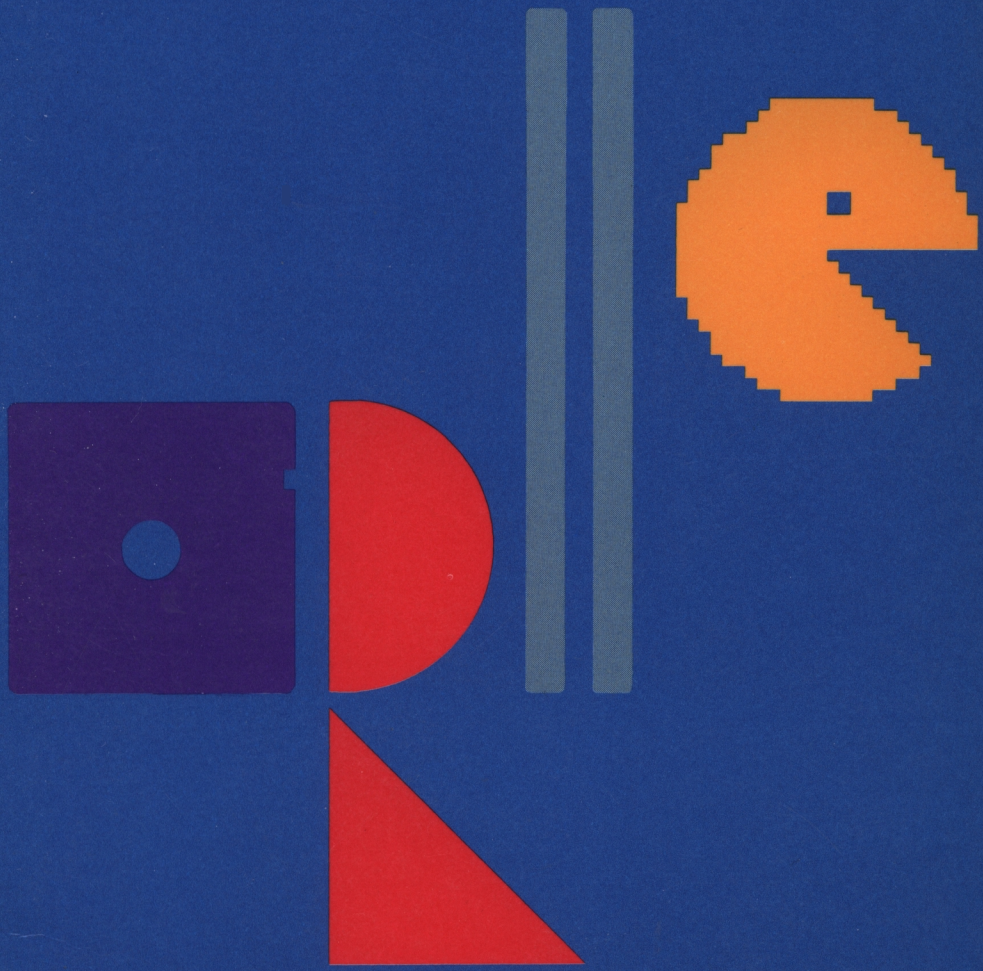
PRENTICE HALL INTERNATIONAL PERSONAL COMPUTER BOOK

UNDERSTANDING ORIC

ORIC 1 (V1.0) and ORIC ATMOS (V1.1)

IAN McLEAN



UNDERSTANDING ORIC

DEDICATION

To Anne, for tea and sympathy

UNDERSTANDING ORIC

Ian McLean

Prentice/Hall  International

Englewood Cliffs, New Jersey London New Delhi Rio de Janeiro
Singapore Sydney Tokyo Toronto Wellington

British Library Cataloguing in Publication Data

McLean, Ian
Understanding ORIC
1. ORIC-1 (Computer)
I. Title
001.64'04 QA76.8.07

ISBN 0-13-477332-2

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY, LTD., *Sydney*
PRENTICE-HALL CANADA, INC., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE, LTD., *Singapore*
PRENTICE-HALL INC., *Englewood Cliffs, New Jersey*
PRENTICE-HALL DO BRASIL LTDA., *Rio de Janeiro*
WHITEHALL BOOKS LIMITED, *Wellington, N.Z.*

Printed in the United Kingdom
by A. Wheaton & Co. Ltd, Exeter

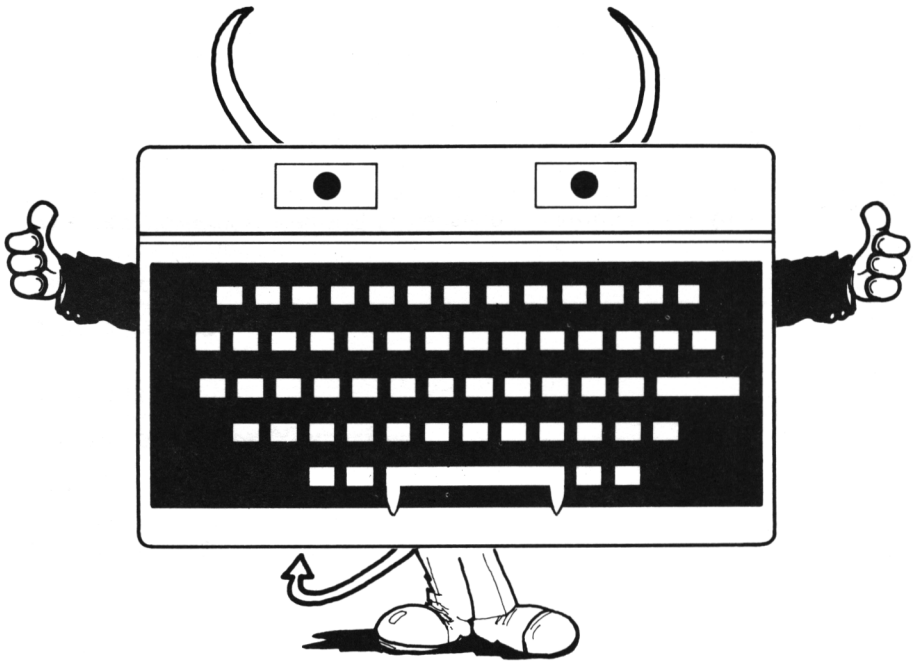
10 9 8 7 6 5 4 3 2 1

Contents

ACKNOWLEDGEMENT

The cartoons following the Preface and on Page 2
were adapted from drawings in 'Graphics Ad Lib No 3'
by Tony Hinwood, published by Business Books

Beware the Evil ORIC



"Dear Sir

We regret the several mistakes which have been made in dealing with your account. These, however, were not the fault of anyone in this organization as they were caused by our computer."

This is the genuine excerpt from a letter sent to a number of people by the accounts department of a well known international company. What is even more suprising is that most of the people to whom the letter was sent accepted the excuse as valid.

The computer has a very bad public image, which is not helped by reports in the popular press. The computer (it would seem) is a ferocious machine – totally evil and much smarter than the poor people on whom it preys. Its main occupation is sending million-pound electricity bills to senior citizens. In its spare time it causes traffic snarl ups and is responsible for every ill which afflicts modern civilization from smog to (the mind boggles) overpopulation.

The only beings allowed into the monster's air conditioned den are a special elite of geniuses who have to have doctorates in at least four different subjects. Any lesser mortal daring to approach the computer – much less touching the keyboard – will be immediately vaporized. Any unauthorized keyboard entry will cause the computer to blow up the world.

Don't believe a word of it.

The computer is just a machine like any other.

It will do only what you tell it to do. It is not very smart and will do exactly what it is told to do – not what you think you told it to do. It will not do what it is not told to do – even though it seems logical to you that that is what it ought to do. You will see what I mean once you start 'programming' the computer (telling it what to do).

YOU ARE IN CONTROL

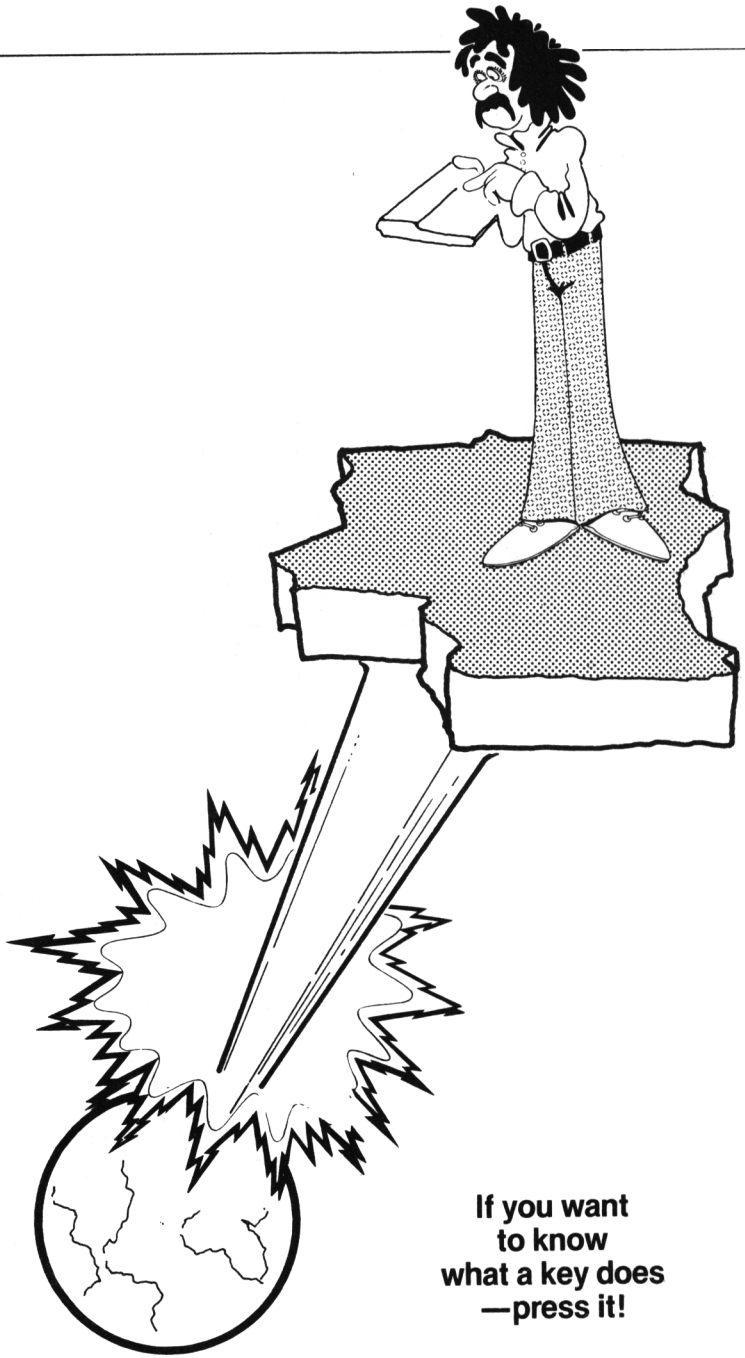
This is the age of the personal computer. Your ORIC 1 is your own property. It is up to you to tell it what to do, and without your instructions it can do nothing.

So if you want to know what a key does – press it.

If you want to know what will happen if you put an instruction into a program – try it.

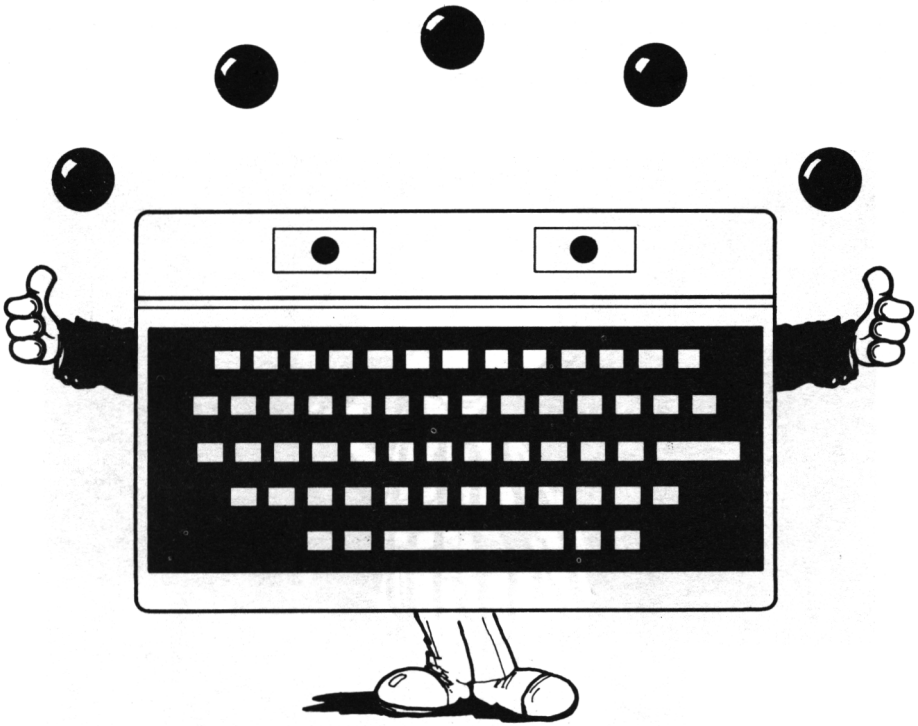
You cannot damage the machine by keyboard entries or program instructions, and sitting down and playing with the machine is fun. Enjoy yourself.

Please treat the programs you find later in the book as examples. They do not represent the only solutions and may not even be the best solutions. Change them in any way you wish. By doing so you will gain a deeper understanding of how the programs work and how the machine operates.


If you want
to know
what a key does
—press it!

CHAPTER 1

The Performing ORIC



"Now he was master of the world he was not quite sure what to do next.

But he would think of something."

Arthur C. Clarke

2001

Congratulations!

You now own an ORIC 1. Possibly the best value-for-money available in home computers at present.

You have sent the children to bed, told all your friends you are going to Outer Mongolia, and put the cat out to play with next door's Alsatian.

Now you are alone with your machine.

What's inna box Daddy?

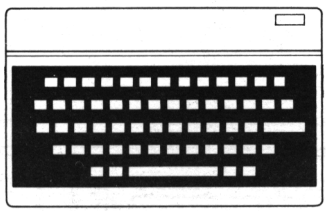
'sit for me?

Can I play wiv it?

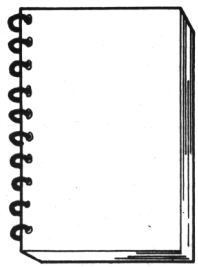
'sit Chissimas - has Sanna brottit?

UNPACKING THE COMPUTER

Carefully open the box and remove the protective packing. You should have the following:

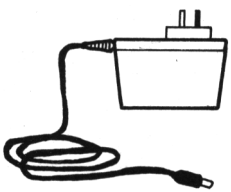


1 Computer



1 User Manual

1 TV Lead



1 Power Supply

1 'Welcome' Tape

1 Tape Recorder Lead

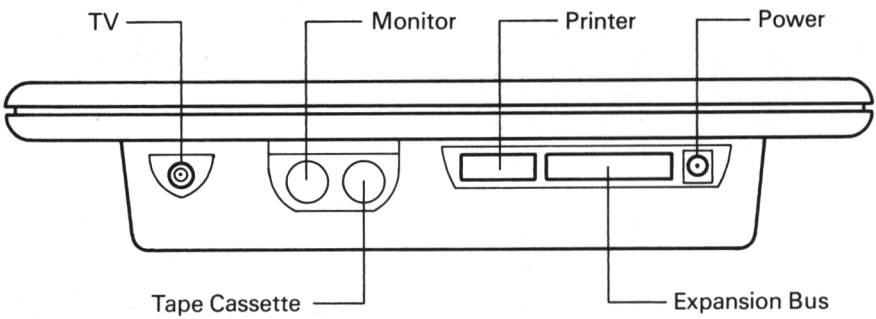
To get your computer working you will need a television set. The computer will work with either a black and white or a colour set, and all the programs in this book will run on whatever set you have. The colour programs will produce different shades of grey on a black and white set.

However, I rather hope you have a colour television set. Some spectacular displays can be generated by the ORIC 1.

For the remainder of the book I will be assuming that a colour set is being used.

CONNECTING UP

If you look at the back of the computer you will see a number of sockets as shown:

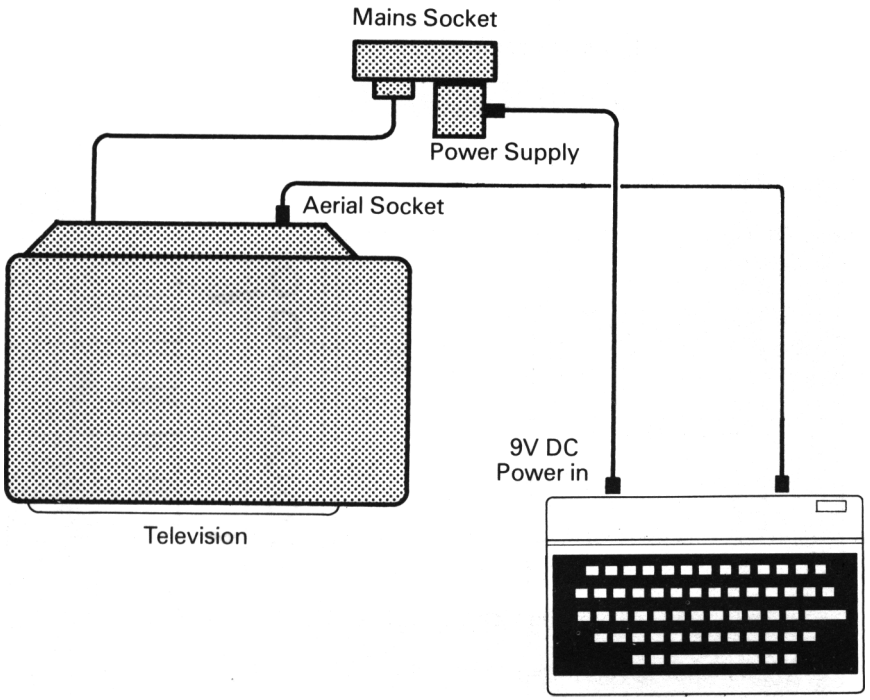


Right now we will be concerned only with the 'Power' and 'TV Output' sockets. We will come to the others later.

The jackplug output of the power supply should **not** be plugged into the power socket in the computer. Plug the power supply into a mains socket. If the mains socket has a switch then switch it on.

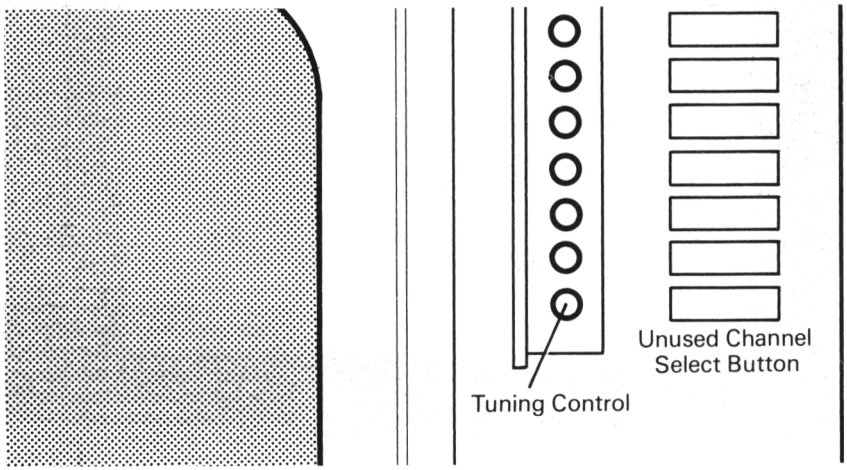
Unplug the aerial from your television and use the special cable provided to connect the 'TV Output' socket on your ORIC 1 to the aerial socket on your television. If your television has both 'UHF' and 'VHF' sockets, use the socket marked 'UHF'.

If your television has an internal aerial you may find a switch next to the (normally unused) external aerial socket. This switches to external aerial and disconnects the internal aerial. If it exists, this switch should be operated.



Switch on the television and plug the power supply into the computer. Turn the television volume control right down.

If your television has channel select buttons (as in the diagram below) press a normally unused button.



Tune the television by turning the tuning control knob until the message:

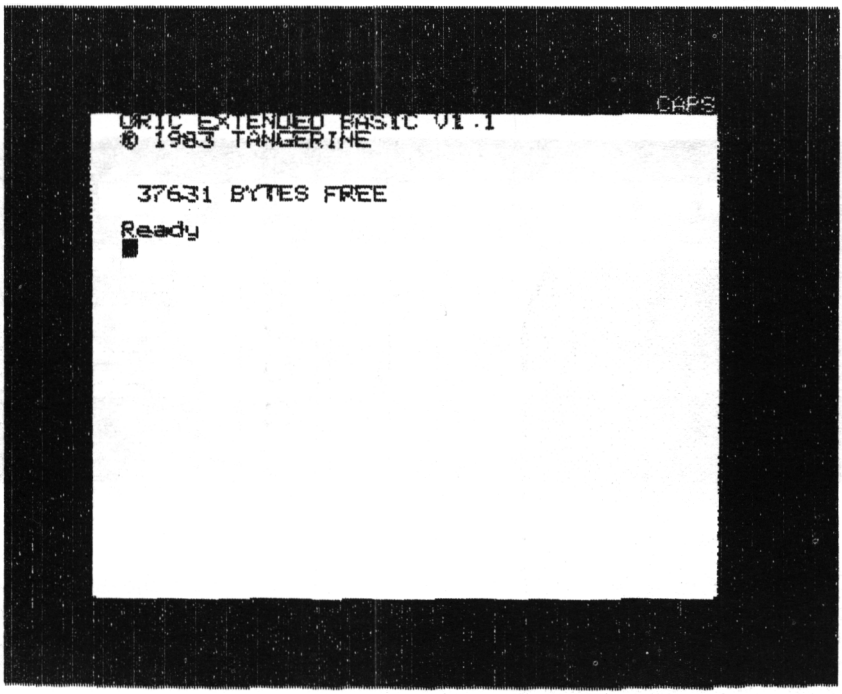
'ORIC EXTENDED BASIC V**
©1983 TANGERINE

***** BYTES FREE

Ready'

appears in black print on a white screen. The numbers represented by the '*' symbol will vary depending on the memory size (16k or 48k) and the machine 'version'. Currently there are two versions, V1.0 and V1.1.

The number of 'bytes free' is less than the total memory size. Some memory is used to control the screen display.

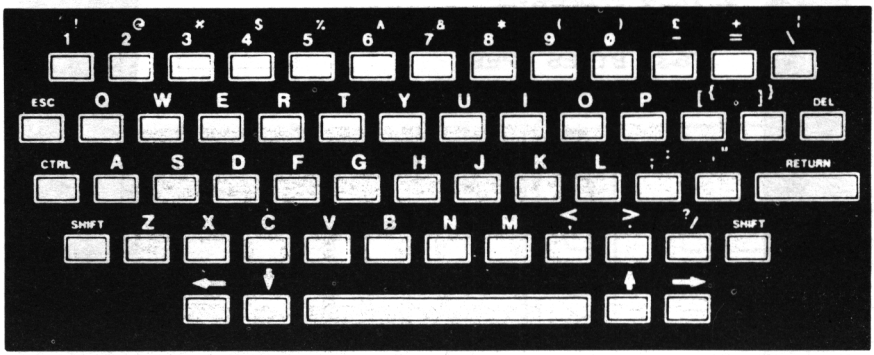


If instead of this message you find a rather odd black and white pattern on the screen, switch the computer back off and on again. Do this by unplugging and reconnecting the jack plug at the rear of the machine. This is hardened and will not wear. Switching power off and on again at the mains seldom works as the power then comes on too gradually to reset the machine correctly.

At the top right of the screen you should see the word 'CAPS' in white letters on the black border. To the left of the screen, below the word 'Ready' you will find a flashing square. This is called the 'cursor'.

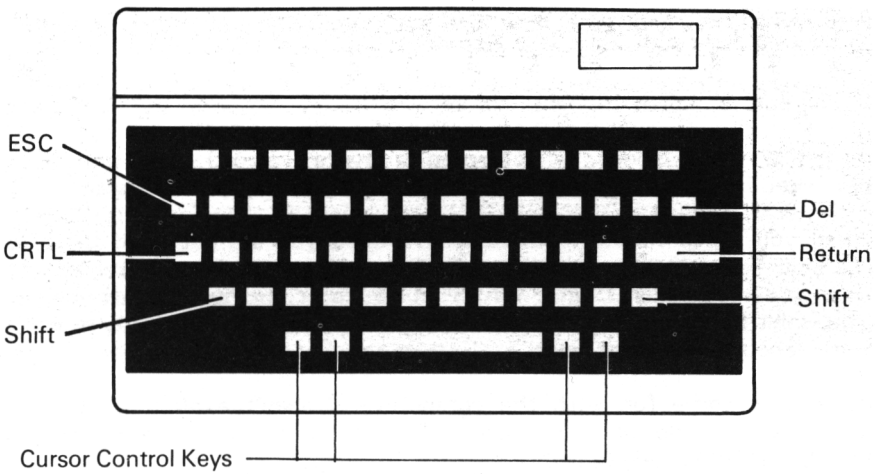
STARTING OFF

Let's have a look at the keyboard.



If you have ever used a typewriter the ORIC keyboard should not hold too many terrors for you. The layout is the standard typewriter 'QWERTY' keyboard with a few keys added. There is plenty of space between the keys and the lettering is clear and uncluttered.

Note the position of the two SHIFT keys, the CTRL (control) key, the ESC (Escape) key, the DEL (Delete) key, the Cursor control keys and the RETURN key.



Here are some important points to note.

1. I am going to ask you later in this chapter to press a SHIFT key (either of the SHIFT keys will do - they are both the same to the computer) and another key 'simultaneously'. What I really mean is that you should press SHIFT key first and hold it down while you press the other key; then release both keys. Many keyboard errors occur through not having the SHIFT key depressed first when a 'shifted' key is required.

2. In exactly the same way I am going to ask you to press the CTRL key and another key 'simultaneously'. In this case the CTRL key is pressed first and held down while the other key is pressed, and then both are released.

3. From now on when I ask you to 'press' a key I will mean press and then release it. If I want you to hold a key down I will say so.

4. If you type in a wrong character press the key marked DEL (just above the large RETURN key at the right of the keyboard). This will delete the last character entered.

Type in **PAPER 1** and press **RETURN.**

If you are lucky the 'paper' – i.e. the area inside the black border – will have changed to red. More probably it will have changed to a shade of grey. If that is the case tune the television until you have a red paper area. This may require delicate tuning. If there is a switch marked 'AFT' beside the tuning controls on the television it may help to turn this to the 'OFF' position.

If instead of changing the colour of the paper area the computer prints the message '?SYNTAX ERROR' on the screen (in computer jargon we talk about it 'returning a syntax error') then you have not typed in **PAPER 1** correctly. In this case please type it in again.

When you typed in the 'command' **PAPER 1** you did not need to press the SHIFT key to get capitals. All alphabetic keys give capital letters when pressed. This is what 'CAPS' in the top right hand corner means. You won't be using lower case (small) letters for some time yet, so don't worry about them right now.

Let's have some fun with the computer. I shan't explain what I am asking you to do at this stage. Please follow my suggestions 'on trust'. All will be explained later.

THE NOISY ORIC

Type in **ZAP** and press the **RETURN** key.

Some commands are not really all that difficult to understand!

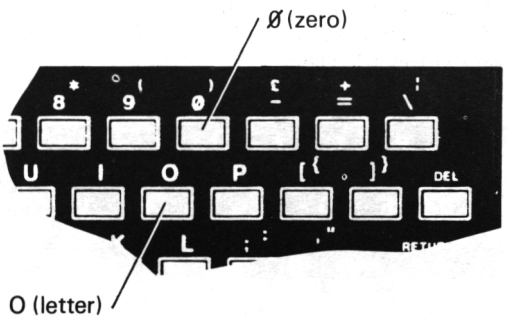
Now try **PING** again followed by the **RETURN** key.

Type in:

SHOOT:WAIT9:PING:WAIT9:ZAP:WAIT5:EXPLODE

(The colon (:) is SHIFT and the key to the right of key L pressed simultaneously.)

Be careful to use the letter O in 'SHOOT' and 'EXPLODE' and not the number Ø (zero). The number Ø (zero) has a stroke through it on the keyboard so that you can tell the difference. A lot of syntax error messages are caused by confusing the two keys.



Press **RETURN.**

You should already have discovered one thing about the ORIC. It can generate a lot of noise – not for ORIC users are the feeble chirps produced by some other microcomputers!

One more burst – and I hope you have thick walls or tolerant neighbours.

Type in:

FOR N=1 TO 1Ø:ZAP:WAIT 5:NEXT

followed as usual by the **RETURN** key.

ALL THE COLOURS

Let's try something quieter.

Type in

```
FOR N=Ø TO 7:PAPER N:WAIT 1ØØ:NEXT
```

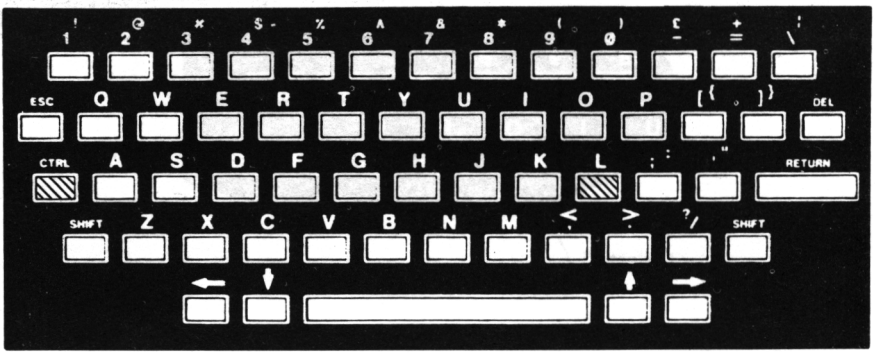
and press **RETURN**

Nice colours – aren't they!

CLEARING THE SCREEN

By now the screen is full of all sorts of rubbish. Let's see how to clear it.

Press CTRL and key L simultaneously.



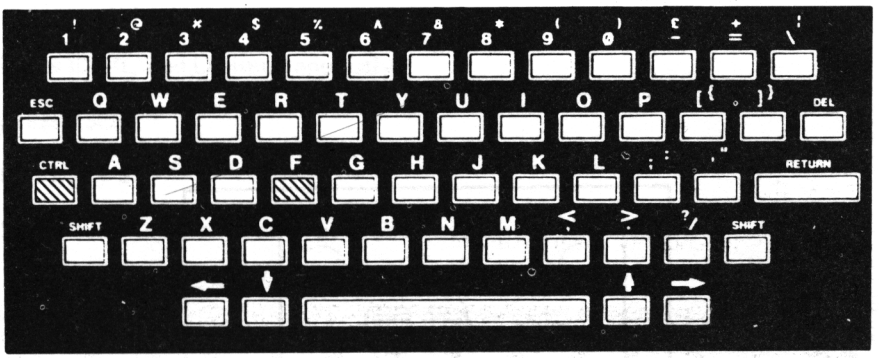
CTRL L – To clear Screen

Hey presto – a nice clean screen!

REMOVING THE KEYCLICK

You will have noticed that when you press a key you get a fairly loud click. This is to let you know the key has been pressed. If you like this idea – good! If, like me, you find the positive feel of the keyboard more than adequate 'feedback', and the keyclick a noisy nuisance, then you will want to know how to get rid of it.

To stop the keyclick press CTRL and key F simultaneously.



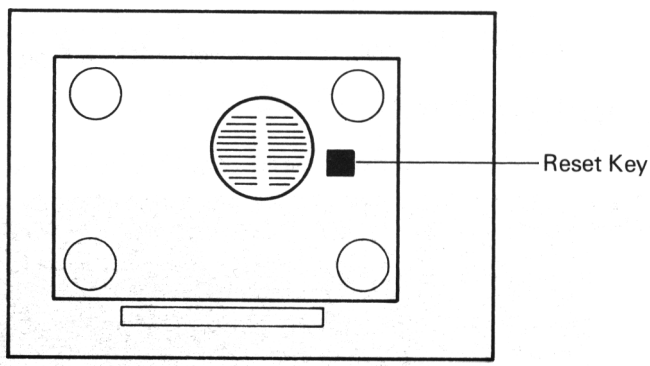
CTRL F – To remove Keyclick

If you want to get the keyclick back press CTRL and key F simultaneously again.

THE RESET SWITCH

Now let's explore the ORIC. Some very odd things are about to happen; so I had better tell you about the 'panic button'.

If you lift up the ORIC gently, so as not to pull the connections out of the back, you will see a small square hole. Inside this hole is a switch. This is the ORIC's reset switch. You will need a pencil or screwdriver to operate the switch. When you do so the screen clears and the 'ready' message is restored.



The advantage of the reset switch is that it is a 'warm start'. This means that 'programs' (see chapter 2) are not lost, as they would be if you powered off and on. If you get lost and cannot get the machine back to sane and sensible operations it is comforting to know the reset switch is there – but try not to use it too often.

SOME FANCY TRICKS

Type in **PAPER 0:INK 3**

and press **RETURN**

Press **CTRL** and **L** simultaneously.

You should have a black screen, blank except for the flashing cursor at top left.

Press **RETURN** to move the cursor down one line.

Press **CTRL** and **D** simultaneously.

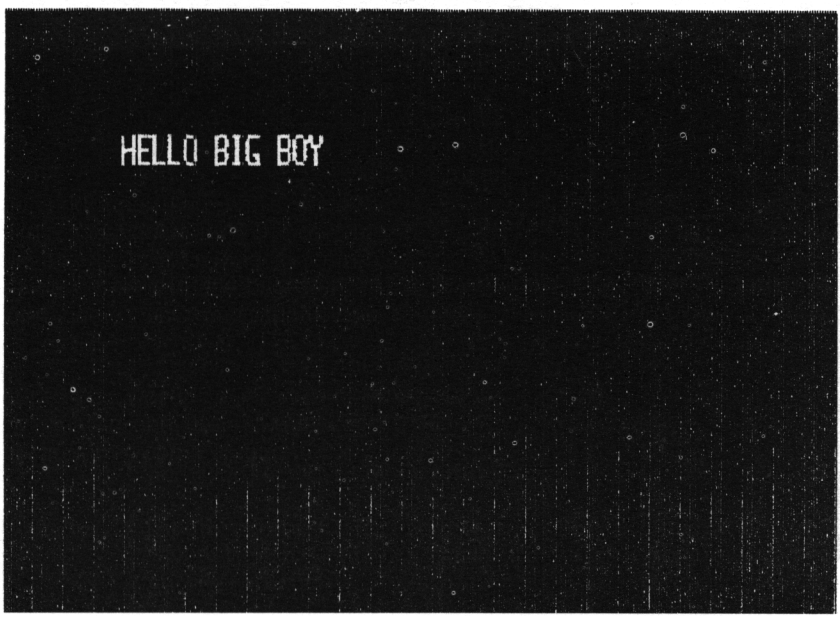
Press **ESC**

Press key **J.**

So far nothing much has happened. However if you type in:

HELLO BIG BOY

You may find the result rather surprising.



Press **RETURN.** Ignore the jumbled message which this produces.

Press **CTRL** and **L** simultaneously to clear the screen and then press **RETURN** to bring the cursor down one line.

Press **ESC** followed by **N.**

Type in:

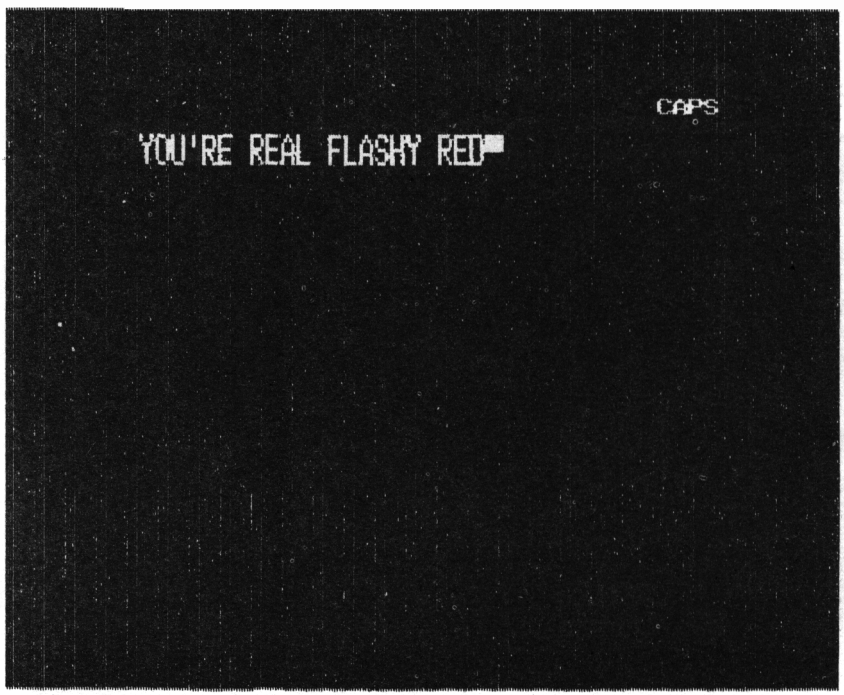
YOU'RE REAL FLASHY

Press **ESC** followed by **A**

Type in:

RED

Howzat!



Press **RETURN**, ignoring the odd message this produces. Clear the screen by pressing **CTRL** and **L** simultaneously and then press **RETURN** to bring the cursor down one line.

Press **ESC** followed by **J**.

Press **ESC** followed by **A** and type in **RED**.

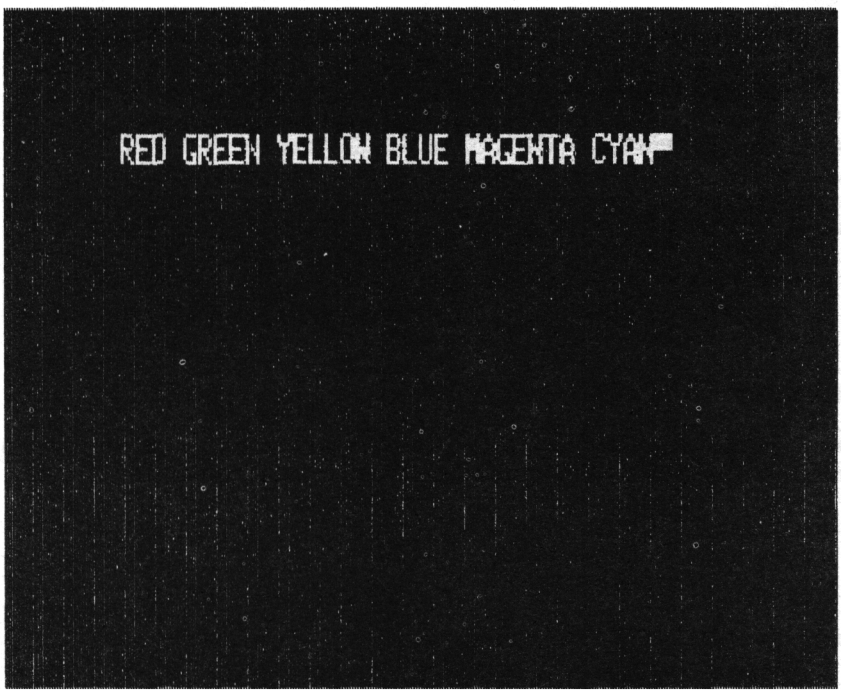
Press **ESC** followed by **B** and type in **GREEN**.

Press **ESC** followed by **C** and type in **YELLOW**.

Press **ESC** followed by **D** and type in **BLUE**.

Press **ESC** followed by **E** and type in **MAGENTA**.

Press **ESC** followed by **F** and type in **CYAN**.



Before we leave the large letters we will try one more demonstration. As before press **RETURN** and ignore the odd message. Press **CTRL** and **L** simultaneously to clear the screen, followed by **RETURN** to move the cursor one space downwards.

Press: ESC followed by **W;**

ESC followed by **@** (@ is SHIFT and 2 simultaneously);

ESC followed by **J;**

and type in **BLACK ON WHITE.**

Move the cursor down two lines using the cursor control key with the downward pointing arrow to the left of the space bar.

Move the cursor to the far left of the screen using the cursor control key with the left pointing arrow.

Press: ESC followed by **V;**
ESC followed by **A;**
ESC followed by **J;**

and type in **RED ON CYAN.**

As before use the cursor control keys to move the cursor down to spaces and left to the start of the line.

Press: ESC followed by **U;**
ESC followed by **B;**
ESC followed by **J;**

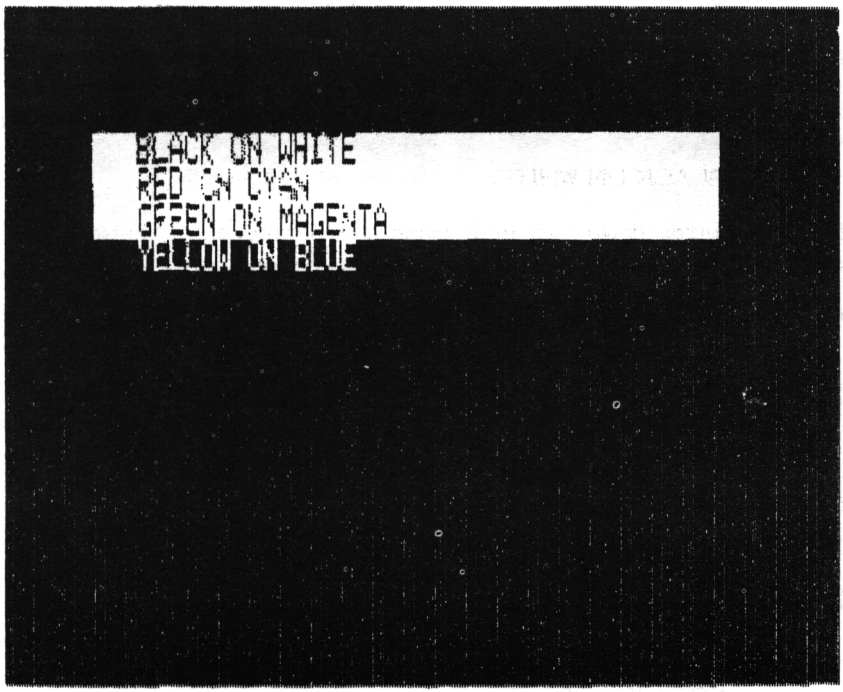
and type in **GREEN ON MAGENTA.**

Use the cursor control keys again to position the cursor at the start of the next double line. You should be getting good at this by now.

Press: ESC followed by **T;**
ESC followed by **C;**
ESC followed by **J;**

and type in **YELLOW ON BLUE.**

Finally press **CTRL** and **Q** together to hide the cursor. I hope you will agree that this is a fairly dramatic display of the machine's keyboard entry facilities.


BLACK ON WHITE
RED ON CYAN
GREEN ON MAGENTA
YELLOW ON BLUE

SOME PRETTY PICTURES

Words **are** boring – **Let's see some pictures!**

The first thing we will do is to get our cursor back. It is difficult to work without it.

Press **CTRL** and **Q** simultaneously.

Press **RETURN** and then press **CTRL** and **L** simultaneously; then press **RETURN** again.

Press **ESC** followed by **K**

and type in

N$£$£$£$£,W[^/ ^

(^ is key 6 and the SHIFT key pressed simultaneously)

Use the cursor control key to move the cursor down and left to the start of the first blank line (as you did in the last demonstration).

Press **CTRL** and **D** simultaneously, and then **CTRL** and **T** simultaneously.

