space bar to be used to stop horizontal movement.

As these decision-making lines tend to slow down the action, line 60 skips them if no key has been pressed on that pass of the loop.

Line 100 actually alters the value of the horizontal co-ordinate X if an appropriate key has been pressed.

If the arrow were to go off the edge of the screen, the program would stop with an error message. Lines 110–120 prevent this by trapping illegal values of X. Lines 130–140 actually PLOT and rub out the arrow.

Line 160 is the end of the FOR. . .NEXT loop. If no hit has been scored, the program goes back to the start and another arrow is fired. Line 170 is to make sure the new arrow does not carry on with any sideways movement of the old arrow.

If a hit has been scored it will be detected by the function SCRN in line 180. 42 is the ASCII code for the star character. In this case, the program stops with an appropriate sound effect. Pressing any key will start the program again (line 200).

Note that this program has been written without REMs in the action loop. This is to make it run as fast as possible. It takes a computer as long to ignore a REM as it does to execute an instruction! This program has also been written with single-letter variable names, as this also seems to speed up action in some cases. However, as X and Y are the normal way of denoting horizontal and vertical co-ordinates, using these as the names of the variables which control movement in these directions just about counts as mnemonic naming anyway! I like to reserve these letters for this purpose.

Listing 5 is a "DUCKSHOOT" game which shows how several things can be made to move at once. A line of ducks (a redefined character) move across the top of the screen, and the player shoots them by pressing the space bar.

```
10 REM Listing 5

20 REM *DUCKSHOOT*

30 REM J.W.F 5-83

35 IF(PEEK(#26A) AND 1)=1 THEN PRINT CHR$(17)

40 FOR DUCK=46624 TO 46630
```

```
50 REM Duck character definition
60 REM memory addresses are for 48K
70 READ A
80 POKE DUCK,A
90 NEXT DUCK
100 DATA 0,26,27,12,30,63,30
110 CLS:PAPER 4
120 PLOT 10,5,0:PLOT 10,7,0
130 DUCK$=CHR$(3)+"D  D  D  D"
140 B=20:REM arrow start point
150 REPEAT
160 FOR X=RND(1)*5 TO 29
170 PLOT X,0,DUCK$
180 IF KEY$=" " THEN FIRE$=" ":SHOOT:SHOTS=SHOTS+1
190 IF FIRE$="" THEN WAIT 10:GOTO 240
200 PLOT 19,B+3," ":REM rubs out arrow
210 PLOT 19,B,"^"
220 IF SCRN (19,B-2)=68 THEN PING:HITS=HITS+1
230 B=B-3:IF B<=0 THEN FIRE$="":B=20:PLOT 19,2," "
240 NEXT X
250 PLOT 29,0,"          "
260 PLOT 1,5,"SHOTS"
270 PLOT 7,5,STR$(SHOTS)
280 PLOT 1,7,"HITS"
290PLOT 7,7,STR$(HITS)
300 UNTIL SHOTS>=20
310 PRINT "Press any key to go again"
320 GET RE$
330 RUN
```

Actually, although there are four ducks on the screen, there are really only two moving objects, the ducks and the arrow, as the ducks are all part of a long string, assembled in line 130. Note that this string effectively rubs itself out, because of the spaces between the ducks, and the colour attribute to the left of the last duck. It is necessary to rub out the line when it has gone as far to the right as it can go. This is done by line 250. There are 10 spaces between the inverted commas.

The movement of the ducks is controlled by the control variable X but the movement of the arrow is completely independent, so the player can fire at any time. It is possible to fire more than once on each pass of the ducks, but not to have more than one arrow on the screen at a time.

A score is displayed on the screen, but it is only updated at the end of each pass of the ducks. This is because putting the score plotting lines inside the FOR. . .NEXT loop was found to slow the action too much.

As, in this game, the direction of the arrow is not under control, it is only necessary to give an impression of its position. It is therefore only PLOTted in every third position (see line 230). This also allows much faster flight. In fact, if you are quick on the trigger you can get two ducks on one pass!

Line 300 stops the game after 20 shots. Note that we test for "greater than or equal to", as it is possible for more than 20 shots to have been fired.

When you have understood the ideas in these two programs, you might like to try to put them together, so that you use the keys to fire and steer an arrow onto a moving target.

Finally, to show how the cursor control keys are used to control movement in four directions, a short program that allows the computer to be used in HIRES mode as a sketch-pad, in a similar way to ETCHA–SKETCH (Listing 6).

```
2 REM Listing 6
5 REM Hires SketchPad
7 REM J.W.P 4-83
10 HIRES
15 C=1
20 CURSET 120,100,C
30 REPEAT
40 GET A$
50 IF ASC(A$)=9 THEN X=1
```

```
60 IF ASC(A$)=8 THEN X=-1

70 IF ASC(A$)=10 THEN Y=1

80 IF ASC(A$)=11 THEN Y=-1

90 CURMOV X,Y,C

100 X=0:Y=0

110 UNTIL ASC(A$)=32
```

This program is very straightforward and you should be able to understand how it works by now. You may like to work out what happens if line 100 is omitted (then take it out and see if you were right!).

Line 110 lets you escape to command mode, so that you can change the INK or PAPER colour, or the value of C, (0 draws in PAPER colour (in other words rubs out), 1 draws in INK colour, 2 inverts the colour it passes over, 3 does nothing), or use the DRAW and CIRCLE commands. You can get back into sketch mode by typing GOTO 30, and you can also use this if you go over the edge of the screen and generate an error message. Remember, typing RUN will destroy your picture!

Chapter 4

ATTRIBUTES, CHARACTERS, AND TIME

Serial Attributes

The ORIC—1 uses the system of serial attributes to control the colour (and other aspects) of the screen display. This system, which is used in all modes, allows a full-colour display with flashing and double-height characters, while using remarkably little memory space in comparison with parallel-attribute systems. It is slightly more limiting in the control it allows over the screen display, but this is not a great disadvantage.

In essence, a serial attribute can be thought of as a special character which, when printed on the screen, controls one particular aspect of everything that appears on the same line and to its right, unless and until another attribute is printed that cancels its effect.

When an attribute is printed on the screen it takes up one character space, which appears as a blank (but a space occupied by a background colour attribute will appear in its own background colour).

The effects that are controlled by the serial attributes are foreground colour, background colour, flash (foreground only, flashing backgrounds are not possible), double height characters, and whether the standard (alphanumeric) or alternate (mosaic graphic) character set is printed.

The global INK and PAPER commands work by placing the appropriate attributes in the two far-left columns of the screen. This is why these two columns cannot be used for characters.

The LORES modes are slightly different to the TEXT and HIRES modes, in that they have no global PAPER command. The column that normally holds the background attributes for the whole screen is, in these modes, used to determine whether the standard or alternate character set is used. A second difference is that when a background colour attribute is placed on the screen, it directly affects only the position it occupies, instead of making the whole line to its right take on the colour. However, any character (or line of characters) placed *directly* to its right will have that background colour. Any gap will cause the background to revert to black. (In this context, a space between words counts as a foreground character, not a gap).

In the TEXT and LORES modes, the attributes can be sent to the screen in either PRINT or PLOT statements, as we saw in Chapter 2. As mentioned in that chapter, the numbers which must be used with PRINT and PLOT are different. Here is a full list of the available effects, and the numbers to use to PRINT and PLOT them on the screen

ATTRIBUTE EFFECT

Foreground	*PLOT*	*PRINT*
Black	0	128
Red	1	129
Green	2	130
Yellow	3	131
Blue	4	132
Magenta	5	133
Cyan	6	134
White	7	135
Character Type		
Standard steady single height	8	136
Alternate steady single height	9	137
Standard steady double height	10	138
Alternate steady double height	11	139
Standard flashing single height	12	140
Alternate flashing single height	13	141
Standard flashing double height	14	142
Alternate flashing double height	15	143
Background		
Black	16	144
Red	17	145
Green	18	146
Yellow	19	147

Blue	20	148
Magenta	21	149
Cyan	22	150
White	23	151

Listing 7 shows how it is possible to have all the foreground and background and flashing colours on the screen at the same time. If you experiment, you will also find you can add alternate and double-height characters.

```
5 REM listing 7
10 REM Printing attributes demonstration"
20 FOR CODE=129 TO 134
30 PRINT CHR$(CODE);"Foreground Colours"
40 PRINT CHR$(CODE+16);"Background Colours"
50 PRINT CHR$(140);CHR$(CODE);"Flashing foreground"
60 PRINT CHR$(140);CHR$(CODE+16);"What does this do?"
70 NEXT CODE
80 GET A$:END
```

To place attributes on the screen in HIRES mode, the PLOT numbers are used, but with a different statement, FILL.

The horizontal colour resolution in HIRES mode is the same as in TEXT mode, i.e. there are 40 colour cells across the screen. Vertically, however, the colour resolution is the full 200 lines. FILL places attributes in a given number of lines downwards, by a given number of cells horizontally, starting at the cursor position. The FILL statement is therefore normally preceded by a CURSET statement.

The horizontal position of the cursor is not critical. Any setting between 0 and 5 will FILL the first column of cells, anywhere between 6 and 11 the second, and so on.

The FILL statement therefore has three arguments. The first specifies how many lines are to be filled, the second how many cells are to be filled, and the third specifies the attribute. Thus, starting with the cursor at 0,0 (top left), FILL 100,1,20 would fill the first column to half-way down the screen with the background blue attribute. The top half of the screen would therefore turn blue. The same FILL statement, but with the cursor at 120,99 would turn the lower right-hand quarter of the screen blue.

As you cannot DRAW over a cell which contains an attribute, it is not normally a good idea to FILL more than one cell horizontally. FILLed cells always appear in the current background colour.

Listings 8 and 9 show some of the things that can be done with the FILL statement in conjunction with CIRCLE. Note that there is a lot of FILLing to be done in the middle part of Listing 8, so there is a delay when nothing much seems to be happening.

```
5 REM Listing 8
10 REM Hires attributes
20 HIRES
30 REM cursor goes to 0,0 on HIRES command
40 FOR BC=0 TO 199
50 COL=INT(RND(1)*8+16)
60 FILL 1,1,COL
70 NEXT BC
80 START=11
90 REPEAT
100 CURSET START,0,3
110 FOR FC=0 TO 199
120 COL=INT(RND(1)*8)
130 FILL 1,1,COL
140 NEXT FC
150 START=START+48
160 UNTIL START>203
170 CURSET 120,100,3
180 FOR C=95 TO 45 STEP-1
190 CIRCLE C,1
200 NEXT C
210 END
```

```
1 REM Listing 9
2 REM ** Flag **
3 REM J.W.P. 4-83
5 HIRES:PAPER 5
10 FOR X=239 TO 6 STEP-6
15 Y=INT(5*X/6)
20 CURSET X,Y-5,3
30 FILL 5,1,18
40 CURSET X,200-Y,3
50 FILL 5,1,21
60 NEXT X
70 CURSET 0,99,3
80 FILL 99,1,18
90 CURSET 6,0,3
100 FILL 99,1,6
110 FILL 99,1,1
120 FOR X=40 TO 200 STEP 15
125 Y=INT(5*X/6)
130 CURSET X,Y,3
140 CIRCLE 20,1
150 CIRCLE 10,1
160 CURSET 240-X,Y,3
170 CIRCLE 20,1
180 CIRCLE 10,1
190 NEXT X
200 PING
```

The fact that background colour attributes only directly affect the position they occupy in LORES modes can be put to effect to draw multi-colour patterns on the screen. Listing 10 is a tapestry of background colour squares, using a simple mathematical base to determine which colour is PLOTted in each position.

```
5 REM Listing 10
10 LORES 0
20 FOR X=0 TO 38
30 FOR Y=0 TO 26
40 IF INT(X/2)<(X/2) THEN C=17
50 IF INT(Y/3)<(Y/3) THEN C=20
60 IF INT(X/2)<(X/2)AND INT(Y/3)<(Y/3) THEN C=19
70 PLOT X,Y,C
80 NEXT Y,X
90 PLOT 0,0,20
100 REPEAT:UNTIL KEY$<>""
```

Multi-colour fields for games can also easily be produced by this method, as we shall see in a later chapter.

There is a peculiarity of the background attributes that can be put to good use in this application. If you use the code numbers normally used with the PRINT statement with the PLOT statement, the actual position occupied by the colour attribute will appear in its complementary (inverse) colour. That is, if you PLOT 129, it will appear cyan (the inverse of red) but will produce the red background colour to its right.

It is also worth noting that the function SCRN can be used to detect the attribute occupying the specified screen position.

Defining New Characters

As both the standard and alternate character sets are down-loaded into RAM on power-up, all the characters can be re-defined. Characters are designed on a six-by-eight grid. Therefore, to redesign a character, you have to specify eight numbers, representing the eight horizontal rows. Each position on each horizontal row contributes a certain value according to its position, and you add these values together to obtain the number which defines that row. Figure 1 gives the values for each position in the horizontal rows, and Figure 2 shows how these are added together to define which positions are on (foreground colour) or off (background). Figures 3 and 4 are

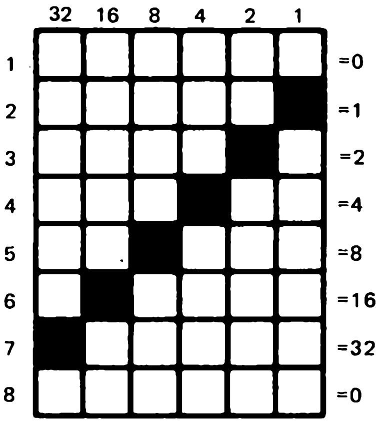

Fig. 1. Values of positions in the row

a couple of demonstration characters, which we will be meeting again later in the book.

Characters are stored in order of their ASCII codes, starting at memory location 46080 in TEXT and LORES modes. In HIRES, they start at 38912. In the 16K version, these addresses are lower by 32768.

The formula for finding the locations for any character is (starting address for mode/model in use)+(ASCII code of character to be redefined*8). This gives the address of the first byte, and the character occupies this and the next seven bytes.

