ATMOS, ORIC-1, New and Old ROMs

Getting MORE from your ORIC

H.E.Hicks

Sigma Technical Press

Getting More from your ORIC

Henry Hicks

Sigma Technical Press

ISBN: 0 905104 56 0

Published by: Sigma Technical Press, 5 Alton Road, Wilmslow, Cheshire, SK9 5DY, U.K.

Typesetting and production: originally input by the author on an Apple IIe using the Applewriter word processing package; subsequent typesetting and production by Designed Publications Ltd.

Distributed by: John Wiley and Sons Ltd., Baffins Lane, Chichester, Sussex, P019 1UD.

Printed and bound in Great Britain by
J. W. Arrowsmith Ltd., Bristol

CONTENTS

CHAPTER 1

Getting Started

This book is intended for those people who own an ORIC computer with version 1.0 or 1.1 ROM, or the new ORIC Atmos (see Figure 1.1), and who want to squeeze as much out of the machine as possible. All the standard facilities are described, as well as some which are very much non-standard.

There are the expected sections on number systems and arithmetic to ensure that the computer owner need only buy one book to satisfy most of his or her needs.

There is also a fairly extensive section about the microprocessor on which the ORIC is based so that more advanced users can learn how to write programs in Assembly language and thus make the most of their investment. This section might also be of interest to those who own machines other than the ORIC but based on the same microprocesor.

Figure 1.1

The designers of the ORIC-1 and ORIC Atmos intended that the machines should form the basis of a computer system, with all the usual peripherals associated with such a system, available to the owner. Already on the market is a four colour printer which can also produce good quality graphics. A modem has been available for some time, allowing users to access remote databases such as those provided by the more far-sighted information providers, and a disc drive system based on 3 inch drives is now available (see Figure 1.2).

Apart from the short description of the ORIC computers and their facilities which follow in this chapter, it will be assumed that the reader has at least absorbed enough of the manual supplied with the machine to put the right connectors into the right holes and switch on.

1.1 The Hardware.

The ORIC computer is a 6502 based machine that is available in two sizes. The smaller version has 16K of user memory and the larger has 48K. The two versions are in fact physically different and so a home-based upgrade from the 16K version to the 48K is not possible. This rather odd situation is brought about by the larger machine actually having 64K of dynamic memory, the top 16K of which is masked out by the internal program memory. The design of dynamic memory chips makes it very difficult to allow upgrading from one memory size to another in a compact system. Consequently there is a board layout difference between the two machines. These should be thought of as two different computers which happen to run the same operating system and the same version of BASIC. The manufacturers of the computer do offer an upgrade based on a trade-in deal so all is not lost.

The keyboard is an almost standard QWERTY layout with four extra cursor control keys for editing purposes. The keys have tactile and audible feedback though the latter is defeatable by typing CTRL-F. (Hold down CTRL and press F at the same time. Usually printed as 1F to save space.)

Along the back of the machine are the connections to the outside world. There is a co-axial power supply socket which uses 9v D.C from a mains adapter. Next to this is the expansion bus connector which has the bus and control lines for the 6502, except for NMI, SYNC and RDY, and some special lines which give greater control over the memory map. Next to this is the parallel printer port which is wired as a Centronics standard interface. This port is not buffered however, so you can only use short cable runs to the printer for reliable operation. The cassette output is next with a relay connected across pins 6 and 7 to give remote control facilities.

Pins 4 and 5 of this connector allow internally generated sounds to be played through an external amplifier system. There is an R.G.B. colour monitor connector next to the sound socket for those with such a device. Certainly colour monitors give a much crisper picture than the average T.V. set. Lastly there is the standard UHF connector for the average T.V. set. The ORIC has two holes in the base through which a screw head and an adjustable potentiometer can be seen. These exist so the signal produced by ORIC can be adjusted to suit the T.V. set. A sort of

fine tuning capability. The internal modulator is set to approximately channel 36.

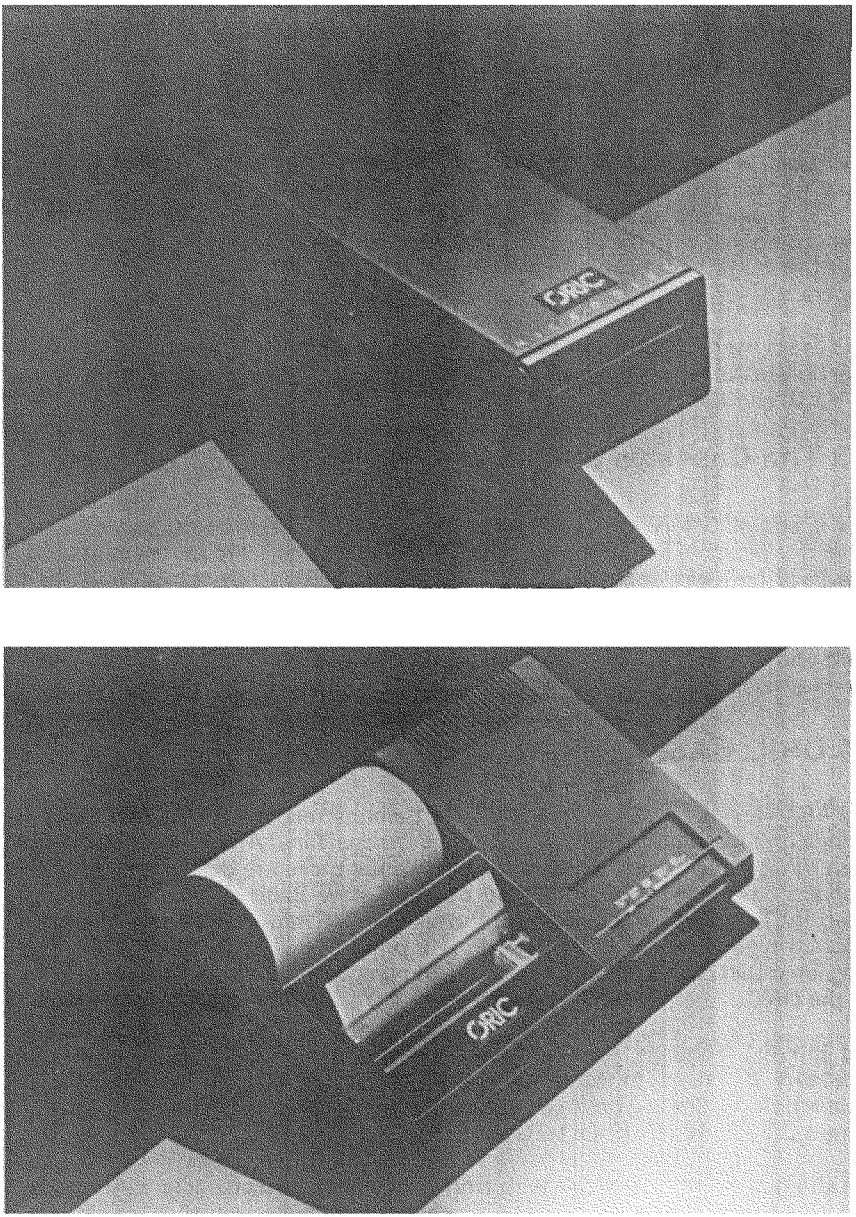


Figure 1.2: The ORIC Disk Drive and Printer

1.2 The ORIC-1, ORIC Atmos, New and Old ROMs

The ORIC Atmos is a recent development of the ORIC-1. It is very similar to the ORIC-1 both in terms of its internal circuitry and its external appearance, the main difference being the FUNCTION key, which is non-functional at present (future versions of the Atmos will use this key for software control).

The Atmos is essentially an ORIC-1 with the new 1.1 ROM (Read Only Memory) which contains all of the computer's permanent software, such as the BASIC interpreter and the operating system. This new ROM has some enhancements when compared with the older 1.0 ROM (as used in the earlier ORIC-1) and the 1.0 ROM 'bugs' have been removed. The new ROM is available to present ORIC-1 owners.

Because of the great similarity of the ORIC-1 and ORIC Atmos, we shall henceforth refer only to the 'ORIC' for brevity. Also, we will refer to the old version of the ROM as 'V1.0' and the new version as 'V1.1', again for brevity.

1.3 The Cassette System.

This is straight - forward to use and has a good pedigree, being based on the Tangerine design which has worked well for many people for many years. The one drawback, for users of the ORIC with the older V1.0 ROM, is the lack of a catalog command, or its equivalent, so records must be kept of names of programs and their positions on the tape. In V1.1 the computer names the programs as it meets them and thus provides an effective catalog of the tape. This new system also provides facilities for saving arrays and for merging programs.

1.4 ORIC BASIC.

The 48K version comes complete with a demonstration program showing the various graphics and sound capabilities of the machine. After this program has been run it can be listed to see how the various tricks are done. Note that this program has been recorded at the optional slow Baud rate to ensure high reliability. The program is rather long and takes over 6 minutes to load. To load this program type CLOAD "",S(RETURN). If a printer is available, a listing of this program would provide a valuable reference in programming the special techniques for the ORIC.

The internal language for the ORIC is standard Microsoft BASIC with the necessary additions to cater for the sound and graphics capabilities. The arithmetic is full floating point with an accuracy up to 9 digits. The numerical range is from 2.93874E10-39 to 1.70141E10+38.

Full string handling is available with strings limited to 255 characters. Arrays can also only have 255 elements and may only be one-dimensional.

Variables may have names of any length, but only the first two characters are used.

Allowable characters are the letters A-Z and the numbers 0-9. In addition, one of three suffix types may be used. $ indicates a string, % indicates an integer and no suffix indicates floating point.

The internal sound commands allow for the production of up to three pure tones at any one time, with the facility to add in noise if required. The noise base frequency may be varied and the envelope of the sound may also be selected out of a total possible of seven types.

The display may either be all text in a mode which is very similar to Teletext, or it may be nearly all graphics, with three lines at the bottom for text. Characters can be in different colours, double or single height, flashing or static. Background colours can also be varied.

The hidden key underneath the computer is connected to the NMI pin of the 6502 and provides a way out of program loops and crashes without destroying programs or data. The screen is cleared when this takes place.

1.5 Peripherals.

As an optional extra, a modem is available which allows computer to computer communication over the standard telephone lines. Obviously the two computers must each know what is going on for this to work. Also available is a printer and a micro disc drive (see Figure 1.2). The disc drive is an Hitachi three inch mechanism, which is compatible with the standard five-and-a-quarter inch drives, and these drives may be mixed in with the three inch ones on the same interface. The interface used can support up to four disc drives at any one time.

1.6 Switching On.

When setting up ORIC for the first time, it may well be that the machine does not start correctly due to the reset circuitry inside 'seeing' too slow a rising edge on the power supply line. If this is suspected, leave the external power pack switched on and unplug the D.C.connector at the back of the machine. On pushing this connector back in a reliable start up will be achieved. This can be tested without a T.V picture because after a restart pressing any key on the keyboard will produce a 'click' sound if the ORIC is working. Note that restart can take a second or two after switch-on to allow the machine to sort itself out.

After tuning in the T.V set to obtain a picture of ORIC'S workings, run the demonstration program to finalise the adjustments as this program gives good quality colour.

1.7 Hints and Tips.

For those who have never used a computer and cassette system before, a few words of advice may not go amiss. Always use good quality cassettes with never

more than 15 minutes worth of tape per cassette. It is very tempting to use longer cassettes for bulk storage but the longer the tape, the thinner it has to be to fit into the cassette and hence the more likely it is to stretch. Also the oxide coating has to be thinner with consequently greater chance of a bad film.

When editing a program, work with two cassettes. The program to be edited is on cassette A and cassette B is blank. Make sure B is blank by using a bulk eraser.

Load the program from A and check that it is what you expected by listing it. (Note that listing can be temporarily stopped by pressing the space bar. Pressing any key will restart the listing.) After editing and testing, save the program onto tape B, making a note of the program name and, if the tape recorder has a counter, the reading of the counter at the start and finish. Tape A may contain more than one program so it is not possible to erase tape A until all programs have been transferred to tape B. This is the second reason for using short tapes. After transferring all programs from A to B, erase A completely with a bulk eraser and now tape B is effectively the same as tape A was at the start of the editing session.

When you have finished a program and are happy with the results, save it onto at least two cassettes. This technique, known as backing up, is of great importance and should ideally be used at all times. Note that during the editing session (until final testing of tape B) two copies of each program exist, those on tape A and the edited versions on tape B. If anything goes wrong with tape B there is always the previous copy to fall back on.

A completely organised system actually uses at least three cassettes. One blank one for recording the edited versions and two originals, one of which is used to provide the programs for editing and the back-up copy which is kept in a safe place. When the edited programs are fixed on tape B, the first two tapes are bulk erased and re-programmed one at a time, testing the first before erasing and re-recording the second.

This all seems like a lot of fuss and bother which will merely increase the time and effort needed in editing. There is no denying that it is a fair amount of work.

Programmers who work on main-frame machines are fortunate in that most of this work is done for them by the operating system. When a new program is edited, a back-up copy is made by the system automatically. However, the first time a cassette program fails to load sucessfully after spending days getting it just right, the wisdom of having a back-up copy will become only too plain.

CHAPTER 2

ORIC's BASIC

There must be very many people who, wondering what all the fuss over microcomputers was about, have gone out and bought a microcomputer to satisfy their curiosity. Some of these people may still be wondering. The answer lies in the fact that computers can be used to do almost anything provided they are programmed correctly. Thus a standard piece of equipment can be used for a very wide range of applications. It does not matter what the problem is, a computer is the answer. If this sounds too good to be true, take comfort in that it is too good to be true, but only just - the manufacturers can now make computers without knowing, or caring, what their machines will be used for and consequently they can be mass produced very cheaply. The microprocessor in your ORIC costs a mere £4 or so retrail. To make the thing into a machine like the ORIC costs a lot more, partly because a microprocessor needs extra bits and pieces attached to it to make it do anything remotely useful, and partly because of the programming language required. The ORIC is already programmed to understand a language called BASIC. This is an acronym for Beginner's All Purpose Symbolic Instruction Code. After you have spent a few hours playing with your computer, using this supposedly simple language, you will realise how long it takes to write a program that really works as you want it to. Time is money.

What is a program? The word has been used several times already and has not yet been defined. Programming and computers are inseparable and in some ways indistinguishable. A program is a series of instructions which must be obeyed exactly. There is no choice. Thus a recipe might read:

"take 1 egg.
now beat it.

While this can be called a program it is undesirable because it is ambiguous. To avoid this computer programs are written in special languages each word or character of which is defined exactly. At one time, in England, French was used exclusively as the legal language because it was, and still is, a language where every word is defined by a committee as having a certain meaning. Consequently, no ambiguity.

The language the ORIC uses is also defined exactly, as it must be, and the

computer will always do exactly what it is told. If your program gives the wrong answer, be assured that it is your program which is at fault. The Americans have characterised this phenomenon with typical transatlantic brevity as GIGO. This stands for Garbage In, Garbage Out.

2.1 Immediate Commands.

A computer can be made to obey commands in two ways. The commands can be part of a program which the machine is following, or they can be typed in on the keyboard followed by (RETURN).

PRINT is a very useful command. For example type in:-

PRINT "HELLO THERE"(RETURN).

Where (RETURN) means 'press the RETURN key'. The computer will respond by printing 'HELLO THERE' on the screen. The PRINT command has been obeyed immediately and for this reason it is referred to as an 'immediate' command.

Practically all the ORIC's commands can be used in immediate mode. Multiple commands can be typed in by ending each one with a colon. For example:-

ZAP:EXPLODE:PING:SHOOT(RETURN)

will result in all the pre-programmed sound effects being produced.

2.2 BASIC Syntax.

The internal language must be used according to its own grammatical rules. This is demonstrated by using the PRINT command in a simple program:-

```
10 PRINT "HELLO THERE"
20 END
```

When typing this in end each line with (RETURN). This program will not actually do anything until the magic command RUN (RETURN) is typed. Then the program is obeyed and 'HELLO THERE' is printed out again.

Notice that each line starts with a number. This is to help both you and the computer to keep track of where you are. Line numbers are all part of the language.

BASIC demands that all programs are written as a series of statements or commands split up into lines and that each line be numbered sequentially. In some earlier BASICs the first character after the line number had to be a space, although this is not so on the ORIC.

Syntactically this is represented as:-

XXspSTATEMENT (RETURN)

where XX=line number.

The second line of the program tells the computer to stop. This may seem rather strange, but some of the earlier versions of BASIC have to be told this, or they will try to execute whatever follows the last statement as if it were part of the program.

The results of such an attempt can be catastrophic. The ORIC is a bit more clever than that and does stop at the end of a program whether or not there is an END statement.

You can find out what the current program looks like by typing LIST (RETURN).

The response to this command is to print out the program as it was typed in. The LIST command can be modified for editing purposes as LIST 20 (RETURN) and only line 20 will be printed. It can also be included in a program so that the program runs and then lists itself.

When writing test programs it is a good idea to use the command NEW between each test. This command tells the computer to forget all about previous programs and start again. This prevents any lines of an old program being included in a new one. This command can be included in a program so that when the program is run, at the end of the run it clears itself out of memory and cannot be listed.

In any program it is considered obligatory to put in statements saying what the program is doing. These statements are called 'remarks' and must either be prefaced by REM or you can just type in '.

```
100 REM ROUTINE TO FIND AREA
110 A=B*H/2 'HALF BASE TIMES HEIGHT
120 RETURN
```

Putting in remarks seems an awful waste of time and space. But, when a program is unearthed after a month or so because it is needed to solve a current problem the lack of remarks in it is absolutely criminal. There is a special command which is only usable in immediate mode called 'EDIT'. This command uses a line number as its argument and allows immediate editing of any line in a program. To use this command type, for example, EDIT 10 (RETURN) where the line number must be a valid one. The standard copy facility can now be used (↑A) to copy those parts of the line that are to be kept, the arrow keys can be used to skip those parts which are not wanted and new parts can be added as normal. Note that if you overtype the old line, it is what you see on the screen that will be entered as the new line.

2.3 The PRINT and INPUT Commands.

When our simple program was run, you may have noticed that the small white block called the cursor moved to the start of the next line after the printed 'HELLO THERE'. The computer has actually had to obey two extra commands to do this.

The first of these is 'CARRIAGE RETURN', or more usually just 'RETURN'. This

command moves the cursor to the beginning of the line that it is on. The second is 'LINE FEED' and this moves the cursor down one line. If an attempted LINE FEED would result in the cursor moving off the bottom of the screen then instead of moving the cursor down the computer moves the text on the screen up. This is called 'scrolling'.

If the simple program is altered to:-

```
10 PRINT "HELLO THERE";
20 END
```

the semi-colon tells the computer not to execute the RETURN and LINE FEED. When RUN the cursor is now left at the end of the printed text.

The PRINT command makes the computer talk back. The INPUT command allows the user to talk to the computer. Another example:-

```
10 PRINT "GIVE ME A VALUE FOR X"
20 INPUT X
30 Y=X*2
40 PRINT "2 TIMES ";X;"=";Y
50 END
```

Line 10 makes the computer tell you what it wants. This is the beginning of interactive programming. If the program said "GIVE ME A VALUE FOR X PLEASE" it could be called 'user friendly'. Line 20 makes the computer accept the entry from the keyboard as the variable X. If something other than a number was entered in response here, the computer would not understand and would give an error message as "REDO FROM START". The program tells the computer to expect a number and when it is given something else it tries to deal with it as a number, gets into difficulties and has to give up.

Line 30 tells the computer to take the number just entered, multiply it by 2, and save it as the variable Y.

Line 40 looks complicated at first, but taken one part at a time it is straightforward. This is how the computer will interpret it:-

PRINT (send what follows to the screen).
" " (everything between these is sent to the screen as is)
; (do not send RETURN or LINE FEED)
X (print the current value of X)
; (do not send RETURN or LINE FEED)
" " (everything between these is sent to the screen as is)
; (do not send RETURN or LINE FEED)
Y (print the current value of Y)

A sample run of this program would give:-

```
GIVE ME A VALUE FOR X
? 5
2 TIMES 5= 10
```

Notice the ? requesting an input from the keyboard. This is generated by the use of the command INPUT. Also notice that it would have been better to put another ';' at the end of line 10 to get the '?' on the same line as the question. In fact the INPUT command can be used to do the job of lines 10 and 20 as:-

```
10 INPUT"GIVE ME A VALUE FOR X";X
```

And the result will be the same as line 10 with a ';' and line 20.

The description of how the computer decoded the PRINT statement in line 40 demonstrates the power of the computer and its vulnerability. The power lies in the fact that the way the computer behaves is determined by the program it is processing. Changing line 40 can completely change the way the computer presents the information. However, if the computer meets something in line 40 which is out of context (like answering FRED when a number is requested) it cannot handle it and the program is stopped and an error message printed. The program is said to have 'crashed'. More on this later.

Two commands exist which can only be used as part of a PRINT statement. These are SPC and TAB. Both of these commands have one argument.

The SPC command tells the computer to print the required number of spaces starting at the current cursor position. For example PRINT SPC(10);"FRED" would result in 'FRED' being printed 10 spaces to the right.

Sadly there is an error in version 1.0 of the ORIC which renders the TAB command virtually useless. In fact it does nothing with arguments less than 13 and then behaves as the SPC command. However this has been corrected in version 1.1.

There is one other peculiarity in V1.0 which is that the PRINT A,B type of command will not give the expected results, probably because TAB does not work. Again version 1.1 has corrected this.

There are two more commands which allow entry from the keyboard. These are GET and KEY$. Our short program could be modified to:-

```
10 PRINT "GIVE ME A VALUE FOR X"
20 GET X
```

and this would behave as before except there would be no '?' to prompt you. The differences between GET and INPUT are: the INPUT command can be used with text; generates the '?' prompt and waits for the entry to be terminated by (RETURN): the GET command cannot be used with text; does not generate a prompt and reports each key as it is pressed.

The KEY$ command also only requires a single keystroke entry. It will read in the

character corresponding to any key that is pressed as it is pressed. However, unlike INPUT and GET, KEY$ does not halt program execution until a key is pressed. To demonstrate this try these two short programs:-

```
10 GET A$
20 PRINT A$
30 GOTO 10
```
and
```
10 A$=KEY$
20 PRINT A$
30 GOTO 10
```

When the first program is run, the PRINT command is only obeyed after the RETURN key is pressed. To stop this program you will either have to switch off or use the hidden button underneath the machine. The second program however prints out each character as it is entered and it can be stopped by typing CTRL-C. There is an extra command in V1.1 which is PRINT @ X,Y;STRING where STRING represents any valid argument for the PRINT command. The effect of this command is to print STRING on the screen starting at the absolute co-ordinates X,Y. Thus PRINT @ 10,12;"HELLO" would give HELLO on the screen 10 spaces in from the left hand edge and 12 lines down.

2.4 Arithmetic Commands.

The ORIC computer has a full range of arithmetic operators and functions. You will see that '*' is used for multiply, '/' for divide and ' ↑ ' for 'raise to the power of'. '+' and '–' have their usual meanings. X is not used for multiply because it would be confused with the variable 'X'. Computers usually do arithmetic just as taught in school. The statement 3+4*5 will result in 23 as the answer. The statement (3+4)*5 will give 35.

Brackets are used as normal to prevent ambiguity. Indeed, the computer will not return two answers to a sum even if you think it is ambiguous. The computer has its rules and it follows them. For example the sum 3+12/6*2 could give the result 4 or it could be 7. To the computer the answer is 7. The computer will do division first, then multiplication then addition and subtraction.

The mathematical functions available are:-

ABS gives the absolute value of a function. That is the numerical value regardless of sign.

```
10 PRINT ABS(-7)
```

will result in '7' being printed.

ATN gives arctangent in radians. Translated this means the angle in radians whose tangent is supplied. Thus A=ATN(.7) gives a value for A of .611 radians. There being 2PI radians in a circle this is an angle of 35 degrees.

COS gives the cosine of an angle in radians. SIN and TAN give the sine and tangent of an angle respectively.

EXP gives the natural exponent of the number supplied. A=EXP(2) gives 7.389 etc. This is the same as the natural base 'e' squared.

LN gives the natural logarithm. This is the opposite of EXP.

LOG gives the logarithm to the base 10 rather than 'e'.

PI simply gives a numerical approximation to this constant.

INT gives the integer part of the number supplied.

SQR gives the square root of the number.

For games and other purposes there is a RND command which produces a pseudo-random number. It is called pseudo-random because it is not a truly random number as generated by ERNIE but has a numerical base and hence is predictable to some degree. Since a computer of a reasonable size would be needed to make the prediction this is not of any great consequence.

A=RND(1) will produce a random number between 0 and 1. In fact any number greater than or equal to 1 in the brackets has the same result. If 0 is used the last calculated number is returned. If a negative number is used the results cease to be random and the same number is produced for each negative number. So -1 will always give the result 2.99196472E10-8.

In some arithmetical problems it is necessary to know whether a number is positive or negative, or maybe zero but not necessary to know how big the number is. This function is achieved by the command SGN which returns -1 if the number is negative, 0 if the number is zero and 1 if it is positive. This sort of test might be used as a first step in finding the roots of a quadratic or higher order equation. For example in this equation:-

X↑2-3X+2=0

The problem is to find X. In the computer the equation is stored as a variable Y :-

Y=X↑2-3*X+2

and this would be calculated by a subroutine. For more information on subroutines see section 2.5.

```
1000 Y=(X↑2-3*X+2)
1010 RETURN
```

assuming that the values of X which satisfy this equation are known to lie between 0 and 10 (this is not unreasonable because in reality some knowledge of the problem would be avalable) we could then write:-

```
10 CLS
20 X=0:GOSUB 1000:A=SGN(Y)
30 REPEAT
40 X=X+.5
50 GOSUB 1000
60 UNTIL SGN(Y)=-A
70 PRINT X
80 END
```

If you RUN this program, you will find that this gives an answer for X of 1.5. Thus at this value Y has changed sign and hence 1.5 must be near a root. If line 40 is changed to X=X+.1 the program takes longer to run but comes up with the answer 1.1. This is leading towards a sucessive approximation technique for solving equations.

2.5 Subroutines.

A subroutine is a device for repeating a fixed section of code at any time in a program. One example has been shown already, admittedly rather trivial. Here is another simple example:

```
10 GOSUB 1000
20 B=A
30 GOSUB 1000
40 PRINT B;"TIMES ";C;"=";B*C
50 END
1000 INPUT "GIVE ME A NUMBER";A
1010 RETURN
```

The command GOSUB 1000 tells the computer to go to line 1000 and execute the commands it finds there as normal until it finds the command RETURN. It then must go back to the command following the GOSUB.

A large program should be made up of a series of subroutines which are called as required by the main procedure. There will sometimes be sections of program which are never repeated and these may be in the main procedure and not called as subroutines. However, it is often the case that a programmer has a subroutine already written from a previous program which does exactly what he wants and he will use this and call it as a subroutine even though it is only used once.

A more advanced type of GOSUB is the CALL command which tells the computer to go to a subroutine which is written in machine code. The CALL command specifies the start address of the subroutine which must end with an RTS instruction. (The machine code equivalent of RETURN). The machine code instructions have to be POKE'd into memory one byte at a time.

There is a special type of subroutine called FN which applies to arithmetical operators. The required formula is defined using DEF FN and can then be used as many times as required. Here is an example:

```
10 DEF FNA(X)=2*PI(X12)
20 FOR I=1 TO 10
30 PRINT "AREA= ";FNA(I)
40 NEXT
50 END
```

When you RUN this program, it prints out the areas of ten circles of radii 1 to 10.

More than one function may be defined and used at once. The above program could be extended to include:-

```
15 DEF FNB(X)=2*PI*X
35 PRINT"CIRCUMFERENCE= ";FNB(I)
```

and the circumferences and areas of the ten circles are printed out.

A subroutine may call another subroutine, which in turn may call a third and so on. This is known as 'nesting' subroutines and the ORIC will support nesting up to 16 levels. When nested subroutines are used there is a special command which allows the RETURN command to skip a level. To demonstrate this properly requires the use of the TRON facility which tells the computer to print out each line number as it executes it. By this means we can follow the execution of a program as it runs. Try this example:-

```
10 CLS
20 TRON
30 GOSUB 1000
40 IF B=10 GOTO 60
50 GOTO 30
60 PRINT B
70 END
1000 A=A+1
1010 GOSUB 2000
1020 RETURN
2000 B=2*A
2010 IF B<9 THEN POP
2020 RETURN
```

If you RUN this program, you will find that the printout is 30, 1000, 2000, 2010, 2020, 40, 50, 30, 1000 etc. finally reaching 70. At line 30 the first subroutine is called, which in turn calls the second. At line 2010 the test passes and so the return stack is POP'ed. Line 2020's RETURN therefore sends the program back to line 40 and not to line 1020. In this program line 1020 is only executed once, just before the program completes.

2.6 Looping

One advantage that computers have over mere mortals is that they will do the same thing over and over again very quickly and accurately without getting bored

and without making mistakes. There are two standard methods of making the computer repeat itself in BASIC. Firstly if the number of repeats to be done is known the FOR-NEXT loop is used.

```
10 FOR I=0 TO 10
20 PRINT "THIS IS LOOP";I
30 NEXT I
40 END
```

This program will print out the message eleven times, followed by the value of I for each time through the loop. The computer will interpret this program as follows. On seeing line 10 it will create a variable I and set it to zero. It will also set line 10 as the return address for future reference. At line 20 it will print the message and the value of I all on one line. At line 30 it will go back to the stored return address and add one to I, test to see if I is greater than 10 and, if not, it will go to line 20 and so on. When the test in line 10 results in I being greater than 10 the computer will not carry on through line 20 but will go to the statement following line 30.

Any variable could be used instead of I such as A or B or C. It is just tradition to use I as the counter in a loop.

As an aside it is quite permissible to have a counter which is stepped by non integer amounts. Thus a program might be:-

```
10 FOR I=0.5 TO 4.5 STEP 0.1
20 GOSUB 100
30 PRINT Y
40 NEXT I
```

In this program it is assumed that the subroutine at line 100 will calculate Y from I. Such a program segment might be used as part of a package to plot a polynomial.

There is a minor error in the ORIC's BASIC for step sizes less than one. The last run through the loop may be left out and the loop terminated sooner than normally expected. This is easily compensated for by setting the final value at one step size greater.

The second method of setting up a loop can be used when the exact number of times the loop has to be executed is not known, but the condition for exit is. This is a REPEAT-UNTIL loop.

```
10 REPEAT
20 X=X+1
30 A=POINT(X,Y)
40 UNTIL A=-1
```

This program segment tests points across a screen line until it finds one which is in the current foreground colour.

REPEAT-UNTIL loops can be nested in the same way that subroutines can, and

there is a similar command to POP to skip one level. This is the PULL command and its use is demonstrated in the following program segment which again uses TRON to show what is happening.

```
5 TRON
10 REPEAT
20 A=A+1
30 REPEAT
40 B=2*A
50 IF B<10 THEN PULL
60 UNTIL B>9
70 UNTIL B=10
80 PRINT A
```

If you RUN this program, you will find that the trace shows execution as 10,20,30,40,50,60,20,30,40,50,60,20 etc. The first execution of line 20 will result in setting A to 1 because on seeing A for the first time the ORIC will give it a value of zero. At line 40 B will be 2. At line 50 B is less than 10 so the repeat stack will be pulled. At line 60 B is less than 9 so the program will go back to the next repeat$address on the stack. Since one address has been pulled it goes back to line 20. This program does nothing useful except to demonstrate PULL.

2.7 LET.

The LET command is becoming obsolete in BASIC. Its original use was to tell the computer to assign a value to a variable. The statement LET A=10 told the computer to set the variable A to 10. In the ORIC the statements LET A=10 and A=10 do the same job.

2.8 READ,DATA and RESTORE.

READ and DATA statements are useful for programs which perform a fixed operation on a series of unrelated variables. An example might be a program to play a simple tune:-.

```
10 PLAY 0,0,0,0
20 READ A
30 PLAY 1,0,2,1800
40 IF A=-1 THEN GOTO 100
50 MUSIC 1,3,A,6
60 WAIT 20
70 GOTO 20
100 PLAY 0,0,0,0
110 END
200 DATA 1,2,3,4,5,6,7,8,-1
```

Tunes cannot be much simpler than this but it does demonstrate READ and DATA. The PLAY and MUSIC commands will be more fully explained later.

When dealing with READ and DATA commands it is sometimes necessary to go through the same data more than once. The READ command uses an internal pointer which is set to the start of the first line containing DATA as its first command. It is not usual to mix DATA with any other commands on the same line.The RESTORE command. sets the pointer back to its position before any READs were carried out. For example:

```
10 DATA 1,2,3,4,5,6,7,8,9,10
20 FOR I=1 TO 4
30 READ A:PRINT A
40 NEXT I
43 RESTORE
45 PRINT "******"
50 FOR I=0 TO 9
60 READ A:PRINT A
70 NEXT I
```

This program will print out the numbers 1 to 4 then a row of asterisks then the numbers 1 to 10. Any more READ commands will fail because there is no data left.

2.9 Testing and Branching.

A common requirement in programming is to test for a certain condition and to do one of two things depending on the result. FOR-NEXT loops and REPEAT-UNTIL loops both use tests in their operation. There is a test facility available to the programmer which can be used in almost any application. This is demonstrated below.

```
10 INPUT"WOULD YOU LIKE TO PLAY A GAME";N$
20 IF N$="YES" THEN 100
30 PRINT "O.K. BYE FOR NOW"
40 END
100 INPUT"HARD (H) OR EASY(E)";N$
etc.
```

Line 10 gets a response from the keyboard and line 20 makes a decision based on that input. If the answer is 'YES' exactly the program will jump to line 100. If the answer is anything else the program will go to line 30.

A conditional test can also be performed with numerical values in a program. For example:

```
10 IF A=0 THEN B=0 ELSE B=1
```

This statement sets B to one of two values dependent on the value of A.

If the program requires multiple branching then the best way to achieve this is through the ON-GOTO or ON-GOSUB commands.

```
10 INPUT"OPTION REQUIRED,0 TO 10";A
```

```
20 ON A GOTO 100,200,300,400,500,600,700,800,900,1000
100 PRINT "YOU CHOSE 1"
110 END
etc.
```

For the program to work properly there would obviously have to be lines 200,300, etc. all of which were valid. Line 20 will send the program down a specific path as chosen by A. If A=1 the program will go to line 100. If A=2 it will go to line 200 and so on. Note that if A is not an integer the program will only use the integer part of A. If in the example an answer of 1.5 is given the program will treat this as if it were 1.

In the case of ON-GOSUB the action is the same except that the program expects to meet a RETURN command and it will then come back to the next line after the ON-GOSUB command.

```
10 INPUT "GIVE ME A NUMBER FROM 1 TO 5";A
20 ON A GOSUB 100,150,200,250,300
30 PRINT "ALL DONE"
40 END
100 PRINT "YOU CHOSE 1"
110 RETURN
150 PRINT "YOU CHOSE 2"
160 RETURN
etc.
```

Thus the subroutine used is dependent on the entry provided.

2.10 PEEK, POKE and Relatives.

There are four commands which relate to memory inspection and change. These are PEEK, POKE, DEEK and DOKE all of which sound something like the nephews of a cartoon character. PEEK is exactly what it sounds like. It tells the computer to look inside a memory location and report what it finds. PRINT PEEK(49152) will give 76 because that is the start of the ROM controlling the ORIC.

POKE is the opposite to PEEK in that it allows the contents of a memory location to be set to a new value. That value must lie between 0 and 255 or the message ILLEGAL QUANTITY ERROR will be printed.

We can use POKE to change a location and PEEK to check that it has been changed. Type CLS (RETURN) to clear the screen and then POKE 48618,65 (RETURN). You should see an 'A' in the middle of the screen. Typing PRINT PEEK(48618) (RETURN) will result in 65 being printed out because that is the code value of the 'A' just POKE'd in. This is actually the decimal equivalent of the value stored in the byte at that location.

DEEK and DOKE do the same sort of things but to two consecutive memory locations at once. However, there is an added complication with these two commands. The microprocessor at the heart of the ORIC has one or two peculiarities. One of these is that it uses its addresses backwards. In this context an address is simply the number by which a location is known. Each address is assumed to require two bytes to fully describe its position. Even the location at the start of memory is referred to as address 0000. (This method of numbering may be unfamiliar. If so please refer to Chapter 4 for a full explanation).

Consider a location somewhere in memory with a general address of XYAB. the value XY is held in one byte and AB in another. The processor in the ORIC would refer to this address as ABXY and not XYAB. For this reason DEEK and DOKE reverse the order of the two memory locations when they operate on them.

To fully understand these two operators it is necessary to have an understanding of how the internal microprocessor actually works. DEEK and DOKE are not really of much use otherwise.

2.11 ASCII.

This is a method of representing alpha-numeric characters by numbers. When you type 'A' on the keyboard the computer actually receives the number 65. 'B' is represented by 66 and so on. Even the number keys on the keyboard are converted to their equivalent ASCII codes when they are entered. '4' would be received as 51. This may seem very strange at first but there are good reasons behind this apparent madness. To learn some of them the reader is directed to the appropriate section in Chapter 5. It is sufficient for now to know that this happens and that a number may be held in the computer as a number known as its ASCII code. For example:-

```
10 INPUT A$
20 PRINT A$
```

If in response to line 10 the sequence '12'(RETURN) is entered then '12' would be printed out but the computer would be holding in its memory the numbers 49 and 50 as the ASCII codes for '1' and '2' respectively. If instead the program were:-

```
10 INPUT X
20 PRINT X
```

and again '12' were entered, you might expect the result to be the same. However, this time the computer is holding 12 in its memory. To demonstrate that this is so try changing line 20 as :-

```
20 PRINT 2*A$ for the first and
20 PRINT 2*X for the second.
```

The first program now will not run because the computer cannot multiply strings.

How do you calculate 2*"HELLO"?

The ASCII codes 0 to 32 are reserved for control functions such as CARRIAGE RETURN (13), LINE FEED (10) and ESCAPE (27). Some of these codes are produced by special keys on the keyboard and more are available by holding down CTRL and simultaneously pressing another key. A complete list of such codes and how to produce them is given in Appendix A.

2.12 Arrays and Strings.

There are occasions when it is useful to be able to store numbers as a set, and to be able to access any member of that set. Such a set of numbers might be the ages of the children in a class. To handle such sets the computer can put these numbers into an 'array'. Consider the example below:-

```
10 DIM A(10)
20 FOR I=0 TO 10
30 INPUT "AGE= ";A(I)
40 NEXT I
```

Line 10 tells the computer to set aside sufficient memory for 11 variables in the array called 'A'. You might have expected only 10 variables but the first entry in the array is A(0) not A(1). The remaining lines read in 11 entries from the keyboard and store them as variables in the array. Now any entry in the array can be accessed by referring to its position in the array. For example A(5), and the position (5) may be the result of a previous calculation. This allows the construction of tables, such as train time-tables, and the required entry can be 'looked-up' by calculating its position in the array.

A string array is a 'string' of characters all held together as an array. An example we have already seen is:-

```
10 INPUT A$
20 PRINT A$
30 END
```

When this simple program is run, any printable characters can be entered. When RETURN is pressed the characters entered will be printed out. The length of A$ in this case is restricted to 11 characters.ORIC will have reserved a space for an array of 11 characters when it first met A$ in line 10, and any attempt to enter more than that will result in the message EXTRA IGNORED. In other words ,all the typing over and above 11 characters was wasted. To get a longer string ORIC must be told that the space is needed, and this is done using DIM. Modify the above program to include:-

```
5 DIM A$(20)
```

Now arrays of up to 21 characters can be entered directly. Due to the fact that the

ORIC has a buffer into which all characters entered from the keyboard are stored and that this buffer is only 78 characters long, we can only enter 78 characters into an array by this method. Strings can be added together using the '+' sign, called concatenating, and consequently strings longer than 78 characters can be built up. The absolute maximum length for an array is 256 characters. Obviously an array must be dimensioned by a statement similar to the one above, but with 255 instead of 20, in order to achieve this. Any attempt to exceed the dimensioned number of characters will result in the message 'STRING TOO LONG IN 250' or whatever the line number happens to be.

The CLEAR command will set all entries in arrays and strings to zero, or empty. When an array is dimensioned, space is set aside for it which will still contain whatever was last put into that space. The CLEAR command will clear out all this rubbish. This is necessary when building up a string or when using entries in an array to accumulate results of a calculation. The array must first be set to zero or the original rubbish will be added in to the answers.

2.13 CHR$, Concatenation and Other String Functions.

The CHR$ function (pronounced CARS) tells the computer to make up the ASCII character that corresponds to the value given. PRINT CHR$(65) will result in A being printed on the screen. CHR$ can be included in other strings by the use of '+'to concatenate or link strings together.

```
10 A$=CHR$(65)+"BILITY"
20 PRINT A$
```

and ABILITY will be printed out. This facility is useful for making up strings which are headed by control and/or escape codes to control what happens next. As an example try:-

```
10 CLS
20 P$="HELLO THERE"
30 B$=CHR$(27)+"W"
40 F$=CHR$(27)+"@"
50 T$=CHR$(27)+"L"
60 A$=B$+F$+T$+P$
70 PRINT A$
```

When run this program gives a flashing "HELLO THERE" in black on a white background. By changing the attributes in lines 30 and 40 any background colour and any foreground colour can be obtained.

One interesting effect is to change line 60 to:-

```
60 A$=B$+T$+P$
```

And when run, nothing is there. Now type INK 0 and the writing appears.

The other main string handling commands relate to string picking- a method of picking out just part of a string. Suppose a program required a YES/NO answer from the operator:-

```
10 INPUT"YES OR NO";A$
20 IF LEN(A$)>3 THEN 100
30 IF LEFT$(A$,1)="Y" THEN 1000
40 IF LEFT$(A$,1)="N" THEN 2000
100 PRINT "BAD REPLY"
110 GOTO 10
```

line 20 tests how long the reply is. Clearly if more than 3 characters have been entered the reply was not YES or NO. Lines 30 and 40 compare the first character in the string with Y and N respectively. If an equality is found the program branches to the appropriate place. If not it ends up in the error routine at line 100.