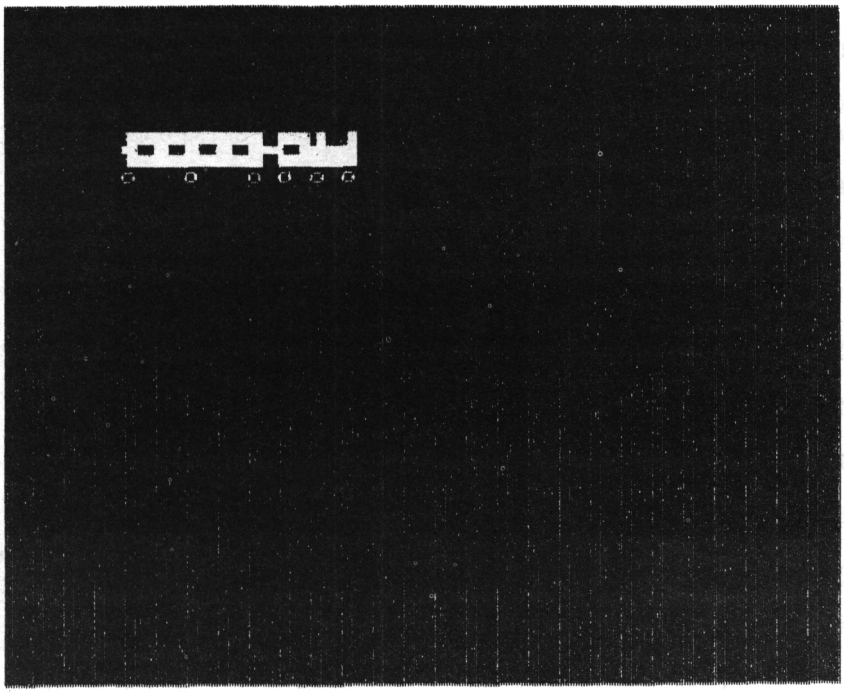
Press key **O** (that is the letter O, **not** zero).

Press the **space bar** three times.

Press key **O** and press the space bar 3 times.

Press **O space O space O space O.**

Press **CTRL** and **Q** simultaneously to hide the cursor.



If pictures you want then pictures you got!

Press **CTRL** and **T** simultaneously, then **CTRL** and **Q** simultaneously.

Press **RETURN** and clear the screen with **CTRL** and **L**.

Type in **HIRES** and press **RETURN.**

The cursor should move down to the bottom of the screen. You should see the word 'Ready' in yellow letters close to the cursor.

Type in: **CURSET 125,99,3:CIRCLE 9Ø,1**

and press **RETURN.**

A circle should appear on the screen.

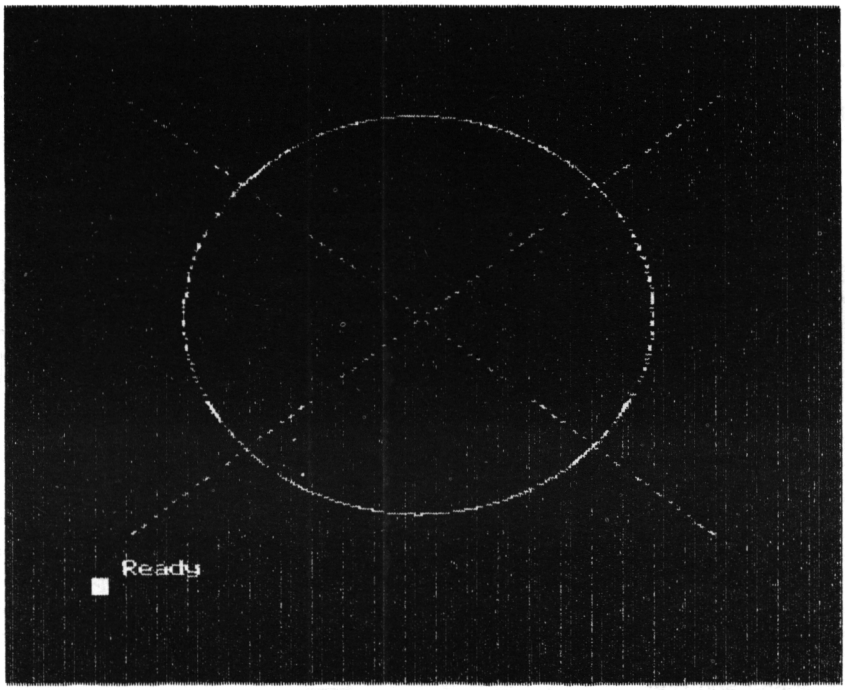
Type in: **CURSET 12,Ø,3:PATTERN 51:DRAW 227,199,1**

and press **RETURN.**

Type in: **CURSET 12,199,3:DRAW 227,-199,1**

and press **RETURN.**

You should now have a circle and two crossed dotted lines.


Ready

Clever enough, you may think, for a very small computer, but ORIC has rather more than that up its sleeve.

What you do now depends on the version of the machine you have.

If you have the 48K ORIC type in:

```
FOR N=#A000 TO #BFE0 STEP 40:POKE N,
INT(RND(1)*7+1):NEXT
```

and press **RETURN**.

If you have the 16K ORIC type in

```
FOR N=#2000 TO #3FE0 STEP 40:POKE N,
INT(RND(1)*7+1):NEXT
```

and press **RETURN**.

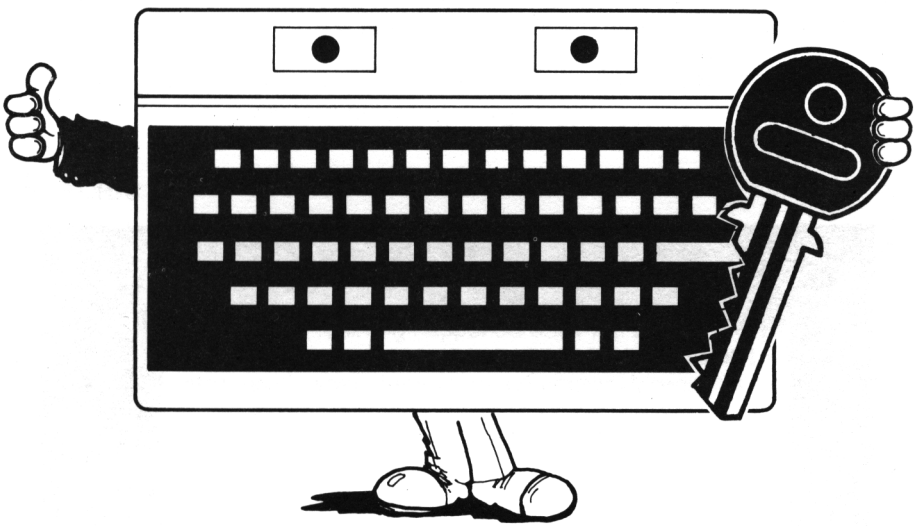
Note – press **RETURN** after **NEXT** in either case **not** after N,.

I hope these demonstrations have convinced you that you have a remarkable and versatile little machine. You may see some pattern in what we have been doing (if not don't worry) and may be interested in what would happen if you changed the instructions about a bit.

Go ahead and try!

CHAPTER 2

The Key to Success



THE KEYBOARD

In the previous chapter you used the keyboard a fair amount and you should have the 'feel' of it by now. The spacing between the centres of the keys is the same as on a standard typewriter.

Press the reset key underneath the machine, or switch power off and on again. This gets the machine back to its initial state.

SHIFTED KEYS

Some of the keys have two characters written above them. Pressing such a key causes the lower of these characters to appear on the screen. Pressing the same key and the SHIFT key (either SHIFT key will do) causes the upper character to appear.

For example pressing key 3 causes a 3 to be printed on the screen. Pressing key 3 and SHIFT simultaneously causes the symbol # to be printed on the screen (symbol # is 'shifted 3')

Practise using the shift keys – type in:

```
!@#$%^ &*()£+¦{} /:" <> ?
```

The symbol ^ (shifted 6) is displayed as ↑ on the screen.

DELETE KEY

The delete key DEL is used to delete keyboard entries if, for example, you press the wrong key or change your mind.

DEL moves the cursor one space to the left and deletes the last entered character to the left of the cursor as it does so. If the cursor is at the far left of a line then DEL will move it to the far right of the line above assuming that there is a character in that position which is part of the current entry. Otherwise DEL has no effect.

If you try to delete entries which have been 'terminated' (by pressing the RETURN key) you will get some odd effects.

RETURN

The RETURN key is used to 'terminate' entries. In the previous chapter you saw that the computer obeyed a command (did what you told it to do) only after RETURN was pressed. For example when you typed in 'ZAP' nothing happened until RETURN was pressed.

RETURN is also used when keying in computer 'programs' – see later in this chapter.

AUTOMATIC REPEAT

Press a key – say key Q – and hold it down until you have a whole line of Q's. Holding a key down has the same effect as pressing it repeatedly. This is a useful feature if you wish to key in, for example, a lot of spaces. It also works with the DEL key – so delete the line.

KEYBOARD BUFFER

Press an alphabetic or a numeric key and hold it down. The character chosen will appear on the screen repeatedly until eventually you will get a 'ping'. When you hear the ping release the key.

You are allowed to type in only a certain maximum number of characters before pressing a RETURN key. When you type in the characters they are put into a place called the **keyboard buffer,** where they are held until **RETURN** is pressed. The keyboard buffer can hold only so many characters. The ping is the ORIC method of warning you that its keyboard buffer is getting full.

LOWER CASE LETTERS

The keyboard on the ORIC, as with most computer keyboards, is similar in operation to that of the standard typewriter. So far, however, it has worked like a keyboard with the 'capitals lock' set. The alphabetic keys have given capital letters when pressed and the SHIFT keys have not affected them.

To remove capitals lock press CTRL and T simultaneously.

The message CAPS, at the top right hand corner of the screen disappears and pressing an alphabetic key gives the lower case (small) letter; pressing the same key with SHIFT gives the upper case (capital) letter.

You would work with CAPITALS LOCK OFF if, for example, you were using the ORIC for typing a letter. Normally, however, you will be working with capitals lock on, as any instructions you give to the computer must be in capital letters.

To put capitals lock back on press CTRL and T simultaneously again.

CONTROL CHARACTERS

In the previous chapter, and above, we have used the CTRL key, pressed along with another key, to make the computer do various things such as clearing the screen, switching the key click on and off, switching capitals lock on and off and so on. Pressing CTRL with another key generates what is called a **control character.** We will look at control characters again in chapter 7.

In the remainder of this book I will indicate a simultaneous CTRL and other key depression in the form **CTRL A, CTRL C** etc. **CTRL A** means that keys CTRL and A are pressed simultaneously.

I shall start by listing the control characters with a brief explanation for reference. Where necessary I shall then give more details. Don't worry if you don't understand some of the explanations right now. We shall look at them again in later chapters.

CTRL A – copies screen characters to keyboard buffer.
CTRL C – stops a program loop or scroll.
CTRL D – causes double print (on/off toggle).
CTRL F – controls key-click (on/off toggle).
CTRL G – causes a 'ping'
CTRL H – moves cursor to left.
CTRL I – moves cursor to right.
CTRL J – moves cursor down.
CTRL K – moves cursor up.
CTRL L – clears the screen.
CTRL M – acts like the RETURN key.
CTRL N – hides a line.
CTRL O – hides any subsequent screen output (on/off toggle).
CTRL P – controls printer (on/off toggle).
CTRL Q – hides cursor (on/off toggle).
CTRL S – hides the screen output (like CTRL O).
CTRL T – controls capitals lock (on/off toggle).
CTRL X – removes a line from program memory.
CTRL Z
CTRL [– affect background and foreground colours for the rest of the line.
CTRL] – allows printing in the far left column.

CTRL A

This control character is used in **editing.** That is changing the contents of the computer's 'memory'. We will look at its use in detail later in this chapter.

CTRL C

This is the 'break' or 'stop' character for the ORIC. You will see it in use throughout this book.

CTRL D

This causes double print. To see this happening key in **CTRL** D and then type in any message. **CTRL** D is what is known as an 'on/off toggle'. This means that if double print is 'off' **CTRL** D puts it 'on' and if double print is 'on' **CTRL** D puts it 'off'.

We don't want double print right now; so key in **CTRL** D again to put it off.

CTRL F, CTRL G, CTRL L

You have used **CTRL** F and **CTRL** L in the previous chapter and they should be familiar to you. **CTRL** F is an on/off toggle. The description of **CTRL** G is (I hope) self explanatory.

CTRL H, CTRL I, CTRL J, CTRL K, CTRL M

These control characters are seldom (if ever) used as direct keyboard entries. It is easier to use the cursor control keys, and RETURN which perform the same functions. The use of control characters not generated by keyboard entries is dealt with in Chapter 7.

CTRL N

If you use this after you have typed in a command then that command will disappear from the screen. However the command will still be carried out if you press RETURN.

CTRL O

This toggles information which would normally be displayed on the screen to the printer.

CTRL S

If you set **CTRL S**, characters entered from the keyboard are not displayed on the screen. A possible use is for 'secret' messages or for applications where entries might spoil a screen display. **CTRL S** is an on/off toggle.

CTRL Q

You used **CTRL Q** in chapter 1. The flashing cursor can detract from screen displays and it is useful to be able to get rid of it. **CTRL Q** is an on/off toggle.

CTRL T

This has been already dealt with in this chapter. **CTRL T** is an on/off toggle.

CTRL X

This cancels the keyboard entries to its left. It can be useful when keying in 'programs' – see later in this chapter.

CTRL Z, CTRL [

These can affect both background and foreground colours of print lines on the screen, and can have a range of other, rather odd, effects depending on which key is pressed after the control character has been keyed in. What these control characters are, in fact, doing is to prepare for some of the machine's 'attributes' – which we will look at next. **CTRL Z** and **CTRL [** are not particularly useful, but it is interesting to play with them and see the odd effects they can produce.

CTRL]

We will look at this character after we have discussed attributes.

ATTRIBUTES

If some of the things we can do with the **CTRL** key seem weird and wonderful, they pale into insignificance beside the very odd effects we saw when we used the **ESC** key.

The effect the ESC key has is determined by which key is pressed directly following it. What the ESC key does is to prepare the computer to put something called an **attribute** into the next free character space on the screen. The key following the ESC key determines which attribute is placed in this space.

The space on the screen into which the attribute is put is left blank. The attribute determines a characteristic of all the characters printed to its right on the screen, until the end of the line or another attribute is reached. There are a number of methods of putting attributes on the screen (or to be more exact into that section of the computer's memory which controls the screen). Here, however, we are considering direct keyboard entries, and so we will look at **ESC** key entries.

The **ESC** key has no effect by itself. Its effect is determined by the key pressed next (compare the **CTRL** key which needs a key pressed with it). When I talk of **ESC** and a key (say for example **ESC** and **A**) below I mean the **ESC** key followed by the key mentioned.

I shall first summarise the effects of **ESC** key entries and shall expand on the summaries where necessary.

ESC and @ – black foreground (ink).

ESC and A – red foreground.

ESC and B – green foreground.

ESC and C – yellow foreground.

ESC and D – blue foreground.

ESC and E – magenta (purple) foreground.

ESC and F – cyan (light blue) foreground.

ESC and G – white foreground.

ESC and H – single height, steady, standard characters.

ESC and I – single height, steady, alternate characters.

ESC and J – double height, steady, standard characters.

ESC and K – double height, steady, alternate characters.

ESC and L – single height, flashing, standard characters

ESC and M – single height, flashing, alternate characters.

ESC and N – double height, flashing, standard characters.

ESC and O – double height, flashing, alternate characters.

ESC and P – black background (paper).

ESC and Q – red background.

ESC and R – green background.

ESC and S – yellow background.

ESC and T – blue background.

ESC and U – magenta (purple) background.

ESC and V – cyan (pale blue) background.

ESC and W – white background.

BACKGROUND AND FOREGROUND

These terms are (I hope) self explanatory. You can use foreground and background attributes at the beginning of a row to set ink and paper colours and in the middle of the row to change these colours.

CTRL]

An attribute normally takes up a character position. Thus when we are entering text normally the first two characters of each line are reserved for attributes. If you wish to print in the two far left hand columns on the screen **CTRL]** allows you to do this. **CTRL]** is an on/off toggle.

FLASHING AND STEADY

If a line of text is preceded by a flashing attribute, it will flash – ie change from background to foreground – about three times per second. Text not preceded by a flashing attribute will remain steady in foreground colour. **ESC** followed by **L, M, N** and **Q** insert flashing attributes.

STANDARD AND ALTERNATE CHARACTERS

As well as the normal characters (A,B,C, ... 1,2,3, etc) ORIC has an alternate character set. These are a collection of shapes which can be used to form pictures, such as for example the train we typed in in Chapter 1. We will be looking at this character set in detail in Chapter 12.

ESC followed by **I, K, M** and **O** insert alternate character attributes.

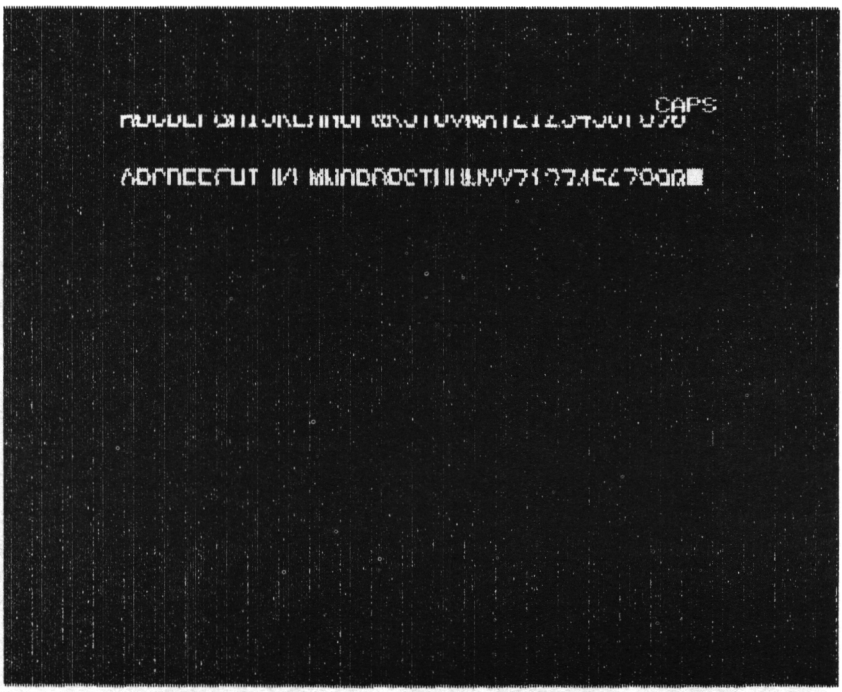
DOUBLE HEIGHT CHARACTERS

The effect of the double height attributes varies depending on the line on the screen in which they are used. If a double height attribute, for example, is put in the very top line (line zero) you will get the lower half of the double height character you type in.

For example if you type in 'A' you will get '▟', if you type 'C' you will get '▙'.

On the other hand if the double height attribute is on the next line down (line 1) you will get the top half of the double height character you type in.

For example if you type in 'A' you will get '▛'; if you type in 'C' you will get '▜'.

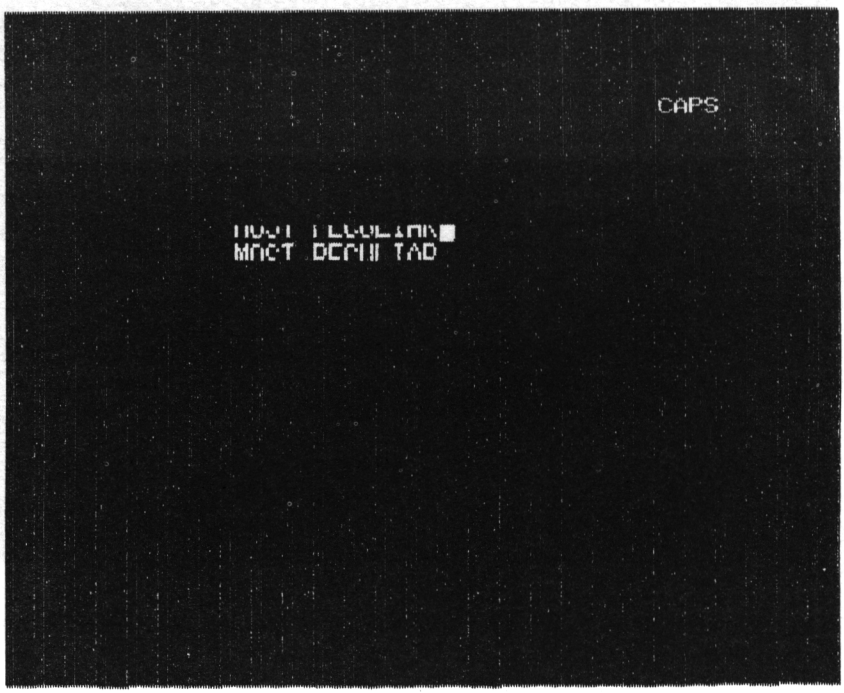


This continues all the way down the screen. You will get bottom halves on lines 0, 2, 4 etc and top halves on lines 1, 3, 5 etc.

Thus to get a double height message you first type the message on an odd numbered line with a double height attribute, and then type the same message on the next line down, again with a double height attribute. The top half of the message on the upper line then joins up with the bottom half on the lower line to give a double height message.

Typing in a message twice, however, is a bit much like hard work. There must, you would think, be an easier way. There is! Remember the rather 'odd' effect of **CTRL D**. The message typed in with this control character set appears on two lines at once – just what you want for double height print. Thus to get double height print, set **CTRL D**, then insert a double height attribute using **ESC** followed by **J** (static, standard characters) or **K** (static, alternate characters), or **N** (flashing, standard characters) or **O** (flashing, alternate characters). When the double height control character is set the attribute will control both lines. Double height attributes also force a black background; so make sure you set a foreground colour other than black. Otherwise you won't see the message.

Also make sure you start at an odd numbered line. Otherwise the effect can be most peculiar.

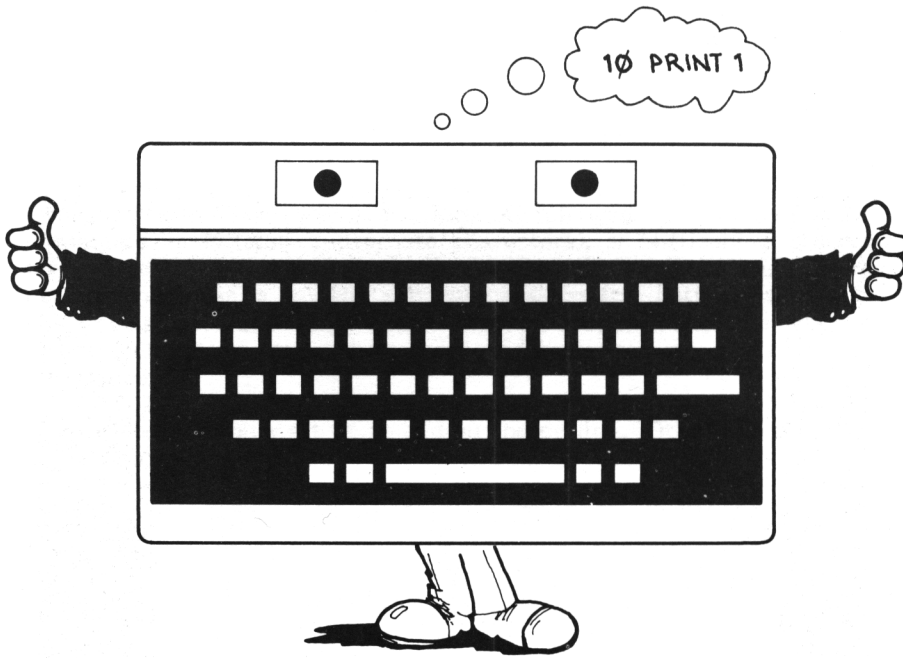


60Hz ATTRIBUTES

Some attributes will upset your screen display, because they are designed to work with American 60Hz television sets. Avoid following **ESC** with **X,Y,^,£,/,¦,]**, or **}**.

CHAPTER 3

Enter the ORIC



ENTERING PROGRAMS

In Chapters 1 and 2 you 'keyed in' some 'commands'. In plain English this means you pressed a few keys and this made the computer do something.

Once the computer had done what you had told it to, however, it forgot the command. To get the computer to repeat the action you would have to type in the same keys all over again.

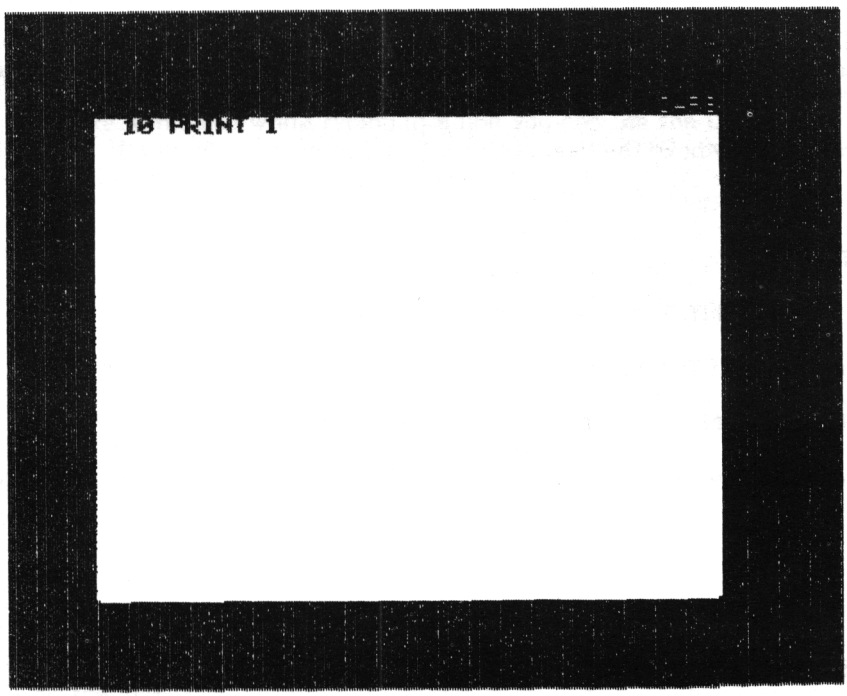
A command which the computer executes immediately and then forgets about is called an 'immediate' command.

Let's try something different; first clear the screen using CTRL L, then type in:

```
10 PRINT 1
```

and press the RETURN key. You don't have to insert a space before and after PRINT, although it makes no difference if you do.

The screen should now look as follows:



Type in RUN and press the RETURN key – the number '1' will appear on the screen. If instead you get the message 'SYNTAX ERROR' repeat the whole procedure (starting at the top of the page) again.

Now here's the trick – type in RUN followed by the RETURN key again.

And again you get the number 1. The computer has remembered the instruction, and will continue to do so until either you delete or change it, or until power is switched off.

By numbering the instruction you have caused the machine to store it in memory. The instruction is therefore no longer an 'immediate' command, but is now a 'program' command. Programs consist of instructions with line numbers. The computer stores these instructions and 'executes' them (carries them out) each time the program is RUN.

You have just entered and run your first program.

LINE NUMBERS

The numbers at the beginning of the program lines do not only define the instructions in these lines as program instructions. They are also one of the factors determining the order in the instructions are carried out.

You already have the line:

```
10 PRINT 1
```

Type in:

```
5 PRINT 3
```

and press **RETURN.**

Type in **RUN** followed by the **RETURN** key to run the program.

You will see that the screen now displays the results:

```
3
1
```

Line 5 is carried out before line 10, even though it was entered after it.

Type in LIST, followed by the RETURN key.

The two lines you have entered will now appear on the screen:

```
5 PRINT 3
10 PRINT 1
```

This is what is know as a 'program listing'. You will see that not only is line 5 'executed' (carried out) before line 10, it is also listed before it.

ENTERING PROGRAMS

When you are typing in programs there are some points to note. You may have noticed that you did not need to put a space between the number at the start of the line (the 'line number') and the instruction which follows it. When you listed the program ORIC put the spaces in for you.

When you finished typing in a line you pressed the RETURN key. This 'terminates' (finishes) the line and 'enters' it into 'program memory' – so that the machine remembers it and carries out the instructions when 'RUN' is entered. Remember to press the RETURN key after typing in each program line throughout this book.

If you make a mistake while typing in a line you can use the DEL key to delete it. If you get completely lost, however, press **CTRL X** to cancel the whole line and start again.

EDITING

LIST the program again. The screen should now read:

```
LIST
5 PRINT 3
1Ø PRINT 1
```

Suppose we want to change line 1Ø to:

```
1Ø PRINT 4
```

There are two methods of doing this:

1. Enter the whole line again.

2. Edit the line.

For a line as short and simple as line 1Ø the first method is the easier. Simply type in

```
1Ø PRINT 4
```

and press RETURN.

You will see that the previous line 1Ø has been replaced by the new line 1Ø.

Suppose you want to change line 5 to:

```
5 PRINT 3+6
```

This time we will use the EDIT facility. To edit a line we must get the cursor to the start of that line. You could use the cursor control keys to move the cursor up to line 5. It is, however, rather too easy to get lost among all the print on the screen if you try to EDIT a line in place. It is much safer, at least until you have gained more practice in EDITing, to move the line you want down to a space below the other printing. To do this type in:

EDIT 5

followed by RETURN.

Press **CTRL A** and see how the cursor moves along the line. Remember that earlier I mentioned that keyboard entries go into a store called the keyboard buffer which holds them until the RETURN key is pressed. Well, moving the cursor over a character on the screen puts that character into the keyboard buffer just as if it had been entered via the keyboard.

Thus if you use **CTRL A** to move the cursor across line 5 until it is in the space to the right of the last character as below

```
5 PRINT 3
```

then the keyboard buffer will hold '5 PRINT 3' just as if you had typed it in.

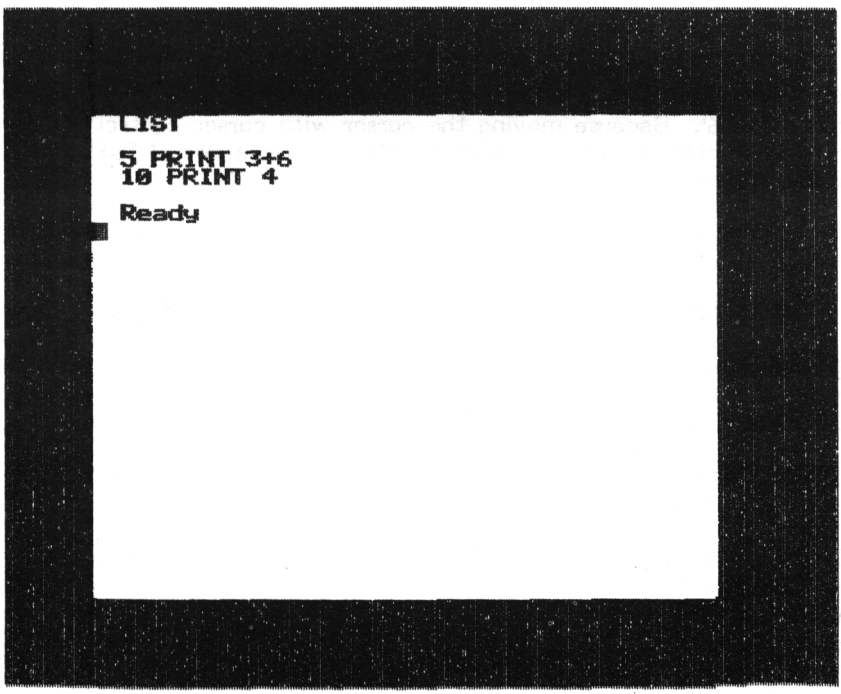
Type in +6. This will appear at the end of the line so that it reads:

```
5 PRINT 3+6
```

The additional entries have also gone into the keyboard buffer. This buffer now holds all the required characters – so press RETURN.

To check that this has worked clear the screen and LIST the program. This should read:

```
5 PRINT 3+6
1Ø PRINT 4
```



Suppose you want to edit a line so that part of it is missed out. You can do this by careful use of CTRL A and the cursor control keys.

Moving the cursor over a character on the screen with CTRL A copies that character into the keyboard buffer. Moving the cursor over a character with the cursor control key does not copy that character into the keyboard buffer.

For example, suppose you want to change line 5 to:

```
5 PRINT 6
```

– that is taking out the characters '3+'.

Move the cursor to the left of the line. If you wish you can use EDIT 5 as before, or you can simply use the line 5 already on the screen.

Move the cursor across the line using CTRL A until it is over the character '3'. This means that the keyboard buffer contains the characters:

```
5 PRINT
```

Use the right cursor control to move the cursor until it is over the character '6'. Because moving the cursor with cursor control does not copy characters into the keyboard buffer, the contents of the keyboard buffer will not be changed.

Finally use CTRL A to move the cursor over the character '6'. This copies this character into the keyboard buffer, which holds:

```
5 PRINT 6
```

This is what we want; so press RETURN.

Clear the screen and LIST the program. This should read:

```
5 PRINT 6
1Ø PRINT 4
```

Caution – when you are changing something in the middle of a line it is very easy to forget to copy the rest of the line with CTRL A. Please watch out for this error.

Hang on to your hats – we are coming to the hard bit!

Suppose you want to insert some characters in the middle of a line without removing any which are there already. At first sight this is not easy – when you type in the new characters you obliterate other characters in the line. Again the secret is in the use of the cursor control keys as well as CTRL A.

This is best explained by an example. Suppose we want to change line 1Ø to:

```
1Ø PRINT 1+3+4
```

When inserting characters into a line you need a bit of clear screen to work on – so use the EDIT facility.

Type in EDIT 1Ø followed by RETURN.

Line 1Ø should be:

```
1Ø PRINT 4
```

Move the cursor over the line with CTRL A until it is over the character '4'.

Move the cursor down one line with the cursor control key and type in:

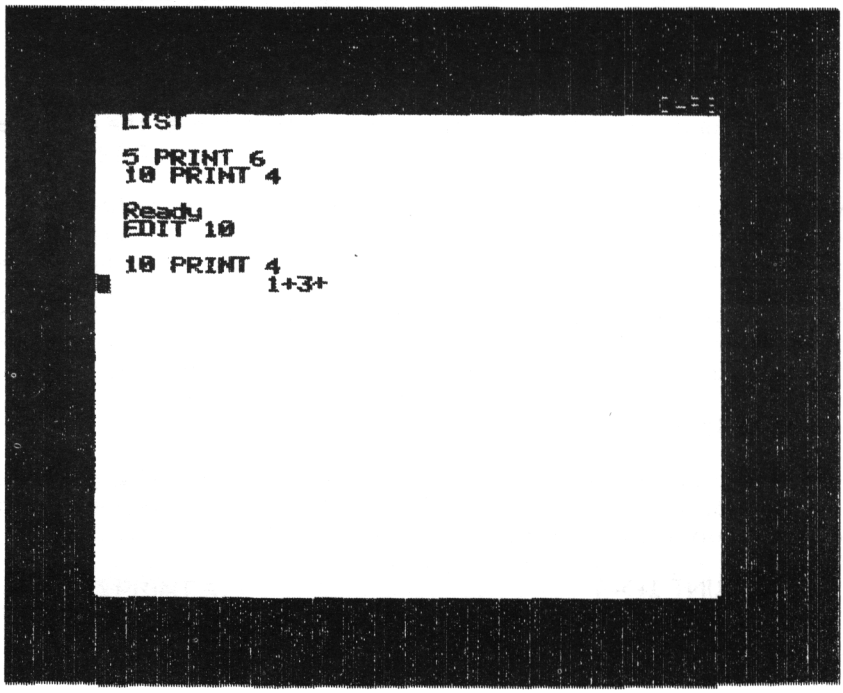
```
1+3+
```

Because moving the cursor over screen characters with CTRL A and typing them in has the same effect – that is to put them in the keyboard buffer – the keyboard buffer will contain

```
1Ø PRINT 1+3+
```

Move the cursor up and to the left using the cursor control keys until it is again over character '4'. Use CTRL A to 'copy' this character into the keyboard buffer, which now contains:

```
1Ø PRINT 1+3+4
```



This is what we want – so press RETURN.

Clear the screen and LIST the program. This should now read

```
5 PRINT 6
1Ø PRINT 1+3+4
```

RUN the program (type in RUN and press RETURN) to get the output:

```
6
8
```

The use of CTRL A is a powerful method of EDITING. It means you can make up a program line by copying anything on the screen. It is, however, a fairly difficult method to get used to, especially as you cannot see what is in the keyboard buffer while you are EDITING.

Don't worry if you get lost at first – a bit of practice and you will soon get the hang of it. Check your EDITING by clearing the screen and LISTING the program.

> **Caution – if you get really lost while EDITING a line remember to use CTRL X to cancel your entry. If you press RETURN you will probably end up with a 'nonsense' line and be worse off than when you started.**

If you change the line number when you EDIT a line then you will create a new line and will not alter the line you started with. This is a useful method of creating new lines whose content does not differ greatly from existing lines.

Again this is best seen from an example. Suppose we wish to add to our existing program the line:

```
2Ø PRINT 1+3+6
```

Use the cursor control keys to move the cursor to the start of line 1Ø and type in the character '2'. This will overwrite the '1' and the cursor will be over the character 'Ø'.

Move the cursor along the line using CTRL A until it is over the final character '4'. Type in a '6'.

The keyboard buffer now holds the characters

```
2Ø PRINT 1+3+6
```

This is what we want – so press RETURN.

Note that line 1Ø is not altered. We used its display on the screen to create line 2Ø but this does not alter line 1Ø in program memory.

Line 1Ø would only be changed had we kept that line number unaltered while editing.

Clear the screen and LIST the program. This should read:

```
5 PRINT 6
1Ø PRINT 1+3+4
2Ø PRINT 1+3+6
```

RUN the program to get the output

```
6
8
1Ø
```

Note – When you are editing a program line after using the EDIT command you may find the cursor is not at the start of the line. This happens when the line takes more than one row on the screen, especially with the V1.Ø machine. Use the cursor control keys to position the cursor before editing.

LINE NUMBERING

You will rembember that the very first line you entered had line number 1Ø, rather than line number 1. Possibly you have already realised why this was done. It let you insert a line before line 1Ø. Usually we number lines in steps of ten (1Ø, 2Ø, 3Ø ...), so that if additional lines are required they can easily be inserted.

DELETING A LINE

If you wish to delete a line in a program type in the line number followed by RETURN.

For example type in 5 and press RETURN.

Clear the screen and LIST the program.

Check that line 5 has disappeared.

As a double check, RUN the program to get the output:

```
8
10
```

NEW

We have finished with the program; so we shall see how to get rid of it. We could delete it a line at a time, but there is an easier way.

Type in NEW and press RETURN.

Clear the screen and LIST. The program has gone – nothing is LISTED.

The NEW command is used to clear away any program in memory and give the programmer a 'clean slate'.

Take care when using this command. Make sure you really do want to clear all the program memory.

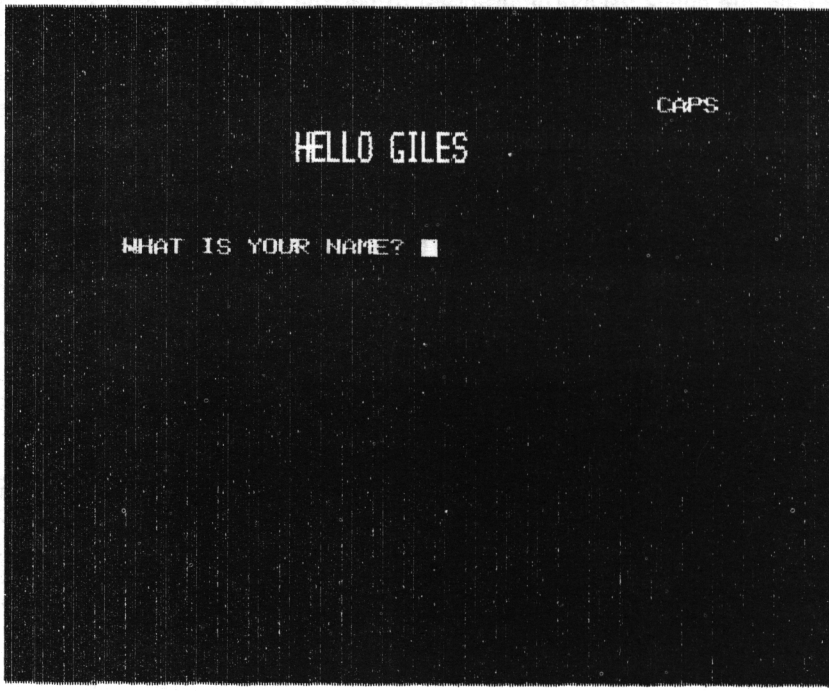
A DIVERSION

You have been working very hard with EDIT, and I hope you didn't get too bored. Before we go on we shall have a little fun. Type in this program. Remember to press the RETURN key after each line. Don't worry about understanding it at this stage.

```
10 PAPER 0:INK 3:CLS
20 REPEAT
30 PRINT:PRINT:PRINT:PRINT
40 INPUT"WHAT IS YOUR NAME";A$
50 IF A$="ANNE" THEN A$="FATSO"
60 PRINT CHR$(12)
70 PRINT CHR$(4);SPC(10);CHR$(27);"JHELLO ";A$
80 PRINT CHR$(4)
90 UNTIL 0
```

Line 70 will take up two lines on the screen but this doesn't matter.

RUN the program. Type in your name when requested, followed by RETURN.

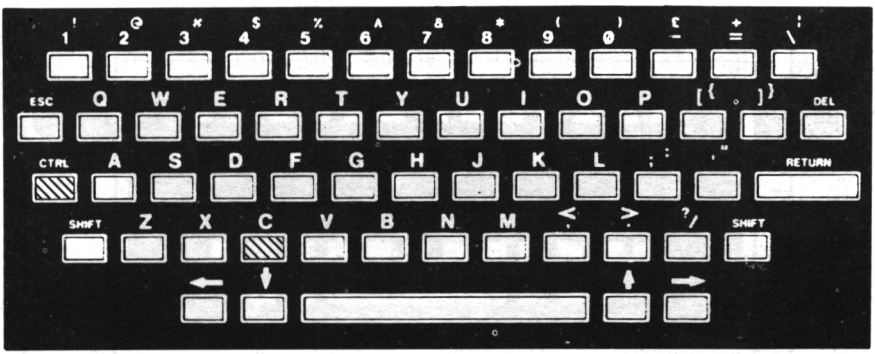


The result of typing in 'ANNE' (followed by RETURN) is interesting. This, as you may have guessed is my wife's name, but I am quite safe as she never reads my books! Try changing line 5Ø to suit your own circumstances.

I apologize to any of my readers called Anne. I don't mean you – honest!

BREAK

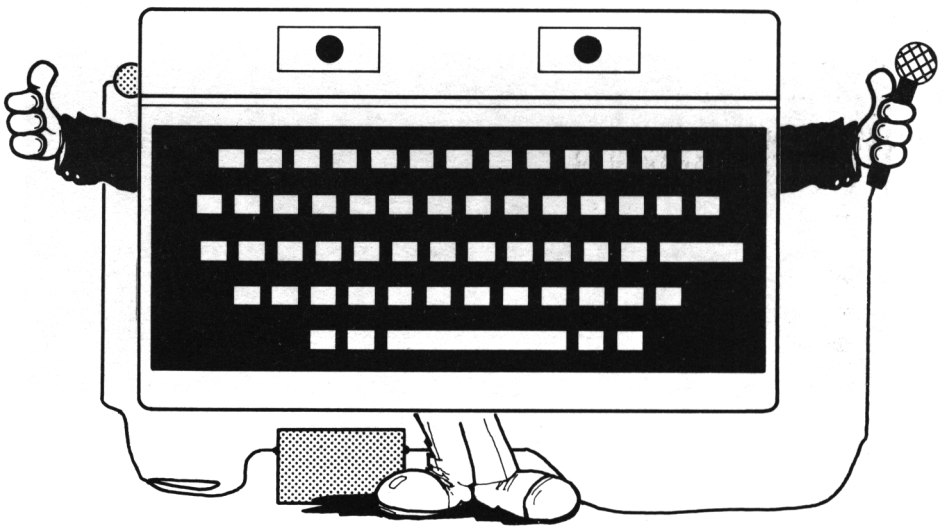
Once you have grown tired of the program you will want to stop it. You may remember that I mentioned CTRL C in the previous chapter and said you would use it throughout the book. This is the first example. Break from the program CTRL C when the computer asks for your name. Don't delete the program just yet, as we will be using it in the next chapter.