The normal way of POKEing the new values into memory is by way of a FOR. . .NEXT loop, holding the numbers to be POKEd in a DATA statement. You will have seen an example of this in the "DUCKSHOOT" game, where the letter D was

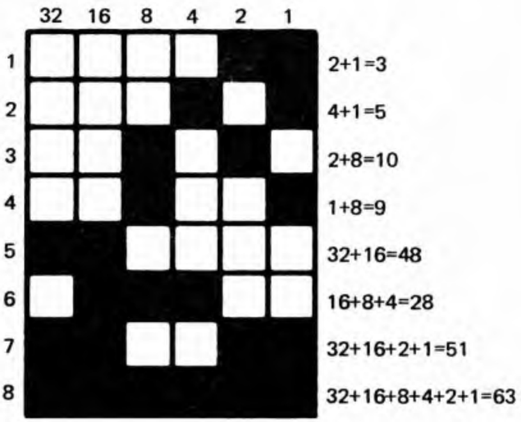

Fig. 2. Examples of values added together to give total row values

turned into a little duck. If you examine this part of the listing, however, you will see that there are only seven numbers in the DATA statement. This is because the bottom row of capital letters is always left empty (it is used for the descenders in lower-case letters) and it was also left empty in the duck, so only the seven rows that were changed had to be specified.

One point you must be careful of in using DATA statements is that when the computer comes to the first READ statement it will commence reading from the first DATA statement in the program, not the one nearest to the READ statement. You must therefore be careful to get the DATA in the right order for the READ statements, and also to correctly match the number of items of data provided to the number of items to

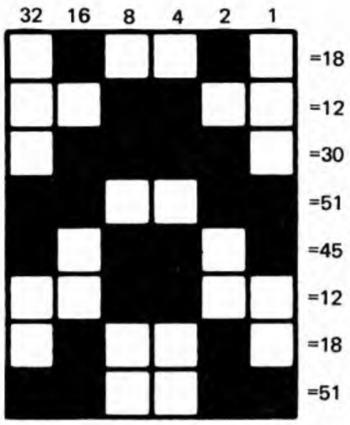

Fig. 3. An "Ugli"

be read, otherwise you may not get the effects intended. If you have too few items of data, you will get an error message.

It is particularly important to keep these points in mind when you write more complex programs using subroutines, where DATA statements may be used for more than one purpose. The DATA statements must be in the correct order for the order in which the subroutines are executed, which is not necessarily the same as the order in which they appear in the program listing.

DATA statements can be spread over more than one line if required. When the computer comes to the last item of DATA on a line, it moves on to the first item in the next DATA statement.

There is a RESTORE statement in ORIC BASIC that sets

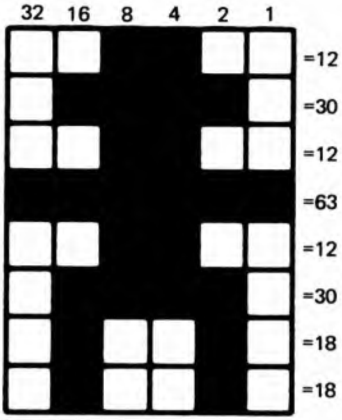

Fig. 4.

the data pointer back to the first item of data in the program. Unfortunately, it cannot be used to set the data pointer to a particular DATA line, as can be done in some other BASICs.

TIME

ORIC BASIC does not have a specific TIME statement, but it does have a WAIT statement, whereby the execution of a program can be held up for a time specified in hundredths of a second. Thus, the statement WAIT 500 will halt program execution for 5 seconds.

In fact, the internal clock used by the WAIT statement is accessible by DEEKing and DOKEing. It is at memory locations 630 and 631 (decimal). This clock counts downwards in hundredths of a second from 65535 (the highest number that can be stored in two bytes) to 0, and then starts again from 65535. When the computer executes a WAIT statement, it sets the clock to the number following the WAIT, and then recommences program execution when it detects the clock crossing zero. If you try putting PRINT DEEK(630) immediately after a WAIT you will find it always returns 65535 (unless ORIC is really on its toes, when you may catch it still at 0).

It is perfectly in order to set and read this clock for other timing purposes in programs, provided we remember that it will be altered by any WAIT statement in the program. The clock is also slowed if the screen scrolls, and this must be borne in mind when using it.

Listings 11 and 12 are two simple reaction games which show how the clock can be used. In Listing 11, after a random pause of between 2.5 and 10 seconds, a cross is PLOTted in the middle of the screen, and the computer PINGS. Line 90 is interesting. This is included to prevent cheating by pressing a key before the cross appears. Line 100 sets the clock to 10000, which gives a maximum possible reaction time of 100 seconds, which should be enough! Line 110 PLOTs the cross, and line 120 waits for any key to be pressed.

```
5 REM Listing 11
10 REM Reaction Test
20 REM J.W.P. 5-83
30 CLS:PLOT 10,10,"Reaction Test Game"
40 WAIT 500
50 CLS
60 PLOT 11,5,"Press any key"
70 PLOT 9,7,"when cross appears"
80 WAIT RND(1)*750+250
90 FLUSH$=KEY$
100 DOKE 630,10000:PING
110 PLOT 18,12,"+"
120 GET A$
```

```
130 T=DEEK(630)
140 RT=(10000-T)/100
150 R$="You took "+STR$(RT)+CHR$(0)+"seconds"
160 PLOT 9,18,R$
170 PRINT"Press any key to continue"
180 GET RE$:GOTO 50
```

When a key is pressed, line 140 subtracts the current reading of the clock from the value it was originally set to, and divides this by 100 to give the time in seconds. Lines 150 and 160 print out the result, and the program then recommences.

Listing 12 works on the same principle, but requires a specific key, selected at random in line 60, to be pressed.

```
5 REM Listing 12
10 CLS
15 PRINT:PRINT:PRINT
20 PRINT TAB(20)"Reaction test"
25 PRINT:PRINT:PRINT
30 PRINT TAB(20)"Press this key:-"
40 PRINT:PRINT:PRINT
50 WAIT RND(1)*500+100
60 A$=CHR$(RND(1)*26+65)
70 PRINT TAB(30)A$
80 DOKE 630,10000
85 FLUSH$=KEY$
90 GET B$
100 T=DEEK(630)
105 PRINT:PRINT:PRINT
110 RT=(10000-T)/100
120 IF A$=B$ THEN PRINT TAB(20)"Correct in ";RT;" seconds":WAIT 500
:GOTO 10
130 PRINT TAB(20)"You took all of ";RT;" seconds..."
140 WAIT 200
150 PRINT:PRINT TAB(20)"....then you got it wrong!"
160 WAIT 500:GOTO 10
```

Chapter 5

USING THE SOUND GENERATOR

One of the ORIC's strongest points is its sound generator which enables both music and complex sound effects to be readily produced. It does not have a sound generator and control system to quite match some of the more expensive machines, but it is much better than most, and is capable of producing any sounds that it could reasonably be expected to generate. Computer sound generators can be rather difficult to master, particularly if you are not familiar with sound synthesis and electronic music techniques. However, it is well worthwhile spending some time learning to use the sound system since sound effects and music can greatly enhance programs, particularly games programs.

Pre-programmed Sounds

The pre-programmed sound effects are a very useful feature of the ORIC and these enable sounds to be added to programs very easily indeed. The PING command produces a fairly high pitched bell-like sound, SHOOT gives a short explosive sound like a gunshot, EXPLODE gives a longer sound like a large explosion, and ZAP provides a brief falling pitch sound. You can hear these for yourself simply by typing the appropriate word into the computer and hitting RETURN.

These commands can be used alone as program lines, or they can be used as part of a larger program line (e.g. IF X = 3 THEN EXPLODE). The price that is paid for the ease with which these commands can be used is that they cannot be varied in any way, and it is not possible to shorten or lengthen the sounds for example, or even to alter the volume level.

SOUND

There are three commands that can be used to produce your own sound effects, or to add music to a program. These are

SOUND, MUSIC, and PLAY. Although SOUND can be used to produce music, and MUSIC can be used to generate sound effects, in general it is much easier to use the MUSIC command when music is required and the SOUND command for the production of sound effects. In this book we will only consider the use of these two commands in their principle roles. The PLAY command is used in conjunction with SOUND and MUSIC to give envelope shaping (to vary the volume of the sound to give the desired effect), or simply to stop the sound generator from operating after the appropriate length of time. The PLAY command does not in itself produce sound, but there would normally be at least one PLAY command used with each SOUND or MUSIC command, or series of commands.

Each time SOUND is used it must be followed by three numbers which define three of the four main parameters of the sound output. The first selects the required sound channel, and there are three channels which produce tones (we will ignore the noise channel for the time being). It is possible to use all three channels simultaneously, but for the time being we will only consider the use of a single channel. The three channels are simply numbered 1, 2, and 3, and when using a single channel it is channel 1 that must be used.

The second number defines what the ORIC manual calls the period of the signal, but this means the period of each cycle of the signal not the duration of the whole sound. In other words it is the frequency or pitch of the sound that this figure determines. Strictly speaking it is the wavelength of the note that this number sets, and the practical importance of this is that the larger this number is made, the lower (*not higher*) the pitch of the tone. Basically what happens in the sound generator is that a tone generator of fixed frequency is divided by an amount equal to the second number used in the sound command, and a doubling of this number therefore gives a halving of pitch.

The ORIC manual does not specify the maximum number that can be used to define period, but the 8912 sound generator used in the machine has 12 bit tone period registers which take a maximum number of 4095, and experiments seem to verify that this is the largest number that can be used. In fact the program does not crash if a larger number is used (up to 65536 anyway), but the sound generator will only respond to the lower 12 bits of the number, and 4095 thus gives the lowest achievable pitch.

A very wide tone range can be obtained using the SOUND command, from a sub-audio signal to an ultrasonic one. A sub-audio pitch will give an audible output because strong harmonics (multiples) of the fundamental frequency will be present, and these will be at audible frequencies. The same is not true if an ultrasonic pitch is used because the fundamental and harmonics will all be too high in pitch to be heard, and there will be no apparent output from the unit!

The third number sets the volume of the signal, and this must be between 1 (minimum volume) and 15 (maximum volume). If 0 is used here the volume is controlled by the PLAY command, but we will not consider this in detail for the time being.

An obvious omission from the sound generator commands is some means of controlling the length of each sound. This parameter is in fact controlled using the WAIT command, which is not just for use with the sound generator, and can be used to put a period of inactivity in any program.

The WAIT command is one of the easiest to understand, and it simply stops the program from progressing any further for a length of time that depends on the number which follows the word WAIT. The delay is in one hundredths of a second, and WAIT 50, for example, would hold up the program for 0.5 seconds. An important point to bear in mind when using this command is that apart from giving an output from the sound generator and producing the TV display the computer can do nothing else while it is executing a WAIT command.

The sound generator is switched off using the command:

```
PLAY 0,0,0,0
```

The following program will give a one second burst of tone:

```
10 SOUND 1,100,12
20 WAIT 100
30 PLAY 0,0,0,0
40 STOP
```

Try running this, and then alter the period, volume, and WAIT figures to see what effect this has on the sound that is produced. A little experimentation of this type will help to familiarise you with the actual sounds that are produced by various parameters in the SOUND and WAIT commands. Note that if you use 2 or 3 as the sound channel in line 10 no output will be obtained, and channel 1 must be used in a simple program such as this. Whichever of the three tone channels is selected there should be no discernable difference in the sound output that is obtained.

If you wish to change the note it is not necessary to use the PLAY 0,0,0,0 command to switch off the sound generator before using the new SOUND command. In fact it is not advisable to do so since this seems to produce a noticeable click during the changeover from one note to another. It is much better just to use two (or more) SOUND and WAIT commands followed by PLAY 0,0,0,0 at the end of the sequence. The following program demonstrates this point, and it simply gives an initial note, a lower note, and then the original note again with all three notes 0.75 seconds long.

```
10 SOUND 1,100,12
20 WAIT 75
30 SOUND 1,200,12
40 WAIT 75
50 SOUND 1,100,12
60 WAIT 75
70 PLAY 0,0,0,0
80 STOP
```

Although the computer cannot carry out commands while it is executing a WAIT cycle, it is quite possible to use commands between the SOUND and WAIT lines so that something (such as a change in the display) coincides with each change in pitch. For instance, adding the following lines to the program above

```
  15 PRINT "NOTE 1"
  35 PRINT "NOTE 2"
  55 PRINT "NOTE 3"
```

should result in NOTE 1, NOTE 2, or NOTE 3 being printed on the screen at the start of the appropriate note from the sound generator. An important point to bear in mind when doing this is that it takes time for the additional commands to be executed by the computer, and this effectively extends the WAIT statements. With a simple instruction such as a short PRINT command this is unlikely to be significant, but it might sometimes be necessary to shorten the WAIT period in order to obtain the desired effect, or in an extreme case it might even be necessary to remove it completely!

If you try using 4,5 or 6 as the sound channel number in the SOUND instruction you should find that the effect is just the same as using the tone channels, and it only seems to be possible to obtain a noise output in conjunction with the PLAY command. Similarly if you try adding several SOUND instructions ahead of the WAIT command (with each SOUND instruction using a different channel number) only the first SOUND command will be executed. It is only possible to obtain multichannel operation from the SOUND instruction if it is accompanied by a suitable PLAY command, as we shall see shortly.

In fact quite good effects can be obtained using a single channel, and it is by no means essential to use several channels in order to obtain good results. However, a single sound command, or just a few strung together, are capable of giving only a rather limited range of relatively uninteresting sounds that are probably of less use than the four preprogrammed sounds.