The command LEN simply returns the length of the string entered in response to the question in line 10. This has nothing to do with the DIM statement nor with any other commands that may set aside space for strings. The length of a string is calculated by counting characters until the character representing RETURN is met. There may well be space left over for more characters but this will be unused.

The LEFT$ command is part of a set of 3 commands used for string picking. The complete set is LEFT$, RIGHT$ and MID$. LEFT$ picks out characters starting from the left hand end of the string. In the example above, only one character, the first, was picked. Any number of characters may be picked up to the length of the string, and beyond. If a string only has 3 characters, asking for LEFT$(A$,4) will return only 3 characters.

RIGHT$ works in the same way as LEFT$ except that counting starts from the right hand end of the string. Thus RIGHT$(A$,3) of ASSAY would return 'SAY'.

MID$ is more complex having one more variable to be defined. This command returns the middle section of a string starting at a specified character and continuing for a specified number of characters. MID$(A$,2,2) of ARRAY would return 'RR'. The picking starts at the second character and continues for 2 characters. MID$(A$,3,5) of ASSEMBLY would return 'SEMBL'.

Lastly STR$ converts a numerical expression into an ASCII string. This command has some applications in print formatting. For example, having calculated an answer, it may be easier to combine its string representation with a description, as in this program:

```
10 A$="THE SQUARE IS "
20 INPUT"GIVE ME A NUMBER";A
30 A=A↑A:B$=STR$(A)
40 P$=A$+B$
50 PRINT P$
```

By stringing together the items to be printed the exact position of each item on the screen can be predetermined and hence the display can be formatted correctly. The opposite of STR$ is VAL. This returns the numerical value of the string. For example A=VAL(I$) would put the numerical value of I$ into the variable A.

The number of significant figures used in the answer can also be controlled by the use of LEFT$. If line 30 is changed to

30 A=A!A:B$=LEFT$(STR$(A),3)

the result will be printed out as N. only. The decimal point takes up one character position leaving only two for numbers and one of those seems to be lost in the conversion. If instead of the 3 a 5 is substituted answers will be printed as N.NN .

2.14 WAIT.

Sometimes programs run too quickly for the operator to see what is happening. To prevent this the WAIT command can be used to slow things down. WAIT has an argument (N) which has a base of 10 milliseconds. A command such as WAIT 100 would result in a delay of 1 second. For example this program goes too quickly to see what is happening:-

```
10 CLS
20 REPEAT
30 I=I+1
40 PRINT SPC(I);"*";
50 PRINT CHR$(#0D);
60 UNTIL I=28
70 END
```

This program is difficult to watch because of the speed of operation. Insert the line 45 WAIT 10 and when RUN everything is clear.

2.15 Graphics

The ORIC computer has 3 ways of controlling the display. These are the TEXT mode, two LORES modes and the HIRES mode. When you first switch on the ORIC the machine starts up in the TEXT mode. In this mode the standard character set is used to display information to the user, and only a small amount of memory space is needed.

The two LORES modes might be described as chunky graphics modes. The difference between the two modes is merely which character set is in use. LORES 0 uses the standard set and LORES 1 the alternate one. The graphics available in both LORES modes are based on pre-defined characters and hence little memory is used up.

The HIRES mode is the high-resolution graphics mode and this allows line

drawing and point plotting with a resolution of 240 dots by 200.

There are a number of special commands which allow the user to make full use of these modes and these are listed in Table 2.1

Table 2.1

TEXT	LORES	HIRES
INK	INK	INK
PAPER	PAPER	PAPER
PLOT	PLOT	CHAR
POS	POS	CIRCLE
SCRN	SCRN	CURMOV
		CURSET
		DRAW
		FILL
		PATTERN
		POINT

TEXT and LORES use the same group of commands and so we shall look at these first.

The INK and PAPER commands simply change the colour of the foreground and background colours respectively.

The PLOT command is used to position a character string using X and Y as the coordinates of its starting point. Note that the left hand column is reserved for background colour information and thus the next column is called column 0. The following example positions an asterisk in the tenth column of row 20 and lists itself.

```
5 CLS
10 LORES 0 ' OR TEXT
20 PLOT 10,20,"*"
30 LIST
```

The POS command is rather strange. The syntax to make it work is A=POS(C). Where C can actually be any valid alphanumeric character. This command simply returns the horizontal position of the cursor regardless of which row it is on. To print the value directly use PRINT POS(C).

SCRN(X,Y) returns the ASCII code at co-ordinates X,Y on the screen. If when in text mode no character exists at those co-ordinates the value returned is 32 this being the ASCII code for a space. In LORES no character is represented by 16, other ASCII codes remaining the same.

HIRES commands

The remaining graphics commands are restricted to HIRES mode. The CURSET command positions an invisible cursor on the screen so that character printing can begin at that point. Note that the top left hand corner of the printed character is put at the cursor position. This command has 3 arguments: the first is the horizontal coordinate to move to: the second is the vertical coordinate: the third is what to do on arrival at the specified point. This last argument can be 0,1,2 or 3. 0 means convert the point to the current background colour: 1 means convert it to the current foreground colour: 2 means invert the colour of the point: 3 means do nothing. Inversion in this context means if the point is in the background colour, convert it to foreground and if in foreground convert it to background.

The CHAR command prints a character at the current cursor position. This command also has 3 arguments. the first is the ASCII code for the character to be printed; the second is the character set to use and the third is the same as for CURSET. The following short program demonstrates both these commands by filling the screen with asterisks.

```
10 HIRES
20 FOR I=0 TO 199 STEP 8
30 FOR J=0 TO 230 STEP 7
40 CURSET J,I,0
50 CHAR 42,0,1
60 NEXT J
70 NEXT I
```

The CURMOV command means move the cursor relative to its present position by the amounts specified. Again this command has 3 arguments and the last one is the same as for CURSET. The first two arguments are the horizontal and vertical displacements respectively. Thus CURMOV 7,0,2 means "move the cursor 7 dots horizontally, 0 dots vertically and invert the point arrived at".

DRAW also has 3 arguments, the third being as before. The other two determine the relative position to which a line must be drawn. Try:-

```
10 CURSET 0,0,0
20 DRAW 230,190,2
```

This should draw a diagonal line across the screen. If DRAW -230,-190,2 is now executed only some of the line is erased due to digitisation error.

The PATTERN command is linked to the DRAW command in that it controls the type of line that is drawn. When the ORIC is switched on DRAW produces a solid line because the pattern register is set to all ones. The PATTERN command is used to change the contents of the register.

```
10 HIRES
20 CURSET 0,0,0
```

```
30 PATTERN $55
40 DRAW 230,199,2
```

This will draw a dotted line because the pattern register is set to 01010101. Add to the above:-

```
50 CURSET 115,100,0
60 CIRCLE 50,2
```

and as well as a dotted line there is a circle of radius 50 center 115,100 also drawn dotted. A further modification is:-

```
10 HIRES
20 CURSET 0,0,0
40 DRAW 230,199,2
50 CURSET 115,100,0
60 FOR I=50 TO 1 STEP -2
70 CIRCLE I,2
80 NEXT I
```

this shows the errors due to digitisation very clearly as a cross.

FILL is a command for setting up fixed pattern areas on the screen. The full command is FILL B,A,N where A is the number of character cells to be filled, B is the number of rows to fill and N is the pattern to use. N must not exceed 127. The command acts as from the current cursor position.

```
10 HIRES
20 FILL 1,40,127
```

produces a solid white line across the top of the screen because 40 character cells (full screen width) have all been filled. Changing the last argument will change the pattern drawn across the screen in the same way as PATTERN works for the DRAW command.

```
10 HIRES
20 FILL 5,10,$55
```

produces a band of vertical black and white lines, 5 rows deep by 10*6 dot positions wide. The $55 sets up the pattern of 01010101 and the other parameters determine the height and width.

The FILL command will always affect a rectangular area of the screen, but sucessive uses of the command combined with CURMOV or CURSET can produce other shapes provided they are made up of individual rectangles.

The POINT command has two arguments, and is used to test whether or not the point at the co-ordinates X,Y is in the current background or foreground colour. A=POINT(100,100) would give A=-1 if the point at 100,100 is 'lit' and 0 if it is not.

2.16 Sounds

The largest single components in the ORIC are the printed circuit boards and the loudspeaker. Because the speaker is so large the sounds that the machine can produce are very realistic.

The sound commands include four immediate commands PING, ZAP, SHOOT, and EXPLODE and each produces the appropriate noise. The remaining three commands can be used to generate either noise based sounds like ZAP or purely musical sounds or a mixture of both.

The PLAY command determines the settings for both SOUND and MUSIC. Its syntax is:

PLAY TE, NE, EM, EP

TE stands for TONE ENABLE and can have the following values.
0= no tone channels
1= channel 1
2= channel 2
3= channels 1 and 2
4= channel 3
5= channels 3 and 1
6= channels 3 and 2
7= channels 3,2 and 1
NE stands for NOISE ENABLE with values:-
0= no noise
1-7= noise on all channels
EM stands for ENVELOPE MODE and can have values 0 to 7 to select the required envelope.
EP stands for ENVELOPE PERIOD

this is a number between 0 and 32767 which controls the period of the envelope waveform. The larger this number is the longer the time between envelope repeats.

The length of each note or chord or sound is determined by a separate WAIT command. Sounds can only be stopped by the program executing another PLAY command.

The MUSIC command gives pure tones only and has four arguments.
MUSIC CH, OC, N, V
CH stands for channel number
OC stands for octave (0 to 6)
N stands for note (1 to 12)
V stands for volume (0 to 15)

Only if the volume parameter is set to zero will the current envelope be used.

The SOUND command can give noise or musical tones. It has three

arguments.

SOUND CH, PE, V

CH stands for channel (1 to 6). 1 to 3 for tones as before, 4 to 6 for tones plus noise.

PE stands for period. This is the period of repetition of the sound and hence is the inverse of the frequency.The larger this number the deeper the sound.

V stands for volume and behaves exactly as it does in the MUSIC command.

2.17 The New Cassette System.

There are some sorely needed additions to the V1.0 commands now available. In V1.0 all that could be done was to CSAVE "FILENAME"[S[where the, S is optional selecting the slower baud rate, and to CLOAD "FILENAME"[,S]. Both commands could be used with addresses to indicate that an area of memory was to be saved or loaded. Thus CSAVE"DATA",A#B000,E#B100,S would save the contents of memory from #B000 to #B100 at the optional slow baud rate under the name "DATA". If "DATA" is then loaded it is put back into the address area from which it was originally saved. Lastly V1.0 allows the specifier AUTO meaning that when the program is loaded it is to be run on completion of a successful load.

In V1.1 there are the following extra specifiers to the CLOAD command.

V will verify that the program on the tape is the same as the one in memory. When in use, the computer will display " Verifying...NAME B" on the status line. (The status line is the text line in inverse video above the normal text screen).

Because of the way some tape recorders work, particularly those with automatic volume control, the new loading routine may well detect an error in the load during the synchronisation period and although the computer will state ERRORS FOUND after the load is complete, the program will be found to be perfectly alright. However AUTO-RUN programs will not run after loading if any errors (real or apparent) are detected. To correct this, a short machine code program has been produced which is listed in Appendix F. If this program is POKE'd in and then saved as a header to an AUTO program it will disable the error routine which stops programs auto-running.

CLOAD "NAME", J will append the program called NAME to the end of the program currently in memory. However, the two programs must not have common line numbers. The program resident in memory must be numbered such that its highest line number is lower than the lowest line number of the program being appended.

Lastly there are two new commands which allow the contents of an array to be saved and loaded. Firstly, STORE A,"NAME"[, S] where the, S is optional, would save array A, which must have been previously dimensioned, under the filename "NAME". Secondly, RECALL A,"NAME"[, S] will reload the array with its variables.

When a file is being loaded the cassette system now displays a message as "Loading...FILENAME B" on the status line of the display. The 'B' is the identifier of a BASIC file. 'C' is used to identify a machine code file, 'R' for a floating point data array, 'I' for an integer array and 'S' for a string array. The use of the identifier restricts the length of a filename in V1.1 to 16 characters.

CHAPTER 3

Inside the Oric

We will now look at the way the ORIC is built and the various bits and pieces used to make it and we will take a closer look at some of the facilities available to the user. Beginners may find some of this chapter incomprehensible due to the unavoidable technicalities involved. Do not despair, you do not need to understand this chapter to use the ORIC's facilities.

Where appropriate we will use hexadecimal notation in this chapter, and all numbers expressed in this form will be preceded by '#'. If you have never met this number system before please refer to the section in chapter 4.

3.1 The Main Chips

The ORIC computer has been designed around three main chips. Firstly the 6502 central processing unit (CPU) which is the brains of the organisation. Next a special purpose logical gate array which was designed specifically for this computer. This chip is made by California Devices Incorporated in Silicon Valley. Thirdly a 6522 versatile interface adapter that gives ORIC the ability to talk to the outside world. The CPU chip used is common to a number of microcomputers and may owe its popularity to Microsoft BASIC which was written for this chip some years ago. A version of Microsoft is used in the ORIC.

3.2 The Resident BASIC and Operating System.

These two live in the main ROM at the top of memory. Since they are in the same chip the separation between them is mainly philosophical. It is not possible to remove just the BASIC and plug in another language, but it is possible to remove both together, as we shall see later.

3.3 The 6502 Microprocessor

This is an eight bit machine which is widely used. At present it is still the market leader of all microprocessors. It is fairly easy to program but does not have as wide a range of instructions as some other microprocessors. Its address capability is 64K, (note that 'K' here means binary thousands not decimal ones, hence

64K=2116); and no special provisions have been made for input or output. Thus any device to which the microprocessor talks or listens must emulate an area of memory. The standard crystal frequency is 1MHZ and its cycle time is 1 microsecond.

3.4 The 6522 Versatile Interface Adapter.

This device is used to transfer information in and out of the ORIC. It drives the sound generator (AY-3-8912) and the printer port, and is used as a time base and as a keyboard interface.

It appears to the microprocessor as 16 sequential memory locations. These locations are the registers of the 6522 and it is through these registers that the 6522 is controlled and that data is passed to the outside world and back.

The 6522 Versatile Interface Adapter is a complex chip and the reader is directed to the data published by the manufacturers for full information. However the way ORIC uses the printer port and Timer 1 is of interest here. On power up the 6522 registers are set up as shown below. (Note all numbers are in hexadecimal).

ORB	BF:BE:BD:BC:BF:etc.
ORA	---
DDRB	F7
DDRA	FF
T1L-L	10
T1C-H	27
T1L-L	10
T1L-H	27
T2L-L	-
T2C-H	-
SR	-
ACR	0:40
PCR	FF:DD:FD:DD
IFR	40
IER	7F
ORA'	0E:7F:0E:BF:OE:DF:etc

In fact the 6522 registers ORB and ORA' are continually updated in the interrupt routine.

Looking first at the Peripheral Control Register it is finally set to #DD. The data sheet shows that this means the outputs CB2 and CA2 are both held low and that the inputs CA1 and CB1 will each detect a low to high transition.

The Auxiliary Control Register is set to #40 which translates as continuous interrupts from Timer 1, which was loaded with #10 in T1-LOW and #27 in T1-HIGH. Timer 1 will be decremented at the system clock rate and will generate an interrupt on reaching zero. At this time the timer is automatically reloaded with the contents of the latch registers and it starts to decrement again whether or not the

interrupt has been acknowledged. Thus a consistent timing is maintained regardless of the load on the processor. As an item of passing interest the timer is loaded with #2710 in hexadecimal notation. This is equal to 10,000 decimal. See chapter 4 for conversion from hex to decimal. Since the system clock for a 6502 is nominally 1016 cycles per second this would result in an interrupt every 10mS. During the interrupt routine the ORIC reads any keyboard character that is input and stores it in a location in memory to be picked up later. (i.e after the interrupt routine has ended).

There is a second timer in the 6522 which is available to the user. This timer normally counts down at the system clock rate of 1MHZ and can cause an interrupt when it reaches zero. Normally that interrupt is not enabled and the counter runs on indefinitely. It can be stopped by POKEing 196 into location #30B, reset to any desired value up to 65535 by writing to locations #308 and #309, started by POKEing 64 into #30B and its current value read from locations #308 and #309 where #308 is the less significant byte and #309 the more significant. Since this timer can only hold a maximum figure of 65535 and this is decremented at a rate of 1016 counts per second it takes just 65mS to reach zero. To expand the capabilities of this timer its interrupt must be enabled. This involves intercepting the normal interrupt routine, testing if the interrupt is Timer 2, taking appropriate action if it is and passing the interrupt back to the ORIC routine if it is not. Timer 2 can only operate as a one-shot timer; in other words it must be reset every time it causes an interrupt. If this does not happpen future interrupts are automatically diasbled.

3.5 The Gate Array.

This chip was specially designed for the ORIC computer and is made by California Devices Incorporated. There are two good reasons for using such a device in a computer. Firstly it can replace a large number of standard integrated circuits and thus bring down the cost of the final product if a sufficiently large number are made. This chip replaces about 90 standard integrated circuits. Secondly, and possibly more importantly in these days of piracy and rip-offs, the chip is only available from C.D.I and only available to ORIC. This means that anyone wanting to make and sell cheap copies has a very hard time ahead of him trying to figure out what this chip does and how it does it.

This device handles all video generation and control as well as refreshing and controlling the dynamic RAM, It also takes an interest in the memory map of the ORIC, allowing the map to be changed under certain conditions. We will deal with these changes later.

3.6 The Memory Map.

A memory map is simply a diagrammatical representation of the way in which the designers of a computer have allocated the available address space. The map in Figure 3.1 shows how ORIC's memory space is split up into the various sections needed by a computer.

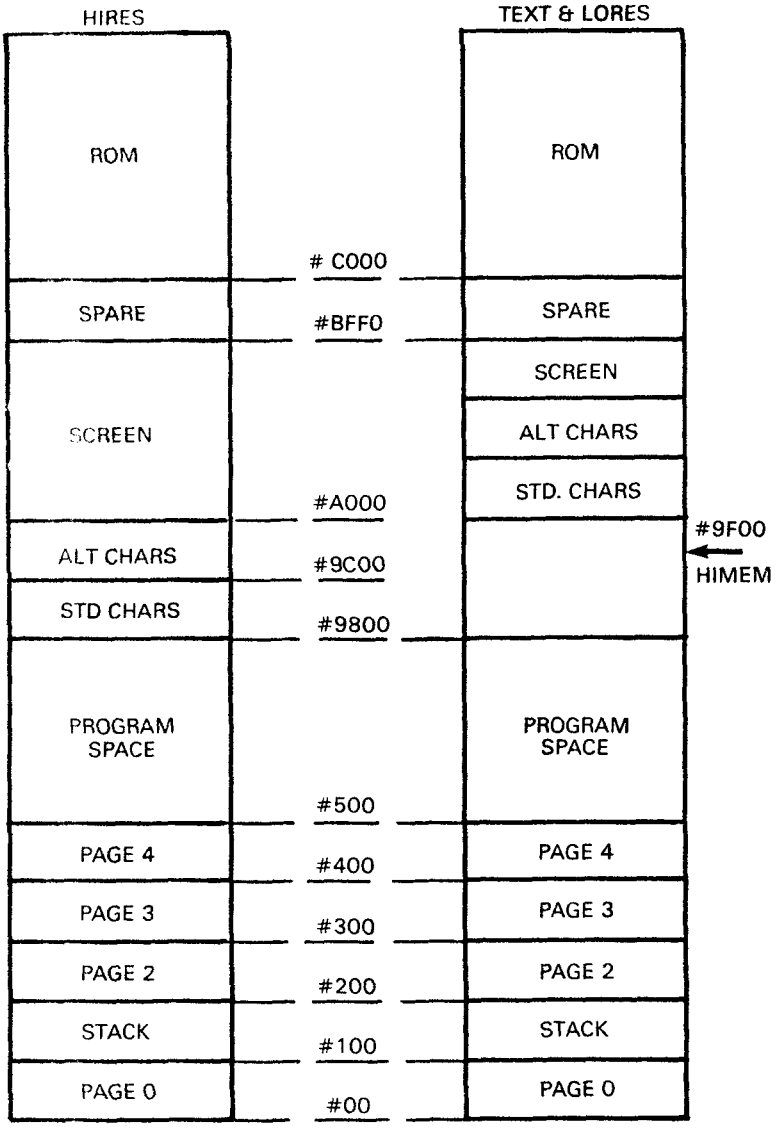


Figure 3.1 Memory Maps

Note that the whole of Page 3 in the memory map is given over to physical I/O addresses. The bottom 16 locations are in the 6522 already mentioned. The remaining 240 are at present unused. It is the designer's intention to use the lower section of Page 3 first leaving the upper section available for user's own devices. Thus ORIC produced hardware will bolt on from the bottom of Page 3 upwards.

There are three other main items of interest in the memory map. These are the two character sets and the screen area. ORIC stores both the standard and alternate character sets in RAM and hence both can be changed by the user at any time.

3.7 Character Sets.

The standard and alternate sets are loaded into RAM at switch-on from the system ROM. This may seem to be a waste of memory space since they must both be in there twice. There is an advantage to this method which outweighs the small disadvantage in terms of lost RAM. Both character sets are accessible to the user and can therefore be modified to suit the user's requirements.

VALUE	128	64	32	16	8	4	2	1	ADDRESS
8	0	0	0	0	1	0	0	0	#B608
20	0	0	0	1	0	1	0	0	#B609
34	0	0	1	0	0	0	1	0	#B60A
34	0	0	1	0	0	0	1	0	#B60B
62	0	0	1	1	1	1	1	0	#B60C
34	0	0	1	0	0	0	1	0	#B60D
34	0	0	1	0	0	0	1	0	#B60E
0	0	0	0	0	0	0	0	0	#B60F

Figure 3.2

The characters sets are produced by setting the appropriate bits in a section of memory. Figure 3.2 shows the memory locations for the character 'A' and which bits must be set to produce the character on the screen. It is obvious that there is a one-to-one relationship between the bits that are set and the dots produced on the screen. Zeros in the memory locations correspond to areas of the screen which are the same size as dots but which are left blank. Figure 3.3 shows a section of the screen and the locations in memory that control the characters printed there. To demonstrate the ease with which the character set can be changed, try this program:- (next page):

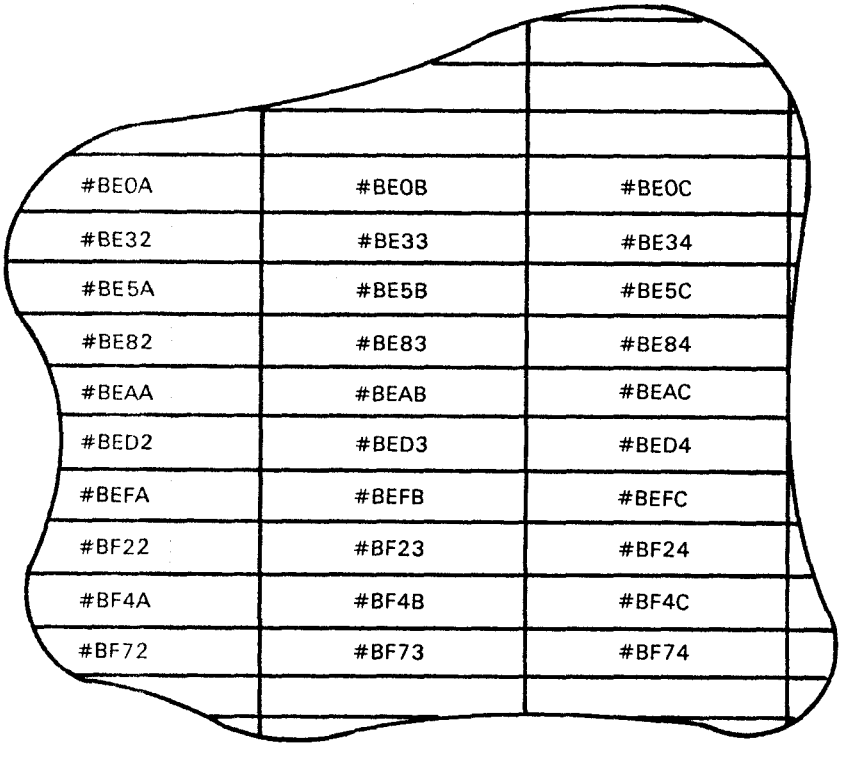


Figure 3.3

```
10 INPUT C
20 FOR I=0 TO 7
30 A(I)=PEEK(C+I)
40 NEXT I
50 FOR I=0 TO 7
60 POKE C+I,A(7-I)
70 NEXT I
80 END
```

Run the program then input the base address of a known character cell, say 46600 as this is the base for 'A'. Now all 'A's printed will appear upside-down. Any character can be inverted. Add some lines to the above program.

```
10 FOR C=46600 TO 47104 STEP 8
75 NEXT C
```

The program now takes a little time to run and as it does so the characters already on the screen will be seen to invert until eventually the whole lot are upside-down.

VALUE	128	64	32	16	8	4	2	1	ADDRESS
3			0	0	0	0	1	1	#B608
18			0	1	0	0	1	0	#B609
18			0	1	0	0	1	0	#B60A
30			0	1	1	1	1	0	#B60B
30			0	1	1	1	1	0	#B60C
30			0	1	1	1	1	0	#B60D
63			1	1	1	1	1	1	#B60E
18			0	1	0	0	1	0	#B60F

Figure 3.4 "Railway Engine"

Running the program a second time will put things back to normal. Even double height characters are inverted by this technique because they are produced by the same methods as single height.

To define special characters, the bit pattern needed has to be worked out. This is most easily done using squared paper and treating each square as a dot. Figure 3.2 shows the capital 'A' and the values this dot pattern represents.

Unfortunately V1.0 ignores commands of the form POKE A,#NN where # represents a hexadecimal number. This has been corrected in V1.1. In V1.0 it meant that the values had to be entered in their decimal form, which is not as convenient. Also notice that the character set is made up of characters cells 6 dots wide by 8 dots high and that only 5 dots are used horizontally and 7 vertically. this gives a one dot separation vertically and horizontally between characters.

Figure 3.4 shows a couple of special characters and the numbers needed to produce them. These characters are inserted as 'A' and 'B' in the following program and the resultant train is animated.

VALUE	128	64	32	16	8	4	2	1	ADDRESS
0			0	0	0	0	0	0	#B610
63			1	1	1	1	1	1	#B611
18			0	1	0	0	1	0	#B612
18			0	1	0	0	1	0	#B613
30			0	1	1	1	1	0	#B614
30			0	1	1	1	1	0	#B615
63			1	1	1	1	1	1	#B616
18			0	1	0	0	1	0	#B617

Figure 3.4 (contd) "Railway Carriage"

```
10 DATA 3,18,18,30,30,30,63,18
20 DATA 0,3,18,18,30,30,63,18
30 I=0
40 REPEAT
50 READ A
60 POKE (#B608+I),A
70 I=I+1
80 UNTIL I=16
90 CLS
100 FOR J=30 TO 2 STEP -1
110 PLOT J,4,"ABBB  "
120 NEXT J
130 GOTO 90
```

the train is jerky in its movements across the screen because it has to move one character position at a time.

The alternate character set is only available in LORES 1 mode. To demonstrate it change line 100 of the previous program to:-

```
50 LORES 1:LIST
```

and RUN the program again. What you are now looking at is a listing of the program in graphics symbols instead of the usual alphabet. Given time you could learn to read this, since it is only a form of substitution code, but why bother when the computer can do it for you. Type CLS(RETURN) or use the hidden key to recover.

The memory map shows that screen memory is always from #BFE0 down whether in TEXT or HIRES modes. ORIC saves enough space in memory to be able to switch between the two modes quickly. Note, however, the position of HIMEM as shown in Figure 3.1. If you switch to HIRES mode after defining a string variable, the string will be corrupted by the alternate character set. If you must have strings and HIRES together, set HIMEM to #97FF to avoid this problem. When running graphics programs in a 16K ORIC care must be taken to avoid running out of memory.

3.8 Mapping The Screen.

The TEXT screen is shown in Figure 3.5 in terms of the memory locations it represents. Thus the memory location #BBA8 contains the ASCII code for the character at the top left hand corner of the usable area. This character is actually the background colour attribute for that line. The next character on the line will be the attribute for the foreground colour. Figure 3.6 shows the screen in HIRES mode with the memory locations mapped as before. The bottom three lines of text after the HIRES screen are not shown. The HIRES screen appears to be 320 dots wide (#28*8=320) but in fact it is only 240 dots wide. This is because only six

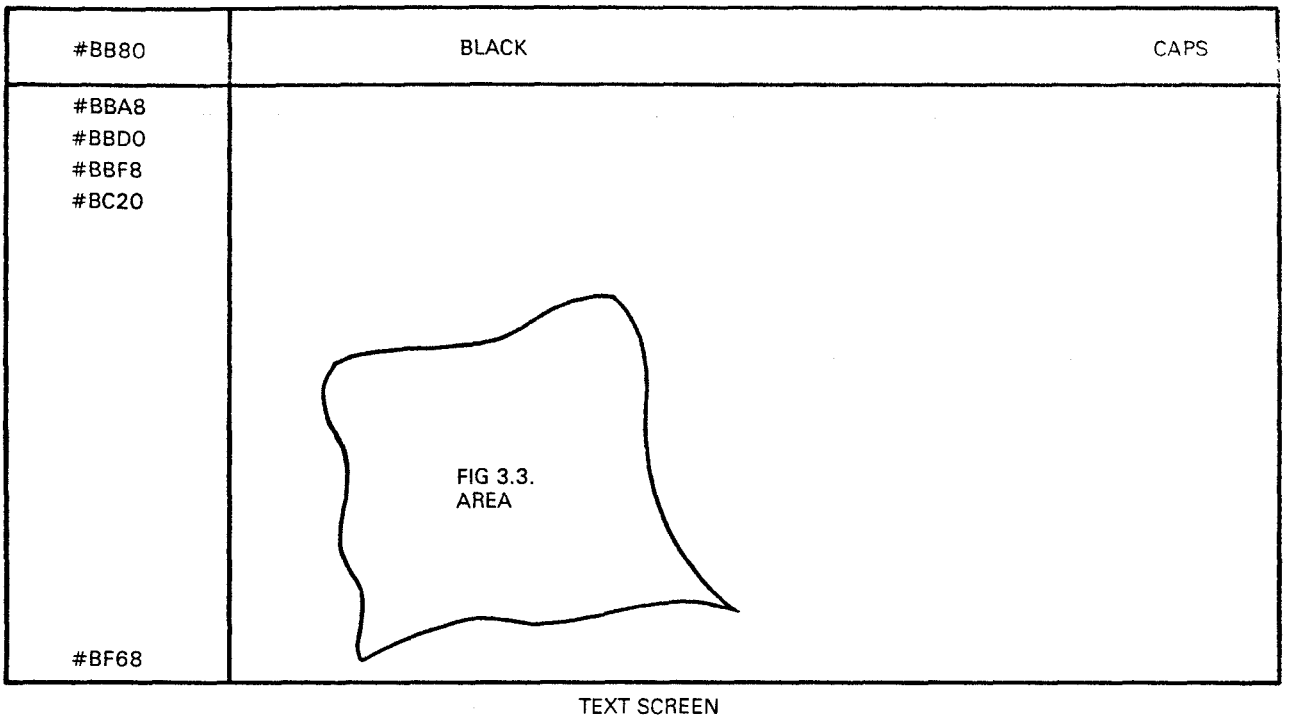

#BB80
BLACK
CAPS
#BBA8
#BBD0
#BBF8
#BC20
FIG 3.3.
AREA
#BF68
TEXT SCREEN

Figure 3.5

bits of each location are used as dots, the top bits being used to either invert or lay claim to the location being used as an attribute.

In HIRES mode no attributes are set on entry to the mode and hence all pictures are drawn in white on black as a default. Attributes may be used as in TEXT and LORES modes.

3.9 Screen and Colour Control.

The position of the cursor on the screen can be controlled from BASIC, or Assembley language by using the appropriate 'control codes'. These codes can be accessed from the keyboard by holding down the CTRL key and pressing the required character key simultaneously. Thus ↑I will make the cursor move forward one character space, called 'horizontal tab', but it will not delete any characters in the way as just printing spaces would. Instead any character the cursor lands on is inverted. The other facilities are ↑H for backspace, which behaves just as ↑I does except the cursor moves the other way, ↑J which moves the cursor down one line (line feed) and ↑K which moves the cursor up one line. There are also ↑L which clears the screen and puts the cursor at the top left hand corner, ↑M which acts as RETURN does and ↑N which clears the row the cursor is on.

All these facilities are available to the programmer by using PRINT CHR$(N); where N represents the numerical value of the character. To find the value of a control character, the sequence starts with A=1, B=2, etc all the way to Z=26.

There are in fact 32 possible control codes in the ASCII character set, so the remaining codes are equivalent to non alphabetic characters. From the ASCII character set in the ORIC manual these characters are seen to be (,/,),↑,#,@. Thus to clear the screen from within a program one can either use CLS or PRINT CHR$(12);. To move the cursor down one line use PRINT CHR$(10);.For horizontal tab use CHR$(9);. To move up one line use CHR$(11);. To backspace use CHR$(8);. Note that the horizontal tab (forward or backward) will 'wrap around', that is if they attempt to move the cursor off the edge of the screen to left or right, it will appear at the opposite edge one line up or down depending on the direction of travel, and will then be able to continue moving.

This gives a method of subsituting for the TAB command which does not work in V1.0. All that is needed is to use:-

```
10 FOR I=1 TO N
20 PRINT CHR$(9);
30 NEXT I
```

where N represents the number of character positions to be moved. This sequence could actually be called as a subroutine using a variable T to make it easy to remember. The program would now look like:-

```
NN T=3
NM GOSUB 1000
```

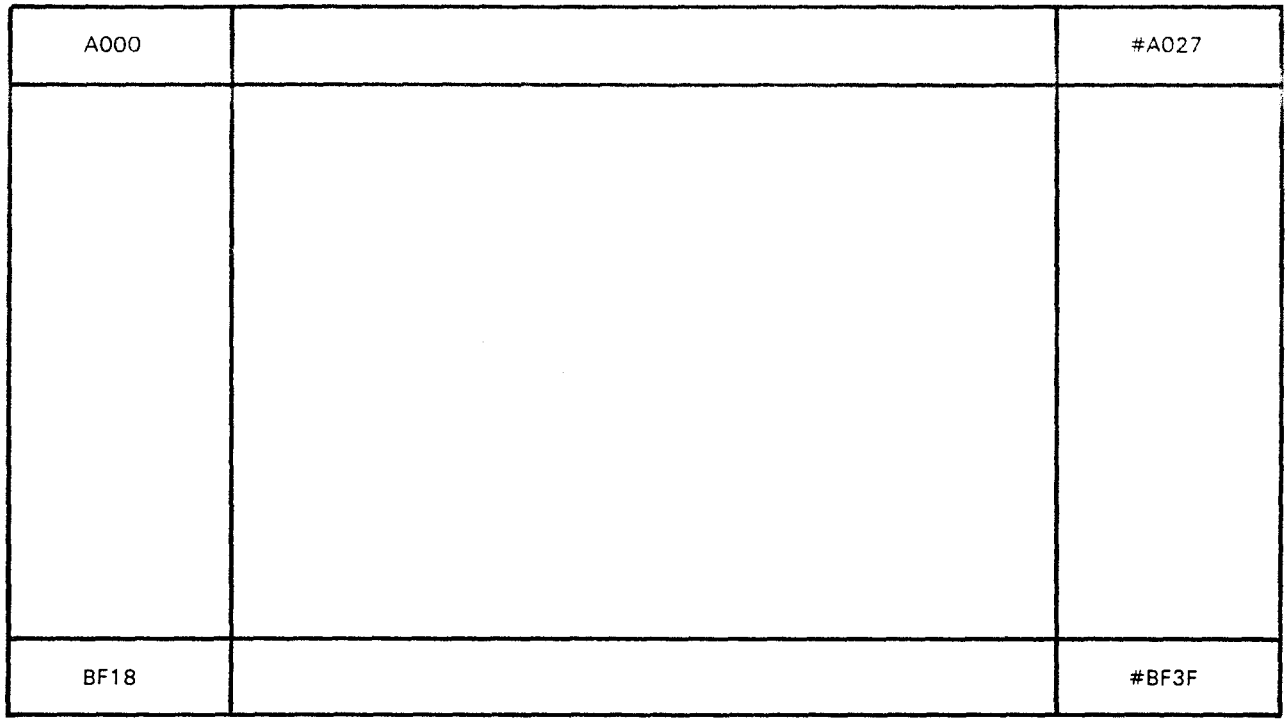


Figure 3.6

and at line 1000 there would be:-

```
1000 FOR I=1 TO T
1010 PRINT CHR$(9);
1020 NEXT I
1030 RETURN
```

A more ambitious, and much faster, solution is to use a machine code subroutine, but this requires much deeper knowledge both of the ORIC and of the microprocessor it uses. However Chapter 6 explains all about the machine code side and Chapter 7 gives the system calls.

This method of 'tabbing' can be expanded to cover any required cursor movements, up, down, left or right and is obviously very useful for formatting information on the screen.

ORIC's colour facility in all modes is based on serial attributes. These work as follows. All characters in screen memory can be considered as commands to the screen controller. If the contents of a screen location is a standard ASCII code the controller prints that character on the screen at that position. If however the location contains an attribute the controller obeys the command represented by that attribute. Thus the numbers 0 to 7 tell the controller to set the foreground colour to the appropriate hue. The numbers 16 to 23 have the same effect but on the background colours. The numbers 8 to 15 affect the character set used and whether or not the characters are to flash and/or be double height. The controller always continues to obey the last attribute met.

Colour In TEXT Mode.

In the TEXT mode the memory locations corresponding to the very left hand column of the screen are used to store background attributes. This column cannot be used for anything else and will be referred to as column −1. Any attempt to use this column for other purposes will fail because the ORIC expects to find background attributes here and treats whatever it finds as if it were such an attribute. The memory locations corresponding to the next column are used to store foreground attributes.

The commands INK and PAPER change these foreground and background attributes respectively. Any new attribute put on the screen by POKEing it in or by using ESCAPE codes will only have effect up to the end of the line it is on because at the start of the next line the embedded attributes will take over. This is a very simple and cheap method of obtaining a multi-colour display. The only drawback is that if it is necessary to have an attribute mid line, the attribute takes up one character space. Provided the attribute can be used in place of the normal space between words there is no problem.

The embedded attributes at the start of each line can be changed individually either by POKEing new values into screen memory at the appropriate locations or by using special control codes. These control codes are part of the ASCII set of codes but are non-printable. That is, there are no characters corresponding to

these codes, their use is to tell the device controlled to take some action other than printing a character. In this case the action is to place a serial attribute. The two codes used are those represented by the numbers 28 and 27. 28 is the code for [and 27 is the code generated by the ESCAPE key.

The serial attributes for the background colours in column –1 are the most difficult to get at without using POKE. In V1.0 the following sequence has to be used:-

PRINT CHR$(28);CHR$(27); CHR$(18)

and this will change the background colour of the line the cursor is on to green. The explanation of this sequence is as follows:-

CHR$(28) is a control code telling the ORIC 'here comes a control code sequence to change the background attribute of the current line'.

CHR$(27) means 'the character immediately following me is a serial attribute, not a character or other control code'.

CHR$(18) is the attribute.

In V1.1 the control characters cannot be successfully used from inside a program and they must be POKE'd in as if into memory locations. However they can be used from the keyboard.

The serial attributes for foreground colours in the next column can be reached by using the PLOT command. PLOT 0,N,C will change the foreground colour of the line N to the colour represented by the attribute C.

The other method of changing foreground attributes is by using ESCAPE again. If the sequence PRINT " "CHR$(27),"BTEXT"(RETURN) is typed in the word 'TEXT' will appear in green. This system only works if these characters are the first to be printed on the line whose colour is to be changed.

The explanation of the PLOT method is:

'0' means 'place in column 0' this being the first column which the PLOT command can reach; column –1 being used for background attributes.
'N' is the line number required.
'C' is the serial attribute to be plotted.

The PRINT sequence is not so easily understood. The space is only required in V1.0 to move the cursor into the text area. In V1.1 the cursor is already in this area.

The next character is ESCAPE and this tells the ORIC that what follows is a control character. The ORIC therefore treats the 'B' as a control character, for which it has a special routine, and the result is put into the foreground attribute column. If the space is omitted the output of the special control character routine would be put into the background colour column without ORIC having been told that a background attribute was coming. Thus the code for a foreground colour is put

into the slot for a background colour and the result is a black background. In V1.1 the situation is complicated by the cursor being in the text space already.

The PLOT command can be used to place foreground and background attributes anywhere on the screen. Thus the background colour of a line can be changed anywhere along its length, with the usual restriction that a printable character cannot take up the same space as an attribute.

To produce the special effects such as double height,alternate character set and flashing characters a second serial attribute has to be used. The attribute for normal standard characters is the default setting in TEXT and LORES modes. If this is changed to give flashing or double height it is usually reset when the embedded attributes are met at the start of the next line. The exception to this is double height. This is set by using ↑D or by PRINT CHR$(4), and it must be reset by CTRL-D again or by PRINT CHR$(4).

When stringing control sequences together to produce effects such as flashing double height characters, care must be taken not to overwrite a previous control code with the next one. The reason for this is that CHR$(27) or ESCAPE does not increment the cursor. (This is a short hand way of saying that the next character to be printed will go into the same location). Thus, if the sequence PRINT CHR$(4)CHR$(27)"N" were used it would produce a solid black line on the screen. Neither CHR$(4) nor CHR$(27) increments the cursor and so, one is printed over the other. Now use PRINT CHR$(4)" "CHR$(27)"NHELLO"and the result should be a double height flashing black HELLO. It may not be if the command resulted in the HELLO starting on an odd numbered line. If that is the case, simply scroll up by one line by typing RETURN until the cursor tries to drop off the bottom of the screen.