CTRL C – To break from program

CHAPTER 4

Loading and Saving Programs



I have put this chapter near the beginning of the book quite deliberately. Why, you will probably ask, learn to save programs on tape before learning how to write them? There are two reasons:

1. Keying in even small programs all the time is tedious, and leads to error. You should get into the habit of saving anything you want to use later as soon as you have 'debugged' it (got it to work). Thus you don't have to debug it several times over. It is useful to save even faulty programs if you are interrupted in the middle of debugging them.

2. There are a number of excellent commercial program tapes available for the ORIC. You will soon be writing your own programs, but that should not prevent your purchase, use and enjoyment of professionally written software.

CASSETTE RECORDER

You will need a cassette tape recorder. You may decide to use one you already have, or you may purchase one especially for use with the ORIC.

You do not need an expensive recorder with stereo or tone control. If you have these facilities then adjust the balance to give single channel recording and the tone control to give maximum treble and minimum bass response.

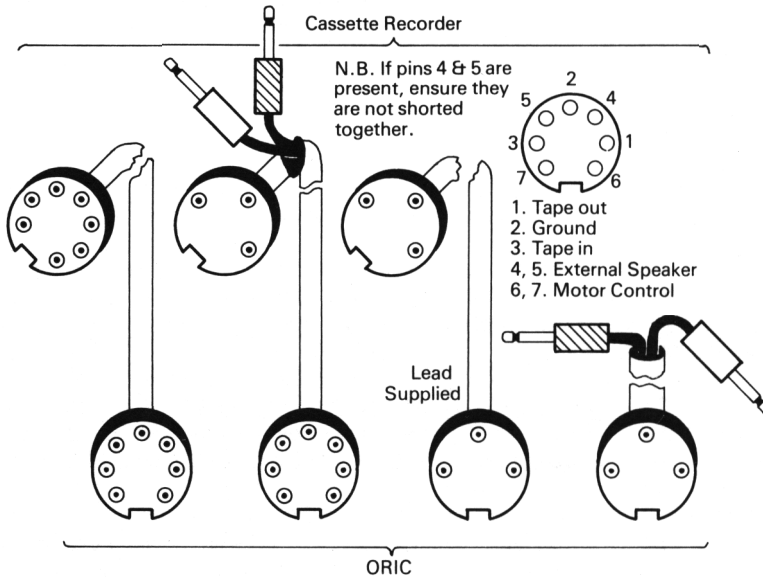
You will find a tape counter extremely useful. In fact I would go so far as to say that if your present machine does not have this facility you should seriously consider purchasing a machine which does.

The cassette connector at the back of the ORIC consists of a round, seven hole socket known as a DIN socket. The connector on your tape cassette recorder could be one of the following:

1. a 3 hole DIN socket;
2. a 5 hole DIN socket;
3. two jack sockets marked EAR and MIC;
4. any of these plus a motor control socket.

If your recorder has a three or five hole DIN socket then use the 3 pin to 3 pin DIN connector provided with your machine. Do **not** use a 5 pin connector.

If you have any other connector on your recorder, then obtain advice from your computer retailer. Motor control (automatic start and stop) is a useful feature and the ORIC provides switching for this. Be careful – some tape recorder motors are too powerful for the ORIC to control. Again your computer retailer should be able to advise you.



DIN Connectors

CASSETTE TAPES

ORIC programs are saved on ordinary audio cassettes – the same cassettes that you would use for recording music. I always pay that little extra for good quality low noise tapes, which, in my opinion, give a worthwhile increase in reliability. You may prefer to use the very short tapes (C1Ø or C15) stocked by computer retailers. These can save time when it comes to finding programs on tape.

TECHNICAL TERMS

When I talk about **Saving** a program I mean recording the program, which is in the computer's program memory, on to cassette tape.

When I talk about **Loading** a program I mean 'reading' the program from the tape cassette and putting it into the computer's

program memory. That is to say putting back into the computer's memory a program which has been saved onto cassette tape at some earlier time.

CSAVE

To save a program to tape cassette, ORIC uses the command CSAVE (which is short for Cassette Save).

We are going to save on to tape the program which we typed in in the previous chapter. If you switched the computer off between chapters, then I am afraid you will have to type the program in again.

Put a new cassette in the recorder and rewind to the start. Set the tape counter to zero.

Use the 'fast forward' control to feed on the tape until the counter is at about ØØ5. This ensures that you do not try to record on the tape leader (the plastic bit at the start of the tape).

Make sure the lead between the computer and the tape recorder is not near the power leads or the lead to the television. Otherwise it could pick up interference from these leads.

To save a program you have to give it a name. This name can be anything you wish provided it is not more than seventeen characters long (V1.Ø machine) or sixteen characters long (V1.1 machine). Let's call our first program DEMO1.

Type in CSAVE "DEMO1", S

Start the tape recorder recording (if your recorder has an automatic start you may not need to do this) and press RETURN.

After a few seconds the message 'Saving DEMO1' (or 'Saving DEMO1 B') should appear at the top of the screen. When saving is complete the word 'Ready' appears on the screen. Stop the tape recorder.

CLOAD

Use the **NEW** command to clear the program memory. We are going to see how to get the program back from tape into the computer. To do this the ORIC uses the command CLOAD (short for Cassette Load).

Wind the tape back until it is at a tape counter number just less that that at which you started to save (ØØØ will do nicely in this case). Type in CLOAD"DEMO1",S taking care that the name DEMO1 is exactly the same as the name used when saving – spaces **do** matter here.

Press RETURN and start the recorder playing. The message 'Searching ..' should appear at the top of the screen, followed by the message 'Loading DEMO1' (or 'Loading DEMO1 B') when the start of the program is detected. When loading is complete the word 'Ready' will appear on the screen. Switch off the tape recorder and RUN the program.

CASSETTE RECORDER PROBLEMS

If the word 'Ready' does not appear try CLOADing again at different volume settings on the tape recorder, remembering to rewind the tape before each attempt. You may need to press the reset switch under the computer to get the machine out of 'searching' mode. If your recorder has EAR and MIC sockets of the same size try changing the connections round.

You may get the error message 'FILE ERROR – LOAD ABORTED' when you try to load. This at least indicates you have something on your tape.

If you still have no success, try saving a smaller program. Type in

```
10 PRINT 1
```

and press RETURN, then go through the saving procedure, clear program memory and try to reload.

If there is still no success try playing the recorder without the computer attached.

If there is nothing on the tape then there could be a fault in your connector or tape recorder. Test the recorder by recording something other than a computer program.

If you get a rather loud, unpleasant, high pitched noise then you have a program (or something very similar) on tape. The recording head of your tape recorder may be out of alignment. Please refer to the maker's instructions for adjusting this.

SLOW AND FAST

Let us, however, assume that nothing has gone wrong and you have recorded the program sucessfully. You probably wondered what the 'S' in the CLOAD and CSAVE commands is for.

When a computer saves information to or loads it from a tape it does it at a certain speed. This is nothing to do with the speed at which the tape moves, which remains constant; it is the speed at which information is sent down the line between the computer and the recorder. This is known as 'the baud rate'.

The ORIC has two baud rates. You have saved and loaded a program at 300 baud, which is a fairly slow rate. The 'S' in the command stood for 'slow'.

The ORIC can also save and load information at 2400 baud – 8 times as fast. This saves you time, and also means that you get 8 times as much information on your tape.

The catch is that, because the information is packed so densely on tape, the slightest flaw in the tape or in the information on it can lead to a 'bad' program – ie. one which will not load. 300 baud is much more reliable than 2400 baud. At the faster speed the alignment of the recording head and the volume setting of the recorder are much more critical.

To save the program at normal speed, wind your tape onto a fresh section (say tape counter number 030) and follow exactly the same procedure, except that the command you use is:

```
CSAVE "DEMO1"
```

followed as usual by RETURN.

Test that your program has been saved by loading it back into memory. Again the procedure is the same as before, except that the command you use is:

```
CLOAD "DEMO1"
```

You may find you need several attempts to get the best volume setting before the program will load.

A program must be loaded back at the same speed (300 or 2400 baud) at which it was saved. Otherwise it will not load.

VERIFYING PROGRAMS

It is convenient to be able to check that a program has been saved correctly while that program is still held in program memory. The V1.1 provides the verify feature which allows ORIC to compare a program on tape with one in memory without corrupting the program in memory.

A special form of CLOAD command is used to verify. This is:

```
CLOAD"...",V
```

or `CLOAD"...",V,S`

depending on whether the program you are verifying has been saved at 2400 or 300 baud.

This is best explained using an example. You should have DEMO1 saved at slow speed at the start of your tape. You should also have DEMO1 in program memory – if not load it in.

Wind the tape back to start. Type in:

```
CLOAD"DEMO1",V,S
```

press RETURN and play the tape.

You should get the message 'Searching' as before, followed by 'Verifying DEMO1 B'. When the message:

```
0 Verify errors detected

Ready
```

appears on the screen then stop the tape. You have verified that the program on tape is the same as that in program memory.

If your verification fails (i.e. you get a non zero count of verification errors) then adjust the volume on your cassette recorder and try again. If you still cannot verify make sure your machine is not version V1.0. **Only** version V1.1 has verify.

If you managed to save DEMO1 at the faster speed, try verifying that save also. Use:

```
CLOAD"DEMO1",V
```

The V1.1 machine will also accept

```
CLOAD"",V,S
```

and `CLOAD"",V`

These commands check the content of program memory against the first program that they find on tape.

JOINING PROGRAMS

Large programs are often made up of a lot of small programs joined together. The V1.1 machine lets you join a program held on tape on to a program held in memory.

Note – this does not work on the V1.Ø machine.

Again this is best illustrated by an example. Clear program memory by typing in NEW and pressing RETURN. Wind the cassette tape on so that you are not overwriting anything you have saved.

Key in the program:

```
1ØØ PRINT"SEE HOW THEY RUN"
```

RUN this program and get the message

```
SEE HOW THEY RUN
```

on the screen.

Save this to tape as DEMO1A. Verify that it has been saved correctly.

Clear the program memory and enter the program:

```
1Ø CLS
2Ø PRINT
3Ø PRINT
4Ø PRINT"THREE BLIND MICE"
5Ø PRINT
```

RUN the program. You should get the message:

```
THREE BLIND MICE
```

on the screen.

Wind the cassette tape back to just before the start of program DEMO1A. Key in:

```
CLOAD"DEMO1A",J
```

or `CLOAD"DEMO1A",J,S`

depending on whether you saved DEMO1A at normal or slow speed. Play the tape.

When loading has fininished (Ready appears on the screen), LIST the resulting program.

You should get:

```
10 CLS
20 PRINT
30 PRINT
40 PRINT"THREE BLIND MICE"
50 PRINT
100 PRINT"SEE HOW THEY RUN"
```

RUN the program to get the message:

```
THREE BLIND MICE

SEE HOW THEY RUN
```

DEMO1A has been joined on the the program in memory.

Caution. When joining two programs together make sure that none of the lines in the second program has the same line number as any of the lines in the first program. Otherwise the resulting program will not work correctly.

FINDING PROGRAMS

When you save a program on to tape, always write down the name of the program, the tape counter reading at which it starts, and whether it was recorded at normal or slow speed. This information can be kept with the cassette. Most tape cassettes come with a card on which contents can be recorded.

If you do not know where a program is on a tape, but do know its name, then the CLOAD command with the name specified will cause the computer to ignore all other programs and only load the program of that name.

For example, suppose you know a program called HIDEANDSEEK is somewhere on a tape and that this program was recorded at 300 baud. To find this program you would rewind the tape to the start, key in:

```
CLOAD"HIDEANDSEEK",S
```

and press RETURN. If you then play the tape the computer will ignore all the programs until it comes to the one called HIDEANDSEEK, which it then loads into memory. The V1.1 machine will indicate at the top of the screen anything else (e.g. other programs) found on tape while it is searching for the named program.

If, on the other hand, you know where a program is, but have forgotten its name then the command

```
CLOAD"" (or CLOAD"",S)
```

will cause the computer to load the first program it comes to on the tape into computer memory. Note there is no space between the inverted commas.

AUTOMATIC START

When you loaded DEMO1 into the computer, you had to type in RUN, followed by RETURN, to start the program. It could be convenient to have the program start as soon as it loaded. To do this we add 'AUTO' to the CSAVE command when saving the program.

If DEMO1 is not in computer memory then load it in.

Save it again, this time using the command:

CSAVE"DEMO1",S,AUTO

if you are saving at 3ØØ baud and –

CSAVE"DEMO1",AUTO

if you are saving at 24ØØ baud.

Reload the program in the normal way. It should now start as soon as it is loaded.

SAVING MEMORY BLOCKS

We have discussed saving programs held in what we have described as 'program memory'. The ORIC has different areas of memory which it uses to store various types of information. For example there are parts of the memory which control what the computer puts on to the screen, while other parts may be used for programs or data.

It is possible to save the information held in specified sections of memory on to tape and to load this information back into these sections later. We will look at this further in Chapter 14.

SAVING LONG PROGRAMS

There are few things more annoying than keying in a long program – say from a magazine – and losing it due to a power failure or because the computer will not save it. ORIC hasn't done this to me yet, but most other computers have! The way to avoid the horrible sinking feeling of seeing two hours' work disappearing into thin air, is to save the program every (say) twenty lines, so that if the mains supply 'hiccups' at line 2ØØØ, you have lines 1Ø to 18ØØ safely on tape.

Unless you have shares in the company which makes it, you won't want to use fresh tape each time you save the same program. Normally you would rewind the tape each time and use the same section of tape for each save.

If you are the cautious type you will probably have visions of the power supply 'hiccup' occurring halfway through saving lines 1Ø to 2ØØØ, so that not only is the program lost but the previous save of lines 1Ø to 18ØØ is also obliterated. To avoid this – and it can and does happen – we use the 'back up' principle.

This involves saving lines 1Ø to 2ØØ on one part of the tape, lines 1Ø to 4ØØ on another, lines 1Ø to 6ØØ on the first section (overwriting lines 1Ø to 2ØØ), lines 1Ø to 8ØØ on the second (overwriting lines 1Ø to 4ØØ) and so on. In this way, even if a disaster does occur during a save operation, you never lose more than 2Ø lines of code.

MORE DIVERSIONS

You know how to use the keyboard and how to save and load programs. The best way to make sure you remember what you have learned is to get in a bit of practice. Here, therefore, are two programs for you to key in, save and RUN.

As before I shall ask you to take these on trust, and treat them just as a bit of fun.

DEMO2

This program demonstrates ORIC's colour capabilities. I suggest you name it DEMO2 when you are saving it.

RUN the program by typing in RUN followed by RETURN. Break with CTRL C. The program 'hides' the cursor to improve the display. You can get it back with CTRL Q, after you have broken the program.

```
1Ø CLS:PAPER1
2Ø FOR N=Ø TO 7
3Ø PLOT8,5+N,23-N
4Ø PLOT28,5+N,17
5Ø READ A$
6Ø PLOT1Ø,5+N,A$
7Ø PLOT9,5+N,N
8Ø NEXT
9Ø PRINT CHR$(17)
1ØØ GOTO1ØØ
11Ø DATA BLACK ON WHITE
12Ø DATA RED ON CYAN
13Ø DATA GREEN ON MAGENTA
14Ø DATA YELLOW ON BLUE
15Ø DATA BLUE ON YELLOW
16Ø DATA MAGENTA ON GREEN
17Ø DATA CYAN ON RED
18Ø DATA WHITE ON BLACK
```

DEMO3

I suggest you name the next program DEMO3. I find this program very useful as I have a six year old son who uses it as homework practice. Break from the program with CTRL C, get back to single print with CTRL D, and restore the cursor with CTRL Q.

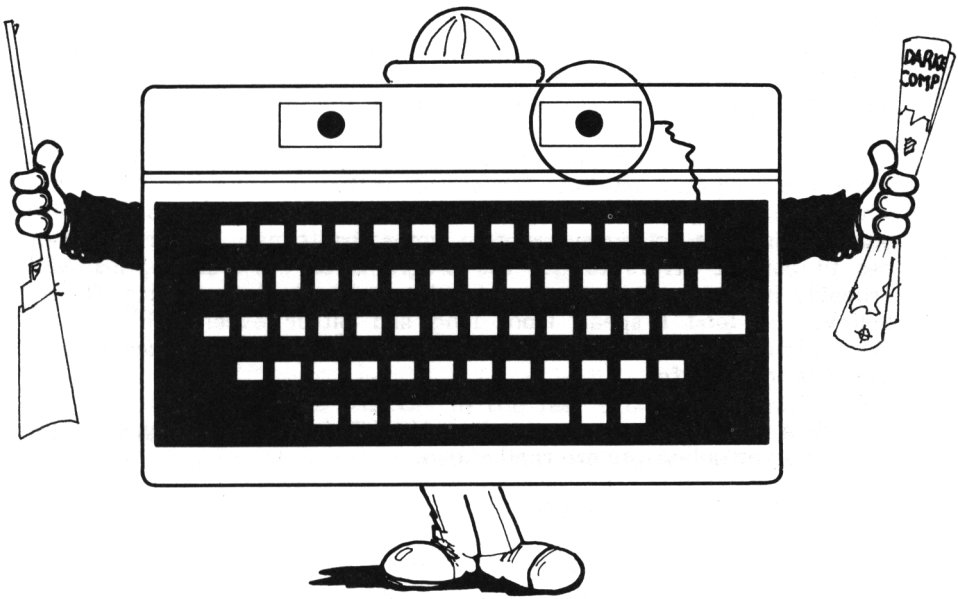
When you are keying in the program you will find that lines 8Ø, 1ØØ

11Ø and 12Ø take up two lines on the screen. This does not affect the program.

```
1Ø PRINT CHR$(17)
2Ø X%=RND(1)*11+1Ø
3Ø Y%=RND(1)*9+1
4Ø PAPER Ø:INK 3
5Ø PRINT CHR$(12)
6Ø FOR N=Ø TO 7
7Ø PRINT:NEXT
8Ø PRINT CHR$(4);CHR$(27);"J";SPC(13);X%-Y%;"+ ";Y%;"= ";
9Ø INPUT A$
1ØØ IF VAL(A$)=X% THEN 11Ø ELSE EXPLODE:
       PRINT CHR$(4):GOTO 5Ø
11Ø PRINT:PRINT:PRINT:PRINT CHR$(27);"J";SPC(18);"CORRECT"
12Ø FOR N=Ø TO 9:ZAP:WAIT 5:NEXT:PRINT CHR$(4):RUN 2Ø
```

CHAPTER 5

Exploring ORIC



GENERAL FEATURES

In this chapter I am going to describe some of ORIC's features of operation and some very useful instructions, which you will see in many computer programs. I shall keep my explanations brief at this stage and shall give more details of some of these features and instructions at the appropriate stage.

Type in NEW and press RETURN to clear program memory. If you wish you may leave the computer with its present background and foreground colours. If you prefer to work with black ink on a white background then type in INK0:PAPER 7 and press RETURN.

REM

The REM instruction causes the computer to ignore everything (even instructions) which follows it until the end of the line is reached.

You may wonder what possible use there could be for an instruction which the computer ignores. The answer is that it is not just computers which read programs – people read them too! It may not astonish you to learn that not every program works first time. The programmer has to correct errors to get the programs to work at all, and will probably return to the program at a later date to improve it. Please believe me (and I speak from long and bitter experience), this is much easier if REM statements are used to indicate what each section of the program is for.

This is a rather extreme example. You do not need to type it into the computer unless you are really keen.

```
10 REM THIS ILLUSTRATES THE USE OF
   THE REM STATEMENT.  I HAVE FOUND
   THAT I CAN TYPE IN COMMANDS LIKE
   ZAP, PING, EXPLODE, PRINT 1 AND
   SO ON AFTER REM BUT THE COMPUTER
   IGNORES THEM.  IF I WANT AN EXPLOSION
   I HAVE TO GO TO LINE 20 AND TYPE-
20 EXPLODE
```

The REM statement may be put in the middle of a line after an instruction provided it is preceded by a colon (:).

For example:

```
10 PING:REM THIS MAKES A NOISE
```

If you are using the REM statement in the middle of a line you can replace all of :REM by an apostrope (').

For example:

```
10 PING'THIS MAKES A NOISE
```

You can use this abbreviation only in the middle of a line, after an instruction. You can't use it at the start. For example:

```
10 'THIS GENERATES AN ERROR
```

does just that!

Take care – remember that everything after REM is ignored until you reach the end of a line.

For example:

```
10 PING:REM A NOISE:ZAP:REM ANOTHER NOISE
```

will only 'ping'. It will not 'zap' because everything after the first REM is ignored.

COLON (:)

We saw the use of the COLON in the last example. It allows you to put more than one instruction on the same line. This is especially useful in IF....THEN decision statements, which we will look at in Chapter 8.

STOP

Clear the program memory with NEW and clear the screen. Type in this program:

```
10 REM A DEMONSTRATION PROGRAM
20 PRINT 1
30 PRINT 2
40 PRINT 3
50 PRINT 4
```

RUN the program. Once it is complete type in:

```
35 STOP
```

and press the RETURN key. Clear the screen and RUN the program again. Note that now only the first two numbers are printed. The STOP command stops the program.

CONT

Type in CONT, and press RETURN.

The program will continue from line 40 on to the end.

CONT is used to restart the program after it has met a STOP command. The program will restart at the first instruction after the STOP command. CONT may be used only as an immediate command. Do not use it in any program.

GOTO

GOTO in a program, followed by a line number, makes the computer go to that line and do what it says there.

Add this line to the previous program (don't forget to press RETURN after you have typed in the line).

```
60 GOTO 20
```

Clear the screen and RUN the program. As before it will print the first two numbers and STOP. Enter CONT (ie type in CONT and press RETURN). The program will print the next two numbers. Then it will go back to the start and print the first two numbers again.

The effect of the GOTO statement is to take the program back to the beginning after line 50 has been executed (carried out), so that lines 20 and 30 are executed again.

GOTO can be used as an immediate command rather than an instruction. This can be useful if you wish to start a program part way through. In the demonstration program given, try the effect of GOTO 30, GOTO 40, or GOTO 50 instead of RUN.

Note – You can also use RUN 30, RUN 40 or RUN 50 instead of GOTO 30 etc in this case.

INFINITE LOOPS

Delete line 35 by typing in 35 and pressing RETURN.

Clear the screen and LIST the program. You should get:

```
1Ø REM A DEMONSTRATION PROGRAM
2Ø PRINT 1
3Ø PRINT 2
4Ø PRINT 3
5Ø PRINT 4
6Ø GOTO 2Ø
```

Study this program carefully. You will see that every time it gets to line 6Ø it will 'loop back' to line 2Ø. Thus the program will execute the instructions in lines 2Ø to 6Ø repeatedly. This is known as an infinite loop.

RUN this program to see the effect of the infinite loop. Stop the program with CTRL C.

SCROLLING

The infinite loop caused the screen display to 'scroll'. This means that when the bottom line of the screen is reached, the next print on the screen will cause all the previous printed lines to move up one space. The printing on the top line disappears from the screen as a result of this.

Another example of scrolling occurs when we list a program which is too long to fit on the screen all at once.

Clear program memory with NEW and clear the screen. Type in this program:

```
10 REM THIS IS A SILLY PROGRAM
20 REM IT DOES NOTHING AT ALL
30 REM
40 REM
50 REM
60 REM
70 REM
80 REM
90 REM
100 REM
110 REM
120 REM
130 REM
140 REM
150 REM
160 REM
170 REM
180 REM
190 REM
200 REM
210 REM
220 REM
230 REM
240 REM
250 REM
260 REM
270 REM
280 REM
290 REM
300 REM
```

You will see that as you are entering the final few lines of this program the lines at the start are 'lost' from the top of the screen. Don't worry, the machine still has them in memory. The reason that the top lines are lost from view is that the machine can display only 26 lines of print on the screen at any one time.

When you LIST the program it will again scroll up the screen and the top lines will be lost. In fact in this case the situation is even worse. Because the computer prints a line space and then the word 'Ready' on the bottom two lines of the screen two more lines are lost from the top.

To stop the scrolling so that you can look at the program listing, press the space bar. To start the listing scrolling again press any key (except SHIFT or CTRL).

If you wish you can list a single line on the screen. For example clear the screen and enter:

```
LIST 5Ø
```

Line 5Ø should be printed on the screen.

Possibly more useful is the ability to list a section of the program – say from lines 2Ø to 6Ø. To do this clear the screen and enter:

```
LIST 2Ø - 6Ø
```

CLS

We saw how to clear the screen by CTRL L. There is also an instruction to do the same thing. Type in CLS and press RETURN. CLS stands for 'Clear Screen'. Unlike CTRL L, CLS can be used in programs.

FOR...NEXT

The **FOR...NEXT** loop, which will be covered more fully in Chapter 8, lets us repeat a process any number of times, with a different value at each stage. It is a quick way of checking the effect of different values without having to rewrite the program each time. The two lines

```
10 FOR N=0 TO 3

100 NEXT N
```

would automatically repeat four times the program lying between lines 10 and 100 setting the value of N to 0, 1, 2 and 3 successively at each point in the program where N is mentioned. 'NEXT N' in line 100 may be abbreviated to 'NEXT'.

LET

The **LET** instruction is used to set one thing equal to another. When you are writing programs you will sometimes be using what are called **'variables'.** As the name suggests these are quantities which vary during the program. These variables are usually represented by letters.

At the start of the program a variable has to set to its 'initial value' – that is, the number it starts at (if this is not zero). This is one of the uses of the **LET** instruction.

For example:

```
LET A=1
LET HEIGHT=6
```

LET can also be used to set up a counter or to add two variables together. For example:

```
LET B=B+1
LET C=N+M
```

Variables will be dealt with more fully in Chapters 8 and 9; so don't worry if you find this a little puzzling at this stage.

'LET' can be missed out – and normally is – in the assignment statement, so that instead of typing in 'LET A=6' we can type in 'A=6'.

WAIT

Another useful instruction is WAIT, followed by a number. This holds up the action for a time 'proportional to' that number, with WAIT 1ØØ giving a delay of about one second, WAIT 5Ø giving a delay of half a second and so on.

INK and PAPER

You have used these instructions previously and I hope they are fairly self explanatory. INK, followed by a whole number between Ø and 7 (inclusive) sets the foreground or 'ink' colour of text on the screen. PAPER, followed by a whole number between Ø and 7 (inclusive) sets the background or 'paper' colour.

The numbers corresponding to each colour, for both INK and PAPER, are:

Number	Colour
Ø	black
1	red
2	green
3	yellow
4	blue
5	magenta (purple)
6	cyan (light blue)
7	white

INK and PAPER are 'global' commands. If you specify a new INK colour, the colour of all the text on the screen changes. If you specify a new PAPER colour the entire background changes to that colour.

Take care not to make INK and PAPER the same colour. If you do you won't see what is printed on the screen.

Clear program memory with NEW and type in this program:

```
1Ø FOR N=Ø TO 7
2Ø CLS
3Ø PAPER N
4Ø INK 7-N
5Ø PRINT"PAPER "N,"INK "7-N
6Ø WAIT 2ØØ
7Ø NEXT
```

You don't have to type in spaces in this program. PAPERN, WAIT2ØØ and FORN=ØTO7 work just as well. The spaces (I hope) make the program easier to read and understand.

RUN the program. It demonstrates eight different combinations of INK and PAPER colours. More combinations are available. Experiment until you find one that you particularly like. The PRINT instruction in line 5Ø will be explained in the next chapter.

You can now use the keyboard, enter, edit, list, save and load programs, change ink and paper colour, use some of the more common machine instructions. These last were chosen to provide you with 'tools' to explore and understand the more advanced features of the machine.

CHAPTER 6

The ORIC Display

PRINT

The PRINT instruction is used to 'print' characters (letters and numbers) on the television screen and not (as you might logically expect) on to printer paper.

On its own, PRINT starts the printing on the screen on the first free line. It can be used as a direct command or as an instruction in a program.

PRINTING NUMBERS

Printing numbers is very straightforward. For example

```
PRINT 5
```

does just that (provided we press RETURN). The number appears at the start of the first free line – i.e. the first line from the top of the screen which does not already have print on it.

Now try

```
?6
```

The number 6 should appear at the start of the next line.

The question mark (?) is used as an abbreviation for PRINT. As PRINT is used a great deal in programs this abbreviation should save you a lot of typing.

Here's an interesting feature – clear program memory and enter:

```
1Ø?5
```

LIST this program and you will get:

```
1Ø PRINT5
```

ORIC lets you use the abbreviation when you are typing in programs, but gives you the unabbreviated instruction when LISTing these programs – a good example of 'user friendliness'.

I shall use PRINT rather than ? throughout this book for the sake of clarity.

You may also have noticed that I didn't put a space between PRINT and 5 in the last example, whereas I did in the one before it. The space is optional – PRINT 5 does the same as PRINT5. Sometimes inserting spaces as in the first of these examples makes your program listings clearer.

What if we want to print a number somewhere in the middle of a line? Try

```
PRINT      6
```

that is, pressing the space key a number of times before the 6. After entering it you will find the 6 printed exactly where it would have been without the spaces! This is because the PRINT command is designed to look for the first number and ignores the spaces. We will see how to deal with this shortly.

PRINTING CHARACTERS

Characters are a different matter. For example if we enter

```
PRINT A
```

The computer prints a Ø at the start of the first free line.

This means that the computer is looking for a number and not a letter. It thinks the 'A' is a variable, and because it has not seen it before, assumes it has value zero.

The trick is to put any characters we want printed inside quotes i.e.:

```
PRINT "A"
```

The 'A' should appear, again at the start of the first empty line.

STRINGS

Characters or numbers within quotes are called STRINGS. We will learn more about them in Chapter 9.

Experiment with different messages inside the quotes. Clear the screen every so often to get rid of accumulated rubbish.

```
PRINT "A SCREEN MESSAGE"

PRINT "    LOTS OF SPACES    "
```

By using spaces we can move a message to any point on a line. The message can include numbers but not as part of calculations. Try the effect of:

```
PRINT "5678"
```

5678 is placed on the screen in exactly the same place as if "ABCD" had been the message. To see the different effects clearly try

```
PRINT 5+6
```

and `PRINT "5+6"`

Without quotes the program works out and displays the result. With quotes it displays exactly what is inside the quotes, neither more nor less. What we do instinctively, ORIC has to be programmed to do. The quotes around anything, whether numbers or letters, tell the micro that they are just a message or label equivalent to a house number or name to be printed exactly 'as is'. We can include as many spaces in such a message as we like: a space is treated in the same way as a letter or a number if inside quotes.

PRINT SEPARATORS

We can also use spaces within quotes to place the result of a calculation where we want it on the screen. We could try:

```
PRINT "    "
```

followed by

```
PRINT 5+6
```

But the separate PRINT commands just put the spaces and the result on successive lines.

SEMICOLON

Now try:

```
PRINT "   ";5+6
```

The semicolon is one of the so called print separators and here it tells the micro that the two items are to follow each other directly. In general a semicolon indicates that we have more than one item following PRINT, and that the items are printed consecutively (one after the other).

To save you time and memory space, ORIC makes the use of the semicolon print separator optional. Where two items are on the same program line ORIC will assume a semicolon between them.

The exception to this is when one number follows another. ORIC then requires semicolons to indicate where one number stops and the next one starts.

For example, compare:

```
PRINT 1;2;3
```

and

```
PRINT 123
```

When numbers are printed on the screen a space is left after the number (before and after the number in the V1.1 machine). This is not, however, the case with strings.

For example try:

```
PRINT "1""2""3"1;2;3;"123"
PRINT "al";"together"
```

If you wish to put spaces between strings then you must include the spaces in the strings, or use separate strings containing the spaces.

If you have a number followed by a string then you don't have to put a space at the start of the string, as there will be a space following the number. If, on the other hand, you have a string followed by a number you will require a space at the end of the string in the V1.0 machine. The V1.1 machine puts spaces both before and after numbers.

Try a few examples to get used to the idea:

```
PRINT "ORIC";" ";1
PRINT 1Ø" GREEN BOTTLES"
PRINT "Please leave ";3;"pints"
(in V1.1 machine  PRINT "Please leave";3;"pints")
PRINT "5+6=";5+6
```

You will see from the third example that ORIC accepts lower case (small) letters in strings, although it does not accept them as commands (or as variables). CTRL T removes the capitals lock and lets you type in lower case letters.

Let us try a letter standing for a number. As we have seen, a letter (or group of letters) set equal to a number is called a 'variable'. Try:

```
A=1Ø:PRINT A"Green bottles"
A=1Ø:PRINT "A Green bottles"
```

The first line should have the intended result

```
1Ø Green bottles
```

and the second will be the silly statement

```
A Green bottles
```

This confirms that the 'A' in quotes is treated just as a letter to be printed and doesn't stand for a number.

The semicolon may also be used at the end of a PRINT statement. This inhibits the line feed so that the next character printed is at the first free space in the current line on the screen instead of the start of the next line down.

This sounds complicated, but an example should make it clearer.

Clear the program memory and type in

```
1Ø CLS
2Ø PRINT: PRINT: PRINT: PRINT
3Ø PRINT"COUNTER READS ";
4Ø FOR N=1 TO 9
5Ø PRINT N;
6Ø WAIT 1ØØ
7Ø NEXT
```

RUN this program, with and without the semicolons at the end of lines 3Ø and 5Ø. If you have the V1.1 machine you do not need a space as the last character of the string in line 3Ø.

Line 2Ø in this program is of interest. The instruction **PRINT** all by itself does not print anything on the screen, but moves the start of whatever is printed next down one line. To demonstrate this try altering the number of **PRINT** instructions in line 2Ø – rembering to separate them with colons. Try the effect of deleting line 2Ø.

COMMA

The comma (,) may be used as a print separator.

The effect of this print separator depends on the version of the machine you have.

In the V1.Ø machine

The comma inserts three spaces between the items printed.

Remember, however, that when a number is printed a space is inserted by the comma, this gives a total of four spaces printed between numbers separated by commas.

Strings don't have extra spaces inserted after them; so a comma puts three spaces between strings.

The number of spaces between items is not affected by the length of the item.

If several commas are used together, each additional comma adds another two spaces.

In the V1.1 machine

The comma divides the screen into columns, each eight characters wide. Characters are not normally printed in the first two horizontal positions (Ø and 1), so that the comma separator starts items at positions 2, 1Ø, 18 etc.

Numbers have spaces in front of them. If you use the comma separator with numbers, then the leading space before each number is printed at the start of the corresponding column. Thus the numbers themselves start at positions 3, 11, 19 etc.

The number of spaces between items is affected by the number of characters in the items, as each item will start at the first free column. If an item is more than eight characters wide (including leading and trailing spaces for numeric items) then it will take up two or more columns. If several commas are used together each additional comma causes a full column width of spaces.

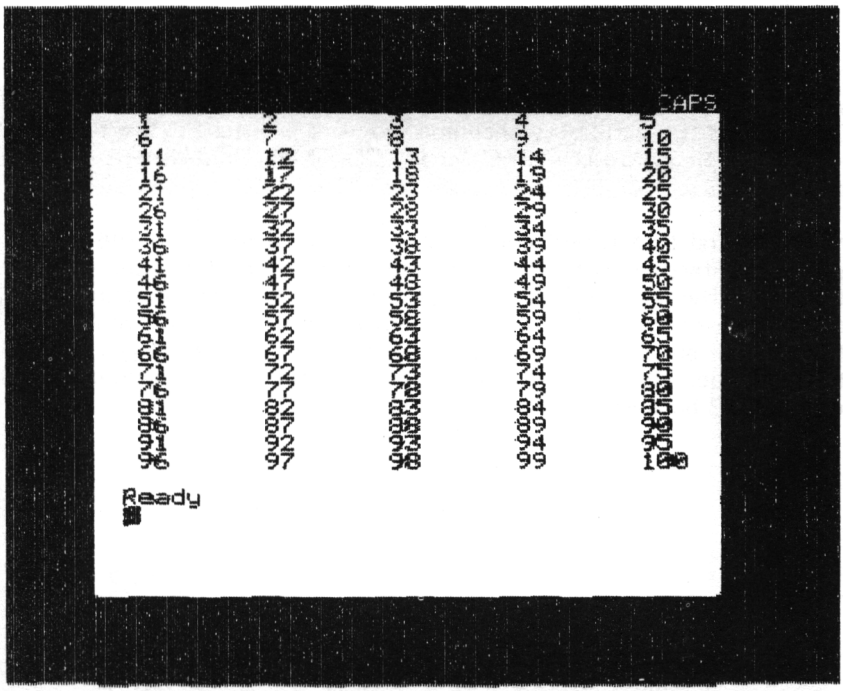
Whatever version of machine you have, try these examples:

```
PRINT 1,2,3
PRINT "A","B","C"
PRINT "A",1,"B"
PRINT 1684,3,18
PRINT"HANDS","KNEES","AND","BOOMPSA","DAISY"
PRINT 1,,2,,3
```

The comma, like the semicolon, may be placed at the end of a PRINT statement.

Clear program memory, type in and RUN:

```
1Ø CLS
2Ø FOR N=1 TO 1ØØ
3Ø PRINT N,
4Ø NEXT
```



The output obtained from this program on the V1.1 machine is shown. You can RUN the program on the V1.Ø machine but you will not obtain the neat 'tabulated' output.

While the comma separator in the V1.1 machine can be used to put items into columns it does not give us control of where these columns are placed on the machine. The TAB separator – **which works correctly only on V1.1 machines** – is rather more flexible.

PRINT TAB (n).....

causes the items following TAB(n), whether words or numbers, to start printing n spaces along the line. It is used to create tables of information in columns. We can use several TAB statements on one line provided each value of n is greater than the one before – the program calculates how much farther it has to move to get to the next TAB value. As an example:

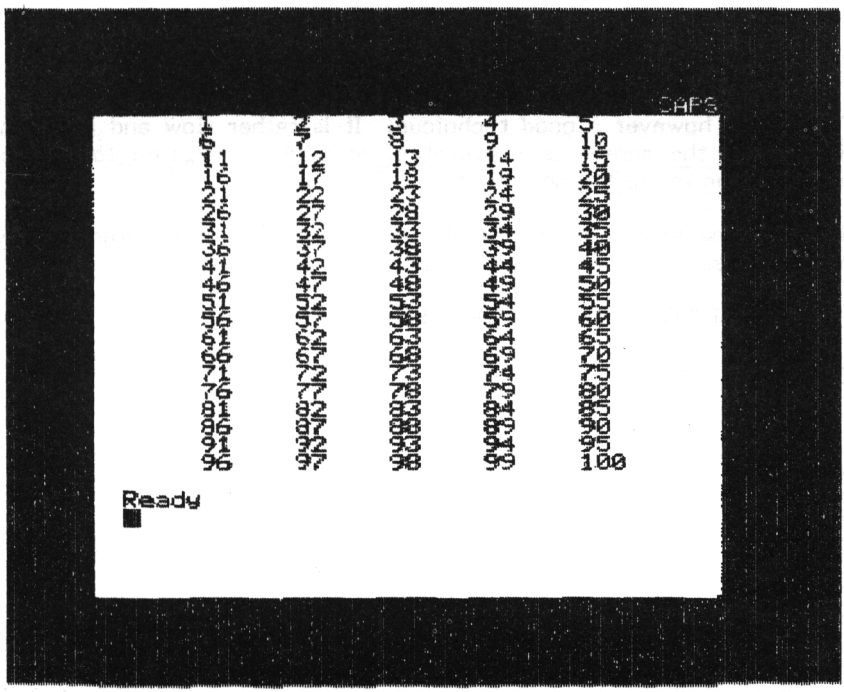
PRINT TAB(5)"FIRST"TAB(15)"SECOND"

will print the word FIRST starting at the 5th character position and SECOND at the 15th. Try changing the print positions to see the effects – for example try to print the word SECOND starting at position 11 and then at position 1Ø.

You will find that it will only print on the original line if the new TAB position is farther along than the last character printed, i.e. with five letters in the word FIRST the letter 'T' is printed at position 1Ø. To attempt to print another word (or number) starting at 1Ø would corrupt the original and so the word SECOND is printed instead at position 1Ø on the next line. Obviously we should do our TAB's in increasing order – we cannot TAB backwards.

Clear program memory and type in this program. Compare the output obtained with that from the previous program. Don't worry about 'STEP' in line 2Ø. We will come to this in Chapter 8.

```
1Ø CLS
2Ø FOR N=1 TO 1ØØ STEP 5
3Ø PRINT TAB(6)N;TAB(12)N+1;TAB(18)N+2
   TAB(24)N+3;TAB(3Ø)N+4
4Ø NEXT
```



SPC

SPC is a useful separator

```
PRINT SPC(n) ...
```

will insert n spaces before the item to be printed.

n must be a positive whole number not greater than 255.

You can place a message anywhere on the screen using SPC. For example, try:

```
CLS:PRINT SPC(255)SPC(249)"MIDDLE OF SCREEN"
```

This is not however a good technique. It is rather slow and any print already on the screen is obliterated, as CLS is required to get the cursor to the top left hand corner.

SPC may be more usefully employed between two print items. Try, for example:

```
PRINT "TEN"SPC(1Ø)"SPACES"
```

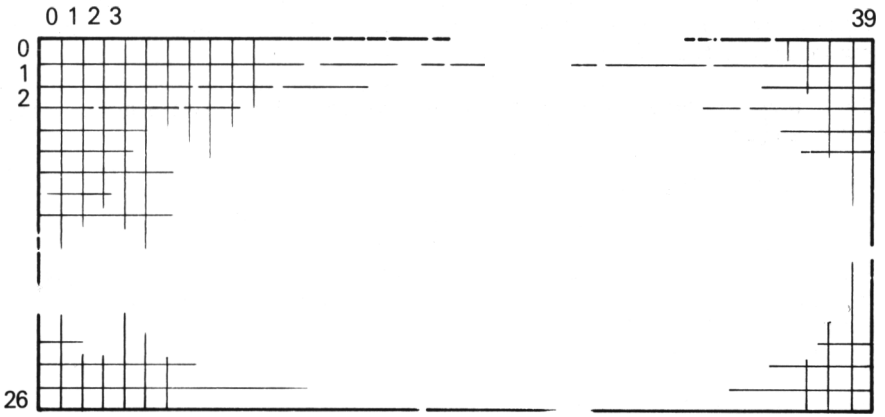
PRINT @

PRINT @ allows you to specify the position on the screen where a print message is to start in terms of the screen column (Ø to 39) and screen row (Ø to 25). **It is implemented only on the V1.1 machine.**

The format is:

```
PRINT @ X,Y;.....
```

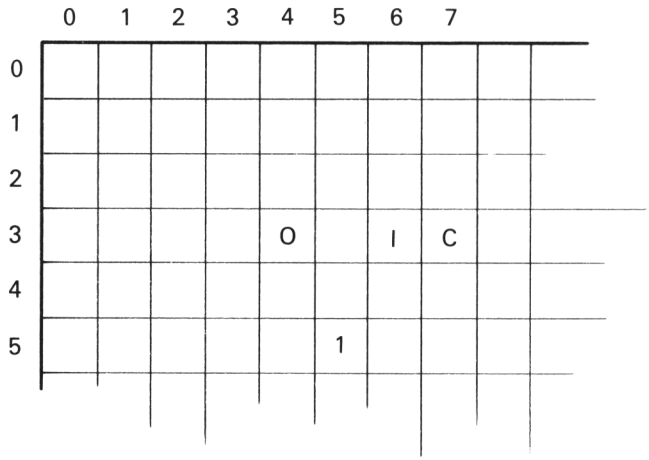
where X specifies column and Y specifies row as shown below.



Thus:

```
PRINT @ 4,3;"ORIC"
PRINT @ 5,5;1
```

would give



The space between PRINT and @ is optional, as is the space between @ and the column number. The semicolon separating the row number and the first print item is compulsory. If you don't put it in you will get a syntax error.

You can use the PRINT @ command to print on columns Ø and 1 provided that you specify black paper and white ink. Any other paper/ink combination will give a very odd effect and could result in part of your message being lost. It is safer to restrict your printing to columns 2 to 39.

We shall see why when we look at attributes in the next chapter.

We can use several types of separator in the same print statement.