In order to obtain really good sound effects it is necessary to vary the volume and (or) pitch of the tone in a way that would require an impractically large number of SOUND instructions. Fortunately, there is an easy solution to the problem, and this is to use a numeric variable for period and (or) volume in the SOUND command, and to use a loop so that the values of the variable (or variables) can be changed in some way. Perhaps a simple sound effect program is the best way of explaining this. The program given below gives a bomb drop sound effect with a falling pitch sound followed by an explosion sound (the latter being the preprogrammed EXPLODE command and not part of the SOUND instruction).

```
10 LET PERIOD = 25
20 SOUND 1,PERIOD,12
30 LET PERIOD = PERIOD + 1
40 IF PERIOD = 150 THEN EXPLODE ELSE GOTO 20
50 STOP
```

The first line sets the initial value of variable PERIOD at 25 which gives a suitably high starting pitch. The next line is the sound instruction and this sets period as variable PERIOD, and sets the volume at 12 (or whatever figure you wish to use). PERIOD is simply incremented by one in line 30, and line 40 checks whether PERIOD has reached a value of 150 (which gives a suitably low finishing note). If it has not, the program is looped back to line 20 where the pitch of the tone generator is reduced. When PERIOD does reach 150 the EXPLODE command is carried out and the program is stopped at line 50. There is no need to include a PLAY 0,0,0,0 program line to halt the output from the sound generator because the EXPLODE command takes over from the sound instruction, and in the normal way it switches off the sound generator once the command has been completed.

PLAY

In order to synthesise most sounds reasonably well it is essential to vary the volume in the appropriate manner. Rather than using a numeric variable for the volume figure in the SOUND command it is easier to use one of the seven preprogrammed envelopes available from the PLAY instruction. However, it is useful to bear in mind that it is possible to use a variable for the volume figure if you wish to do so, perhaps because the PLAY command does not provide a suitable envelope shape for your requirements.

When using the PLAY instruction in conjunction with one or more SOUND commands it is essential to use 0 for the volume level in the SOUND command or commands, since this switches control of the volume level to the PLAY instruction that follows. This command uses four numbers which select any tone channels that are required, the noise channel (if required), the desired envelope shape, and the duration of the envelope.

Taking the tone enable figure first, this is from 0 to 7 with 0 being used to cut off the tone channels (if only noise is to be used). Assuming that channel 1 will be used where only a single tone is required, and channels 1 and 2 will be employed if two channels are needed, the numbers to use are 1 and 3 respectively. All three channels are selected using number 7. The ORIC manual gives the full list, but normally only the four modes mentioned above will be needed.

The same system of numbering is used to select which channel or channels the noise signal is mixed into. It is important to note that there is only one noise source, not three. Assuming that tone channel 1 will always be in use, it is only necessary to use 1 as the noise enable figure. This is also suitable if only the noise (and no tone) is to be used. 0 is used to suppress the noise generator.

The envelope modes are shown diagramatically in the ORIC manual, and are all fairly straight forward. Mode 1 gives an almost instant rise to full volume followed by a much slower fall in volume until the signal becomes inaudible. Many natural sounds are of this basic type, including many string and percussive instruments, explosions, etc. Mode 2 is the inverse of this with a slow build up and almost instant fall to zero. This gives a rather weird and unnatural effect.

Mode 3 is similar to mode 1, but the envelope automatically repeats itself. Mode 6 is the repeating version of mode 2.

In mode 4 the volume is smoothly varied up and down, and this is another repeating envelope. Modes 5 and 7 are similar, with the volume level gradually building up to maximum and staying there until a new note is started or the sound output is halted by a PLAY 0,0,0,0 command. The difference between the two is that in mode 7 there is a straight forward rise in volume until the maximum level is reached, but in mode 5 the volume starts at maximum, steadily falls to zero first, and then builds up to maximum. The envelope diagram for mode 5 in the ORIC manual seems to be slightly in error incidentally (in the early versions anyway). There also seems to be an error in the diagram for mode 4 where it shows the sound level starting at minimum, whereas it does in fact start at maximum volume. As mode 4 gives a repeating envelope this will not often be of any practical importance though.

The fourth figure in the PLAY instruction controls the duration of the envelope. Where the envelope is a repeating type it is the duration of a single rise and fall in volume that is controlled by this figure. An important point to bear in mind is that a WAIT command must be used after the PLAY instruction to prevent the next SOUND and PLAY commands from being reached too quickly. In some cases it is necessary to have the envelope period figure and the WAIT time matched reasonably accurately in order to obtain the proper envelope shape.

As an example of this, let us assume that envelope mode 1 is to be used, and that the WAIT command will give a note duration of one second using WAIT 100. If the envelope period is made too long the sound will be cut off abruptly before it has had time to naturally drop to zero. If the envelope period is made too short the sound will fall to zero before the WAIT period ends and the next note is produced.

For modes 1 and 2 the figure for the envelope period should be 20 times larger than the figure used in the WAIT command. In our example this would obviously give an envelope period figure of 2000. The maximum figure that can be used for the envelope period is 65535 (and not 32767 as stated in the manual). Of course, you can purposely mismatch the WAIT and envelope period numbers if this gives the effect you require.

With the other envelope modes the question of matching these two figures does not normally arise. With the repeating modes the envelope period would normally be made quite short in relation to the WAIT time so that a fairly large number of repeats would be produced. Similarly, modes 5 and 7 would normally be used with quite short envelope periods when compared to the length of the note set by the WAIT instruction. However, in modes 3 and 6 the length of one envelope cycle (i.e. a single rise and fall in volume) is the same as for modes 1 and 2, and the rise in volume of mode 7 is also on the same time scale. Note though, that in modes 4 and 5 (presumably because both the attack and decay times are long) a given envelope period figure gives double the envelope duration when compared with the other modes.

If we now take a simple example sound effect using the SOUND and PLAY instructions, the program given below produces a burst of tone which has a fast attack and slow decay (mode 1), and is similar in nature to the preprogrammed PING command.

```
10 SOUND 1,25,0
20 PLAY 1,0,1,2000
30 WAIT 100
40 STOP
```

The program is very simple in operation with line 10 selecting channel 1, a fairly high pitch, and giving control of the volume to the PLAY instruction that follows in line 20. This enables tone channel 1, disables the noise generator, selects envelope mode 1, and matches the envelope period to the following WAIT command (i.e. uses a figure 20 times larger than that in the WAIT instruction). Line 30 and the envelope period give a note duration of one second.

An advantage of this program over the PING command is that the pitch of the sound can be changed by using a different pitch value in the sound command. You might like to try using envelope mode 2 and hear the way the change in envelope shape totally changes the character of the sound.

A more bell-like sound can be produced by adding a second sound channel, as shown in the program that follows.

```
10 SOUND 1,25,0
20 SOUND 2,37,0
30 PLAY 3,0,1,2000
40 WAIT 100
50 STOP
```

Here channel 2, with a somewhat lower pitch than channel 1, has been added at line 20, and the PLAY command at line 30 has been modified to enable tone channels 1 and 2. All three channels can be used in the manner shown below:

```
10 SOUND 1,25,0
20 SOUND 2,37,0
30 SOUND 3,75,0
40 PLAY 7,0,1,2000
50 WAIT 100
60 STOP
```

All that has been done here is that channel 3 has been enabled and the pitch selected at line 30, and the tone enable figure in the PLAY instruction at line 40 has been changed to 7 to enable all three channels.

The program listed below uses a repeating envelope (mode 4) and demonstrates how this can be used to produce an (amplitude) modulated tone.

```
10 SOUND 1,100,0
20 PLAY 1,0,4,50
30 WAIT 300
40 PLAY 0,0,0,0
50 STOP
```

The modulation frequency is set by the figure for the envelope period in line 20, and you might like to experiment with values other than 50. Remember that lower figures give a higher modulation frequency. You might also like to try out the other repeating envelope modes to discover the different sounds that can be produced. It is only really possible to learn how to get the effects you require by experimenting with the sound generator and becoming familiar with the basic sounds that are available to you. The programs in the sound section of the ORIC manual are also helpful in this respect.

It is possible to have both pitch and amplitude modulation, and the program shown below demonstrates a simple way of achieving this.

```
10 LET PERIOD = 50
20 SOUND 1,PERIOD,0
30 PLAY 1,0,1,150
40 LET PERIOD = PERIOD + 5
50 IF PERIOD = 200 THEN PLAY 0,0,0,0 ELSE GOTO 20
60 STOP
```

This is a simple loop similar to that used in the falling pitch program described earlier, but in this case envelope mode 1 is used so that the sound level falls until the program loops back to the SOUND and PLAY commands again. The volume is then restored to maximum, but immediately starts to fall until the program loops back to lines 20 and 30 again, when full volume is then restored. Thus, although a non-repeating envelope mode is used, the looping action of the program gives a repeating action. The frequency of the amplitude modulation is controlled by the time it takes the computer to complete each loop, but a WAIT instruction could be included to slow things down slightly and give a lower modulation frequency if desired. However, the fact that the pitch reduces in steps rather than continuously would become much more apparent, although this could still give an interesting effect.

Noise Channel

The noise channel is primarily of use for producing explosive sounds such as the preprogrammed EXPLODE and SHOOT sounds. The program below gives a sort of machine gun effect by using the noise generator in conjunction with one of the repeating envelope modes.

```
10 SOUND 4,3,0
20 PLAY 0,1,3,200
30 WAIT 300
40 PLAY 0,0,0,0
50 STOP
```

In line 10 channel 4 is used to enable the noise generator, the pitch value of 3 selects quite high frequency noise (a high pitched hissing sound), and 0 for the volume level hands control of the volume to the subsequent PLAY instruction in the normal way. Something which must be kept in mind when using the noise generator is that the number used to select the pitch is between 0 and 31 because the 8912 sound generator chip has only a 5 bit noise period register. The highest pitch is produced using a pitch figure of 0 and 31 gives the lowest pitch. This obviously gives far fewer pitch options than when using the tone generators, which can have pitch values of up to 4095. The nature of noise signals is such that a range of 32 pitches is quite satisfactory in practice.

In the PLAY command at line 20 zero is used as the first parameter so that all the tone channels are disabled. 1 is used for the next parameter to enable the noise generator, and envelope mode 3 is used as this gives a fast attack, like the sound produced by a gun. A figure of 200 for the envelope period gives a realistic modulation frequency (or fire rate if you prefer), but you can try changing this value to speed up or slow down the rate of fire if you wish.

Most effects only require the use of noise or the tone channels, but it is possible to use both at once, and this can sometimes give quite effective results. The program listed below gives a sound rather like engine noise.

```
10 SOUND 1,1000,0
20 SOUND 4,10,0
30 PLAY 1,1,3,50
40 WAIT 300
50 PLAY 0,0,0,0
60 STOP
```

The first two lines select suitably low tone and noise pitches to give a reasonable simulation of engine noise. In the PLAY command at line 30 tone channel 1 is enabled and so is the noise generator. Envelope mode 3 is used and an envelope figure of 50 produces the fairly fast modulation rate that this sound effect demands. As with the previous example programs, it is a good idea to try varying some of the parameters to discover what effect this has, and to see if you can improve on the effect. You might also like to try adding in tone channels 2 and 3 by adding extra SOUND instructions and altering the PLAY command at line 30.

One final point about the SOUND and PLAY commands is that it is not essential to use envelope shaping if you are using more than one channel. By using in the PLAY instruction the appropriate tone enable and noise enable figures for the channels used in the SOUND commands, zero can be used for both the envelope mode and envelope period figures. The sound then simply switches on and off with virtually instant attack and decay. This is demonstrated in this simple program:

```
10 SOUND 1,100,12
20 SOUND 2,150, 8
30 SOUND 3,200,6
40 PLAY 7,0,0,0
50 WAIT 300
60 PLAY 0,0,0,0
70 STOP
```

In actual fact this is a more versatile way of generating sounds since the volume of each channel can be different (the volume levels being set by the final figure in the SOUND command or commands). By stringing together sets of SOUND and PLAY instructions it would be possible to vary the volume of each channel and thus produce any desired envelope shaping, and the pitches of the tones could be varied as well. In this way a vast number of complex effects could be generated, but even with this system streamlined and arranged to be as quick and simple to use as possible, it would still be a rather long and awkward way of doing things, and would probably not be a practical proposition. For most purposes the preprogrammed envelopes will give more than adequate results.

MUSIC

The MUSIC command is used in conjunction with the PLAY command in exactly the same way as the SOUND instruction.

The only difference between the two is that the MUSIC command has been designed to make it simple to generate correct musical notes, and a seven octave range (including all semitones) can be covered.

There are four parameters which must be included in each MUSIC instruction, and these are channel, octave, note, and volume. Channel and volume are used in exactly the same way as their counterparts in the SOUND command (including the use of 0 as the volume figure if envelope shaping is to be provided by the PLAY instruction).

Selecting the desired note simply entails specifying the required octave using a number of between 0 and 6 (0 is the lowest octave — 6 is the highest), and then using the appropriate number from 1 to 12 for the required note within that octave. A list of notes and their corresponding numbers is given below:

A	10
A#	11
B	12
C	1
C#	2
D	3
D#	4
E	5
F	6
F#	7
G	8
G#	9

Incidentally, middle C is the C in octave 3.

Tunes can be played by simply stringing together sets of MUSIC and PLAY instructions, and this gives optimum versatility, but it is much easier to use a DATA statement or statements in the manner shown in listing 13.

```
5 REM Listing 13

10 FOR N=1 TO 8

20 READ A,B,C

30 MUSIC 1,A,B,0
```

```
40 MUSIC 2,A-1,B,0

45 MUSIC 3,A+1,B,0

50 PLAY 7,0,1,700

60 WAIT C

65 NEXT N

70 PLAY 0,0,0,0

80 DATA3,1,25,3,3,25,3,5,25,3,6,25

90 DATA3,8,25,3,10,25,3,12,25,4,1,100
```

This is quite straight forward with three parameters in the DATA statements, A is the octave, B is the note, and C is the length of the note. The sample figures in the DATA statements give a simple scale of C major, but obviously any desired tune could be produced using this method, and any note within the compass of the MUSIC command can be obtained. All the data could in fact be placed in a single DATA instruction in this instance, and it has been placed in two lines simply to demonstrate that long sets of data can be accommodated in this way. The only limitation on the number of DATA lines that can be used are those imposed by the available memory space, and very long sequences of notes could normally be used if desired.