Add another control character to the above and make it PRINT CHR$(4) " "CHR$(27)"N"CHR$(27)"A HELLO"and the result will be a flashing red double height HELLO. Note that more spaces are not needed because there are characters in the print sequence which already increment the cursor. Another CHR$(4) will now have to be printed to prevent more double height characters being printed.

There is a table of attributes in Appendix A showing the colours and special effects available and the control codes required to produce them.

Colour In LORES Modes.

Both LORES modes put white characters on a black background as their default settings. The screen locations corresponding to the very left hand column (column–1) now contain the attribute for the character set to be used. In LORES 0 mode this will be the number 8 and in LORES 1 it will be 9. Thus the character set used can be altered by changing attributes. This change will only last up to the end of the changed line because at the start of the next line the embedded attribute will take over.

The remainder of the screen locations are filled with the number 16. This is the

attribute for a black background. Thus changing background colour by changing one location will only affect the background colour for that location. Multicoloured backgrounds are only achievable in TEXT or HIRES modes. However by using background attributes multicolour block displays can be produced in LORES modes.

The attributes in column-1 also have the effect of turning the foreground colour back to white. Try this program:-

```
10 LORES 1
20 PLOT 0,0,8
30 PLOT 1,0,1
40 PLOT 2,0,"HELLO"
50 PLOT 7,0,9
60 PLOT 8,0,"HELLO"
70 PLOT 0,1,"HELLO"
80 END
```

When RUN this gives a black background with a red 'HELLO' followed by some red block characters on lne 0. On line 1 there are the same set of block characters as line 0 but in white and lower down the screen some more block characters which actually represent 'Ready' and on the start of the next line the flashing white cursor. The program's actions are as follows:

Line 10 puts the ORIC into LORES mode using the alternate character set.
Lines 20 and 30 place two serial attributes in line 0. These attributes are '8' for normal character set and '1' for red foreground colour.
Line 40 puts 'HELLO' after these attributes.
Line 50 puts the attribute for alternate character set after 'HELLO' on line 0
Line 60 puts 'HELLO' after the last attribute on line 0
Line 70 puts 'HELLO' on line 1

Thus the alternate character attribute on line 0 has changed the character set only, but the attribute at the start of line 1 has changed the foreground colour as well.

Apart from these differences the LORES mode is exactly the same as the TEXT mode, even as far as producing double height flashing characters. However, note that in LORES 1 mode the attribute '0' must be used to produce double height flashing alternate characters. See Appendix 'A'.

Colour In HIRES Mode.

In HIRES mode the system is exactly the same except that characters are now only one dot high and PRINT and related commands cannot be used. Colour control codes have now to be POKE'd into memory. Again a control code, or attribute, will take up the space of a normal character. In HIRES each memory location is considered to be divided up as shown in Figure 3.7. If bit 7 is set all following bits are considered to be inverted. Consider the case with bit 7 always zero.

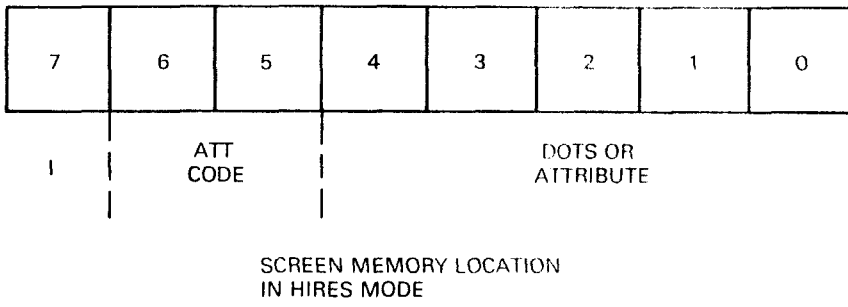


Figure 3.7

If bit 6 is set the location is used to put dots on the screen. Setting the appropriate bit will light that point. If bit 5 is set, but not bit 6, the location is still considered to be used to plot points and the point represented by bit 5 is lit. If neither bits 6 or 5 are set then the location is used as an attribute and bits 0 to 4 contain the code. See appendix A for the list of codes and colours.

Again if it is required to change colours mid screen no foreground information can be plotted over the position occupied by the attribute code. The foreground attributes that control flashing will still work in HIRES mode. Those for double height and the alternate character set behave as those for the standard characters, which is what might be expected for characters only one dot high.

3.10 Input and Output.

We will now look at the provisions made for using ORIC to control external devices or for it to be controlled by them. In order for this to take place there have to be ways of making ORIC talk to the outside world and of making it listen. Such ways are called input and output or just I/O for short.

The Printer Port.

Let's look first at the printer connection and examine how this operates an external printer. This output is taken from the internal 6522 chip already mentioned and the connections used are listed in Table 3.1. Notice that the whole of Port A is used for the data lines and that two extra lines are used for control. The method of operation is as follows:-

Table 3.1

PRINTER	PRINTER PORT CONNECTOR	CONNECTIONS 6522(PIN)	6522 (FUNCTION)
STROBE	1	14	PB4
D0	3	2	PA0
D1	5	3	PA1
D2	7	4	PA2
D3	9	5	PA3
D4	11	6	PA4
D5	13	7	PA5
D6	15	8	PA6
D7	17	9	PA7
ACK	19	40	CA1
GND	2-20 INC.		

When ORIC sends a character to the printer, it writes the required value to the port A pins of the 6522, then sets the strobe line low for a short time and waits for the Acknowledge line to detect a low to high transition. If no such transition is detected after 15 seconds or so, ORIC decides the printer is dead and puts up a message to that effect on the screen. It is only during the time that the strobe line is low that the data on the printer port is valid. If a program uses an LPRINT CHR$(#XX) statement the printer port pins will take on the value XX while the strobe line is low. A method of latching and acknowledging this data is shown in Chapter 5.

The Expansion Connector.

This connector gives access to most of the 6502's bus and control lines. The missing lines are, for those who are interested, SYNC, RDY, and NMI. There are some special purpose lines to aid interfacing to the ORIC and these are called MAP, I/O CONTROL and ROM-DISABLE.

I/O Line.

The I/O line is pulled low whenever a Page 3 address is decoded internally. This saves a lot of address decoding for the user because only 8 address lines need to be decoded to arrive at a unique location within a Page. Some methods for doing this are shown in Chapter 5.

I/O CONTROL.

I/O CONTROL allows the user to mask the internal 6522 out of the address space. This is done by pulling this line low. Thus connecting I/O to I/O CONTROL gives the user access to the whole of Page 3.

ROM-DISABLE.

The ROM-DISABLE line is used to disable the internal Read Only Memory. Pulling this line low allows external memory to take control. Since there is now no operating system the external program must be completely self supporting in whatever it does. This line gives ORIC the ability to call external routines by making an external access through Page 3. An external circuit recognises the address and pulls ROM-DISABLE low thus enabling its own program to be paged in. The advantage of this technique is that extra program memory can be added without taking up RAM space.

MAP.

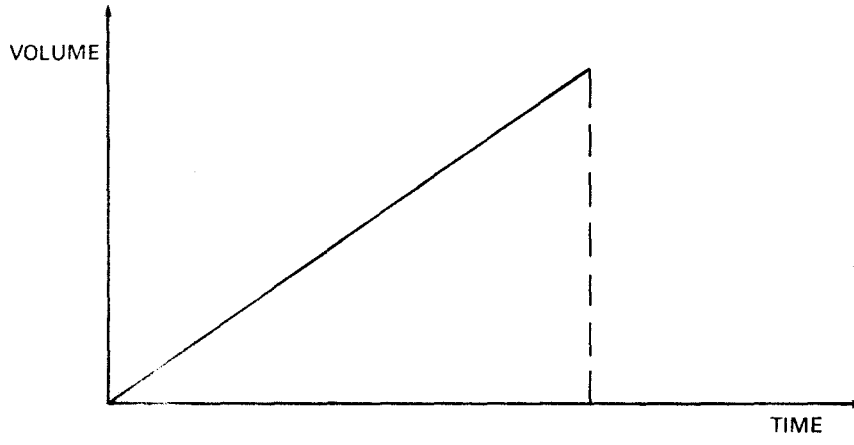
The MAP control is more complex since its operation is dependant on the current address. If MAP is pulled low when ORIC is accessing the top 16K of memory (normally ROM) the internal ROM is disabled and the internal RAM is enabled. If MAP is pulled low when the rest of memory is being accessed the internal RAM is disabled and external memory or any external device may take over. Thus external hardware can be mapped in anywhere in the RAM memory space. The MAP line could be controlled via Page 3 and hence allow ORIC to page hardware in and out of its memory map. The MAP line must be synchronised with Q2 and must go low at least 80nS before Q2 goes high and it must stay low for at least 250nS. See chapter 5 for circuit diagrams.

3.11 Sound.

The MUSIC, PLAY and SOUND commands were described in rather bald terms in Chapter 2. Now is the time to put some hair on these descriptions, so to speak. ORIC's sounds are generated by a special chip which can produce three distinct notes all at once. Thus this chip looks like three sound generators all in the same package. Each generator is controlled by a 'channel' in the sound chip. The word 'channel' is used because this section 'channels' commands to the appropriate sound generator in the chip. The sound generator to be used is selected by a channel number in a command.

Using the sound generator from BASIC is reasonably simple. The main points to watch are that once a sound has been started it can only be stopped by issuing a PLAY 0,0,0,0 command and that to use one of the available envelopes the volume section of the SOUND or MUSIC command must be set to 0 and the required envelope selected in the last two sections of the PLAY command.

We had better start by defining the term 'envelope' and then describe the differences between the seven envelopes available. The envelope of a sound is the variation of the volume of that sound with time. Thus a sound that starts off softly and builds up would have an envelope as shown in Figure 3.8. Add a sharp cut off and this is the same as the ORIC's envelope mode 2. The manual gives a pictorial representation of all the envelopes. Envelope 1 is the reverse of 2 in that the sound starts at a high volume and decays away. This type of envelope is said to have a strong 'attack', meaning that the initial increase in volume is very steep.



RAMP ENVELOPE

Figure 3.8

Both envelopes 1 and 2 are of finite length. That is to say that after their period has expired the envelopes are not re-started.

Envelope 3 is a repetitive sawtooth giving a sharp attack followed by a steady decay. The period of this envelope is the time between successive peaks of the envelope.

Envelope 4 is a steady rise followed by a steady decay. This could be used to simulate the 'beat' produced by two notes which are very close to each other.

Envelope 5 is a slow rise followed by a steady level. This envelope does not repeat and the period refers to the time taken to complete the rise.

Envelope 6 is the reverse of 3 giving a sudden rise (or attack) followed by a steady decay. The period is as for 3.

Envelope 7 is a linear rise followed by a steady level. Apart from the way the initial rise in volume is arranged this envelope behaves as 5 does.

The production of pure tones in each of three channels is the simplest starting point for practical demonstration. The program below brings in the 3 channels one after another and then turns them all off at once.

```
10 MUSIC 1,4,4,4
20 MUSIC 2,4,7,4
30 MUSIC 3,4,11,4
40 PLAY 1,0,0,0
50 WAIT 100
60 PLAY 3,0,0,0
70 WAIT 100
80 PLAY 7,0,0,0
90 WAIT 100
100 PLAY 0,0,0,0
```

Lines 10 to 30 inclusive define the note and volume settings for each of the 3 channels. Lines 40, 60 and 80 turn on the channels in order and lines 50, 70 and 90 control the time for which each chord is played. Finally line 100 stops the sound generation. Notice the peculiar numbering system used in the PLAY command to bring in the different channels. Line 40 brings in channel 1, line 60 brings in channels 1 and 2 and line 80 brings in all 3 channels. The reason the numbering system looks strange is because the computer is using the lower 3 bits in the number to control a channels selected. Thus bit 0 controls channel 1, bit 1 controls channel 2 and bit 2 controls channel 3. If all this still sounds like gibberish to you, read the section in Chapter 4 on number systems and, hopefully, all will be made plain.

This program can be modified to demonstrate the use of envelope commands.

Each PLAY command can change the envelope in use to produce special effects. Try this version of the same program:-

```
10 MUSIC 1,4,4,0
20 MUSIC 2,4,7,0
30 MUSIC 3,4,11,0
40 PLAY 1,0,4,100
50 WAIT 100
60 PLAY 3,0,6,200
70 WAIT 100
80 PLAT 7,0,7,900
90 WAIT 150
100 PLAY 0,0,0,0
```

The MUSIC commands in lines 10 to 30 now have their volume arguments set to 0. If this is not done the envelope sections of the PLAY commands will have no effect. Line 40 uses envelope 4 with a period of 100. This last argument is scaled in milliseconds. Thus the period of 100 gives a repetition of the envelope every tenth of a second. Line 60 selects envelope 6 with a period of 200, thus repeating every fifth of a second, and line 80 selects envelope 7 with a period of 900 giving a repetition of 0.9 seconds. Envelope 7 does not repeat and the period now refers to the time taken for the active part of the envelope to complete. The active part being in this case the part between minimum and maximum volume.

The SOUND command is capable of producing pure tones, like the MUSIC command, or noise or a mixture of both. Looking first at its use in producing pure tones, the following program gives very similar results to the original one using the MUSIC command.

```
10 SOUND 1,100,5
20 SOUND 2,120,5
30 SOUND 3,85,5
40 PLAY 1,0,0,0
50 WAIT 100
60 PLAY 3,0,0,0,
70 WAIT 100
80 PLAY 7,0,0,0
90 WAIT 100
10 PLAY 0,0,0,0
```

The first major difference is that the note actually played is determined by a number which is not related to a musical scale. This number is the period of the note. The following table and graphs (see Figure 3.9) are based the author's measurements.

Number	**Period**	**Frequency**
50	.8mS	1250 HZ
100	1.6mS	640.HZ
150	2.4mS	412.HZ
200	3.2mS	312.5HZ
250	4.0mS	250.0HZ

This shows a linear relationship between the number used in the SOUND command and the period of the note produced, as might be expected.

When noise is to be added to the sounds produced set the noise enable bit in the PLAY command and use one of the noise channels in the SOUND comand. The noise produced will take the frequency of the SOUND command as its base. To produce the best results from the noise facility it is necessary to make use of the envelope section of the PLAY command. Try these two short programs to hear the different effects:-

```
10 SOUND 4,150,0
20 PLAY 1,1,2,2000
30 WAIT 200
40 PLAY 0,0,0,0
```

and now change line 20 to:-

```
20 PLAY 1,1,1,2000
```

As an aside, try this program:-

```
10 SOUND 4,24,0
20 PLAY 1,0,1,1000
30 WAIT 200
40 PLAY 0,0,0,0
```

You should hear a sound which is fairly close to that produced by the PING command, but it is at a lower volume and does not have the same reverberation time. Obviously not all the sound generator's facilities are available to the BASIC programmer.

To make sound effects on their own, set the Tone Enable bit in the PLAY command to zero. This next segment makes an impact type noise.

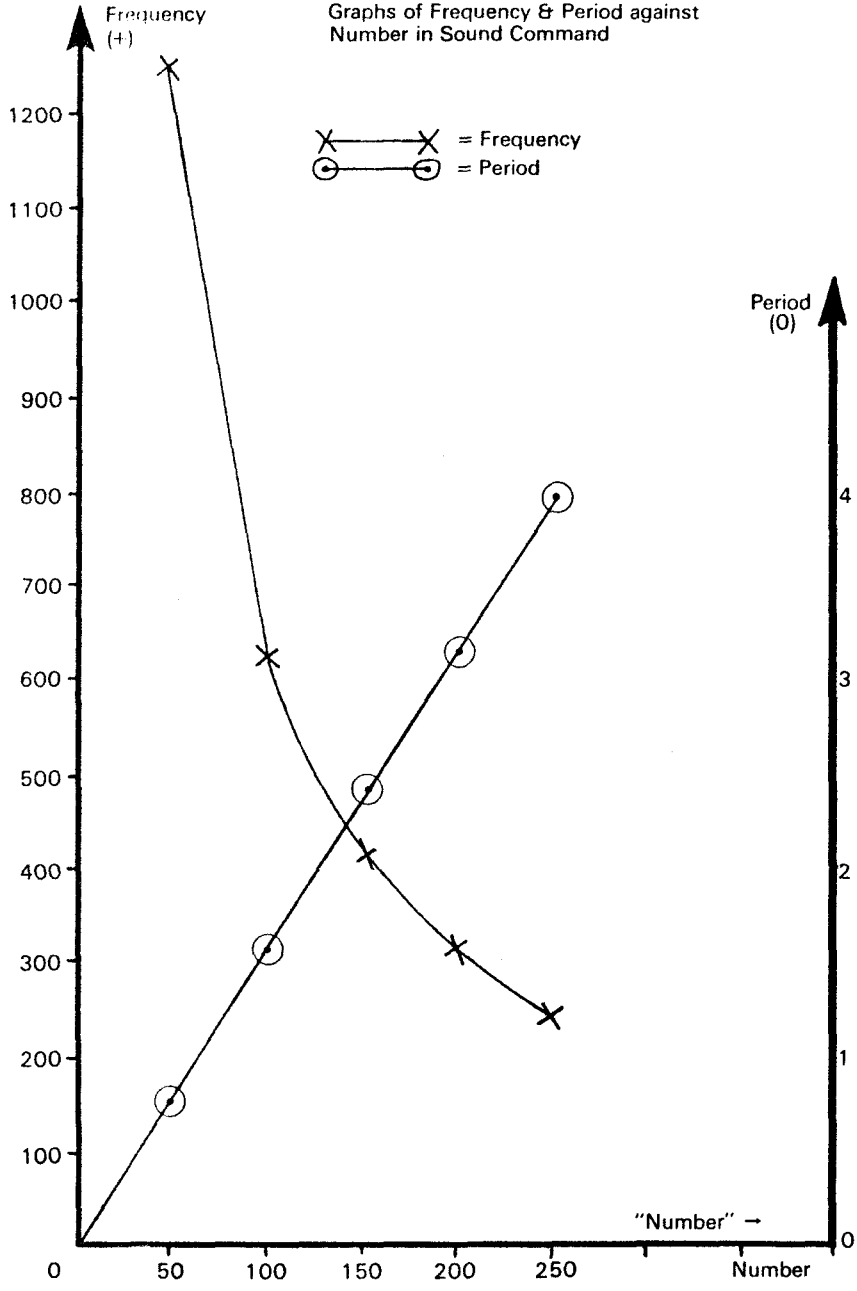
```
10 SOUND 4,300,0
20 PLAY 0,1,1,1000
30 WAIT 200
40 PLAY 0,0,0,0
```

Changing the middle parameter of the SOUND command in line 10 changes the base frequency of the noise produced. Using 600 instead of 300 gives a sound much more like SHOOT except that again the volume is lower and the attack is less. Numbers above 700 start to produce high frequency sounds again.

Notice that if the Noise Enable bit of the PLAY command is set then you will always hear noise in the output. That noise may not be what you expected to hear if you have not used a channel number greater than 3 in a previous SOUND command. If a noise has been defined by correct use of a SOUND command, it will stay defined until changed by another SOUND command using the same channel number. If the following program is entered:-

```
10 SOUND 4,600,0
20 PLAY 0,1,1,1000
30 WAIT 100
40 PLAY 0,0,0,0
```

and RUN, and then line 10 is changed to MUSIC 1,4,4,0 and the program RUN again, no difference is discerned. However, press the hidden Reset button and RUN the new program again. Now you will hear the difference.


Graphs of Frequency & Period against Number in Sound Command
Frequency (+)
= Frequency
= Period
Period (0)
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
4
3
2
1
0
"Number" →
0
50
100
150
200
250
Number

CHAPTER 4

How Computers Think.

This chapter is a general one on how computers work. It is hoped that after reading, and understanding, this chapter you will have all the necessary knowledge of computer hardware and behaviour to be able to tackle the more advanced topics dealt with in Chapter 6.

The first sections of this chapter are given over to number systems because there is a need to understand them if you intend understanding the internal workings of computers.

We shall then look at how a computer executes a program in some detail and finally describe the way a high level language like BASIC works and compare it to other possible methods of implementing a computer language.

4.1 The Binary System.

The word Binary means 'based on 2', and the arithmetic associated with this system can therefore only concern itself with devices that have two possible values. The convention is to use the values of 0 and 1 to represent the two possibilities. In contrast to this, decimal arithmetic is based on 10 and the devices in this system can have any one of 10 possible values (0-9), 0 being the first and 9 being the tenth. A single digit in the decimal system can therefore have any value between 0 and 9 and in binary arithmetic a single digit can have the values 0 or 1 only.

To a computer, binary arithmetic is the easier system. The computer's memory can be thought of as being made up of a large number of switches which are either on or off. If a switch is on, then that represents a 1, if it is off it represents a 0. To work in decimal arithmetic, each switch would have to have 10 possible positions and this makes for very cumbersome switches.

In order to be able to deal with numbers larger than 1 the computer strings together groups of switches. Traditional groupings are 4, 8, 16 and 32 allowing the machine to handle numbers up to 15, 255, 65535, and 42706225 in one go. The ORIC uses a grouping of 8 switches, so we shall concentrate on this arrangement. From now on we shall use the more usual name for these switches and call them BITS. Tradition has it that the word BIT is a contraction of Binary Digit but it is equally likely that the word came first, as being a bit of information, and the explanation came later.

Binary arithmetic uses position to determine the value of a digit in exactly the same way that decimal arithmetic does. The very right hand digit is the least significant, or smallest, and the very left hand digit is the most significant, or largest. To give an example, consider the decimal number 123. The 3 at the right hand end means 3 units, the 2 in the middle means 2 tens and the 1 at the left hand end means 1 hundred. The digits used are multipliers, then, for the value of the position they occupy. The position values are arrived at by using a power series of the numerical base of the system. In the decimal system the base is 10 and so the power series is 10↑0, 10↑1, 10↑2, and so on. Hence we obtain the position values 1, 10, 100 etc.

In the binary system the base is 2 and so the power series is 2↑0, 2↑1, 2↑2, 2↑3, and so on giving position values of 1, 2, 4, 8, 16 etc. Thus, the binary number 10 is actually 0 units and one 2 giving 2. In decimal, this same number represents 0 units and 1 ten.

Any binary number can be evaluated by writing down the position values above the digits and then adding up those values which have a 1 beneath them and ignoring those with a 0. For example, consider the binary number 10101010. This actually represents the number 170 in decimal notation. To demonstrate that this is so let us use the method described.

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

We add up the position values which are above a 1 and obtain 128+32+8+2=170.

The computer stores its numbers using the same convention of least significant on the right and most significant on the left. This is only a convention and it would be possible to do things the other way round.

There is one other convention which ought to be mentioned at this point. It is customary in the decimal system to start writing down a number with the first non zero digit. Thus although we may have to add 29 to 123 we only write 29 not 029. This is called 'leading zero suppression'. When dealing with binary arithmetic where the system in use has a fixed number of bits (as is the case with an eight bit machine such as the ORIC) it is customary to write down the values for all eight bit positions even if they are zero. Because binary notation is being used a letter 'B' is written down after the number to indicate this. Thus the number we first used would be written down as 10101010B.

Binary Arithmetic.

Now that we know what a binary number is and how to convert it into decimal, we can look at how binary numbers are added and subtracted. Multiplication is also possible using multiple additions, but apart from that we shall make no mention of other arithmetic functions here, these being beyond the scope of this book.

When you were taught arithmetic at school you may dimly remember chanting such things as 'one three is three, two threes are six, three threes are nine....' and so on. This sort of teaching is now unpopular with teachers, or so the story goes,

which is a pity because this is in fact the only way that we can learn how to do arithmetic with any speed. Even addition in the decimal system has to be learnt off by heart. If you are asked to add 9 to 4 you will answer 13 not because you have 'worked it out' but because you have learnt and remembered the relationship between these three numbers in just the same way as you learnt your tables. You can, of course, work out the sum 9+4 from first principles but it would take too long to do and is extremely difficult to do in your head. To demonstrate this, how do you prove that 9+4 is 13 and not some other number?

This leads on to another advantage of the binary system as far as computers are concerned. Instead of having to remember a large number of possible additions, only three are needed. The first is 0+0=0 which may seem rather trivial but must be defined for the computer, or it will not know what to do. The second is 1+0=1, and the third is 1+1=10. If this last looks like heresy, remember that it is really a binary sum expressing the same result as the decimal sum 1+1=2.

Let us now look at eight bit addition, using what we already know.

```
 10101010          170
 01010101 +         85 +
 --------          ---
 11111111          255
 --------          ---
```

The numbers 10101010 and 01010101 are said to be complementary because where one of them has a 1, the other has a 0. Adding a number to its complément will always produce 11111111B. In case that was too easy, try this one:

```
 10101010          170
 00001000 +          8 +
 --------          ---
 10110010          178
 --------          ---
```

In this sum, the two 1's in the fourth column from the right add together to give 0 in column 4 and 1 to carry into the next column. There was only a 0 to be added to there and so no more carries were generated. Now try this one:

```
   10101010          170
   10000000 +        128 +
   --------          ---
1  00101010          298
   --------          ---
```

Here the only carry generated was from the left-most column (value 128) and

there are no more columns left to hold it. To human arithmeticians this sort of thing is no problem. We merely create an extra column, value 256, to hold the extra bit. A machine cannot do the same trick and has to have provision built into it to handle all such contingencies. The way the 6502 handles this is to store the extra bit in a separate memory location which is part of the Status Register. The programmer can then test the Status Register after each addition to see if such a carry bit was generated and he/she can take whatever action he/she deems appropriate. For more information on the Status Register, see Chapter 6. Here is a more tricky sum:

```
  10101010                    170
  01110110 +                  118 +
 ----------                  -----
1 00100000                    288
 ----------                  -----
  111111
```

This sum has an interesting situation in the 32's column. Here we have 1+1 plus a carried 1 from the previous column. This results in a 1 and 1 to carry; the rest of the sum is as before. Again a carry bit is generated into the 256 column and would be stored in the Status Register as before. As an aid to understanding, the carry bits have all been printed under the columns into which they carry to show the mechanics of the sum.

So much for binary addition. With a little practice it is possible to become as proficient in this as in the decimal variety. We shall now turn our attention to subtraction, which should hold no terrors for us since we know how to do this in the decimal system and the same principles are used. We have already seen the sum 1+1=10 as the first step in addition. We can turn this round to demonstrate the first step in subtraction; 10–1=1. We can explain this in exactly the same terms as used in the decimal system.

```
  10
   1 -
 ----
  01
 ----
```

The reasoning goes as follows:- Subtract 1 from 0. Cannot do this because 0 is smaller than 1, therefore borrow 1 from next column, making it zero. Now subtract 1 from 10 to give 1 and the subtraction is complete because we have run out of digits to subtract. Notice that to do this subtraction we had to know that 10–1=1, so that in fact the demonstration had to be known in order for it to be done. This may look like a fiddle, but it is just the same as describing how to subtract 9 from 13 to get 4 because again the answer has to be known. This next sum is just as obvious:

```
  10101010                    170
  00001000 -                    8 -
 ----------                  -----
  10100010                    162
 ----------                  -----
```

In this simple subtraction we have merely removed the 1 which occupied the 8's column. Let's try another:

00001000	8
00000100 -	4 -
00000100	4

The example demonstrates that 10-1=1 in whichever column the subtraction is being performed.

In the decimal system, if a larger number is subtracted from a smaller the result is a negative number. This is indicated by the prefix '-'. In a computer there is nowhere to store such a prefix and so a different method of indicating a negative number has to be used if they are to be allowed. Also the system has to be flexible enough to let the programmer decide whether or not he wants to use negative numbers. The method chosen is to use the most significant digit as a sign indicator. If this is a 1, the number can be thought of as negative and if this bit is a 0 the number is positive. This means that 8 bits can represent numbers from +127 to -128. However, the system must allow a number to be positive or negative and yet still give the correct answer when it is added to another number. Let us look at this problem in a little more detail.

We have already used the number 10101010 and said that it was equal to decimal 170. But, this number has its top bit set to 1 and so could be considered to be a negative number. Suppose then that we said that this number was -42, this being the sum of the remaining values which have a 1 in their positions. Look again at the first addition which we did:-

10101010	170
01010101 +	85 +
11111111	255

and try it as the addition of a positive and a negative number.

10101010	-42
01010101 +	85 +
11111111	-43

But 11111111B would represent -127 using our system and hence this leads to an incorrect answer. Our system does not work, so we throw it out and try another.

We will stick to using the top bit as indicating that the number is negative. Let us expand on that and say that if that bit is set, it represents −128 and that all the other values will be positive and thus reduce the value of the negative number. By this method 10101010 would represent −128 + 42=−86. Now try our sum again.

10101010	−86
01010101 +	85 +
11111111	−1

And the sum is now −128+64+32+8+4+2+1=−1 so this system works. It also leads to the unlikely result of 11111111B being −1 but worse things happen at sea.

The fact that the sum of a number and its complement is −1, gives us another method of doing subtraction. It is a simple task to produce the complement of a number, so let us denote the complement with a bar over the top of the number like this:

$\bar{c}$

where c represents a binary number. We now have $c+\bar{c}=-1$. Suppose we now want to subtract one number from another. In other words we want to solve the equation a−c=x where, as usual, x is the unknown. From $c+\bar{c}=-1$ we have $-c=\bar{c}+1$ and substituting in the equation for c gives $a+\bar{c}+1=x$. So we can find x without doing any subtraction at all. We need to complement c, add it to a, add 1 to the result and that will give the answer.

The binary system just described can be used for 'multiple precision' arithmetic just as easily as for single precision. This means that a number can be stored in more than one byte, allowing the programmer to use 8, 16, 24 or even 32 bit arithmetic. In each case the top bit can still be used as a sign bit.

This leaves only one possible source of confusion. What happens if a large (comparatively large, that is) number has another number added to it, both numbers being positive, and the sum is such that it sets the top bit. If signed arithmetic is being used then the answer would look like a negative number which is clearly nonsense. To allow the programmer to test for this the 6502 has an overflow flag in the Status Register. If the top bit of the accumulator is set by an addition, then the overflow flag in the Status Register will be set to 1, indicating that although the number looks negative it is in fact positive. Programmers can test for this condition after each addition and take such corrective action as they deem necessary.

The B.C.D. System.

B.C.D. stands for Binary Coded Decimal and it is a system which combines both binary and decimal systems. The binary notation is used but numbers up to 9 only

are allowable. To represent the number 9 only 4 binary digits are needed, 1001 being the bit pattern. Thus two BCD digits can be packed into a Byte. This system is attractive to accountants because it allows the setting up of sums using a fixed number of significant digits in the computer and keeps track of each digit individually. In binary arithmetic the actual layout of the answer in decimal notation is not clear until all the computations have been completed and the answer is presented. With BCD each digit can be followed through all stages.

The BCD system uses the computer's memory in blocks of 4 bits, or Nibbles, one Nibble per digit. Thus the number 9 as we have already seen is 1001B; the number 10 in decimal would be 0001 0000B. This may seem a rather surprising result, but remember in BCD 9 is the maximum digit per nibble, so any number greater than 9 has to use two or more BCD digits. Thus 99 would be 1001 1001B and 100 would be 0001 0000 0000B.

We can now see that BCD is written down exactly as decimal numbers are except that each decimal digit is coded into its binary equivalent-- hence the name.

BCD Arithmetic.

This system is really just the same as decimal arithmetic except that the digits used are expressed in four bit binary code. For example the sum 9+6=15 would be

```
    1001        9
    0110        6 +
---------   -------
0001 0101       15
---------   -------
```

Had this sum been in binary the result would have been 111 because 6 is the binary complement of 9. To achieve the desired result in BCD we would then have to add 6 to that result, as shown below:-

```
    1111       15
    0110 +      6 +
---------   -------
0001 0101       21
---------   -------
```

By using this technique, we can simulate BCD addition by testing for a result being greater than 9 and if it is, adding 6 to it to adjust the result. Some microprocessors have a special instruction called Decimal Adjust which does just that. When using BCD with these processors the procedure is to do the arithmetic as normal and then Decimal Adjust the result to produce BCD.

The 6502 processor has the ability to do BCD arithmetic directly. In the Status Register there is a Decimal flag which can be set or cleared by the programmer. If

this flag is set the processor interprets all arithmetic operations as being BCD type. When the flag is cleared the processor will perform normal binary arithmetic. The manufacturers of the 6502 do not guarantee the state of any of the Status Register flags except the interrupt flag at power on or Reset and consequently it is good programming practice to set or clear the decimal flag as required before doing any arithmetic.

The Hexadecimal System.

This is the last number system we shall look at and we will only go as far as to describe how this system is used to represent binary numbers. It is possible to perform hexadecimal arithmetic but the effort involved is usually too great in comparison to the gains made. It is far quicker, and undoubtedly more accurate, to use the computer to perform any operations that are needed in this system.

The reason this system exists is to be able to represent all 16 possible states of a Nibble by a single character. To achieve this, the first 9 states are represented by the decimal numbers they are equivalent to and the states 10–15 are represented by the letters A to F to give a truly alpha-numeric system. The table below shows the way the systems compare.

Binary	decimal	hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

This system can obviously be used to describe the contents of a Byte by using two hexadecimal digits. Thus the number we started with in the binary system, 10101010B would be AA in the hexadecimal one.

Before leaving number systems and going on to other fields it is worth pointing out that now we have so many different ways of representing the same number it is imperative to state which system is in use. This is usually done by the use of prefixes or suffixes attached to the number itself. Thus a binary number would

usually end with a capital 'B' as 10101010B. Where all eight bits are used this may seem superfluous, but it is still a good habit partly because it keeps you in the way of stating the system used and also because meeting a number like 100 on its own could be very confusing. Decimal numbers are sometimes priviledged in that they alone have no suffix or prefix. Hexadecimal numbers are sometimes represented as 0AAH, for example, the leading 0 being required by some machines to tell it that a number is following and the H at the end tells it the system. A more common method is to prefix the number with either $ or #. The ORIC computer uses the # sign (called HASH) and so that is the method we shall use from now on. Our original number would therefore be written as #AA.

4.2 How Computers Are Made.

A program is a set of stored instructions. A computer is a device that is capable of executing those instructions. In order to be able to do this the computer must firstly be able to store the instructions, and so it must have a memory. Secondly it must be able to execute those instructions so it must have a means of decoding them and acting on them. The part of the computer that looks after these functions is called a Central Processing Unit or CPU. Lastly it must have a way of communicating the result, since an answer which is known only to the computer is all very well, but it does not provide any satisfaction to anyone. Therefore a computer must have Input and Output devices, usually lumped together under the term I/O.

This then describes our general purpose computer and a model is shown in Figure 4.1.

There are some extra parts of the model which we have not yet mentioned and these are all to do with the internal operation of the machine. For the computer to execute an instruction in the stored program it has to transfer that instruction into the CPU. This is done by the Data Bus. The memory has to be told which instruction is wanted and this is done by the Address Bus. Lastly, the memory has to be told whether the CPU is reading information, such as an instruction,from it or writing information, such as the result of a calculation, to it. This job is done by the Control Bus.

We now have a general picture of a computer with the data in the memory being read and written by the CPU with the various buses keeping all the parts serviced with the information required. These buses are actually just collections of wires, as many wires being used as there are bits of information to transfer. Thus the Address Bus has 16 wires because, in our machine, there are 65536 possible memory locations and, to give each one a unique address, requires that many bits. The Data Bus has only 8 wires because the data is moved around in Bytes. The control bus in this machine has only 2 wires because the 6502 only needs a timing wire to keep all peripherals in step and a Read or Write wire (R/W') to tell the memory whether the CPU wishes to read data or to write it. Other control wires are available on the 6502 but only these two are essential.

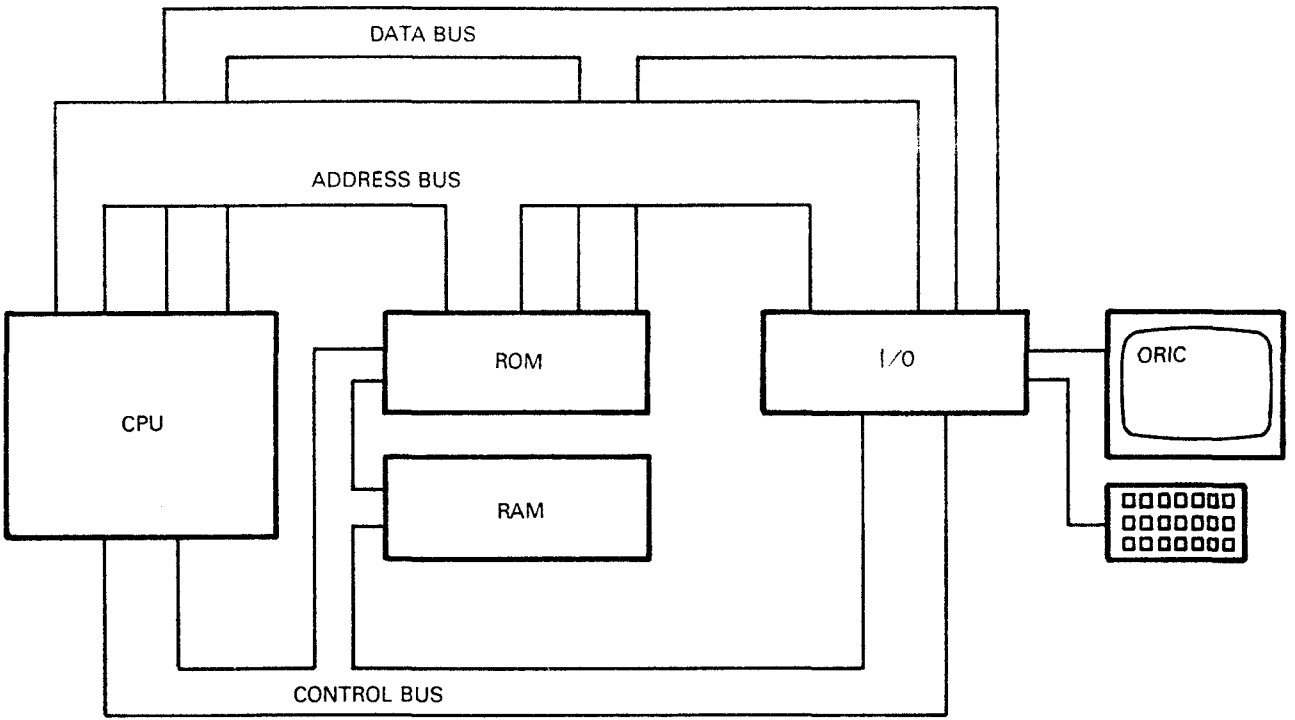


Figure 4.1

The I/O section depends on the application of the computer. General purpose machines, such as the ORIC, have a keyboard and an interface to a T.V. set or a monitor so that the user can enter information and receive answers on a screen. There may also be a printer for hard copy of some or all results and there may be disc drives or a cassette recorder for storage of the user's programs. Lastly there may be some specialist interfacing hardware to allow the computer to control some external device such as an oven or a small electric motor. Special purpose computers would only have the interfacing hardware required for their own special purposes and they would be programmed only to perform a fixed set of tasks. Such special purpose machinew might be washing machine controllers, engine controllers for up-market motor cars and lorries, ticket vending machines, arcade games and so on. The list of applications is practically endless because computers can control anything which can be controlled and usually do it better than anything else can.

4.3 Program Storage And Execution.

If we had a device which allowed us to examine the memory of a computer, we would see something like the picture given here.

Address	**Data**
#C000	#4C 59 EA 4C 75 C4 40 C9
#C008	#A4 C6 E3 CF E3 CF 8B CC

The data is presented in blocks of 8 Bytes. All the numbers used are hexadecimal. This appears to be a rather meaningless load of miscellaneous numbers, and to us that is what it is. But, to the 6502 processor this is a meaningful program and is the only sort which it can understand. This is what is known as 'machine code' and it is the lowest level, in terms of programming, to which we can sink. The BASIC language, which ORIC is programmed to understand, looks nothing like this and the 6502 cannot obey BASIC commands directly. It must have a program, writen in machine code, which it follows and which interprets the actions required by BASIC.

Writing programs in machine code, while possible, is not recommended. It is too fiddly, too prone to error and far too tedious for one's sanity. The preferred method is to write in a language called Assembly Language and then use a special program, called an Assembler, to convert that into machine code. This is equivalent to writing in machine code except that we now use a set of 'words' to describe what the machine is to do and the Assembler translates these into machine code for us. There is a one to one ratio between Assembly Language and machine code, the use of the former merely makes life easier for the programmer.

The segment of machine code which we looked at is stored in the computer's Read Only Memory (ROM) which means that this program cannot be altered and is there immediately we switch on, so that the machine has something to do straight away. If such a memory did not exist, the computer would have no program to execute when switched on and would sit there - figuratively speaking - for a very

long time! From this you may have deduced that a computer must always have a program to execute and that it is executing a program all of the time. Obviously, because we humans are so slow at doing things compared to the speed of the computer, it spends most of its time looking to see if we have done something and if not, looking again.

The program which the computer designers have provided with the machine is really in two parts. The first part is called the 'operating system', and this is concerned with such things as input, output and system timing. The second part, which makes use of the first part, is the BASIC language itself which is concerned with interpreting the commands which form the BASIC program typed into the machine by the user. In some machines it is possible to remove the BASIC language, leaving the operating system in place, and plug in another language entirely. The ORIC computer treats the two parts as one entity in this sort of operation, and they are either both in or both out.

Because the computer is executing a program all the time, it is worth mentioning exactly what it is doing. Let us imagine that the machine has just finished one instruction as we join the action. The sequence of events would be:-
1. Fetch the next instruction.