Try:

```
PRINT SPC(5)1,2;"BUCKLE MY SHOE"
PRINT @ 6,7;"HELLO"TAB(2Ø)"DOLLY"
```

(The second example will work on the V1.1 machine only.)

ARITHMETIC

The computer is used to process information – we have been looking mainly at words, and where they are to be printed. We have also slipped in the occasional sum. For example:

```
PRINT 5+6
```

gave the answer 11.

We can do addition, subtraction, multiplication and division provided we get used to some unfamiliar symbols. The keyboard has the + and – signs but no conventional x and ÷ . Instead we use * and / so that 3 plus 4 multiplied by 5 divided by 6 is written

```
3+4*5/6.
```

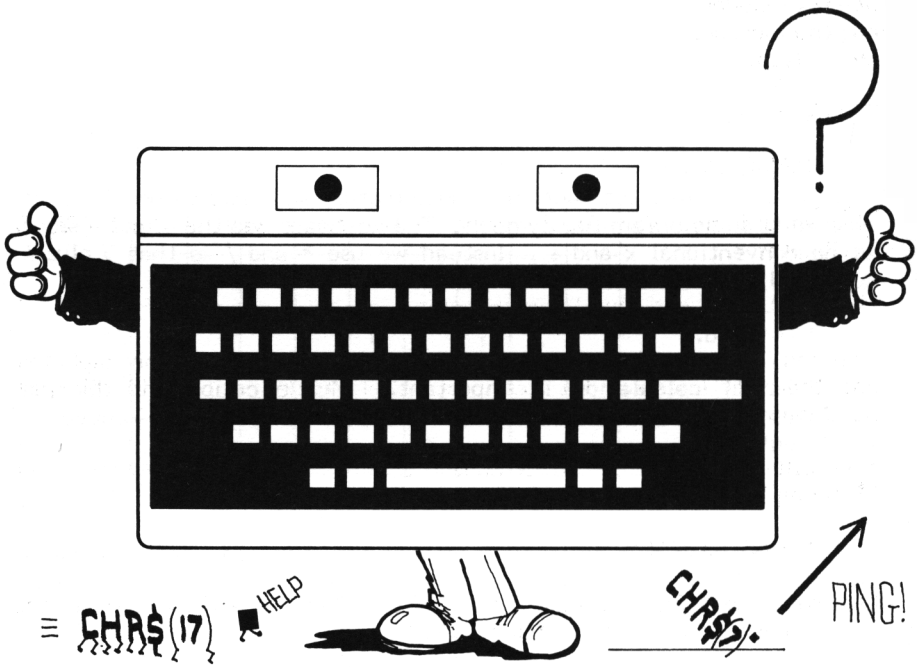
The order of calculation is important. Multiplication and division are taken first, addition and subtraction second, so the above is worked out as

```
3+{(4*5)/6}
```

We can use these calculations, as well as the many advanced functions covered in Chapters 8 and 9, as part of almost any program line, including PRINT statements.

CHAPTER 7

Some Odd Characters



CHR$

Each character which you can type in from the keyboard and display on the screen has a number associated with it. These numbers are known as ASCII codes, and are listed in Appendix A.

CHR$ converts a number to the string containing the associated character. For example the ASCII code for A is 65. Thus:

PRINT CHR$(65) is the same as PRINT "A".

CONTROL CHARACTERS

If you look at Appendix A you will see that the ASCII codes start at 32. Codes less than 32 do not correspond with any characters on the screen. Such codes generate characters known as **control characters.**

For example, try:

```
PRINT CHR$(4)"DOUBLE"
```

This should give you double print in the same way as did CTRL D in Chapter 2.

Use CTRL D to get back to single print.

You may have noticed that D is the fourth letter in the alphabet and that CRTL D did the same as PRINT CHR$(4). Could this just be a coincidence? Let's find out.

You will recall that CTRL G caused a 'ping'. G is the seventh letter in the alphabet; so try –

```
PRINT CHR$(7)
```

One more test for the sceptics. CTRL Q hides the cursor. Q is the seventeenth letter in the alphabet; so try –

```
PRINT CHR$(17)
```

Use either CTRL Q or PRINT CHR$(17) to get the cursor back.

Here is a list of the control characters you may find useful in programs. Some peripheral devices (such as certain types of printer) use other control codes, which will be listed in the booklets which accompany these devices.

CHR$(1) Copies a character on the screen to the keyboard buffer

CHR$(4) Double print (on/off toggle)

CHR$(6) Keyclick (on/off toggle)

CHR$(7) Causes a ping

CHR$(8) Moves cursor one space to left

CHR$(9) Moves cursor one space to right

CHR$(10) Moves cursor down one line

CHR$(11) Moves cursor up one line

CHR$(12) Clears the screen

CHR$(13) Simulates a return key depression

CHR$(17) Hides the cursor (on/off toggle)

CHR$(19) Hides the screen output (on/off toggle)

CHR$(20) Controls capitals lock (on/off toggle)

CHR$(29) Allows printing in far left column (on/off toggle)

Control characters must be preceded by an instruction such as PRINT.

Some of the control characters are more useful than others. You will probably find that codes 4, 8 to 12 and 17 are the most common, but the others are available if you wish to use them.

ESCAPE

There is another control character which I have not put on the general list – it is so important that it merits a section all to itself.

This is CHR$(27) which simulates an ESC key depression, and hence enables us to put attributes on the screen under program control.

We saw in Chapter 2 how attributes could be used to generate some unusual (and rather spectacular) screen displays. CHR$(27) lets us do the same things, but now we can use programs to generate these displays automatically, change them, and move them about.

The character following CHR$(27) is not printed but instead determines the attribute, which affects all the characters following it until either the end of the line is reached or a second attribute is encountered which cancels the effect of the first.

When planning screen displays it is important to remember that attributes are printed as blank characters and each take up one character position on the screen.

Let's first list the effects of the various characters which could follow CHR$(27). We will then see some examples of these in action.

PRINT CHR$(27)"@" – black ink
" "A" – red ink
" "B" – green ink
" "C" – yellow ink
" "D" – blue ink
" "E" – magenta ink
" "F" – cyan ink
" "G" -white ink
" "H" – single height, steady, standard
" "I" – single height, steady, alternate
" "J" – double height, steady, standard
" "K" – double height, steady, alternate
" "L" – single height, flashing, standard
" "M" – single height, flashing, alternate
" "N" – double height, flashing, standard
" "O" – double height, flashing, alternate
" "P" – black paper
" "Q" – red paper
" "R" – green paper
" "S" – yellow paper
" "T" – blue paper
" "U" – magenta paper
" "V" – cyan paper
" "W" – white paper

First try a few examples to get used to the idea:

```
PRINT CHR$(27)"A"CHR$(27)"T"SPC(12)"RED ON BLUE"
PRINT CHR$(27)"E"CHR$(27)"LTHIS IS A FLASHY EXAMPLE"
PRINT CHR$(27)"C"CHR$(27)"ISOME VERY ODD CHARACTERS"
```

Make up a few more of these for yourself. Clear the screen occasionally so that you don't get lost.

A program, which can be repeated, is usually of more use than an immediate command. So try:

```
1Ø PAPER Ø:INK 3'BLACK PAPER YELLOW INK
2Ø PRINTCHR$(12)CHR$(4)CHR$(17)
3Ø REM CLEAR SCREEN, DOUBLE PRINT, HIDE CURSOR
4Ø PRINTSPC(9)CHR$(27)"JBIG YELLOW LETTERS"
5Ø PRINT:PRINT:PRINT
6Ø PRINTSPC(1Ø)CHR$(27)"T"CHR$(27)
  "JON A BLUE LABEL "CHR$(27)"P"
7Ø PRINT:PRINT:PRINT
8Ø PRINTSPC(9)CHR$(27)"A"CHR$(27)"V"CHR$(27)
  "JALSO RED ON CYAN "CHR$(27)"P"
9Ø GOTO 9Ø
```

This program has a number of interesting points.

Firstly I have set the paper colour to black. If you do not specify a background colour when controlling a display with attributes, then your background for that line will go black by default. Setting the general background to black saves code and usually gives neater displays. Try changing line 1Ø to PAPER 7:INK 3, and you should see what I mean.

Because I have specified a 'global' ink colour of yellow all printing will be in yellow unless I change the ink colour on a specific line by using an attribute – as I have done in line 8Ø.

I have used PRINT CHR$(12) rather than CLS to clear the screen. This instruction not only clears the screen, but also moves the cursor down one line (from screen line zero to screen line 1). As we saw in chapter 2, we need to do this to get the double height print working correctly.

Program line 5Ø moves the cursor from screen line 1 to screen line 5 (skips lines 2,3 and 4). As line 5 is an odd numbered screen line double print will work correctly there. In general double size print lines must have double spaces (or multiples thereof) between them – or, alternatively, no spaces at all. Try the effect of altering lines 5Ø and 7Ø to each contain five PRINT instructions. Alter them again so that they each contain only one PRINT instruction.

Finally line 9Ø puts the program in a continuous loop so that the word 'Ready' does not appear to spoil the display. Break from this loop with CTRL C. Don't forget to restore the cursor (CTRL Q) and cancel double print (CTRL D) after breaking from the program.

Play around with this program, or use it as a model to write similar ones for yourself.

Entering PAPER 7:INK Ø:CLS will clear the screen and get back to black ink on white paper.

Here is another program which you may find interesting. Try to work out exactly what is happening.

```
1Ø PAPER Ø:INK 2
2Ø PRINTCHR$(12)CHR$(17)
3Ø FOR N=1 TO 21
4Ø PRINTCHR$(4)SPC(N)CHR$(27)"K N S£ S£ S£ S£ ,W [ ^ /^ "CHR$(1Ø)
5Ø PRINTCHR$(4)SPC(N+1)"o"SPC(3)"o"SPC(3)"o o o o";
   CHR$(11)CHR$(11)CHR$(11)
6Ø WAIT 1Ø
7Ø NEXT
8Ø PRINTCHR$(17)
```

PLOT

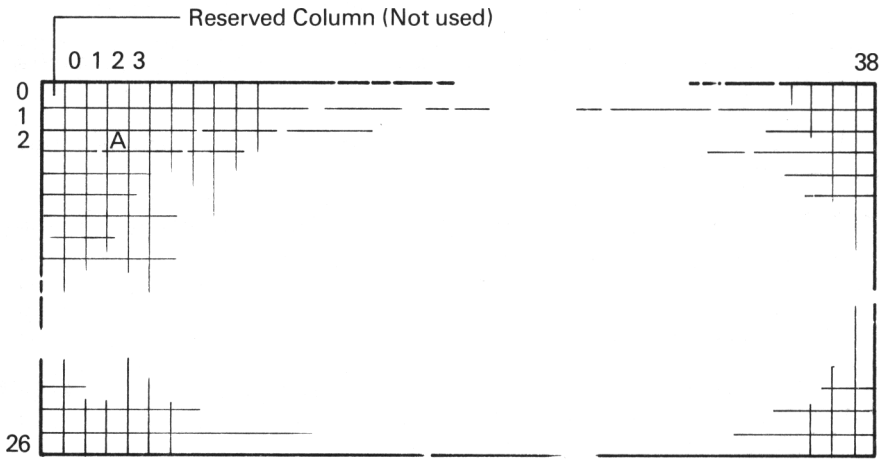
You can position a message anywhere on the screen using SPC or TAB with PRINT. However this could take a bit of working out. What is required here is an instruction which prints on the screen starting at a specified number of spaces along from the left of the screen and a specified number of lines down from the top.

If you have the V1.1 machine you can do this with PRINT @. The PLOT instruction, however, works with both versions of the machine. The format is:

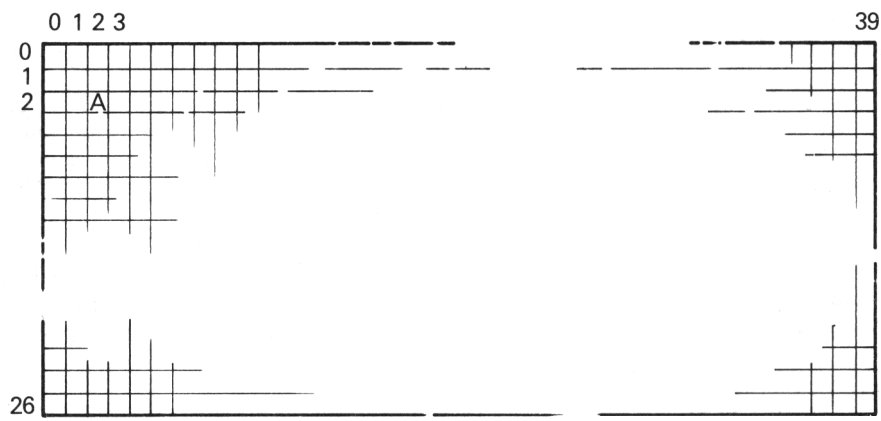
```
PLOTX,Y,"..."
```

where X is the number of spaces across the screen and Y is the number of lines down the screen.

The line and column numbers are slightly different for each version:



V1.0 Machine



V1.1 Machine

For example PLOT 2,2"A" would print the letter A in the positions shown.

PLOT is normally used only with strings. If you try:

```
PLOT 6,6,65
```

you will find that, instead of the number 65, the letter A is printed in screen position 6,6.

```
PLOT 2Ø,14,9Ø
```

will print the letter Z.

The PLOT instruction interprets the number following it as an ASCII code. Refer to Appendix A and try a few examples for yourself.

In the examples given I have not used the two left hand columns. Let's find out what happens if you do. Clear the screen and enter:

```
    PAPER 7:INK 4:PLOT Ø,Ø,"TOP LEFT" (V1.Ø machine)
or  PAPER 7:INK 4:PLOT 1,Ø,"TOP LEFT" (V1.1 machine)
```

Not very inspiring. Clear the screen again and enter

```
    PAPER Ø:PLOT Ø,Ø,"TOP LEFT" (V1.Ø machine)
or  PAPER Ø:PLOT 1,Ø,"TOP LEFT" (V1.1 machine)
```

TOP LEFT should now appear in white ink – although you have set the INK colour to blue.

Try changing the message so that all the letters on the screen are blue. Enter

```
INK 4
```

The message turns blue – but you have lost the first letter. It now reads OP LEFT. Messages placed on the screen by the PLOT command so that they start at column Ø appear in white ink. If you try to change the ink colour you lose the first letter of the message. This is because the second column from the left is then used to hold the attributes controlling the foreground (ink) colour of the screen display.

If you have the V1.1 machine you can also PLOT in the far left hand column. If you do this the entire row turns black and the message appears in white ink.

LORES Ø

Up until now you have been using the 'TEXT', mode for displaying screen messages. The machine goes into TEXT mode automatically when you switch it on or when you press the reset switch. There is, however, another mode which allows you to use column Ø for your messages, by setting the 'default' colours of black paper and white ink so column Ø is not required for attributes. This is Low Resolution Mode Ø or LORES Ø for short.

This mode is entered using the instruction LORES Ø either in a program or as a direct command.

LORES Ø is also the instruction which clears the screen in this mode. If you use CLS or CTRL L the machine goes back into TEXT MODE. If you use an INK instruction to change foreground colour you will lose all of the printing in column Ø. Attempting to change the background colour with a PAPER instruction has an odd effect. Try it and see.

The screen cannot be scrolled in LORES Ø mode. If you try to print below line 26 the TEXT screen scrolls back into view.

The PRINT instructions are used in the normal way in LORES Ø mode.

The instruction TEXT can be used to get back into text mode. This, however, does not clear the screen or restore normal paper colour. CLS does all of these.

PLOTTING ATTRIBUTES

We saw that attempting to PLOT a number on the machine resulted in that number being treated as an ASCII code. The smallest ASCII code number is 32 – what happens if we PLOT smaller numbers?

The answer is that PLOTting numbers less than 32 is yet another method of putting attributes on to the screen and a very convenient method in that it is easy to specify the position where you wish the attribute to be placed.

PLOT X,Y,Ø – black ink
" 1 – red ink
" 2 – green ink
" 3 – yellow ink
" 4 – blue ink
" 5 – magenta ink
" 6 – cyan ink
" 7 – white ink
" 8 – single height, steady, standard
" 9 – single height, steady, alternate
" 1Ø – double height, steady, standard
" 11 – double height, steady, alternate
" 12 – single height, flashing, standard
" 13 – single height, flashing, alternate
" 14 – double height, flashing, standard
" 15 – double height, flashing, alternate
" 16 – black paper
" 17 – red paper
" 18 – green paper
" 19 – yellow paper
" 2Ø – blue paper
" 21 – magenta paper
" 22 – cyan paper
" 23 – white paper
" 24-31 – may affect screen synchronization

I would advise against PLOTting numbers 24 to 31. Some of the effects produced may not be very good for your television.

The next program shows how attributes PLOTted on to the screen can affect a display. See if you can work out what is happening. Break with CTRL C and restore the cursor with CTRL Q.

```
1Ø PRINT CHR$(12)CHR$(17)
2Ø PAPER 7:INK Ø
3Ø PLOT5,11,"HERE'S YOUR NAME IN LIGHTS"
4Ø PLOT5,12,"HERE'S YOUR NAME IN LIGHTS"
5Ø WAIT 15Ø
6Ø PLOT2,11,1Ø:PLOT2,12,1Ø
7Ø WAIT 5Ø
8Ø FOR N=1 TO 6
9Ø PLOT11,11,N:PLOT16,11,7-N:PLOT21,11,Ø
1ØØ PLOT11,12,7-N:PLOT16,12,N:PLOT21,12,Ø
11Ø WAIT 5Ø
12Ø NEXT
13Ø PLOT11,11,Ø:PLOT11,12,Ø
14Ø PLOT16,11,Ø:PLOT16,12,Ø
15Ø WAIT 5Ø
16Ø FOR N=1 TO 6
17Ø PLOT11,11,16+N:PLOT16,11,23-N:PLOT21,11,23
18Ø PLOT11,12,23-N:PLOT16,12,16+N:PLOT21,12,23
19Ø WAIT 5Ø
2ØØ NEXT
21Ø PLOT11,11,23:PLOT11,12,23
22Ø PLOT16,11,23:PLOT16,12,23
23Ø GOTO 7Ø
```

ATTRIBUTES WITH PRINT @

If you have the V1.1 machine there is yet another method of putting attributes at selected positions on the screen. This is to use the PRINT @ instruction followed by CHR$(27).

Try:

```
CLS:PRINT @9,12;CHR$(27)"A"CHR$(27)"TRED ON BLUE"
```

INPUT

The INPUT instruction is used to get information from the keyboard, while the program is running.

Key in and RUN the following program:

```
1Ø INPUT"WHAT NUMBER";N
2Ø PRINT N,1Ø*N
3Ø GOTO 1Ø
```

Line 1Ø contains the INPUT instruction. This will print on the first free line on the screen whatever is in between the inverted commas. It will then wait for you to type in a number, followed by the RETURN key.

N may be positive or negative. It need not be a whole number. The INPUT instruction puts a space and a question mark after the message. The INPUT instruction works in the same way if you don't put a message after it. Then only the question mark 'prompt' appears on the screen. Try changing line 1Ø to:

```
1Ø INPUT N
```

If you want to make sure that only whole numbers (integers) are entered, put a percent(%) sign after the variable. Any number keyed in will be rounded down to the nearest integer value.

Try changing lines 1Ø and 2Ø to:

```
1Ø INPUT N%
2Ø PRINT N%
```

If you enter letters when ORIC is expecting a number, ORIC will ask you to re-enter the correct information by giving the error message REDO FROM START.

A second form of the INPUT instruction allows you to enter both letters and numbers. This form puts the inputted information between quotes and so turns it into a string (see earlier in this chapter). This happens whether the information entered is letters or numbers.

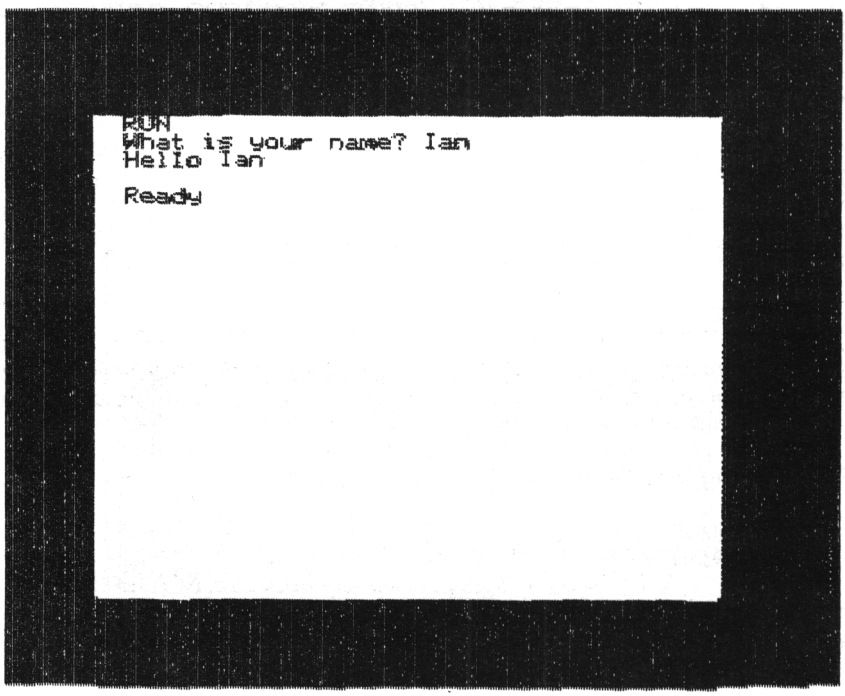
To input string information the format is

INPUT"....";A$** or simply **INPUT A$

Note the $ sign. This denotes a string.

The next program demonstrates the use of this format:

```
1Ø INPUT "What is your name";A$
2Ø PRINT "Hello "A$
```

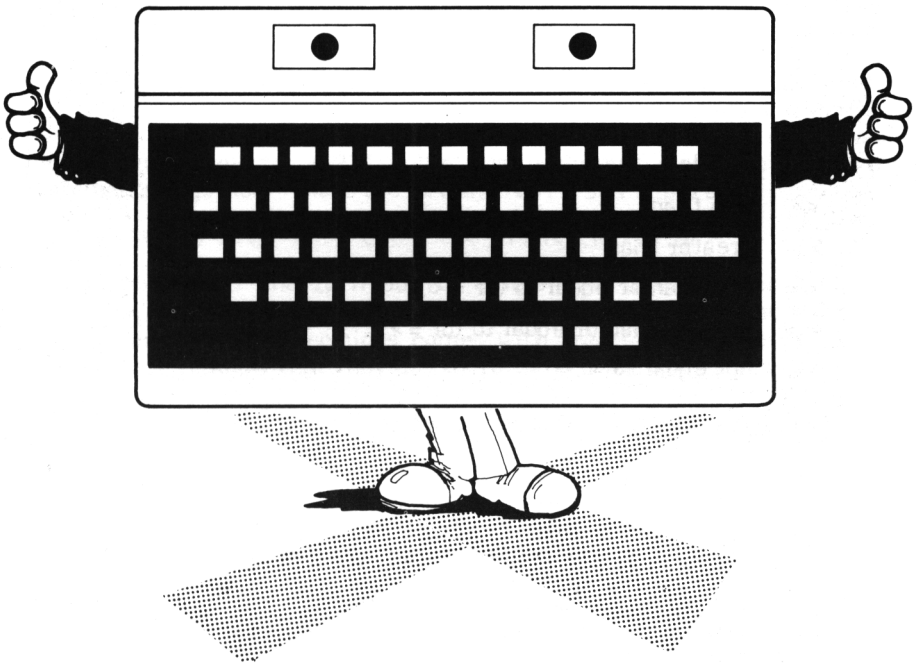


Remember that for both forms of the INPUT instruction you must press the RETURN key after your entry.

If you have a message in your INPUT statement you must have semicolon separator between the message and your INPUT variable. If you have no message you don't need a semicolon.

CHAPTER 8

It's Make Your Mind Up Time



DECISIONS

Most of the programs we have seen so far have just run through from start to finish. This is unusual as a computer is a very good decision maker. We can get the computer to decide whether to execute a command or not, or get it to do lines in different order depending on what has gone before. Some examples are more useful than words.

IF ... THEN

Let us consider the IF ... THEN ... command. This is used as it is written. IF something is true THEN do something. Here is an example.

```
IF X=5 THEN PRINT "X=5"
```

What comes after the THEN must be a command and what comes after the IF must be some sort of expression which is either true or false.

The ORIC understands a number of symbols in the IF statement and these include:

=	equals
<	less than
>	greater than
<=	less than or equal to (or =<)
>=	greater than or equal to (or =>)
<>	not equal to

This program shows some of these in action.

```
10 INPUT"WHAT NUMBER";N
20 IF N=5 THEN PRINT"YOU TYPED 5"
30 IF N<0 THEN PRINT"YOUR NUMBER IS NEGATIVE"
40 IF N<>8 THEN PRINT"THAT WASN'T 8"
50 IF N<=1 THEN GOTO 10
60 IF N>10 THEN STOP
70 GOTO 10
```

Try this program and see if you understand what is going on. Notice that you don't have to use PRINT statements. You can have GOTO or STOP or anything else in "IF" commands.

For example consider this silly program.

```
10 INPUT"WHAT NUMBER";N
20 IF N>5 THEN N=32
30 PRINT "YOU TYPED IN "N
40 GOTO 10
```

This program will give the 'correct' answer if your number is 5 or less.

AND, OR, NOT

Consider this program. It comments on ages less than 2, greater than 11Ø and between 13 and 19. Thus a number of decisions can be made all about the same variable.

```
1Ø INPUT "WHAT IS YOUR AGE";A
2Ø IF A<2 THEN PRINT "I DON'T BELIEVE YOU"
3Ø IF A>11Ø THEN PRINT "YOU MUST BE AN ALIEN"
4Ø IF A>12 AND A<2Ø THEN PRINT
     "YOU ARE IN YOUR TEENS"
5Ø GOTO 1Ø
```

Notice in line 4Ø how I have used the word AND to make the decisions more complicated for ORIC. We can use other words to string IF statements together. These are OR and NOT. So I could say:

```
IF X<1Ø AND A=5 OR Y=X THEN GOTO 2Ø
```

This means GOTO line 2Ø if both X is less than 1Ø and either A equals 5 or Y equals X. This **is** very complicated and is given as an example only. Using multiple decisions comes with practice and it is sufficient at this stage to know that they are there to be used.

NOT is also used in decisions:

```
IF NOT A=B THEN .....
```

is the same as

```
If A<>B THEN .....
```

or in English

```
IF A IS NOT EQUAL TO B THEN .....
```

IF ... THEN ... ELSE

The IF ... THEN ... ELSE construction is an extension of IF ... THEN, but gives more powerful and neater decisions. It tests if a condition or number of conditions are satisfied. If they are it performs a number of operations, otherwise it performs an alternate set of operations.

For example lines 4Ø and 5Ø in the last program could be put together into one line:

```
4Ø IF A>12 AND A<2Ø THEN PRINT "YOU ARE IN YOUR TEENS"
   ELSE GOTO 1Ø
```

GOTO, when used after THEN or ELSE may be missed out, so that in this example we can have ELSE 1Ø instead of ELSE GOTO 1Ø.

BEWARE: Computers, like ORIC, are more logical than we are. They do things just as instructed. The IF statement is the source of many errors in computing.

Look at this program.

```
1Ø INPUT"WHAT NUMBER";N
2Ø IF N<5 THEN PRINT"N<5"
3Ø IF N>5 THEN PRINT"N>5"
```

We have failed to tell ORIC what to do if N is equal to five. In this case it will do nothing but in longer programs this type of error can be disastrous.

The IF ... THEN ... ELSE ... structure is less prone to this type of error as ELSE covers all conditions not specified by IF. The program becomes

```
1Ø INPUT"WHAT NUMBER;"N
2Ø IF N<5 THEN PRINT"N<5" ELSE PRINT "N>=5"
```

The OR expression is true when either or both conditions are satisfied:

```
1Ø INPUT"FIRST NUMBER";X
2Ø INPUT"SECOND NUMBER";Y
3Ø IF X=5 OR Y=6 THEN PRINT "BINGO" ELSE 1Ø
```

Note how even when X=5 and Y=6 the PRINT statement takes place.

If we wish we can use the IF statement in a program loop to execute a part of a program a certain number of times.

Suppose we want to print out the numbers 1 to 1Ø. We could do it like this.

```
1Ø X=1
2Ø PRINT X
3Ø X=2
4Ø PRINT X
5Ø .
   .
  etc
```

There is however, a method which uses fewer instructions. Let's try:

```
10 X=1
20 PRINTX
30 X=X+1
40 IF X <11 THEN 20
```

This program has two important points. First, line 30 seems to be mathematical nonsense. X cannot equal X+1. To a computer this means 'let the memory location called X take on the value of the old value of X with one added to it'. Put another way 'let X become X+1'. However we say it, the effect of line 30 is to add one on to X. In line 10 we set X to 1, and so the first time we do line 30, X will become 2. The next time we do line 30, X will become 3 and so on.

Secondly line 40 has IF X <11 THEN ... The program stops when X=11 or is greater than 11. We wish the program to print up to ten and therefore we stop it when X has reached 10+1 or 11. Another common error in computer programs is to do a loop once more than is required or once less than is required.

We can use a similar program to print out all the even numbers between one and ten inclusive.

```
10 X=2
20 PRINT X
30 X=X+2
40 IF X>10 THEN STOP ELSE 20
```

It is important to look at line 40 again. If we had the statement IF X=11 THEN STOP or some such then the program would never stop as X is never equal to 11; it steps from 10 to 12. For safety's sake we could use the >= sign to stop a loop.

We can 'count down' to launch a rocket using the same sort of program.

```
10 X=10
20 PRINTX
30 X=X-1
40 IF X <=0 THEN PRINT "FIRE":STOP ELSE 20
```

MULTIPLE INSTRUCTION LINES

Now I've done something else in line 40. Not only have I been over cautious by using 'X <=0' where 'X=0' would do, I've also used a ':' and a further command.

Generally if the 'IF' statement is not true then the rest of the line is ignored until ELSE is encountered. If the 'IF' statement is true then the rest of the line preceding ELSE is executed.

FOR ... NEXT

Program loops are so important that they have been given a special set of instructions. FOR NEXT loops do exactly what we have done just now but without the 'IF' statement. These loops have already been mentioned briefly in chapter 5.

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

This will print out the numbers from 1 to 10. In fact any lines between the FOR and the NEXT will be done 10 times.

STEP

We used the instruction X=X+2 to allow us to print out in steps of two. The FOR ... NEXT command also can be used for this by defining the STEP.

```
10 FOR X=2 TO 10 STEP 2
20 PRINT X
30 NEXT
```

We can "count down" by:

```
10 FOR X=10 TO 1 STEP -1
20 PRINT X
30 NEXT X
40 PRINT"FIRE"
```

If you don't use the STEP part of the FOR command then the computer assumes you mean +1.

This can lead to a funny result. The FOR ... NEXT loop is always executed once even if it is not expected to. So:

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

will print out 10 then carry on. X will be 11 and as this is greater than 1 the computer thinks it has finished the loop.

We need not step in units. So:

```
10 FOR X=0.014 TO 0.131 STEP 0.017
20 PRINT X
30 NEXT
```

will also work.

We can even use variables in the loop. Try:

```
10 INPUT"HOW MANY";Y%
20 FOR X=1 TO Y%
30 PRINT X
40 NEXT
```

While we are on the subjects of loops these are two things you should never do.

1. Do not reset the loop variable continually to a constant value within a loop.

```
10 FOR X=1 TO 30
20 X=5
30 PRINT X
40 NEXT
```

2. Do not jump into the middle of a loop.

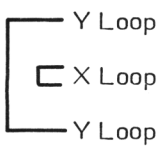
```
10 GOTO 30
20 FOR X=1 TO 10
30 PRINTX
40 NEXT
```

I'll leave you to work out why not. Just try the programs as they are if you don't understand and ORIC will tell you.

It is possible to "**nest**" loops. This is not a command word but refers to a program 'structure' (a way of constructing programs).

```
1Ø FOR Y=Ø TO 1Ø
2Ø FOR X=Ø TO 1Ø
3Ø PLOT X,Y,"*"
4Ø NEXT X
5Ø NEXT Y
```

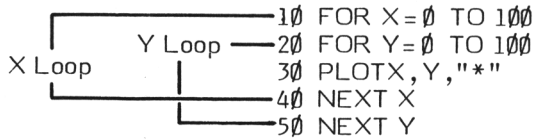
The X loop, lines 2Ø, 3Ø and 4Ø are "nested" inside lines 1Ø and 5Ø. So for each value of Y from Ø to 1Ø, X goes through its range. Diagrammatically this is shown:



Lines 4Ø and 5Ø may be shortened to

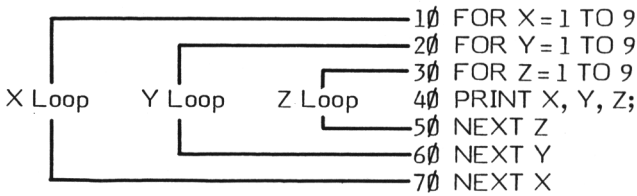
```
4Ø NEXT X,Y
```

The next program will NOT work.



as the nesting is all mixed up.

We can have several levels of nesting if we wish. For example:



Again lines 5Ø, 6Ø and 7Ø may be shortened to:

```
5Ø NEXT Z,Y,X
```

REPEAT ... UNTIL

Sometimes we do not know exactly how many times we wish to go round a loop. What we wish to do is repeat an action or number of actions until a condition is satisfied. If ORIC could make hay or recognise sunshine, it would rephrase an old proberb:

```
REPEAT make hay UNTIL NOT weather = sunshine
```

More realistically we could use the REPEAT ... UNTIL loop to detect that a particular number is entered:

```
10 REPEAT
20 INPUT"WHAT'S MY AGE";A
30 UNTIL A=16
40 PRINT"THAT'S RIGHT, SWEET 16"
50 PRINT"AND NEVER BEEN KISSED"
```

A common use of REPEAT ... UNTIL is to allow data to be entered via the keyboard until an 'end of data' input is received. Suppose for example that we wished to average an unknown number of values. We do know that none of these values will be 99, and so we can use 99 as a 'data terminator' – that is we enter 99 when we have entered all the values.

A program to illustrate this is:

```
10 REPEAT
20 T=T+V
30 N=N+1
40 INPUT"NEXT VALUE";V
50 UNTIL V=99
60 PRINT"AVERAGE IS "T/(N-1)
```

(If you have the V1.1 machine you don't need the space after IS in line 60).

This program works provided at least one value is entered before the data is terminated by entering 99.

Note – In programs similar to the above I shall put a space at the end of a string preceding a number. These spaces are necessary in the V1.0 machine. If you have the V1.1 machine you can leave them out.

REPEAT ... UNTIL loops can contain FOR ... NEXT loops nested within them, and vice versa. REPEAT ... UNTIL loops can be nested in the same way as FOR ... NEXT loops. For example, try:

```
1Ø REPEAT
2Ø : T=Ø:N=Ø
3Ø : REPEAT
4Ø :   T=T+V
5Ø :   N=N+1
6Ø :   INPUT"NEXT VALUE";V
7Ø : UNTIL V=99
8Ø : PRINT"AVERAGE IS "T/(N-1)
9Ø UNTIL T/(N-1)=2Ø
1ØØ PRINT"THAT'S JUST WHAT WE WANT"
```

I have used a colon directly after the line numbers in lines 2Ø to 8Ø. This allows spaces to be put in front of the instructions so that an indented listing is obtained. Where you have nested loops, indented listings can make your programs easier to understand.

If you wish to set up a continuous loop then REPEAT ... UNTIL Ø may be used.

SUBROUTINES

Program loops are very useful when we want to do the same thing many times. There are times however when we wish to do something a number of times in a program but not as a loop.

Suppose, for example, we wanted to add a 25% mark up on the purchase cost of a number of items to obtain a selling price P, and print out the result. This could be done by:

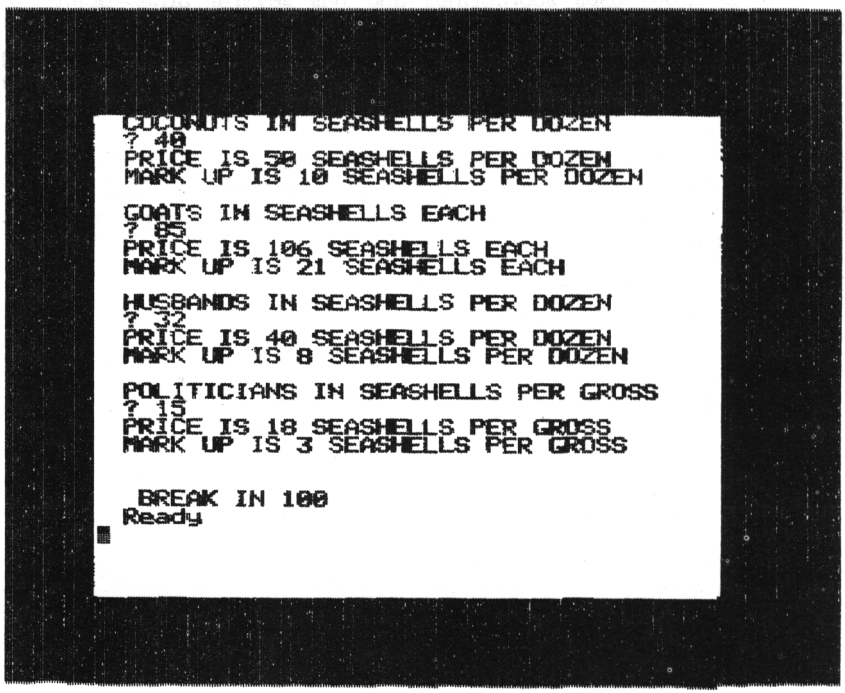
```
100 INPUT "COST";C
110 D=0.25*C
120 P=C+D
130 PRINT"SELLING PRICE = "P
140 PRINT"DISCOUNT = "D
```

In a long program we might need to do this ten or twenty times. That is a bit like hard work; so ORIC allows us to treat this sort of program as a subprogram, or **subroutine.** We 'call' a subroutine by the instruction GOSUB followed by the line number where the subroutine starts.

The next program uses subroutines and has several other points of interest. It was developed by the gentle inhabitants of the tropical paradise Sco'lan. Although the cannibal tribe on the neighbouring island En-guland consider the Sco'ish people (or Haggizbashirz) to be primitive, this program shows that they have their priorities correct.

The program uses string variables – which we saw with the INPUT instruction in the last chapter. It also uses integer variables so that all answers are whole numbers. This is because broken seashells are not negotiable currency.

```
10 CLS
20 A$="COCONUTS":B$="SEASHELLS PER DOZEN"
30 GOSUB 200
40 A$="GOATS":B$="SEASHELLS EACH"
50 GOSUB 200
60 A$="HUSBANDS":B$="SEASHELLS PER DOZEN"
70 GOSUB 200
80 A$="POLITICIANS":B$="SEASHELLS PER GROSS"
90 GOSUB 200
100 STOP
110 REM ##########################
200 REM SUBROUTINE
210 PRINT A$" IN "B$
220 INPUT C%
230 D%=0.25*C%
240 P%=C%+D%
250 PRINT"PRICE IS "P%B$
260 PRINT"MARK UP IS "D%B$
270 PRINT
280 RETURN
```



Note the instruction RETURN in line 28Ø. This tells ORIC to return from the subroutine to the next instruction following the GOSUB 'call'. All such subroutines must end in a RETURN statement.

I have 'called' the subroutine in lines 3Ø, 5Ø, 7Ø and 9Ø; once it was executed the program returned to lines 4Ø, 6Ø, 8Ø and 1ØØ respectively.

I had to be careful to ensure that the program did not accidently 'fall' into the subroutine without being called and I prevented this with the STOP at line 1ØØ. Otherwise the program would have gone on to line 11Ø after returning from the subroutine called at line 9Ø.

If the STOP instruction was not there the program would fail when it reached the RETURN. Try it; remember you cannot break your computer by making program mistakes.

It is possible for one subroutine to call another. For example:

```
1Ø INPUT"NUMBER BETWEEN 1&5";N
2Ø GOSUB 1ØØ
3Ø PRINT N:STOP
1ØØ REM CHECK NUMBER
11Ø IF N<1 OR N>5 THEN GOSUB 2ØØ
12Ø RETURN
2ØØ REM WRONG NUMBER
21Ø PRINT"NUMBER OUTSIDE RANGE 1 TO 5"
22Ø PRINT"DEFAULT NUMBER 4 WILL BE USED":N=4
23Ø RETURN
```

A subroutine may even call itself. This is called **recursion.** For example:

```
1Ø PRINT"SELECT THE SERVICE YOU REQUIRE"
2Ø PRINT"1. BALANCE ENQUIRY"
3Ø PRINT"2. DEPOSIT"
4Ø PRINT"3. WITHDRAWAL"
5Ø GOSUB 1ØØ
6Ø IF N=1% THEN PRINT"YOU CAN'T AFFORD A BALANCE
   ENQUIRY"
7Ø IF N=2% THEN PRINT"GIMME THE MONEY"
8Ø IF N=3% THEN PRINT"YOU'VE GOT TO BE JOKING"
9Ø STOP
1ØØ REM GET AND CHECK NUMBER
11Ø INPUT"WHAT IS YOUR CHOICE";N%
12Ø IF N% <1 OR N%>3 THEN PRINT"INVALID ENTRY":GOSUB 1ØØ
13Ø RETURN
```

ON ... GOTO

There is a neater method of constructing a program to allow the user to make a choice from a list or menu on the screen. This is the ON ... GOTO instruction, which is of the form:

```
ON N GOTO 1ØØ, 2ØØ, 3ØØ
```

This will cause the program to jump to line 1ØØ if N = 1, to line 2ØØ if N = 2 and to line 3ØØ if N = 3. If N is not a whole number it is rounded down. If N is outside the specified range then the program will carry out the instruction directly following the ON ... GOTO instruction.

The next program does the same as the previous one but uses the ON ... GOTO instruction.

```
1Ø PRINT"SELECT THE SERVICE YOU REQUIRE"
2Ø PRINT"1. BALANCE ENQUIRY"
3Ø PRINT"2. DEPOSIT"
4Ø PRINT"3. WITHDRAWAL"
5Ø INPUT"WHAT IS YOUR CHOICE";N
6Ø ON N GOTO 1ØØ, 11Ø, 12Ø
7Ø PRINT "INVALID ENTRY":GOTO 5Ø
1ØØ PRINT"YOU CAN'T AFFORD A BALANCE ENQUIRY":STOP
11Ø PRINT"GIMME THE MONEY":STOP
12Ø PRINT"YOU'VE GOT TO BE JOKING":STOP
```

ON ... GOSUB

On ... GOSUB works in a manner very similar to ON ... GOTO, but it is possibly rather more useful. Rather then merely jumping to the lines specified on the appropriate value of the variable it carries out subroutine jumps. For example:

```
1Ø REPEAT
2Ø INPUT"WHAT NUMBER";N
3Ø ON N GOSUB 1ØØ, 2ØØ, 3ØØ, 4ØØ
4Ø PRINT
5Ø UNTIL F=1:END
1ØØ PRINT"A PARTRIDGE IN A PEAR TREE":RETURN
2ØØ PRINT"TWO TURTLE DOVES":RETURN
3ØØ PRINT"THREE FRENCH HENS":RETURN
4ØØ PRINT"THAT'S ALL FOLKS":F=1:RETURN
```

END in line 5Ø does much the same as STOP except that after END the program can't be restarted with CONT.