All three channels are used, but channels 2 and 3 play the same note as channel 1. However, they are set an octave lower and an octave higher respectively, and this is a very simple but effective way of obtaining a more interesting and musical output from the sound generator. An important point to bear in mind is that lines 65 and 70 could be transposed, but the sound generator would then be switched off after each note. This is undesirable as it would produce a click sound after each note; something that is avoided by having these instructions in the order shown in listing 13.

If a program of this type is used as a subroutine it will only operate properly the first time it is used (see the section on character definition in chapter 4). The alternative is to READ the data into an array, and a simple example of this is given in one of the programs in a subsequent chapter.

It would not normally be necessary to mix noise and music, although an attempt to simulate a cymbal or similar sounds using the noise generator could obviously be made. The noise generator cannot be controlled using the MUSIC command, but there seems to be no problems if SOUND and MUSIC commands are mixed and used in conjunction with the same PLAY command. It is therefore possible to use MUSIC commands to generate musical notes and to use a SOUND command to add in noise and control the pitch of the noise.

Chapter 6

DECISIONS

When ORIC is asked to make a decision – either in an IF. . .THEN statement or at the UNTIL part of a REPEAT. . . UNTIL loop, it evalutes whatever comes after the IF or UNTIL to determine whether it is TRUE or FALSE. TRUE and FALSE are in fact BASIC functions, returning values of –1 and 0 respectively. We can prove that TRUE and FALSE have numeric values by the following lines which can be typed into the computer in command mode.

```
PRINT TRUE
-1
PRINT 3=3
-1
A=6:PRINT A=6
-1
PRINT FALSE
0
PRINT 3=7
0
A=6:PRINTA=7
0
```

In fact, any variable with the value –1 can be regarded as TRUE, as in this example.

```
A=-1:IF A THEN PRINT "YES"
YES
```

On the face of it, you would expect IF A THEN. . . to produce a syntax error message, but in fact the IF. . .THEN statement simply evaluates whatever follows the IF, and if it has a numeric value the syntax requirements are satisfied.

If the statement following IF is TRUE, then the computer executes everything following THEN. If the IF statement is FALSE the computer ignores the statement following THEN and goes on to the next line. An extension to the IF. . .THEN structure is the statement ELSE. When ELSE is used (and its use is optional) if the statement following IF is TRUE everything after THEN is executed, and everything after ELSE is ignored. If the statement is FALSE, everything after ELSE is executed, and the instructions after THEN ignored.

With REPEAT. . .UNTIL loops, the computer will branch out of the loop when the statement after UNTIL is TRUE. You can now understand how we produce an infinite loop by the statement UNTIL FALSE. FALSE can never be TRUE, so the loop goes on for ever. UNTIL FALSE is a standard way of producing an infinite loop for short investigatory or demonstration loops, like the first two programs in this book.

It follows that a loop ending with UNTIL TRUE would only execute once (remember REPEAT. . .UNTIL loops always execute at least once).

It is possible to test for more than one condition using the logical operators AND and OR. AND requires all the set conditions to be TRUE. OR requires one (or more) of the conditions to be true. AND and OR can be used together, as in this example:

```
100 IF A=6 AND B=7 OR C=D THEN F=F+1
```

However, when using lines like this, it must be realised that as far as the computer is concerned, IF is one long expression, and the rules of operator precedence must be borne in mind if the intended result is to be produced. AND and OR are low-priority operators, and will always be executed last, and AND has precedence over OR.

In this example A=6, B=7 and C=D would be evaluated first, to return either 0 or −1. Then the results of A=6 and B=7 would be ANDed and if both were TRUE this would return −1, and then this would be ORed with the result of C=D, and if either of these were −1, the whole expression would be TRUE, and the variable F would be increased by 1.

If in doubt about which parts of a complex expression will be evaluated first, remember that it does no harm to enclose the parts that must be evaluated first in brackets, even if the brackets are not in fact necessary.

It can be seen from the above explanation that the logical and bit-by-bit operations of AND and OR in fact amount to the same thing (if you do not understand AND and OR as applied to binary arithmetic, an explanation will be found in the chapter on interfacing).

ORIC also has the logic operator NOT. This reverses the result of a test. For example we could use UNTIL NOT TRUE instead of UNTIL FALSE. NOT is not the most useful of operators, but when needed it is usually indispensible. It is most often used in conjunction with functions. For example, the function SGN(X) will return −1 if X is negative, 0 if X is zero and +1 if X is positive. We cannot use a line like IF SGN(X) THEN PRINT "NEGATIVE" because in fact all non-zero values are regarded as TRUE, but it is possible to use the line IF NOT SGN(X) THEN PRINT "POSITIVE".

So far we have only considered testing numeric variables for equality, but we can also use the tests less than, greater than, less than or equal to, greater than or equal to, and not equal to, and by using NOT we can also test for not less than and not greater than.

We can also apply tests to string variables. When string variables are tested, the ASCII codes are compared on a character-by-character basis. In effect, this means that words can be tested for alphabetical order.

Often, we will want several things to happen in response to an IF. . .THEN statement. Sometimes the required results can be obtained by using a multiple statement line, but ORICs' maximum line length of 78 characters is restricting. Long multiple statement lines can also be hard to follow. An alternative is to use the IF. . .THEN statement to skip over the lines containing the things we don't want to happen. When doing this it is not necessary to use THEN and GOTO. One or the other will suffice, and most programmers omit the GOTO.

However, when we want a complex series of events to follow an IF. . .THEN statement, an altogether more elegant way of doing things is to use subroutines, and these will be explored in the next chapter.

The program in this chapter (listing 14) is a guess-the-number game that relies almost entirely on IF. . .THEN statements. The loop in lines 80—100 generates a random number.

This loop method was found to be necessary as, from a cold start, the computer always generated the same set of random numbers. It isn't supposed to! Lines 110 to 170 test the size of the number and drop hints as to what it might be. A problem is that there is a bias towards large numbers, so it nearly always says the number might be someone's telephone number! You might like to add a few clues of your own for the high numbers to get round this.

```
5 REM Listing 14
10 REM Guess the number game
20 REM IF...THEN demonstration
30 CLS:TRIES=1:ODFF=10000:PAPER 2
40 PRINT:PRINT:PRINT
50 PRINT "I am thinking of a number..."
60 PRINT:PRINT"...it is between 1 and 10,000...":PRINT
70 PRINT CHR$(129);"Press any key to play."
80 REPEAT
90 NUMBER=INT(RND(1)*10000)
100 UNTIL KEY$<>""
110 PRINT:PRINT"Here is a clue...":PRINT
120 IF NUMBER<100 THEN PRINT"It could be someone's age..."
130 IF NUMBER>100 AND NUMBER<400 THEN PRINT"It could be a cricket s
core..."
140 IF NUMBER>300ANDNUMBER<1000 THENPRINT"It could be the pages in
a book..."
150 IF NUMBER>1000 AND NUMBER<3000 THENPRINT"It could be the size o
f a car engine"
160 IF NUMBER>3000 THEN PRINT"It could be someone's telephone numbe
r..."
170 IF NUMBER<50 OR NUMBER>9550 THEN PRINT"I am near my limit..."
180 PRINT:PRINT "Try no. ";TRIES:PRINT
190 PRINT:INPUT "Guess the number";GUESS
200 PRINT
210 IF GUESS<0 OR GUESS>10000 THEN PING:PRINT "SILLY!":GOTO 190
220 IF GUESS=NUMBER THEN 340 ELSE DFF=ABS(NUMBER-GUESS)
```

```
230 PRINT
240 IF NUMBER<GUESS THEN PRINT "Too big ";
250 IF NUMBER>GUESS THEN PRINT "Too small ";
260 IF DFF<10THENPRINT "but you are very close"
270 IF DFF>10 AND DFF<50THEN PRINT"butyou are quite close"
280 IF DFF>50 AND DFF<200 THEN PRINT". :PRINT
290 IF DFF>200 THEN PRINT"by a long chalk!"
300 IF NOT SGN(DFF-ODFF)THEN PRINT:PRINT"You are further out than l
ast time!"
310 TRIES=TRIES+1:ODFF=DFF
320 PRINT:PRINT CHR$(129);"Press any key to try again"
330 GET A$:CLS:PRINT:PRINT:PRINT:PRINT "Last guess ";GUESS:PRINT:GO
TO 170
340 PRINT CHR$(142);CHR$(148);CHR$(129);"CORRECT! ";CHR$(146)
350 PRINT CHR$(142);CHR$(148);CHR$(129);"CORRECT! ";CHR$(146)
360 PRINT:PRINT:PRINT
370 PRINT CHR$(129);"Press any key to try again"
380 GET RE$:RUN
```

The guess is input at line 200. Line 210 is a garbage filter for out-of-range input. If the guess is correct line 220 sends the program to the win sequence at the end (lines 340 onward). If the guess is not correct the rest of this line after ELSE calculates the size of the error.

The rest of the program drops hints as to what the next guess should be, depending on the size and direction of the error. Note the effect of the semi-colons at the end of the lines 240 and 250.

Line 300 is interesting. It uses the function SGN and the operator NOT to indicate whether the error on the current guess is greater than the error on the last guess. A logic problem here is that it regards an equal error as being greater.

Some of the lines in this program push ORIC's acceptable line length to the limit. In some cases, it is only possible to type them in by using ? in place of PRINT. The word appears in full on subsequent listings. It may also be helpful to remember that as far as ORIC is concerned, the variable names NU and NUMBER are the same.

Chapter 7

STRUCTURED PROGRAMING

So far, the programs in this book have been fairly simple, so it has been easy to write them in a straight-through fashion, starting at the beginning and running through to the end, sometimes with a line at the end to send the program back to some point near the beginning, avoiding the necessity of continually retyping RUN.

As programs become longer and more complex, it becomes more difficult to write them in this fashion. In particular, matters become difficult when we want a number of things to happen following an IF. . .THEN statement. There will also be some events which are required to occur several times in the run of a program, and it is wasteful of time and memory to have to write the same program lines in several places.

The answer to these problems is the subroutine. A subroutine is a section of a program designed to do a specific task in the program, and which can be called by the main program whenever the particular task has to be performed.

A subroutine is called by the statement GOSUB N where N is the first line number of the subroutine. At the end of the subroutine, the statement RETURN is used. This returns the program to the statement following the GOSUB. It is very bad practice to end a subroutine with a GOTO. RETURN should always be used. If necessary, the statement after the GOSUB can be a GOTO — on the same line if necessary.

It is perfectly permissible to call a subroutine from within another subroutine. This is called nesting. Nesting to a maximum depth of 16 is possible with ORIC, though it is doubtful if this would often be necessary or desirable.

Writing programs as a series of subroutines is known as structured programming. In fact, the loop structures FOR. . . NEXT and REPEAT. . .UNTIL are also program structures, as are user-defined functions. It would be possible to create a loop by using IF. . .THEN. . .GOTO at the end to send the program back to the first line of the loop, but I am sure you would agree that it is easier to use (and easier to follow) listings using the special structures provided.

Structured programming has become something of a sacred cow, especially among educationalists. With this has grown the idea that since it is good for you, it must taste nasty! Structured programming is held to be desirable, but difficult. This is most certainly *not* the opinion of the author! I personally find it much easier to conceive and write a long program as a series of subroutines than as one long stream of lines, and there can be no denying that a structured program is easier to modify and extend. When the program does not run as intended, it is also normally easier to track down and rectify the faults.

Structured programs can be easier to follow than unstructured ones, and we can make them easier to follow by *intelligent* use of REMS. I stress intelligent because I have seen very few programs where the REMS were really of any help. Too often, programs are littered with lines like 220 INPUT NAME$: REM gets users name, but with nothing to explain what, for example, a complex string-manipulating routine is meant to achieve.

It is always a good idea to put a REM on a line that calls a subroutine, to say what the subroutine does. It is also a good idea to make the first line of a subroutine a REM, again saying what the routine achieves. The more structured a program is, the more important this becomes, as the main section of a full-structured program may consist entirely of GOSUBS, in conjunction with IF. . .THENs.

Just how far programs should be structured is a matter of conjecture. The purists hold that every individual task should be a separate routine, even if it is only executed once in a program. This is really a counsel of pedantry, and can lead to programs that are over-long and slow-running. However, any task, however trivial, that takes more than two or three lines and is performed more than once in a program, can usefully be made a separate subroutine.

There are some things, such as dimensioning arrays and defining functions, that should not really be done within subroutines. These jobs should be done, preferably, at the very beginning of a program, immediately after the programmers' self-indulgent title!

It is important that a subroutine should never be executed except by being called by a GOSUB. If this should happen, a RETURN WITHOUT GOSUB error occurs when the computer comes to the RETURN statement. Subroutines are normally placed after the main program, with high line numbers, and the main program terminated with an END statement, so that this cannot occur.