2. Add 1 to the instruction pointer so that it is already pointing to the next instruction.

3. Execute the current instruction.

4. Go back to 1.

Such is a day in the life of a computer!

4.4 How BASIC Works.

As you will have realised by now, the ORIC computer has been programmed by its designers to understand the BASIC language as described in Chapter 2, and this program is permanently in the machine. The programs that are subsequently written by a user are stored in Random Access Memory (RAM) and these memories forget what they hold when the power is switched off. Hence the machine has to be loaded with your program every time you switch on.

BASIC is known as an 'interpretive' language as compared to a 'compiled' one. Since the ORIC uses BASIC we will describe how an interpreter works first and then look at compiler systems.

The program provided by the system designers can be thought of as a large dictionary. When the computer executes a BASIC command it does so by looking up that command in its dictionary and executing the machine code program it finds under that heading. Thus, the complete definition of the language is resident in the machine all of the time. The part of the program which controls the editing functions available to the user are really part of the BASIC language.

When a program is entered into the machine, either from the keyboard or from some other device such as a cassette recorder, it is not stored in the format which we see when we list the program. All the BASIC commands are converted into 'tokens' in order to save space and to increase operating speed. (A token is a number greater than 128). Also, when a program is saved to cassette or disc, it is the tokenised version that is saved, again to save space and to improve speed. To fully explain this peculiar method, consider these two lines of program:-

```
20 FOR I=0 TO 60
#16 #05 #04 #00 #8D #20 #49 #D4 #30 #C3 #20 #36 #30 #00
```

The upper line is a standard BASIC line of code, easily understood by all. The lower line is how that line of code is stored in the computer, and the meaning of these hexadecimal numbers is as follows:-

#16 #05 is the start address of the next line in reverse order. Thus this means that the next line starts at address #516.
#14 #00 is the line number, again in reverse order with least significant digit first. this means line 0020.
#8D is the token for FOR.
#20 is the ASCII code for a space.
#49 is the ASCII for I
#D4 is the token for =
#30 is the ASCII for 0
#C3 is the token for TO
#20 is the ASCII for a space
#36 is the ASCII for 6
#30 is the ASCII for 0
#00 means end of line

The alternative method to this would require that the whole lot was stored in ASCII code, taking 3 Bytes to hold FOR instead of 1. In this example there does not appear to be much gained by tokenising as most of the line is ASCII already. But, consider the program line PRINT HEX$(PEEK(#501+I)). Here, the ASCII equivalent would take 25 Bytes and the tokenised version 14. Since most BASIC programs look more like this than line 20 above, the savings in space are considerable.

The second point to consider when looking at the advantages of tokenising is that the token is in fact a number, and hence can be used directly to calculate the address in memory at which the entry to its routine is to be found.

To produce this tokenised program there is a special routine in the computer called, not surprisingly, a tokenising routine which converts the BASIC typed in by the user. When the program is listed there is a reverse routine, called a de-tokeniser, which produces the original text from the tokenised version. There are some additions and subtractions made by these routines firstly, to save space when tokenising, and secondly to improve readability when de-tokenising. For example if we type in a line such as:-

10PRINT"HELLO"

and then list it the ORIC will print on the screen:-

10 PRINT"HELLO"

It has inserted a space between the line number and the first command although there was no such space in the line typed in. Secondly, if you examine the original text described in this section you will see that there is a space between the line number, 20, and the first command, FOR, but in the tokenised version no such space exists.

In all modern versions of BASIC, spaces are used merely to improve the readability of the text and good tokenising routines take them out. Similarly good de-tokenising routines put them back in. It is a truism that the more work a section of program has to do, the longer that section has to be and so the system designer has to set a limit to the abilities of his system. The limit for the ORIC appears to be the spaces after the line number.

That completes a rather brief look at the various program sections that go into the make up of an interpretive language like BASIC. To sum up, there is an editor section, a tokeniser and a de-tokeniser, a dictionary and a control section which coordinates all the various sections. We will now look at a compiler system, for comparison, and see how the same facilities are provided.

All computer systems have to have some method of entering and modifying programs. This is called editing and, in machines like the ORIC, the editor is part of the language built into the machine. In the type of system we are now going to look at the editor is a separate program, which can itself be edited and modified and thus tailored to suit individual needs. Let us look at how a program is created and run on a compiler type system.

Firstly the user loads or 'calls' the editor to allow him or her to type in the original text or "source" code of the program. Notice that the code has already been worked out using pencil and paper before the computer is even approached. Having entered the program using the editor, the user saves it either on floppy disc or hard disc or even in the computer's main memory. The exact method used will depend on the size of the computer installation, the hardware available and the number of users on the system. The editor program is then exited and a second program called a compiler is loaded or called. The user's program is then processed by the compiler to produce machine code which can then be executed by the computer.

To complete this picture, it should be mentioned that some compilers produce an intermediate code which is halfway between high level language and the machine code which a processor actually understands. A second stage of compiling is then used to produce executable code. The reason behind this two stage approach is that the first stage is independent of the computer which will ultimately run the final program. That means that the first part of the compiler is usable by all

programmers regardless of the machine they are using. The second part of the compiler is machine dependent and this is actually the smaller part. The first part of the compiler does the error checking and reporting and all of the syntax checking which makes for a large program.

This split compiler technique is often used where the machine code finally produced is to be run on a different type of machine to the one being used at the time to produce the code. This is called 'cross compiling', not because it is done in anger but because the final code is to be carried across to another machine, the 'target' computer, to be run. This is the method used to produce the programs that control washing machines and other such devices which now have a microcomputer doing all the brain work.

Lastly, mention should be made of those compilers which produce assembly language as their output. This may seem to be a very peculiar method of working but again there are hidden advantages. One advantage is that there will be in existence an assembler program for the processor in question because that program is virtually the first to be written. Secondly, the production of a form of code which is understandable by a human means that the performance of the compiler can be checked for efficiency.

This brief description highlights the main differences between an interpretive language, like BASIC, and compiled ones like FORTRAN. The advantages and disadvantges of each system can also be deduced from these differences.

The main advantage of interpreting is that the user has everything available to him all at once. A program can be stopped, changed and run again with very little effort. The compiler system requires that the whole program be re-compiled if any change at all is made. This can make compiler systems very slow in use and leads necessarily to the concept of modular programs, where any change is hopefully limited to the module which contains the part to be changed and, thus, only that part needs be re-compiled. The disadvantages of interpreting are:

Firstly, the final program is slower than a compiled one because the job that the compiler did has still to be done but now it must be done while the program is running.

Secondly, the interpreter takes up a lot of the usable memory of the machine and, hence, the space available to the user is limited.

This latter problem can be alleviated to some extent by 'memory mapping' techniques which are hardware methods of allowing the computer to use a larger memory space than it can address at one time. This sounds a little like the Dr Who's description of why the Tardis is larger inside than out, but all it comes down to is substitution. Imagine the memory map as bricks one on top of the other. One brick is used by the interpreter, another by the user, another by the operating system and so on. When the user's program is being read, it must be in the memory map. However, when the instruction to be executed has been obtained, that brick can be put aside, for the time being, and another put in its place. This system can increase the total memory space almost indefinitely,at the expense of

more and more hardware, without changing the amount of memory avaliable at any one time.

Compiler systems can have almost the whole of the memory space available to the user for his program. The whole of memory is not usually available because there still has to be a small loader program to read in the applications program. In the case where the user's program is to be put into ROM and run from the moment the computer is switched on, the whole of memory is available. Typical speed gains of a compiled program over an interpreted one are as much as 15 or 20 times faster. With some computers (not the ORIC, leastways not yet) you can get a BASIC compiler to combine the ease of learning BASIC with a speed of execution approaching machine code.

Before leaving this subject it ought to be pointed out that one statement in a high level language results in many instructions in machine code. One method of measuring the efficiency of a compiler is to measure how much machine code it produces for a given high level program. Since there is no ORIC BASIC compiler, measuring BASIC by this technique is an empty exercise, so the approach here is to measure how long a given program takes to execute. This has led to several so-called 'benchmark' types of program and all computer manufacturers will quote figures showing how their machine is better than the competition's. Unless you have a really pressing interest in speed, benchmark programs are a useful toy and not much else. The reason is that different interpreters have different strengths and weaknesses, just like the humans who wrote them, and hence it is possible to produce a program that runs faster on machine A than on machine B and vice versa. What the user should really look for are the facilities provided by the computer and whether these match up to his requirements.

Memory Usage By BASIC.

The BASIC interpreter in ORIC is a program, just like any other, and it requires some RAM in which to perform its calculations, to store results and 'messages'. The results it stores will mostly be those required by a user's program, but it must also store such things as the next statement to be executed, the last error message number and other housekeeping information. The 'messages' it stores take the form of flags to remind itself that some event has occurred. In the ORIC the character sets are also put into RAM and the interpreter itself takes up the space above #C000.(See Chapter 3).

The user is mostly unaware of the exact location of anything in memory and can write programs in BASIC without bothering about where, in memory, his programs are or what the interpreter does with the rest of the space. When the user starts to write programs in assembly language and to address memory locations directly himself, rather than through the interpreter, then he needs to know a little about what locations are free to be used and which are not. It can also be helpful to know something about the use the interpreter makes of other memory locations since this can lead to quick ways of achieving some results.

Zero page is probably the most important RAM area to a programmer so we shall look at this area first. Locations not used by the operating system appear to be:

#03 to #0B; #26 to #31; #8B to #8F. Something of what the other locations are used for is known. Locations #0C and #0D are used by the status line system call which is described in Chapter 7. The input buffer is from #35 to #84. This is the area into which the ORIC puts all keyboard-entered characters until RETURN is met. It then analyses the contents of this buffer and takes the appropriate action. If a machine code program does not expect much in the way of keyboard entry, then the upper part of this buffer may be used.

The floating point accumulator is stored in locations #D0 to #D5 as shown below:-

D0	Exponent
D1	High order (mantissa)
D2	
D3	
D4	Low order (mantissa)
D5	Sign (0=+ve −1=-ve)

Rather strangely, there is a subroutine stored in zero page from #E2 to #F9. This particular subroutine has to be in RAM in order for it to work, but it is still odd to find zero page used for such a purpose. The routine is used to load the next character from the program space or from the input buffer into the accumulator. By using RAM to hold this routine it has been possible to make this section of code very compact and, therefore, fast in operation, which in turn leads to a faster BASIC interpreter. Part of this routine, locations #E9 and #EA are the text pointers which are together the address in RAM of the next character to be accessed by the interpreter. This character may come from the keyboard buffer, as in immediate mode, or from a program.

Other known locations in zero page are #9A and #9B which store the start address of BASIC programs, which is location #501. Locations #9C and #9D and #9E and #9F also seem to be related to program storage. #9E and #9F seem to hold an address a few bytes beyond the end of the stored program, while #9C and #9D hold the actual end address.

The rest of known RAM usage is in page 2, apart from page 1 which is taken up by the 6502's stack. Quite a lot of page 2 is unused, but this is not of much consequence to a user because this page cannot be used for indirect addressing (see later) except for JMP commands.

The following information has been supplied by the maker's of the ORIC computer:-

LOCATIONS V1.0	V1.1	DESCRIPTION
	21F	GRA: =1 graphics mode. =0 text mode.
	220	SXTNK: =1 16K =0 48K.
	238	XVDU: soft vector to VDU routine.
	23B	XGETKY: soft vector to get key routine.
	23E	XPRTCH: soft vector to printer output routine.
	241	XSTOUT: soft vector to status line o/p routine.
228	244	INTFS: soft vector to interrupt routine.
22B	247	NMIJP: soft vector to NMI routine.
230	24A	INTSL: return from interrupt handler.
	26A	MODE: see separate table.
	272	TIMER: used for keyboard.
	274	TIMER: used for cursor flash.
	276	TIMER: spare.
	24D	TSPEED: 0=fast 1=slow.
	24E	KBDLY: delay for keyboard auto repeat.
	24F	KBRPT: repeat rate for auto repeat.
	256	PWIDTH: printer width.(Normally 80).
	257	VWIDTH: screen width. (Normally 40).
	2E0	PARAMS: transfer buffer for graphics/sound routines.

The explanation of 'soft vectors' is to be found in Chapter 7 along with descriptions of the system calls available and how to use them. The page 2 locations are listed here merely to give as complete a picture as possible of the memory usage of the machine. Lastly, if you look in locations #293 onwards after loading a tape file using V1.1, you will find the name of the file in this area.

The remaining known locations in page 2 all relate to screen handling. In V1.0 the start of screen RAM is held in #26D and #26E and the number of screen lines in #26F. Also, the current cursor row is held in #268 and current cursor column in #269. #26A is used to hold a number of flags related to screen handling and character input. This location is used for the same purposes in both versions and is detailed below:

BIT 7	Spare
BIT 6	Spare
BIT 5	Mask columns 0,1:1=on
BIT 4	1=Last character was Escape
BIT 3	1=Key click off
BIT 2	Spare
BIT 1	1=VDU on
BIT 0	1=Cursor on

#2DF is used to hold the character read from the keyboard in V1.0. The ASCII code corresponding to that character is put here and the top bit of the location is also set.

In V1.1 the screen handling routines appear to be quite different and the screen locations are not as they were. Two screen addresses are now held in page 2. At #278 and #279 there is the address #BBD0 which is the start of the second text line and in #27A and #27B there is #BBA8 which is the address of the start of the first text line. #268 and #269 hold the current cursor row and column respectively as before. Location #268 is now merely used as a counter and not as an absolute position pointer as before. The number of screen lines is held in #27E.

CHAPTER 5

Practical Control Applications.

We are now going to look at some of the ways in which a computer can be used as a controller. To a large number of people, just sitting at a computer keyboard and jiggling the keys is sufficient. But to the rest of us, computers are here to do things just as mundane as mowing the lawn and washing up. In order to do routine chores, the computer must have information about the device it is controlling and it must be able to send information to the device to tell it what to do next. This flow of information to and from the computer is usually known as Input and Output or just I/O for short.

5.1 Principles Of Input And Output.

If you want to design a computer based controller, you will need to have knowledge not just of the computer, nor of the computer and what it is to control but also of the bits which link the computer to the controlled device - these bits are called the 'interface'. The interface converts the signals of one world into the signals of another. For example, the ORIC, in common with all other small computers, uses a 5 volt power supply and all internal signals are some fraction of this voltage. (To be precise, a 0 is represented by 0–0.6v and a 1 by 2.4 –5.0v). If we want to use the ORIC to control an electric light bulb, we must arrange for the signal levels in the ORIC (of 0–0.6 and 2.4–5.0v) to be converted to 0v and 240v respectively. This conversion is the job of the interface circuitry and we must be very careful to get it right or we may have 240v connected to the ORIC.

The 6502 processor has no special provision for input or output and so any interface has to start off by pretending to be a memory location. By this is meant that it must look just like any other memory location to the 6502 and it must remember what it was told because, otherwise, the 6502 will have to keep reminding it and that is very wasteful of the processor's time. There are special purpose chips to do this kind of work; the 6522 is one such device and the use the ORIC already makes of this has been described in Chapter 4. There are also simpler devices called 'latches' which can be used to remember what the computer said to them and, hence, pass that information on to the rest of the interface.

The 6522 has an internal set of 16 registers and it looks to the processor like 16

consecutive memory locations. A latch looks like just one memory location. To use either device properly, there must be special circuitry to stop the device being activated when addresses other than its own are accessed by the processor. This circuitry is called address decoding circuitry and quite a lot of work has been done for the user already in the ORIC by setting aside Page 3 for input and output and by supplying a special signal to indicate when a Page 3 address has been decoded. This leaves the user with 256 possible memory locations to decode, a much simpler task than decoding the full memory map of 65536 locations.

As regards supplying information to the computer, the rules are much the same, except that they now work the other way. If it is desired to know whether or not a lamp is turned on, the 240v supply to the lamp must be changed to a 2.4–5.0v signal in such a way as to be absolutely certain that the original 240v cannot possibly reach the computer, and this signal must be presented to the computer as a memory location. The memory location has to have the same address decoding as before, and for the same reasons.

5.2 Parallel Communication.

This follows on directly from the principles discussed above. The latch used as a memory location to act as an output port (to use its technical name) has all of its output pins connected by wires directly to a peripheral device. Because these wires run side by side, (in parallel) this is called a parallel output system. The advantages of this method are, firstly, that up to 7 data bits can be transferred at once and, secondly, the hardware required is fairly simple. To complete the connection between the computer and the peripheral, nine wires in all are needed making life a bit more complicated.

The explanations for the strange numbers used above are as follows. An eight way port can only transfer 7 data bits because one line has to be used to signal to the peripheral device that new data is available. There have to be 9 wires in all because one extra wire is needed as a 0v connection to act as a reference line against which the voltages on the other lines are measured.

Such a system can be used as a basic communication port by the following method. Firstly, the computer makes sure that the line used for signalling new data (usually called a 'strobe' line) is high. It then sets the required data onto the other 7 lines and pulls the strobe line low. A wait state is now entered, the duration of which depends on how quickly the system designer thinks the peripheral device receiving the data can respond and then the strobe line is set high and the cycle repeated.

There is an obvious flaw in this method, and that is that the computer does not know if the peripheral device received the data correctly or if it had sufficient time to deal with it. The way to test for both of these is to introduce 'handshaking'. This requires another connection between the computer and the peripheral, bringing the total of lines up to 10. The tenth line is used by the peripheral to signal to the computer that it has accepted the current data. There are two possible ways in which this can be done. Firstly, the method is as before but when the computer

sets the strobe line low it then looks at the handshaking line and waits until it detects that this line is low. At this point the computer thinks 'all is well' and carries on as before. The problem now is that if the handshaking line is held low continuously, the computer does not recognise this as a fault condition. This leads to the use of what is called 'full handshaking'. The computer starts off by ensuring that the strobe line is high and then it tests the handshake line to make sure that it is also high. If it is not, a fault is signalled, or the computer goes back to the start again. If all is well, the computer puts data on the data lines and pulls the strobe line low. It now looks at the handshake line and waits till it is pulled low by the peripheral. If nothing happens after a fixed period of time, the computer aborts the data transfer and signals a fault to the operator. All being well, the handshake line is pulled low by the peripheral, at which point the computer releases the strobe line and continues looking at the handshake line until it too goes high. If it does not, after the timeout interval a fault is again signalled. When the handshake line goes high, the system is back to the start condition with both lines high and both the computer and the peripheral aware that one lump of data has been transferred without fault and the cycle can be safely repeated.

This last method allows data transfer to take place safely between machines of very different operating speeds because it ensures that the faster machine has to run at the speed of the slower. It can also be used for computer to computer communication even though one or both computers are doing other things as well.

5.3 Serial Communication.

The term serial communication refers to the method of sending information one bit at a time. In the parallel system, 7 data bits were sent at once; in this system only one bit is sent. To further simplify the connections between sender and receiver, there is no handshaking line as described for the parallel connection. The simplest serial link consists of just two wires, one for the reference 0v level and one for the data. Since there is no strobe line either, some other mechanism must be used to take its place. The mechanism used is that of timing. With the advent of crystal oscillators it has become possible to time events to very fine limits and serial communications systems make full use of this.

Let us first look at the way a serial transmitter makes up the message it is going to send and then look at how the receiver makes sense of it. Assume that the message to be sent is the ASCII code for U. The seven bit code will be 1010101B and the serial transmitter will be loaded with this information. It will also know how fast to send the message either by default (the settings it assumes at power up) or by later programming.

Having been given a message to send the transmitter will first send a start bit. This means that the transmitter will set its output pin to the voltage level corresponding to a start bit for a period of time that corresponds to one bit period. The transmitter will then look at the first message bit to be sent. In our case this is a 1, so the transmitter will set its output pin to the state which corresponds to a 1 for one bit period. These descriptions of the state of the output are a little devious because the voltage level on the pin my be +5v to represent a 1 or it may be 0v. It all depends on

the conventions being used. However, the rest of the message is sent in the same way with the output pin being set to correspond to 0's and 1's as the bit pattern being sent dictates, for a bit period each. When all the message has been sent, one or more stop bits are also sent to signify end of group. A stop bit corresponds to the opposite voltage to a start bit. If two stop bits are being sent, the transmitter will hold its output voltage at that level for two bit periods.

This completes the transmission of the message and a graph of voltage against time for the transmission is shown in Figure 5.1.

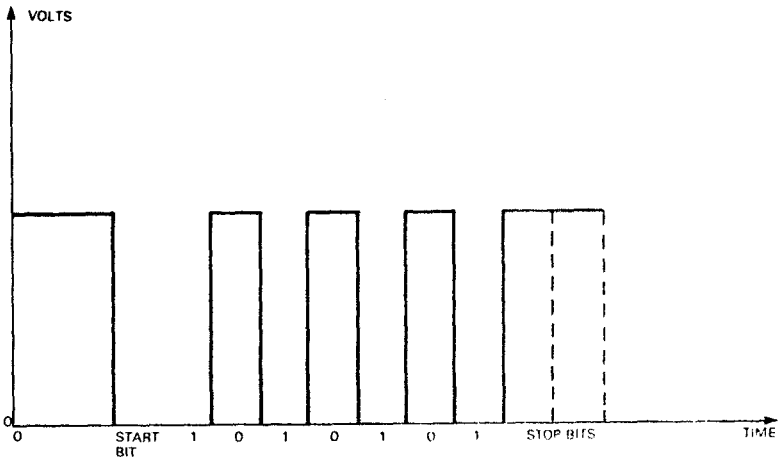


Figure 5.1

Notice that this method of transmitting does not have any knowledge of the state of the receiver. For all that we know at present the receiver may not even be there and the message would still be sent. Methods exist for the receiver to send information back to the transmitter and we will look at one or two of these later.

When the transmitter sends the first start bit, the receiver is alerted that something is happening by the change of state of its input pin (which we have assumed is connected to the transmitter's output pin). The receiver must have two other facts. Firstly it must know how long a bit period is going to be and secondly it must know how many data bits are going to be sent. In our example we sent 1 start bit, seven data bits and one or two stop bits. The number of stop bits is not critical, but assume for now that we sent two.

Having received the leading edge of a start bit, the receiver waits for one half of a bit time and then tests its input pin again. If the start bit is still there, the receiver assumes that a valid transmission is taking place. It now waits for one bit time and tests the input pin again. The voltage on the input pin will now correspond to the first data bit and so the receiver can log this in as the most significant bit. It now waits for another bit time and then logs in the next bit and so on for the total number of bits to be received. When all 7 bits have been received the two stop bits are used to make sure the input pin is set back to its original voltage level. The receiver does not bother to log in the stop bits so the actual number sent is largely irrelevent.

Each message sent is timed as from the leading edge of the first start bit and all bits sent are logged in at half way points, thus minimising errors due to timing differences between receiver and transmitter. The other main point to note is that the number of bits sent can be varied under program control. Thus the serial system could send 7, 8, or even 9 bits at a time, whereas the parallel method is fixed at the number of wires used. (Actually the parallel method can send less than the number of wires available, but obviously not more). The disadvantage of the serial method is that both ends have to know what is going on and both have to operate at the same rate.

Serial communication standards allow the use of two other lines to prevent transmission taking place when the receiver is not ready. These two lines are called CTS (Clear To Send) and RTS (Request To Send). As might be expected the transmitter first asserts RTS to signal to the receiver that it wants to send a message. The receiver asserts CTS to say 'O.K'. If at any time CTS is released, the transmitter must stop sending and must not start again until CTS is again asserted. This technique allows a slow printer to receive information at its printing rate.

The second method for the receiver to send information back to the transmitter is to use a second serial transmission link. In practice this only means adding 1 wire to those we already have. The transmitter can now make enquiries of the receiver using standard control codes and the receiver can make appropriate replies.

No mention has been made so far of what constitutes a transmitter or receiver. While it is possible to program a microprocessor to take on this task it is usually considered to be a waste of the processor's time to be involved in such mundane operations and the job is delegated to a special purpose chip. This device looks like two memory locations to the processor, and when given a message to send in the form of a byte, it sends it over the serial line and signals back to the processor that it has finished and is waiting for the next byte to send. Similarly, the device can receive serial information and convert it into a data byte which the processor can then read. Again the processor can be signalled that a byte is ready or it can interrogate the chip on a regular basis (called 'polling') to check for data being available. Notice that if the polling technique is used, the time interval between polls has to be less than the time taken to receive one byte or there is a danger that a received byte will be overwritten by a second one before it can be read in by the processor.

5.4 ORIC's Printer Port.

This is a standard parallel output port used to drive printers with a Centronics type interface. The maximum number of data bits is 8 and there is one strobe line and one acknowledge line. The protocol used to drive the printer has already been described in Chapter 3, but a brief re-capitulation may not come amiss.

1 ORIC puts data on the 8 data lines.
2 ORIC pulls the strobe line low.
3 ORIC releases the strobe line.
4 ORIC waits for Acknowledge line to go from low to high.

If the acknowledge line does not do what it is supposed to do within about 15 seconds, ORIC assumes the printer is dead and prints a message to that effect on the screen.

The printer port is driven from the internal 6522 which has two 8 way I/O ports as well as 4 other special pins. The I/O port used for the printer is also used to drive the sound chip as well and hence it is not available for general purpose I/O. However it can be used for output if the circuit it is driving can be made to look like a Centronics printer. In fact this is not a very difficult thing to do and a circuit to latch the printer output is shown in Figure 5.2. All that is needed is a 74LS374 latch with the strobe line connected to the clock input of the latch and also connected to the acknowledge input of the ORIC. To demonstrate that this actually does work the circuit of Figure 5.3 is used where the outputs of the latch are buffered by the 74LS240 and used to drive 8 LEDs to signal the state of the data lines.

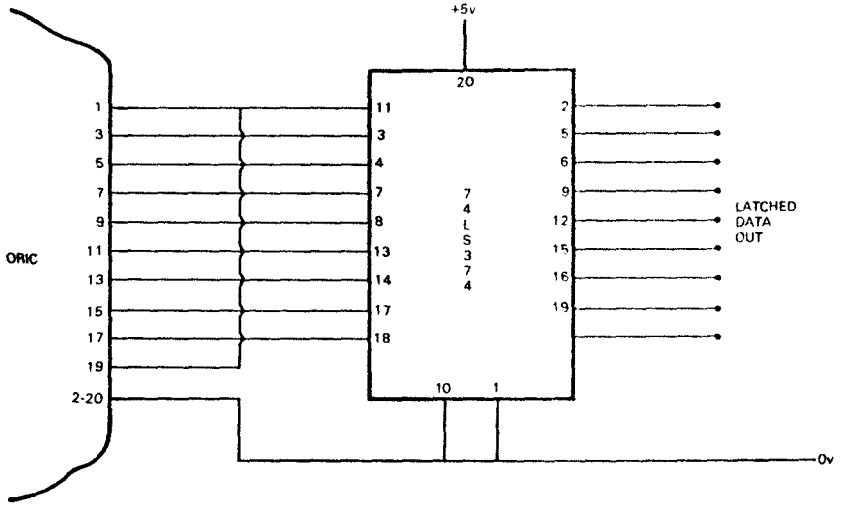


Figure 5.2

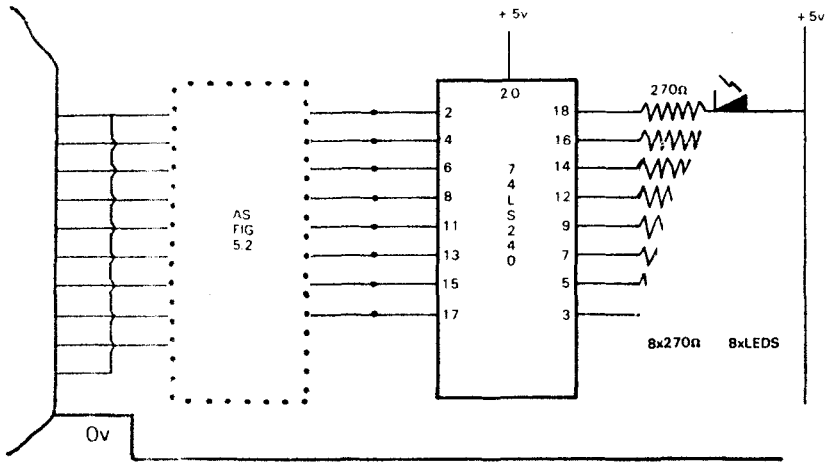


Figure 5.3

The following short programs demonstrate the method of using such an output.

Binary Counting Demonstration.

```
10 FOR I=0 TO 255
20 LPRINT CHR$(I);
30 WAIT 40
40 NEXT I
50 GOTO 10
```

This makes the LEDs count up in binary. Note the semi-colon at the end of line 20. If this is left out the ORIC will send the characters corresponding to RETURN and LINE FEED after CHR$(I) and the LEDs will thus permamently display #0A.

Moving Dot.

```
10 I=.5
20 REPEAT
30 I=I*2
40 LPRINT CHR$(I);
50 WAIT 40
60 UNTIL I=128
70 GOTO 10
```

This makes each LED in turn light up and hence a dot of light seems to move along the row.

Moving Dot Back And Forth.

```
10 I=0.5
20 REPEAT
30 I=I*2
40 LPRINT CHR$(I);
50 WAIT 40
60 UNTIL I=128
70 REPEAT
80 I=I/2
90 LPRINT CHR$(I);
100 WAIT 40
110 UNTIL I=1
120 GOTO 20
```

Now the moving dot moves backwards and forwards along the row.

Bouncing Dots.

```
10 DATA 129,66,36,24,36,66
20 FOR I=0 TO 5
30 READ A
40 LPRINT CHR$(A);
50 WAIT 40
60 NEXT I
70 RESTORE
80 GOTO 20
```

Two dots of light, one at each end, move towards the centre where they meet, bounce off each other and go back to their start positions and then repeat.

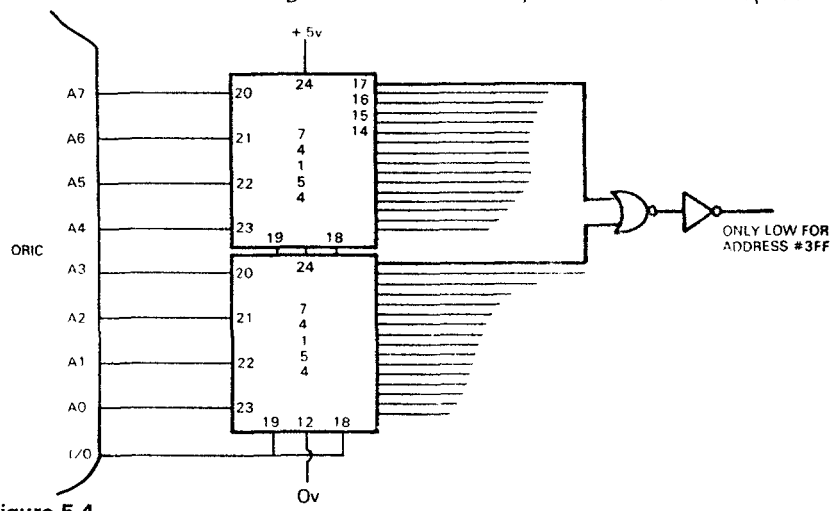


Figure 5.4
Pins 18 and 19 on each decoder go to I/O: Pin 24 on each decoder goes to +5v.

5.5 I/O Via Page Three.

The printer port can only be used to output a bit pattern. This means it can only ever be used to drive displays, such as the LED display just described, or motor driven window displays such as those used in toy shops where all that is needed is to start and stop motors to a fixed time scale. If some knowledge of what is happening outside the computer is needed then a different approach must be used. We must now use the I/O section of the memory map which, as discussed in Chapter 3, is the address range #300 to #3FF or Page 3 as it is usually known.

Table 5.1

Address	A7	A6	A5	A4	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
300–30F	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
310–31F	0	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
320–32F	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
330–33F	0	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
340–34F	0	1	0	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
350–35F	0	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
360–36F	0	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
370–37F	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
380–38F	1	0	0	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
390–39F	1	0	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
3A0–3AF	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
3B0–3BF	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
3C0–3CF	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
3D0–3DF	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
3E0–3EF	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
3F0–3FF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

The ORIC treats all addresses in Page 3 as I/O and a special line, appropriately called I/O, is pulled low when such an address is decoded internally. This greatly simplifies things for the hardware designer because now only 8 address lines need to be decoded to give a unique address within Page 3. Consider the circuit diagram shown in Figure 5.4. The 74154 is an address decoding chip which accepts 4 input lines and decodes the 16 possible states of those lines into 16 output lines. If the address lines A4-A7 are fed into this chip then it will produce a different output for every increase in the address of 16. Table 5.1 explains this more clearly.

From the table it can be seen that an address in the range #350–#35F will make line 6 go low. A second 74154 is used to decode the lower address lines A0–A3, and the outputs from this chip go low for individual byte addresses in the range 0 to #F. Because each 74154 is only enabled when a Page 3 address is decoded,

using the I/O line, their outputs will only respond to addresses in the required range. When an I/O address is decoded, other than for the internal 6522 in the range #300 to #30F, then the I/O CONTROL line must be pulled low to disable the internal 6522. The reason for this is that the internal address decoding only goes as far as decoding a Page 3 address, and it is up to the designer of the I/O circuitry to mask out the internal 6522.

We now have a choice of lines which, between them, allow us to decode any byte address in Page 3. If, for example, we wanted address #356 we would choose line 3 from the top 74154 and line 6 from the bottom one. These two lines would both be low only when address #356 was accessed by the ORIC. If we combine these two lines in a NOR gate and invert its output the resultant line will only go low for that address.

This technique can be used either to write to latches such as the 74LS373 used on the printer port or it can be used to read in from a device such as the 74LS244. In order to be certain of latching and/or reading data at the right time (i.e. when we know that the data is valid) we must synchronise our external circuitry with the ORIC's own clock. We do this by using Q2 to ensure that our device select line is low only during the time that Q2 is high. The complete circuit is now shown in Figure 5.5. It is not strictly necessary to time the input accesses using Q2 but it is good practice, and it is necessary to time the outputs.

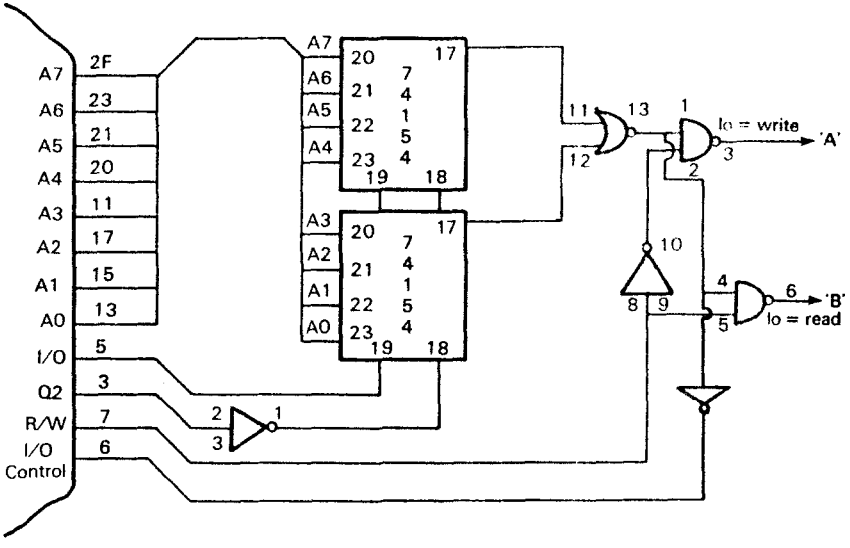


Figure 5.5 (a) Outputs A and B lead to Figure 5.5 (b)

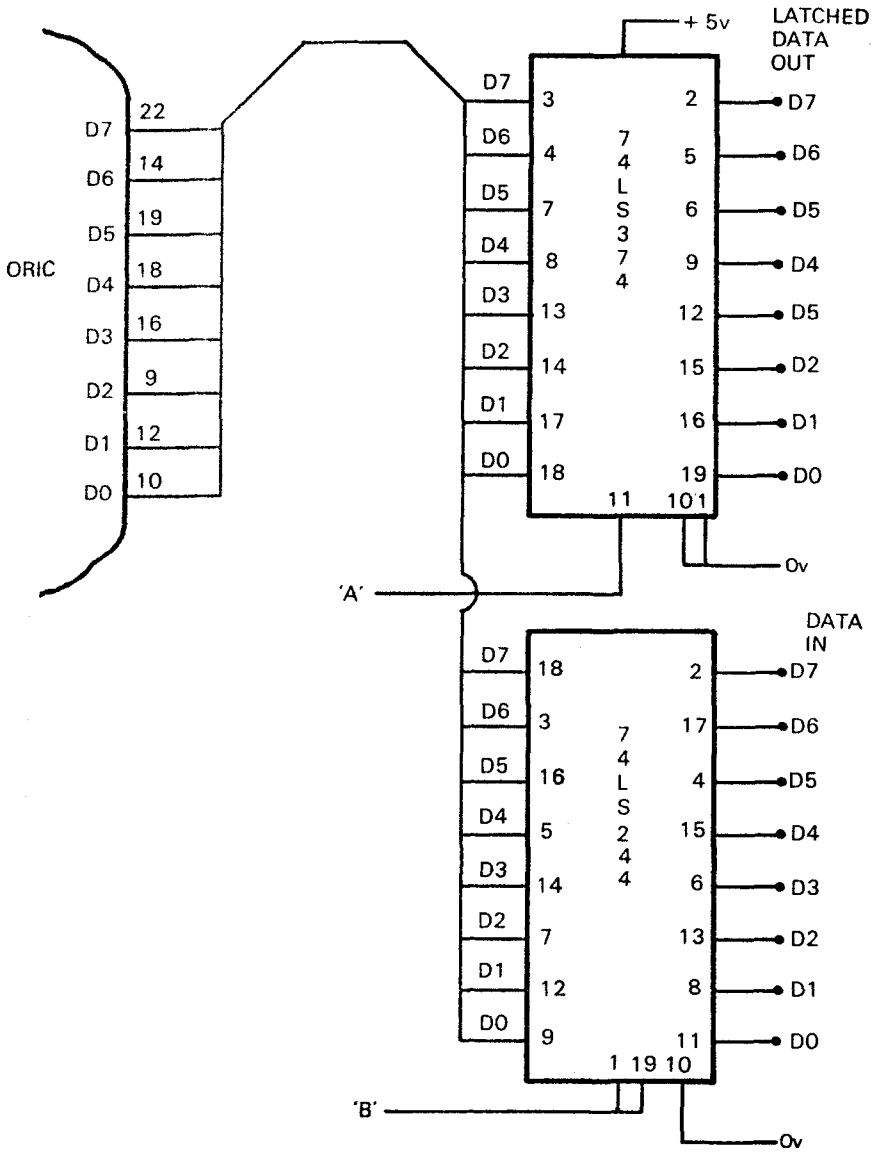


Figure 5.5 (b) A and B take their inputs from Figure 5.5(a)

If the components shown in Figure 5.5 are added to the circuit of Figure 5.4 we have a complete input and output system with 8 input lines and 8 output lines. The obvious way to test that this system works is to connect the outputs to the inputs and then check that whatever is written out is equal to whatever is read in. If the circuit in Figure 5.4 is used exactly as shown then the relevant addresses are #3FF for the output and #3FF for the input. The following program could be used for testing.

```
10 A%=0
20 FOR I=0 TO 255
30 POKE #3FF,I
40 IF PEEK(#3FF)<>I THEN A%=1:J=I:I=255
50 NEXT I
60  IF A%=1 THEN PRINT "ERROR AT ";J
70 GOTO 10
```

This program will cycle through all the possible values for the output port and check that the input port receives the identical value. If it does not, the program will print ERROR AT and then the value at which the test failed. It will in any case try again because line 70 sends it back to the beginning regardles of the test in line 40.

The system described can be used for input and output of digital signals and can be extended if required to accomodate up to 240 input and/or output ports. It is not recommended to try to use all 256 possible addresses for user I/O because to do so means completely disabling the internal 6522 which uses address #300 to #30F. However for expanded I/O systems, and even for a system no bigger than the one described, it is more efficient in terms of hardware to use an external 6522 and to master the extra programming required to make it work.

5.6 Using An External 6522

The 6522 is one of a family of chips which were produced to relieve the micro-computer system designer of the task of designing his or her own input and output structures. This particular device could be called second, or even maybe third, generation and consequently it has a wide range of built in facilities. It is not necessary to use all of these facilities all of the time, indeed it might well be impossible to do so.

The primary purpose of any I/O device is to transfer information either to or from the micro-processor. In the 6522 this is achieved by the use of two 8-bit ports which can be treated rather like the two latches described in the previous section with the exception that each bit of each latch can be either input or output and can even have its direction changed midstream, so to speak. Associated with each port is a direction register which is set up by the micro-processor. The data direction register determines whether the bits in a port are input or output. Bit 0 of a data direction register controls bit 0 of its port, bit 1 controls port bit 1 and so on. If a bit in a data direction register (DDR) is set to 0 then the corresponding bit in the port is an input, if the DDR bit is a 1 then the port bit is an output. The two ports are

referred to as ports A and B and the corresponding DDR's as DDRA and DDRB.

This system produces a very powerful and flexible I/O arrangement since any pin can be input or output or even swapped between the two as necessary. However, it is worth noting that if the direction of a pin is changed from input to output, the voltage level on that pin cannot be predicted. Thus having changed it from input to output, it must then be written to, to guarantee that it is at a known state.

When a 6522 is connected to a micro-processor in the approved manner, it looks to the micro-processor like 16 consecutive memory locations. The first and last of these locations are the normal ports B and A respectively. The third and fourth locations are the direction registers B and A respectively. The second location is a repeat of port A but using this location requires that handshaking of the data takes place. There are 6 locations used for the two timers available and the location which is used as a shift register. That leaves 4 locations to deal with. Two of these are control registers which between them govern the manner in which the various timers, control pins and shift register behave. Lastly there are two locations controlling the 6522's ability to interrupt the micro-processor.