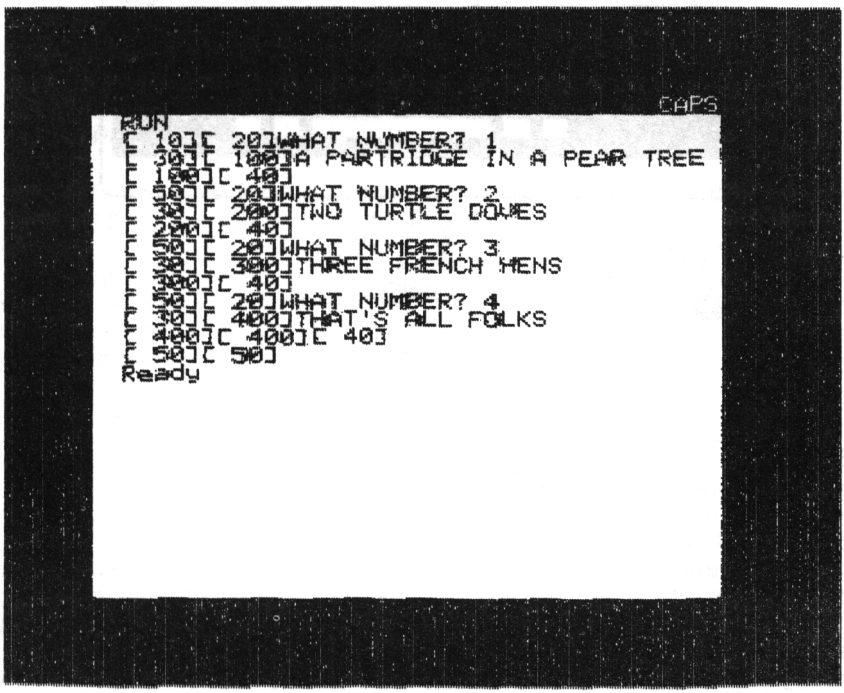
TRON AND TROFF

One effect of having decisions in programs is that they no longer work in strict line order. For example the last program may go to any one of lines 1ØØ to 4ØØ before line 4Ø.

When you are debugging such programs it is often useful to trace the exact order in which the lines are 'executed'. To do this you can put the instruction TRON (short for Trace On) in a program. Try the effect of adding the line:

```
5 TRON
```

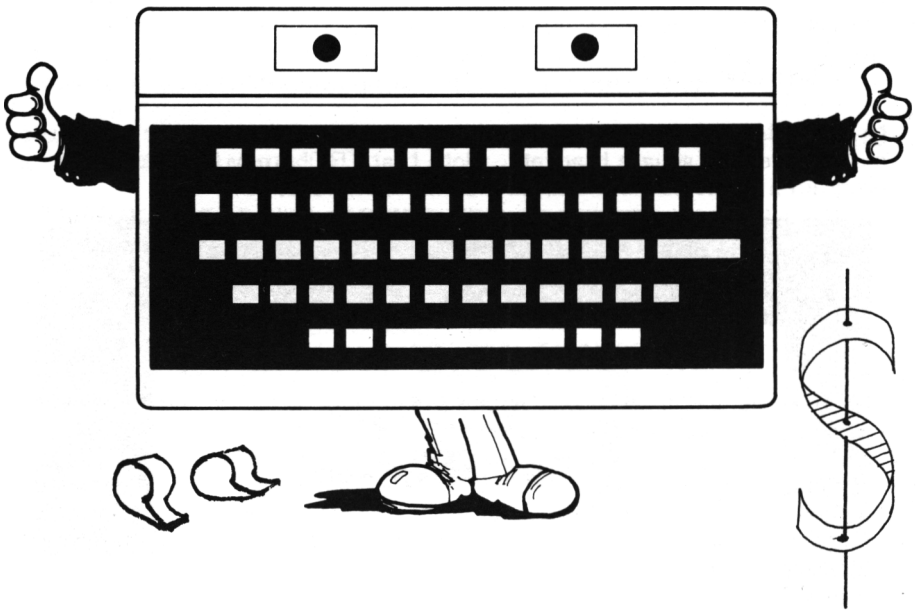
to the last progam. As each line is executed that line number is printed on the screen in square brackets. Note that TRON may be used only in a program line. It cannot be an immediate command.



If you wish to look only at a part of a program, then the instruction TROFF switches the trace back off. Try the effect of inserting TROFF at various points in the program. The NEW command turns off the trace as well as clearing program memory.

CHAPTER 9

A Way With Words



STRINGS

Words on the ORIC are called 'Strings' as they are strings of characters. So "ALAS POOR ORIC I KNEW HIM WELL" is a string which is 3Ø characters long. Remember a space is also a character. In fact anything you type in between the quotes, whether it is a number, a letter, a space, a symbol or whatever, is part of the string. To store this string in the computer we give it a name. This name is called a **string variable.** ORIC has some fairly strict rules about variables which we will look at later in this chapter. In the meantime we will take the easy way out and represent string variables by a single upper case (capital) letter followed by the symbol $ to represent a string.

Try the following:

```
1Ø A$="THIS IS A STRING"
2Ø B$="So is this"
3Ø C$="Numbers like 1,2 and even 3*4/6"
4Ø PRINT A$
5Ø PRINT B$
6Ø PRINT C$
```

You will see that a string can contain both lower case and upper case letters. However a string variable must be upper case – a$="wrong" is not acceptable to ORIC.

CONCATENATION

We can add strings together. This is called concatenation. For example:

```
1Ø A$="Hello"
2Ø B$="I'm ORIC"
3Ø C$=A$+B$
4Ø PRINT C$
```

will print HelloI'm ORIC. Because we didn't have a space between Hello and I'm, none has been added.

Any character created by CHR$ is itself a string and can be added to other strings. This is a particularly useful feature if we want to put a block of characters on two lines on the screen. This is best illustrated by an example.

```
1Ø A$=CHR$(27)+"I7K"
2Ø REM TWO ALTERNATE CHARACTERS
3Ø A$=A$+CHR$(1Ø)+ CHR$(8)+ CHR$(8)+CHR$(8)
4Ø REM ONE SPACE DOWN AND THREE SPACE LEFT CHARACTERS
5Ø A$=A$+CHR$(27) + "IUZ"
6Ø REM TWO MORE ALTERNATE CHARACTERS
7Ø CLS: PRINT CHR$(17)'CURSOR OFF
8Ø PRINT:PRINT:PRINT SPC(17)A$
9Ø GOTO 9Ø
1ØØ REM BREAK FROM PROGRAM WITH CTRL C
```

FRE

Another interesting feature of this program was that the string variable A$ was used to hold increasingly longer strings by being set equal to its previous contents plus some new characters. This is a convenient method of building up long strings, but it has a serious disadvantage. It uses up a lot of memory space. This is because all the versions of A$, not just the current one, are held in memory. This does not matter in a small program, but it could cause you to run out of memory space in a large one.

The instruction FRE lets you know how much memory space you have left. For example:

```
PRINT FRE(Ø)
```

used as an immediate command tells you the number of 'bytes' of memory remaining, and

```
1ØØ IF FRE(Ø)<15Ø THEN PRINT"NO MORE ENTRIES"
```

could be used in a data handling program.

Possibly even more useful is the form of the command:

```
FRE("")
```

If this is inserted in a program it will erase from the memory all versions of strings except the final versions.

STRING DECISIONS

We can make decisions using strings:

```
1Ø INPUT"DO IT AGAIN - Y/N";Y$
2Ø IF Y$="Y" THEN PRINT"IT IS DONE":GOTO 1Ø ELSE STOP
```

For strings to be equal they must contain exactly the same characters in the same order. Spaces do matter and upper case characters are not the same as lower case. If in the last example you entered 'YES' instead of 'Y' the program would stop, because "Y" does not equal "YES".

COMPARING STRINGS

We can compare strings using the greater than (>) and less than (<) symbols. These do not refer to the length of strings, but instead to the ASCII values of the characters in the strings.

For example the ASCII value of A is 65, and of B is 66.

So "A" is less than "B".

If the first characters of both strings are identical the second characters are compared, and so on until a difference is detected.

So "AA" is less than "AB";
"AAB" is greater than "AAA".

The next program lets you key in and compare any two strings. Use it until you are happy with the idea. Note that the comparisons are made on the string contents, not on the letters used to represent string variables.

```
10 REPEAT
20 INPUT"FIRST STRING";A$
30 INPUT"SECOND STRING";B$
40 IF A$<B$ THEN PRINT A$"<"B$
50 IF A$>B$ THEN PRINT A$">"B$
60 IF A$=B$ THEN PRINT A$" = "B$
70 UNTIL B$="STOP":STOP
```

Stop the program by entering 'STOP' as the second string.

As alphabetical order of the characters A to Z corresponds with their ASCII code number order we can use comparisons to sort strings into alphabetical order. Be careful, however, to remember that the lower case numbers all have ASCII codes higher than any of the upper case numbers, so that "a" is greater than "Z".

If you prefer:

if A$="EVIL" and B$="evil" then A$ is the lesser "evil";
if A$="Good" and B$="GOOD" then A$ is the greater "Good".

VAL

VAL changes a string of numerical data into a number. For example:

```
1Ø A$="2"
2Ø PRINT A$,VAL(A$)
```

will print 2 2 . Not at first sight very inspiring. In the first place, however, the program printed a string and in the second place a number. You cannot mix strings and numbers; so you cannot write PRINT 2+A$ in the same expression, but can write PRINT 2+VAL(A$).

VAL can also act on mathematical symbols such as +, -, * and /. Thus:

```
1Ø A$="5+14-6*3"
2Ø PRINT A$"="VAL(A$)
```

would print 5+14-6*3=1

If VAL is used on an alphabetic string it gives the value zero – even VAL(ORIC)=Ø.

STR$

STR$ does the opposite to VAL – it converts a number into a string.

Thus STR$(1234)="1234".

An advantage of changing a number into a string is that we then use the PLOT instruction to place it anywhere on the screen. For example –

```
PLOT 17,12,1234
```

gives an error, whereas –

```
PLOT 17,12,STR$(1234)
```

gives the desired effect.

Note – In the V1.Ø machine STR$ used with positive numbers inserts an illegal character at the start of the string. To correct this use A$=STR$(....):A$=RIGHT$(A$,LEN(A$)-1). LEN and RIGHT$ are described overleaf. This fault has been corrected on the V1.1 machine.

Another, and possibly even greater, advantage of changing a number into a string is that we may then use the powerful string manipulation instructions which ORIC makes available to us. Let's have a look at these.

LEN

First we can find out the length of a string using the LEN instruction.

```
10 INPUT"What is your name";N$
20 PRINT"Your name has "LEN(N$)"letters"
```

Line 20 uses LEN to count the number of letters in N$ and print this number out.

LEFT$

LEFT$ allows us to obtain a specified number of characters from the start of a string. For example:

```
PRINT LEFT$("SILLY EXAMPLE",5)
```

would print 'SILLY'.

Remember that spaces are characters and are counted, so that:

```
A$="A BETTER EXAMPLE":PRINT left$(A$,5)
```

would print 'A BET'.

RIGHT$

RIGHT$ allows us to obtain a specified number of characters from the end of a string. For example:

```
PRINT RIGHT$("A BETTER MOUSETRAP",4)
```

will print 'TRAP'.

MID$

LEFT$ and RIGHT$ are special cases of the general string slicing instruction MID$. The general form of this expression is:

```
MID$(A$,A,B).
```

This will give B characters of string A$, starting at position A and moving to the right. This sounds complicated, but a few examples should clear it up. For example:

```
PRINT MID$("12345",3,2)
A$="A STRING":PRINT MID$(A$,1,3)
B$="I LIKE ORIC":PRINT MID$(B$,5,2)+MID$(B$,9,2)
```

will print

```
34
A S
KERI
```

respectively.

If you do not specify a second number, MID$ will give a string starting at the selected position and going to the end of the string being sliced. For example:

```
PRINT MID$("FRED",2)
```

will print 'RED'.

MID$(A$,1,N) is the same as LEFT$(A$,N)
MID$(A$,N,1) gives the Nth character in a string.

We can use MID$ to print a word backwards as in the next program.

```
10 INPUT"WHAT WORD";W$
20 FOR N=LEN(W$) TO 1 STEP-1
30 PRINT MID$(W$,N,1);
40 NEXT
50 PRINT
60 GOTO 10
```

KEY$

Another method of entering a string from the keyboard is to use KEY$.

KEY$ used as part of an instruction (e.g. A$=KEY$) scans the keyboard to see if a key is pressed. Unlike the INPUT instruction, however, it does not cause the program to wait until you have typed in something followed by RETURN. If a key is pressed, KEY$ takes the value of the string containing the keyed-in character. If no key is pressed, KEY$ records the 'empty string'.

For example:

```
If key P is pressed KEY$ ="P"
If key 1 is pressed KEY$ ="1"
If no key is pressed KEY$ =""(empty string)
```

Using KEY$ clears the keyboard buffer – hence the form of the instruction A$=KEY$ which stores the entry more permanently in a string variable such as A$. If a key is held down KEY$ will not detect it again until the Repeat Key facility causes it to be re-entered into the keyboard buffer.

If two keys (not including SHIFT or CTRL) are held down at the same time KEY$ will not detect the second key pressed until the first is released.

To see the effect of KEY$ use the program:

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

GET

GET causes the program to stop until any key other than SHIFT or CTRL is pressed. GET is followed by a variable, for example GETA$, GETB.

```
10 GET A$
20 PRINT A$
30 WAIT 200
40 GOTO 10
```

with the previous program.

You may have wondered why I put line 3Ø in this program. You will find out when you come to break from the program using CTRL C – not easy – is it?

You have to use CTRL C just after you have pressed another key. Line 3Ø gives you time to do this.

If you wish to make ORIC inactive use:

```
1Ø GET B$
2Ø GOTO 1Ø
```

You can't get out of this with CTRL C as the GET instruction simply stores this entry – and all others – in string variable B$. To break from this loop you have to use the Reset key under the computer.

If a numeric entry is expected, GET may be followed by a numeric rather than a string variable. This has the disadvantage that a TYPE MISMATCH error will usually result from a non-numeric entry.

GET is often used simply to halt the program until any key is pressed. This is useful if, for example, you put a lot of text on the screen and ask the user to press any key to continue once he or she has read this. Here is another example which is popular with my children but not with my wife. It uses some instructions we haven't discussed yet, but you shouldn't find it difficult to work out what they do.

```
1Ø GET A$
2Ø ZAP:WAIT 5:SHOOT:WAIT 1Ø:PING:WAIT 5:EXPLODE
3Ø GOTO 1Ø
```

ASC

The ASC instruction gives the ASCII value of the first character of the string to which it is applied.

For example, the ASCII value associated with 'A' is 65.

```
So ASC("A")=65
   ASC("Another Example")=65
```

ASC used with the null or empty string "" (no space between inverted commas) gives an ILLEGAL QUANTITY ERROR.

CHR$

CHR$ is the opposite of ASC. It converts the code of a string character to that character.

Thus, taking the previous example,

PRINT CHR$(65) does the same as PRINT "A"

We have already discussed CHR$ in some detail in Chapter 7.

VARIABLES

We already know that a number may be stored as a numeric variable and a string as a string variable. Until now we have used single upper case letters, followed by $ if the variable is a string variable.

This was done for the sake of simplicity, but it is not the best programming practice. It is clearer, for example, to use NAME$ as a variable to hold names, rather than just N$. The line:

```
10 SHIPS = LINERS + FERRIES
```

is rather more descriptive than:

```
10 S = L + F
```

We can use any number of characters (within the limits imposed by the keyboard buffer) for both string and numeric variables, provided we observe these rules.

1. String variables must end with $, integer variables with %.

2. All letters must be upper case (capitals).

3. Variables may contain numbers and spaces, but the first character must be a letter.

4. Variables may not contain characters other than letters numbers and spaces (e.g. !,*='+) except for the $ and % following string and integer variables respectively.

5. Variables may not contain at any place within them letters which form any instruction or function used by the computer. For example we can't use TOTAL because it starts with TO (which is used in FOR ... NEXT loops), we can't use FRED$ because it starts with FRE, and we can't use PALLETS because it contains LET.

Breaking any of these rules will cause a syntax error. There is, however, one more restriction. This does not cause syntax errors, but could cause your programs to give the wrong results. Only the first **two** characters of a variable are used by the computer.

Thus the line we looked at previously:

```
1ØØ SHIP$=LINERS+FERRIES
```

is read by ORIC as

```
1ØØ SH=LI+FE
```

It is good programming practice to give your variables full meaningful names, but this is for our benefit, not for the computer's.

SHIPS and SHELLFISH are completely different things to us, but if used as variables they are both the same thing, SH, to ORIC.

For example:

```
1Ø SHIPS=1ØØ
2Ø SHELLFISH=1ØØØ
3Ø PRINT SHIPS
```

would print 1ØØØ, not 1ØØ. As far as ORIC is concerned the variable SH was set to 1ØØ in line 1Ø, changed to 1ØØØ in line 2Ø and printed in line 3Ø.

Similarly:

```
1Ø INPUT"FIRST NAME";NAME1$
2Ø INPUT"SECOND NAME": NAME2$
3Ø PRINT"HELLO "NAME1$" AND "NAME2$
```

will have a result you may not expect. Try it and see.

CLEAR

When you start a program using the immediate command RUN, all the numeric variables are set to zero and all the string variables are set to the empty string. You may wish to do this in the middle of a program as well. If so the instruction to use is CLEAR. Try

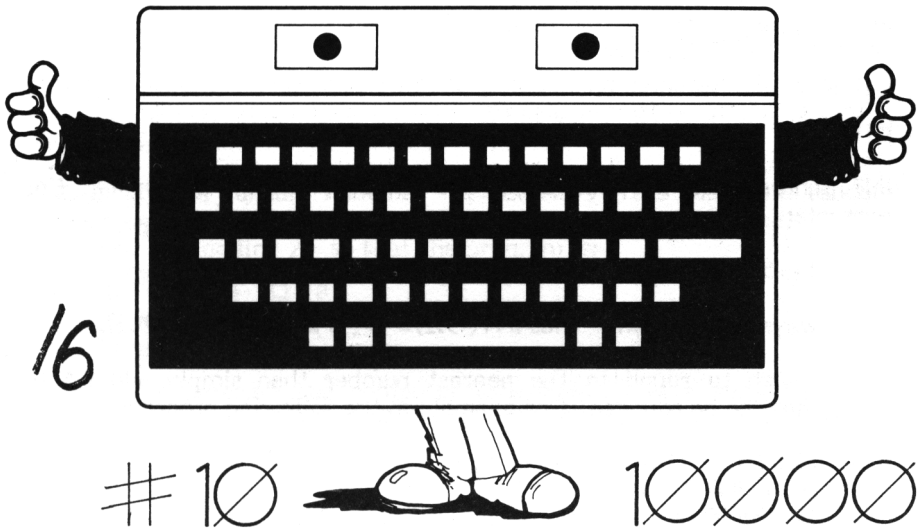
```
1Ø N=6:A$="AB"
2Ø CLEAR
3Ø PRINT N+1;A$+"A"
```

with and without line 2Ø.

CLEAR also clears all array elements (see next chapter).

CHAPTER 10

The Numbers Game



NUMBERS

It is time we looked at some numbers. As many of you will not be mathematicians, I'll keep this as painless as I can.

SGN

SGN lets us look at the sign of a number. If a number is positive then SGN will be +1; if a number is negative SGN will be -1. If a number is zero SGN will be zero. For example:

```
10 INPUT"WHAT NUMBER";N
20 IF SGN(N)=0 THEN PRINT"NUMBER = 0"
30 IF SGN(N)=-1 THEN PRINT"THAT'S NEGATIVE"
40 IF SGN(N)=1 THEN PRINT"A POSITIVE INPUT"
```

ABS

The ABS function completely ignores the sign, and will always return the absolute, or positive, value of a number. So ABS (-5) = 5, ABS(6) = 6, ABS(-23.6) = 23.6, and so on.

INT

INT ignores the decimal point and numbers coming after it. For example:

INT(5.1)=5, INT(5.6)=5, INT(5.0)=5

INT always rounds down. Thus INT(-5.1)=-6 (-6 is lower than -5.1).

If you wish to round to the nearest number then simply add 0.5 to the number to be rounded. For example in the following program:

```
10 INPUT"WHAT NUMBER";N
20 N=INT(N+0.5)
30 PRINT N
```

N will always be printed rounded to the nearest number.

RND

RND gives a random number between 0 and 1. This number can equal zero but cannot equal 1; although it can come very close to it. A statistician will tell you that the numbers are not quite truly random, but they certainly appear so to us ordinary mortals.

You must have a number in brackets after RND. This can be any number, but for all practical purposes we need use only RND(1), RND(0) and RND(-1)

RND(1) gives a random number between 0 and 1. It gives a different random number each time when used normally. For example each time you RUN the program:

```
10 FOR N=1 TO 10
20 PRINT RND(1)
30 NEXT
```

you will get a different sequence of random numbers.

RND(-1) 'seeds' the random number generator. This means that it determines exactly where in the sequence of random numbers RND(1) will start. Add the line:

```
5 X=RND(-1)
```

to the program. The first time you RUN the program you will get a set of random numbers as before. When you RUN it again, however, you will get the same set of random numbers.

RND(0) gives a random number. It, however, gives the same random number each time the same RND(0) in a program is executed. Delete line 5 and change line 20 so that the program is:

```
10 FOR N=1 TO 10
20 PRINT RND(0)
30 NEXT
```

In the next program the computer uses RND(1) in a guessing game. As RND(1) cannot equal 1, INT(RND(1)*10) gives a number between 0 and 9 (because INT rounds down).

```
10 CLS
20 PRINT"GUESS A NUMBER BETWEEN 1 AND 10"
30 X=1+INT(RND(1)*10)
40 : REPEAT
50 :   GET A$
60 :   Y=VAL(A$)
70 :   IF Y>X THEN PRINT Y"IS TOO HIGH"
80 :   IF Y<X THEN PRINT Y"IS TOO LOW"
90 :   ATTEMPTS=ATTEMPTS + 1
100 : UNTIL Y=X
110 PRINT"WELL DONE "Y"IS CORRECT"
120 PRINT"YOU TOOK "ATTEMPTS"ATTEMPTS"
```

MATHEMATICS

The next seven sections cover some 'heavy' maths. If this puts you off you can skip them and go on to 'READ, DATA, RESTORE'. If you do decide to do this, you will still have these sections to refer back to should you come across any of the functions they describe in programs.

POWERS

The ^ sign is used to 'raise to the power' of a number, or 'multiply a number by itself 'power' times'. So 2^2 = 2 x 2 = 4, or 7^4 = 7 x 7 x 7 x 7 = 24Ø1. We can also have 2^(1/2) or 2^(Ø.5) = 1.4142136. Although entered as ^ from the keyboard this sign is displayed as ↑ on the screen.

SQR

A number raised to the power one half is also known as the square root of that number. Square roots can also be found by using the expression SQR on the ORIC. So to print out the first ten numbers and their square roots we could write this:

```
1Ø FOR X=1 TO 1Ø
2Ø PRINT X,SQR(X)
3Ø NEXT
```

LOG

LOG gives the logarithm of a number to the base 1Ø . This means that if 1Ø ^X = Y then LOG(Y) = X. If you went to school before the days of the pocket calculator you probably learned about this logarithm.

EXP AND LN

EXP and LN are mathematical opposites. LN is a 'natural' logarithm. If (e)^X = Y then LN(Y) = X, where e is a number approximately equal to 2.718281828. If you understand this you may want to use the function in mathematical programs. If you don't – don't worry. It doesn't have much use except in such programs.

EXP stands for exponential and describes the way many events in nature are controlled. For instance, radioactive decay and population explosions are exponential. If you wish to study these EXP will be useful. If X=LN(Y) then Y=EXP(X).

PI

PI is also available on the ORIC. PI or π is 3.1415927 approximately. You can use it to find areas and circumferences of circles. The area of a circle is PI x radius x radius. The circumference is 2 x PI x radius.

```
1Ø INPUT"CIRCLE RADIUS";RAD
2Ø PRINT"AREA="RAD 2*PI
3Ø PRINT"CIRCUMFERENCE="2*PI*RAD
```

SIN, COS, TAN, ATN

Speaking of PI and circles brings to mind some trigonometry. ORIC allows you to calculate sines (SIN), cosines (COS), and tangents (TAN). The inverse of TAN, Arctan (ATN), is also there. If you know what I mean by these I need not go on. If you are not familiar with these terms, never mind, they are not needed for many programs; except (possibly) for SIN which is quite useful for drawing curves. We will look at this in Chapter 12.

I have to add, for those who are familiar with trigonometry, that ORIC expects angles in radians not degrees. To convert radians to degrees divide by PI and multiply by 18Ø, or, if you like, one degree is approximately Ø.Ø174533 radian.

DEF FN

We can define our own mathematical functions. The procedure is to have a line, before the first call to the function, of the form

DEF FN ... (the space is optional)

This means define function. The function name follows the same rules as those for variable names. You must also use brackets after the name. These should contain the variable (or one of the variables) used in the function. After this you put an equals sign and then what you want the function to do.

For example:

```
1Ø DEF FNA(X)=5+X
```

would give a result equal to 5 plus the number in brackets whenever called.

To call a function you simply name it. Thus

```
2Ø PRINT FNA(6)
```

would print 11.

If you have a function with more than one variable you may still define it. What you do in this case is to define the function in terms of one of the variables and set the other variable or variables as required before calling the function.

For example:

```
10 DEF FNS(A)=A^2+B
20 B=6
30 PRINT FNS(2)
40 PRINT FNS(3)
50 B=10
60 PRINT FNS(2)
```

would print:

```
10
15
14
```

Functions are used to do often repeated arithmetic operations. We could, for example, calculate areas and circumferences using functions.

```
10 DEF FNCIRCUM(RAD)=2*PI*RAD
20 DEF FNAREA(RAD)=PI*RAD^2
30 : REPEAT
40 :   INPUT"RADIUS";RAD
50 :   PRINT"CIRCUMFERENCE="FNCIRCUM(RAD)
60 :   PRINT"AREA="FNAREA(RAD)
70 :   PRINT
80 : UNTIL RAD=0
90 END
```

Enter a zero radius to break from the program.

READ, DATA, RESTORE

There are three very useful commands which we have not looked at so far. These are the READ, DATA and RESTORE instructions.

The READ statement reads from a DATA statement, so

```
10 DATA 10,20,30,40, HELLO
20 READ X,Y,X,A,B$
```

would be the same as

```
1Ø X=1Ø
2Ø Y=2Ø
3Ø Z=3Ø
4Ø A=4Ø
5Ø B$="HELLO"
```

The data can be anywhere in the program. If the computer encounters a DATA statement whilst executing a program then it just ignores it. You don't need inverted commas round a string in a DATA statement unless you wish to include leading or trailing spaces.

It can be useful to group data such as name and age so that you can see them in the program. Look at the following program.

```
1Ø DATA Joe Smith,23
2Ø DATA Fred Bloggs,48
3Ø DATA Alice Jones,27
4Ø DATA Sue Smith,15
1ØØ FOR K=1 TO 4
11Ø READ N$,A
12Ø PRINT "Name is "N$,"Age is "A
13Ø NEXT
```

If you try to READ more data than there are you will get an OUT OF DATA error. Every time you do a READ you get the next DATA item wherever it is in the program.

There are times when you want to start at the beginning again. To do this use the command RESTORE. If you try to read the same lot of data twice without using RESTORE you will again get an 'OUT OF DATA' error.

ARRAYS

A very useful way of handling things in the computer is by arrays. An array is a set of numbers all called by the same variable name but distinguished by a number in brackets.

DIM

To allow the ORIC to recognise an array we could start with DIM statement to 'dimension' it. DIM A(14) will reserve 15 locations called A(Ø) to A(14). If we don't dimension the array ORIC will reserve 11 locations (numbered from Ø to 1Ø) for it.

We could then load the array – ie:

```
1Ø DIM A(14)
2Ø A(Ø)=6
3Ø A(6)=43
4Ø A(14)=127
```

It is much better to load an array with either an INPUT or DATA statement. For example:

```
1Ø DIM A(14)
2Ø FOR K=Ø TO 14
3Ø READ A (K)
4Ø NEXT
5Ø DATA 1,2,3,5,1,2,2,4,9,15,6,43,8,2Ø,31
```

Note how easy it is to refer to a(K) as the Kth number stored in A. RUN this program and enter PRINT A(Ø) PRINT A(1), PRINT A(2) etc as direct commands.

Array names follow the same rules as do variable names. Thus we could store 25 prices, for example, in an array dimensioned by DIM PRICES (24).

We may also have arrays with more than one dimension. For example:

```
DIM B(2,2)
```

represents an array of nine locations, rather like a noughts and crosses grid.

```
B(Ø,Ø)B(Ø,1)B(Ø,2)
B(1,Ø)B(1,1)B(1,2)
B(2,Ø)B(2,1)B(2,2)
```

Each element can contain a number.

You can think of a two dimensional array as name (row, column). However, ORIC could use

```
DIM BIGARRAY (3,3,3,3,3,)
```

I don't know what you would want to use such an expression for, but when you do it's available.

You do not have to dimension multidimensional arrays provided that you don't require any of the dimension numbers to be larger than 10. It is, however, wise to do so.

If you did not dimension array B in the last example, ORIC would have assumed an 11 element square array and reserved space for 121 items instead of the 9 you actually needed. If you did not dimension BIGARRAY ORIC would have attempted to reserve space for 11x11x11x 11x11 items and would have run out of memory. An array item takes up at least two bytes of memory.

Arrays can hold numbers of strings. Numeric arrays can be **real** or **integer**. Integer arrays can hold only whole numbers (integers). Real arrays can hold numbers with figures after the decimal point (e.g. 1.2, 0.34). An integer array has the symbol % after the array name. A string array has the symbol $ after the array name.

String arrays can hold strings in exactly the same way as numeric displays hold numbers:

```
10 DIM NAME$(15)
20 FOR K=0 TO 15
30 READ NAME$(K)
40 NEXT
50 DATA PETER,SIMON,WALTER,TONY,LOUIE
60 DATA JOHN F.,JOHN G.,IAN,ARTHUR,JOHN K.
70 DATA HUGH,JACK,FRANK,TOM,ERIC,PHIL
80 PRINT NAME$(7)" IS MAGIC"
90 REM LINE 80 IS RATHER SILLY
```

If we wished we could hold strings in multi dimensional arrays. Single dimensional string arrays however, tend to be the most common.

FUNNY NUMBERS

This section looks like more boring maths. I would, however, ask you to give it some attention.

If you know how ORIC counts you will find a lot of things easier to understand.

Let's take any number – say 5831

This may be written as

```
 5x1ØØØ
+8x1ØØ
+3x1Ø
+1
```

or

```
 5x1Øx1Øx1Ø
+8x1Øx1Ø
+3x1Ø
+1
```

in other words each digit position as we move to the left is multiplied by a higher 'power' of ten, or a number ten times larger than was the one to the right of it.

There is nothing 'magic' about the number 1Ø. We have grown used to working with powers of ten simply because most of us have ten fingers (eight fingers and two thumbs if you want to be pedantic).

Digital computers like ORIC, however, have only two 'fingers' or two things to count with. A signal may either be there or not there. A switch may be open or closed. A condition may be 'true' or 'false'. All this is represented by the digits 1 (on) or Ø (off).

Ultimately ORIC sees everything in terms of an enormous number of ones and zeroes. These are all it can recognise.

We count in decimal, using ten counting symbols or numbers (Ø to 9). ORIC counts in binary, using only two counting symbols (Ø and 1). So how can it count with only two symbols?

Well Ø (decimal) = Ø (binary)
1 (decimal) = 1 (binary)

But what does ORIC do when it reaches two? It has no more symbols to use, and so it must do what we do when we reach ten and run out symbols. It moves one space to the left.

So 2 (decimal) = 1Ø (binary)
3 (decimal) = 11 (binary)

ORIC has run out of symbols again; so it must move another space to the left.

4 (decimal) = 1ØØ (binary)
5 (decimal) = 1Ø1 (binary)
6 (decimal) = 11Ø (binary)
7 (decimal) = 111 (binary)

Thus 111 (binary) is:

```
 1x2x2
+1x2
+1
```

which equals 7 (decimal)

I hope you see the pattern which is emerging

1111 (binary) is:

```
 1x2x2x2
+1x2x2
+1x2
+1
```

which equals 15 (decimal).

Each column moved to the left is a power of two (instead of a power of ten) greater than the one to the right of it. Thus they go up 1,2,4,8,16,32 and so on.

A single binary digit (1 or Ø) is called a bit for short. One of ORIC's memory locations holds eight bits or one byte.

Thus the maximum number one memory location can hold is:

11111111 (binary).

This equals:

```
 1x1
+1x2
+1x4
+1x8
+1x16
+1x32
+1x64
+1x128 (left hand column)
```

that is 255 (decimal).

Thus a typical memory location could hold – say – the binary number

1Ø1ØØ111.

This equals

1+(1x2)+(1x4)+(Øx8)+(Øx16)+(1x32)+(Øx64)+(1x128)

= 1+2+4+32+128 = 167 (decimal).

It is very inconvenient for us to write out numbers like 11Ø11Ø11; so we represent them with what are called hexadecimal numbers. Hexadecimal numbers are simply a shorthand way of writing binary numbers. They are a convenience to us – ORIC uses only binary.

To get a hexadecimal number we group the binary number into lots of four. Thus a single byte, which is 8 bits, is grouped in two lots of four.

i.e. 1Ø11 Ø1Ø1

Each of these groups will be represented by a separate character – easy for values up to nine:

```
ØØØØ=Ø
ØØØ1=1
  ...
1ØØ1=9
```

For the binary numbers from 1Ø1Ø to 1111 we use the symbols A to F respectively.

So 11 (decimal) = 1Ø11 (binary) = B (hexadecimal)
16 (decimal) = ØØØ1 ØØØØ (binary) = 1Ø (hexadecimal)

ORIC can work with hexadecimal numbers up to FFFF. To indicate a number is hexadecimal put the symbol # in front of it.

Thus PRINT#FFFF would give 65535
PRINT#1A would give 26.

To convert from decimal to hexadecimal use HEX$. For example PRINT HEX$(26) gives #1A. As neither # nor A can be printed as numbers HEX$ generates a string. HEX$(Ø) generates the string "#" in the V1.Ø machine and the string "#Ø" in the V1.1 machine.

AND, OR

We saw AND and OR in Chapter 8, where they let us make decisions based on two or more conditions. AND and OR are also logical functions which operate on binary numbers.

For example:

1 AND Ø = Ø
Ø AND 1 = Ø
Ø AND Ø = Ø
1 AND 1 = 1

Note that 1 AND 1 is not the same as 1 + 1! Also:

1 OR 1 = 1
Ø OR 1 = 1
1 OR Ø = 1
Ø OR Ø = Ø

AND and OR operate a bit at a time. Thus:

1Ø11 AND 11Ø1 = 1ØØ1 (all binary)

If you don't understand this don't worry. You can write all your programs without logical operators. They give, however, one more weapon in your armoury should you decide to employ them.

For example:

```
10 INPUT"NUMBER BETWEEN 0 AND 7";N%
20 N=7 AND N%
```

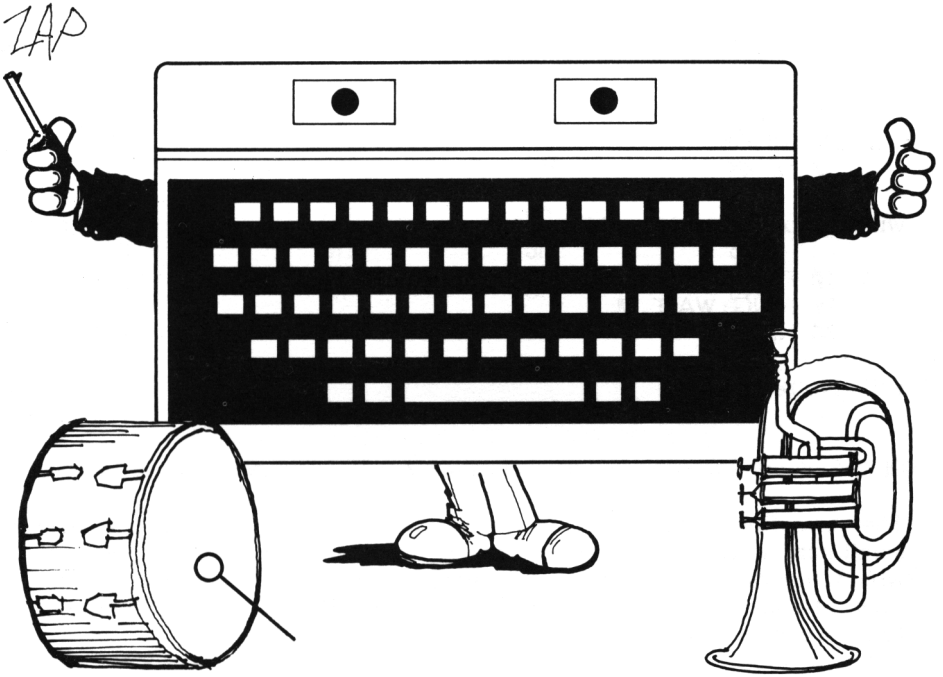
will ensure N is a whole number between 1 and 7 whatever is entered.

```
10 FOR N=1 TO 20
20 PRINT 10+5*(N AND 1)
30 NEXT
```

will print 10 when N is even and 15 when N is odd.

CHAPTER 11

Sound It Out



ORIC has four predefined sound commands. These (I hope) are self explanatory:

ZAP
PING
SHOOT
EXPLODE

One of the good features about ORIC sound is that you only need the instruction to start the sound going. While the sound is being generated the program can be doing something else. You can have a bullet going across the screen while SHOOT is sounding.

If you want a series of predefined sounds you need to put a delay after each, to give it time to finish.

Thus:

```
10 FOR N=1 TO 10
20 PING
30 NEXT
```

will give only one ping, while,

```
10 FOR N=1 TO 10
20 PING:WAIT 10
30 NEXT
```

will give ten pings.

PLAY

The predefined sounds, useful though they may be, are only a very small part of ORIC'S sound repertoire. ORIC has three sound channels, which means it can play three different sounds at the same time. Each of these three sound channels can generate either pure (musical) tone or can be mixed with noise to give special effects.

Before experimenting with ORIC'S sound ensure the keyclick is off. CTRL F controls the keyclick. If you get a sound on continuously then CTRL F followed by any key (RETURN is usually best) will normally switch it off.

The PLAY command switches the three sound channels on and off and determines which channels can be combined with the noise generator.

PLAY does not by itself make any sound. It enables (switches on) channels which can then be used by the instructions SOUND and MUSIC. The PLAY instruction most commonly used is PLAY 0,0,0,0. This switches off all sound.

When the machine is first switched on, channel 1 is enabled. No noise mixing is enabled on power up.

The format of the PLAY command is:

PLAY TE,NE,EM,EP

where TE, NE, EM, EP are all numbers.

TE stands for tone enable and is a number between Ø and 7.

The next table shows how TE values control the tone channels.

TE	CHANNEL 3	CHANNEL 2	CHANNEL 1
Ø	off	off	off
1	off	off	on
2	off	on	off
3	off	on	on
4	on	off	off
5	on	off	on
6	on	on	off
7	on	on	on

Thus, for example, when TE = 3 channels 1 and 2 are enabled; when TE = 7 all channels are enabled.

Compare the effect on the channels with the binary numbers in the previous chapters. Can you see a connection? Try representing 'off' by Ø and 'on' by 1.

NE stands for noise enable and selects the channels which may have noise mixed in with the tone.

NE follows the same pattern as shown in the table for TE – i.e. when NE = 3 (for example) noise may be mixed with tone in channels 1 and 2.

We will return to EM and EP shortly. Keep them at zero in the meantime.

SOUND

If a channel is enabled, the instruction SOUND causes a sound to be generated by that channel. The instruction is of the form:

SOUND C,P,V

where C, P and V are numbers representing Channel, Period and Volume.

C selects the channel on which the instruction operates.

When C is 1, Channel 1 is used for pure tone.
" " " 2, " 2 " " " " "
" " " 3, " 3 " " " " "

When C is 4, Channel 1 is used for tone and noise mixed.
" " " 5, " 2 " " " " " " " .
" " " 6, " 3 " " " " " " " .

V selects the volume. The normal range is from 1 to 15. When V equals 1 the sound is fairly quiet; a value of 3 or 4 is good for experimentation; a value of 6 or 7 is loud enough for most purposes.

P selects how high or low the note is, but in a rather odd way. P stands for period and not (as you might expect) for pitch. This means that the smaller P is, the higher the sound that is generated. A value of 238 is about mid range on a piano key board. The lowest note is generated when P = 4Ø96. The highest I can hear is generated when P = 4.

The next program lets you try different values of P. Pressing any key stops the sound and lets you enter another value. A value of zero stops the program.

```
1Ø REPEAT
2Ø INPUT"PERIOD NUMBER";P
3Ø IF P=Ø THEN STOP
4Ø PLAY 1,Ø,Ø,Ø
5Ø SOUND 1,P,4
6Ø GET A$
7Ø PLAY Ø,Ø,Ø,Ø
8Ø UNTIL Ø
```

To try the effect of introducing noise, change line 4Ø and 5Ø to:

```
4Ø PLAY 1,1,Ø,Ø
5Ø SOUND 4,P,4
```

This program may be extended so that three period numbers can be entered and three channels (with or without noise) sounding at once. Experiment with this for yourself.

ENVELOPE

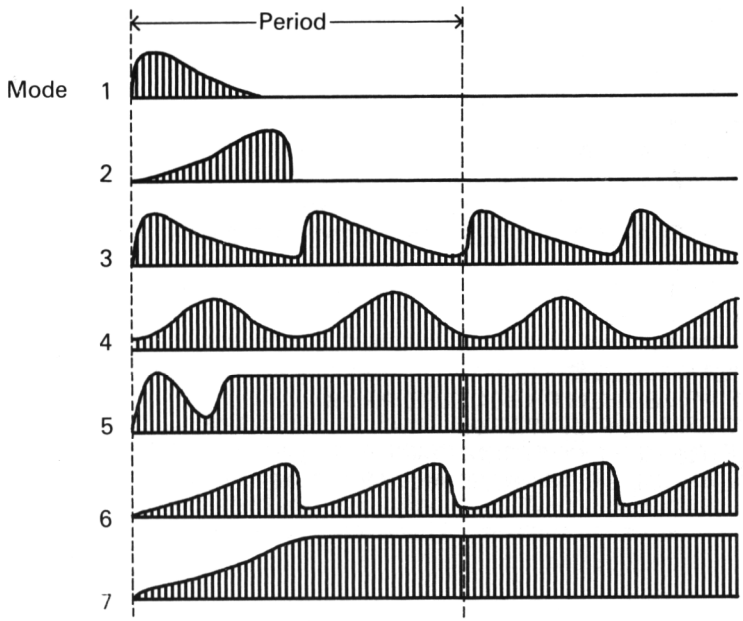
You will recall that I promised to come back to the EM and EP parameters in the PLAY instruction. These are used when, instead of a sound at constant volume, we want a sound which varies in volume at different stages.

For example we may want a sound to build up gradually in volume and then die away quickly (slow attack, fast decay). Or we may want a sound which starts at maximum volume and dies away gradually (fast attack, slow decay).

EM can be used to select one of seven 'envelopes' with varying attacks and decays.

EP is the envelope period measured in milliseconds (thousandths of a second). Do not confuse this timing with that of the WAIT command, which is in hundredths of a second.

The next diagram shows the seven envelopes available. The period length is also shown. Modes 1 and 2 stop the sound after half a period. Modes 3 to 7 are continuous.



If you specify an envelope in your PLAY command then any subsequent sound will be controlled by that envelope. You may choose the volume of your sounds as you did previously, but if you wish you can specify a volume figure Ø. This lets the PLAY command control your volume and tends to make the envelope more effective.