Listing 15 is a fairly complex game program, written in a highly structured form.

```
5 REM Listing 15
10 REM**************
20 REM*            *
30 REM* UGLICHASE *
40 REM*            *
50 REM* J.W.P.'83 *
60 REM*            *
70 REM**************
80 DOKE#400,0
90 IF (PEEK(618)AND1)=1 THEN PRINT CHR$(17)
100 IF (PEEK(618)AND8)=0 THEN PRINT CHR$(6)
110 IF (PEEK(524)AND128)=128 THEN PRINT CHR$(20)
120 CLS:LORES 0:INK 0
130 DIM UG(4,1)
140 DIM TUNE(7,2)
150 GOSUB 1000:REM Character definitions
160 GOSUB 2000:REM Fill music array
170 GOSUB 3000:REM Background fill string
180 GOSUB 4000:REM Draws playfield
190 GOSUB 5000:REM Scoring
200 GOSUB 6000:REM Plots maze blocks
210 GOSUB 7000:REM Plots characters
220 GOSUB 8000:REM Moves man
230 GOSUB 9000:REM Moves uglis
240 IF E=1 THEN GOSUB 10000:GOTO 180
```

```
250 IF E=2 THEN GOSUB 11000:GOTO 180
260 GOTO 220
270 END

999 REM Character definitions
1000 FOR C=46416 TO 46431
1010 READ A
1020 POKE C,A
1030 NEXT C
1040 DATA 12,30,12,63,12,30,18,18
1050 DATA 18,12,30,51,45,12,18,51
1060 RETURN
1999 REM Fills music array
2000 FOR N=0 TO 7
2010 READ O,P,D
2020 TUNE(N,0)=O
2030 TUNE(N,1)=P
2040 TUNE(N,2)=D
2050 NEXT N
2060 DATA 3,10,30,3,10,15,3,10,15,3,10,15
2070 DATA 3,12,30,3,7,30,3,12,30,4,2,90
2080 RETURN
2999 REM Background fill string
3000 FOR C=1 TO 29
3010 F$=F$+CHR$(22)
3020 NEXT C
3030 RETURN
3999 REM Draws playfield
4000 FOR LX=4 TO 34
4010 PLOT LX,2,20
4020 PLOT LX,22,20
4030 NEXT LX
4040 FOR LY=3 TO 21
4050 PLOT 4,LY,20
```

```
4060 PLOT 34,LY,20
4070 NEXT LY
4080 FOR FY=3 TO 21
4090 PLOT 5,FY,F$
4100 NEXT FY
4110 RETURN
4999 REM Scoring
5000 S$=CHR$(2)+"SCORE"+STR$(PTS)
5010 IF PTS>DEEK(#400) THEN DOKE#400,PTS
5020 HS$=CHR$(2)+"HISCORE"+STR$(DEEK(#400))
5030 PLOT 3,1,S$
5040 PLOT 23,1,HS$
5050 RETURN
5999 REM Plots maze blocks
6000 FOR BLOCK=1 TO 75
6010 BX=RND(1)*28+5
6020 BY=RND(1)*19+3
6030 IF SCRN(BX,BY)=150 THEN 6010
6040 PLOT BX,BY,150
6050 NEXT BLOCK
6060 RETURN
6999 REM Plots characters
7000 X=20:Y=12
7010 PLOT X,Y,"*"
7020 FOR C=0 TO 4
7030 UG(C,0)=INT(RND(1)*26+6)
7040 IF (UG(C,0)>17)AND(UG(C,0)<23)THEN 7030
7050 UG(C,1)=INT(RND(1)*18+4)
7060 IF (UG(C,1)>9)AND(UG(C,1)<15)THEN 7050
7070 PLOT UG(C,0),UG(C,1),"+"
7080 NEXT C
7090 RETURN
7999 REM Moves man
8000 DOKE 630,INT(RND(1)*300)
```

```
8005 REPEAT
8010 M$=KEY$:IF M$=""THEN 8110
8020 M=ASC(M$)
8030 OX=X:OY=Y
8040 IF M=8 THEN X=X-1
8050 IF M=9 THEN X=X+1
8060 IF M=10 THEN Y=Y+1
8070 IF M=11 THEN Y=Y-1
8080 IF SCRN(X,Y)=150 THEN X=OX:Y=OY:SHOOT:PTS=PTS-5:GOTO 8110
8090 PLOT OX,OY,22
8100 PLOT X,Y,"*"
8110 GOSUB 5000
8120 UNTIL DEEK(630)>300
8130 RETURN
8999 REM Moves uglis
9000 FOR C=0 TO 4
9010 IF SCRN(UG(C,0),UG(C,1))=150 THEN9090
9020 PLOT UG(C,0),UG(C,1),22
9030 IF UG(C,0)>X THEN UG(C,0)=UG(C,0)-1
9040 IF UG(C,0)<X THEN UG(C,0)=UG(C,0)+1
9050 IF UG(C,1)>Y THEN UG(C,1)=UG(C,1)-1
9060 IF UG(C,1)<Y THEN UG(C,1)=UG(C,1)+1
9070 IF SCRN(UG(C,0),UG(C,1))=150 THEN ZAP:PTS=PTS+5:GOTO 9090
9080 PLOT UG(C,0),UG(C,1),"+"
9090 NEXT C
9100 REM next part checks win or lose
9110 FOR T=-1 TO 1
9120 FOR U=-1 TO 1
9130 IF SCRN(X+T,Y+U)=43 THEN E=1
9140 IF SCRN(X+T,Y+U)=20 THEN E=2
9150 NEXT U,T
9160 GOSUB 5000:REM Scoring
9170 RETURN
9200 FOR T=-1 TO 1
```

```
9999 REM losing sequence
10000 EXPLODE
10010 PTS=0:OPTS=0
10015 FLUSH$=KEY$
10020 PRINT CHR$(129);"PRESS ANY KEY"
10030 GET A$
10040 CLS
10070 LORES 0:INK 0
10080 E=0
10090 RETURN
10999 REM winning sequence
11000 GOSUB 12000
11010 PTS=PTS+20
11020 IF PTS-OPTS=45 THEN PTS=PTS+5
11030 GOSUB 5000
11035 FLUSH$=KEY$
11040 PRINT CHR$(129)"PRESS ANY KEY"
11050 GET A$
11060 OPTS=PTS:E=0
11070 CLS
11080 LORES 0:INK 0
11090 RETURN


11999 REM Plays tune
12000 FOR N=0 TO 7
12010 MUSIC 1,TUNE(N,0),TUNE(N,1),0
12020 MUSIC 2,TUNE(N,0)+1,TUNE(N,1),0
12030 MUSIC 3,TUNE(N,0)-2,TUNE(N,1),0
12040 PLAY 7,0,7,250
12050 WAIT TUNE(N,2)
12060 NEXT N
12070 PLAY 0,0,0,0
12080 RETURN
```

This may not be the most original of games, but it is entertaining and has a definite strategy to it.

The playfield has a blue border, and is scattered with random red blocks, on a cyan background. Your little man starts in the middle of the field, and you can move him around, one position at a time, using the cursor keys. There are also five Uglis in the field, and these will advance towards your man. The idea is to move so that the Uglis are attracted onto the red blocks. When this happens, they are zapped, and you score 5 points. Your man cannot move onto the red blocks. If you try to do so, you will hear a warning shot, and you lose 5 points. You score 20 points if you move adjacent to the blue border, and the round ends. If you zap all 5 Uglis and make it to the border without losing any points on the blocks, you gain 5 bonus points. However, if an Ugli manages to make it onto a position adjacent to your man, you will hear an explosion, and you lose the game!

Points from winning rounds are carried over, and the idea is to achieve as high a score as possible. A hiscore is also displayed, and is carried forward, win or lose, as long as the program is running. The authors' personal best at time of writing is 995. Yes, it was very frustrating!

If you examine this listing, you will see that, after the authors self-indulgent title, the early part consists of minor tasks, like turning off the cursor and CAPS message that make the screen untidy, that only need to be done once. It would not be worth writing these as a subroutine. Then comes dimensioning of two arrays, which should always be done early in the main program. The remainder of the main program (up to line 270) consists almost entirely of calling subroutines.

The first three routines are performed once only, but being fairly complex, are better written as separate routines rather than in the main program. All the remaining routines are executed several times. In particular, the scoring routine is called from several points within the program. Note that this routine is called both from the main program and from within other subroutines. There is one subroutine, the one that plays the tune, that is only called from within another routine. The DATA to play the tune is read into an array as the tune is played several times. This is difficult to achieve using the DATA statement direct.

Most of this program uses ideas we have already met, but the following points are of interest.

In the subroutine from line 1000 (note how the descriptive REM has been put on line 999 so it is never executed – small point but it saves time), by using two successive characters to redefine (the + and *) we can do the necessary POKEing with a single loop. The data for the two characters has, however, been put in two separate DATA statements.

The subroutine from line 3000 makes a long string of cyan background attributes which fill the playfield with an impressive sweep.

The subroutine from 4000 actually draws the playfield. Note how the top and bottom borders are drawn together, as are the two sides. The last part, 4080 to 4100 uses the string from the previous routine to fill the middle.

The scoring routine is really to print the scores on the screen. The calculations have to be done in several places, by the nature of the game. This routine does, however, update the hiscore as necessary. The hiscore value is stored in the spare memory area from #400 onward. This is a throwback to the early, less elegant prototype version of this program, where CLEAR had to be used after a losing round, so a variable could not be used to store the hiscore value. In the current version, an ordinary variable could be used instead, but this method has been retained to show how the spare memory can be used for things other than machine code.

The subroutine from 6000 plots the random blocks. Note that the value plotted is 150, which is actually the value for background cyan used with the PRINT statement. It is used here so that the blocks appear in red, but so that the square immediately to the right of a block does not turn red if a character moves on to it, as would occur if a value of 16 (normal PLOT background red) was used. Line 6030 is to prevent two blocks being PLOTted in the same place.

The subroutine from line 7000 onwards PLOTs the characters. The man is always started from the same point, but the Uglis are placed in random positions. Lines 7040 and 7060 are to prevent these starting positions being too close to the man.

The method of moving the man is the same as used in previous programs. Line 8000 sets a random time up to 3 seconds in which you have your turn. Line 8080 detects if you try to move onto a red square, and penalises you accordingly. Note that the character is rubbed out after each move with a background attribute rather than a space. Line 8120 ends the routine when the time is up.

The method of moving the Uglis is really quite simple, though it never fails to impress the uninitiated. The X and Y co-ordinates of each Ugli are checked in turn, and if not the same as the co-ordinates of the man, the difference is reduced by one. Line 9070 detects when an Ugli moves onto a red block, and zaps it! Line 9010 is to prevent it from subsequently re-appearing. Lines 9110 to 9150 check all the squares adjacent to the man character, and the one he is on, to detect either an Ugli or the blue border, terminating the round on either condition, via lines 240–250. You may feel that this should be a separate routine, so it could also be called after the moves man routine. You may care to see if you can do this yourself (but remember you will also have to put duplicates of lines 240–250 after the moves man routine has been called from the main program if it is to have any effect).

The remaining subroutines are the winning and losing sequences, and are fairly straightforward. Line 11020 adds the bonus points when earned.

The program is restarted from line 180, which calls the draws playfield subroutine. None of the earlier routines needs to be repeated. Indeed, an OUT OF DATA error message would occur if we tried to repeat the first two.

Chapter 8

DATA FILING IDEAS

After the fun and games of previous chapters, we now look at some of ORIC's more serious aspects.