The above brief description demonstrates that the 6522 is no toy. To make full use of the device requires a complete technical description, which is beyond the scope of this book, and a fair amount of experimentation. This latter method cannot be recommended too highly. It is all very well to read books on such subjects and on the strength of this, proclaim expertise. We cannot claim any skills in any subject until we have actually gone out and done it.

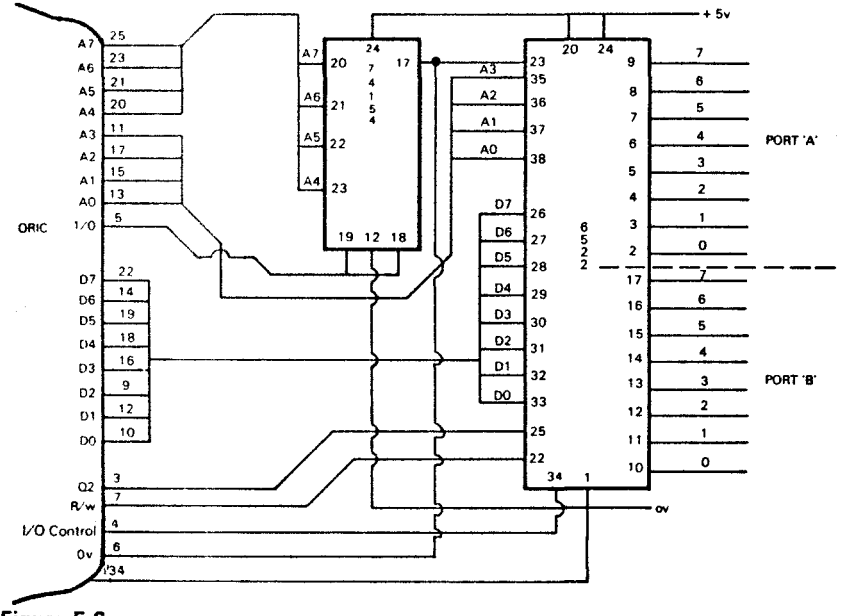


Figure 5.6

Returning to our 6522, the circuit diagram in Figure 5.6 shows how to connect such a device to the ORIC using the facilities available on the expansion connector. If this circuit is followed exactly, the base address of the 6522 will be at #3F0 and hence it will occupy the locations #3F0 to #3FF. For experimental purposes we shall use an LED output monitor similar to the one used on the printer port and a switch array to simulate changing input conditions. These additions are shown in Figure 5.7.

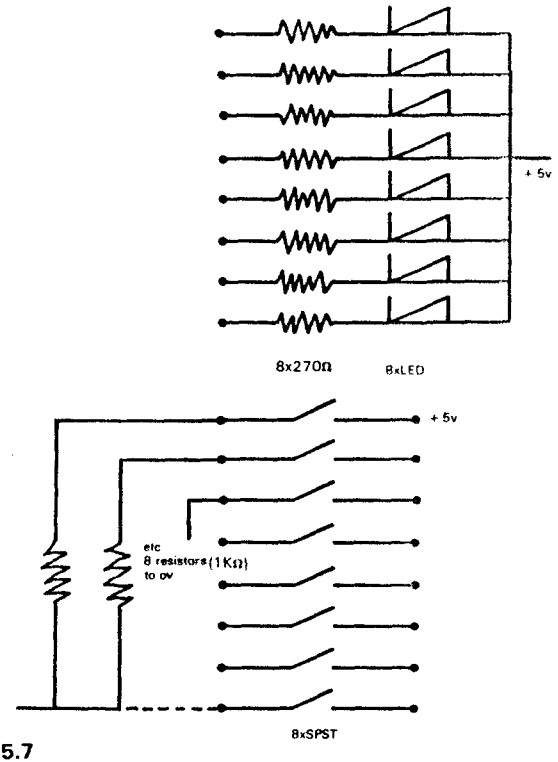


Figure 5.7

To program this 6522, the data direction registers are at addresses #3F2 and #3F3. To set both ports to be outputs, these locations should be set to all ones. This is done using POKE #3F2,255 and POKE #3F3,255. Now any ouput pin can be set high or low by using POKE again. Thus to set the lower 4 bits of port A high and the upper 4 bits low the instruction would be POKE #3FF,15. For port B the instruction would be POKE #3F0,15. In V1.0 the second argument to the POKE command has to be a decimal number but in V1.1 a hexadecimal one can be used.Thus POKE #3FF,#F would be correct in V1.1.

When using a port, or bits of a port, as inputs, the first step is to set the appropriate bits of the data direction register to zero. Thus, to set port A to all inputs the command would be POKE #3F3,0. Now, when reading the inputs use PEEK(#3FF).

The next instructions must mask out those bits which are not required. In assembly language there are logical commands to perform this function. Using BASIC there are no such commands and other ways must be found. One such method is to continually multiply the PEEK'd number by 2, which is equivalent in assembly language to shifting the number left by one, and subtracting 256 if the number exceeds this value, until the required bit is multiplied out. If the result is greater than 256 the bit required was a 1, and if less than 256 then it was a 0.

5.7 Optical Isolation.

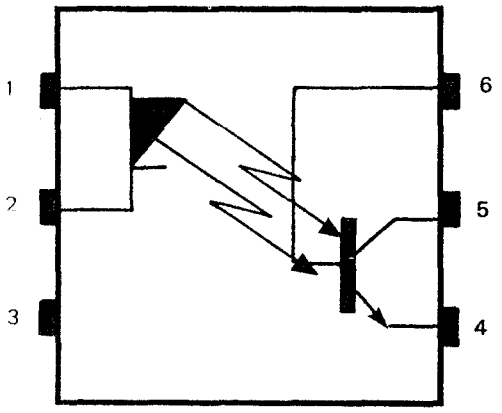


Figure 5.8

Mention has already been made of the fact that an interface must serve not only as a signal conditioner but also as a protective device to prevent the worst excesses of the outside world from reaching the computer. One of the best and cheapest ways of doing this is to use optical isolation.

An optical isolator consists of a light emitting diode and a photo-transistor both arranged so that the light from the diode will activate the transistor. These two are sealed into a light proof package and are provided with separate external connections on opposite sides of the package. The arrangement is shown in Figure 5.8. There is no electrical connection between the LED and the transistor, they are 'isolated' from each other, the only method of passing information between the two is with the beam of light. If the supply to the LED comes from one circuit, for example from the output of a computer, and the supply to the transistor from another, for example a 12v D.C. supply to drive a relay, then the computer can operate the relay without there being any electrical connection between the two.

The circuit is shown in Figure 5.9. The relay could be used to switch mains supply without there being any danger of the mains becoming connected to the computer's output should the relay suffer catastrophic failure.

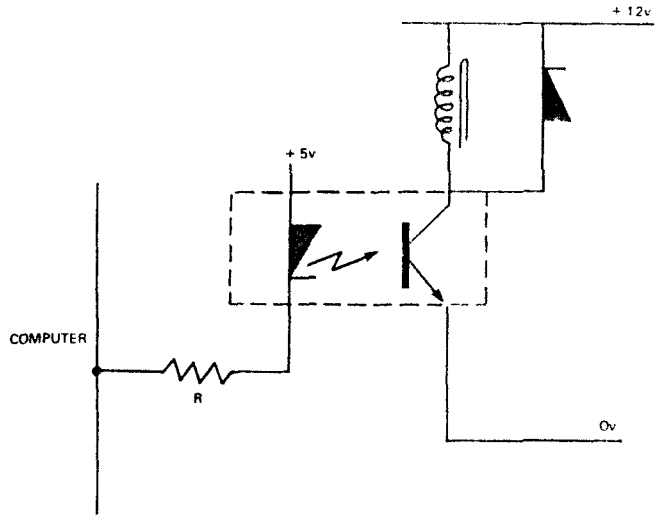


Figure 5.9

Such a failure might seem rather remote, but remember that a good designer has to take into consideration that it might not be he who wires up the final circuit, that someone might spill coffee on the workings, or worse, or even that the unthinkable might happen and the relay just fail catastrophically.

Before going on to the design of such circuits, we should note that using a standard relay to switch mains voltage is not a good idea. The contacts will arc and spark and produce a large amount of electrical interference of the type to which computers are very sensitive. If a mains device is to be controlled, then use the tool for the job, a solid state relay. This relay has built in optical isolation and also uses a solid state switch to turn the mains on and off so that no arcing or sparking occurs. The design is further arranged so that the relay can only switch on or off at zero current points in the mains cycle and consequently there is virtually no interference produced at all. This is demonstrated in Figure 5.10.

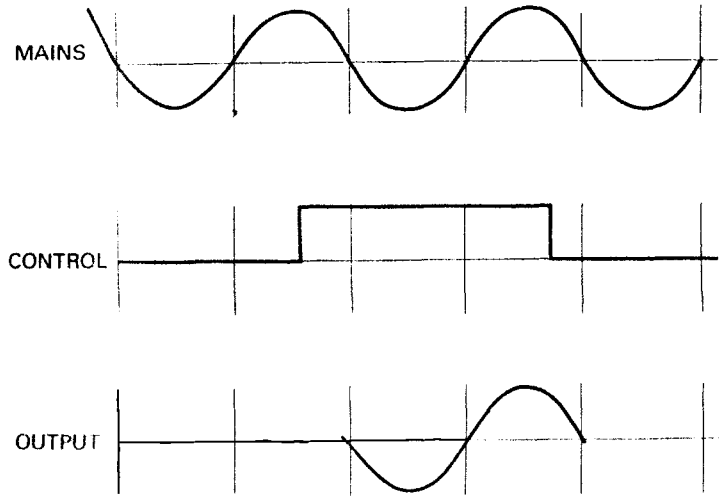


Figure 5.10

Solid state relays are available in all sorts of sizes and power ratings and, if used correctly, are far more reliable than their mechanical counterparts.

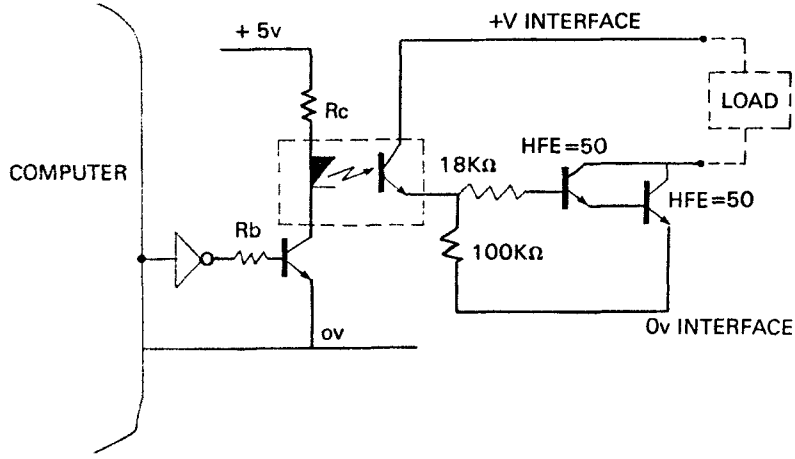


Figure 5.11

The circuit in Figure 5.11 is a complete optically isolated interface for any computer. Before dealing with the arithmetic involved in designing such a circuit, a word or two about the components used is in order:

Firstly, the output of the computer interface driver chip, which may be a 6522 or a 74LS374 latch, is inverted before being used to drive the first transistor. The reason for this is that the 6522 chip sets all its outputs high at switch on or Reset and hence if the output were not inverted the transistor would be switched on also. This would result in all peripheral devices being switched on when the computer was switched on, which is an unsafe condition. If a 74LS374 latch is to be used then care must be taken to ensure that the correct data is written to the latch immediately after powering up the computer because the state of the latches output cannot be guaranteed at switch on. For this reason the 6522 chip is strongly recommended for control applications. Latches may be used in situations where it does not matter if the peripheral device is on or not. The microsystem designer may well decide to use a latch even so because he has total control over the program which the computer executes at switch on and hence he can arrange for it to rush round and set the output latches to safe conditions.

Secondly, a transistor is used to drive the LED instead of this being driven directly from an output port. The reason for this is that the maximum voltage level on an output pin of a TTL compatible device can only be guaranteed to be +2.4v. Thus, although at switch on the output of such a device would be high, it may only be as high as +2.4v leaving 2.6v to drive current through the LED. It may be argued that in a particular interface design this amount of drive is a sufficiently small risk to be ignored. Do not pay any attention to such arguments. The job of an interface designer is to produce safe, working, reliable circuits, not to take risks. If the output of this port is inverted, the resulting output voltage cannot be higher than 0.6v at switch on and this is not enough to turn on a transistor. Consequently the design is safe and risk free. The cost of an inverter is a very small price to pay for peace of mind.

Lastly the arrangement of the driver transistors which are connected to the load is chosen because this is the system which allows the most efficient operation. Using this method, the current drawn by the interface when there is no current throught the load is merely the leakage current of the devices used, a micro-amp or two perhaps. When the interface circuit is switched on by the computer, apart from the necessary current throught the LED, the only current not used by the load is the base drive current to the first transistor of the output pair. Consequently this arrangement makes very efficient use of the power available.

Looking at the detailed design of such a circuit, we will start with the output device. This only has to supply the base current to transistor T1 and hence will not be unduly overworked. The first step is to determine how much current is available to drive the LED. If this is being driven from a separate supply used solely for this purpose then clearly there is all the power available which we choose to have. However, if the LED is being driven from the computer's own supply, which is the more likely case, then we must be much more careful in our design. The ORIC's own power supply is designed to supply the power consumed by the ORIC itself and no more. Clearly, there is only a small demand that can be made on this supply

or it may be overloaded and destroyed. To put numbers into the argument, the ORIC is driven from a mains adapter which produces 9v with a power capacity of 5.4 VA. Translated, this means that the adapter can supply 9v at 5.4/9 amps giving 0.6 amps. Inside the ORIC is a regulator which converts the 9v to 5v and this regulator is capable of handling 1.5 amps. Thus, any overload is going to destroy the mains adapter rather than the internal regulator. Modern regulators are so constructed that destroying them by overload is practically impossible anyway, so we need not worry too much on that score.

The ORIC itself draws about .6 amps when running so the external adapter is fairly fully loaded. Even so, an extra 10mA on top of the existing 600 is not going to make a big difference so we can assume that there is 10mA available for the LED. Note that if a large number (greater than 2) of optically isolated interfaces are being envisaged then a separate interface power supply is essential.

With this in mind we shall limit the LED current to 10mA. The voltage drop across the diode will be 1.6v so that the resistor needed is given by the formula:-

Rc=(5–1.6–Vce)/10 Kohms

Where Vce is the voltage drop across the transistor, typically 0.2v.
Thus Rc is 320 ohms. The nearest preferred value is 330 ohms and hence this will be our choice.

If we assume that the transistor has a current gain of about 50, then we can calculate the required base current as ib=(collector current)/current gain.

Thus ib=10/50 mA=200micro-amps.

The lowest voltage we can guarantee out of the inverter is 2.4v, as mentioned already, and so the base resistor Rb is given by:-

Rb=(2.4-Vbe)/200 Mohms. Where Vbe is the voltage drop between the base and the emmitter of the transistor, typically 0.6v. Thus Rb=9Kohms. The nearest preferred value to this is 9.1Kohms and again this will be our choice.

We must now look at the optical isolator itself. The main characteristic of such a device is that the output transistor will allow a current to flow which is a fixed percentage of the current in the LED. This percentage is called the transfer ratio of the device. It is unfortunate that transfer ratios depend on the actual current through the LED rather than being fixed, but this is another complication we have to live with. However, we now know the LED current is to be 10mA and so we can look up the transfer ratio of the isolator in the maker's data sheet. Devices are available with transfer ratios between 10% and 400% depending on the price. Let us be modest and assume a transfer ratio of 20%. The current available in the output transistor of the optical isolator will therefore be 10*20/100mA giving 2mA. This current has two possible paths. The first is through the 100Kohm resistor to 0v and the second is through the 18Kohm resistor and the two transistors to 0v. The path through the two transistors is represented by a voltage drop of 1.2v, 0.6v per transistor. It is possible to calculate the current split exactly,

but let us take a different approach. Assume, for the sake of argument, that the voltage at the emitter of the isolator transistor is 24v. How much current will each circuit now draw? Through the 100Kohm resistor there will be 24/100mA or .24mA. Through the two transistors and the 18Kohm resistor there will be (24–1.2)/18mA or 1.26mA. Thus the total current drain will be 1.5mA, so the output circuitry used cannot draw as much current as is available and hence we have just calculated the current that they will draw.

Since the output transistors will have a drive current of 1.26mA and assuming again that each has a current gain of 50, the drive capability of the final circuit is 1.25*50*50mA=3.15Amps. This sort of capability ought to be enough for anyone. Should a greater drive be required, the easiest solution is to increase the current gain of one or both of the output transistors by choosing a type with the gain required. This will involve more expense but is probably the most reliable solution.

Note that the figure of 3.15Amps is a maximum capability at 24v. If a 12v supply were used the capability would be reduced unless circuit changes were made. In this case the 18K resistor would have to be reduced to bring the drive current into the base of the first output transistor back to its original level. The method is fairly simple. Ignore the 100K resistor for the moment. (Pretend it is not there). The voltage at the emitter of the isolator transistor is 12v maximum. Base drive required is 1.26mA. Thus the value of the resistor is given by :-

R=(12–1.2)/1.26K giving R=8.75K. The nearest preferred value is 8.2K so choose that. The current available from the opto-isolator transistor is still 2mA since we have not changed the LED current and so the 100K resistor can stay because it will draw an extra 0.12mA only.

The reason for the 100K resistor being used at all is to provide a path to 0v for the stored charge in the base region of the first output transistor so that when the computer switches the optical isolator off, the output transistor also switches off fairly smartly. As a secondary effect this resistor provides a path for leakage current in the output transistors, which may only amount to a micro amp or so but it is nice to have somewhere to go.

The same sort of calculations are used for the input circuit shown in Figure 5.12. The arguments will not be repeated here since the previous essay contains all the necessary information to enable readers to work it all out for themselves.

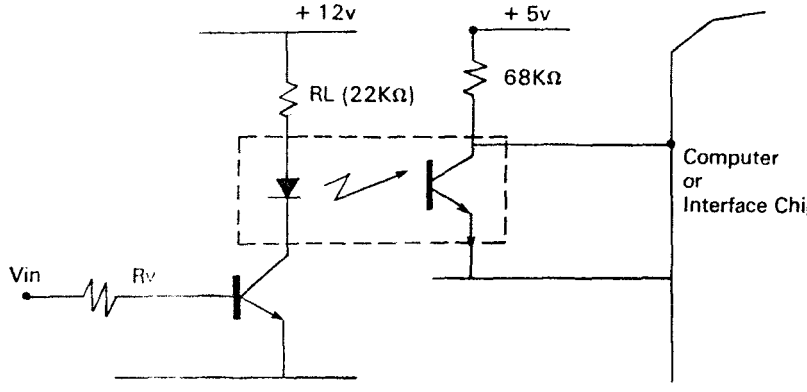


Figure 5.12

CHAPTER 6

Assembly Language Programming

In this chapter we are going to delve a little deeper into the ORIC and find out what makes it tick. Mention has been made already of the fact that the processor at the heart of this machine is a 6502 and in this chapter we are going to explore this device. It has previously been described both as a micro-processor and as the brains of the organisation. It is now time to find out what is meant by these terms.

A processor is a device that interprets and executes commands. These commands may be part of a program, as in a computer, but that is not essential to the processor. A micro-processor is a one-chip implementation of a processor. In these impersonal days all chips are given numbers rather than names to identify them, and the number we are now most interested in is 6502.

The commands a processor interprets take the form of a fixed number of bits. The common fixed numbers in use today are 8 and 16 although 4, 12 and 32 bit processors exist. The 6502 accepts 8 bit commands and we usually express these in hexadecimal notation.

6.1 Assembly Language for the ORIC.

The 6502 processor can interpret approximately 200 different commands. These commands are grouped into families such as LOAD, STORE, COMPARE and so on with each family having more or less members, depending on how important the processor's designers felt a certain family was. It is possible to write programs for the 6502, or for any processor for that matter, in 8 bit form but it is obviously a very slow method of programming and rather prone to error. To make life easier for the programmer the designers have produced a language called Assembly Language which is an organised collection of mnemonics with a grammatical structure. Programs can be written in this language and translated directly into 8 bit code by a special program called an Assembler.

This system relieves the programmer of an immense amount of drudgery and also

checks the program for grammatical errors. In other words, it prevents the programmer from trying to make the processor do something it has not been designed to do.

The best way of describing how this actually works is by illustration. We will use a very short segment of a program listing.

```
START       LDA @#80        A9 80
            STA #7C00       8D 00 7C
```

There are usually no line numbers in Assembler. Labels are used, as in the first line, to mark places to which the processor must be directed. This is similar to GOTO in BASIC.

The segment above shows a label at the start of the first line, followed by a mnemonic, for the instruction to load the accumulator (LDA). The @ sign means that it is actually the number following the @ sign that must be loaded into the accumulator and the # sign means that the number is in hexadecimal. Different Assemblers use different characters for these purposes but the results are the same. The second line has no label and the mnemonic means store the contents of the accumulator at the absolute address which follows, again in hexadecimal.

To the right of the Assembly language are the hexadecimal numbers which the Assembler has produced. LDA # is thus A9, STA is 8D and the 00 and 7C are the address 7C00 with the bytes reversed. This byte reversal of addresses is a peculiarity of the 6502 and explains why DEEK and DOKE reverse the order of the bytes that they operate on. (Because if you DOKE $400, #7C00 then at #400 the byte #00 will appear and at #401 the byte #7C giving an address just as that produced by the Assembler. Similarly DEEKing $400 will return #7C00 giving a valid address).

Programs written in Assembly language run extremely quickly. For example, the command LDA @#80 would be carried out in 2 clock cycles. Since the standard clock frequency for the 6502 is 1MHZ, this translates to 2 micro-seconds. (2*101-6 seconds). In other words the 6502 could do this 500,000 times every second.

To be able to use Assembly language at all, some knowledge of how the 6502 processor is made is essential. The term used to describe a processor's internal structure is 'architecture'. Figure 6.1 shows the architecture of the 6502 which is not very useful from our point of view. However, from this picture a 'programmer's model' can be derived and this is shown in Figure 6.2. As might be expected from the name, the programmer's model contains the information needed for a programmer to actually use the device. A knowledge of the instruction set is also required.

The programmer's model contains an accumulator, two index registers - X and Y, a status register, a stack pointer and a program counter. The program counter always contains the address of the next instruction to be executed.

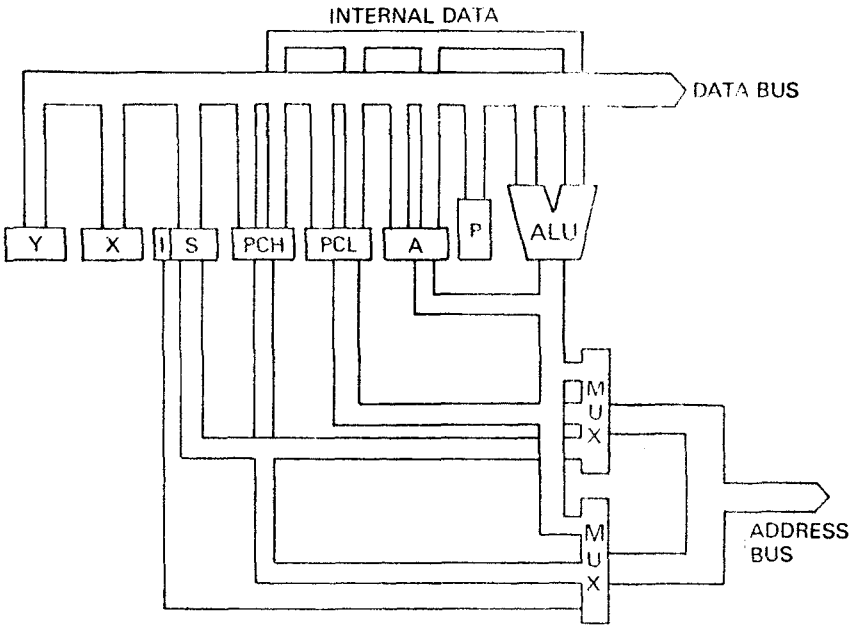


Figure 6.1

The accumulator is the register that is used for all arithmetical and logical operations. It can be loaded either from memory or immediately (as in LDA @#80), or the X or the Y registers can have their contents transferred into it.

The X and Y index registers are used either to keep a count of how many times a certain operation has been performed, or as aids in addressing areas of memory.

The status register is really a collection of bistables (electronic devices which can have one of two states) which remember certain things about the last instruction to be executed. For example, the LDA instruction affects the Z and N flags in the status register. If, after executing this instruction the accumulator contained a negative number, then the N flag would be set. However the LDA instruction does not affect the C flag, so it will remain in whatever state it was in before this command was executed. This carrying on of bits of the status register from one instruction to the next can become important when writing code to interface physical devices to a computer.

The status register comprises 8 bits, as might be expected in an 8-bit machine, but only 7 of these bits are used. The spare one is advertised as being for future

expansion but doubtful if the manufacturers will bother now as 16 and 32 bit machines are taking over the next generation of computers.

The flags stored in the status register are Negative, Overflow, Break, Decimal, Zero and Carry.

In the following descriptions of what event sets what flag, it is assumed that the description applies to an instruction which is capable of modifying that flag. For information on that subject, refer to the instruction set definitions in this chapter.

The Negative flag is set to a one by an instruction which produces a negative result. If the result was positive, the N flag will be cleared.

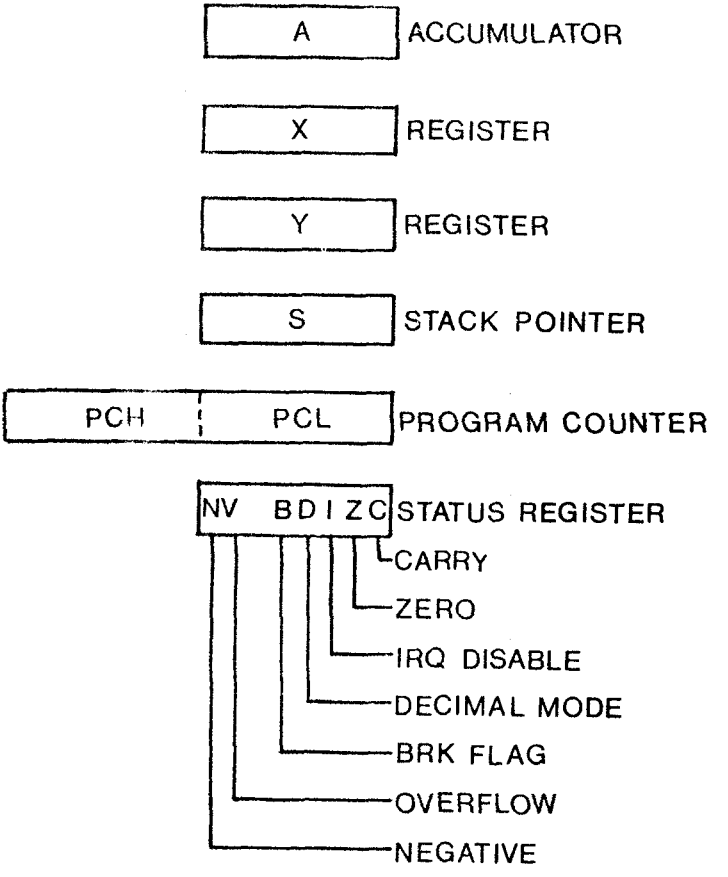


Figure 6.2 PROGRAMMER'S MODEL

The Overflow flag is set by binary overflow from bit 6 to bit 7. See the section on binary arithmetic in chapter 4 for a full explanation.

The Break flag is set when a BREAK instruction is executed, to distinguish it from an interrupt. See BREAK description in instruction section in this chapter.

The Decimal flag is set and cleared by software only and is used to force the processor to perform decimal or binary arithmetic. See chapter 4.

The Interrupt flag is used to allow, or disallow, interrupts. It is set by power on, or reset, and can be cleared or set by software. When an interrupt occurs, it is also set to prevent an interrupt being interrupted until it is ready.

The Zero flag is set to a 1 by an instruction which produces a zero result. If result is non-zero, the Z flag is cleared.

The Carry flag is set by instructions which result in the machine producing a number greater than 255. Thus, if the accumulator held #FF and 1 was added to it, the accumulator would now hold #00, and the carry flag would be set. (As an aside, the Z flag would also be set by such an operation).

The Status register's main use is to allow the results of one instruction to be tested by another, so that the program flow can be modified according to the results of the test.

The stack pointer is the most difficult concept to understand. The usual explanation is to start by describing what is meant by a stack. This method has at least a very long tradition behind it, so we will stick with it now.

A stack is an area of memory used as a last in, first out store. It is called a stack because its operation is just the same as a stack of plates in a spring loaded holder in a canteen. When a plate is put on top of the stack, the spring is compressed a small amount, and that plate is the only one visible. Thus, the last plate to be put on the stack is the first one to be removed.

The stack pointer points to the first empty location in an area of memory from #1FF to #100. Again, due to the nature of the 6502, the stack is used backwards with #1FF being the first location to be used. Whenever a command is executed which results in a byte being put on to the stack, the 6502 puts that byte into the location being pointed to by the stack pointer, and then decrements the stack pointer to be ready for the next byte. When a byte is to be removed, the mechanism is reversed. The stack pointer is first incremented, and then the byte that it points to is used.

The location the stack pointer is now pointing to is considered by the system to be empty, since the contents have been used. However, as you may or may not be aware, the system only takes a copy of the contents of a location when it reads it, so the original contents are still there undisturbed until written over by a new operation. This information can sometimes be helpful to an advanced programmer

who is trying to debug a section of machine code which makes extensive use of the stack. He can freeze the action, so to speak, and then examine the stack's area of memory, to reveal exactly what was put onto the stack and in what order.

You will notice that the stack pointer is only 8 bits long and yet the area of memory use by the stack is #1FF to #100. In the 6502, the leading 1 in these addresses is tacitly assumed. This has advantages and disadvantages. The advantage is that everyone knows where the stack is and there is thus no confusion. The disadvantage is that the stack area cannot be moved and hence it is not possible to have different stacks for different tasks. In some applications this disadvantage can be capitalised on, because the stack can be used to pass information between different parts of a program. In chapter 8 a subroutine called WRITER demonstrates this very clearly.

The last point to note about the stack is that the pointer will overflow from #100 back to #1FF, without telling the user it has done so. It is unlikely you will write a program which makes such extensive use of the stack that this will happen, at least not on purpose, but it can happen by accident. When it does, old data is corrupted and the program crashes. In layman's terms, this sort of crash is equivalent to the machine going away and hiding, because even after recovery, there is nothing left of the original program worth having. You may be lucky and have some bits left which look alright but do not rely on them. Reload and start again.

Writing assembly language programs consists of using these registers in conjunction with the computer's main memory in such a way as to achieve the desired ends. Memory is usually only used to hold the results of a calculation or logical operation. It is the registers which are used to do all the real work. There are some instructions which directly modify memory locations, such as increment, rotate and so on, but addition, subtraction and compare must all use registers.

6.2 Addressing.

A computer's memory consists of a number of bytes which are arranged in order. The processor refers to a required byte by its number. The 6502 processor can address up to 65536 different memory locations with the minimum address being 0 and the maximum 65535. In hexadecimal, these numbers would be #000 and #FFFF. The processor has no knowledge of RAM, ROM or I/O devices. It treats all memory locations identically and it is up to the programmer to ensure that all goes well. It is possible to tell the processor to write data into ROM, but clearly such an operation will fail in as much as the data in the ROM will not be changed. The processor will be unaware of this and will carry on, just as if it had written to RAM.

We can consider the memory to be a row of terraced houses on one side of the street and numbered sequentially from 0 to 65535. There may be gaps in the row where no houses have been built and provided the processor is not directed to one of these, all will be well. The address of each house is its number in the sequence,

and for simple programming applications, this method is sufficient. This addressing technique is usually referred to as 'absolute' addressing because the number used as the address is directly and absolutely the address of the byte required. It is also one of the only two modes where the 6502 uses 16 bit addresses in the instruction. Zero page addresses are a special case of absolute addresses because here the address need only be 8 bits long. To prevent confusion with other addressing modes a special op-code is used for instructions which reference zero page. This allows a two byte instruction to uniquely reference a location directly, so zero page locations can be accessed very quickly. For this and other reasons which will be explained as the remaining addressing modes are described, zero page locations are of special importance to the programmer.

However, it is desirable to be able to calculate an address or to indirect an address, for reasons which hopefully will become clear later. We shall look at what is meant by these terms and how these methods can be used to speed up certain programming operations in the following sections.

Before going on to higher things it is worth memtioning the other two humble addressing modes; 'immediate' and 'implied' addressing.

Immediate addressing means the number to be used is contained in the program being followed and is, in fact, the very next byte after the instruction. We have already shown an example of this mode in the program segment labelled START. When this instruction is obeyed the processor will put the number #80 into the accumulator because #80 is the byte following the instruction.

Implied addressing is the term used to describe instructions in which the addresses to be used are already contained in the instruction. For example, the instruction TYA means transfer the contents of the Y register into the accumulator. No further information is needed for the processor to execute this.

Indirect Addressing.

This means the address the processor is directed to contains the address of the byte containing the data required. There is immediately a problem, because a complete address requires two bytes to hold it, and a single address is only one byte. For the 6502 it is tacitly understood that indirect addresses are held in two consecutive bytes and the instruction references the first of them only. Thus, if the program specifies JMP (#206) (Jump indirect on location #206), it means 'go to locations #206 and #207 and string their contents together to form an address to which you must jump'. In our example, if #206 held #80 and #207 held #40, the indirect address to which the processor would jump is #4080. Notice that again, the address is held in reverse order.

The 6502 is peculiar in that JMP is the only instruction that can use plain indirection. All other instructions which use indirection have also to index the address on the X or on the Y registers. Before going into such complications we had better describe what is meant by indexed addressing.

Indexed Addressing.

This is a mode of addressing which adds the contents of either the X or the Y register to the primary address, to form a secondary address, which is then used by the processor to find the data. For example, a program line might be LDA #8000, Y (load the accumulator from #8000 +Y). This would result in the processor adding the contents of the Y register to #8000 and then loading the accumulator from the resulting address. If the Y register contained #4A, the resulting address would be #804A and the effect of the instruction would be the same as one which read LDA #804A. However, it is more useful than that because the same instruction can be used with different values in the Y register. This means that blocks of memory up to 256 bytes long can be accessed. For example:-

```
        LDY #FF
OUTPT   LDA #8000,Y
        STA #BBA8,Y
        DEY
        BPL OUTPT
        etc.
```

This short segment would output the locations #80FF to #8000 and put them into screen memory form #BCA7 to #BBA8. To do the same thing, using just absolute addressing, is possible but much more involved.

Exactly the same results are obtained by using the X instead of the Y register, if the instruction supports both modes. (The LDX instruction can be indexed on Y but not on X, for example).

The 6502 has a special case of indexed addressing when the primary address is in zero page. Using this mode results in a slightly shorter program because the assembled code is two bytes long instead of three, but there are no other advantages. Not all instructions can use both registers for indexing. As a general rule any instruction which references the accumulator can use both, other instructions can only use the X register. The exceptions are LDX and STX which can only use the Y register.

To sum up, an indexed address is a primary address to which the contents of the X or the Y register are added to form a calculated address.

Indexed Indirect Addressing.

The 6502 uses this mode of addressing as the only indirect mode for all instructions except JMP. There are also two distinct forms of this mode and each form uses its own index register. The two forms are called post-indexing and pre-indexing. Both forms require the primary address to be held in two consecutive bytes in zero page and both add the contents of one of the index registers to form a final address. They differ in the sequence in which the operations are performed. We shall look at post indexing first.

Consider the case where the locations #80 and #81 contain #80 and #4A respectively. Treated as an indirect address pair the 6502 would consider these to string together to form #4A80. If the Y register contained #20 then the instruction LDA(#80),Y would result in the accumulator being loaded with the contents of location #4A80+#20=#4AA0. The contents of the Y register have been added to the address after it has been concatenated.

For pre-indexed addressing, let us keep the numbers the same but this time the X register contains #20 and the instruction looks slightly diferrent: LDA(#80,X). In this case the address is found by counting from location #80 by #20 to reach location #A0. The indirect address is then contained in locations #A0 and #A1. This addressing method is used when a table of addresses is set up in zero page and it is required to access an address in that table. Notice that the X register must be incremented or decremented twice to move the effective address along the table because each indirect address occupies 2 bytes.

The final point to note about both these methods is that they give the same indirect address when the index registers both contain zero. Thus, if an indirect address has been calculated and stored in zero page, in reversed order, either addressing mode may be used as long as the X or Y register used is first set to zero.

Of the two addressing modes, the post indexed mode is used much more often than pre-indexed, for non zero index registers. In fact, although the author has written, and debugged, several K of assembly language for the 6502 for various purposes, the pre-indexed addressing mode has never yet been used except with a zero X register.

The reasoning behind the way these addressing modes are implemented is to try to keep all addresses used to 8 bits only. Because of this constraint, all indirect addresses have to be indirected through zero page to prevent ambiguity. The exception to this rule, JMP indirect, is allowed to have 16 bits in its address field and it can therefore be indirected through any pair of locations.

6.3 Stack Operations.

The comparison of the stack to the canteen plate holder/dispenser has already been made and we will now look at how the 6502 makes use of the stack and the uses the programmer can put it to.

There are a number of instructions which push bytes onto the stack or which pull bytes off it. For example, PHA pushes a copy of the accumulator on to the stack, PLA pulls a byte off the stack and puts it into the accumulator. PHP pushes the status register onto the stack, PLP pulls a byte off the stack and puts it into the status register. Notice the careful wording in the descriptions just given. The 6502 does not know what was pushed onto the stack when it is performing pull operations. Thus the code PHP; PLA would result in a copy of the status register being pushed onto the stack and that copy being pulled off and out into the accumulator. These two instructions together have resulted in a transfer of status register to accumulator which cannot be done by a single instruction.

It is good programming practice, when entering a subroutine, to save the current registers of the processor so that vital information is not lost. It is practically mandatory to do exactly the same on entering an interrupt routine because the programmer does not know where the processor will be in the main program when the interrupt occurs. If, for example, the main program had just loaded the accumulator with a value when the interrupt occurred and the interrupt routine also used the accumulator, then if the contents of the accumulator are not saved before the interrupt routine changes them, the main program will have lost its data.

The most common method of saving registers, in both subroutines and interrupt routines, is to use the stack. In some cases zero page memory is used but as zero page locations have such a high premium on their use, this is not recommended as normal procedure.

On entry to either routine, the following section of code could be used:-

```
PHA
TYA
PHA
TXA
PHA
```

This saves the accumulator, X and Y registers. It may also be necessary to save the status register when entering a subroutine because some programs use the Carry flag to pass binary information between program segments and the subroutine called may corrupt the status register. In the case of interrupts the status register is saved anyway by the system.

On leaving the subroutine, the registers must be unstacked in exactly the reverse order or a transfer between registers will result.

```
PLA
TAX
PLA
TAY
PLA
RTS or RTI
```

and the original situation before the subroutine is restored.

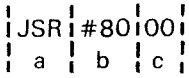
The instruction JSR (jump to subroutine) itself makes use of the stack and this will be described next.

JSR and RTS.

Subroutines are used in Assembly language for exactly the same reasons as in BASIC. They are called by the instruction JSR <label> where <label> represents an address. Subroutines must end with RTS (Return From Subroutine), which sends the processor back to the instruction following the JSR <label.>

When the instruction is executed, the current program counter is pushed onto the stack before the address represented by the label is substituted. This results in the return address (now on the stack) being one less than the correct return address. This state of affairs is unique to the 6502 and it comes about as follows:-

Normal program execution takes the form of fetching an instruction, incrementing the program counter, decoding the instruction and executing it. When JSR is met, the same format is followed. However, JSR must be a 3 byte instruction because the subroutine's address must be 16 bit. The processor attempts to correct this by incrementing the program counter, before pushing it onto the stack. To fully correct it, it should increment it twice, and this it does not do. Consider the code line shown:-



When first encountered, the program counter is pointing to byte 'a'. This byte is fetched and the program counter is incremented to point to 'b'. On decoding the instruction, the program counter is incremented and pushed onto the stack (taking up two bytes). However, it is still only pointing to 'c' instead of to the next instruction. To correct this, the RTS instruction adds one to the program counter after pulling it off the stack and before substituting it for the old one.

It may seem to be a useless piece of information since the system does in fact get it right in the end. However, when debugging programs by examining the stack, it has to be remembered that the return addresses on the stack are one short of the actual arrival addresses. Also, some clever program segments use the stack to force a return to a different address to that expected, and to do this, it has to be known that the address required must be decremented before being pushed onto the stack. For example:-

```
LDA @#80
PHA
LDA @#A4
PHA
RTS
```

If executed, this would result in the processor returning to #804B, not #804A. The subroutine WRITER demonstrates this technique in a practical way. (See chapter 8.4)

Interrupt and RTI

It is a cold winter's night, and you are sitting by the fire reading a good book. Just when the master detective is about to reveal all, there is a tap on your shoulder and a voice says, "Excuse me, but could you spare a minute?" "Yes, of course" you reply, marking your place in the book. "What can I do for you?".

You have just been interrupted while enjoying yourself having a quiet read. You

now have to go off and mow the lawn, wash the dishes or fetch someone from the railway station. When you come back after completing whatever mission you have been given, you pick up the book and start reading again where you left off. (You may be wondering about having to mow the lawn on a cold winter's night, and for the record, so are we).