You can use the next program to select envelope modes, sound periods, and noise or pure tone. This is a useful program for investigating sound effects. You could also make the envelope period keyboard selectable. Pressing any key stops the sound and allows another choice. Selecting envelope mode zero stops the program.

```
1Ø REPEAT
2Ø INPUT"ENVELOPE MODE";E
3Ø INPUT"SOUND PERIOD";P
4Ø INPUT"DO YOU WANT NOISE (Y/N)",Y$
5Ø IF Y$="Y" THEN C=4:N=1 ELSE C=1:N=Ø
6Ø PLAY 1,N,E,5ØØ
7Ø SOUND C,P,Ø
8Ø GET A$
9Ø PLAY Ø,Ø,Ø,Ø
1ØØ UNTIL E=Ø
```

The SOUND instruction is useful for noises and special effects. Here are a few to try:

```
1Ø REM CHOO-CHOO TRAIN
2Ø REPEAT
3Ø PLAY 1,1,2,5Ø
4Ø SOUND 4,1Ø,Ø:WAIT 15
5Ø UNTIL KEY$<>"":REM EMPTY STRING
6Ø PLAY Ø,Ø,Ø,Ø:REM PRESS ANY KEY TO END
```

```
1Ø REM RUNNER
2Ø REPEAT
3Ø PLAY 1,1,1,1ØØ
4Ø IF PR=8 THEN PR=14 ELSE PR=8
5Ø SOUND 4,PR,Ø:WAIT 2Ø
6Ø UNTIL KEY$<>"":REM EMPTY STRING
7Ø PLAY Ø,Ø,Ø:REM PRESS ANY KEY TO END
```

```
1Ø REM DRUMS
2Ø REPEAT
3Ø : FOR N=1 TO 8
4Ø : PLAY 1,1,1,35
5Ø : SOUND 4,35,Ø
6Ø : WAIT 5
7Ø : NEXT
8Ø WAIT 1Ø
9Ø : FOR N=1 TO 2
1ØØ :PLAY 1,1,1,3ØØ
11Ø :SOUND 4,3ØØ*N,Ø
12Ø :WAIT 6Ø
13Ø :NEXT
14Ø UNTIL KEY$<>"":REM EMPTY STRING
15Ø PLAY Ø,Ø,Ø,Ø'PRESS ANY KEY TO END
```

MUSIC

The SOUND instruction is useful for sound effects like the ones we have just heard. It could be used for playing tunes, but ORIC has another instruction, MUSIC, which is provided for this purpose.

The MUSIC instruction takes the form:

```
MUSIC C,OC,N,V
```

where C, OC, N and V are numbers. C is the channel select, and is a whole number between 1 and 6. This works in the same way as the channel select in the SOUND instruction.

OC stands for octave and is a whole number between Ø and 6.

N stands for note and is a whole number between 1 and 12.

V stands for volume and is a whole number between Ø and 15. This works in the same way as the volume select in the SOUND instruction.

For the MUSIC instruction to have any effect, the channels it uses must be switched on. This is done by the PLAY instruction. The length of the note may be determined by a WAIT instruction. The note may be switched off by using the PLAY instruction to switch off the relevant channel. PLAY Ø,Ø,Ø,Ø switches off all sound.

Before we look at MUSIC in more detail, try this program. It lets you use ORIC as an organ. Please make sure the keyclick and capitals shift are switched off before you RUN the program. Pressing the SHIFT key with the alphabetic keys gives lower notes.

```
10 REPEAT
20 GET A$:A=ASC(A$)
30 A=ABS(A-32):IF A>83 THEN A=83
40 OCT%=A/12
50 NT=A-12*OCT%+1
60 PLAY 1,0,0,0
70 MUSIC 1,OCT%,NT,5
80 WAIT 50
90 PLAY 0,0,0,0
100 UNTIL A=0'SPACE BAR STOPS PROGRAM
```

This is only the bare bones of a program to let you hear all the notes. Once you are happy with the MUSIC instruction you can expand the program so that the notes are played in the order that keys are placed on the keyboard, so that you can select volume and note length and so that all three channels may be selected at the same time.

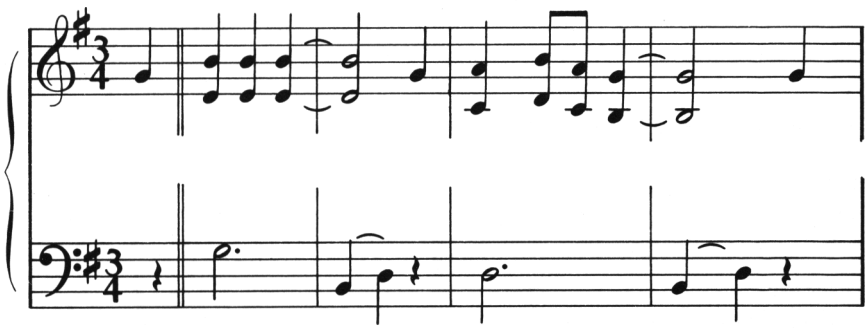
To get a 'tremulo' effect replace lines 60 and 70 by

```
60 PLAY 1,0,4,100
70 MUSIC 1,OCT%,NT,0
```

MUSICAL THEORY

If you are going to play tunes on ORIC you will need to know a little musical theory. If you know all about music you can skip over this section very quickly.

Even simple tunes can look daunting on sheet music. For a start there are usually two sets of five lines joined together.



You will see that the top set of lines starts with the symbol 𝄞, while the bottom line starts with the symbol 𝄢.

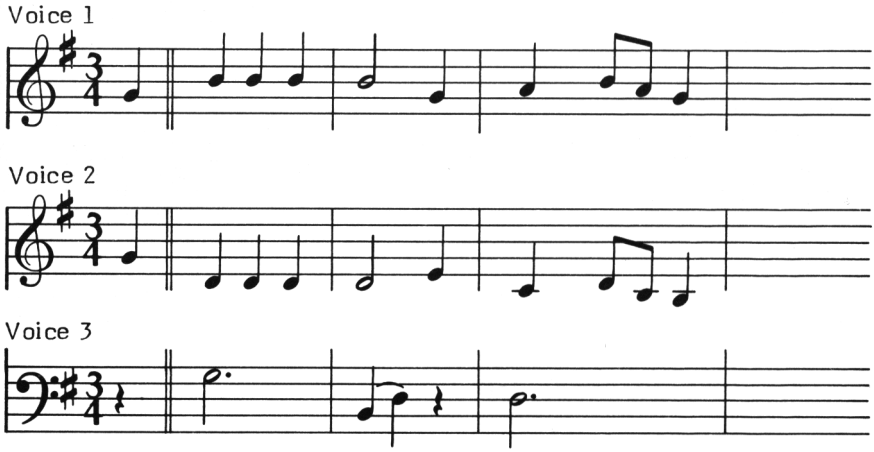
𝄞 is called the 'treble clef';

𝄢 is called the 'bass clef'.

HARMONY

As you probably know the symbols of the form ♩ stand for musical notes. However, you will see that most of the notes in this tune are of the form ♩ .

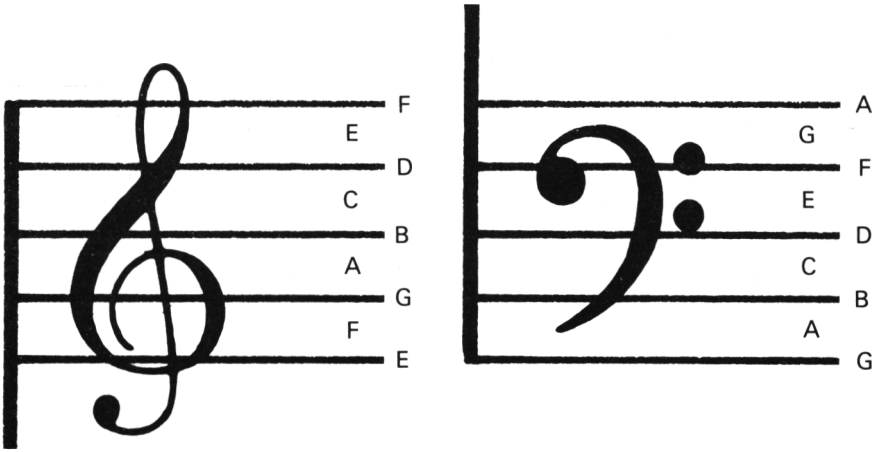
A lot of the sheet music you come across will have this sort of multiple note. This is because the tunes are written so that they can be sung in 'harmony', i.e. two or more people sing and each sings different notes. You have probably heard harmony groups such as 'barber shop' quartets. The multiple notes are a shorthand way of writing the music. You could write the tune as:



The tune has three 'voices'. ORIC has three channels to play them on.

PITCH

The five lines on which sheet music is written are called the **stave.** Again you may know that notes written near the top of the stave are of a higher pitch than those written near the bottom. For convenience notes are given 'names' which determine their pitch. These names consist of the letters A to G. Thus the position of a note on the lines can be represented by a letter.



You will probably have heard of the musical scale:

doh, re, me, fah, soh, la, te, doh

The eight notes of the scale are called an 'octave'. The scale of 'C major' is an octave which starts and ends with C. That is:

C = doh, D = re, E = me, F = fah, G = soh, A = la, B = te, C = doh.

Similarly a scale of A starts and ends with the note A.

There is, however, a complication. Not all of the steps in frequency (changes in pitch) between the notes are the same. For example the change in pitch between the notes B and C is only half of the change in pitch between C and D.

To allow for this a scale or octave is split into twelve 'semitones'.

These are:

A
A# or B♭
B
C
C# or D♭
D
D# or E♭
E
F
F# or G♭
G
G# or A♭

The symbol # means 'sharp' and ♭ means 'flat'. So G# is a semitone higher than G and A♭ is a semitone lower than A. G# is the same note as A♭ .

I shall not go into scales in detail – suffice it to say that the scale which sounds 'right' to most of us is the major scale. This goes up in semitone steps as follows:

doh (2 steps) re (2 steps) me (1 step) fah (2 steps) soh (2 steps) la (2 steps) te (1 step) doh

so that the scale of C major is C, D, E, F, G, A, B, C.

And the scale of G Major is G, A, B, C, D, E, F#, G.

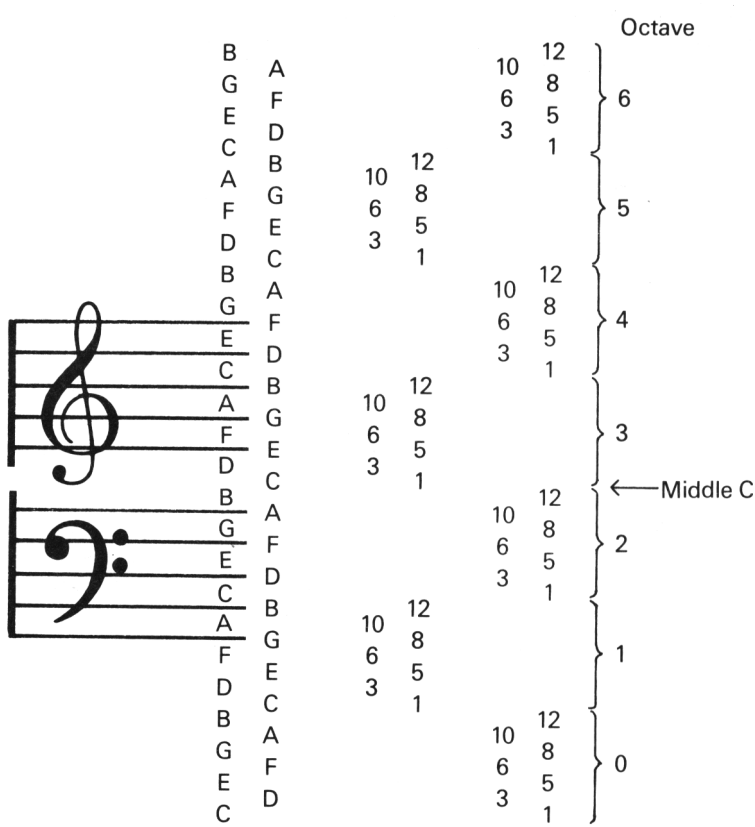
Work out other scales for yourself. You will see that only the key of C Major has no sharp or flat notes.

In the tune we looked at earlier, you will see that the line representing F has the 'sharp' symbol #. This means that notes written on this line should be played as F# rather than F. You may be interested to know that the tune is written in the key of G Major, but this knowledge is not essential in order to convert the music into a computer program.

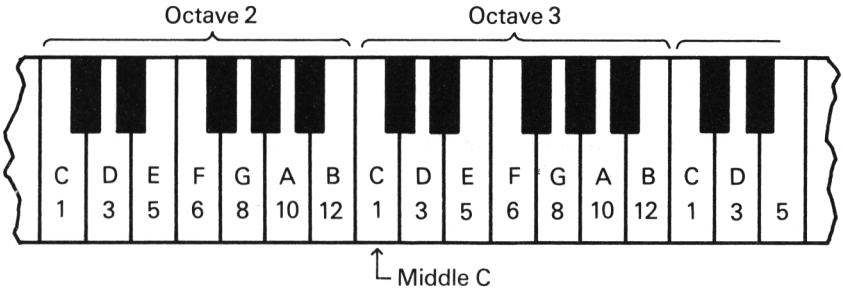
ORIC gives you seven full octaves numbered from Ø (lowest) to 6 (highest). Each octave starts at note C (lowest) and ends at note B (highest). Each octave, as we know, contains twelve semitones. Thus the third parameter (N) of the MUSIC instruction is a number from 1 to 12, each number corresponding to a note:

Note	N Parameter
B (highest)	12
A#	11
A	1Ø
G#	9
G	8
F#	7
F	6
E	5
D#	4
D	3
C#	2
C (lowest)	1

The next diagram gives the ORIC numbers corresponding to positions on, above and below the staves. Where a sharp or flat is written on the line or the space at the start of the line, then all the notes on that line or space should have one added to their numbers (for sharp) or one subtracted (for flat).



Middle C is so called because it is the note C near the middle of a piano keyboard.



If there is a sharp or flat symbol affecting a note on the stave, then the corresponding note in other octaves will also be affected. For example in the tune shown F is set to F#. This means that F will be played as F#, and will have note number 7 instead of 6, in all seven octaves.

In some tunes you will find sharp and flat symbols written in the middle of the line. You will also find the symbol ♮ which converts a note to its 'natural' value if it is normally sharp or flat. Symbols in the middle of the stave are called 'accidentals' and affect any notes on the line or space on which they occur up until next vertical line (that is until the end of the 'bar' in which they appear).

DURATION

We have looked at the pitch of the note. Now we will consider its duration.

It is normal to define a 'single beat' note as ♩ (this is called a crotchet).

A 'half beat' note is ♪ (a quaver).

A 'quarter beat' note is 𝅘𝅥𝅯 (a semiquaver).

Sometimes the tails of the notes are joined together, i.e.

 is two 'half beat' notes.

 is two 'quarter beat' notes.

A 'double beat' note is written 𝅗𝅥 (minim).

A 'four beat' note is written 𝅝 (semibreve).

A dot after the note increases the duration of the note by half as much again – for example

 is 'one and a half beat' note.

 is a 'three beat' note.

Sometimes you will see notes linked by a curved line. If the notes are of the same pitch and next to each other they should be treated as one single note. If they are different pitches, or if there are notes between them, they should be treated as two separate notes.

Finally we come to 'rests'. As the name implies these are pauses in the music. There are two of these:

 – pause for a single beat

 – pause for half a beat.

TIME

The figure after the clef at the start of the music gives the 'time', or the number of beats in a bar. Usually this is $\frac{2}{4}$, $\frac{3}{4}$, $\frac{6}{8}$, or $\frac{4}{4}$, although other values are possible. $\frac{4}{4}$ time is sometimes denoted by C or ¢. For the sake of simplicity take $\frac{3}{4}$ as being the same as $\frac{6}{8}$. There are differences, but we are not delving too deeply into musical theory.

The number of beats in a bar should equal the top figure. Thus:

$\frac{2}{4}$ time has two beats to the bar;

$\frac{3}{4}$ time has three beats to the bar;

$\frac{4}{4}$ time has four beats to the bar.

All complete bars have the same number of beats. This is a useful check to make sure you have not forgotten a note or used an incorrect note length when programming a tune.

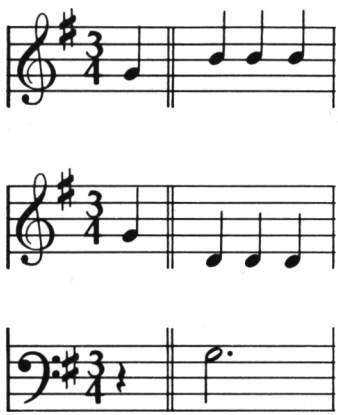
PROGRAMMING METHOD

The first thing I do when changing sheet music into a computer program is to find the shortest note in the music. This could be a quaver or a semiquaver. I then set a variable TIME equal to the time, in hundredths of a second, that I wish the shortest note to last. All the other notes will last for a multiple of this value.

For example if the shortest note were a semiquaver and I set TIME equal to 1Ø, then a full beat would take 4 x 1Ø = 4Ø hundredths or Ø.4 of a second. If the tune were in $\frac{3}{4}$ time a bar would play in 1.2 seconds.

The advantage of this is that by varying the value of TIME, which is set at the beginning of the program, I can speed the tune up or slow it down until I get the speed I want.

Let's look at the tune we started with. We shall take this up to the end of the first full bar. Once you have seen how to change musical notation into program instructions you use the same technique for a whole tune.

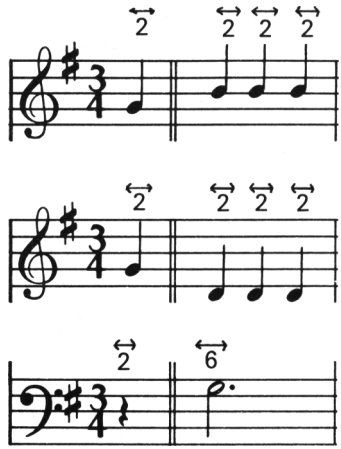


Let's make the two treble voices channels 1 and 2 and the bass voice channel 3. The shortest note in this example is a quaver; so we will make time = 17, giving a time of about one second for a bar.

Thus the first line is:

```
1Ø TIME = 17
```

Split the music up into periods each equal to the shortest note.



In the first of these periods voices 1 and 2 are both playing the note G above middle C, which is octave 3 note 8. Voice 3 is silent.

Thus the next instructions turn on channels 1 and 2 and play the selected notes.

```
2Ø PLAY 3,Ø,Ø,Ø
3Ø MUSIC 1,3,8,5
4Ø MUSIC 2,3,8,5
```

This is going to remain the same for two periods.

Thus the next line is:

```
5Ø WAIT TIME*2
```

Where the next note is the same as the last one, but we want to hear them as two separate notes, we have to turn the appropriate channel off and on again for a short time. Where the next note is different we don't have to do this. Thus in the next period we don't have to switch on channels 1 and 2. We do, however, have to switch on channel 3. Thus the next lines are:

```
6Ø PLAY 7,Ø,Ø,Ø
7Ø MUSIC 1,3,12,5
8Ø MUSIC 2,3,3,5
9Ø MUSIC 3,2,8,5
1ØØ WAIT TIME*2
```

Next we have to switch channels 1 and 2 off for a short time and back on again to get distinct notes. Channel 3 remains on. Thus:

```
11Ø PLAY 4,Ø,Ø,Ø:WAIT 2
12Ø PLAY 7,Ø,Ø,0
13Ø MUSIC 1,3,12,5
14Ø MUSIC 1,3,3,5
15Ø MUSIC 3,2,8,5
16Ø WAIT TIME*2-2
```

In line 16Ø we adjust the length of note to allow for the small delay in line 11Ø.

The next two TIME periods are the same again. Thus:

```
17Ø PLAY 4,Ø,Ø,Ø:WAIT 2
18Ø PLAY 7,Ø,Ø,Ø
19Ø MUSIC 1,3,12,5
2ØØ MUSIC 2,3,3,5
21Ø MUSIC 3,2,8,5
22Ø WAIT TIME*2-2
```

Finally we have to switch all sound off:

```
23Ø PLAY Ø,Ø,Ø,Ø
```

You could go through an entire tune in this fashion, but this would be rather tedious. Let's see if we can avoid repeating all these MUSIC and PLAY instructions.

If a channel is switched off we can set a MUSIC command to play a note on it. The command has no effect. This means that we could have three MUSIC commands at every stage even though not all the channels are played.

Thus the program could use a standard pattern of PLAY, MUSIC and WAIT instructions, and read all the instruction parameters from DATA statements. Thus this program does the same as the last:

```
1Ø TIME=17
2Ø RESTORE
3Ø READ A'TOTAL NUMBER OF NOTES
4Ø FOR K=1 TO A
5Ø READ C1,S1,E1,P1
6Ø READ L1,T1,N1,V1,L2,T2,N2,V2,L3,T3,N3,V3
7Ø READ PERIODS
8Ø READ C2,S2,N2,P2
9Ø WAIT TIME*R-2
1ØØ PLAY S,T,U,V:WAIT 2
11Ø IF KEY$<>"" THEN PLAY Ø,Ø,Ø,Ø:STOP
12Ø NEXT
13Ø DATA 4,3,Ø,Ø,Ø,1,3,8,5,2,3,8,5,Ø,1,Ø,2,3,Ø,Ø,Ø
14Ø DATA 7,Ø,Ø,Ø,1,3,12,5,2,3,3,5,3,2,8,5,2,4,Ø,Ø,Ø
15Ø DATA 7,Ø,Ø,Ø,1,3,12,5,2,3,3,5,3,2,8,5,3,4,Ø,Ø,Ø
16Ø DATA 7,Ø,Ø,Ø,1,3,12,5,2,3,3,5,4,2,8,5,2,Ø,Ø,Ø,Ø
```

This program does not look much shorter than the one before, although it is probably a little quicker to key in. If, however, we were playing 4ØØ notes instead of 4, the advantages of the second method would be rather more obvious. Each note added requires only one extra line of data, plus an alteration in the first data item.

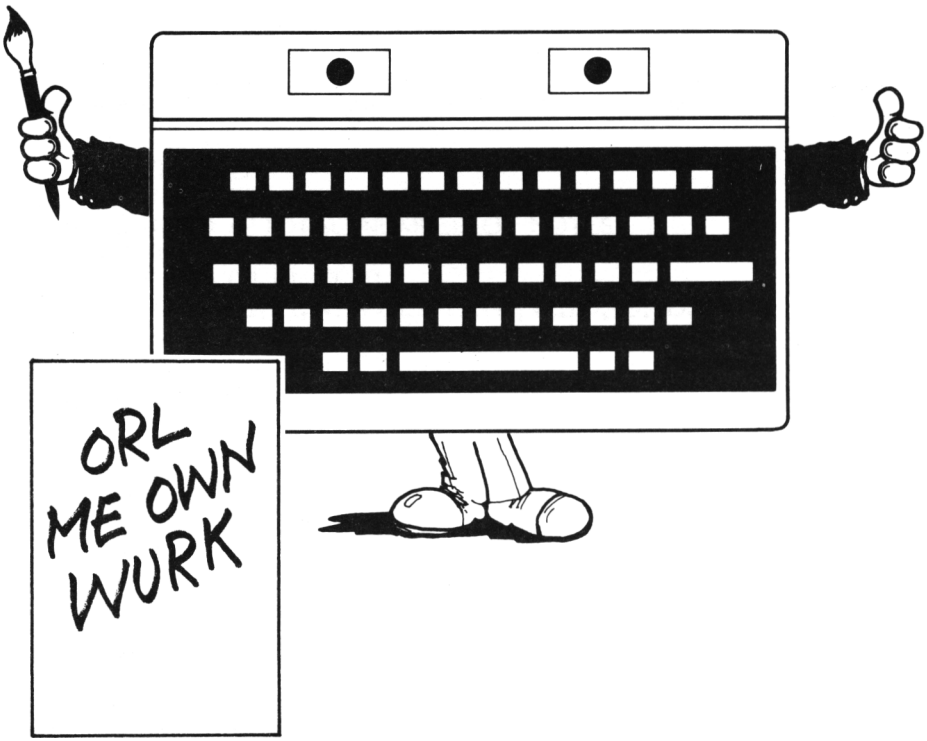
The number of data items could be reduced by putting quantities which remain constant (envelope mode, envelope period, volume, channel select) into lines 5Ø to 1ØØ. As it stands, however, the program is more versatile. It allows us to vary the volume of parts of the tune for all channels or for each channel; it allows us to change the envelope mode, possibly to introduce tremulo; it allows us to switch noise onto any channel for special effects.

Line 11Ø lets us stop the program by pressing any key.

Adapt this program to write tunes of your choice. Experiment with different effects.

CHAPTER 12

ORIC The Artist



LOW RESOLUTION GRAPHICS

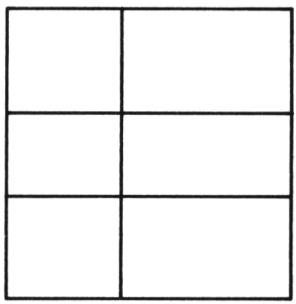
We have already seen that the ESC key followed by I, K, M or O, or CHR$(27) followed by "I", "K", "M" or "O", create the alternate character set. These alternate characters can be used to draw low resolution, or 'chunky' characters on the screen. We saw alternate characters used to print a shape on the screen in Chapters 1,7, and 9.

ALTERNATE CHARACTER SELECTION

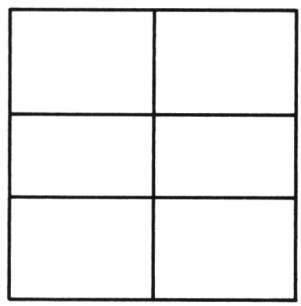
If you are building up a picture using chunky graphics, you can select the characters you require for each part of the picture by referring to Appendix A.

It is, however, often easier to work out what a character should be than to try and find it on a list. Fortunately the procedure to do this is fairly straighforward.

To generate alternate characters, ORIC divides a character position into six. The divisions are not quite equal.

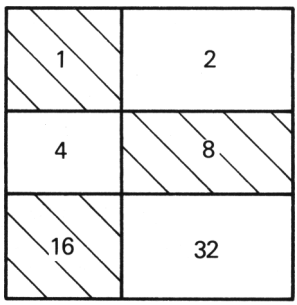


V1.0 Machine

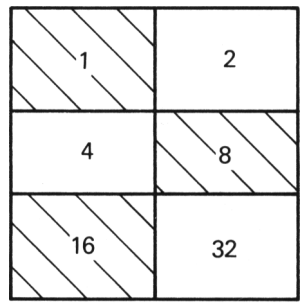


V1.1 Machine

The alternate characters are created by inking in selected areas and leaving others at paper colour. Let's take an example:



V1.0 Machine



V1.1 Machine

I have numbered the areas 1, 2, 4, 8, 16, 32. Can you work out the significance of these numbers? Look at binary notation in Chapter 10 and work out the binary equivalents if you need a clue.

Just follow this simple procedure to generate whatever character you want:

1. Draw the character position, divide it up and number the areas as I have done.

2. Select the areas you want to be inked in.

3. Add up the numbers in the selected areas. In the example I have given, 1 + 8 + 16 = 25

4. Add 32 to this number (ASCII codes of normal characters start at 32). In the example given the result would be 25 + 32 = 57.

The number you end up with will be equal to the ASCII code for the alternate character you want.

To print the chosen shape on the screen we could either use the ASCII code directly:

```
PRINT CHR$(27)"I"CHR$(57)
```

or we could look up Appendix A to find the character which corresponds to that ASCII code. In the example chosen the character '9' has ASCII code 57. Thus we could use:

```
PRINT CHR$(27)"I9"
```

Practise using this procedure. You will find it becomes routine fairly quickly.

When you are printing a number of characters merged together to form a picture you will usually find that it is more convenient to define a string containing all these characters. You may remember we did this in Chapter 9.

LORES 1

It is awkward in the V1.Ø machine to place a character in a selected position on the screen using PRINT. It is much easier to use PLOT.

It would also be more convenient if we did not need to use CHR$(27)"I....." to generate alternate characters.

ORIC allows us to use PLOT and to miss out CHR$'27)"I" by providing a screen display mode known as LORES 1. In LORES 1 all characters are normally printed as alternate characters and do not have to be specified as such.

We looked at LORES Ø in Chapter 7. LORES 1 is similar in many ways, but rather more useful (in my opinion).

To put ORIC into LORES 1 mode, use the instruction LORES 1 either as an immediate command or in a program. This instruction clears the screen and sets the default colours of black paper and white ink.

If you use CLS or CTRL L in LORES 1 the machine goes back into TEXT mode. Clear the screen with the instruction LORES 1.

As with LORES Ø, you can use column Ø in LORES 1 provided you do not wish to change the ink colour.

The next program generates a shape on the screen which could (with a bit of imagination or the wrong pair of spectacles) be taken for a flying saucer. You can move the saucer round the screen using the cursor control keys. Pressing the space bar stops the program.

To move a shape on a screen we print spaces over the shape delete it and then print it again in a position just next to where it was before.

```
10 LORES 1:PRINT CHR$(17)
20 SAUCER$=CHR$(70)+CHR$(87)+CHR$(36)
30 X=18:Y=13
40 : REPEAT
50 : PLOT X,Y,SAUCER$
60 : GET A$
70 : IF ASC(A$)=8 AND X>1 THEN DX=-1
80 : IF ASC(A$)=9 AND X<36 THEN DX=1
90 : IF ASC(A$)11 AND Y>0 THEN DY=-1
100 : IF ASC(A$)=10 AND Y<25 THEN DY=1
110 :PLOT X,Y,"   ":REM THREE SPACES
120 :X=X+DX:Y=Y+DY:DX=0:DY=0
130 :UNTIL ASC(A$)=32
140 CLS:PRINT CHR$(17)
```

Note – this program will work with both V1.1 and V1.0 machines, but as a result does not use the full screen of either. If you have the V1.1 machine you can, if you wish, use PRINT @ instead of PLOT.

In Chapter 7 we saw that we can use PLOT commands to put colour attributes on the screen. We could make the saucer turn red when it enters the right hand side of the screen by adding the line:

```
125: IF X>18 THEN PLOT X-1,Y,1
```

You can print normal characters in LORES 0 if you wish to mix text and graphics. Attribute 8 will switch to the standard set and attribute 9 will switch back to the alternate set. Try:

```
10 LORES 1
20 PLOT 4,13,8:PLOT 5,13,"STANDARD":PLOT 13,13,9
30 PLOT 14,13,"ALTERNATE":PLOT 23,13,8
40 PLOT 24,13,"STANDARD"
```

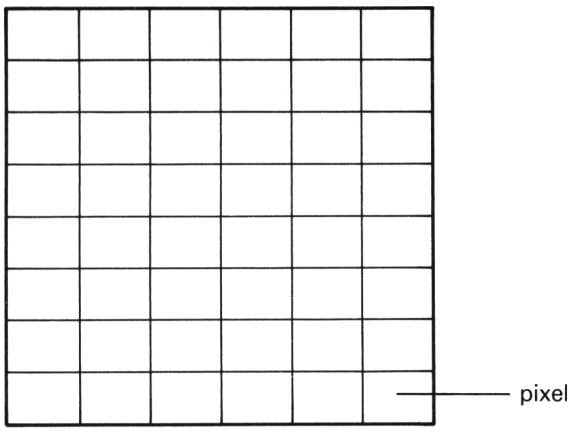
You can define a global ink colour other than white in LORES 1 provided you do not PRINT or PLOT in column 0. Attempting to define a global paper colour has an odd effect. The background colour can be changed a line at a time by PLOTting background attributes (see Chapter 7).

USER DEFINED GRAPHICS

Alternate characters are useful for large chunky shapes, but sometimes much more detail is required. More detail, or higher resolution, may be obtained by 'redefining' characters – that is by changing the shape of a character to the shape you want.

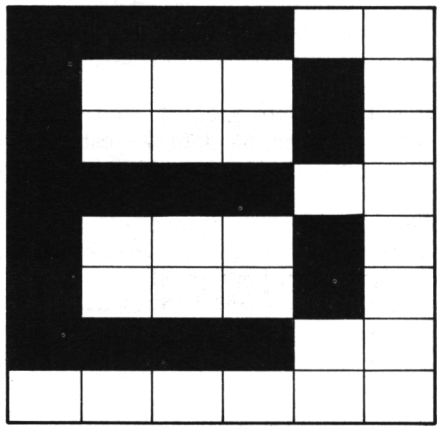
The picture on the television screen controlled by the computer is made up of a large number of dots – just like the pictures in a newspaper. These dots are called picture elements or pixels.

There are 48 pixels in each character position. If you imagine that every character position on the screen is divided into a six by eight grid, then every space in the grid is a pixel.



1 Character space

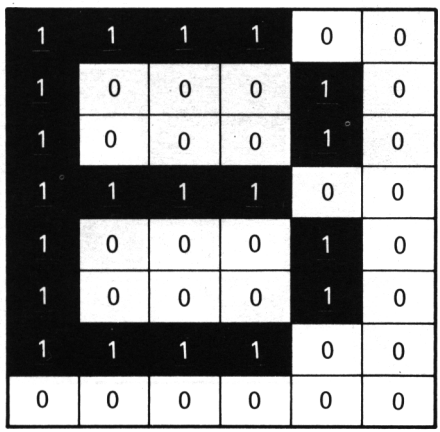
To generate a character, selected pixels are inked in:



The pattern for each character is held in ORIC'S memory. Every character takes up eight memory locations – one for each row of pixels.

Computers hold information as binary numbers. ORIC remembers character shapes by giving every inked pixel a bit value 1 and every uninked pixel a bit value Ø.

Thus the letter B is represented by:



Adding leading zeroes to give 8 bit bytes results in the binary pattern:

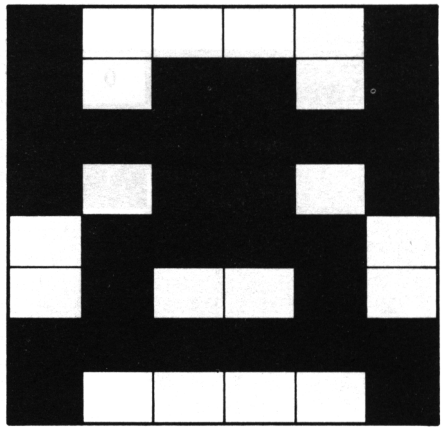
```
ØØ11 11ØØ
ØØ1Ø ØØ1Ø
ØØ1Ø ØØ1Ø
ØØ11 11ØØ
ØØ1Ø ØØ1Ø
ØØ1Ø ØØ1Ø
ØØ11 11ØØ
ØØØØ ØØØØ
```

It is convenient to write this in hexadecimal. This gives

```
#3C
#22
#22
#3C
#22
#22
#3C
#ØØ
```

To define our own graphic shapes we change the numbers held in the memory locations where ORIC stores graphic shapes.

First choose a shape – for example:



Write in 1 where a pixel is inked, Ø where it is not.

1	0	0	0	0	1
	0	1	1	0	1
1	1	1	1	1	1
1	0	1	1	0	1
0	1	1	1	1	0
0	1	0	0	1	0
1		1	1	1	1
1	0	0	0	0	1

This gives you the binary pattern you require – i.e.

```
1Ø ØØØ1
1Ø 11Ø1
11 1111
1Ø 11Ø1
11 1111
Ø1 ØØ1Ø
11 1111
1Ø ØØØ1
```

You could convert these binary numbers to decimal values. It is, however, much easier to use hexadecimal notation.

Thus the pattern becomes:

```
#21
#2D
#3F
#2D
#3F
#12
#3F
#21
```

The next step is to choose a character to change into your required shape. Make this choice carefully – remember that the character will keep its new shape until you either change it again, or press the reset key, or switch the computer off.

If for example you chose to change R, it would make instructions such as PRINT and READ look most peculiar on your listings!

One approach is to alter characters like @ and } , which you do not use much. If, however, you are going to change a number of characters it is probably easier to change lower case letters such as p,q etc. All your instructions and variables use upper case; so this is safe enough provided you don't want lower case messages on the screen and provided you do not use letters contained in 'Ready', 'Searching', 'Loading' and 'Saving'. A third approach is to change the alternate characters and use LORES 1 mode.

In this example we are going to change the letter 'b' to the required shape. To do this we are going to change the memory locations which currently hold the shape of character 'b' so that instead they hold the values needed to make the shape we have chosen. This raises two problems:

1. How do we find out where in memory the shape for character 'b' is held?

2. How do we change the contents of these memory locations?

To find out where the shape of a character is held we use one of two formulae, depending on whether the computer is the 16k or 48k version.

In TEXT and both LORES modes, the address of the first of the 8 memory locations which hold the shape of character 'b' is given by:

46ØØ+8*ASC("b") for 48k machines

or 13312+8*ASC("b") for 16k machines.

To get the start address of any other character shape you use this formula, but put the character required inside the inverted commas. To redefine alternate characters change the numbers 46ØØ and 13312 to 471Ø4 and 14336 respectively.

POKE and PEEK

To change the contents of a memory location use the instruction POKE. The format is

POKE ADDRESS, NUMBER

Where ADDRESS equals the address of the memory location you wish to change, and NUMBER is the value that you wish to put in that address.

PEEK is the opposite of POKE. It lets you find out the number that is stored in a memory location. Thus

PRINT PEEK (14ØØØ)

would print on the screen the number stored in memory address 14ØØØ. PEEKing an address does not alter the contents of memory.

In this case we want to change memory, so that POKE is the instruction we use. We want to POKE the numbers to make our chosen shape into the eight memory locations which at present contain the shape for 'b'.

If you have a 16k machine type in:

```
1Ø A = 13312 + 8 * ASC ("b")
```

If you have a 48k machine type in

```
1Ø A = 46Ø8Ø + 8 * ASC ("b")
```

The rest of the program is the same for both machines:

```
2Ø FOR N = Ø TO 7
3Ø READ B
4Ø POKE A + N,B
5Ø NEXT
1ØØ DATA #21,#2D,#3F,#2D,#3F,#12,#3F,#21
```

RUN this program. Remove the capitals lock and hold the B key down. The character 'b' has been replaced by the shape you defined.

LIST the program. Even on the program listing 'b' has been altered. Pressing the Reset key will restore 'b' to its normal shape.

Note – In the V1.Ø machine the POKE command will not accept a hexadecimal number as the second parameter – i.e. you can't have POKE 16ØØØ,#21. You can have A = #21 : POKE 16ØØØ,A. Both parameters may be hexadecimal in the V1.1 machine.

CARTOON MOVEMENT

You can move a character about the screen by printing a space over the character and then printing the character in an adjacent position. Another method of simulating movement is to print a character on the screen and then overprint it with another character which is only slightly different. The next program illustrates this. If you have a 16k machine change 46Ø8Ø in line 1Ø to 13312.

```
1Ø A=46Ø8Ø+8xASC("p")
2Ø : FOR K=Ø TO 8 STEP 8
3Ø :   FOR N=Ø TO 7
4Ø :   READ B
5Ø :   POKE A+N+K,B
6Ø : NEXT N,K
7Ø CLS
8Ø A$="p q p q":B$="p q p q"
9Ø : REPEAT
1ØØ :PLOT 13,13,A$:WAIT 5Ø
11Ø :PLOT 13,13,B$:WAIT 5Ø
12Ø :UNTIL KEY$<>"":REM NO SPACE
13Ø END'ANY KEY STOPS PROGRAM
14Ø DATA #21,#2D,#3F,#2D,#3F,#12,#3F,#21
15Ø DATA #ØØ,#ØC,#1E,#2D,#3F,#21
```

SCRN

User defined graphics are used a great deal in computer games. Another useful function for such applications is SCRN.

```
SCRN (X,Y)
```

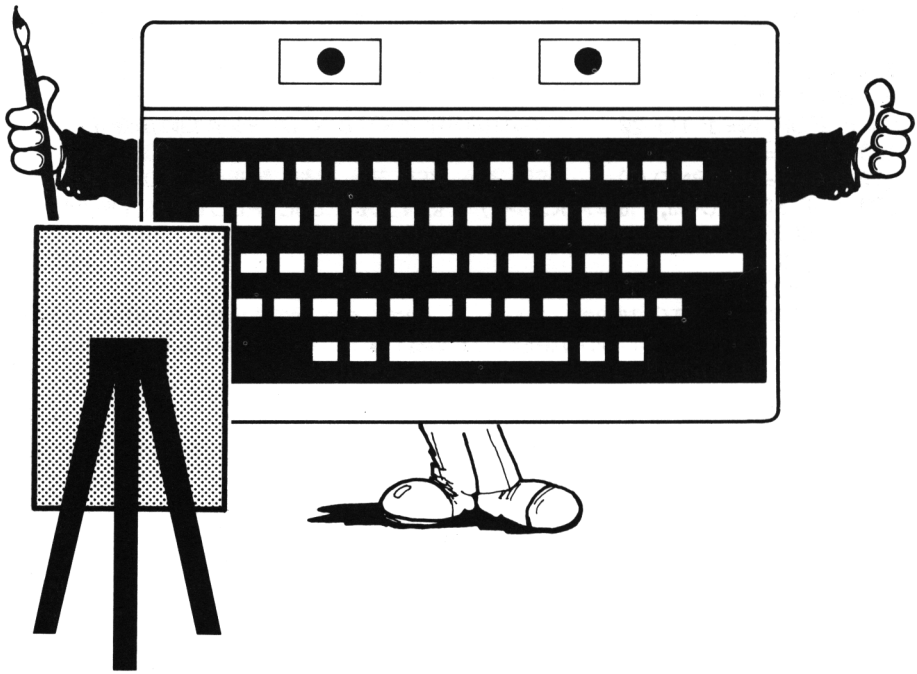
returns the ASCII value of the character at position X,Y on the screen. This is useful if you want to take action when a character which moves about the screen reaches a particular position.

In the next program particles of matter (denoted by M) and antimatter (A) appear at random positions on the screen. If matter and antimatter appear in the same place there is an explosion.

```
10 REPEAT
20 CLS:PAPER 7:INK 4
30 : FOR N=1 TO 20
40 : X%=2+37*RND(1)
50 : Y%=26*RND(1)
60 : PLOT X%,Y%,"M"
70 : NEXT
80 WAIT 50
90 : FOR N=1 TO 20
100 : X%=2+37*RND(1)
110 : Y%=26*RND(1)
120 : IF SCRN(X%,Y%)=ASC("M") THEN 150 ELSE PLOT X%,Y%,"A"
130 : NEXT
140 UNTIL 0
150 EXPLODE
160 : FOR N=0 TO 10
170 : PAPER 1:WAIT 5:PAPER 7:WAIT 5
180 : NEXT
190 INK 0:CLS:END
```

CHAPTER 13

More ORIC Artistry



HIRES

When we looked at user defined graphics we saw that each character position could be divided into 48 pixels. ORIC has a high resolution graphics mode which lets you work with screen pixels instead of screen characters, so that you can draw finer lines and make more detailed pictures.

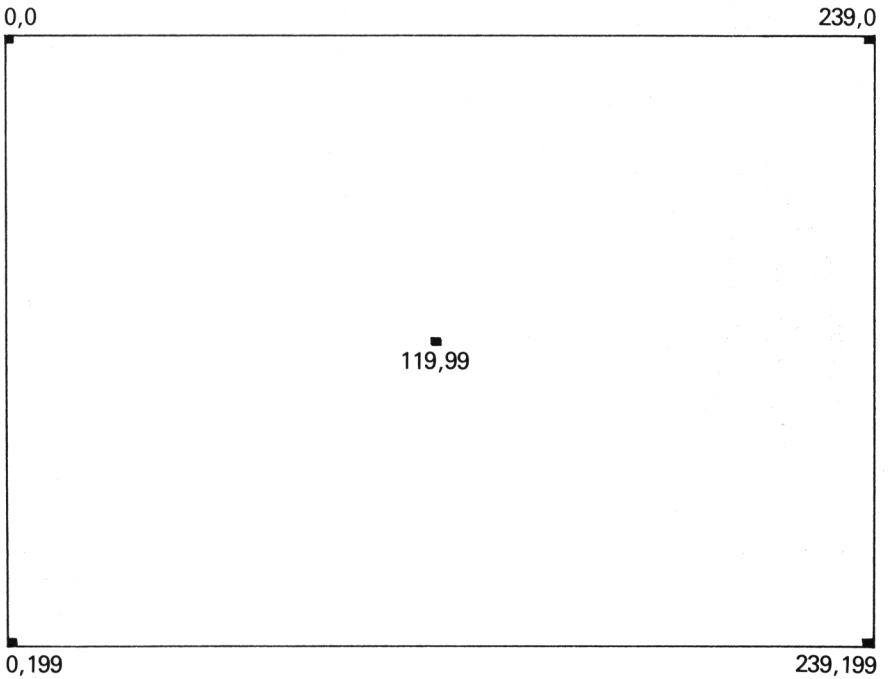
The instruction HIRES puts ORIC into the high resolution mode. HIRES may be used as an immediate command or as a program instruction.