The basis of this chapter is the program in listing 16, which enables the creation of a data file of telephone numbers and names. It has facilities to enter data, extend the file, display all the entries, search for and display a particular entry, and to search for and amend a particular entry. There is also a facility to make a cassette copy of the data, which can be loaded back for re-use.

```
5 REM Listing 16
10 REM ********************
20 REM *                  *
30 REM *     TELEFILE     *
40 REM *                  *
50 REM *   J.W.P. 5/83    *
60 REM *                  *
70 REM ********************
80 CLS:PAPER 3:INK 4
90 TEXT:GRAB
100 HIMEM #9D00
110 IF (PEEK(618)AND1)=1 THEN PRINT CHR$(17)
120 IF (PEEK(524)AND128)=0 THEN PRINT CHR$(20)
130 DIM NAME$(50):DIM NUM$(50)
140 IF PEEK(#9D00)=170 THEN GOSUB 10000:REM recovers filed numbers
from memory
150 GOSUB 6000:REM Prints title
160 PRINT"  1.  Enter names & numbers."
170 PRINT
180 PRINT"  2.  File numbers on tape."
190 PRINT
```

```
200 PRINT"  3.  Search for a number."
210 PRINT
220 PRINT"  4.  Alter an entry."
230 PRINT
240 PRINT"  5.  Display all entries."
250 PRINT:PRINT:PRINT
260 PRINT CHR$(129);"   SELECT BY NUMBER"
270 GET CHOICE$:CHOICE=VAL(CHOICE$)
280 ON CHOICE GOSUB 1000,2000,3000,4000,5000
290 GOTO 150
300 END
997 :
998 :
999 REM entry subroutine
1000 GOSUB 6000
1010 REPEAT
1020 ENTRIES=ENTRIES+1
1030 INPUT"  Enter name";NAME$(ENTRIES)
1040 PRINT
1050 INPUT"  Enter number";NUM$(ENTRIES)
1060 PRINT
1070 PRINT" 'N' to exit, any key to continue":GET A$
1080 PRINT
1090 UNTIL (A$="N")OR(A$="n")
1100 RETURN
1997 :
1998 :
1999 REM filing subroutine
2000 MEMLOC=#9D00
2010 POKE MEMLOC,170
2020 FOR PSN=1 TO ENTRIES
2030 FOR NAME=1 TO LEN(NAME$(PSN))
2040 MEMLOC=MEMLOC+1
2050 CODE=ASC(MID$(NAME$(PSN),NAME,1))
```

```
2060 POKE MEMLOC,CODE
2070 NEXT NAME
2080 :
2090 :
2100 MEMLOC=MEMLOC+1
2110 :
2120 POKE MEMLOC,255:REM end of word marker
2130 :
2140 :
2150 FOR NUM=1 TO LEN(NUM$(PSN))
2160 MEMLOC=MEMLOC+1
2170 CODE=ASC(MID$(NUM$(PSN),NUM,1))
2180 POKE MEMLOC,CODE
2190 NEXT NUM
2200 :
2210 :
2220 MEMLOC=MEMLOC+1
2230 POKE MEMLOC,255:REM end of number marker
2240 :
2250 :
2260 NEXT PSN
2270 :
2280 :
2290 PRINT:PRINT:PRINT
2300 INPUT"Enter your filename";FILE$
2310 POKE MEMLOC+1,170
2320 CSAVE FILE$,S,A#9D00,E(MEMLOC+1)
2330 RETURN
2997 :
2998 :
2999 REM search subroutine
3000 GOSUB 6000
3010 PRINT"  Search by name facility."
3020 PRINT:X=0
```

```
3030 INPUT"  Enter the name";SEARCH$
3040 FOR PSN=0 TO ENTRIES
3050 IF SEARCH$=NAME$(PSN) THEN X=1:GOSUB 9000
3060 NEXT PSN
3070 IF X=0 THEN GOSUB 8000
3080 GOSUB 7000
3090 RETURN
3997 :
3998 :
3999 REM alter entry subroutine
4000 GOSUB 6000
4010 PRINT "  Entry alteration facility"
4020 PRINT
4030 INPUT"  Enter the name";SEARCH$
4040 PSN=0
4050 REPEAT
4060 PSN=PSN+1
4070 UNTIL SEARCH$=NAME$(PSN) OR PSN>ENTRIES
4080 IF PSN>ENTRIES THEN GOSUB 8000
4090 GOSUB 9000
4100 PRINT
4110 PRINT:PRINT"Change name?":GET A$:PRINT
4120 IF A$="Y" OR A$="y THEN INPUT"New name";NAME$(PSN)
4130 PRINT
4140 PRINT"Change number?":GET A$:PRINT
4150 IF A$="Y" OR A$="y" THEN INPUT"New number";NUM$(PSN)
4160 RETURN
4997 :
4998 :
4999 REM display entries subroutine
5000 GOSUB 6000
5010 FOR PSN=1 TO ENTRIES
5020 PRINT NAME$(PSN),SPC(20-LEN(NAME$(PSN))),NUM$(PSN)
5030 PRINT
```

```
5040 IF PSN/5=INT(PSN/5) THEN GOSUB7000:GOSUB 6000
5050 NEXT PSN
5060 GOSUB 7000
5070 RETURN
5997 :
5998 :
5999 REM title subroutine
6000 CLS
6010 STAR$="************************************"
6020 TITLE$=CHR$(148)+CHR$(131)+CHR$(130)+"TELEPHONE FILE    "+CHR$(
147)
6030 PRINT STAR$
6040 PRINT SPC(5),TITLE$
6050 PRINT SPC(5),TITLE$
6060 PRINT STAR$
6070 PRINT:PRINT:PRINT
6080 RETURN
6997 :
6998 :
6999 REM "Press space..." subroutine
7000 S$=CHR$(129)+"Press <SPACE> to continue"
7010 PLOT 5,23,S$
7020 REPEAT
7030 GET A$
7040 UNTIL A$=" "
7050 PLOT 5,23,"
7060 RETURN
7997 :
7998 :
7999 REM entry not found routine
8000 PRINT
8010 PRINT"  There is no entry for that name"
8020 PRINT
8030 PRINT"  Please use facility 5 for a"
```

```
8040 PRINT"        manual search"
8050 GOSUB 7000
8060 POP:RETURN
8997 :
8998 :
8999 REM Prints search names
9000 S=20-LEN(NAME$(PSN))
9010 PRINT
9020 PRINT NAME$(PSN),SPC(S),NUM$(PSN)
9030 PRINT
9040 RETURN
9997 :
9998 :
9999 REM data recovery subroutine
10000 MEMLOC=#9D00:C=0
10010 IA$="":IU$=""
10020 REPEAT
10030 C=C+1
10040 REPEAT
10050 MEMLOC=MEMLOC+1
10060 IF PEEK(MEMLOC)=255 THEN 10090
10070 M$=CHR$(PEEK(MEMLOC))
10080 IA$=IA$+M$
10090 UNTIL PEEK(MEMLOC)=255
10100 NAME$(C)=IA$
10110 :
10120 REM that was a name
10130 :
10140 REPEAT
10150 MEMLOC=MEMLOC+1
10160 IF PEEK(MEMLOC)=255 THEN 10190
10170 M$=CHR$(PEEK(MEMLOC))
10180 IU$=IU$+M$
10190 UNTIL PEEK(MEMLOC)=255
```

```
10200 NUM$(C)=IU$
10210 :
10220 REM that was a number
10230 :
10240 IA$="":IU$=""
10250 UNTIL PEEK(MEMLOC+1)=170
10260 ENTRIES=C
10270 RETURN
```

This program takes the ideas on using subroutines discussed in the previous chapter a stage further. The program consists of a main section (up to line 270) which is a menu of all the facilities available. The user can choose from them by pressing the appropriate number key. This selects the particular subroutine which performs the task. The subroutine to recover previously-taped data is selected automatically when appropriate. This is discussed later. There are also a number of service routines that are called by the principal subroutines, but not directly by the user.

The program starts with the usual initialising lines to set the colours, set the caps lock (to on in this case) and turn off the cursor. I personally dislike flashing cursors. Since this program calls for user-input you may feel it is better left on, in which case line 120 may be omitted. Lines 90 and 100 will be discussed when we come to the tape filing section. Line 130 dimensions two arrays to hold the names and numbers. As mentioned in the first chapter, this could be done with one two-dimensional array, but I think using two arrays makes for a clearer program in this instance. Line 140 selects the automatic recovery subroutine, and again will be covered later.

Lines 150 to 260 print the title page on the screen. Note that printing the main title has been made a separate subroutine as it is used several times in different parts of the program.

In line 270 we use GET to obtain user-input. Note that we use it to obtain a string input, as if we used it to input a number directly, pressing a letter key would cause an error, stopping the program and losing any data. When a number key is pressed, it is converted into a numeric variable by the function VAL. A letter key returns the value zero. Line 280 uses the ON. . .GOSUB statement which is very valuable in menu-driven programs. The value of the variable CHOICE determines which subroutine the program branches to. If CHOICE=1 the first line number (1000) is selected, if 2 the second (2000) is selected, and so on. Note that it is the position in order after the GOSUB that counts, not the actual line numbers. They only go up in thousands because I like to make things tidy. It would work if I had used intervals of 500 or 300, or irregular intervals.

ORIC's ON. . .GOSUB statement is particularly user-friendly. If the value of CHOICE is 0, as when a letter key is pressed, or greater than five, the program simply falls through line 280, and line 290 sends the program back to the start of the menu section. Most other computers with ON. . .GOSUB statements generate an error if the variable is out of range, and require an extra error trapping line.

Line 290 also sends the program back to the start of the menu sequence after the RETURN from each subroutine.

The first subroutine is the one to input the data into the arrays. It is really quite straightforward. The variable ENTRIES is used to count the number of items entered, and is used in several of the subroutines. Note that the first (zero) element in each array is wasted. It is difficult to write a routine that uses the first element, but allows further entries to be made without over-writing previous ones.

The second subroutine is the one that creates a tape file of the data. In the version of ORIC BASIC on which this book is based, the only functional tape commands are CLOAD AND CSAVE which can be used to save programs (which are not saved with variables) or blocks of memory. There are no specific commands to save arrays. We therefore have to create our own data structure in an area of free memory, and then CSAVE the contents of this area. This sounds awfully difficult, but in fact it isn't.

First, let us consider how ORIC stores string variables and arrays. In the area of memory immediately above the program, ORIC uses 3 bytes of memory to store the length of each string variable, and the address of the first character of the string. The actual characters are stored from the top of the available memory area (called HIMEM) downwards. The default setting of HIMEM (that is, the position it assumes on switch on) is #9EFF. It is raised to #B3FF if the GRAB command is given, and lowered to #97FF if the RELEASE command is given or if HIRES modes is selected. (It follows from this that if you intend entering string variables, and later changing to HIRES mode in the same program, you should use the RELEASE command at the start of the program, or you will lose some or all of your strings).

To reserve an area of memory as a workspace to file our data, we use GRAB in line 90 to protect memory up to #B400, and then use HIMEM #9D00 in line 100. ORIC then uses the area from #9CFF downwards to store the strings, and we use #9D00 upwards to store them in a form suitable for recording.

Line 2000 sets the byte at address #9D00 to 170 decimal. Line 140 uses this to determine whether any data has been loaded from tape. Lines 2020 to 2070 then slice each name into individual characters, convert them into their ASCII codes, and then POKE them into successive memory locations.

At the end of each name, lines 2100–2120 POKE the value 255 into the next byte as a marker for the end of the word. This is used by the recovery routine. Lines 2150 to 2230 then do the same thing for the corresponding number. The loop between lines 2020 and 2260 repeats until all the entries have been converted.

Line 2310 POKEs the byte after the last number to 170, again as a marker for the recovery routine.

Line 2300 asks for a suitable name for the file, and it is then recorded on tape by line 2320. If your tape recorder does not have automatic motor control, start it recording before pressing RETURN after entering the file name. After recording, the program returns to the menu.

Note that the program provides the start and end addresses for the block of memory to be recorded. We do not need to know what they are, as they are encoded on the tape along with the data. In fact, there is no point in specifying start and end addresses when reloading tapes, as they always reload at the encoded addresses.

Whilst it is perfectly possible to record a block of memory from within a program, it is not possible to reload while the program is running. This is because there is a pointer in ORIC's memory that is set to the last address of the last file to be loaded. This is supposed to contain the highest address used by the program file. (The address of this pointer is #9C.) Since, after we have loaded our data file, this pointer will be set higher than our setting of HIMEM, ORIC will assume it has run out of memory space, and stop with the appropriate error message.

So, in order to recover our data, it is imperative that the data tape is loaded first, followed by the program tape, then all will work correctly. This explains why line 140 can be used to recover data automatically. Unused bytes in RAM in ORIC are set to 85, so if line 140 finds #9D00 set to 170, it knows there is data to be loaded. Be warned that if you have a lot of numbers in the file it can take a considerable time to recover them, and there may be quite a pause between running the program and the menu appearing.

If you examine the recovery routine in lines 10000–10270 you will see that it is almost the opposite of the filing routine. Each byte in turn is converted from ASCII code into character, and these are joined together (lines 10080 and 10180) to reform the names and numbers. When a byte set to 255 is discovered, the (completed) name or number is placed in the appropriate array. When line 10250 finds a byte after a number set to 170, it ends the process, sets the value of ENTRIES to the appropriate value (line 10260) and exits the subroutine.

Note that in the recovery routine we use REPEAT. . .UNTIL loops, as we do not know in advance the length of each name or number, or the number of entries, whereas in the filing routine we used FOR. . .NEXT loops.

It may be that in the future simpler methods of data recording and recovery with ORIC will be revealed. In the meantime, this method works and is reliable.

It will be noted that line 2320 records the data at the slow, 300 baud rate. This is necessary for reliable operation with my particular cassette recorder. If you can successfully record and reload programs at the fast rate, you should also be able to use 2400 baud for data.

The subroutine from line 3000 is the search routine. After asking for the search string to be input, at line 3030, it compares it with all the entries in the array, and prints out any match. If it finds more than one match it prints them all. The printing is done by a separate subroutine (from line 9000), and this avoids a long multiple-statement line at line 3050. This print routine is also used elsewhere in the program. If no match is found, another subroutine, from line 8000, prints a suitable message. A "Press SPACE to continue" message is printed at the bottom of the screen, and again, this is a separate routine as it is used in several places in the program.

The subroutine from line 4000 searches for an entry like the previous routine, but it stops if a match is found. It then has facilities to change either the name, or number, or both. Notice how we can use INPUT after an IF. . .THEN statement, also how we can make the computer accept either an upper or lower case Y.

A problem with both the search routines is that the computer will only respond to an exact match. Even one space too many or too few will cause an entry to be passed by. It is therefore a good idea to standardise the format of entries as much as possible, like using capital letters only, and putting a full stop but no space after initials.

If no match is found, this routine calls the same subroutine as the previous one.

There is no specific deletion facility in this program, but this subroutine can be used to over-write an entry that is no longer required. Don't forget you have to record the new version whenever you make an amendment!

The last subroutine prints out all the entries in groups of five, line 5040 interrupting the program as appropriate. The subroutine used in the search routines could have been used to print the entries, but as it takes only a single line, this was considered unnecessarily complex. Note how SPC has been used to print the names and numbers neatly in columns. Try the effect of omitting the commas in lines 5020 and 9020.

The service subroutines are really quite straightforward, but line 8060 in the ,entry not found, routine is interesting. It has the effect of making the program go straight back to the menu, instead of through the subroutine that called the subroutine. This avoids having to press the space bar twice to get out of the search routine if no match is found, and also is an elegant way of skipping lines 4090 onwards in the alteration routine, which are not appropriate if no match is found.

In the press space routine, you will note that the value 129 is used to give the foreground red colour to this line. You may remember from the chapter on serial attributes, and the game in the previous chapter, that when we use the high values for background attributes with PLOT statements, the actual position they occupy appears in the inverse colour. With foreground attributes, they appear in the inverse of the current background colour, in this case blue. In fact, this value was used because it was originally intended to PRINT this line, but when I changed it to PLOT, I rather liked the effect of the blue square to the left of the line, and decided not to change it.

This program is not pefect. No program is. All are capable of being extended and improved. I hope you will experiment with it, perhaps try to add search routines that try to match only the first few letters, to make it easier to find an entry if you cannot remember exactly what you entered.

Remember, nothing you type in on ORIC's keyboard can harm it, and once you have a copy of a working program on tape, it does no harm to change it. If the modifications don't work, just pull the plug and reload the original, and start again.

If you aren't sure if a programming idea will work, try it and find out. Don't worry about error messages. They are tools for the adventurous, not traps for fools. Practically every program in this book contains some line that I was not sure about until I tried it. It's the only way to learn.

Chapter 9

INTERFACING

Using a computer for control and measurement applications seems to be an increasingly popular aspect of home-computing. Probably the majority of ORIC owners will not use the machine for anything other than games, filing, or similar applications that require standard extras such as a TV set or cassette recorder, but for those who do wish to try using the unit as the basis of something a little more exotic the ORIC has good potential.

There are several sockets on the rear of the machine, one of which is a phono socket which connects to a TV set (which acts as a monitor) via the lead supplied with the computer. There is also an RGB monitor output which enables a colour monitor to be used with the machine.