The preceding description is more or less what happens when you are interrupted in the middle of one task so that you can be made to perform some other task which is considered to be more important than the one you were engaged in. You may take issue on the subject of priorities, but since you have no control over them, this is largely irrelevant. A computer can be interrupted in exactly the same sort of way, except that an electronic equivalent of a tap on the shoulder has to be used, and it will also return to its original task exactly where it left off.

The 6502 processor has a special input pin which is used to interrupt it. This input pin must normally be held at a high voltage level (+2.4 to +5.0v) and when it is desired to interrupt the processor, this pin must be pulled to low voltage level (0.0 to 0.6v). When this happens, the processor will complete any instruction that it is in the middle of and then save its current program counter and status register on to the stack. This requires three stack operations because the program counter is 2 bytes long. The processor now loads a new program counter from locations #FFFE and #FFFF and starts operating the program from the address thus formed. The address is stored in reverse order, the usual thing for the 6502, with the low address in #FFFE and the high in #FFFF. If, for example, #FFFE held #20 and #FFFF held #80, the interrupt routine would then start at #8020.

At the end of the interrupt routine the instruction RTI (Return from Interrupt) instructs the processor to reload the old status register and program counter from the stack and carry on as before. If this is to work properly, any stack operations performed during the interrupt routine must exactly balance, so that when the instruction RTI is reached the stack is in the same state as it was when the routine was entered. Should the stack have been pushed more often than it had been pulled, or vice-versa, the result will be disastrous.

The facility to interrupt the processor is a very powerful tool in optimising the use made of the processor, particularly when driving slow peripheral devices. If the computer is driving a printer, for example, which can print at say 200 characters per second, then the processor would spend most of its time waiting for the printer. To prevent this, a printer buffer is arranged in the computer's memory and the computer sends the first character in the buffer to the printer, then goes back to whatever processing tasks it has to do. When the printer has finished with that first character it interrupts the computer, which then sends it the next character and so on. The benefits of this scheme are plain to see. The computer only sends characters to the printer when the latter is ready to receive them and the rest of the time is spent in performing other more important tasks. The disadvantage of the system is that a chunk of memory has to be set on one side as a printer buffer, but with the price of memory being what it is, and still falling, this is not much of a disadvantage.

From the programmer's point of view, all that he/she has to remember is that the interrupt routine is 'vectored' through locations #FFFE and #FFFF and the 6502 uses the stack to store its status and return address. He/she must make sure there is space on the stack to store this information, and he must take care not to interfere with it during the interrupt routine, or a system crash will result.

The RTI instruction has an implied address, since everyone knows where the return address is to be found even though it is not possible to know what that address is. (Not possible as far as the processor is concerned before the event).

Branching.

All the instructions we have met so far which change the program counter (JMP: JSR: RTS: RTI) do so unconditionally. That is to say, the instructions JMP<label> will always be executed and will always result in the next instruction to be executed being the one at <label>. This next group of instructions may be executed or they may not depending on the conditions which prevail at the time. The conditions which determine whether or not an instruction will be executed are contained in the status register. As previously described the register contains a number of flags, some or all of which may be affected by certain instructions. If this all sounds a bit vague, do not be disheartened. All will shortly become clear and always remember that computers were designed by human beings and must therefore be relatively easy.

To illustrate the above paragraph, let us look at the instruction LDA in some detail. This tells the processor to load the accumulator and the data to use is defined in the rest of the instruction. Thus, LDA # means 'load immediate'. LDA #8000 means 'load from absolute address #8000' and so on through all the addressing modes supported by this instruction. All these slightly different versions of the same instruction have one thing in common. They all affect the same flags in the status register. In the case of LDA the flags affected are the N flag and the Z flag. This means, if the value loaded into the accumulator, regardless of where it came from, is zero, then the Z flag will be set and the N flag will be cleared because zero is not a negative number. If a positive number is loaded, the Z and N flags will both be cleared. If a negative number is loaded, the N flag will be set and the Z flag cleared. This instruction will not affect the status of any of the other flags.

Now, having used this instruction to load the accumulator we can use one or more of the branching instructions to direct the processor to different parts of the program depending on the value loaded. The processor can be made to branch on the carry flag being clear, (BCC); the carry flag being set (BCS); the zero flag being set (BEQ); the zero flag being clear (BNE); the negative flag being set (BMI); the negative flag being clear (BPL); the overflow flag being set (BVS); the overflow flag being clear (BVC).

Clearly, in the case of LDA, the only branching instructions which are relevant are BEQ, BNE, BPL and BMI. If another branch instruction is used, it will be acting on the status flag set by a previous instruction. Note that this is sometimes good programming practice since it allows a common instruction to be executed before

the branching takes place. (To explain this further: if whether a branch took place or not, the accumulator had to be loaded from a certain location, and loading the accumulator did not affect the status flag to be tested, then it makes sense to load the accumulator first, and then branch).

Branching instructions can only send the processor forward through the program by 128 bytes and backwards by 127 bytes. The reason for this is that the top bit of the argument to the branch instruction is used as a sign bit in exactly the same way as described in the chapter on binary arithmetic.

Let us look at a couple of examples of Assembly language to illustrate these instructions. One example has been used already, but it will bear repetition.

```
        LDY @#FF
OUTPT   LDA #8000,Y
        STA #BBA8,Y
        DEY
        BNE OUTPT
        etc.
```

The first instruction loads the Y register with #FF (255 decimal). The second instruction is labelled with OUTPT and it loads the accumulator from #8000 indexed on Y. The third instruction stores the contents of the accumulator at #BBA8 indexed on Y. The fourth decrements Y and the fifth branches back to the label OUTPT if the zero flag in the status register is clear. The instruction DEY affects the negative and zero flags in the status register. Therefore this short program will continue looping through from the label to the branch and back again until the Y register reaches zero.

Example 2:-

```
COMP    LDA (#F2),Y
        AND @#DF
        CMP LABEL,X
        BNE NOTME
        INY
        INX
        CPX @3.
        BCC COMP
        BCS MEE
```

This is slightly longer and contains a number of possible branches. It also demonstrates the comparison instruction. The first instruction, labelled COMP, loads the accumulator from location #F2 indexed on the Y register, so presumably the Y register has been set to some known value (probably zero) already. The next instruction strips out one of the bits and the third compares the result with the byte contained in LABEL indexed on X. Again, the X register must have been set to a known value previously and that value was also probably zero. The compare instruction affects the negative, zero and carry flags so the branch will take place if

the zero flag is clear. Assuming the program does not branch at this point, the X and Y registers are both incremented and the X register is then compared to 3. The way the 6502 performs such a comparison is worth detailing here. Firstly, it sets the carry flag. Then it subtracts the 3 (or other number as used in the program) from the register being compared and sets or clears the status flags according to the result. However, the result is not stored and the contents of the register are not altered. If, in this case, the 3 is greater than the X register, the carry flag will be cleared and the negative flag set. If the 3 is equal to the X register, the carry flag will remain set and the zero flag will be set because the result of the subtraction was zero. If the 3 is less than the X register, the carry flag will be set and the other two flags will be cleared. All comparisons are between registers and some other location and all behave in the manner just described.

Thus, in the program we are looking at, if the X register is less than 3, the program will branch back to COMP. If the X register is equal to or greater than 3, it will branch to MEE, wherever that is.

This program illustrates one final point about branching. The last branch was not BEQ, which would work just as well as BCS most of the time, but used BCS on purpose. The reason is perhaps a little difficult to understand. Suppose we assume the existence of gremlins or other such creatures which can inhabit the insides of computers and like machinery. These gremlins can change the value of a bit in the memory at random time intervals, using methods of their own. (If you find out how they do it, write a book about it and make a fortune). Some of these methods are known and are almost as hard to believe as gremlins. For example, the case of some memory chips is made of plastic. This plastic has some tarry compounds in its make-up to render it light-proof. The tarry compounds contain a small amount of radio-active material, which was discovered by Mme Curie. This material decays slowly and sometimes an emitted particle hits a memory cell and a bit is lost or gained at random. Other gremlins are probably due to interference on the mains supply to the computer. A good programmer takes care to prevent his/her program from crashing due to such an unlikely event. The way this is done is to ensure that branch instructions will always eventually happen even if they do not at the first attempt. Going back to our example, suppose it read:-

```
CPX @3
BCC COMP
BEQ MEE
```

Now if a gremlin strikes and the X register is effectively incremented twice and not just once at the critical time, neither branch will take place even though one of them should. Change the last instruction to BCS and it will take place eventually. This caters for the processor going round the loop more times than intended, regardless of whether the extra trips are due to outside interference or to bad programming.

6.4 The 6502 Instruction Set.

All machines are built with a limit on their capabilities. Cars can only be driven, with any continuing success, on roads specially built for the purpose. Similarly, micro-processors can only be made to do certain things. The limits of achievement of a micro-processor are defined by its instruction set. This is simply a collection of the various commands which the processor can obey. In this section we shall look at each instruction in turn and briefly describe its action and the addressing modes which can be used with it. This is not really a section to be read as you might read a novel or maybe even parts of this book, but rather should be used as a reference section for when writing assembly language programs, to make sure you understand the full implications of each instruction as you use it.

Firstly, it would be helpful to describe the normal conventions used by most 6502 assemblers. We shall look only at those covering addressing modes because the facilities available on different assemblers are all accessed by different methods and to try to describe them all would take another book.

Immediate Addressing.

This is usually denoted by '#' or '@'. In what follows we shall use '@'. Example LDA @7 would load the accumulator with 7.

Zero Page Addressing.

The argument is a single byte. It may or may not be a hexadecimal number. Example LDA 100: LDA #64. Both these instructions result in the same number in the accumulator.

Hexadecimal Notation.

The '#' sign is used by the ORIC and we shall continue with this convention.

Absolute Addressing.
The argument is a double byte expressed in normal order. Example STA 6400: STA #8000.

Absolute Addressing indexed on X or Y registers.

The argument is a double byte, followed by a comma and the register used. Example STA #7C00,X.

Indirect Addressing.

The two forms available make different use of brackets to differentiate between them. The argument is a single byte since all indirect addresses (except JMP) are through zero page. Example LDA(#80),Y post indexing. LDA (#80,X). pre indexing.

JMP also uses brackets, but it cannot be indexed. JMP (#207).

Accumulator.

The argument is a single 'A'. When this is used the instruction only affects the accumulator. Example ROL A. Meaning rotate left the accumulator by one position.

The Actual Instruction Set.

For each instruction, we give the mnemonic, a description, the addressing modes and status register.

Add with carry. Mnemonic:ADC

Add the contents of the memory specified to the accumulator along with the carry bit. The result of the addition is put in the accumulator.
Examples:- ADC @7 add 7 to accumulator
ADC #80 add contents of location #80.

Addressing modes available:-

Absolute	6D
Accumulator	--
Absolute,X	7D
Absolute,Y	79
Immediate	69
Implied	--
Indirect	--
Zero page	65
Zero page,X	75
Zero page,Y	--
(Indirect,X)	61
(Indirect),Y	71
Relative	--

Status Register N V B D I Z C
* * * *

Logical AND. Mnemonic:AND

Carry out the logical AND of the accumulator and the specified memory location. The accumulator will hold the result.

Examples:- AND @55
AND #60

Addressing modes available:-

Absolute	2D
Accumulator	--
Absolute,X	3D
Absolute,Y	39
Immediate	29
Implied	--
Indirect	--
Zero page	25
Zero page,X	35
Zero page,Y	--
(Indirect,X)	21
(Indirect),Y	31
Relative	--

Status Register N V B D I Z C
* *

Arithmetic shift left. Mnemonic: ASL

Shift the contents of the specified location to the left by one bit position. Bit 7 moves into the Carry bit in the status register, and a zero is moved into bit 0

Example:- ASL A shift accumulator
ASL #60 shift zero page

Addressing modes available:-

Absolute	0E
Accumulator	0A
Absolute,X	1E
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	06
Zero page,X	16
Zero page,Y	--

(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* * *

Branch carry clear. Mnemonic BCC

If the carry flag is zero, branch to the indicated address. The address must lie within the range +127 bytes and −128 bytes. Normal assemblers allow branching to a label and do all the binary arithmetic.

Example:- BCC NEXT

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	90

Status Register N V B D I Z C

Branch carry set. Mnemonic:BCS

If the carry flag is 1, branch to the indicated address. Otherwise same as BCC.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	B0

Status Register N V B D I Z C

Branch if zero. Mnemonic: BEQ

If the zero flag is 1, branch to the indicated address. Otherwise same as BCC.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	F0

Status Register N V B D I Z C

Compare memory bits with accumulator. Mnemonic: BIT

The accumulator and the memory location are logically ANDed together. If they are equal, the Z flag is set, reset otherwise. Also the top two bits of the memory location are put into the V and N bits of the status register respectively. Thus, if location had bit 7=1 and bit 6=0, the status register will have bit N=1 and bit V=0.

Addressing modes available:-

Absolute	2C
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	24
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
M7 M6 *

Branch on minus. Mnemonic: BMI

If the N flag is set, branch to the indicated address. Othewrwise same as BCC.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--

(Indirect,X)	--
(Indirect),Y	--
Relative	30

Status Register N V B D I Z C

Branch on non-zero. Mnemonic: BNE

If the Z flag is 0, branch to the indicated address. Otherwise same as BCC.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	D0

Status Register N V B D I Z C

Branch on plus. Mnemonic: BPL

If the N flag is reset (=0), branch to the indicated address. Otherwise same as BCC.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	--

Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	10

Status Register N V B D I Z C

Break. Mnemonic: BRK

Software interrupt. Current program counter+2 and status register are pushed onto the stack. The program then vectors through locations #FFFE abd #FFFF just as for a normal interrupt. However, the B flag in the status register is set before it is pushed. Notice that the program counter+2 is pushed onto the stack. It may be that this results in the program counter not pointing to the next instruction, in which case action will have to be taken. It is normal practice to follow BRK by a null instruction, NOP to allow for this.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	00
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
*

Branch on overflow clear. Mnemonic: BVC

If overflow flag is clear, branch to the indicated address.
Otherwise same as BCC.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	50

Status Register N V B D I Z C

Branch on overflow set. Mnemonic: BVS

If the overflow flag is set, branch to the indicated address.
Otherwise same as BCC.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	70

Status Register N V B D I Z C

Clear carry. Mnemonic: CLC

Reset the carry flag to zero. It may have been zero already, but so what? clear it anyway. This is necessary before using ADC to prevent spurious addition of left over carry bits.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	18
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
*

Clear decimal flag. Mnemonic: CLD

Reset the decimal flag to zero. Any arithmetical instructions will now use the binary system.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	D8
Indirect	--

Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
*

Clear interrupt bit. Mnemonic: CLI

Reset interrupt bit to zero, thereby enabling interrupts. At power on or reset, the interrupt bit will always be set, thus disabling interrupts until the programmer is ready for them.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	58
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
*

Clear overflow flag. Mnemonic: CLV

Reset overflow flag to zero. What else can one say?

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	B8
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
*

Compare. Mnemonic: CMP

The effect is to subtract the location specified from the accumulator and to modify the status register according to the result. No memory locations or registers are actually changed and the result is not stored. If the two quantities are equal, the Z flag is set, reset otherwise. If the memory location is greater than the accumulator, the carry flag is cleared. If the location is less than or equal to the accumulator, the carry flag is set. The N flag is set or cleared by the sign bit of the result.

Addressing modes available:-

Absolute	CD
Accumulator	--
Absolute,X	DD
Absolute,Y	D9
Immediate	C9
Implied	--
Indirect	--
Zero page	C5
Zero page,X	D5
Zero page,Y	--
(Indirect,X)	C1
(Indirect),Y	D1
Relative	--

Status Register N V B D I Z C
* * *

Compare to X. Mnemonic: CPX

The X register is used instead of the accumulator.

Addressing modes available:-

Absolute	EC
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	E0
Implied	--
Indirect	--
Zero page	E4
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* * *

Compare to Y. Mnemonic: CPY

This is the same as CMP except that the Y register is used.

Addressing modes available:-

Absolute	CC
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	C0
Implied	--
Indirect	--
Zero page	C4
Zero page,X	--
Zero page,Y	--

(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* * *

Decrement. Mnemonic: DEC

The memory location specified has its contents reduced by one. If the location contained zero, after decrementing it would contain #FF.

Addressing modes available:-

Absolute	CE
Accumulator	--
Absolute,X	DE
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	C6
Zero page,X	D6
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* *

Decrement X. Mnemonic: DEX

The contents of the X register are reduced by one. Same as DEC.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--

Immediate	--
Implied	CA
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* *

Decrement Y. Mnemonic: DEY

The same as DEX but affects the Y register.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	88
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* *

Exclusive or. Mnemonic: EOR

The memory location and the accumulator are Exclusive-ored together. The result is best demonstrated by a truth table:-

memory	Acc.	Result
0	0	0
1	0	1
0	0	1
1	1	0

The truth table is applied to the corresponding bits of the location and the accumulator, and the result ends up in the accumulator. Example, if memory location contained #55, the accumulator #27, the result of E-oring them would be #72

Addressing modes available:-

Absolute	4D
Accumulator	--
Absolute,X	5D
Absolute,Y	59
Immediate	49
Implied	--
Indirect	--
Zero page	45
Zero page,X	55
Zero page,Y	--
(Indirect,X)	41
(Indirect),Y	51
Relative	--

Status Register N V B D I Z C
* *

Increment. Mnemonic: INC

Opposite to decrement.

Addressing modes available:-

Absolute	EE
Accumulator	--
Absolute,X	FE
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	E6
Zero page,X	F6
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* *

Increment X. Mnemonic: INX

Opposite to DEX.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	E8
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* *

Increment Y Mnemonic: INY

Opposite to DEY.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	C8
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

tatus Register N V B D I Z C
* *

Jump to address. Mnemonic: JMP

Put address specified into the program counter.

Addressing modes available:-

Absolute	4C
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	--
Indirect	6C
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C

Jump to subroutine. Mnemonic: JSR

The contents of the program counter+2 are saved on the stack, and the new address is loaded into the program counter. Program execution will now take place as from the new address.

Addressing modes available:-

Absolute	20
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C

Load accumulator. Mnemonic: LDA

The accumulator is loaded from the specified address.

Addressing modes available:-

Absolute	AD
Accumulator	--
Absolute,X	BD
Absolute,Y	B9
Immediate	A9
Implied	--
Indirect	--
Zero page	A5
Zero page,X	B5
Zero page,Y	--

(Indirect,X)	A1
(Indirect),Y	B1
Relative	--

Status Register N V B D I Z C
* *

Load X register. Mnemonic: LDX

Load the X register from memory.

Addressing modes available:-

Absolute	AE
Accumulator	--
Absolute,X	--
Absolute,Y	BE
Immediate	A2
Implied	--
Indirect	--
Zero page	A6
Zero page,X	--
Zero page,Y	B6
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* *

Load Y register. Mnemonic: LDY

Load Y register from memory.

Addressing modes available:-

Absolute	AC
Accumulator	--
Absolute,X	BC
Absolute,Y	--
Immediate	A0

Implied	--
Indirect	--
Zero page	A4
Zero page,X	B4
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* *

Logical shift right. Mnemonic: LSR

Shift the specified location right by one bit position. Bit 7 will become a zero. The least significant bit (bit 0) is transferred to the carry bit.

Addressing modes available:-

Absolute	4E
Accumulator	4A
Absolute,X	5E
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	46
Zero page,X	56
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
0 * *

No operation. Mnemonic: NOP

Do nothing for 2 clock cycles. Can be useful as a time delay or as a place where a patch can be inserted later.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	EA
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C

Or memory with accumulator. Mnemonic: ORA

Logical OR of memory with accumulator. The result is stored in the accumulator.

Addressing modes available:-

Absolute	0D
Accumulator	--
Absolute,X	1D
Absolute,Y	19
Immediate	09
Implied	--
Indirect	--
Zero page	05
Zero page,X	15
Zero page,Y	--
(Indirect,X)	01
(Indirect),Y	11
Relative	--

Status Register N V B D I Z C
* *

Push A. Mnemonic: PHA

A copy of the accumulator is pushed onto the stack.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	48
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C

Push status. Mnemonic: PHP

A copy of the status register is pushed onto the stack.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	08
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C

Pull accumulator. Mnemonic: PLA

The next byte is pulled off the stack and put into the accumulator.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	68
Indirect	--
Zero page	--
Zero page	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* *

Pull status. Mnemonic: PLP

The next byte is pulled off the stack and put into the status register.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	28
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--

(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* * * * * * *

Rotate left. Mnemonic: ROL

The contents of the speciified address are rotated to the left by one bit position. The carry bit is used to fill bit 0. Bit 7 goes into the carry bit.

Addressing modes available:-

Absolute	2E
Accumulator	2A
Absolute,X	3E
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	26
Zero page,X	36
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* * *

Rotate right. Mnemonic: ROR

The contents of the specified memory location are rotated right by one bit position. The carry bit goes into bit 7 and bit 0 goes into the carry bit.

Addressing modes available:-

Absolute	6E
Accumulator	6A
Absolute,X	7E
Absolute,Y	--

Immediate	--
Implied	--
Indirect	--
Zero page	66
Zero page,X	76
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* * *

Return from interrupt. Mnemonic: RTI

Restore the program counter and the status register from the stack and adjust the stack pointer accordingly.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	40
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* * * * * * *

Return from subroutine. Mnemonic: RTS

Remove the program counter from the stack, add 1 to it and put it into the program counter. Adjust the stack pointer accordingly.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	60
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C

Subtract with borrow. Mnemonic: SBC

Subtract from the accumulator the contents of the address specified and the inverse of the carry flag. The result is left in the accumulator. (The inverse of the carry flag is the borrow).

Addressing modes available:-

Absolute	ED
Accumulator	--
Absolute,X	FD
Absolute,Y	F9
Immediate	E9
Implied	--
Indirect	--
Zero page	E5
Zero page,X	F5
Zero page,Y	--
(Indirect,X)	E1
(Indirect,Y)	F1
Relative	--

Status Register N V B D I Z C
* * * *

Set carry flag. Mnemonic: SEC

Set the carry flag to one.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	38
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
*

Set decimal mode. Mnemonic: SED

Set the decimal mode flag in the status register.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	F8
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
*

Set interrupt disable. Mnemonic: SEI

Set the interrupt flag in the status register to 1 to disable interrupts.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	78
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
*

Store accumulator. Mnemonic: STA

Store a copy of the contents of the accumulator in the specified location.

Addressing modes available:-

Absolute	8D
Accumulator	--
Absolute,X	9D
Absolute,Y	99
Immediate	--
Implied	--
Indirect	--
Zero page	85
Zero page,X	95
Zero page,Y	--

(Indirect,X)	81
(Indirect),Y	91
Relative	--

Status Register N V B D I Z C

Store X register. Mnemonic: STX

Store a copy of the X register in the specified memory location.

Addressing modes available:-

Absolute	8E
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--
Zero page	86
Zero page,X	--
Zero page,Y	96
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C

Store Y register. Mnemonic: STY

Store a copy of the Y register in the specified memory location.

Addressing modes available:-

Absolute	8C
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	--
Indirect	--

Zero page	84
Zero page,X	94
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C

Transfer accumulator to X. Mnemonic: TAX

Put a copy of the accumulator into the X register.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	AA
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* *

Transfer accumulator to Y. Mnemonic: TAY

Put a copy of the accumulator into the Y register.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--

Absolute,Y	--
Immediate	--
Implied	A8
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* *

Transfer stack pointer into X. Mnemonic: TSX

Put a copy of the stack pointer into the X register

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	BA
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* *

Transfer X to accumulator. Mnemonic: TXA

Put a copy of the X register into the accumulator.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	8A
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* *

Transfer X to stack pointer. Mnemonic: TXS

Put a copy of the X register into the stack pointer.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	9A
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C

Transfer Y to accumulator. Mnemonic: TYA

Put a copy of the Y register into the accumulator.

Addressing modes available:-

Absolute	--
Accumulator	--
Absolute,X	--
Absolute,Y	--
Immediate	--
Implied	98
Indirect	--
Zero page	--
Zero page,X	--
Zero page,Y	--
(Indirect,X)	--
(Indirect),Y	--
Relative	--

Status Register N V B D I Z C
* *

CHAPTER 7

ORIC's Operating System

In this chapter we are concerned with those sections of the ORIC's ROM which are not identifiable as BASIC. The technical term for this is the ORIC's operating system. It might be helpful to describe what is meant by this phrase.

Imagine, if you can, that you yourself are the BASIC interpreter. You sit in a little room all by yourself. There is a letter box marked IN on one wall and another marked OUT on the opposite wall. On the third wall there is a clock. Your job is to read all the incoming messages and if you can understand them, take the necessary action. This action always results in you sending out messages sooner or later. If you do not understand an incoming message you send out a message saying SYNTAX ERROR or some such remark.

As you sit in your little room, possibly pondering on the incredible obtuseness of whoever it is that keeps sending you these ludicrous messages, you have no idea at all that you are actually inside an ORIC computer. You are similarly unaware that all incoming messages originate from a keyboard or that all your outgoing messages end up on someone's T.V. screen. The clock on the wall is driven by an agency which is another of life's mysteries. In short, it does not matter to you where the room is, as long as the facilities provided are exactly the same as those you have here, you will be able to function properly.

The room just described and the message carrying and time-keeping facilities are what the operating system provides. This section of the ORIC's program knows all about the keyboard and the T.V. screen (well, almost all) and keeps track of the time and so on. It relieves the BASIC interpreter of a whole load of tasks all of which are associated with the actual hardware of the computer.

The advantages of structuring a computer's make-up in this way are two-fold. Firstly, having written a BASIC interpreter it can be used in any type of hardware simply by altering the operating system so that the BASIC part of the program always appears to be in that same room. Secondly, machine code programmers can use the message carrying and time keeping facilities in the machine independently of BASIC because these sections are now identifiable as separate routines. It is this latter advantage that interests us here. We are now going to look

at those system routines that are available in the ORIC and describe how to use them. Because we have two operating systems to describe, each section of this chapter will be split into two parts, one for each operating system.

7.1 System Calls.

The exact definition of this phrase is a little difficult. It is probably best defined by an example. In the ORIC computer we know that the screen memory is from #BBA8 to #BFE0. Thus, we can put characters on the screen by POKEing them straight in or by using the PRINT and related commands. From machine code the same two options are open to us. We can either do the equivalent of POKE or we can use the system call provided to do the job for us. POKEing or its equivalent may well be the faster method but using the system call is safer. The reason is that on this particular version of the ORIC the screen memory is from #BBA8 to #BFE0 but on subsequent versions it may not be. Hence, POKEing straight into screen memory works with this version of the machine but our program would have to be modified to run on later models. To someone who is only ever going to own the one machine this is of little consequence. To a professional programmer who is writing programs to run on machines for which there are already several versions it is very important. He does not want to have to write one program for each version of the machine,nor does the supplier want to stock separate programs for the various versions. Thus, the good programmer always uses system calls where these are available.

The range of activities of system calls is the same, usually, as the range of activities of the current operating system. Thus these calls may be used to provide input and output facilities, or they may set a flag somewhere in memory to indicate that certain conditions are now required. For example location #26A in both versions contains certain flags. If bit 0 of this location is set to 0, and the rest of the location left as it was, the cursor is rendered invisible. Set this bit back to a 1 and the cursor is visible again. The designers of the ORIC might well have written a system call just to change this bit because future versions of this machine may not use location #26A in this way. If there was a system call and if its entry point was always the same from version to version (easy enough to do using jump tables) then programs which use the system call rather than changing the location directly are protected against future alterations in the machine.

V1.0 Calls

The range of system calls available in V1.0 is rather restricted compared to V1.1. However, a number of useful things can be done. The first requirement of most programs is to print characters on the screen. This call is undocumented by the machine's designers but there is a subroutine at #CC12 which seems to work. To use this routine from machine code, the accumulator must hold the character to be printed on the screen. Thus to print a message on the screen we might use:-

```
OUTPUT      LDY @0
NEXT        LDA MSG,Y
            BMI OUTEND
            JSR #CC12
            INY
            JMP NEXT
OUTEND      RTS
MSG         ASC "HELLO THERE"
            DFB #FF
```

This routine works as follows; on entry the Y-register is set to zero. Then the accumulator is loaded with the first character of the message, in this case 'H'. This is tested to see if it is negative and since it is not, the system call is used to print the character 'H' on the screen at the current cursor location. The system call also takes care of the routine chore of moving the cursor along by one character position so that the next character to be printed will not overwrite the last one. Now the Y-register is incremented and the routine jumps back to NEXT. The cycle is then repeated and the 'E' is printed. This continues until the byte #FF is loaded, which is negative, at which point the routine exits. Notice that this routine has not bothered to save any registers, nor has it sent any control characters to the screen. The routine at #CC12 will accept all the standard control characters and act on them correctly.

The next requirement is usually to read a character from the keyboard. Again, in V1.0, there is no recognisable system call to do this. However this version of the ROM uses location #2DF to pass the keyboard character between routines. A machine code program can poll this location for characters. This short program section demonstrates the method:-

```
INPUT       BIT #2DF
            BPL INPUT
            LDA #2DF
            PHA
            LDA @0
            STA #2DF
            PLA
            AND @#7F
            RTS
```

The reason that this routine works is that the location #2DF not only contains the ASCII code of the character typed in at the keyboard but also has its top bit set when a new character is present, thus making it appear to be negative. The input routine uses the BIT command to test the sign of the location and when this goes negative the contents are loaded into the accumulator and saved on the stack. The accumulator is then cleared and stored in #2DF, thus clearing that location and preventing a single character from being read twice. The saved contents are then pulled off the stack and the top bit stripped off before returning to the calling program. Remember that the character is put into location #2DF during an

interrupt routine, which is transparent to this polling routine.

When this particular routine is called, it will not return until a key is pressed, which may not be exactly what is wanted. It may be that a program merely wants to know if a key has been pressed and only if one has to fetch the code at this location. If no key has been pressed, carry on anyway. This next routine achieves this end:-

```
INPK    BIT #2DF
        BMI GOTU
        CLC
        RTS
GOTU    LDA #2DF
        PHA
        LDA @0
        STA #2DF
        PLA
        AND @#7F
        SEC
        RTS
```

This routine uses the same mechanisms as previously, but with one addition. The carry flag is used to signal to the calling program whether or not a character was found. If this routine returns with the carry flag set, then the accumulator contains the character code. If the carry flag is clear, then there is no character present. Thus the calling program would be something like:-

```
        JSR INPK
        BCC NONE
CHAR    ....
        .....
```

where NONE is the label for the place the program goes to when there is no character ready and CHAR is where it goes when one is ready.

There is a special routine for printing messages on the status line of the text screen. This routine starts at #F82F and on entry it requires the registers to contain the following information.

Accumulator	Message address (low)
Y-Register	Message address (high)
X-Register	Horizontal position to start (0-39)

The message itself must end with a zero byte. This routine uses locations #0C and #0D in zero page to set up the indirect address of the message and it then effectively POKES the characters directly into screen RAM. Try the following program:-

```
10 I=0
20 REPEAT
30 READ A
```

```
40 POKE #400+I,A
50 I=I+1
60 UNTIL A=#FF
70 DATA #A9,#8D,#A0,#EB,#A2,#00
80 DATA #20,#2F,#F8,#60,#FF
90 CALL #400
```

This program should tell you who the culprits are!

V1.1 Calls.

This version of the operating system is much more advanced, as might be expected, and it contains a greatly enhanced range of calls which are available to the user. We shall start with the output and input routines as before.

The call to print a character on the screen now resides at #F77C and the character to be printed must be in the X-register, not the accumulator. This routine will also obey all the standard control codes. Its use is exactly as in V1.0 with the exceptions, thus;

```
OOTPT     LDY @0
NEXT      LDX MSG,Y
          BMI OUTEND
          JSR #F77C
          INY
          JMP NEXT
OUTEND    RTS
MSG       ASC "HELLO THERE"
          DFB #FF
```

There is now a call to read the keyboard which lives at #EB78. This routine is effectively the same as the second of the routines listed in V1.0 for reading a character in that this routine always returns after being called with the accumulator containing the character found. If the accumulator is negative, then it also contains a valid character, if it is positive then no character was found. Thus to see if a key has been pressed:-

```
LOOK      JSR #EB78
          BMI GOTU
NOKEY     ....
          ....
```

where again GOTU is where the program goes to when a character has been typed and NOKEY is where it goes when one has not.

The special status line message routine now lives at #F865 and requires the registers setting up on entry as before. The following program demonstrates its use:-

```
10 I=0
20 REPEAT
30 READ A
40 POKE #400+I,A
50 I=I+1
60 UNTIL A=#FF
70 DATA #A9,#B7,#A0,#ED,#A2,#00
80 DATA #20,#65,#F8,#60,#FF
90 CALL #400
```

You should now see the word "TANGERINE" at the start of the status line and the word "CAPS" at the end of the line has gone into double height characters, or at least, it has tried to but because the line underneath is blank only the top half of the characters is visible.

There is another output call to send characters directly to the printer. In this case the character to be sent must be in the accumulator. The state of the other registers is irrelevant. All registers will have been corrupted when this routine returns. Its address is #F5C1. On return the accumulator contains the interrupt flag register of the internal 6522. This routine will return if the printer is not plugged in, but will wait for the printer to be ready if it is plugged in and busy.

There is also a complete set of cassette interface routines. The first resides at #E75A and its job is to output a leader to the tape so that subsequent loading or reading operations can synchronise to the leader to prevent error. The character sent to the recorder is ASCII SYN (Hex 16). The makers claim that 9 characters will be sent to the recorder but looking at the code actually in the ROM suggests that 259 characters will be sent. The actual code is:-

```
E75A        LDX @02
E75C        LDY @03 (HERE)
E75E        LDA @#16
E760        JSR #E65E
E763        DEY
E764        BNE #F8 (E75E)
E766        DEX
E767        BNE #F5 (E75E)
E769        RTS
```

To test out how many SYN characters are sent to the cassette recorder, rewrite this routine slightly and move it into RAM as:-

```
START   400   LDA @0        A9 00
        402   STA 8         85 08
        404   STA 9         85 09
        406   LDX #2        A2 02
        408   LDY #3        A0 03
NEXT    40A   JSR INCR      20 04 14
        40D   DEY           88
```

```
        40E   BNE NEXT    D0 FA
        410   DEX         CA
        411   BNE NEXT    D0 F7
        413   RTS         60
INCR    414   INC 8       E6 08
        416   BNE OUT     D0 02
        418   INC 9       E6 09
OUT     414   RTS         60
```

This is exactly the same routine as before except that it does not load the accumulator with #16 and the cassette output routine has been substituted by a routine that simply counts the number of times it has been called. When this program is run, the resulting number held in #8 and #9 is 259. To make the synchronise routine perform its advertised task two changes are necessary in the source listing resulting in three changes in the machine code. The first change is to move the branch address back by one instruction, to the place marked HERE. The second is to load the X-register with 3 and not 2. Sadly neither of these changes can be made by a user.

The corresponding routine to this one synchronises to the leader just produced. This routine is at #E735. The last two cassette routines are to write a byte to the cassette recorder and to read a byte from it. The write routine is at #E56E and the read routine at #E6C9. These routines exist so that an advanced programmer can use the cassette recorder to store data directly from within his machine code program and can, of course, restore that data from within the same program. One possible use of such mechanisms is in word-processing where text has to be saved and retrieved if the system is to be of much use.

7.2 Soft Vectors

A vector is an address stored in two consecutive memory locations. When a particular event occurs the processor is directed to that address by reading the contents of these two locations. The 6502 processor has three events which cause it to read certain locations and treat their contents as addresses.

The first such event is the power-on or reset event. It must be obvious that when a micro-processor is first switched on it must perform a fixed sequence of tasks, the last of which is to go to a known address and either use the contents of that address as code or as the address of the start of the code. When the 6502 is first switched on it takes the contents of locations #FFFC and #FFFD and puts them into the program counter as address low and address high respectively. It then starts executing the code found at the address thus generated.

The other two events are the two interrupts which the 6502 allows. The first interrupt is called the Non-Maskable interrupt (NMI for short) and this is vectored through locations #FFFA and #FFFB. This interrupt is called non-maskable because the 6502 cannot prevent itself being interrupted by this event.

The second interrupt is maskable which means that the 6502 can ignore such requests until it is ready for them. This is achieved by setting or clearing a bit in the status register of the machine. See Chapter 6 for the two instructions SEI and CLI which do just this. This interrupt is vectored through locations #FFFE and #FFFF.

Soft Vectors (V1.0)

The vectors found in the memory locations just detailed are;

#FFFA	&	#FFFB	#022B	NMI
#FFFC	&	#FFFD	#F42D	POWER ON
#FFFE	&	#FFFF	#0228	IRQ

Power on or reset is not a 'soft' vector but it is included here for completeness.

The code found at #022B is #4C, #30, #F4 which translates into 'jump to location #F430'. That found at #0228 is #4C, #03, #EC meaning 'jump to location #EC03. Both of these vectors can be changed and used for other purposes. However the user's IRQ routine should end by jumping to #EC03 or the ORIC will lose the use of its keyboard.

There is another way of using interrupts which are not generated by the internal timer. When an interrupt occurs the ORIC checks the timer to see if that was the cause of the trouble. If it was not a jump to #230 is made. The code normally found here is #40 meaning RTI. Obviously this code can be replaced by user's own code to deal with externally generated interrupts. Note that such code should preserve all registers.

Soft Vectors (V1.1)

The table of vectors in this version is as follows:-

#FFFA	&	#FFFB	#0247	NMI
#FFFC	&	#FFFD	#F88F	POWER ON
#FFFE	&	#FFFF	#0244	IRQ

The code found at #247 is #4C, #B2, #F8 meaning 'jump to location #F8B2' and that found at #244 is #4C, #22, #EE meaning 'jump to location #EE22'.

The return from interrupt handler is now at #24A and there is space available to patch this area to a jump, as at #247, but no more.

7.3 Special Commands.

There are two undefined commands which are left for the user to write his own. A sort of do-it-yourself facility. Both of these commands execute jumps and the jump address can be altered to suit the user's own routines.

The first of these commands is the '!'. This causes the processor to execute a jump indirect on #2F5. The normal code here is the address of the error routine. This address differs form version to version, but its effect is the same. The code at #2F5 can be changed by a user to make this command jump to his own routine. One such command is demonstrated in Chapter 8. The method is fairly straightforward. Firstly, set up the machine code routine that performs the required task, making sure that the routine does not produce a net alteration in the processor's stack and that it ends in RTS. Secondly, set the contents of locations #2F5 and #2F6 to point to the entry address of the routine, just like any other soft vector.

The simplest example is to put the code #60 at #400, meaning RTS, and type in DOKE #2F5,#400. Now type !(RETURN) and no error message is printed. The processor has been directed to the RTS instruction and it has come safely back again.

The second of these commands is the '&'. This time the soft vector is at #2FC and #2FD, preceded by the code #4C at location #2FB meaning JUMP. Thus the contents of #2FC and #2FD can be changed in the same way as before to give another user defined command. There is an additional twist to this command because the ORIC views this as a function call, in the same way as in BASIC the FN command sends the processor off to the defined function. This means that the '&' command can pass results to the processor through the floating point accumulator. In fact a number has to be passed using this call or an error will result. The correct syntax for this command is '&(N)' where N can be any number. The result of this command is firstly that the argument N is normalised and put into the floating point accumulator and then the code beginning at #2FB is executed. To make full use of this command requires some understanding of floating point arithmetic.

The floating point accumulator in the ORIC is from #D0 to #D5 with these locations used as follows:-

#D0	EXPONENT
#D1	MANTISSA(HIGH)
#D2	MANTISSA
#D3	MANTISSA
#D4	MANTISSA(LOW)
#D5	SIGN

A floating point number is stored as +- a*2↑n where 'a' is the mantissa and 'n' the exponent. This method of representing numbers is not as fearsome as it looks. For example the number 8 would be represented by 1*2↑3. Thus the argument to the '&' command is stored in the floating point accumulator in this format. This number is 'normalised' first meaning that maximum use is made of the storage facilities to keep as many decimal places in the computer as possible. To demonstrate again, the number 32 if normalised to a decimal format would be stored as .32*10↑2. If stored in binary format it would be .1*2↑6. You will notice that normalised numbers have no digits to the left of the decimal point, and that they are organised so that the mantissa has its first non-zero digit immediately to

the right of the point and the exponent is adjusted accordingly.

The way the ORIC uses its locations is shown by the way the numbers +8, -8, and 1 are stored:-

#D0	#84	#84	#81
#D1	#80	#80	#80
#D2	#00	#00	#00
#D3	#00	#00	#00
#D4	#00	#00	#00
#D5	#00	#FF	#00
	+8	-8	1

Thus, if #D5 is negative, the number is negative. Also the exponent is actually (#80+N) where N is the actual power. This gives a simple method of dealing with exponents of the type a-10, because now all exponents are positive.

Unfortunately the designers of the ORIC have not published the entry points for their own floating point routines so users will have to make up such routines as they require. There are a number of books on the market which describe floating point arithmetic in some detail and interested readers should obtain one of these.

7.4 BASIC Entry Points.

A number of the graphics and sound routines have been made available to the machine code programmer, but only in V1.1. The majority of these routines require some parameters to be passed to them and an area of memory has been set aside starting at #2E0 for this purpose. The address #2E0 will be referred to as the label PAR in the rest of this chapter, thus PAR+1 will mean address #2E1. In all cases the location #2E0 should be set to minus one before the routine is called. Also note that all parameters passed can be two bytes long and that the least significant byte is the first byte, which is as usual for a 6502. The called routines actually increment the location #2E0 if an error is found in the calling parameters, so setting this to -1 means that if an error is found this location will be zero when the routine returns. If there is no error, it will be negative and can be tested using the BIT command.