When ORIC goes into high resolution mode the screen turns black, apart from three lines at the bottom which remain at low resolution and can be used for text entry.

HIGH RESOLUTION SCREEN

The high resolution screen is made up of a total of 48000 pixels – 240 across the screen by 200 down the screen. Distance across the screen is called the X coordinate. Distance down the screen is called the Y coordinate.

The next diagram shows the positions of various 'important' points on the screen.



The graphics cursor is a single pixel. Unlike the text cursor it does not flash, as this could spoil your display. The graphics cursor is itself invisible, although it is possible to ink the pixel at which the graphics cursor is currently located.

When the machine goes into high resolution mode, or whenever the HIRES instruction is used, the graphics cursor is positioned at location Ø,Ø.

The instruction HIRES clears the high resolution area of the screen. CLS works only on the low resolution area at the bottom.

CURSET

The CURSET instruction moves the graphics cursor to a specified pixel. The format is

```
CURSET X,Y,FB
```

where X and Y are the coordinates of the pixel, and FB is a number which determines what the CURSET instruction does to the pixel once the cursor gets there.

If FB is Ø the pixel is set to the background (paper) colour.

If FB is 1 the pixel is set to the foreground (ink) colour.

If FB is 2 the pixel colour is 'inverted'. If it was originally in foreground colour it is set to background colour, and vice versa.

If FB is 3 the pixel colour is unchanged.

The next program prints dots at random positions on the screen. If the same position is selected twice the dot disappears.

```
1Ø REM NASTY DISEASE
2Ø HIRES
3Ø REPEAT
4Ø X%=24Ø*RND(1):Y%=2ØØ*RND(1)
5Ø CURSET X%,Y%,2
6Ø UNTIL KEY$ <>"":REM NO SPACE
7Ø TEXT:END'ANY KEY STOPS PROGRAM AND RETURNS
   MACHINE TO TEXT MODE
```

CURMOV

Sometimes you may wish to move the cursor a specified number of pixels in the X and/or Y directions. You could work out the current cursor position (not always easy) and calculate from this the actual address of the new cursor position required. This, however, is rather tedious.

CURMOV does all this for you. You specify the number of pixels in each direction which you wish the cursor to move and ORIC does the arithmetic. The format is:

```
CURMOV X,Y,FB
```

Where X and Y specify the movement relative to the current cursor position and FB is the same as before. In this instruction X and Y may be either positive or negative. The next program demonstrates this:

```
10 HIRES
20 CURSET 69,49,3
30 FOR SIDE=100 TO 20 STEP -10
40 CURMOV SIDE,0,1
50 CURMOV 0,SIDE,1
60 CURMOV -SIDE,0,1
70 CURMOV 0,-SIDE,1
80 CURMOV 5,5,3
90 NEXT
100 GET A$
110 TEXT:END' ANY KEY RETURNS MACHINE TO TEXT MODE
```

DRAW

The DRAW instruction draws a straight line from the current cursor position to a new cursor position. The new cursor position is specified in the same way as for CURMOV, that is to say it is relative to the current cursor position. The format is

```
DRAW X,Y,FB
```

Where X and Y specify the 'difference' in X and Y coordinates between the initial cursor position and the final cursor position. X and Y may be positive or negative.

If FB = 0 the line is drawn in background colour. This is mainly used to delete lines. To ensure total erasure, lines should be deleted in the same direction in which they were drawn.

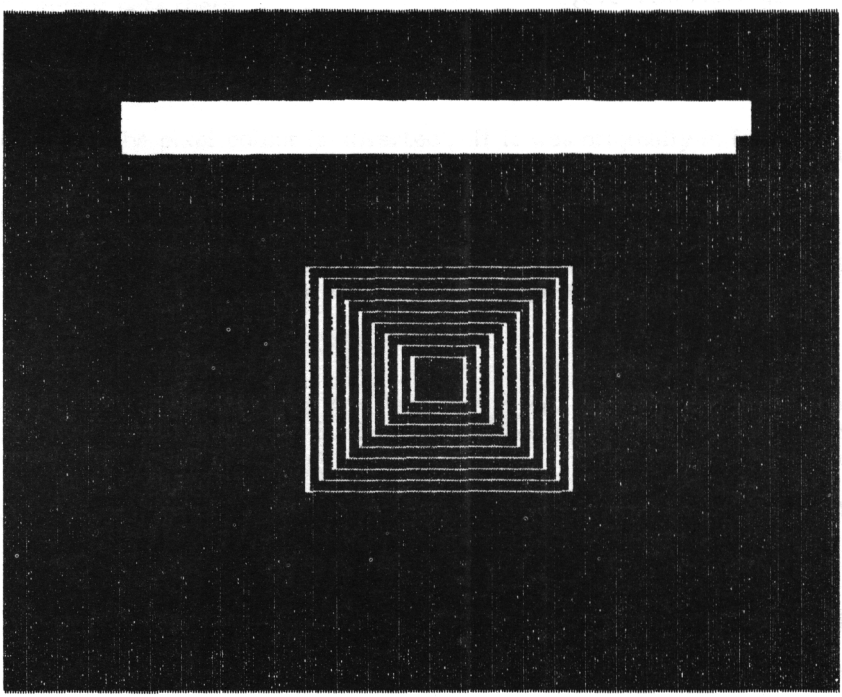
If FB = 1 the line is drawn in foreground colour.

If FB = 2 then any pixels in the line currently in foreground colour are set to background colour; any in background colour are set to foreground colour. Some interesting effects can be created by this feature.

If FB = 3 the cursor moves without affecting the screen display. This is not often used as CURMOV does the same.

We can adapt the previous program to fill in the lines between the dots:

```
1Ø HIRES
2Ø CURSET 69,49,3
3Ø FOR SIDE = 1ØØ TO 2Ø STEP -1
4Ø DRAW SIDE,Ø,1
5Ø DRAW Ø,SIDE,1
6Ø DRAW -SIDE,Ø,1
7Ø DRAW Ø,-SIDE,1
8Ø CURMOV 5,5,3
9Ø NEXT
1ØØ GET A$
11Ø TEXT:END'ANY KEY RETURNS MACHINE TO TEXT MODE
```

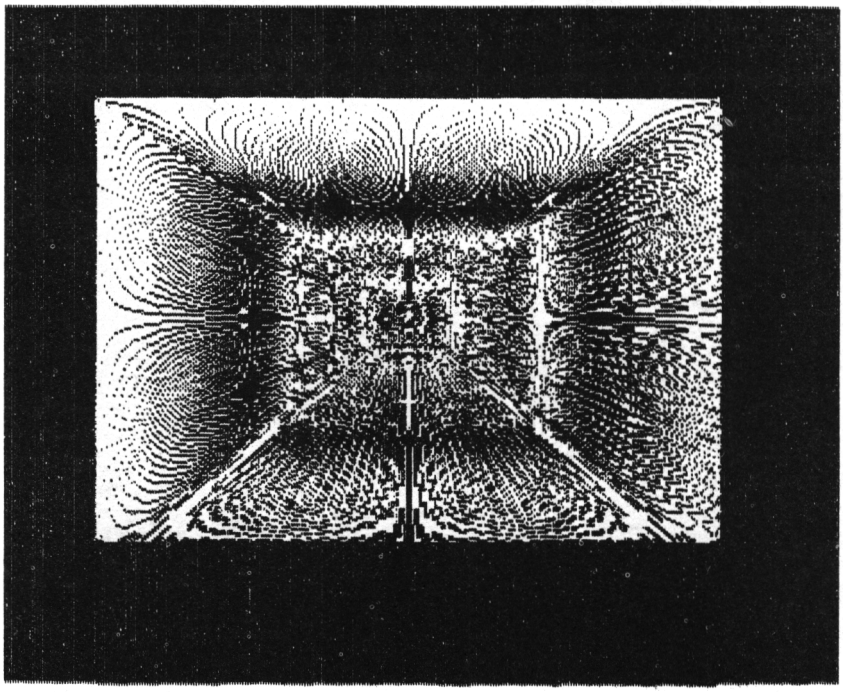


This program can be further amended to delete each square before it draws the next one:

```
10 HIRES
20 REPEAT
30 CURSET 69,49,3
40 : FOR SIDE=100 TO 20 STEP -10
50 :   FOR N=1 TO 0 STEP -1
60 :   DRAW SIDE ,0,N
70 :   DRAW 0,SIDE,N
80 :   DRAW -SIDE,0,N
90 :   DRAW 0,-SIDE,N
100 :   WAIT 50*N
110 :   NEXT N
120 : CURMOV 5,5,3
130 : NEXT SIDE
140 UNTIL KEY$ <>"":REM NO SPACE
150 TEXT:END
```

Using DRAW with FB=2 gives what are known as 'interference fringes' between adjacent lines. Try

```
10 PAPER 0:HIRES:PRINT CHR$(17)
20 FOR X=0 TO 239
30 CURSET X,0,3
40 DRAW 239-2*X,199,2
50 NEXT X
60 FOR Y=0 TO 199
70 CURSET 0,Y,3
80 DRAW 239,199-2*Y,2
90 NEXT Y
100 GET A$
110 PRINT CHR$(17):TEXT:PAPER 7
```



GLOBAL COLOUR IN HIRES MODE

You can use the full high resolution screen area if you wish to draw in white on a black background. You can change the global paper and ink colours, but if you do this you will lose anything drawn on the first two columns on the screen. This would prevent you from inking in any pixel with an X coordinate less than 12.

To illustrate this add the line:

```
95 PAPER 4:INK 3
```

to the previous program.

There are two more interesting points in this program. Firstly the low resolution screen is set to paper colour black before going into HIRES mode to give an all-over black background. White paper colour is restored on re-entering TEXT mode. Secondly the cursor is hidden after the machine goes into HIRES mode, and restored before leaving this mode.

CIRCLE

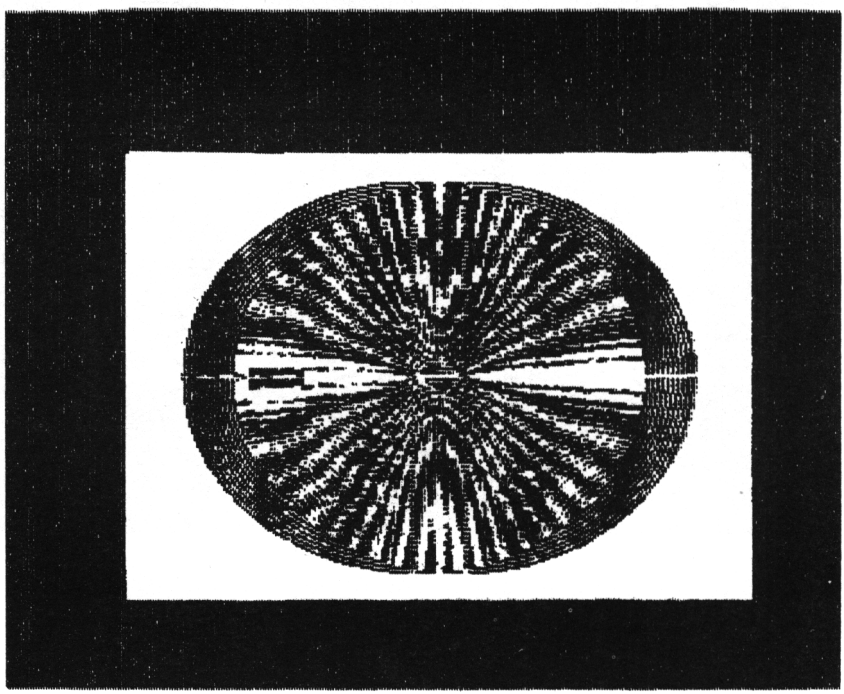
The CIRCLE instruction draws a circle whose centre is the current cursor position. The format is

```
CIRCLE R,FB
```

Where R is the radius and FB is the same as for the DRAW instruction.

You can draw interference patterns with CIRCLE too:

```
1Ø PAPER Ø:HIRES:PRINT CHR$(17)
2Ø PAPER 6:INK 1
3Ø FOR R=2 TO 89 STEP 2
4Ø CURSET 11Ø,1ØØ,3:CIRCLE R,2
5Ø CURSET 13Ø,1ØØ,3:CIRCLE R,2
6Ø NEXT
7Ø GET A$
8Ø PRINT CHR$(17):TEXT:PAPER 7
```

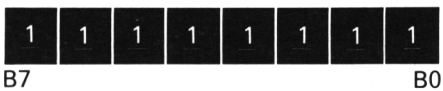


Note – When using CIRCLE, DRAW and CURMOV take care that you do not specify numbers which are outside the limits of the screen. If you try to draw off the screen you will get errors.

PATTERN

ORIC has a special feature which lets you draw dotted lines. This is controlled by an 8 bit store called the **'pattern register'.**

In the pattern register a 1 gives a dot and a Ø a space. When the machine is switched on all bits in the register are set at 1. This gives solid lines:



The binary number stored in the pattern register on power up is therefore:

```
1111 1111
```

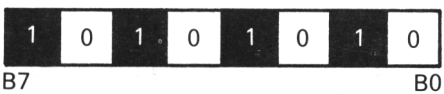
This equals #FF or 255 (decimal).

The contents of the pattern register may be altered by the instruction:

```
PATTERN N
```

where N is a number between Ø and 255 (#ØØ and #FF)

Thus to get a dotted line we set the pattern register to



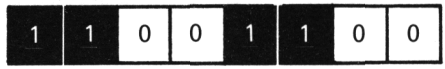
Representing a dot by a 1 and a space by a Ø gives the binary number:

```
1Ø1Ø 1Ø1Ø
```

which is equivalent to #AA.

Try:
```
1Ø HIRES
2Ø PATTERN #AA
3Ø DRAW 239,199,1
4Ø GET A$:TEXT
```

Dashes may be drawn by setting the pattern register to:



This gives 11ØØ 11ØØ(binary) or #CC.

Try:

```
1Ø HIRES
2Ø PATTERN #CC
3Ø CURSET 12Ø,1ØØ,3
4Ø CIRCLE 9Ø,1
5Ø GET A$:TEXT
```

Other PATTERN numbers you could try are:

```
1111 ØØØØ (binary) or #FØ (longer dashes)
1111 1ØØØ (binary) or #F8
1111 1Ø1Ø (binary) or #FA
```

Switching the computer off and on again, or pressing the Reset key, or using the instruction:

```
PATTERN #FF
```

restores the pattern register to its original state.

POINT

The POINT instruction is used to determine whether or not a pixel is INKED.

If the pixel at screen location X,Y is INKed, then POINT (X,Y) =-1.

If the pixel at screen location X,Y is not INKed, then POINT (X,Y) = Ø.

POINT could be used in games programs. For example if you were 'steering' a predefined shape through a maze of dots and lines, POINT could tell you if an obstacle has been or is about to be hit. Another use of POINT is to fill in areas by INKing in pixels until a previously INKed pixel is detected.

The next program demonstrates this:

```
10 HIRES
20 CURSET 20,100,1
30 DRAW 70,-50,1
40 DRAW 0,70,1
50 FOR X=20 TO 89
60 Y=120
70 REPEAT
80 CURSET X,Y,1
90 Y=Y-1
100 UNTIL POINT(X,Y)=-1
110 NEXT
120 GET A$:TEXT
```

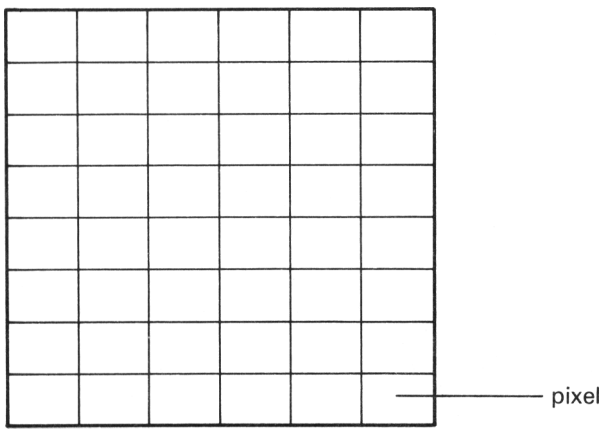
FILL

The method of filling described works well for small or irregular shapes. It is, however, rather slow. The FILL instruction can fill large rectangular areas of the screen very quickly. The format is:

```
FILL Y,X,N
```

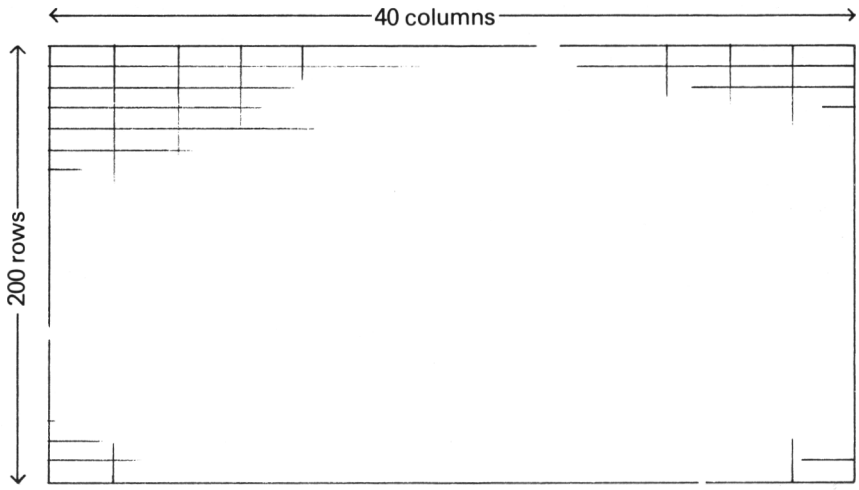
where Y, X and N are all numbers. Y and X define the area to be filled and N defines the binary pattern with which this area is filled. Let's look at how the screen is divided up.

You will recall that each character space is divided into a 6 x 8 array of pixels:



1 Character space

For the purposes of the FILL instruction, six horizontal pixels may be grouped into a block. Thus the screen division becomes



The FILL instruction fills Y rows each X blocks wide with a pattern defined by N. N defines the binary pattern in each block, so that if the entire block is to be filled – i.e. all six pixels in a block inked – N will equal 111111 (binary) or #3F or 63 (decimal).

Try: 1Ø HIRES
2Ø FILL 16,4,#3F

This fills 16 rows (each 1 pixel deep) by 4 columns (each 6 pixels wide) in the top left hand corner of the screen.

Try: 1Ø HIRES
2Ø FILL 16,4,#2A

This fills each block with evenly spaced dots, hence filling the same area as before with vertical lines.

Y is a whole number between 1 and 199. X is a whole number between 1 and 4Ø. For a straightforward filling operation ORIC assumes at least one pixel (the first one) in the block is inked. Thus for normal fill N is a whole number between 1ØØØØØ (binary) and 111111 (binary), or #2Ø and #3F, or 32 and 63.

The start position of the block to be filled is defined by the position of the graphics cursor when the FILL instruction is carried out. The graphics cursor position at the start of the fill defines the top left hand corner of the block. After the fill is complete the graphics cursor position in the V1.1 machine is at the bottom left hand corner of the block, ready to start the next block directly underneath if required. In the V1.Ø machine the position of the graphics cursor after a FILL is not defined. Use CURSET to redefine the cursor position in this case.

Try:
```
1Ø HIRES
2Ø CURSET 1Ø8,92,3
3Ø FILL 16,4,#3F
```

Unless we move the cursor again the next fill operation will start where the last one left off (in the V1.1 machine).

Try:
```
1Ø HIRES
2Ø CURSET 84,52,3
3Ø FILL 32,4,#2A
4Ø FILL 32,4,#3F
5Ø FILL 32,4,#38
```

Add the lines:

```
35 CURSET 116,52,3
45 CURSET 148,52,3
```

in the V1.Ø machine.

HIGH RESOLUTION ATTRIBUTES

If the third number of the FILL instruction is less than 32 (#2Ø) ORIC treats this number as an attribute, which will affect all of the blocks following it on the row on which it is placed.

The attributes associated with each number are similar to those for the low resolution screen – i.e.

Ø – black ink	16 – black paper
1 – red ink	17 – red paper
2 – green ink	18 – green paper
3 – yellow ink	19 – yellow paper
4 – blue ink	2Ø – blue paper
5 – magenta ink	21 – magenta paper
6 – cyan ink	22 – cyan paper
7 – white ink	23 – white paper

Attribute numbers 12 to 15 cause the foreground colours to flash.

You can have all of the background colours and all of the foreground colours within the one character position.

Try:
```
1Ø HIRES
2Ø CURSET 117,96,3
3Ø FILL 8,1,33
4Ø CURSET 1Ø5,96,3
5Ø FOR N=Ø TO 7
6Ø FILL 1,1,N
7Ø NEXT
8Ø CURSET 129,88,3
9Ø FILL 24,1,6
1ØØ REPEAT
11Ø : FOR N=16 TO 23
12Ø : CURSET 111,88,3
13Ø : FILL 24,1,N
14Ø : WAIT 5Ø
15Ø : NEXT
16Ø UNTIL KEY$<>"":REM NO SPACE
17Ø TEXT:END
```

Who said ORIC could use only two colours in high resolution mode!

This looks better on a larger area. The purpose of the last program is to demonstrate how finely colours can be defined.

To see the effect of flashing foreground colours add the lines:

```
95 CURSET 95,96,3
96 FILL 8,1,15
```

POKING ATTRIBUTES

Each block of six pixels on the high resolution screen, and each character space in the low resolution screen, is controlled by one location in memory. By POKEing a number between Ø and 31 into that memory location we can put an attribute directly on to the screen.

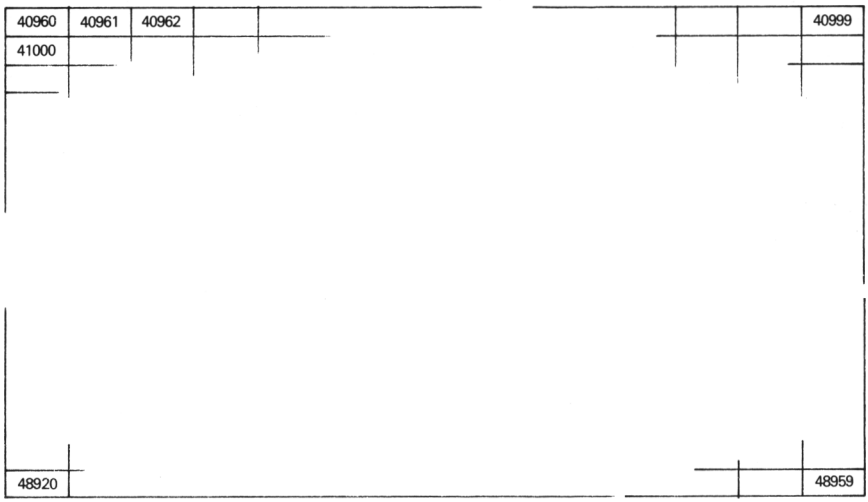
This has no advantages that I can discover in the low resolution mode, where PLOT is easier to use because you don't have to work out the memory location of a character space.

In high resolution mode, however, using POKE saves us from having to move the graphics cursor to select the appropriate block. If you have to put a lot of attributes on to the screen, using POKE could save memory space.

The high resolution screen contains 2ØØ rows each of 4Ø blocks. 8ØØØ memory locations are, therefore, required to control the screen.

These are numbered from 4Ø96Ø to 48959 in the 48k ORIC and from 8192 to 16191 in the 16k machine. The lowest numbered block is in the top left hand corner. The block next to this on the same row has a number greater by one, and so on along the row. The block at the start of the next row has a number which is forty greater than the block above it, and so on.

Thus for the 48k ORIC we have



48K ORIC HIRES screen memory

Hint You may find the start address for high resolution memory easier to remember in hexadecimal. It is #AØØØ for the 48k machine and #2ØØØ for the 16k machine.

CHAR

You put characters on to the low resolution screen using PRINT, PLOT and INPUT. In HIRES mode, however, INPUT and PRINT work only on the low resolution screen at the bottom (and PRINT usually causes a scroll of this screen). PLOT causes a 'DISP TYPE MISMATCH ERROR' message in this mode.

To put characters on to the high resolution screen we use the instruction CHAR. This has the format

```
CHAR A,M,FB
```

where A,M and FB are numbers.

A is a whole number between 32 and 128 and equals the ASCII code of the character to be printed.

M equals Ø for a standard character and 1 for an alternate character.

FB equals Ø if the character is in foreground colour and 1 if it is in background colour. If FB=2 then the pixels currently in foreground colour are set to background colour and vice versa. This may be used to build up composite characters. If FB=3 the instruction has no effect.

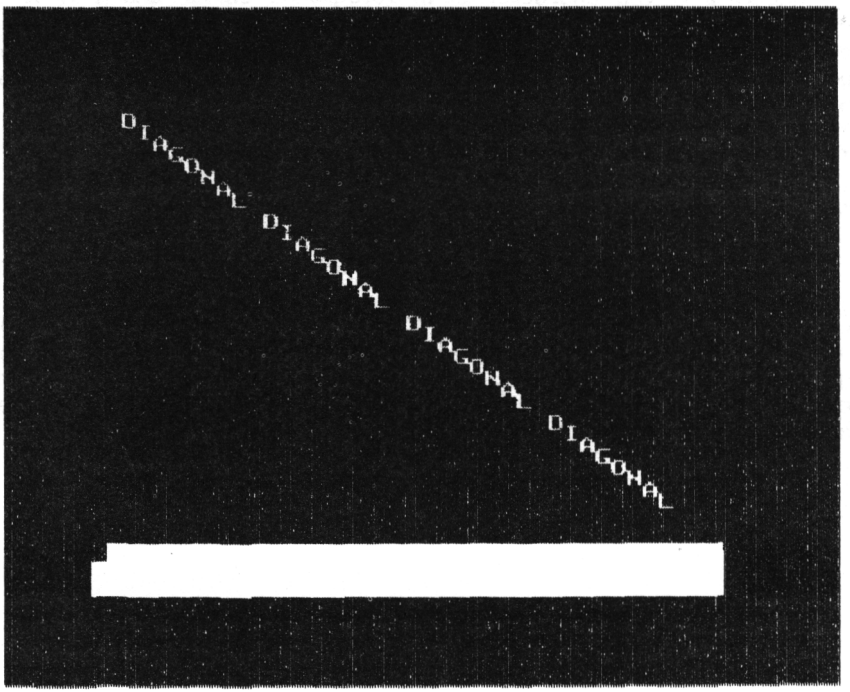
Try: HIRES:CHAR 65,Ø,1

```
HIRES:CHAR 65,1,1
```

```
HIRES:CHAR 127,Ø,1:CHAR 65,Ø,Ø
```

CHAR puts only one character at a time on the screen. The position of the character on the screen is determined by the position of the graphics cursor. The CHAR instruction does not move the graphics cursor; so you have to move it with CURMOV or CURSET to get a string of characters.

```
Try: 1Ø HIRES
     2Ø CURSET 12,8,3
     3Ø FOR N=1 TO 4
     4Ø : FOR K=1 TO 9
     5Ø : READ A
     6Ø : CHAR A,Ø,1
     7Ø : CURMOV 6,5,3
     8Ø : NEXT
     9Ø RESTORE
     1ØØ NEXT
     11Ø REPEAT
     12Ø : FOR N=4Ø96Ø TO 48962 STEP 4Ø
     13Ø : REM 8192 TO 16152 IN 16k ORIC
     14Ø : POKE N,INT(7*RND(1))+1
     15Ø : NEXT
     16Ø UNTIL KEY$<>"":REM NO SPACE
     17Ø TEXT:END
     18Ø DATA 68,73,65,71,79,78,65,76,32
```



USER DEFINED CHARACTERS

If you compare the memory locations used for the screen in high resolution mode with those used for the character set in the low resolution modes you will see that the two overlap. It follows therefore that ORIC stores the characters somewhere else in HIRES mode.

In fact the memory set aside for the normal character set starts at location 38912 (location 6144 for 16k machine) and for the alternate character set at 40836 (7168) in HIRES mode.

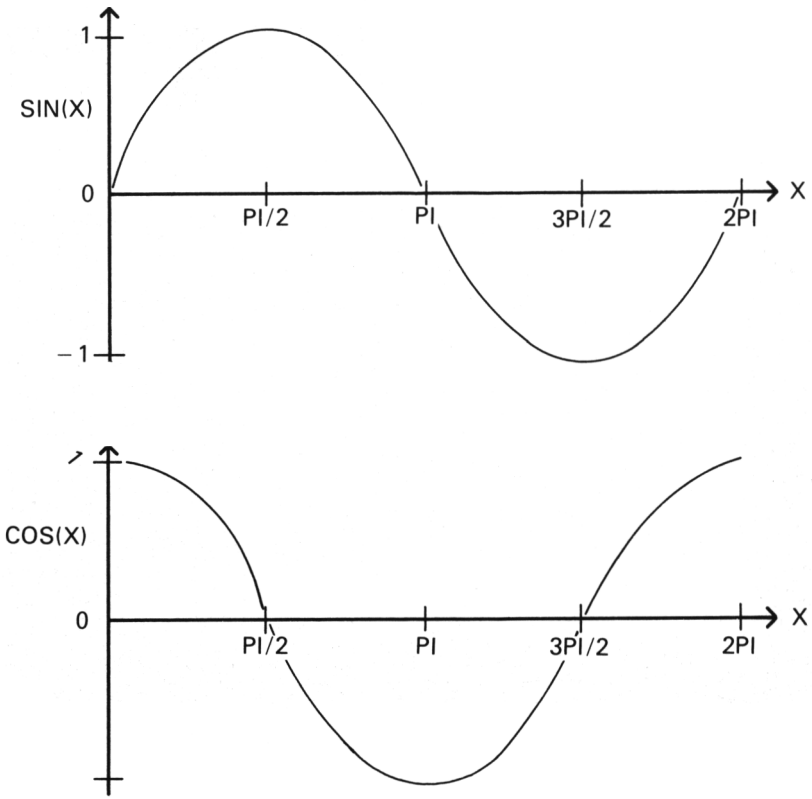
It is, however, rather inconvenient to have to remember two sets of locations. If you redefine a character in a low resolution mode it will remain redefined in high resolution mode. Therefore, although you can redefine characters in HIRES mode, it is better (I think) to do all characters definition before going into this mode.

DRAWING CURVES

It is useful to be able to draw curves on the screen, both to build up pictures for graphics displays and to provide curved paths for graphics characters to follow in games programs.

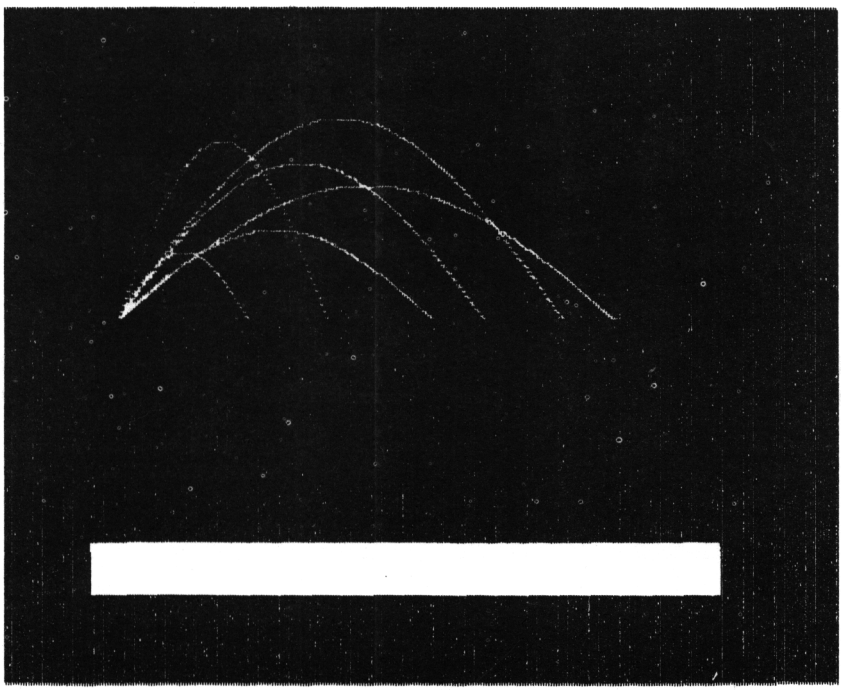
Curves are described using mathematical functions, so that, unfortunately, I shall have to bring in a little maths. If this puts you off, please ignore the rest of this chapter.

Two useful functions for describing curves are SIN and COS. These look like



Try:
```
10 HIRES
20 DEF FNCURVE(X)=B*SIN(X*PI/A)
30 REPEAT
40 INPUT"LENGTH";A
50 INPUT"HEIGHT";B
60 : FOR X=1 TO A
70 : CURSET 10+X,100-FNCURVE(X),1
80 : NEXT
90 UNTIL KEY$<>"":TEXT:END
```

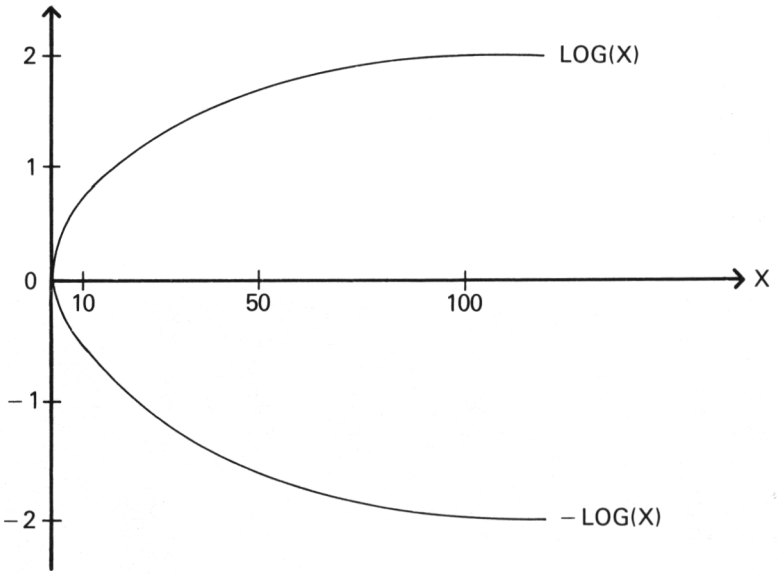
Heights between 20 and 90 and lengths between 30 and 220 will give a set of trajectories.



Try the effect of changing line 20 to:

```
30 DEF FNCURVE(X)=B*X*SIN(X*8*PI/A)/A
```

LOG(X) could also be useful.



This could give the trajectory of a U.F.O. taking off. -LOG(X) could give the trajectory of a U.F.O. descending quickly and then hovering near the ground.

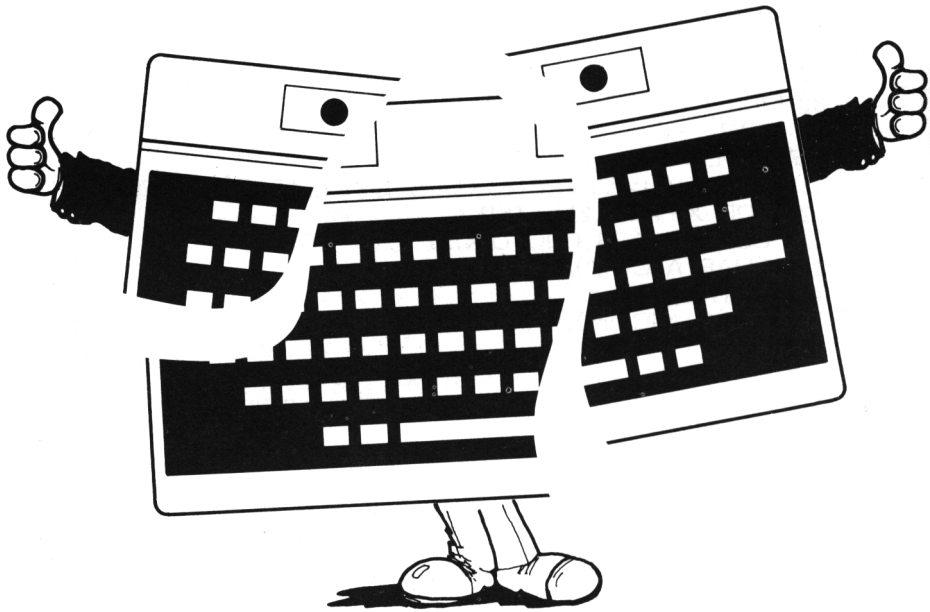
For example:

```
1Ø FOR N=1 TO 7
2Ø READ A
3Ø POKE 46Ø8Ø+ASC("b")+N,A
4Ø REM 13312 IN 16k ORIC
5Ø NEXT
6Ø HIRES
7Ø FOR N=Ø TO 199
8Ø FILL 1,1,INT(7*RND(1))+1
9Ø NEXT
1ØØ REPEAT
11Ø IF RND(1)>Ø.5 THEN GOSUB 1ØØØ ELSE GOSUB 2ØØØ
12Ø FOR X=1 TO 211 STEP 6
13Ø CURSET X+11,FNUFO(X),3
14Ø CHAR ASC("b"),Ø,1
15Ø WAIT 1Ø
16Ø CHAR ASC("b"),Ø,Ø
17Ø NEXT
18Ø UNTIL KEYS <>"":TEXT:END
1ØØØ DEF FNUFO(X)=100+X*6Ø*SIN(X*8*PI/21Ø)/21Ø
1Ø1Ø RETURN
2ØØØ DEF FNUFO(X)=LOG(4*X)*6Ø
2Ø1Ø RETURN
3ØØØ DATA #38,#1C,#Ø1,#Ø7,#Ø7,#ØE,#1C,#38
```

Make sure you type this program in and RUN it initially in TEXT mode. It will take a few seconds before anything happens. I hope it is worth the wait.

CHAPTER 14

Bits 'n Pieces



DATA HANDLING

We can use DATA statements to hold data items in a program. If, however, we wish to do anything with such items it is better to put them in an array.

The next program demonstrates how a two dimensional string array can be loaded with names and telephone numbers from the keyboard and how names starting with any selected letter, along with the corresponding telephone number, can be printed on the screen.

```
1Ø PRINT CHR$(12)
2Ø DIM A$(2Ø,1)
3Ø FOR N=Ø,TO 2Ø
4Ø INPUT"NAME";A$(N,Ø)
5Ø IF A$(N,Ø)="EOF" THEN 9Ø
6Ø INPUT "TELEPHONE NUMBER";A$(N,1)
7Ø NEXT
8Ø REM ############################
9Ø PRINT CHR$(12)
1ØØ INPUT"FIRST LETTER OF NAME";F$:PRINT
11Ø FOR N=Ø TO 2Ø
12Ø IF A$(N,Ø)="EOF" THEN 1ØØ
13Ø IF LEFT$(A$(N,Ø),1)=F$ THEN PRINT A$(N,Ø)
   "'S NUMBER IS "A$(N,1):PRINT
14Ø NEXT:END
```

FILE HANDLING – STORE AND RECALL

It is useful to be able to store the data we have loaded into an array on to tape, so that we can use it again in the same or other programs. This can be done in the V1.1 machine provided the array has been dimensioned.

Note – If you are going to store an array onto tape it must be dimensioned, even if it has less than eleven elements. Remember that the DIM command clears all array elements – so dimension the array before you load it.

The information in an array is stored on tape in what is known as a **file.** The first step in creating a file is to choose a name for it. Here we are going to save array A$ in a file called NAMES.

Break from the program by listing out all the names or by using CTRL C. Type in:

```
STORE A$,"NAMES"
```

or `STORE A$,"NAMES",S`

depending on whether you are saving at normal or slow speed.

Switch your cassette recorder on to record and press ENTER.

The message:

```
Saving ... NAMES S
```

should appear at the top of the screen. When the information in A$ has been saved into file NAMES the 'Ready' message will appear. Rewind the tape to the start of the file.

The information held in A$ is now stored on tape in a file called NAMES. This information can be loaded into any two dimensional string array provided that this array has been dimensioned and provided that it is as big as or bigger than A$.

We shall now see how to retrieve the information from tape. Clear program memory and enter the program.

```
10 DIM B$(20,1)
20 CLS
30 PRINT"REWIND YOUR TAPE TO START OF FILE"
40 PRINT"AND THEN PLAY TAPE"
```

Line 50 will depend on whether you have saved your file at normal or slow speed. It will be either:

```
    50 RECALL B$,"NAMES"
or  50 RECALL B$ ,"NAMES",S
```

The rest of the program is:

```
60 PRINT CHR$(12)
70 INPUT"FIRST DIGIT OF TELEPHONE NUMBER"; D$
80 PRINT
90 FOR N=1 TO 20
100 IF B$(N,0)="EOF" THEN 70
110 IF LEFT$(B$(N,1,),1)=D$ THEN GOSUB 150
120 NEXT
130 END
140 REM SUBROUTINE
150 PRINT"TELEPHONE NUMBER",B$(N,1)
160 PRINT"NAME",,,B$(N,0)
170 PRINT
180 RETURN
```

RUN this program. Line 50 will RECALL the information stored in the file NAMES and will place it in the array B$. This information is used in lines 60 to 180. Note the messages which appear at the top of the screen when you are RECALLing a file. These are very similar to the messages you get when LOADing a program.

B$ was given the same dimensions as A$ in the previous program. The file could also be loaded into a larger array. If, for example, line 1Ø were changed to:

```
1Ø DIM B$(3Ø,1)
```

the program would still work. Elements B$(21,Ø) to B$(3Ø,Ø) and B$(21,1) to B$(3Ø,1) would contain nothing and could be loaded from the program if required. You can therefore load part of an array from a file and the rest from a program, or even by immediate commands. You may then store the resulting array as another file.

You can also store real and integer arrays in files. For example the programs:

```
1Ø REM FILE CREATION
2Ø DIM C(12)
3Ø FOR N=Ø TO 12
4Ø READ A
5Ø C(N)=A
6Ø NEXT
7Ø CLS
8Ø PRINT"START RECORDING"
9Ø WAIT 2ØØ
1ØØ STORE C,"REAL",S:END
11Ø DATA 1,2,1.5,6,8,16.3,14,3,4.6,8,2,Ø,5
```

and

```
1Ø REM FILE RETRIEVAL
2Ø DIM D(14)
3Ø CLS
4Ø PRINT"REWIND TAPE AND PLAY"
5Ø RECALL D,REAL,S
6Ø D(13)=42
7Ø D(14)=6.6
8Ø FOR N=Ø TO 14
9Ø PRINT D(N),
1ØØ NEXT
11Ø END
```

will store a thirteen element real array in tape file "REAL", will recall that file and store it in a fifteen element array, will load the two blank elements of that array and will print out the array contents.

These two programs may be adapted to work with integer arrays. This is left as an exercise for the reader.

When RECALL is used and a file is loaded into a single element array, then that array must be dimensioned to have at least eleven elements. Otherwise an 'OUT OF MEMORY' error will occur.

The message 'Errors found' may appear on the screen when a file is being recalled. This does not necessarily mean that the retrieved information is incorrect, but it is safer in this situation to either check the information or recall the file again.

GRAB

In a program which uses a lot of memory for arrays and strings, or in any other large program which runs only in a low resolution mode, you will find GRAB a useful instruction. GRAB lets you use memory space which would otherwise be set aside for the high resolution screen. This gives you over seven thousand extra memory locations. GRAB cannot be used if peripherals (such as the MODEM) which affect the machine's 'memory map' are attached.

RELEASE

RELEASE cancels out the GRAB instruction and lets you use HIRES mode. Anything stored in the memory locations made available by GRAB will be lost if RELEASE is used. GRAB and RELEASE must, therefore, be used with care.

SAVING SCREEN DISPLAYS

You can save a specified section or 'block' of memory on to tape by using a special form of the CSAVE instruction. This has the format:

```
CSAVE"....",Aaddr1,Eaddr2
```

for normal speed and

```
CSAVE"....",Aaddr1, Eaddr2,S
```

for slow speed.

The numbers 'addr1' and 'addr2' are the addresses of the first and the final memory locations respectively of the block you wish to save.