The parallel printer output is the 20 way IDC connector, and it is advisable, if possible, to check before you buy a printer that it will operate properly with the ORIC. There are minor variations on parallel (and series) printer ports, and problems of incompability between a particular computer and a particular printer do sometimes arise.

At the time of writing ORIC's own colour printer was imminently due for release, this of course should operate perfectly with ORIC–1.

The cassettte socket is a 7 way DIN type and permits control of the motor in the cassette recorder provided the latter has a remote socket. There is also an audio output here which seems to give adequate output to drive a hi-fi or similar amplifier so that increased volume can be obtained, but there does not seem to be any way of muting the internal loudspeaker of the ORIC (although this is not a great drawback). It is possible to use the audio output as an input if desired, incidentally. If you find that when loading a program from cassette the output of the recorder is reproduced loudly through the internal loudspeaker of the machine and loading of the program does not occur, this almost certainly means that the cassette lead is not connected correctly and the audio output from the recorder has been connected to the audio output of the computer. If a cassette lead specifically for the ORIC cannot be obtained it seems to be possible to use a BBC cassette lead without any problems. Most cassette recorders require a lead fitted with two 3.5mm jacks and one 2.5mm jack, but some others require a DIN connector and you must be careful to obtain the correct type for your recorder. With the type that use jack plugs it is essential to connect the two 3.5mm plugs correctly (one to the earphone socket and the other to the microphone or line input socket), but if you are in any doubt about this there is little risk of anything being damaged if you determine the correct method of connection by trial and error.

Expansion Socket

The expansion socket is the 34 way IDC type, and it is probably best not to try experimenting with any home-constructed add-ons to this unless you are reasonably sure that you understand what you are doing. The ORIC is not easily damaged by accidental short circuits and similar mishaps at the expansion port, but there is the potential for causing expensive damage if a serious mistake should be made.

All the connections one would hope for are present at the expansion port, including the address bus (A0 to A15), the data bus (D0 to D7) and a number of control lines (including the clock signal at pin 5). Although the ORIC uses a 6502A microprocessor, which is the 1.5MHz version of the popular 6502 microprocessor, the clock signal is at a frequency of 1MHz. Standard 1MHz peripheral interface devices such as the 6821 and 6850 can be connected to the expansion port, and it is not necessary to use the more expensive A series devices.

The 6502 is normally used with straight forward memory mapped input and output devices. In other words, the input and output devices appear to the 6502 as if they were ordinary RAM or ROM memory locations, and a gap or gaps for the input and output circuits (including the internal ones of the computer) must be left somewhere in the 64K addressing range of the microprocessor. Reference to the memory map in Appendix A of the ORIC manual shows that there are spare memory locations from BFE0 to BFFF inclusive. However, an important point to bear in mind is that the 48K version of the ORIC has 64K RAM chips which occupy every memory location within the address range of the 6502. The ROM which contains the basic interpreter and operating system is used to overlay (approximately) the 16K at the upper end of the memory.

The practical importance of this is that it slightly complicates interfacing to the expansion port. There is no real problem when using this port as an output because this simply means that any data sent to an output device will also be sent to RAM which shares the same memory location. It is difficult to envisage circumstances under which this would be of any importance, or even where it would become apparent to the user at all.

The situation is rather different when information is fed into the expansion port. When the microprocessor reads the memory location where the input device is situated it will also read the RAM at that location. This would result in both the input device and the RAM placing an output onto the data bus simultaneously. This is not likely to result in any damage to any of the hardware, especially when one takes into consideration the very short period of time that both devices would be producing an output onto the data bus. However, it is obviously not a very satisfactory state of affairs, and the data from the input device would almost certainly be corrupted.

There is a simple solution to the problem, since the ORIC is designed so that internal devices such as the RAM can effectively be disconnected from the data bus when reading from an input device. The relevant pins on the expansion port are 1 (MAP), 2 (ROMDIS), and 6 (I/O Control). These are normally in the high logic state, but taking them low when reading from an input circuit enables the data to flow unhindered into the machine.

Simple Circuits

The basic method of interfacing to the expansion port is perhaps best explained with the aid of simple output and input port circuits, such as those shown in Figure 5 and Figure 6 respectively. We will consider the circuit of Figure 5 first as this is the more straightforward of the two.

Output Port

The top twelve address lines (A4 to A15) are decoded by IC1 to IC3. IC1 is a dual 4 input NAND gate and IC2 is a single 8 input NAND gate. The output of a NAND gate goes low if all its inputs are high, but IC1a is used as a simple inverter which is interposed between A14 and one input of IC1b. IC3 is a quad 2 input NOR gate, but in this circuit only two of the four gates are used. These simply combine the outputs of IC1b and IC2 so that a low output is only produced from IC3b when both these outputs are low. Thus a brief negative pulse is produced by IC3b when address lines A4 to A15 are high, apart from A14 which must be low (due to the inclusion of an inverter here). This occurs with any address from BFF0 to BFFF (in Hex), and this block is within the range of spare addresses mentioned earlier. If more than one input/output port is required the lower four address lines could also be decoded, or the use of a special PIA device such as a 6522 or 6821 might be a more practical approach.

IC4 is a 74LS273 octal D type flip/flop, and this is used to provide eight latched outputs from the data bus. This device latches on a negative transition of the CP input at pin 11. A suitable pulse is supplied to IC4 by IC3b whenever an address in the range BFF0 to BFFF appears on the address bus. The POKE command can be used to send the desired number to the data bus (and at one of the appropriate addresses). For example, POKE #BFFF,255 would send 255 to the output port (all eight outputs in the high state). Note that the # sign must be used to show that the address is given in Hex. It does not matter which particular address in the range BFF0 to BFFF is used, and it is not even necessary to always use the

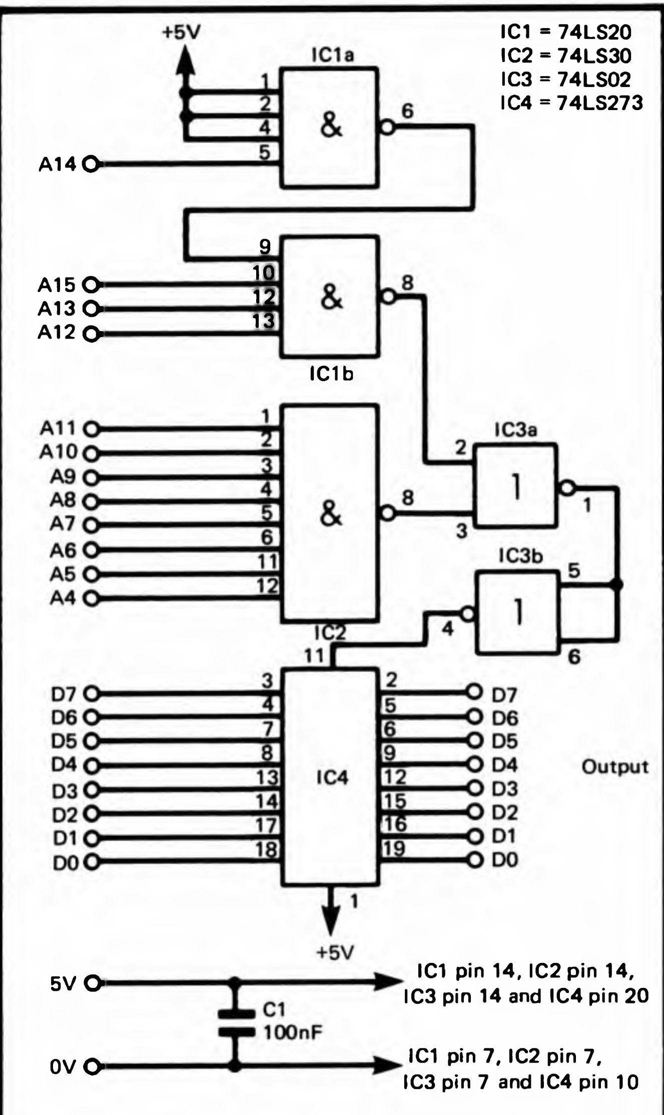

Fig. 5. A simple output port for the Oric 1

same one!

In order to understand what set of output states a given number will generate it is necessary to have a basic understanding of the binary numbering system. All that it is really necessary to know is that each output represents 0 if it is low, or a number between 1 and 128 (for an eight bit or eight line port anyway) if it is high. Each data line represents a different number, as shown in the table below:

D0	1
D1	2
D2	4
D3	8
D4	16
D5	32
D6	64
D7	128

By using a number in the range 0 to 255 inclusive it is possible to obtain any desired set of output states. For instance, to set D2, D4, and D5 high it would be necessary to send 52 to the output port (4 + 16 + 32 = 52).

When POKEing a number to an output port it is acceptable for the number sent to be a numeric variable (i.e. POKE #BFFF,X), but the variable must be an integer. The basic INT command can of course be used to ensure that the variable is an integer if necessary. Also, the variable must be a positive number within the 0 to 255 range, and not a higher or negative number.

Power for the output port can be taken from the two supply terminals of the expansion port. The ORIC manual does not specify the maximum current that can be drawn, but experimentation suggests that up to at least 100 milliamps can be taken without any difficulties arising.

Input Port

The input port of Figure 6 uses what is basically the same address decoding as the output port just described. However, there is a difference in that a 3 input NOR gate is used in this

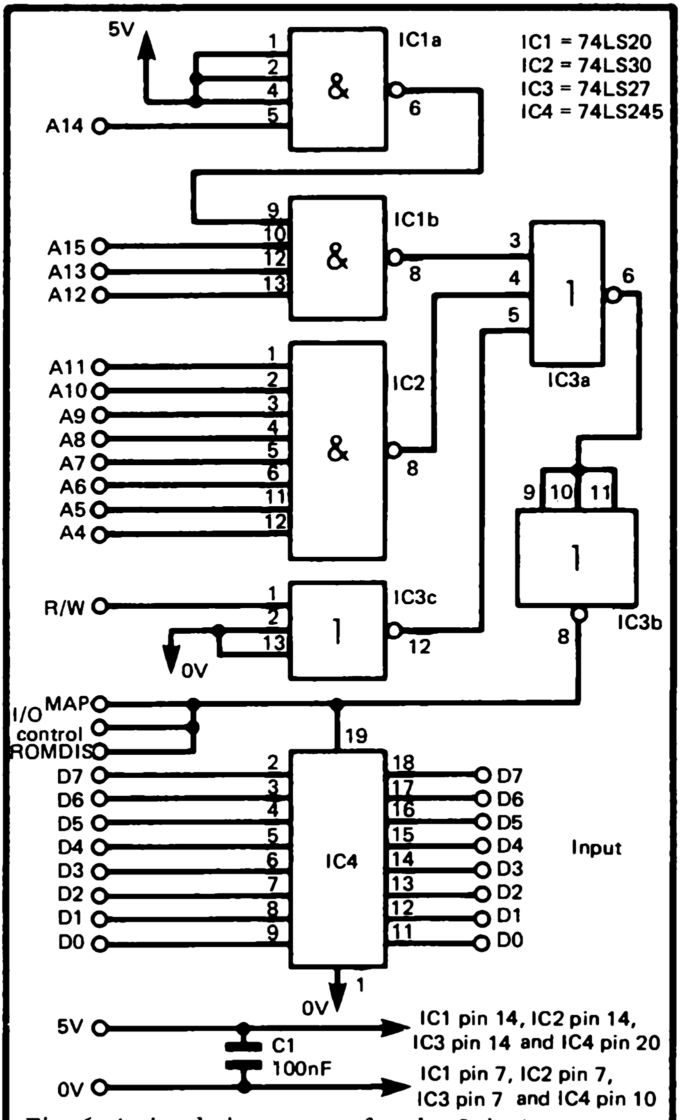

Fig. 6. A simple input port for the Oric 1

circuit, whereas a 2 input type was used in the first one. This has been done so that the R/W (read/write) line can also be gated along with the upper twelve address lines. This simply ensures that an accidental write operation to the input port cannot result in the microprocessor placing an output onto the data bus at the same time as the input port. The R/W line is high during read operations, and IC3c is therefore used as an inverter which gives the required negative signal to IC3a when the input port is read.

IC4 is a 74LS245 octal transceiver, but it is used here with the send/receive terminal (pin 1) permanently held low so that the device is always in the receive mode. It would in fact be possible to use an octal tri-state buffer such as the 74LS244 for IC4 but the pin layout of the 74LS245 is more convenient and this device is probably a more practical choice.

Pin 19 of IC4 is the negative chip enable input, and this will normally be in the high state so that IC4 is disabled and its outputs are at a high impedance. This is essential as the device would otherwise be permanently providing an output on the data bus and would prevent proper operation of the computer. Only when an address between BFF0 and BFFF is read will the negative chip enable input be taken low and the outputs of IC4 then placed onto the data bus whatever logic levels are fed to their inputs at that instant.

The MAP ROMDIS and I/O Control terminals of the expansion port are all taken low when the port is read so that internal devices of the computer are all disconnected from the data bus while the read operation takes place. It is essential that these three lines should only be taken low at this time since the normal working of the computer is blocked while they are in this state (in other words the microprocessor is only able to communicate with an external device while these inputs are low) If you try manually taking these lines briefly to the low state the computer will simply stop operating and is unlikely to start again when they are returned to the high state since the operating system will have crashed (although switching off and turning the unit on again should restore normal running).

Data is read from the port using the PEEK command (i.e.

PRINT PEEK (#BFFF) or X = PEEK (#BFFF)) As was the case with the output port, any address from BFF0 to BFFF can be used when communicating with the input port, and it is not necessary to always use the same address, although it would be advisable to do so to avoid possible confusion. The number obtained from the port is an eight bit binary type, but this is converted by the computer into a decimal number of between 0 and 255. This is just like writing to the output port but in reverse. For example, if input lines D0, D1, D2 and D3 are high the number returned would be 15 (1 + 2 + 4 + 8 = 15)

When using more complex input/output ports it would be necessary in most cases to use more of the control lines of the computer, such as the reset, clock, and IRQ (interrupt request) lines, but these are all available at the expansion port. Appendix F of the ORIC manual gives full details of the connections to all the sockets on the rear of the machine.