GRAPHICS

ROUTINE	ADDRESS	PAR+1	PAR+3	PAR+5	REGISTERS CORRUPTED
CURSET	#F0CB	X-VALUE	Y-VALUE	FB VALUE	A,X,Y
CURMOV	#F0FD	X-VALUE	Y-VALUE	FB VALUE	
DRAW	#F110	X-VALUE	Y-VALUE	FB VALUE	

CHAR	#F12D	CHAR	0=STD;1 =ALT	FB VALUE	
CIRCLE	#F37F	RADIUS	FB VALUE		
PATTERN	#F11D	PATTERN	VALUE		
POINT	#F1C8	X-VALUE =F/GND	Y-VALUE	On return PAR+1=0=B/GND;=1	
FILL	#F268	No.rows	No. cells Value		
PAPER	#F204	Colour			
INK	#F610	Colour			

SOUND

PING	#FA9F				
SHOOT	#FAB5				
EXPLODE	#FACB				
ZAP	#FAE1				
KBEEP	#FB14	Normal	keyclick		
CONTBP	#FB2A	Control	keyclick		
SOUND	#FB40	Channel	Period	Volume	
MUSIC	#FC18	Channel	Octave	Note	PAR+7=Volume
PLAY	#FBD0	Tone ch.	Noise Ch	Env. mode	PAR+7=Env Per.
W8912	#F590	Acc.=reg.No	X=o/pdata		A,X,Y

The last call allows an advanced programmer to write directly into the registers of the W8912 sound chip. In fact all of the above routines will only be of use to advanced programmers and the author hesitates to tell such people how to use system calls!

V1.0 Screen Control

The V1.0 operating system stores information about the video display in RAM locations #268 to #26F. The system uses these locations to keep track of where the screen memory starts, how many rows can be printed, where the cursor is and so on. By juggling with the contents of these locations various effects can be achieved.

Locations #26D and #26E contain the address of the start of screen RAM. This address is #BB80 in TEXT and LORES modes and #BF40 in HIRES mode. Location #26F contains the number of rows avaialble on the screen. In TEXT and LORES modes this number is 27 and in HIRES mode it is 3. Thus the TEXT screen starts at #BB80 and can print out on 27 lines.

In HIRES mode the text screen starts at #BF40 and can print 3 lines.

The use that can be made of this information is in setting up permament headings on the screen which cannot be scrolled away or cleared by the usual methods. Try

this short program:-

```
10 CLS
20 PRINT "HEADING LINE 1"
30 PRINT "HEADING LINE 2"
40 PRINT "HEADING LINE 3"
50 PRINT "HEADING LINE 4"
60 DOKE #26D,#BC20
70 POKE #26F,23
80 FOR I=0 TO 40
90 PRINT "NEXT LINE"I
100 NEXT I
```

When this program is run, the four "HEADING" lines remain fixed at the top of the screen while the rest of the screen is scrolled.

This technique can be adapted to include footings by reducing the number of lines to be scrolled. Before doing the modification, reset using the underneath button and then add the line:-

```
65 PLOT 1,26,"FOOTING"
```

and change line 70 to:-

```
70 POKE #26F,22
```

Now the word "FOOTING" will remain at the bottom of the screen as well as the four heading lines at the top and only the area of screen in between these will be scrolled.

The ORIC stores information about the current cursor position in #268 and #269. #268 contains the number of the row the cursor is on (0 to 27) and #269 contains the column (0 to 39). By altering these numbers the current cursor location can be changed and hence printing can be made to occur at any position on the screen. This can already be done from BASIC using the PLOT command but machine code programs do not have easy access to the BASIC routines.

V1.1 Screen Control

The screen control subroutines have been completely re-written for V1.1 and the old locations are not used. Now locations #27A and #27B and #278 and #279 contain the screen addresses. These addresses are the start of the first and second text lines respectively. Unfortunately these addresses are not used in the same way as before and setting a protected header under V1.1 is not done the same way. Thus to set a protected header these two addresses must be changed together. Further, to keep the display readable, the locations #27A and #27B should contain the address of the start of a line (#BBA8, #BBD0, #BF8, #C20,...etc) and locations #278 and #79 the address of the start of the next line. If these criteria are not met, strange and unpredictable things can happen. Thus to set a header use:-

```
10 CLS
20 PRINT "HEADING LINE"
```

```
30 DOKE #278,#BBF8
40 DOKE #27A,#BBD0
50 FOR I=0 TO 40
90 PRINT "NEXT LINE"I
100 NEXT I
```

This leaves just one line at the top of the screen as a header. To produce a header of more then one line is apparently not possible, at least not if the screen is to be scrolled more than about 4 times. The problem arises with the frame synchronisation generation and there does not appear to be a simple solution.

The cursor control is also different under V1.1 with location #268 containing the line number the cursor is thought to be on and location #27E containing the number of lines on the screen. Changing location #27E which normally holds 27 will alter the number of lines scrolled. Thus changing this location to 26 will leave the bottom line unscrolled. Changing location #268 merely changes the line number which the computer thinks the cursor is on. The computer treats this location as a counter and continually compares its contents with location #27E. When they are equal, the screen is scrolled on the next line feed. Consequently changing the contents of location #268 can induce early scrolling or delay scrolling. This gives another means of protecting a footing.

Both of these versions use location #26A as a mode byte, the functions of which have already been detailed in Chapter 4. However, here is the table again for completeness:-

BIT	FUNCTION
7	SPARE
6	SPARE
5	MASK FOR COLUMNS 0,1 1=MASK ON
4	1=LAST CHARACTER WAS ESCAPE
3	1=KEYCLICK OFF.0=KEYCLICK ON
2	SPARE
1	1=VDU ON.0=VDU OFF
0	1=CURSOR ON.0=CURSOR OFF.

The only function which may need amplifying is that for bit 5. If this bit is set to a '1' subsequent characters can be printed in the protected columns which control the background and foreground attributes in TEXT mode.

7.5 Simple De-Bugging Techniques For Assembly Language.

While it is relatively easy to write programs in assembly language once you have acquired the knack, it is not so easy to de-bug those same programs on a machine like the ORIC. Having spent ages typing in all the data statements that go into making up your masterpiece you then type CALL #400, or wherever and the machine goes away and hides. What has happened and why? Hours spent studying the listings and comparing the data in the machine with what you worked out will not reveal the answer. The only way is to actually watch the program at work. De-bugging programs can be bought and these are more or less useful depending, largely, on how much is paid for them and on who wrote them. What is presented here is a de-bugging method which can be adopted by anyone. Naturally it requires some machine code and the version presented here has all been de-bugged and is known to run. This will hopefully provide a starting place for your own, more complicated de-bugging tools.

The centre of all de-bugging is to print out on the screen the contents of the program counter, the accumulator, and the other registers. Without this information no real work is possible. Since BASIC already has handy printing commands, these will be used. Thus the program will be a mixture of BASIC and machine code.

The simplest de-bugging technique of all, known as break-pointing, is to remove some of the program being tested and insert a jump to a known subroutine. After this jump has taken place, the removed piece of code can be replaced or not, depending on the style you wish to adopt. For example, suppose a program such as this is being tested:-

```
400     LDA @#40
402     STA #48
404     LDA @#FA
406     STA #09
408     LDY @#0
40A     LDA (8),Y
```

Provided the absolute addresses of all the statements in the program are known, as shown above, then the method is to replace some of the code with a JSR ADDRESS command as shown:-

```
400     LDA @#40
402     STA #08
404     JSR #B000
407     RTS
408     LDY @#0
40A     LDA (8),Y
```

When the machine code program is run it will jump to location #B000 after the second instruction of the original program has been executed. At location #B000 there is a small segment of machine code which saves all the processor's registers

in zero page. The inserted RTS instruction returns the processor to BASIC. Notice that JSR instructions are used here because they ensure that the current program counter is saved on the stack and can therefore be pulled off and stored for later examination. The BASIC section of the program prints out these locations nicely formatted on the screen and then waits for the next command from the user.

To set up such a system the program must first POKE the required machine code segment into #B000, or wherever, and then prompt the user for a command. The first command must be to set a breakpoint, which results in the machine code program under test being modified as above. This routine must save the old data from the machine code under test because it will have to be replaced later. The user can then tell the de-bugger to start running the machine code program under test at an address before the breakpoint address. The de-bugging program simply does a CALL #ADDRESS because the routine at#B000 ends with RTS. Hence the statements after the CALL # are formatted print commands to show the registers.

Here follows a simple de-bug program:-

```
10 DIM P(10):CLS PRINT
20 FOR T=0 TO 10
30 PRINT CHR$ (9) ;
40 NEXT I
50 PRINT CHR$(4) " "CHR$(27) "J" CHR$(27)"ASIMPLE DEBUG"
60 PRINT:PRINT CHR$(4):PRINT SPC (5);
70 PRINT CHR$(27)"DP.C. A  X  Y  ST  SP"
80 DOKE $26D,$BC48
90 POKE $26F,22
PRINT HEX$(DEEK(4))" "HEX$(PEEK(6))" "HEX$(P



100 GOSUB 1000
110 PRINT:PRINT:PRINT:PRINT
120 INPUT"BREAKPOINT ADDRESS= ";A
130 P(1)=PEEK(A):P(2)=PEEK(A+1):P(3)=PEEK(A+2):P(4)=PEEK(A+3)
140 POKE A,32:POKE A+1,0:POKE A+2,144:POKE A+3,96
150 INPUT"START ADDRESS= ";B
160 PRINT" PRESS ANY KEY WHEN READY"
170 GET A$
180 CALL B
190 CLS
200 PRINT SPC(5);
210 PRINT HEX$(DEEK(4))" "HEX$(PEEK(6))" "HEX$(PEEK(7))" "HEX$
(PEEK(8))" "HEX$(PEEK(9))" ";
220 PRINT HEX$(PEEK(3))
230 GOSUB 2000:GOTO 110
```

```
1000 I=0
1010 REPEAT
1020 READ A
1030 POKE #9000+I,H
1040 I=I+1
1050 UNTIL A=#FF
1060 RETURN
1070 DATA #85,#06,#86,#07,#84,#8,#08
1080 DATA #68,#85,#09,#BA,#86,#03,#68
1090 DATA #85,#32,#38,#E9,#02,#85,#04
1100 DATA #68,#85,#31,#E9,#00,#85,#05
1110 DATA #A5,#31,#48,#A5,#32,#48,#60
1120 DATA #FF
2000 POKE A,P(1):POKE A+1,P(2):POKE A+2,P(3):POKE A+3,P(4)
2010 RETURN
```

Notice that this program always leaves the machine code program under test in the condition in which it found it. This method is considered to be safer than having to clear breakpoints after debug because it is so easy to forget to clear those points and save a program to cassette with a breakpoint still embedded in it. Also this program is designed to work under V1.0. Readers with V1.1 will have to modify lines 190 on and use the PRINT @ X,Y;STRING command to produce the desired effect. Also lines 80 and 90 must be ommited under V1.1.

Lastly because this de-bugger is so very simple it will not like it if the machine code under test produces a net change in the stack. To put it another way, if the tested program pulls the stack, and then does not push it back before the breakpoint is met, catastrophe will probably result. It is possible to write de-buggers that split the stack so that the de-bugger has one half and the program under test the other but this is beyond the scope of a simple program.

CHAPTER 8

Useful ORIC Programs

This chapter is a collection of programs which demonstrate some of the abilities of the ORIC computer. Some of these programs are intended to make life easier for the home programmer while programs are under development. Others are just examples of techniques or facilities provided. We shall first look at ways of handling data on the screen.

8.1 Simple BASIC.

Sideways scroll (left)

```
10 FOR I=0 TO 26*40 STEP 40
20 FOR J=0 TO 36
30 A= #BBAA+J+I:B=A+1
40 POKE A,(PEEK(B))
50 NEXT J
60 NEXT I
```

All BASIC programs using FOR...NEXT loops take a long time to execute, so this could be better described as a sideways ripple. If converted to machine code the results should be fast enough for anyone.

Sideways scroll (Right)

```
10 FOR I=0 TO 26*40 STEP 40
20 FOR J=36 TO 0 STEP -1
30 A=# BBAA+J+I:B=A-1
40 POKE A,(PEEK(B))
50 NEXT J
60 NEXT I
```

The same effect as before but this time going the other way.

Upwards scroll

```
10 FOR I=0 TO 25*40 STEP 40
20 FOR J=0 TO 36
30 A= #BBAA+J+I:B=A+40
```

```
40 POKE A,(PEEK(B))
50 NEXT J
60 NEXT I
```

Downwards scroll

```
10 FOR I=25*40 TO 0 STEP -40
20 FOR J=0 TO 36
30 A=#BBAA+J+I:B=A-40
40 POKE A,(PEEK(B))
50 NEXT J
60 NEXT I
```

Fill screen with random characters.

```
10 FOR I=0 TO 26*40 STEP 40
20 FOR J=0 TO 36
30 A=#BBAA+J+I
40 POKE A,(93*RND(1)+32)
50 NEXT J
60 NEXT I
```

This completes a family of programs all based on the same arithmetic. The location $BBAA is the memory location corresponding to the first printable character position on the screen. The number of printable characters on a line is 38, but lines are 40 characters apart because of the 2 serial attributes at the left hand edge of the screen. These scrolling programs will scroll entire screen contents, which includes serial attributes, so that predefined colour blocks will move with the characters they were originally associated with.

The next program demonstrates the use of serial attributes in producing apparent motion. The ORIC only has one screen 'page' (i.e. it is not possible to draw or write on two pages and flip the screen between the two), so programmers must make the best use possible of the facilities provided.

Serial Attributes

```
10 CLS
20 FOR I=0 TO 11
30 PLOT 19+I,I+2,"*"
40 PLOT 19-I,I+2,"*"
50 NEXT I
60 FOT I=11 TO 0 STEP -1
70 PLOT 19+I,24-I,"*"
80 PLOT 19-I,24-I,"*"
90 NEXT I
100 FOR I=0 TO 26
110 PLOT 0,I,7
120 NEXT I
```

```
130 FOR I=3 TO 23
140 PLOT 19,I,7
150 NEXT I
160 I=1
170 REPEAT
180 I=I+1
190 PLOT 0,I,0:WAIT 10
200 PLOT 0,I,7
210 UNTIL I=24
220 REPEAT
230 I=I-1
240 PLOT 19,I,0:WAIT 10
250 PLOT 19,I,7
260 UNTIL I=3
270 GOTO 160
```

Lines 10 to 90 clear the screen and draw a diamond shape in asterisks. Lines 100 to 150 change the left hand edge attributes for foreground colours to white and put a second column of foreground attributes down the centre line of the diamond. The diamond is now invisible. Lines 160 to 210 change the foreground attributes down the left hand edge of the screen sequentially so that the first is for black, and all the rest white; the second is for black, and the rest white; the third is for black and so on until they have all been set for black once and back to white again. Lines 220 to 260 do the same but for the second column of attributes. Line 270 makes the whole movement sequence repeat.

When developing programs,it sometimes happens that a line or two is missed out when the program is first typed in! The missing lines then have to be inserted and this ruins the even spacing of the line numbering sequence 10, 20, 30...etc. It can also happen that so many lines have to be inserted, there is no longer any room for them, unless fractional line numbers are allowed! Line 10 1/2 is likely to be rejected however. For these reasons, a renumbering program is quite useful. The following short program should be added on to the end of a program under development and left there until development is complete, at which time this segment can be removed and the final version saved, numbered in a beautifully even sequence.

Renumbering Program

```
10 REM PROGRAM
20 REM UNDER
30 REM DEVELOPMENT
40 REM WOULD BE
50 REM HERE
60 END
70 I=#501
80 INPUT"START AT";J
90 INPUT"STEP";T
100 REPEAT
```

```
110 A=DEEK(I)
120 DOKE(I+2),J:J=J+T
130 I=A
140 UNTIL A=0
150 END
```

When the time for renumbering comes, type in the immediate command GOTO 70 (or whatever the line number the statement I=#501 is on) and set up the constants as requested. When you list your program, all will be as smooth as you could wish.

With this system you can renumber upwards or downwards. That is, you can go from 10, 20, 30, to 5, 10, 15, or from 1, 5, 7, 11 to 10, 20, 30, etc as desired. The program is very simple, however, and does not check for statements of the type GOTO 75 and hence if your program contains any of these, you will have to sort these out yourself. As an aid to this kind of sorting, always include a REM statement immediately before the line number which is to be referenced. For example:-

```
130 REM SCREEN DUMP ROUTINE
140 I=# BBAA
150 .....
```

etc.

Line 140 is the line referenced by a GOSUB or GOTO statement and after renumbering it is now relatively easy to find because there is a REM statement immediately before it. To make it really stand out, use:-

```
130 REM**************
```

and at the GOTO or GOSUB use:-
```
60 GOSUB 130 '************
```
A second GOSUB or GOTO would use:-
```
170 REM@@@@@@@@@@@@@@@
```
and be referenced by:-
```
50 GOSUB 170 '@@@@@@@@@@@@@
```
and so on.

When the program is complete, the REM's can be removed so that the space consumed by the program is reduced. This may have to be done before final completion on a 16K machine and is the reason for not referencing REM statements in GOTO and GOSUB commands.

String Arrays

Although the manual for the ORIC does not state the fact explicitly, the computer does support the use of an array of strings. This is very useful when required to arrange entries in alphabetical order or in order of achievement. The following

program sorts names into alphabetic order:-

```
10 CLS
20 INPUT"NUMBER OF NAMES=";N
30 DIM A$(N+1)
40 FOR X=1 TO N
50 INPUT "NAME IS?";A$(X)
60 NEXT X
70 CLS:M=2
80 GOSUB 180
90 FOR X=1 TO N
100 FOR J=N TO X+1 STEP -1
110 IF A$(J)<A$(J-1) THEN B$=A$(J-1):A$(J-1)=A$(J):A$(J)=B
120 M=12
130 GOSUB 180
140 NEXT J
150 NEXT X
160 END
170 REM PRINT ROUTINE
180 FOR K=1 TO N
190 PLOT M,(5+K),A$(K)+"    "
200 NEXT K
210 RETURN
```

Lines 20 to 60 ask for the number of entries, then the entries themselves and stores them in an array of strings. Lines 70 to 80 print out the entries in their original order, down the left hand side of the screen. Lines 90 to 150 sort the entries as a 'bubble sort' and print each effort on the screen next to the original order. At the end of the program the screen display shows both the original order and the finally sorted order, side by side.

The only point to note is the PLOT statement in line 190 where several spaces are added on to each string as it is PLOTted. The reason for this is to make sure that a short string completely overwrites and erases a longer one. Without these spaces, if FRED were to replace MICHAEL, the actual result on the screen would be FREDAEL.

As the program stands, the entries are sorted with the smallest value string at the top, lowest value strings at the bottom. This can be reversed by changing the '>' sign in line 190 to '<'.

To make the workings of this program even more watchable the following listing is included. This is reproduced in full even though it duplicates some of the above program in order to avoid any mis-understanding.

```
10 CLS
20 INPUT"NUMBER OF NAMES=";N
30 DIM A$(N+1)
40 FOR X=1 TO N
50 INPUT "ENTRY=";A$(X)
```

```
60 NEXT X
70 CLS:M=2
80 GOSUB 180
90 M=20
100 GOSUB 180
110 FOR X=1 TO N
120 FOR J=N TO X+1 STEP -1
130 IF A$(J)<A$(J-1) THEN GOSUB 230
140 NEXT J
150 NEXT X
160 END
170 REM*******
180 FOR K=1 TO N
190 PLOT M,(5+K),A$(K)+"    "
200 NEXT K
210 RETURN
220 REM@@@@@@@@
230 B$=A$(J):C$=A$(J-1)
240 FOR P=0 TO 7
250 WAIT 10
260 PLOT M-P,5+J,B$+"    "
270 PLOT M+P-1,4+J," "+C$
280 NEXT P
290 PLOT M-P,5+J,"          "
300 PLOT M-P,4+J,B$
310 PLOT M+P-1,4+J,"          "
320 PLOT M+P-1,5+J,C$
330 FOR Q=0 TO 7
340 WAIT 10
350 PLOT M-P+Q,4+J," "+B$
360 PLOT M+P-1+Q,5+J,C$+"    "
370 NEXT Q
380 A$(J)=C$ :A$(J-1)=B$
390 RETURN
```

Provided you type this in with no mistakes, you should see the action of a 'bubble sort' quite clearly on your screen.

This type of program also lends itself to sorting out a class list in order of achievement, in an examination for example. Again, this next program has some duplication.

```
10 CLS
20 INPUT"NUMBER OF NAMES=";N
30 DIM A$(N+1)
40 FOR X=1 TO N
50 INPUT"NAME=";A$(X)
60 INPUT"SCORE=";A(X)
70 NEXT X
```

```
 80 CLS
 90 FOR X=1 TO N
100 FOR J=N TO X+1 STEP -1
110 IF A(J)<A(J-1) THEN GOSUB 170
120 NEXT J
130 NEXT X
140 GOSUB 230
150 END
160 REM*****
170 T=A(J):U=A(J-1)
180 B$=A$(J):C$=A$(J-1)
190 A(J)=U:A(J-1)=T
200 A$(J)=C$:A$(J-1)=B$
210 RETURN
220 REM@@@@@
230 FOR K=1 TO N
240 R$=STR$(A(K))
250 PLOT 10,5+K,A$(K)
260 PLOT 20,5+K,B$
270 NEXT K
280 RETURN
```

As this program stands, the scores will be printed in green due to a peculiarity associated with the STR$ command. To overcome this slight problem, modify line 240 to read:-

```
240 R$=STR$(A(K)):R$=RIGHT$(R$,(LEN(R$)-1))
```

This now strips out the first character of R$ which is the serial attribute responsible for the green colour.

8.2 LORES Graphics.

This program is an automatic histogram maker. It uses low resolution graphics to achieve its ends and to make life simple, a block character is defined as the character used to produce the vertical columns of the histogram. This limits the range of the result to 26 and makes the resolution (1/26)*100%= 3.8%. To obtain better resolution would require adapting this program to use the high resolution mode of the ORIC, making it altogether more complicated and defeating the object, which was to produce a simple, colourful histogram:-

Histograms

```
10 CLS
20 FOR I=0 TO 7
30 POKE(46856+I),255
40 NEXT I
50 DIM C(50),V(50)
```

```
60 PRINT CHR$(17)
70 GOTO 90
80 PRINT "TOO MANY COLUMNS"
90 INPUT"NUMBER OF COLUMNS=";N
100 IF N>17 THEN 80
110 FOR J=1 TO N:F=0
120 PRINT "COLOUR OF "J"th COLUMN=";
130 INPUT C(J):IF C(J)>7 THEN PRINT"COLOUR VALUE TOO GREAT":GOTO
    120
140 INPUT"VALUE OF COLUMN=";V(J)
150 IF V(J)>26 THEN PRINT"VALUE TOO GREAT":GOTO 140
160 NEXT J
170 LORES 0
180 FOR J=0 TO 26
190 PLOT 0,(26-J),STR$(J)+"-"
200 NEXT J
210 FOR K=3 TO 37 STEP 2
220 PLOT K,26,"-"
230 NEXT K
240 FOR J=1 TO N:D+2*J+2
250 FOR K=1 TO V(J)
260 PLOT D,(26-K),C(J)
270 PLOT D+1,(26-K),"a"
280 NEXT K
290 NEXT J
300 GOTO 300
```

This program is a little cunning in one or two places, but is not difficult to understand. Lines 10 to 40 clear the screen and re-define, the character 'a' as a solid block. Lines 50 to 70 set aside memory for the two arrays needed by the program and turn the cursor off. Lines 80 to 160 ask for colour information and value information for each column required. Lines 170 to 200 put the machine into LORES mode and draw a vertical axis with numbers on it. Lines 210 to 230 draw a horizontal axis using the characters '–' and ". Lines 240 to 290 actually plot each column up the screen by plotting a serial attribute first and then the redefined character 'a'. Each column is separated from its neighbour by the serial attribute for the next column. It would also be possible to include a legend at the top right hand corner of the screen showing which colour represented which quantity. This is left to the reader as an exercise.

8.3 HIRES Graphics.

This program is a demonstration of high resolution colour graphics and is slightly more restful than watching goldfish.

Relaxation!

```
10 HIRES
```

```
20 FOR N= #A000 TO #BF18 STEP 40
30 POKE N+1,INT(RND(1)*7)+1
40 POKE N,INT(RND(1)*7)+16
50 NEXT N
60 CURSET 120,100,3
70 FOR X=95 TO 1 STEP -1
80 CIRCLE X,1
90 NEXT X
100 A=1:D=23
110 I=0
120 REPEAT
130 B=40961+40*I:C=#BF18-40*I
140 POKE B,A:POKE C,D
150 I=I+1
160 UNTIL B>#BF17
170 A=A+1:IF A>7 THEN A=1
180 D=D-1: IF D<16 THEN D=23
190 GOTO 110
```

Line 10 puts the machine into HIRES mode. Lines 20 to 50 plant random background and foreground colour attributes down the left hand edge of the screen. Lines 60 to 90 draw a set of concentric circles in the middle of the screen, these circles being seen in random foreground colours. Lines 100 to 180 sequentially alter the background and foreground attributes in the very left hand columns, cycling through all possible attributes for each. These cycles are arranged so that the foreground attributes change from the top down and the background ones from the bottom up. The HIRES mode errors due to digitisation can be seen as background colours 'spotting through' the foreground colour disc.

Another use of a high resolution computer is to draw graphs. These graphs can either be results obtained in an experiment, or a mathematical function under investigation. The next program sets up axes centred on the screen and then puts out any desired polynomial. (A polynomial is an equation of the form Y=AX↑N + BX↑(N-1) +CX↑(N- 2) ... etc. A quadratic equation is a polynomial or order 2. Thus Y=AX↑2 + BX +C, which is well known to mathematical students is a polynomial of order 2 with coefficients A, B, C of X↑2, X↑1, and X↑0. This first program simply plots the relevant points.

Plotter

```
10 CLS
20 INPUT"ORDER OF POLYNOMIAL";O
30 FOR I=0 TO O
40 PRINT "COEFFICIENT OF X"I"=;:INPUT A(I)
50 NEXT I
60 HIRES
70 CURSET 120,100,3
80 DRAW 100,0,1:DRAW -200,0,1
90 CURSET 120,100,3
```

```
100 DRAW 0,-100,1:DRAW 0,199,1
110 X=-10
120 REPEAT
130 X=X+1:GOSUB 180
140 CURSET A,B,1
150 UNTIL A=239
160 END
170 REM*******
180 Y=0
190 FOR I=0 TO 1 STEP -1
200 Y=Y+A(I):Y=Y*X
210 NEXT I
220 Y=Y+A(0):B=100-Y
230 IF B>199 THEN B=199
240 IF B<0 THEN B=0
250 A=X*12+120
260 IF A>239 THEN A=239
270 RETURN
```

Lines 10 to 50 ask the order of the polynomial and the coefficients of X. These coefficients are then stored in an array A(I). Lines 60 to 100 set the machine into HIRES mode and draw the central axes. Lines 110 to 160 repeatedly calculate Y for the next value of X and plot the required point on the screen. The subroutine at lines 180 to 270 calculates the value of Y for a given value of X and makes sure that the point to be plotted lies within the allowable range. This does mean that any points outside the range appear on the border of the screen, but this is not likely to cause much ambiguity.

Notice that the values used for X have been scaled up to effectively expand this axis by a factor of 12. This was found to be necessary to make a reasonably sized diagram on the screen. To test the program, try a simple square law. Answer 2RETURN to the ORDER OF POLYNOMIAL question and then 0,0 and 1 in answer to the coefficient questions respectively.

8.4 Music.

We shall now try some music programs to demonstrate the power of the ORIC's sound generator as a musical box in three voices. None of the following programs make any use of the envelope commands. Even so, they give good value for money.

The old favourite "Twinkle twinkle little star" is first, starting off as a statement of the tune, and then harmonising using two and then three voices.

```
 10 REPEAT
 20 PLAY H,0,0,0
 30 READ A,B,C,D,E,F,G,H
 40 MUSIC 1,A,B,7
 50 MUSIC 2,C,D,7
 60 MUSIC 3,E,F,9
 70 PLAY G,0,0,0
 80 WAIT 15
 90 UNTIL G=0
100 DATA 4,1,1,1,1,1,1,1
110 DATA 4,1,1,1,1,1,1,0
120 DATA 4,1,1,1,1,1,1,1
130 DATA 4,1,1,1,1,1,1,0
140 DATA 4,8,1,1,1,1,1,1
150 DATA 4,8,1,1,1,1,1,0
160 DATA 4,8,1,1,1,1,1,1
170 DATA 4,8,1,1,1,1,1,0
180 DATA 4,10,1,1,1,1,1,1
190 DATA 4,10,1,1,1,1,1,0
200 DATA 4,10,1,1,1,1,1,1
210 DATA 4,10,1,1,1,1,1,0
220 DATA 4,8,1,1,1,1,1,1
230 DATA 4,8,1,1,1,1,1,1
240 DATA 4,8,1,1,1,1,1,1
250 DATA 4,8,1,1,1,1,1,0
260 DATA 4,6,1,1,1,1,1,1
270 DATA 4,6,1,1,1,1,1,0
280 DATA 4,6,1,1,1,1,1,1
290 DATA 4,6,1,1,1,1,1,0
300 DATA 4,5,1,1,1,1,1,1
310 DATA 4,5,1,1,1,1,1,0
320 DATA 4,5,1,1,1,1,1,1
330 DATA 4,5,1,1,1,1,1,0
340 DATA 4,3,1,1,1,1,1,1
350 DATA 4,3,1,1,1,1,1,0
360 DATA 4,3,1,1,1,1,1,1
370 DATA 4,3,1,1,1,1,1,0
380 DATA 4,1,1,1,1,1,1,1
390 DATA 4,1,1,1,1,1,1,1
400 DATA 4,1,1,1,1,1,1,1
410 DATA 4,1,1,1,1,1,1,0
420 DATA 4,8,4,1,1,1,3,1
430 DATA 4,8,3,5,1,1,3,0
440 DATA 4,8,3,8,1,1,3,1
450 DATA 4,8,4,1,1,1,3,0
460 DATA 4,6,3,10,1,1,3,1
```

```
470 DATA 4,6,4,3,1,1,3,0
480 DATA 4,6,3,12,1,1,3,1
490 DATA 4,6,3,8,1,1,3,0
500 DATA 4,5,4,1,1,1,3,1
510 DATA 4,5,3,5,1,1,3,0
520 DATA 4,5,3,8,1,1,3,1
530 DATA 4,5,4,1,1,1,3,0
540 DATA 4,5,4,1,1,1,3,1
550 DATA 4,5,3,10,1,1,3,0
560 DATA 4,3,3,12,1,1,3,3
570 DATA 4,3,3,12,1,1,3,0
580 DATA 4,8,4,1,1,1,3,1
590 DATA 4,8,3,5,1,1,3,0
600 DATA 4,8,  3,8,1,1,3,1
610 DATA 4,8,4,1,1,1,3,0
620 DATA 4,6,4,3,1,1,3,1
630 DATA 4,6,4,1,1,1,3,0
640 DATA 4,6,3,12,1,1,3,1
650 DATA 4,6,3,10,1,1,3,0
660 DATA 4,5,3,8,1,1,3,1
670 DATA 4,5,3,5,1,1,3,0
680 DATA 4,5,4,1,1,1,3,1
690 DATA 4,5,3,10,1,1,3,0
700 DATA 4,5,3,8,1,1,3,1
710 DATA 4,5,3,10,1,1,3,0
720 DATA 4,3,3,12,1,1,3,1
730 DATA 4,3,3,8,1,1,3,0
740 DATA 4,1,1,1,3,1,5,5
750 DATA 4,1,3,5,3,1,7,4
760 DATA 4,1,3,8,3,1,7,5
770 DATA 4,1,3,10,3,1,7,0
780 DATA 4,8,3,12,2,8,7,5
790 DATA 4,8,4,3,2,8,7,4
800 DATA 4,8,3,12,2,8,7,5
810 DATA 4,8,3,8,2,8,7,0
820 DATA 4,10,4,1,2,6,7,3
830 DATA 4,10,4,1,2,8,7,0
840 DATA 4,10,4,1,2,10,7,3
850 DATA 4,10,4,1,2,12,7,0
860 DATA 4,8,3,5,3,1,7,1
870 DATA 4,8,3,6,3,3,7,1
880 DATA 4,8,3,8,3,5,7,5
890 DATA 4,8,4,1,3,5,7,0
900 DATA 4,6,4,3,3,6,7,3
910 DATA 4,6,4,3,3,5,7,0
920 DATA 4,6,4,1,3,3,7,5
930 DATA 4,6,3,12,3,3,7,0
940 DATA 4,5,4,1,3,1,7,5
950 DATA 4,5,3,12,3,1,7,0
```

```
 960 DATA 4,5,3,10,3,1,7,5
 970 DATA 4,5,3,8,3,1,7,0
 980 DATA 4,3,3,8,2,6,7,5
 990 DATA 4,3,4,1,2,6,7,0
1000 DATA 4,3,3,12,2,8,7,5
1010 DATA 4,3,3,8,2,8,7,0
1020 DATA 4,1,3,5,3,1,7,7
1030 DATA 4,1,3,5,3,1,7,7
1040 DATA 4,1,3,5,3,1,7,7
1050 DATA 4,1,3,5,3,1,7,0
1060 DATA 1,1,1,1,1,1,0,0
```

'**Drink To Me Only**' follows, again in three part harmony without using envelope commands.

```
  10 REPEAT
  20 PLAY H,0,0,0
  30 READ A,B,C,D,E,F,G,H
  40 MUSIC 1,A,B,8
  50 MUSIC 2,C,D,8
  60 MUSIC 3,E,F,8
  70 PLAY G,0,0,0
  80 WAIT 30
  90 UNTIL G=0
 100 DATA 4,12,4,12,4,8,7,0
 110 DATA 4,12,4,12,4,8,7,0
 120 DATA 4,12,4,10,4,7,7,0
 130 DATA 5,1,4,8,4,5,7,1
 140 DATA 5,1,4,7,4,3,7,0
 150 DATA 5,1,4,5,4,1,7,0
 160 DATA 5,3,4,8,3,12,7,2
 170 DATA 5,1,4,8,4,5,7,0
 180 DATA 4,12,4,7,4,3,7,0
 190 DATA 4,10,4,5,4,1,7,6
 200 DATA 4,12,4,5,4,1,7,0
 210 DATA 5,1,4,10,1,1,3,0
 220 DATA 5,3,4,12,1,1,3,2
230 DATA 4,8,4,12,1,1,3,0
 240 DATA 5,1,4,8,4,5,7,0
 250 DATA 4,12,4,8,4,3,7,5
 260 DATA 4,12,4,5,4,3,7,0
 270 DATA 4,10,4,7,4,3,7,0
 280 DATA 4,8,4,8,4,5,7,1
 290 DATA 4,8,4,7,4,3,7,1
 300 DATA 4,8,4,5,4,1,7,1
 310 DATA 4,8,4,3,3,12,7,1
 320 DATA 4,8,4,5,4,1,7,1
 330 DATA 4,8,4,8,4,5,7,0
 340 DATA 4,12,4,12,4,8,7,4
```

```
350 DATA 4,12,4,8,4,8,7,0
360 DATA 4,12,4,8,4,7,7,0
370 DATA 5,1,4,7,4,3,7,1
380 DATA 5,1,4,8,4,5,7,0
390 DATA 5,1,4,10,4,7,7,0
400 DATA 5,3,4,8,4,5,7,6
410 DATA 5,1,4,8,4,5,7,0
420 DATA 4,12,4,7,4,3,7,0
430 DATA 4,10,4,5,4,1,7,6
440 DATA 4,12,4,5,4,1,7,0
450 DATA 5,1,4,10,1,1,3,0
460 DATA 5,3,4,12,1,1,3,2
470 DATA 4,8,4,12,4,8,7,0
480 DATA 5,1,4,8,4,5,7,0
490 DATA 4,12,4,3,4,3,7,3
500 DATA 4,12,4,3,4,5,7,0
510 DATA 4,10,4,3,4,7,7,0
520 DATA 4,8,4,8,4,8,7,1
530 DATA 4,8,4,5,4,1,7,1
540 DATA 4,8,4,8,4,5,7,0
550 DATA 4,8,4,12,4,8,6,6
560 DATA 1,1,4,12,4,8,6,0
570 DATA 5,3,1,1,1,1,1,0
580 DATA 5,3,1,1,1,1,1,0
590 DATA 4,12,1,1,1,1,1,0
600 DATA 5,3,4,12,4,3,7,0
610 DATA 5,8,4,12,4,3,7,1
620 DATA 5,8,4,8,3,12,7,0
630 DATA 5,3,4,12,4,3,7,0
640 DATA 5,3,4,12,4,8,7,6
650 DATA 4,12,4,8,4,8,7,0
660 DATA 5,3,4,12,4,3,7,0
670 DATA 5,3,4,12,4,3,7,1
680 DATA 5,3,4,8,3,12,7,0
690 DATA 5,3,4,12,4,3,7,0
700 DATA 5,5,5,1,4,1,7,1
710 DATA 5,5,4,12,4,3,7,0
720 DATA 5,3,4,8,4,5,7,0
730 DATA 5,3,4,10,4,7,7,4
740 DATA 5,1,4,3,4,7,7,0
750 DATA 4,12,4,5,4,8,7,0
760 DATA 4,12,4,8,4,3,7,5
770 DATA 4,12,4,5,4,3,7,5
780 DATA 4,12,4,8,4,3,7,4
790 DATA 4,10,4,8,4,3,7,7
800 DATA 4,10,4,8,4,3,7,5
810 DATA 4,10,4,7,4,3,7,0
820 DATA 4,12,4,8,1,1,3,2
830 DATA 4,12,4,8,1,1,3,0
```

```
 840 DATA 4,12,1,1,4,8,5,0
 850 DATA 5,1,4,10,4,8,7,3
 860 DATA 5,1,4,10,4,7,7,0
 870 DATA 5,1,4,8,4,5,7,0
 880 DATA 5,3,4,7,4,3,7,4
 890 DATA 5,1,4,5,4,3,7,0
 900 DATA 4,12,4,8,4,3,7,0
 910 DATA 4,10,4,7,4,3,7,6
 920 DATA 4,12,4,7,4,3,7,0
 930 DATA 5,1,4,8,4,5,7,0
 940 DATA 5,3,4,8,3,12,7,6
 950 DATA 4,8,4,8,3,12,7,0
 960 DATA 5,1,4,8,4,5,7,0
 970 DATA 4,12,4,8,4,3,7,5
 980 DATA 4,12,4,5,4,3,7,0
 990 DATA 4,10,4,7,4,3,7,0
1000 DATA 4,8,4,8,3,12,7,7
1010 DATA 4,8,4,8,3,12,7,7
1020 DATA 4,8,4,8,3,12,7,7
1030 DATA 4,8,4,8,3,12,7,7
1040 DATA 4,8,4,8,3,12,7,7
1050 DATA 4,8,4,8,3,12,7,0
1060 DATA 1,1,1,1,1,1,0,0
```

The last tune in this series is **"Bobby Shafto"**,again in three part harmony but this time with the envelope command being used for some of the arrangement. It is a matter of opinion as to whether or not the use of the envelope command improves the result.

```
  10  L=1 : P=4000
  15 REPEAT
  20 PLAY M,0,L,P
  30 READ A,B,C,D,E,F,G,H,J,K,M
  40 MUSIC 1,A,B,C
  50 MUSIC 2,D,E,F
  60 MUSIC 3,G,H,J
  70 PLAY K,0,L,P
  80 WAIT 15
  90 UNTIL K=8
  95 PLAY 0,0,0,0
 100 DATA 4,11,0,4,3,0,3,11,0,7,0
 110 DATA 4,11,0,4,4,0,4,1,0,7,0
 120 DATA 4,11,0,4,6,0,4,3,0,7,0
 130 DATA 5,4,0,4,8,0,4,4,0,7,0
```

```
140 DATA 5,3,0,4,10,0,4,6,0,7,0
150 DATA 5,6,0,4,12,0,4,8,0,7,0
160 DATA 5,3,0,4,6,0,4,10,0,7,0
170 DATA 4,11,0,4,11,0,4,11,0,7,0
180 DATA 4,6,0,4,10,0,5,1,0,7,0
190 DATA 4,6,0,4,8,0,4,11,0,7,0
200 DATA 4,6,0,4,6,0,4,10,0,7,0
210 DATA 4,11,0,4,8,0,4,8,0,7,0
220 DATA 4,10,0,4,10,0,4,6,0,7,0
230 DATA 5,1,0,4,8,0,3,1,0,7,0
240 DATA 4,10,0,4,6,0,4,6,0,7,0
250 DATA 4,6,0,1,1,0,4,4,0,5,0
260 DATA 4,11,0,4,6,0,4,3,0,7,0
270 DATA 4,11,0,4,8,0,4,4,0,7,0
280 DATA 4,11,0,4,6,0,4,6,0,7,0
290 DATA 5,4,0,1,1,0,1,1,0,1,0
300 DATA 5,3,0,4,10,0,4,6,0,7,0
310 DATA 5,6,0,4,11,0,4,8,0,7,0
320 DATA 5,3,0,4,10,0,4,6,0,7,0
330 DATA 4,11,0,1,1,0,1,1,0,1,0
340 DATA 5,1,0,4,8,0,4,4,0,7,0
350 DATA 5,4,0,4,8,0,3,1,0,7,0
360 DATA 5,1,0,4,10,0,4,6,0,7,0
370 DATA 4,10,0,4,10,0,4,4,0,7,0
380 DATA 4,11,10,4,11,10,4,3,10,7,7
390 DATA 4,11,10,4,11,10,4,3,10,7,0
400 DATA 4,11,10,3,11,10,4,3,10,7,0
410 DATA 1,1,10,1,1,10,1,1,10,0,0
420 DATA 5,3,10,4,6,10,3,11,10,7,6
430 DATA 5,6,10,4,6,10,3,11,10,7,6
440 DATA 5,3,10,4,6,10,3,11,10,7,6
450 DATA 4,11,10,4,6,10,3,11,10,7,6
460 DATA 5,3,10,4,6,10,3,11,10,7,6
470 DATA 5,6,10,4,6,10,3,11,10,7,6
480 DATA 5,3,10,4,6,10,3,11,10,7,6
490 DATA 5,3,10,4,6,10,3,11,10,7,0
500 DATA 4,1,10,4,6,10,3,10,7,6
510 DATA 5,4,10,4,6,10,3,10,10,7,6
520 DATA 4,1,10,4,6,10,3,10,10,7,6
530 DATA 4,10,10,4,6,10,3,10,10,7,6
540 DATA 5,1,10,4,6,10,3,10,10,7,6
550 DATA 5,4,10,4,6,10,3,10,10,7,6
560 DATA 4,1,10,4,6,10,3,10,10,7,6
570 DATA 4,1,10,4,6,10,3,10,10,7,0
580 DATA 5,3,10,4,6,10,3,11,10,7,6
590 DATA 5,6,10,4,6,10,3,11,10,7,6
600 DATA 5,3,10,4,6,10,3,51,10,7,6
610 DATA 4,11,10,4,6,10,3,11,10,7,6
620 DATA 5,3,10,4,6,10,3,11,10,7,6
```

```
630 DATA 5,6,10,4,6,10,3,11,10,7,6
640 DATA 5,3,10,4,6,10,3,11,10,7,6
650 DATA 5,3,10,4,6,10,3,11,10,7,0
660 DATA 5,1,10,4,8,10,3,10,10,7,6
670 DATA 5,4,10,4,8,10,3,10,10,7,2
680 DATA 5,1,10,4,8,10,4,4,10,7,6
690 DATA 4,10,10,4,8,10,4,4,10,7,0
700 DATA 4,11,10,4,6,10,4,3,10,7,7
710 DATA 4,11,10,4,6,10,4,3,10,7,6
720 DATA 4,11,10,4,6,10,4,3,10,7,7
730 DATA 4,11,10,4,6,1,0,4,3,10,7,0
740 DATA 1,1,1,1,1,1,3,11,10,4,0
750 DATA 1,1,1,1,1,1,3,11,10,4,0
760 DATA 5,4,10,4,8,10,3,11,10,7,3
770 DATA 5,4,10,4,8,10,4,4,10,7,0
780 DATA 5,6,10,4,11,10,4,3,10,7,3
790 DATA 5,6,10,4,11,10,4,6,10,7,0
800 DATA 1,1,1,1,1,1,4,3,10,4,0
810 DATA 1,1,1,1,1,1,3,11,10,4,0
820 DATA 4,6,10,4,3,10,3,6,10,7,3
830 DATA 4,6,10,4,3,10,3,6,10,7,0
840 DATA 4,8,10,4,4,10,3,6,10,7,3
850 DATA 4,8,10,4,4,10,3,11,10,7,0
860 DATA 4,10,10,4,6,10,3,10,10,7,3
870 DATA 4,10,10,4,6,10,4,1,10,7,0
880 DATA 1,1,1,1,1,1,3,10,10,4,0
890 DATA 1,1,1,1,1,1,3,6,10,4,0
900 DATA 1,1,1,1,1,1,3,11,10,4,0
910 DATA 1,1,1,1,1,1,3,11,10,4,0
920 DATA 5,8,10,4,11,10,3,11,10,7,3
930 DATA 5,8,10,4,11,10,4,4,10,7,0
940 DATA 5,6,10,4,11,10,4,3,10,7,3
950 DATA 5,6,10,4,11,10,4,6,10,7,0
960 DATA 1,1,1,1,1,1,4,3,10,4,0
970 DATA 1,1,1,1,1,1,3,11,10,4,0
980 DATA 5,4,10,4,8,10,4,1,10,7,3
990 DATA 5,4,10,4,8,10,4,4,10,7,0
1000 DATA 5,6,10,4,10,10,4,3,10,7,0
1010 DATA 5,4,10,4,8,10,3,10,10,7,0
1020 DATA 5,3,10,4,6,10,3,11,10,7,7
1030 DATA 5,3,10,4,6,10,3,11,10,7,0
1040 DATA 1,1,1,1,1,1,3,11,10,4,4
1050 DATA 1,1,1,1,1,1,3,11,10,4,0
1060 DATA 5,3,0,4,11,0,1,1,1,3,0
1070 DATA 5,6,0,4,11,0,1,1,1,3,0
1080 DATA 5,3,0,4,10,0,1,1,1,3,0
1090 DATA 4,11,0,4,10,0,1,1,1,3,0
1100 DATA 5,3,0,4,8,0,4,11,0,7,0
1110 DATA 5,6,0,4,8,0,4,11,0,7,0
```

```
1120 DATA 5,3,10,4,6,10,4,10,10,7,7
1130 DATA 5,3,10,4,6,10,4,10,10,7,0
1140 DATA 5,1,0,4,4,0,4,8,0,7,0
1150 DATA 5,4,0,4,4,0,4,8,0,7,0
1160 DATA 5,1,0,4,3,0,4,6,0,7,0
1170 DATA 4,10,0,4,3,0,4,6,0,7,0
1180 DATA 5,1,0,4,1,0,4,4,0,7,0
1190 DATA 5,4,0,4,1,0,4,4,0,7,0
1200 DATA 5,1,10,4,4,10,4,1,10,7,7
1210 DATA 5,1,10,4,4,10,4,1,10,7,0
1220 DATA 5,3,0,4,6,0,3,11,0,7,0
1230 DATA 5,6,0,4,6,0,3,11,0,7,9
1240 DATA 5,3,0,4,8,0,4,4,0,7,0
1250 DATA 5,11,0,4,8,0,4,4,0,7,0
1260 DATA 5,3,0,4,10,0,4,6,0,7,0
1270 DATA 5,6,0,4,10,0,4,6,0,7,0
1280 DATA 5,3,10,4,11,10,4,8,10,7,7
1290 DATA 5,3,10,4,11,10,4,8,10,7,0
1300 DATA 1,1,1,1,1,1,1,1,1,0,0
1310 DATA 1,1,1,1,1,1,1,1,1,0,0
1320 DATA 1,1,1,1,1,1,1,1,1,0,0
1330 DATA 1,1,1,1,1,1,1,1,1,0,0
1340 DATA 5,1,10,4,11,10,4,8,10,7,7
1350 DATA 5,1,10,4,11,10,4,8,10,7,0
1360 DATA 5,4,10,4,10,10,4,6,10,7,7
1370 DATA 5,4,10,4,10,10,4,6,10,7,0
1380 DATA 5,1,10,4,8,10,4,4,10,7,7
1390 DATA 5,1,10,4,8,10,4,4,10,7,0
1400 DATA 4,10,10,4,6,10,4,3,10,7,1
1410 DATA 4,10,10,4,4,10,4,1,10,7,0
1420 DATA 4,11,10,4,3,10,3,11,10,7,7
1430 DATA 4,11,10,4,3,10,3,11,10,7,7
1440 DATA 4,11,10,4,3,10,3,11,10,7,7
1450 DATA 4,11,10,4,3,10,3,11,10,7,0
1460 DATA 5,11,10,4,11,10,3,11,10,7,7
1470 DATA 5,11,10,4,11,10,3,11,10,7,7
1480 DATA 5,11,10,4,11,10,3,11,10,7,7
1490 DATA 5,11,10,4,11,10,3,11,10,7,0
1500 DATA 1,1,1,1,1,1,1,1,1,8,0
```

Notice that because of the bar's rest, the usual way of terminating the tune will not do and another way has to be found.

8.5 Simple Machine Code.

We shall now look at some simple machine code programs. These have all been 'worked through', from assembly language to hexadecimal code, to demonstrate the method. The first *is a simple screen access type program.*

	Assembly	location	code
	LDX @0	400	A2 00
LABEL	STX # BBAA	402	8E AA BB
	INX	405	E8
	JMP LABEL	406	4C 02 04

To enter this code into the machine, a special program has to be used:-

```
10 I=0
20 REPEAT
30 READ A
40 POKE # 400+I,A
50 I=I+1
60 UNTIL A=#FF
70 DATA #A2, #0, #8E, #AA, #BB
80 DATA #E8, # 4C, # 02, # 04, # FF
```

When run this program puts the machine code one just written into memory at location #400 onwards. To run the machine code program type CALL #400<*RETURN*>. All this program does is to cycle the first character location on the screen through all possible values. This will probably make your picture jump about a bit, but it can be stopped by using the hidden reset key.

This program can be used to fill the screen with any desired character very quickly.

	Assembly	Location	Code
START	LDA @#AA	400	A9 AA
	STA#50	402	85 50
	LDA @#BB	404	A9 BB
	STA#51	406	85 51
INP	LDY @0	408	A0 00
	LDA @#41	40A	A9 41
POKE	STA (#50),Y	40C	91 50
	INY	40F	C8
	CPY @#25	40F	C0 25
	BCS INCR	411	B0 03
	JMP POKE	413	4C 0C 04
INCR	CLC	416	18
	LDA # 50	417	A5 50
	ADC @40	419	69 28
	STA #50	41B	85 50
	LDA # 51	41D	A5 51
	ADC # 0	41F	69 00
	STA # 51	421	85 51
	CMP @ # C0	423	C9 C0
	BCS END	425	B0 03
	JMP INP	427	4C 08 04
END	RTS	42A	60

And again, this program is entered into the machine using POKE to achieve the desired result. As it stands, this program will fill the screen with the letter 'A'. If another character is required, the contents of location #40B must be changed. From the ASCII character set, the '#' sign is represented by the number 35. So type POKE #40B,35<RETURN> followed by CALL #400<RETURN> and you will have a screen full of hash signs.

The Branch instructions in the above example had their arguments calculated by a standard Assembler. In line 7, the BCS instruction has an argument of 03 meaning branch forward 3 bytes if the carry is set. To see how this number is arrived at, remember that the computer loads the instruction B0, meaning BCS, then increments the program counter, then decodes the instruction. The number 3 is then loaded and the P.C again incremented so that it is now pointing to the byte 4C. The byte labelled INCR is the third byte along from this.

To complete this description of branching instructions a short example of branching backwards follows:-

```
START   LDX @0          400         A2 00
LABEL   STX #BB80       8E 80 BB
        INX             E8
        BNE LABEL       D0 FA
        BEQ LABEL       F0 F8
```

Again, when the instruction D0 is met, the program counter is pointing to the byte FA. This is loaded and the program counter incremented to point to F0. Now the distance in bytes between F0 and LABEL is 6. Thus the argument is FA representing -6. Remember that FF=-1, thus FE=-2, FD=-3, FC=-4, FB=-5, FA=-6.

This sort of utility can be combined with the special 'undefined' commands left for the user to play with. Since we now have a useful command built in to the ORIC at #400, add the following:- DOKE#2F5,#400 <RETURN>. Now the '!' key will call the routine. Type !RETURN and the screen is again filled with hash signs.

A more useful utility is a renumber program written in machine code which can be POKE'd into memory and then called using the '!' directive. This allows programs to be developed without having to worry about adding on a lump of code, or finding out the start address of that code after each renumber. A word of warning first, however. When dealing with a program like this, where there is a lot of machine code, type the program in and save it onto cassette before running it. Now if anything goes wrong, the original program can be recovered and checked for errors. This is a good tip for any long program, not just those which contain a lot of machine code.

```
10 I=0
20 REPEAT
30 READ A
```

```
 40 POKE (#400+I),A
 50 I=I+1
 60 UNTIL A=#FF
 70 DOKE #2F5,#400
 80 DATA #A9,#01 #85,#50,#A9,#05,#85
 90 DATA #51,#A9,#00,#85,#53,#85,#55
100 DATA #A9,#0A,#85,#52,#85,#54,#A0
110 DATA #00,#B1,#50,#85,#56,#C8,#B1
120 DATA #50,#85,#57,#18,#A5,#50,#69
130 DATA #02,#85,#50,#A5,#51,#69,#00
140 DATA #85,#51,#A0,#00,#A5,#52,#91
150 DATA #50,#C8,#A5,#53,#91,#50,#18
160 DATA #A5,#54,#65,#52,#85,#52,#A5
170 DATA #55,#65,#53,#85,#53,#A5,#56
180 DATA #85,#50,#F0,#07,#A5,#57,#85
190 DATA #51,#4C,#14,#04,#A5,#57,#D0
200 DATA #F7,#60,#FF
```

In case of error, an assembly listing is included in Appendix D.

After this program has been typed in, save it and then RUN it. Nothing dramatic happens since all it does is set up a machine code program starting address $400 and set up the indirect address of the '!' command to point to it. Renumber a few lines or add some lines as 13 REM TEST and so on and then type !<RETURN>. The correct response is for the computer to reply 'Ready' almost immediately. If it does not, something has gone awry and you must recover by using the hidden switch underneath the machine, or by switching off and starting again. If you have to do the latter, you can be sure you have some really bad mistakes in there somewhere. Check through the listing until you find the errors and try again.

You will have noticed that this program does not give you any choice as to the line numbers to be used. Those of you with some skill in machine code may like to use the published listing and modify it, so that the start number and step can be separate, and hence individually adjustable. At present 'start line number' = 'step' = 10 and both must be changed together. This can be done by POKEing #40F with the required number. For example POKE #40F,1<RETURN> followed by !<RETURN> should result in a program numbered 1, 2, 3 etc.

A further modification would be to automatically change the line number referenced by GOTO and GOSUB commands. Unfortunately the way the ORIC stores its programs makes this very difficult as a stand alone program. The problem is that referenced line numbers are stored as ASCII text not as binary numbers. This means that if a referenced number had to be changed from 5, say, to 10, the number of bytes used would increase, hence the whole program would have to be moved up through memory. This in turn means that all the start of next line addresses would have to be changed also. While this sort of thing is not impossible, the program needed would be far too long for this essay.

As a final point, look at the way the DATA statements have been arranged. This

method allows easy checking of the statements and gives a neat layout to the whole program.

The final offerings in this chapter use parts of the ORIC's operating system to achieve their ends. The first is a real time clock program which puts up a 24 hour readout in the top right hand corner of the screen. This readout is fairly permanent since the program is actually all in machine code and hence cannot be stopped or overwritten. However when other programs are loaded from or saved to cassette, the clock stops because during tape transfers the ORIC disables interrupts. The way the clock program works is to divert the normal interrupt through a machine code routine stored at location #B000 which does all the work, then returns control to the normal interrupt routine. This is possible because both interrupts are 'soft-vectored' through RAM locations as described in chapter 7. It is therefore fairly easy to divert them and use them for our own purposes.

The ORIC has an internal timer which interrupts it every 10mS. This timer is part of the 6522 chip and its operation has been described in chapter 3. The real time clock program sets up a counter to determine the 100th interrupt and increments the clock on every such count.

```
 10 I=0
 20 REPEAT
 30 READ A
 40 POKE (#400+I),A
 50 I=I+1
 60 UNTIL A=#FF
 70 DATA #48,#A9,#C8,#85,#08,#A9
 80 DATA #B8,#85,#09,#A9,#63,#85
 90 DATA #0A,#78,#A9,#00,#8D,#29
100 DATA #02,#A9,#B0,#8D,#2A,#02
110 DATA #58,#68,#60,#FF
120 I=0
130 REPEAT
140 READ A
150 POKE (#B000+I),A
160 I=I+1
170 UNTIL A=#FF
180 DATA #48,#98,#48,#8A,#48,#C6,#0A
190 DATA #10,#55,#A9,#63,#85,#0A,#A5
200 DATA #04,#18,#69,#01,#C9,#3C,#B0
210 DATA #05,#85,#04,#4C,#43,#B0,#A9
220 DATA #00,#85,#04,#A5,#05,#18,#69
230 DATA #01,#C9,#3C,#B0,#05,#85,#05
240 DATA #4C,#43,#B0,#A9,#00,#85,#05
250 DATA #A5,#06,#18,#69,#01,#C9,#18
260 DATA #B0,#05,#85,#06,#4C,#43,#B0
270 DATA #A9,#00,#85,#06,#A5,#06,#A0
280 DATA #00,#20,#66,#B0,#A9,#2F,#20
290 DATA #81,#B0,#A5,#05,#20,#66,#B0
```

```
300 DATA #A9,#2F,#20,#81,#B0,#A5,#04
310 DATA #20,#66,#B0,#68,#AA,#68,#A8
320 DATA #68,#4C,#03,#EC,#A2, #00,#E8
330 DATA #38,#E9,#0A,#10,#FA,#CA,#69
340 DATA #0A,#09,#30,#85,#07,#8A,#09
350 DATA #30,#20,#81,#B0,#A5,#07,#20
360 DATA #81,#B0,#60,#91,#08,#C8,#60
370 DATA #FF
380 CLS
390 INPUT "HOURS=";H
400 IF H0 THEN 390
410 IF H23 THEN 390
420 POKE 6,H
430 INPUT"MINUTES=";M
440 IF M0 THEN 430
450 IF M59 THEN 430
460 POKE 5,M
470 POKE 4,0
480 PRINT"PRESS ANY KEY TO START"
490 GET A#
500 CALL #400
510 END
```

For V1.1 change line 120 to
120 DATA #FF,#F0,#07,#AA,#20,#7C,#F7,#C8
to accommodate the differences between the two systems.

When run, this program sets up two machine code segments in memory, one starting at #400 and the other at #B000. The first segment changes the interrupt vector at locations #229 and #22A to point to the code at #B000. The program puts these code segments in place before setting the clock time in lines 380 to 470, then the call to #400 actually starts the clock.

For V1.1 change line 90 to:
90 DATA #0A,#78,#A9,#00,#8D,#45
and line 100 to:
100 DATA #02,#A9,#B0,#8D,#46,#02
and line 320 to
320 DATA #68,#4C,#22,#EE,#A2,#00,#E8

These changes are necessary because the soft vectors in V1.1 are different.

To prevent errors, an assembly language listing is included in Appendix E.

The final program uses the system call at #CC12 to print a character on the screen to demonstrate a method of communicating with a user from inside an assembly language program. The message printed is, in this instance, irrelevant.

```
  5 GRAB
 10 I=0
 20 REPEAT
 30 READ A
 40 POKE (#B100),A
 50 I=I+1
 60 UNTIL A=#F3
 70 DATA #20,#13, #B1,#0C, #48, #45, #4C
 80 DATA #4C,#4F, #20,#57, #4F, #52, #4C
 90 DATA #44,#0A, #0D,#FF, #60, #18, #68
100 DATA #69,#01, #85,#09, #68, #69, #00
110 DATA #85,#0A, #A0,#00, #B1, #09, #C9
120 DATA #FF, #F0, #07,#20, #12, #CC, #C8
130 DATA #4C,#20, #B1, #98, #18, #65, #09
140 DATA #85,#09, #A5,#0A, #69, #00, #48
150 DATA #A5,#09, #48,#60, #F3
160 PRINT "PRESS ANY KEY WHEN READY"
170 GET A#
180 CALL #B100
190 END
```

The GRAB command is necessary because without it, the ORIC will overwrite the machine code program in this area of memory. After this program has been run, any call to #B100 will result in the screen being cleared and the message "HELLO WORLD" being printed out on the top line of the screen.

Again the whole listing is included in Appendix F.

The subroutine WRITER demonstrates the use of the stack to force a new return address for a subroutine as mentioned in chapter 6.

APPENDIX A

Control Codes

Control Character	CHR$ Code	Effect
D	4	Auto double height
F	6	Key click toggle
G	7	Bell
H	8	Horizontal tab
I	9	Backspace
J	10	Line feed
K	11	Vertical tab
L	12	Clear screen
M	13	Return
N	14	Clear row
P	16	Printer toggle
Q	17	Cursor toggle
S	19	V.D.U. toggle
T	20	Caps Lock
]	28	Protected column toggle

Serial Attributes- Colour

Colour	Foreground Esc code	value	Background Esc code	value	Paper/ink
BLACK	@	0	P	16	0
RED	A	1	Q	17	1
GREEN	B	2	R	18	2
YELLOW	C	3	S	19	3
BLUE	D	4	T	20	4
MAGENTA	E	5	U	21	5
CYAN	F	6	V	22	6
WHITE	G	7	W	23	7

To print red letters use PRINT" "CHR#(27)"ATEXT" and the word 'TEXT' will appear in red. The space has to be printed in V1.0 to prevent the escape code being printed in column-1.

To produce a blue background in V1.0 use PRINT CHR#(28)CHR#(27) "T". In V1.1 use POKE to put background attributes in place at the beginning of the line.

Combining under V1.0 to give red letters on a blue background results in:-

```
PRINT CHR#(28)CHR#(27)"T"CHR#(27)"ATEXT".
```

Use string concatenation to make this usable as:-

```
10 A#=CHR#(28)+CHR#(27)+"T"
20 A#=A#+CHR#(27)+"A"
30 PRINT A#"HELLO THERE"
```

Serial atrributes- Non-colour

Function	Esc.code	Value
Normal Std	H	8
Normal Alt	I	9
Double Ht Std	J	10
Double Ht Alt	K	11
Flashing Std	L	12
Flashing Alt	M	13
D/H Flash Std	N	14
D/H Flash Alt	O	15
Text 60 Hz	X	Strange
Text 60 Hz	Y	results
Text 50 Hz	Z	unless
Text 50 Hz	{	you know
Gra 60 Hz	¦	exactly what
Gra 60 Hz	}	you are
Gra 50z	~	doing
Gra 50 Hz	<	here.

To print double height flashing characters in any colour use:-

```
PRINT CHR#(4)" "CHR#(27)"N"CHR#(27)"ATEXT"
```

or using concatenation:-

```
10 A#=CHR#(28)+CHR#(27)+"T"
20 A#=A#+CHR#(4)+CHR#(27)+"N"
30 A#=A#+CHR#(27)+"A"
40 PRINT A#"HELLO THERE"
50 PRINT CHR#(4)
```

Line 50 is needed to stop the double height printing from carrying on. This program will produce a flashing red double height "HELLO THERE" on a half blue half white background. To put the flashing characters on a completely blue background modify the program to:-

```
10 A#=CHR#(28)+CHR#(27)+"T"
20 A#=A#+CHR#(4)+CHR#(27)+"N"
30 A#=A#+CHR#(27)+"A"
40 PRINT A#"HELLO THERE"
50 PRINT CHR#(4)
60 PRINT A#
```

For V1.1 the CHR#(28) can be omitted because it has no effect in this system. To put a new background attribute into the protected column POKE has to be used.

APPENDIX B

Token Table

ABS	D8	ASC	EC	CALL	BF	CHAR	B0
CHR#	ED	CIRCLE	AD	CLEAR	BD	CLS	94
CLOAD	B6	CONT	BB	COS	E2	CURMOV	AB
CURSET	AA	CSAVE	B7	DATA	91	DEEK	E7
DEF	B8	FN	C4	DIM	93	DOKE	8A
DRAW	AC	END	80	EXP	E1	EXPLODE	A4
FALSE	F0	FILL	AF	FOR	8D	TO	C3
STEP	CB	POS	DB	NEXT	90	FRE	DA
GET	BE	GOSUB	9B	GOTO	97	GRAB	9F
HEX#	DC	HIMEM	9E	HIRES	A2	IF	99
THEN	C9	ELSE	C8	INK	B2	INPUT	92
INT	D7	KEY#	F1	LEFT#	F4	LEN	E9
LET	96	LIST	BC	LLIST	8E	LN	E0
LOG	E8	LORES	89	MID#	F6	MUSIC	A8
NEW	C1	ON	B4	PAPER	B1	PATTERN	AE
PEEK	E6	PI	EE	PING	A6	PLAY	A9
PLOT	87	POINT	F3	POKE	B9	POP	86
POS	DB	PRINT	BA	PULL	88	READ	95
RELEASE	A0	REM	9D	REPEAT	8B	RESTORE	9A
RETURN	9C	RIGHT#	F5	RND	DF	RUN	98
SCRN	F2	SGN	D6	SHOOT	A3	SIN	E3
SOUND	A7	SPC	C5	SQR	DE	STOP	B3
STR#	EA	TAB	C2	TAN	E4	TEXT	A1
TROFF	85	TRON	84	TRUE	EF	USR	D9
VAL	EB	WAIT	B5	ZAP	A5		

APPENDIX C

6502 OP-CODES.

OP-CODE	INSTRUCTION	OP-CODE	INSTRUCTION
00	BRK	20	JSR
01	ORA (IND,X)	21	ANDF (IND,X)
02	Future	22	Future
03	Future	23	Future
04	Future	24	BIT Zero Page
05	ORA Zero Page	25	AND Zero Page
06	ASL Zero Page	26	ROL Zero Page
07	Future	27	Future
08	PHP	28	PLP
09	ORA Imm	29	AND Imm
0A	ASL A	2A	ROL A
0B	Future	2B	Future
0C	Future	2C	BIT Abs
0D	ORA Abs	2D	AND Abs
0E	ASL Abs	2E	ROL Abs
0F	Future	2F	Future
10	BPL	30	BMI
11	ORA (IND) ,Y	31	AND (IND) ,Y
12	Future	32	Future
13	Future	33	Future
14	Future	34	Future
15	ORA Zero Page,X	35	AND Zero Page, X
16	ASL Zero Page,X	36	ROL Zero Page,X
17	Future	37	Future
18	CLC	38	SEC
19	ORA Abs,Y	39	AND Abs,Y
1A	Future	3A	Future
1B	Future	3B	Future
1C	Future	3C	BIT Abs
1D	ORA Abs,X	3D	AND Abs,X
1E	ASL Abs,X	3E	ROL Abs,X
1F	Future	3F	Future

40	RTI	60	RTS
41	EOR (IND<X)	61	ADC (IND,X)
42	Future	62	Future
43	Future	63	Future
44	Future	64	Future
45	EOR Zero Page	65	ADC Zero Page
46	LSR Zero Page	66	ROR Zero page
47	Future	67	Future
48	PHA	68	PLA
49	EOR Imm	69	ADC Imm
4A	LSR A	6A	ROR A
4B	Future	6B	Future
4C	JMP	6C	JMP Ind
4D	EOR Abs	6D	ADC Abs
4E	LSR	6E	ROR Abs
4F	Future	6F	Future
50	BVC	70	BVS
51	EOR (IND) ,Y	71	ADC (IND) ,Y
52	Future	72	Future
53	Future	73	Future
54	Future	74	Future
55	EOR Zero Page,X	75	ADC Zero Page,X
56	LSR Zero Page,X	76	ROR Zero Page,X
57	Future	77	Future
58	CLI	78	SEI
59	EOR Abs,Y	79	ADC Abs,Y
5A	Future	7A	Future
5B	Future	7B	Future
5C	Future	7C	Future
5D	EOR Abs,X	7D	ADC Abs,X
5E	LSR Abs,X	7E	ROR Abs,X
5F	Future	7F	Future

80	Future	A0	LDY Imm
81	STA (IND,X)	A1	LDA (IND,X)
82	Future	A2	LDX Imm
83	Future	A3	Future
84	STY Zero Page	A4	LDY Zero Page
85	STA Zero Page	A5	LDA Zero Page
86	STX Zero Page	A6	LDX Zero Page
87	Future	A7	Future
88	DEY	A8	TAY
89	Future	A9	LDA Imm
8A	TXA	AA	TAX
8B	Future	AB	Future
8C	STY Abs	AC	LDY Abs
8D	STA ABS	AD	LDA Abs
8E	STX Abs	AE	LDX Abs
8F	Future	AF	Future
90	BCC	B0	BCS
91	STA (IND),Y	B1	LDA (IND) ,Y
92	Future	B2	Future
93	Future	B3	Future
94	SYT Zero Page, X	B4	LDY Zero Page,X
95	STA Zero Page,X	B5	LDA Zero Page,X
96	STX Zero Page,Y	B6	LDX Zero Page,Y
97	Future	B7	Future
98	TYA	B8	CLV
99	STA Abs,Y	B9	LDA Abs,Y
9A	TXS	BA	TSX
9B	Future	BB	Future
9C	Future	BC	LDY Abs,X
9D	STA Abs,X	BD	LDA Abs,X
9E	Future	BE	LDX Abs,Y
9F	Future	BF	Future

C0	CPY Imm	E0	CPX Imm
C1	CMP (IND,X)	E1	SBC (IND,X)
C2	Future	E2	Future
C3	Future	E3	Future
C4	CPY Zero Page	E4	CPX Zero Page
C5	CMP Zero Page	E5	SBC Zero Page
C6	DEC Zero Page	E6	INC Zero Page
C7	Future	E7	Future
C8	INY	E8	INX
C9	CMP Imm	E9	SBC Imm
CA	DEX	EA	NOP
CB	Future	EB	Future
CC	CPY Abs	EC	CPX Abs
CD	CMP Abs	ED	SBC Abs
CE	DEC Abs	EE	INC Abs
CF	Future	EF	Future
D0	BNE	F0	BEQ
D1	CMP (IND) ,Y	F1	SBC (IND) ,Y
D2	Future	F2	Future
D3	Future	F3	Future
D4	Future	D4	Future
D5	CMP Zero Page,X	F5	sbc Zero Page,X
D6	DEC Zero Page,X	F6	INC Zero Page,X
D7	Future	F7	Future
D8	CLD	F8	SED
D9	CMP Abs,Y	F9	SBC ABs,Y
DA	Future	FA	Future
DB	Future	FB	Future
DC	Future	FC	Future
DD	CMP Abs,X	FD	SBC Abs,X
DE	DEC Abs,X	FE	INC Abs,X
DF	Future	FF	Future

APPENDIX D

Renumber Source Listing

```
0050            AL          EQU #50
0051            AH          EQU #51
0052            KNL         EQU #52
0053            KNH         EQU #53
0054            SNL         EQU #54
0055            SNH         EQU #55
0056            TML         EQU #56
0057            TMH         EQU #57
0000
                                        *PROGRAM STARTS*
                            ORG #400
0400:A9 01      START       LDA @$1
0402:85 50                  STA AL
0404:A9 05                  LDA @$05
0406:85 51                  STA AH   ;set up start address
0408:A9 00                  LDA @0
040A:85 53                  STA KNH
040C:85 55                  STA SNH
040E:A9 0A                  LDA @10
0410:85 52                  STA KNL
0412:85 54                  STA SNL  ;set up base=10;step=10
0414:A0 00      BACK        LDY @0
0416:B1 50                  LDA (AL),Y
0418:85 56                  STA TML
041A:C8                     INY
041B:B1 50                  LDA (AL) ,Y
041D:85 57                  STA TMH ;get address of next line
041F:18                     CLC
0420:A5 50                  LDA AL
0422:69 02                  ADC @2
0424:85 50                  STA AL
0426:A5 51                  LDA AH
0428:69 00                  ADC @0
042A:85 51                  STA AH   ;add 2 to add. of current line
042C:A0 00                  LDY @0
042E:A5 52                  LDA KNL
0430:91 50                  STA (AL) ,Y
0432:C8                     INY
```

```
0433:A5 53                  LDA KNH
0435:91 50                  STA (AL) ,Y     ; set 1st line no. to 10
0437:18                     CLC
0438:A5 54                  LDA SNL
043A:65 52                  ADC KNL
043C:85 52                  STA KNL
043E:A5 55                  LDA SNH
0440:65 53                  ADC KNH
0442:85 53                  STA KNH ;add step to base
0444:A5 56                  LDA TML
0446:85 50                  STA AL
0448:F0 07                  BEQ MAYBE
044A:A5 57                  LDA TMH
044C:85 51      CONT        STA AH ;put add. of next line into current add.
044E:4C 14 04               JMP BACK
0451:A5 57      MAYBE       LDA TMH
0453:D0 F7                  BNE CONT
0455:60                     RTS
```

APPENDIX E

Real Time Clock for ORIC

See section 8.5 for details

```
                ORG #400
0400:48         PHA         ;save acc
0401:A9 C8      LDA @#C8    ;screen add lo
0403:85 08      STA 8       ;Set up indirection lo
0405:A9 BB      LDA @#BB    ;Screen ad hi
0407:85 09      STA 9       ;Set up indirection hi
0409:A9 63      LDA @99     ;Set up counter
040B:85 0A      STA #A      ;in #A
040D:78         SEI         ;Disable interrupts
040E:A9 00      LDA @0      ;Redirect interrupt vector
0410:8D 29 02   STA #229    ;Ditto(STA@45 for V1.1)
0413:A9 B0      LDA @#B0    ;ditto
0415:8D 2A 02   STA #22A    ;ditto(STA @46 for V1.1)
0418:58         CLI         ;Re-enable interrupts
0419:68         PLA         ;Restore acc.
041A:60         RTS         ;Return whence you came
                ORG#B000    ;Origin for interrupt routine
B000:48         PHA         ;Save registers
B001:98         TYA         ;ditto
B002:48         PHA         ;ditto
B003:8A         TXA         ;ditto
B004:48         PHA         ;ditto
B005:C6 0A      DEC #A      ;decrement & test counter
B007:10 55      BPL NOTYT   ;If not negative do nothing
B000:A9 63      LDA @99     ;Reset counter
B00B:85 0A      STA #A      ;ditto
B00D:A5 04      LDA 4       ;Get seconds
B00F:18         CLC
B010:69 01      ADC @1      ;Add one
B012:C9 3C      CMP @60     ;Has one minute gone by?
B014:B0 05      BCS MINS    ;Yes
B016:85 04      STA 4       ;No
B018:4C 43 B0   JMPSHOW     ;Tell the world
B01B:A9 00 MINS LDA @0      ;Reset seconds counter
B01D:85 04      STA 4       ;ditto
B01F:A5 05      LDA 5       ;Get minutes
B021:18         CLC         ;
B022:69 01      ADC @1      ;Add one
B024:C9 3C      CMP @60     ;Has one hour gone by?
B026:B0 05      BCS NHRS    ;Yes
```

```
B028:85 05          STA 5       ;No
B02A:4C 43 B0       JMPSHOW     ;Tell the world again
B02D:A9 00 NHRS     LDA @0      ;Reset minutes
B02F:85 05          STA 5       ;ditto
B031:A5 06          LDA 6       ;Get hours
B033:18             CLC
B034:69 01          ADC @1      ;Add one
B036:C9 18          CMP@24      ;Has one day gone by?
B038:B0 05          BCS MN      ;Yes
B03A:85 06          STA 6       ;no
B03C:4C 43 B0       JMPSHOW     ;Tell the world again
B03F:A9 00 MN       LDA @0      ;Reset hours
B041:85 06          STA 6       ;ditto
B043:A5 06 SHOW     LDA 6       ;Get hours
B045:A0 00          LDY @0      ;Reset display counter
B047:20 66 B0       JSR CONV    ;Display hours
B04A:A9 2F          LDA @#2F    ;Put in /
B04C:20 81 B0       JSR OUTPT   ;See above
B04F:A5 05          LDA 5       ;Display minutes
B051:20 66 B0       JSR CONV    ;ditto
B065:A9 2F          LDA @#2F    ;Another /
B056:20 81 B0       JST OUTPT   ;ditto
B059:A5 04          LDA 4       ;Get seconds
B05B:20 66 B0       JST CONV    ;Display seconds
B05E:68             PLA         ;Restore registers
B05F:AA             TAX         ;ditto
B060:68             PLA         ;ditto
B061:A8             TAY         ;ditto
B062:68             PLA         ;ditto
B063:4C 03 EC       JMP#EC03    ;Go to ORIC interrupt routine
B066:A2 00 CONV     LDX @0      ;Set counter to zero
B068:E8 GOON        INX         ;Add one to it
B069:38             SEC         ;Set carry ready for subtraction
B06A:E9 0A          SBC @10     ;Subtract 10
B06C:10 FA          BPL GOON    ;If still positive do it again
B06E:CA             DEX         ;Negative so decrement counter
B06F:69 0A          ADC @10     ;Restore number
B071:09 30          ORA @#30    ;Add ASCII header
B073:85 07          STA 7       ;Save it
B075:8A             TXA         ;X holds, no. of tens
B076:09 30          ORA @#30    ;Add ASCII header
B078:20 81 B0       JSR OUTPT   ;Print it
B07B:A5 07          LDA 7       ;Get units
B07D:20 81 B0       JSR OUTPT   ;Print them
B080:60             RTS         ;End of story
B081:91 08OUTPT     STA (8) ,Y  ;Put code in place on screen
B083:C8             INY         ;Increment position counter
B084:60             RTS         ;Return whence you came
0000                END
```

APPENDIX F

Writer Subroutine for 6502 systems

```
CC12                    WRTCH
CC12            WRTCH   EQU #CC12 (F77C for V1.1)
0009            RAMP0   EQU #9
000A            RAMP1   EQU #A
B100            ORG#B100
B100:20 13 B1   START   JSR WRITER
B103:0C                 DB #0C
B104:48 45 4C 4C        DB "HELL"
B018:4F 20 57 4F        DB "O WO"
B10C:52 4C 44           DB "RLD"
B10F:0A 0D FF           DB #0A,#0D,#FF
B112:60                 RTS
B113:18                 CLC             ;REMOVE UNWANTED BIT
B114:68                 PLA             ;GET RETURN ADDRESS
B115:69 01              ADC @1          ;ADD I BECAUSE RETURN
B117:85 09              STA RAMP0       ;ADDRESS IS ONE TO FEW
B119:68                 PLA             ;GET REST OF ADDRESS
B11A:69 00              ADC @0          ;ADD CARRY BIT IF ANY
B11C:85 0A              STA RAMP1       ;SAVE IT
B11E:A0 00              LDY @0          ;SET COUNTER TO ZERO
B120:B1 09      WROUT   LDA (RAMP0),Y   ;FETCH CHARACTER
B122:C9 FF              CMP @#FF        ;IS THIS THE LAST?
B124:F0 07              BEQ WREND       ;YUP
B126:20 12 CC           JSR WRTCH       ;NOPE;PRINT IT (In V1.1 put TAX before this instruction)
B129:C8                 INY             ;INCREMENT COUNTER
B12A:4C 20 B1           JMP WROUT       ;GO BACK ROUND AGAIN
B12D:98                 TYA             ;FETCH NUMBER OF
                                         CHARACTERS
B12E:18                 CLC             ;CLEAR UNWANTED BIT
B12F:65 09              ADC RAMP0       ;ADD START ADDRESS
B131:85 09              STA RAMP0
B133:A5 0A              ADC @0          ;ADD CARRY BIT IF ANY
B137:48                 PHA             ;PUSH ON TO STACK
B138:A5 09              LDA RAMP0       ;FETCH REST OF ADDRESS
B13A:48                 PHA             ;PUSH THAT TOO
B13B:60                 RTS             ;GO TO PUSHED ADDRESS
0000:                   END             ;IS NIGH
```

APPENDIX G

Cassette Loader Program

(only for V1.1)

```
                                ORG #281
0281:08                         PHP
0282:78                         SEI
0283:AD F9 FF          THEN     LDA #FFF9
0286:C9 01                      CMP @1
0288:D0 28                      BNE NOW
028A:AD B6 E4                   LDA #E4B6
028D:C9 A2                      CMP @#A2
028F:D0 15                      BNE GOSTO
0291:A0 09                      LDY@9
0293:B9 B5 02          HERE     LDA #02B5,Y
0296:99 21 02                   STA #0221,Y
0299:88                         DEY
029A:10 F7                      BPL HERE
029C:A9 21             AGAIN    LDA @#21
029E:8D 45 02                   STA #245
02A1:A9 02                      LDA @2
02A3:8D 46 02                   STA #246
02A6:4C 67 E8          GOSTO    JMP #E867
                                ORG #02B1
02B1:00                         DFB 0
02B2:4C B6 E7          NOW      JMP #E7B6
02B5:B5 48                      PHA
02B6:A9 00                      LDA @0
02B8:8D B1 02                   STA #2B1
02BB:68                         PLA
02BC:4C 22 EE                   JMP #EE22
```

INDEX

A message from the publisher

Sigma Technical Press is a rapidly expanding British publisher. We work closely in conjunction with John Wiley & Sons Ltd. who provide excellent marketing and distribution facilities.

Would you like to join the winning team that published these highly successful books? Specifically, **could you successfully write a book that would be of interest to the new, mass computer market?**

Our most successful books are linked to particular computers, and we intend to pursue this policy. We see an immense market for books relating to such machines as:

DRAGON
THE BBC COMPUTER
APPLE
TANDY
SINCLAIR
OSBORNE
ATARI
IBM PC
SIRIUS
NEWBRAIN
COMMODORE

and many others

If you think you can write a book around one of these or any other popular computer — or on more general themes — we would like to hear from you.

Please write to:

Graham Beech
Sigma Technical Press
5 Alton Road,
Wilmslow,
Cheshire, SK9 5DY,
United Kingdom.

or, telephone 0625-531035

The Ultimate "How-To" Book for the ORIC-1 and Atmos Computers

The ORIC-1 and ORIC-ATMOS are exciting and powerful British computers with a dazzling specification.

With all of their available "add-ons" they are amazingly versatile machines and yet, there are very few books about them, still fewer that explain *exactly* how the ORIC computers work - and how to squeeze more power out of them. This book makes no assumptions – it takes you through ORIC BASIC and explains how to extend the use of BASIC by using *system calls, machine code* and the like. For the hardware enthusiast, Henry Hicks literally takes the ORIC apart and puts it back together again - explaining exactly what each chip does. There is a full description of the ORIC's CPU – the 6502 chip - and how it relates to *assembler language programming*. The needs of the more adventurous programmer are also covered - you'll find many stimulating hints and tips - including many advanced programming techniques.
Importantly, a part of the book is concerned with *communication* to other devices. Henry Hicks is an expert in this field and explains, in simple terms, the uses of add-on hardware and how to control it.

With or without an ORIC, you can't get by without our range of computer books. Write for a catalogue to:

£6.95

Sigma Technical Press
5 Alton Road
Wilmslow
Cheshire SK9 5DY

ISBN 0 905104 56 0