Logic AND

When using an input port it is not always all eight inputs that are of interest, and it may be that just one or two lines are all that need to be read. There is no way of reading less than the full eight lines, but there is a simple way of masking any input lines that are of no interest and effectively reading only those that are significant.

This is achieved using the logic AND function. Here we are using the AND command in what is often termed the bitwise role, and not in the more simple role which we have used in programs given earlier in this book, where it is used in much the same way as the plain English word "and". When employed in the bitwise mode it is analagous to a logic AND gate.

Probably the best way to explain the operation of the AND command is to take a simple example of its use. We will assume that we wish to read an input port but only wish to know the state of line D7 (which equals 0 when low or 128 when high). In order to read just one bit the number from the input port is ANDed with whatever number that bit represents when it is in the high state, or 128 in this case. Thus using the command PRINT PEEK (#BFFF) AND 128 would give the answer 0 if D7 is low, or 128 if it is high.

What actually happens when two numbers are ANDed is that they are compared bit by bit, and a 1 is produced in the answer only if there is a 1 in that bit of both the first number AND the second. For example, if the number returned from the input port is 15 and it is ANDed with 128 this gives the following result:

15	=	00001111
128	=	10000000
0 (answer)	=	00000000

Obviously the answer is zero since no column has a 1 in both numbers. If the number returned from the input port was 143 the result would be as follows

143	=	10001111
128	=	10000000
128 (answer)	=	10000000

It can be seen from this that using a 0 in a bit of one of the numbers ensures that there must be a 0 in that bit of the answer, but if a 1 is used in a bit of one number the result in that bit of the answer is equal to the number in the corres-ponding bit of the other number. By setting the lower seven bits to 0 the lower seven bits of the answer must be 0, but by setting the top bit (most significant bit or MSB) to 1 the answer is a true reflection of that bit in the number from the input port. Of course, one of the numbers does not need to be a reading from an input port, and it is possible to AND any two eight bit binary numbers in this way. By using the appro-priate masking number it is possible to read any one bit, or more than one bit if necessary (3 would be used to read the two least significant bits for instance).

Logic OR

This is another command that can be used virtually as an ordinary English word, or as a logic operation. When used as a logic operation it is used in much the same way as AND, but when the two binary numbers are compared a 1 appears in the answer if there is a 1 in that bit of the first number OR the second. Thus ORing 128 and 15 would give the following result:

```
15              =          00001111
128             =          10000000
143 (answer)    =          10001111
```

ORing 143 and 128 would give this result:

```
128             =          10000000
143             =          10001111
143 (answer)    =          10001111
```

There is another logic operation which is available, and this is exclusive OR. However, this does not seem to be included in ORIC BASIC and could only be obtained using machine code. This logic operation is similar to the standard OR function, but a 1 is only produced in the answer if there is a 1 in that bit of one OR other of the numbers being processed. A 1 is not produced in the answer if there is a 1 in that bit of both the numbers being processed. If 15 and 128 are exclusive ORed the answer would be the same as when using the standard OR function. However, if 128 and 143 are exclusive ORed the following result is obtained:

```
128             =          10000000
143             =          10001111
15 (answer)     =          00001111
```

This obviously differs from the similar previous example using the ordinary OR function.

Chapter 10

ODDS AT THE END

This chapter is a miscellany of useful odds and ends, and explains some of the mystery lines used in the programs in this book.

Some of ORIC's operations, such as the keyclick control and the caps lock, are controlled either from the keyboard or from within programs by a toggle on-off action. This makes it hard to ensure from a program that, for example, the keyclick is turned off so it doesn't interfere with sound effects. If you put the statement PRINT CHR$(6) in to turn the keyclick off, if the user has already turned off the keyclick with CTRL F before running the program, it will turn it on again!

Fortunately, there is a telltale byte in memory which can be PEEKed to determine whether these toggle controls are set. It is at location 618 decimal (#26A). This byte contains the status of the keyclick, cursor, printer, VDU protected column and auto double-height toggles. Location 524 decimal (#20C) contains the status of the caps lock toggle.

To determine whether a particular control is on or off, we have to determine whether a particular bit in the byte is high or low. We can do this by using the Boolean AND operator. Thus, if we AND PEEK(618) with 1 it will check the first bit, which reveals the cursor status, and if we AND it with 8 it checks the keyclick status. So the line:

```
IF (PEEK(618)AND1)=1 THEN PRINT CHR$(17)
```

will ensure the cursor is turned off. Reversing the test ensures it is always turned on, thus:

```
IF(PEEK(618)AND1)=0 THEN PRINT CHR$(17)
```

To check for caps lock set we AND PEEK(524) with 128. Using =0 will ensure caps on, using =128 will ensure caps off.

The numbers to be used with PEEK(618) are as follows:

1	CURSOR
2	VDU
4	PRINTER
8	KEYCLICK
16	NOT USED
32	PROTECTED COLUMN
64	AUTO DOUBLE HEIGHT
128	NOT USED

The appropriate bits are, in most cases, high when the functions are on, but the keyclick bit is low when on. The protected column bit is low when the column is protected. Though bit 5 is described as unused, it seems to be associated with column 1 in the TEXT and LORES modes (the column that controls the INK colour in TEXT mode). However, it has no useful function for the programmer. Bit 8 is apparently completely unused.

It is possible to control these functions by POKEing these bytes with appropriate values, but in some cases this does not seem to do everything that is done by using PRINT CHR$(X), so on the whole it is not to be recommended.

Useful Memory Locations

These addresses are all the low bytes of two-byte locations, and should be read with the function DEEK, rather than PEEK. All these addresses are in Hex

#9A	Start address of program file.
#9C	Top of program file.
#9E	Top of dynamic variables
#A0	Top of variable arrays.
#A2	Bottom of string variable storage
#A6	Current setting of HIMEM
#F42D	First address of start-up routine.

Typing CALL #F42D will have the same effect as pulling the plug out!

Simple Renumber Program

It is normal, when writing programs, to number the lines with intervals of ten to allow for insertion of extra lines if necessary. It sometimes happens that a lot of extra lines need to be inserted, and a gap of ten is insufficient. This short routine can be typed in at the end of a program, and used to renumber the lines back to intervals of ten (or any other required number), allowing further insertions.

This simple routine takes no account of GOTOs and GOSUBs, so you have to alter these manually each time the routine is used. The routine can be called at any time by typing GOTO 25000. Line 24999 is to ensure the routine never executes without being called.

```
24999 END
25000 M=1281:N=10: S=10
25010 REPEAT
25020 DOKE M+2,N
25030 M=DEEK(M)
25040 N=N+S
25060 UNTIL DEEK(M+2)=24999
```

In line 25000, N is the number to be given to the first line, S is the step size. These can be altered as required.

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

205	First Book of Hi-Fi Loudspeaker Enclosures	95p
221	28 Tested Transistor Projects	1.25p
222	Solid State Short Wave Receivers for Beginners	1.25p
223	50 Projects Using IC CA3130	1.25p
224	50 CMOS IC Projects	1.35p
225	A Practical Introduction to Digital IC's	1.25p
226	How to Build Advanced Short Wave Receivers	1.95p
227	Beginners Guide to Building Electronic Projects	1.95p
228	Essential Theory for the Electronics Hobbyist	1.95p
RCC	Resistor Colour Code Disc	20p
BP1	First Book of Transistor Equivalents and Substitutes	1.50p
BP6	Engineers and Machinists Reference Tables	75p
BP7	Radio and Electronic Colour Codes and Data Chart	40p
BP14	Second Book of Transistor Equivalents and Substitutes	1.75p
BP24	52 Projects Using IC741	1.25p
BP27	Chart of Radio Electronic Semiconductor and Logic Symbols	50p
BP32	How to Build Your Own Metal and Treasure Locators	1.95p
BP33	Electronic Calculator Users Handbook	1.50p
BP34	Practical Repair and Renovation of Colour TVs	1.25p
BP36	50 Circuits Using Germanium, Silicon and Zener Diodes	1.50p
BP37	50 Projects Using Relays, SCR's and TRIACs	1.95p
BP39	50 (FET) Field Effect Transistor Projects	1.75p
BP40	Digital IC Equivalents and Pin Connections	3.50p
BP41	Linear IC Equivalents and Pin Connections	3.50p
BP42	50 Simple L.E.D. Circuits	1.50p
BP43	How to Make Walkie-Talkies	1.95p
BP44	IC555 Projects	1.95p
BP45	Projects in Opto-Electronics	1.95p
BP47	Mobile Discotheque Handbook	1.35p
BP48	Electronic Projects for Beginners	1.95p
BP49	Popular Electronic Projects	1.95p
BP50	IC LM3900 Projects	1.35p
BP51	Electronic Music and Creative Tape Recording	1.95p
BP52	Long Distance Television Reception (TV-DX) for the Enthusiast	1.95p
BP53	Practical Electronic Calculations and Formulae	2.95p
BP55	Radio Stations Guide	1.75p
BP56	Electronic Security Devices	1.95p
BP57	How to Build Your Own Solid State Oscilloscope	1.95p
BP58	50 Circuits Using 7400 Series IC's	1.75p
BP59	Second Book of CMOS IC Projects	1.50p
BP60	Practical Construction of Pre-amps, Tone Controls, Filters & Attn	1.45p
BP61	Beginners Guide to Digital Techniques	95p
BP62	Elements of Electronics – Book 1	2.25p
BP63	Elements of Electronics – Book 2	2.25p
BP64	Elements of Electronics – Book 3	2.25p
BP65	Single IC Projects	1.50p
BP66	Beginners Guide to Microprocessors and Computing	1.75p
BP67	Counter Driver and Numeral Display Projects	1.75p
BP68	Choosing and Using Your Hi-Fi	1.65p
BP69	Electronic Games	1.75p
BP70	Transistor Radio Fault-Finding Chart	50p
BP71	Electronic Household Projects	1.75p
BP72	A Microprocessor Primer	1.75p
BP73	Remote Control Projects	1.95p
BP74	Electronic Music Projects	1.75p
BP75	Electronic Test Equipment Construction	1.75p
BP76	Power Supply Projects	1.75p
BP77	Elements of Electronics – Book 4	2.95p
BP78	Practical Computer Experiments	1.75p
BP79	Radio Control for Beginners	1.75p
BP80	Popular Electronic Circuits – Book 1	1.95p
BP81	Electronic Synthesiser Projects	1.75p
BP82	Electronic Projects Using Solar Cells	1.95p
BP83	VMOS Projects	1.95p
BP84	Digital IC Projects	1.95p
BP85	International Transistor Equivalents Guide	2.95p
BP86	An Introduction to Basic Programming Techniques	1.95p
BP87	Simple L.E.D. Circuits – Book 2	1.35p
BP88	How to Use Op-Amps	2.25p
BP89	Elements of Electronics – Book 5	2.95p
BP90	Audio Projects	1.95p
BP91	An Introduction to Radio DX-ing	1.95p
BP92	Electronics Simplified – Crystal Set Construction	1.75p
BP93	Electronic Timer Projects	1.95p
BP94	Electronic Projects for Cars and Boats	1.95p
BP95	Model Railway Projects	1.95p
BP96	C B Projects	1.95p
BP97	IC Projects for Beginners	1.95p
BP98	Popular Electronic Circuits – Book 2	2.25p
BP99	Mini-Matrix Board Projects	1.95p
BP100	An Introduction to Video	1.95p
BP101	How to Identify Unmarked IC's	65p
BP102	The 6809 Companion	1.95p
BP103	Multi-Circuit Board Projects	1.95p
BP104	Electronic Science Projects	2.25p
BP105	Aerial Projects	1.95p
BP106	Modern Op-Amp Projects	1.95p
BP107	30 Solderless Breadboard Projects – Book 1	1.95p
BP108	International Diode Equivalents Guide	2.25p
BP109	The Art of Programming the 1K ZX81	1.95p
BP110	How to Get Your Electronic Projects Working	1.95p
BP111	Elements of Electronics – Book 6	3.50p
BP112	A Z-80 Workshop Manual	2.75p
BP113	30 Solderless Breadboard Projects – Book 2	2.25p
BP114	The Art of Programming the 16K ZX81	2.50p
BP115	The Pre-Computer Book	1.95p
BP116	Electronic Toys Games and Puzzles	2.25p
BP117	Practical Electronic Building Blocks – Book 1	2.25p
BP118	Practical Electronic Building Blocks – Book 2	2.25p
BP119	The Art of Programming the ZX Spectrum	2.95p
BP120	Audio Amplifier Fault-Finding Chart	65p
BP121	How to Design and Make Your Own PCBs	2.25p
BP122	Audio Amplifier Construction	2.25p
BP123	A Practical Introduction to Microprocessors	2.25p
BP124	How to Design Electronic Projects	2.25p

BERNARD BABANI BP129

An Introduction to Programming the ORIC-1

- This book has been written for readers wanting to learn more about programming and how to make best use of the ORIC—1 microcomputer's many powerful features. Most aspects of the ORIC—1 are covered, the omissions being where little could usefully be added to the information provided by the manufacturer's own manual.

- Starting with simple commands and programs, the more complex topics such as animated graphics and using sound commands are introduced and illustrated by longer and more sophisticated programs. The many programs included in the book are mostly games and these serve as an interesting way of demonstrating points and gaining programming experience even if you eventually intend to produce programs for more serious applications.

- The text is divided into the following chapters: 1, Variables and Codes; 2, Ins and Outs; 3, Animation and Loops; 4, Attributes, Characters and Time; 5, Using the Sound Generator; 6, Decisions; 7, Structured Programming; 8, Data Filing Ideas; 9, Interfacing; 10, Odds at the End.

- Essential reading for all ORIC—1 owners be they beginners or seasoned programmers.

GB £ NET +001.95

ISBN 0-85934-104-6

00195

£1.95

9 780859 341042