Building up a screen display sometimes requires a large program and takes a lot of time. One use of this special CSAVE instruction is to save to tape the contents of the memory locations which control the screen display. When this information is loaded back into the computer at a later date the picture saved will appear again on the screen. There is no need to save the program which generated the display.

In the 16k ORIC the low resolution screen is controlled by memory locations #3B8Ø to #3FEØ (15232 to 16352). Therefore to save the low resolution screen display to tape on the 16k machine we would use:

```
CSAVE "SCREEN1",A#3B8Ø,E#3FEØ
```

or, if the lower speed is used:

```
CSAVE "SCREEN1",A#3B8Ø,E#3FEØ,S
```

To load the display back into the machine we would first ensure the machine is in a low resolution mode and then use:

```
CLOAD "SCREEN1",A#3B8Ø,E#3FEØ
```

or

```
CLOAD "SCREEN1",A#3B8Ø,E#3FEØ,S
```

depending on the speed at which the information was saved.

In the 48k ORIC the low resolution screen is controlled by memory locations #BB8Ø to #BFEØ (48,ØØØ to 4912Ø).

The high resolution screen in the 16k ORIC is controlled by memory locations #2ØØØ to #3FEØ (8192 to 16352), and in the 48k ORIC by memory locations #AØØØ to #BFEØ (4Ø96Ø to 4912Ø).

You may use either decimal or hexadecimal numbers with these commands. Please make sure your machine is in the correct mode before you load the screen display and load it at the same speed at which it was saved.

SAVING CHARACTER SETS

If you have redefined a number of characters in a program it is convenient to be able to save the character set and load it into the machine whenever you wish to use the program. This saves having to redefine the character set each time the program is used.

The same technique is used to save character sets as is used to save screen displays. In the low resolution modes the standard character set for the 16k ORIC is held in memory locations #34ØØ to #38ØØ (13312 to 14336) and the alternate character set in memory locations #38ØØ to #3B8Ø (14336 to 15232).

The corresponding memory locations in the 48k ORIC are #B4ØØ to #B8ØØ (46Ø8Ø to 471Ø4) and the standard character set and #B8ØØ to #BFEØ (471Ø4 to 4912Ø) for the alternate character set.

It is less usual for the character set to be saved and loaded in high resolution mode. Should you wish to do this the relevant memory locations are #18ØØ to #1CØØ (6144 to 7186) for the standard character set and #1CØØ to #2ØØØ (7186 to 8192) for the alternate character set in the 16k ORIC. In the 48k ORIC the memory locations are #98ØØ to #9CØØ (38912 to 39936) for the standard character set and #9CØØ to AØØØ (39936 to 4Ø96Ø) for the alternate character set.

TRUE and FALSE

TRUE and FALSE are numbers. TRUE equals -1 and FALSE = Ø. Their purpose is to make some programs a little easier to read.

For example

```
1ØØ  IF POINT(X,Y)=TRUE  THEN PRINT"INKED PIXEL"
```

and

```
1Ø  REPEAT
------------
1ØØ  UNTIL FALSE
```

POP

When a subroutine is called the computer stores the address of the instruction to which control is returned when the subroutine is complete in a memory location called the 'top of the stack'.

This is rather technical. What it means is that ORIC has to know where to return to when it meets a RETURN instruction. When one subroutine calls another then the return addresses for both subroutines are held on stack, with the return address of the most recently called subroutine on the top.

If you use a GOTO instruction to jump out of a subroutine, instead of exiting via the RETURN instruction, then the return address for that subroutine is not removed from the top of stack. This could cause an error the next time a RETURN instruction is met.

To prevent this we 'POP' any unwanted return addresses out of the stack using the instruction POP.

If you don't quite follow this explanation don't worry. It is a bit complicated. Use the instruction POP whenever you jump out of a subroutine rather than leaving via the RETURN instruction. It could be used when an error is deleted within a subroutine, resulting in a jump to a standard error routine. The subroutine return address is POPed.

In general it is considered good programming practice to leave subroutines only via RETURN instructions, so that the use of POP should be kept to a minimum. Nevertheless the facility is there if you wish to use it.

POS

POS returns the position of the text cursor on the screen. This is a useful feature in (say) programs for 'word processing' where if a word typed in at the end of a line is too long it is taken to the next line.

In the V1.Ø machine

```
A=POS
```

sets the variable A (or any other specified variable) equal to a number between Ø and 39 depending on the horizontal cursor position on the screen. Ø represents the left hand edge.

In the V1.1 machine

```
A=POS(Ø)
```

is the same as A=POS(1) on the V1.Ø machine, and

```
A=POS(1)
```

sets A equal to a number between Ø and 255 depending on the position of the pen of the ORIC printer (see the next Chapter).

PULL

PULL is rather similar to POS. It allows you to jump out of a nested REPEAT ... UNTIL loop using GOTO and be returned back into the original loop by the next UNTIL instruction you encounter.

RUN the program:

```
1Ø TRON
2Ø REPEAT
3Ø:   REPEAT
4Ø:   GOTO 6Ø
5Ø:   UNTIL Ø
6Ø F=F+1
7Ø PRINT F
8Ø UNTIL F=1Ø
9Ø END
```

and note the order of instruction execution. Add the line:

```
35 PULL
```

RUN the program again and see how PULL changes this order.

The use of PULL should be kept to a minimum. Like POP it could be used before (or after) a jump out of a loop to an error routine.

ARRAY AND STRING CORRUPTION

If you are keying in a very long program, or if you are switching between TEXT and HIRES modes, there is a risk that some strings or arrays may be corrupted. This is because the area of memory set aside for BASIC programs overlaps the area used to store strings and arrays.

HIMEM is used to alter the amount of memory that can be occupied by BASIC programs, so this corruption can be avoided by resetting HIMEM to #97FF.

Use the instruction:

HIMEM #97FF (48k) or HIMEM #17FF (16k)

near the start of any program where strings or arrays are used to hold information.

We shall meet HIMEM again the the next chapter.

INPUT/OUTPUT DEVICES

At the time of writing there are very few peripheral devices available other than monitors, televisions, cassette recorders and printers, all of which have special sockets or 'ports'. In order to select any address in memory to read data from or write data to, ORIC puts electrical signals on 16 wires known as the address lines. Once the memory location is selected data is written to or read from it by putting electrical signals on 8 wires known as the data lines.

ORIC treats an I/O (Input/Output) device in much the same way as it treats an area of memory. It addresses the device and then either reads information from it or sends information to it.

Some I/O devices require to be serviced urgently. These generate a signal called an interrupt which cause the computer to stop what it is doing and deal with the device immediately.

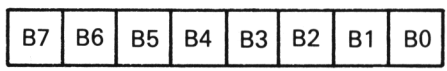
All sixteen address lines, all eight data lines, an interrupt request (IRQ) line and a ROM disable line ROMDIS are brought out of the machine at the Bus Expansion Port. Other 'system signals' available at that port are MAP, $\phi 2$, I/O, R/W, RESET, and I/O control. Unless you are an electronics engineer or computer systems specialist you do not need to worry about the functions of these. It is sufficient to know that sufficient signals are readily available at this port to allow almost any peripheral device to be controlled by the machine. Some devices such as disc controllers will require extensions to the software, but most are likely to be controlled by simple PEEK and POKE instructions.

ALL ABOUT ATTRIBUTES

Throughout the book we have been using attributes to control the background and foreground colours in screen displays. I have taken a rather pragmatic approach where attributes are concerned, concentrating more on what to do rather than why it is done.

This explanation is for those who want to know the 'why' as well as the 'what'. It is not essential reading. You can use attributes quite well by following simple procedures for plotting, printing or poking them on to the screen without going deeply into the explanation.

An attribute is a number which is contained in the memory block which controls the screen. We can write numbers in several ways. Let's write the number in a memory location controlling the screen as a binary number. Each memory location holds eight bits – BØ to B7.



The number is identified as an attribute if bits B5 and B6 are both zero.

Thus the numbers Ø to 31 (decimal) – ØØØØ ØØØØ to ØØØ1 1111 (binary) – are identified as attributes when held in memory locations which control the screens.

128 to 159 (decimal) – 1ØØØ ØØØØ to 1ØØ1 1111 (binary) are also attribute numbers. We shall come to them shortly.

In the low resolution screen a single memory location controls a character position. We saw in Chapter 12 how the number in the character position (assuming it is between 32 and 127) causes the computer to look at the numbers in other memory locations to get the shape of the character in that position whether it is a standard, alternate or user defined character.

However the numbers which define the character shape are not put directly into the memory location which controls that position on the screen. Thus whatever their value these numbers cannot be attributes. An attribute must be put directly in a memory location dedicated to screen control.

Thus in the TEXT or LORES screen we can have only one attribute per character position.

In low resolution we can put an attribute directly into the required memory location using POKE or PLOT. If we use POKE we have to work out exactly which memory location controls the character position where we wish to put the attribute. Using PLOT lets us specify this character position by its X and Y coordinates on the screen.

We can also put an attribute on to the screen using

```
PRINT CHR$(27)"....".
```

What happens here is that the escape character CHR$(27) alters the ASCII code held in the memory location following it by setting bit B6 of that ASCII code to zero.

Let's take an example:

```
PRINT CHR$(27)"A"
```

The ASCII code for A is 65 (decimal) or #41. In binary notation this is

```
Ø1ØØ ØØØ1
```

Changing bit B6 to a zero gives

```
ØØØØ ØØØ1
```

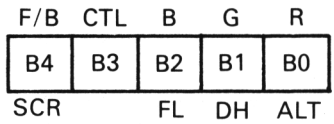
which is the attribute code '1'.

Thus: PRINT @X,Y;CHR$(27)"A"

is the same as

```
PLOT X+1,Y,1
```

We now know that an attribute is identified by zeroes in bits B5 and B6. The effect the attribute has is determined by bits BØ to B4. Let's look at these in more detail:



Consider bit B3 first. This is a control bit. When B3 is Ø the rest of the bits are fairly easy to understand. They control the colour.

B3=Ø

BØ controls the RED

B1 controls the GREEN

B2 controls the BLUE

B4 switches between background and foreground

All colour on your television screen is made up from a combination of red, green and blue light. Mixing light beams is not quite the same as mixing paint. For example mixing red and green light gives yellow.

Thus if bit B4 = Ø (foreground), B3 = Ø, B2 = Ø (no blue light), B1 = 1 (green light), BØ = 1 (red light) we get a yellow foreground attribute. Changing B4 to 1 will give a yellow background attribute. Converting this to decimal gives code 3 for a yellow foreground and code 19 for a yellow background attribute.

When bit B3 = 1 things are slightly more complicated. We have to consider bit B4 as a control bit also.

B3 = 1, B4 = Ø

BØ = Ø for standard and 1 for alternate characters

B1 = Ø for single height and 1 for double height characters

B2 = Ø for flashing and 1 for steady characters

In this case BØ and B1 are effective only on the low resolution screen.

Thus for example B4 = Ø, B3 = 1, B2 = 1, B1 = Ø and BØ = 1 gives flashing, single height, alternate characters. Converting to decimal gives code 13 for this attribute.

B3 = 1, B4 = 1

These attributes are to be avoided, especially if B1 = Ø (6Ø Hz attributes). They affect screen synchronization and their affects are unpredictable.

A foreground or a flashing/double height/alternate attribute affects the row of character positions following it but causes a blank space in the screen character position corresponding to the memory location holding the attribute.

A background attribute not only affects the character positions in the line following it but also causes a change of background colour in the character position corresponding to the memory location holding the attribute.

Thus when specifying background and foreground colours in a line of screen display it is usual to specify first the background then the foreground attribute. This is the case in both high and low resolution screens.

Finally let's get back to bit B7. This is the 'inverse' bit. It inverts the RED, GREEN and BLUE signals controlled by bits BØ to B2 when bit B3 is zero.

Taking the example of the yellow foreground attribute (code 3) – this has RED and GREEN on and BLUE off. If we set bit B7 the code becomes 131 and the colours are inverted. RED and GREEN are off, BLUE is on. Thus the yellow foreground attribute becomes a blue foreground attribute.

A great deal can be done using attributes – enough in fact to fill another book! Some spectacular effects can be generated by printing shapes or messages on the screen using the same ink and paper colour so that they are initially invisible and then changing attributes on the screen. As attributes are numbers they can be held in memory along with other numbers defining a shape and the whole block of memory (known as a **graphics macro**) transferred to locations controlling the HIRES screen. Suddenly changing backgroung attributes, again usually in HIRES mode, can cause 'death rays' to shoot very quickly across the screen in games programs.

There is not space in an introductory book to go into this in detail. I hope you will now investigate what can be done with attributes yourself and that my suggestions will be of assistance.

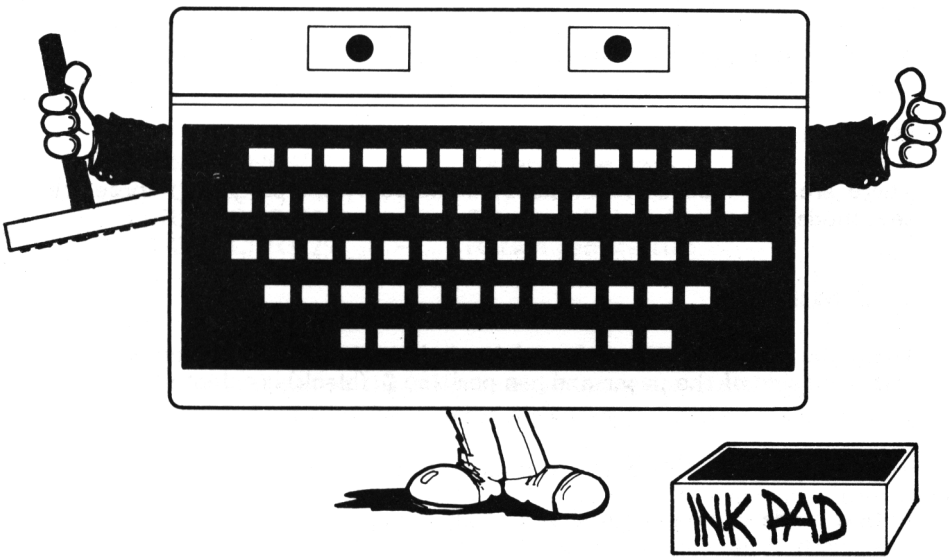
In the HIRES screen each character position is controlled by eight separate memory locations. The pixels on the screen are directly controlled by bits B5 to BØ of the numbers in the memory locations.

You may remember that when using the FILL instruction to ink in pixels on the screen the first pixel always had to be inked. In other words B5 had to be a 1. The reason for this is that otherwise the pattern is interpreted as an attribute if the FILL starts at the start of a character position. Remember the condition for an attribute – B5 and B6 are Ø.

When you put a character on to the HIRES screen using CHAR then the numbers defining the character are put into the memory which controls the screen. This is because in HIRES you have a direct 1 pixel = 1 bit correspondence. Why then are these numbers, some of which are below 32 (decimal) not treated as attributes?. The answer is that the CHAR instruction **always** sets bit B6 of any number it puts in screen control memory location to 1. Thus you can use CHAR safely for its intended purpose (clever!).

CHAPTER 15

Get Into Print



THE ORIC PRINTER

The ORIC printer uses four tiny ballpoint pens to write or draw on a roll of plain paper. You can select black, blue, red or green ink under program control.

The printing speed is 12 characters per second. Plotting (drawing) speed is 52 mm per second in the horizontal direction (x-axis) and 73 mm per second in the vertical direction (y axis).

PEN POSITIONS

The normal pen positions are:

Ø	black
1	blue
2	green
3	red

You can if you wish put a different coloured pen in any position. I shall assume that the pens are as normal.

PRINT MODES

There are two modes which may be selected by the program. These are text mode and graphics mode.

TEXT MODE

When you first switch on, the printer is in text mode. The pen is at the left hand side of the paper and pen position Ø (black) is selected.

CHARACTERS PER LINE

In text mode you can normally print 8Ø characters on a line. To select a line width of 4Ø characters use the instruction:

POKE 49,53 (V1.Ø) or POKE 598,4Ø (V1.1)

To return to 8Ø characters per line use:

POKE 49,93 (V1.Ø) or POKE 598,8Ø (V1.1)

LLIST

LLIST lists the program on the printer, in the same way as LIST lists on the screen.

LPRINT

LPRINT prints messages on the printer. Its operation is very similar to PRINT.

Try: LPRINT 26
LPRINT"HELLO FOLKS!"
LPRINT CHR$(65)

LPRINT may be used with TAB, comma and apostrophe separators in the same way as PRINT. LPRINT @ is not, however, allowed.

CONTROL CODES

LPRINT may be used with various control codes. I shall list these briefly and then look at them in more detail.

LPRINT CHR$(8) – Backspace
LPRINT CHR$(11) – Reverse line feed
LPRINT CHR$(17) – Select text mode
LPRINT CHR$(18) – Select graphics mode
LPRINT CHR$(29) – Rotate pen position

CHR$(8)

LPRINT CHR$(8) causes the print head to move one position backwards. This is useful for creating 'composite' characters. For example:

```
10 REM CONTINENTAL SEVEN
20 LPRINT 7;
30 LPRINT CHR$(8);
40 LPRINT CHR$(45)'MINUS
```

CHR$(11)

LPRINT CHR$(11) effectively moves the print head up one row (in fact it moves the paper down one row). This is very useful for 'superscription' – i.e. creating bold characters by overprinting.

Try:
```
10 A$="PRINT BOLD"
20 LPRINT A$
30 LPRINT CHR$(11);
40 LPRINT A$
```

CHR$(17)

LPRINT CHR$(17) puts the printer into text mode. It has no effect if the printer is already in text mode.

CHR$(18)

LPRINT CHR$(18) puts the printer into graphics mode. We shall look at this in more detail shortly.

CHR$(29)

LPRINT CHR$(29) rotates the pen position. If the pen is at position Ø (black), then LPRINT CHR$(29) changes it to position 1 (blue). The next LPRINT CHR$(29) will change to position 2 (green) and so on.

Try
```
1Ø FOR N=Ø TO 4
2Ø READ A$
3Ø LPRINT A$
4Ø LPRINT CHR$(29);
5Ø NEXT
6Ø DATA BLACK,BLUE,RED,GREEN, AND BACK TO BLACK
```

Note. LLIST, and LPRINT CHR$(8), CHR$(11) and CHR$(29) work only in text mode.

GRAPHICS MODE

The instruction LPRINT CHR$(18) puts the printer into graphics mode. In this mode the print 'map' is considered as being divided into steps each Ø.2 mm long. There are 48Ø of these steps in the horizontal direction and any number (in theory) in the vertical position (up or down). For practical purposes we will limit vertical movement between -999 (999 steps down the paper) and +999 (999 steps up the paper) from the origin. If we want to extend this range we can reposition the origin.

When the printer is put into graphics mode the origin is positioned at the left hand side of the drawing area and pen position Ø is selected. Thus if we drew a line 48Ø steps long in the horizontal direction (x=+48Ø) we would get a black horizontal line right across the drawing area.

IMPLICIT COMMANDS

To control the printer in graphics mode, ORIC uses what are known as 'implicit commands'. These are all based on the LPRINT instruction, which must be followed by a string. The first character in the string determines what the command is. This first character is not, itself, printed.

Compare this with PRINT CHR$(27) (see Chapter 7) where the first character in the string determines the display format.

As before I shall first list these commands and then discuss them in more detail. Use the manual paper feed on the printer so that you have fresh paper for each demonstration.

LPRINT"A" – Return to text mode
LPRINT"Cn" – Select ink colour (n is Ø ,1 ,2 or 3)
LPRINT"D" – Draw line from current pen position to absolute point specified
LPRINT"H" – Return to origin
LPRINT"I" – Set origin
LPRINT"J" – Draw line from current pen position to relative point specified
LPRINT"Ln" – Specifies the line type (n is Ø to 15).
LPRINT"M" – Move pen to absolute point specified without drawing a line
LPRINT"P" – Print characters without returning to text mode
LPRINT"Sn" – Specifies the size of characters (n is Ø to 63)
LPRINT"Qn" – Specifies the direction of print (n is Ø, 1, 2 or 3)
LPRINT"R" – Move pen to relative point specified without drawing a line
LPRINT"X" – Draw an axis from the present pen position and mark it at specified intervals.

LPRINT"A"

This moves the pen to the left of the drawing area without drawing a line and puts the printer back into text mode. LPRINT CHR$(17) may also be used to put the printer into text mode.

LPRINT"Cn"

When the printer is put into graphics mode the pen is in position Ø (black). If no colour is specified black lines will be drawn. LPRINT"Cn" specifies the colour:

n = Ø – black
n = 1 – blue
n = 2 – green
n = 3 – red

The chosen colour will remain until another colour is selected.

```
Try: 1Ø LPRINT CHR$(18)
     2Ø LPRINT"C3"
     3Ø LPRINT"PROSES ARE RED"
     4Ø LPRINT"C1"
     5Ø LPRINT"PVIOLETS ARE BLUE"
     6Ø LPRINT"C2"
     7Ø LPRINT"PGRASS IS GREEN"
     8Ø LPRINT"CØ"
     9Ø LPRINT"PWHAT A BORING PROGRAM"
     1ØØ LPRINT"A"
     11Ø REM RETURN TO TEXT MODE
```

LPRINT"D"

LPRINT"D,X,Y" draws a line from the current pen position to point X,Y where X is the number of steps from the origin in the X direction and Y is the number of steps from the origin in the Y direction. Absolute coordinates were discussed in chapter 13.

X is in the range -48Ø to +48Ø (depending on the position of the origin). Y is in the range -999 to +999.

A number of points may be specified one after the other. Try:

```
1Ø LPRINT CHR$(18)
2Ø LPRINT"DØ,3ØØ,3ØØ,3ØØ,3ØØ,Ø,Ø,Ø"
3Ø LPRINT"A"
```

This draws a line from the origin (Ø,Ø) to point (Ø,3ØØ), then from this point to (3ØØ,3ØØ), then to (3ØØ,Ø) and then back to the origin. Thus a square of side length 3ØØ steps is drawn.

LPRINT"H"

This returns the pen to the origin without drawing a line. Try

```
1Ø LPRINT CHR$(18)
2Ø LPRINT"C3"
3Ø FOR Y=Ø TO 3ØØ STEP 3Ø
4Ø A$="D3ØØ,"+STR$(Y)
5Ø FRE(""):REM NO SPACE
6Ø LPRINT A$
7Ø LPRINT"H"
8Ø NEXT
9Ø LPRINT"A"
```

LPRINT"I"

This changes the origin to the current pen position

```
Try: 1Ø LPRINT CHR$(18)
     2Ø FOR N=Ø TO 2
     3Ø GOSUB 1ØØ
     4Ø LPRINT"I"
     5Ø NEXT
     6Ø LPRINT"A"
     1ØØ LPRINT"DØ,1ØØ,1ØØ,1ØØ,1ØØ,Ø,Ø,Ø,1ØØ,1ØØ"
     12Ø RETURN
```

LPRINT"J"

This works like LPRINT"D" except that X and Y are stated relative to the current pen position. Relative coordinates are discussed in Chapter 13.

```
Try: 1Ø LPRINT CHR$(18)
     2Ø LPRINT"JØ,3ØØ,3ØØ,Ø,Ø,-3ØØ,-3ØØ,Ø"
     3Ø LPRINT"A"
```

LPRINT"Ln"

This specifies the type of line drawn. Solid lines are drawn unless otherwise specified. There are 16 types of line including solid lines:

n	Line type
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

Try adding:

```
15 LPRINT"L6"
```

to the previous program.

LPRINT"M"

This works in the same way as LPRINT"D" except that no line is drawn when the pen moves.

LPRINT"P"

This allows characters to be printed while the printer remains in graphics mode. The characters must be in a string. We saw LPRINT"P" in use earlier when we looked at LPRINT"Cn".

LPRINT"Sn"

This lets you set the size of the character. You can have 64 different character sizes; n takes a value between Ø (smallest and 63 (largest).

The number of characters you get on a line depends on the character size. The smallest characters (n=Ø) will be printed 8Ø to the line. The largest characters (n=63) will require a whole line each.

The next program will work out how many characters of a specified size may be put on one line.

```
1Ø INPUT"CHARACTER SIZE (Ø-63)";SIZE
2Ø NUM=INT(8Ø/(SIZE+1))
3Ø PRINT"SIZE "SIZE"GIVES "NUM"CHARACTERS PER LINE"
```

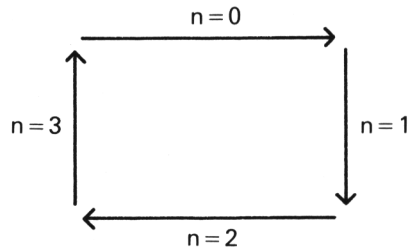
The printer specification guarantees the accuracy of this formula only for a size range Ø to 15 (8Ø to 5 characters per line). I have found it gives a useful guide throughout the range.

Try:
```
1Ø LPRINT CHR$(18)
2Ø LPRINT"C3"
3Ø LPRINT"S38"
4Ø LPRINT"PAN"
5Ø LPRINT"C2"
6Ø LPRINT"S18"
7Ø LPRINT"PORIC"
8Ø LPRINT"CØ"
9Ø LPRINT"SØ"
1ØØ A$="EXTRAVAGANZA "
11Ø B$=A$+A$+A$+A$+A$+A$: FRE("")
12Ø LPRINT"P"B$
13Ø LPRINT"A"
```

If no print SIZE is specified SØ is assumed.

LPRINT"Qn"

Normally ORIC will print characters from left to right. LPRINT"Qn", where n may be Ø, 1, 2 or 3, enables you to specify other directions – i.e. top to bottom, bottom to top and right to left.




```
Try: 1Ø LPRINT CHR$(18)
     2Ø LPRINT "S7"
     3Ø LPRINT"M96,Ø"
     4Ø A$="SQUARE"
     5Ø FOR N=Ø TO 3
     6Ø LPRINT"Q"STR$(N)
     7Ø LPRINT"P"A$
     8Ø NEXT
     9Ø LPRINT "SØ":LPRINT"CØ":LPRINT"A"
```

LPRINT"R"

This works in the same way as LPRINT"J" except that no line is drawn when the pen moves.

LPRINT"X,A,S,I"

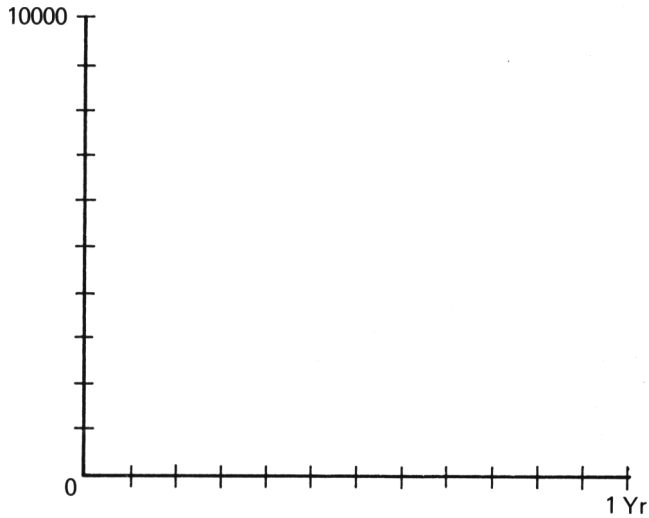
This is used to draw axes for graphs. This is a bit mathematical – if maths turns you off then ignore this instruction.

A, S and I are numbers. A is either Ø or 1 and selects the axis to be drawn:

A = Ø gives the vertical axis (Y-axis)
A = 1 gives the horizontal axis (X-axis)

Axes are normally marked off into sections by small dashes called 'grads'. For example, suppose the X axis represented time – say one year. This axis could be marked off in 12 equal sections each representing 1 month.

Suppose the Y axis represented total sales of a product. The total axis length could represent 1ØØØØØ sales and this could be marked of in ten equal sections each representing 1ØØØØ sales.



S represents the size, in 2mm steps, of each section or 'graduation' on the axis. It is a number between -999 and +999. If S is negative the axis is drawn downwards (Y axis) or to the left (X axis).

I specifies the number of sections and is a number between 1 and 255.

Try:
```
1Ø LPRINT CHR$(18)
2Ø LPRINT"M24Ø,-27Ø"
3Ø LPRINT"I"
4Ø LPRINT"X1,3Ø,7"
5Ø LPRINT"H"
6Ø LPRINT"X1,-3Ø,7"
7Ø LPRINT"H"
8Ø LPRINT"XØ,25,1Ø"
9Ø LPRINT"H"
1ØØ LPRINT"XØ,-25,1Ø"
11Ø LPRINT"A"
```

OTHER PRINTERS

ORIC drives printers by what is called a parallel or 'Centronics' output. It can work with any printer which has a Centronics input, provided that you can obtain a suitable connector. Although the Centronics 'interface' uses standard signals, these signals may be brought in on different pins on different printers.

You don't have to worry about this if you use the ORIC printer, as this printer has a connector made especially for the ORIC output.

PIN CONNECTIONS

If you have another printer you may be able to purchase a suitable connector from a computer retailer. If not you will have to make one up – or find a tame electronics expert to do this for you.

The signals from the ORIC output are:

Pin	Signal
1	STB
3	DØ
5	D1
7	D2
9	D3
11	D4
13	D5
15	D6
17	D7
19	ACK

All even numbered pins are connected to Ground.

The input signal pin numbers for any parallel printer should be given in the documentation supplied with it.

For example the connections for the Microline 80 printer are:

Pin	Signal
1	STB
2	D0
3	D1
4	D2
5	D3
6	D4
7	D5
8	D6
9	D7
10	ACK

Pins 19 to 30 are connected to ground.

THE TRS-80 COLOUR PRINTER

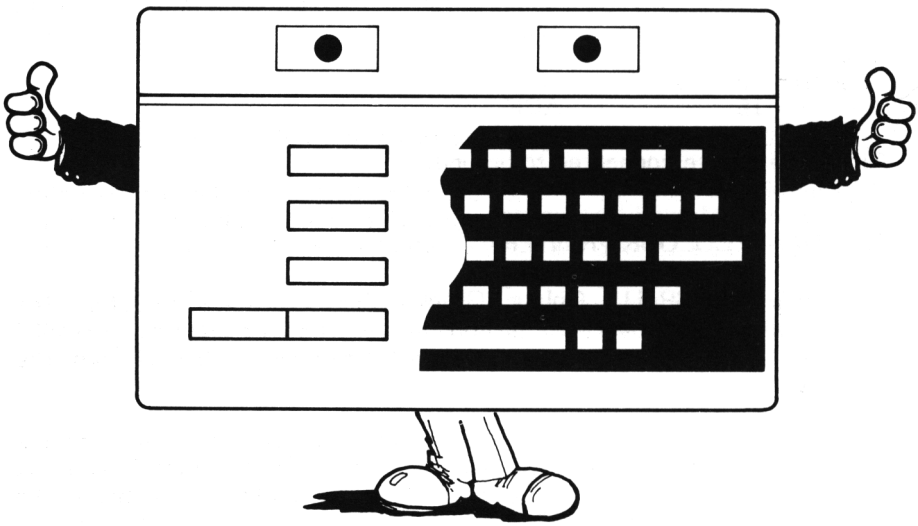
The TRS-80 CGP-115 Colour Graphics Printer from Radio Shack is very similar to the ORIC printer. Assuming you have the correct connector, any program you write to control the ORIC printer will also control this device.

POS(1)

In the V1.1 machine POS(1) returns the horizontal position of the print cursor. See Chapter 14.

CHAPTER 16

Machine Code



MACHINE CODE

Computers work only with ones and zeroes. When you tell ORIC to do something, using words which are comprehensible to humans (PRINT, DRAW, ZAP), these commands are translated into strings of binary numbers which ORIC can understand. There is a group of programs held permanently in computer memory which does this. This group of programs is known as the Operating System.

The binary numbers which the computer understands are called 'machine code'. If you know which numbers do what you can communicate directly with the machine in its own language.

Hexadecimal notation is a convenient shorthand method of writing binary numbers. When we are typing machine code into the machine we normally use hexadecimal format.

To help us remember what each number does we give each one a name, or mnemonic.

Thus (for example):

1110 1000 (=#E8) is given the mnemonic INX

A program to change these mnemonics back into machine code is called an assembler. A program to change a high level (English-like) language such as BASIC into machine code is called a compiler or an interpreter.

The programs held within a computer's operating system to perform the latter function are usually interpretive. A compiler converts a BASIC program to machine code before that program is loaded into the computer.

Programming in machine code is not difficult, although some machine code programmers pretend otherwise. In the main it is merely tedious. Machine code programmers do have to know a little more about the innards of the machine.

THE 6502 MICROPROCESSOR

ORIC is controlled by the 6502 microprocessor system. The microprocessor is working all the time while the machine is switched on. It reads binary numbers from memory, interprets these numbers as instructions, data or memory addresses, and carries out operations on the basis of this information.

To do this the microprocessor uses special number stores known as registers. These registers are not part of the machine's main memory and are identified by name rather than by address number.

They are:

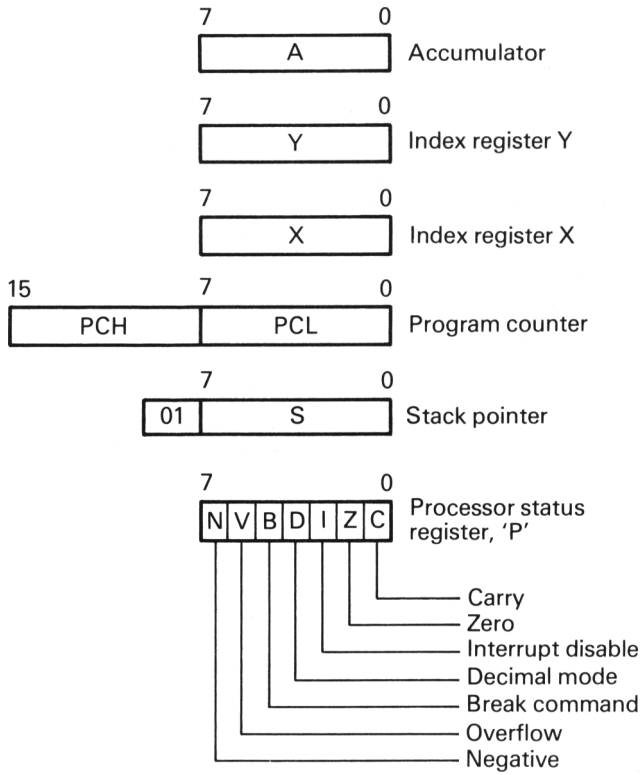
The Accumulator (A)

The Index Registers (X and Y)

The Program Counter (PC)

The Stack Pointer (S)

The Status Register (P)



The accumulator is involved in most of the arithmetic and logical operations carried out in the machine. The number resulting from these operations is usually stored in the accumulator.

The index registers are used to store values for counting, timing and 'indexing' (identifying a memory address or series of addresses with reference to a 'base' address).

The Program Counter holds the memory address of the instruction currently being carried out.

The Stack Pointer holds the address of the start of a section of memory known as the Stack. The Stack is used to hold information when the microprocessor is diverted from its main task to go and do something else. For example subroutine return addresses are stored in the Stack.

The Status Register is a collection of single bits or 'flags' which may be set (to 1) or reset (to Ø) depending on the result of operations within the machine. For example bit BØ of the Status Register is the Carry Flag. If an arithmetic operation carried out in the microprocessor had a result greater than 255 this flag would be set to 1.

ASSEMBLY LANGUAGE

Let's look again at the names or 'mnemonics' we give to machine code instructions.

For example:

> 111Ø 1ØØØ is given the mnemonic INX

This instruction causes the number held in index register X to be incremented by 1. That is the new number held in X after the instruction is carried out is equal to one more than the number which was previously held in X before the instruction was carried out.

INX is short for INcrement X.

The number 111Ø 1ØØØ is known as the operation code or op-code for the instruction INX. Op-codes are normally written in hexadecimal.

Thus:

> #E8 is the op-code for INX.

In general, the mnemonics used to represent numbers which the microprocessor interprets as instructions refer to the various registers and flags. For example:

> STA is STore the Accumulator in memory
>
> LDA is LoaD the Accumulator
>
> SEC is Set the Carry Flag.

The group of mnemonics thus generated is known as the **Assembly Language** for the 65Ø2 microprocessor. A list of 65Ø2 Assembly Language instructions is given in Appendix G.

'Machine code programmers' do not actually program in machine code. They write their programs in assembly language and then convert this to machine code. The assembly language program is known as the 'source code'. The resulting machine code program is known as the 'object code'.

Not all of the machine code numbers are op-codes. Consider, for example, the instruction:

```
LDA# 21
```

This means LoaD the Accumulator with the hexadecimal value 21.

Note – In assembly language all numbers are hexadecimal. The # symbol here does not signify hexadecimal. It means that the accumulator is being loaded directly with a number rather than with the contents of another register or a memory location.

The machine code for LDA# 21 is:

```
#A9 #21
```

#A9 is the op-code for LDA#, but #21 is simply a number, which is loaded into A.

Another example is:

```
STA Ø4Ø2
```

This means STore the Accumulator in the memory location whose address is Ø4Ø2 (hexadecimal) – i.e. change the number held in that memory location so that it is equal to the number held in the accumulator.

The machine code for this is:

```
#8D #Ø2 #Ø4
```

#8D is the op-code for STA.

#Ø2 and #Ø4 give the address at which the accumulator is to be stored. Addresses in machine code are written least significant byte first (yuk!)

Machine code numbers may be op-codes, data or addresses. The microprocessor inteprets them depending on what has come before.

CALL

CALL enables a machine code routine to be entered from BASIC. CALL is a BASIC instruction. It addresses the memory location where the first instruction of the machine code is stored. If, for example, a machine code routine starts at memory address #4ØØ, then the command:

```
CALL #4ØØ
```

will transfer control to the machine code routine.

RTS

RTS is an assembly language instruction. RTS Stands for Return from Subroutine and its op-code is #6Ø. If the machine code routine is CALLED from a BASIC program, RTS will return control to the next instruction in that program.

If CALL is used as a direct command, RTS returns the machine to direct mode.

RTS is normally the last instruction of any user generated machine code routine.

PUTTING MACHINE CODE INTO MEMORY

Machine code is a series of numbers. These can be placed in memory locations using the POKE command directly or in a BASIC program.

This is best illustrated by an example. The next routine places the number #41 in memory location #Ø415. It is to be placed in memory starting at location #Ø4ØØ

```
LDA# 41
STA Ø415
RTS
```

Assembling this program in machine code gives:

```
#A9 #21
#8D #15 #Ø4
#6Ø
```

We could POKE#A9 into memory location #Ø4ØØ, #21 into memory location Ø4Ø1 and so on. Alternatively we could use:

```
1Ø FIRST=#4ØØ
2Ø NUM=6
3Ø FOR K=Ø TO NUM
4Ø READ X
5Ø POKE FIRST+K,X
6Ø NEXT:END
7Ø DATA #A9, #41:REM LDA# 41
8Ø DATA #8D,#15,#Ø4:REM STA Ø415
9Ø DATA #6Ø:REM RTS
```

RUN this program. It will place the machine code routine in memory, but will not execute it. If you examine memory location #Ø415 it is unlikely to contain #41.

If, however, you enter the command:

```
CALL #Ø4ØØ
```

the computer will execute the machine code routine and then return to the 'Ready' state. Memory location #Ø415 will now contain #41.

HIMEM

Where can machine code routines be placed in memory? The answer is that, in theory, they can be put anywhere in user RAM. In practice, however, we have to be careful that machine code is not overwritten by BASIC programs.

There is an area between memory locations #Ø4ØØ and #Ø42Ø which is reserved for machine code. This, however, gives only 32 locations. Another home must be found for larger machine code programs.

We can put machine code programs at the very top of user memory. We saw in Chapter 14 that we can set the maximum memory address which BASIC programs can use with HIMEM.

Normally this address is the top of user memory, #97FF for the 48k machine and #17FF for the 16k machine. If however we change HIMEM to a lower value we can stop BASIC programs using the top locations in user memory, hence leaving 'safe' locations for machine code.

Suppose we wished to reserve 24Ø locations to hold machine code – let's call this 255 locations so we have a few spare. 255 is #FF.

Thus:

```
HIMEM #97ØØ (48k)
```

or

```
HIMEM #17ØØ (16k)
```

would prevent BASIC programs using the top 255 memory locations. We then POKE in the machine code routine starting at #97Ø1 (48k) or #17Ø1 (16k).

DEEK and DOKE

We have seen that, when POKEing in machine code, we store numbers greater than 255 – usually memory addresses – in two adjacent memory locations and that we store the lower order byte first.

Thus to store the number #4Ø26 in memory locations #Ø4Ø2 and #Ø4Ø3 we use

```
POKE #Ø4Ø2,#26
POKE #Ø4Ø3,#4Ø
```

(in the V1.Ø machine we have to convert #26 and #4Ø to 38 and 64 respectively.)

To retrieve the number and print it out we would use:

```
PRINT PEEK(#Ø4Ø2)+256*PEEK(#Ø4Ø3)
```

or if we wished a hexadecimal result:

```
PRINT HEX$(PEEK(#Ø4Ø2)+256*PEEK(#Ø4Ø3))
```

ORIC provides an easier method of doing this with the DOKE statement (Double pOKE) and the DEEK function (Double pEEK).

The equivalent operations using DOKE and DEEK would be:

```
DOKE #Ø4Ø2,#4Ø26
PRINT DEEK(#Ø4Ø2)
PRINT HEX$(DEEK(#Ø4Ø2))
```

USER DEFINED INSTRUCTION (!)

! can be defined as a BASIC instruction using a machine code routine. The address of the start of this routine is DOKEd into memory location #2F5 (ie stored in #2F5 and #2F6). ! can then be used in the BASIC program.

For example the next program transfers 255 bytes of memory from #7ØØ-#7FE to #9ØØ-#9FE.

```
        LDX# FF
LOOP    LDA 6FF,X
        STA 8FF,X
        DEX
        BNE LOOP
        RTS
```

Assembling this gives the object code

```
#A2 #FF
#BD #FF #Ø6
#9D #FF #Ø8
#CA
#DØ #F7
#6Ø
```

To define ! the program is

```
1Ø FOR N=Ø TO 11
2Ø READ A
3Ø POKE #4ØØ+N,A
4Ø NEXT
5Ø DOKE #2F5,#4ØØ
6Ø DATA #A2,#FF
7Ø DATA #BD,#FF,#Ø6
8Ø DATA #9D,#EF,#Ø8
9Ø DATA #CA
1ØØ DATA #DØ,#F7
11Ø DATA #6Ø
```

! may be used as part of a program or as a direct command. Compare the time taken by the machine code routine called up by ! with the time it would take to transfer 255 memory locations using PEEK and POKE.

USER DEFINED FUNCTIONS

&(X) where X is an integer in the range Ø to #FFFF, may be defined using a machine code routine. To define &(X) we store the address of the start of the routine in memory locations #2FC and #2FD using the DOKE command.

&(X) may then be used with any suitable command. For example:

```
PRINT &(6)
IF &(Ø)>1Ø THEN ....
```

USR may also be regarded as a user defined function, but a much more versatile and general purpose one than &(X).

USR feeds a value into a reserved area in memory used by the computer in arithmetic operations and known as the floating point accumulator. This enables the USR function to accept real numbers and return a real value. Real number operations are more difficult to implement in machine code than are integer number operations.

The USR routine is written in machine code and USR is defined by the statement:

```
DEF USR add
```

Where add is the address of the first machine code instruction.

The statement:

```
PRINT USR(X)
```

will put the value X in the floating point accumulator, perform the operations defined by the associated machine code, extract the resulting number from the floating point accumulator and print it on to the screen.

USR and & normally call up machine code routines in the computer's Read Only Memory (ROM). These routines are part of the machine's operating system.

There is no room here for a detailed study of 65Ø2 Assembly Language programming or of the ORIC operating system. If you wish to delve deeper into the former there are many books available. Among the more useful are those by Rodnay Zaks (Sybex) and Lance Leventhal (Osborne McGraw-Hill).

LOADING and SAVING MACHINE CODE

Machine code programs may be saved to or loaded from tape by the memory block form of the CSAVE and CLOAD commands.

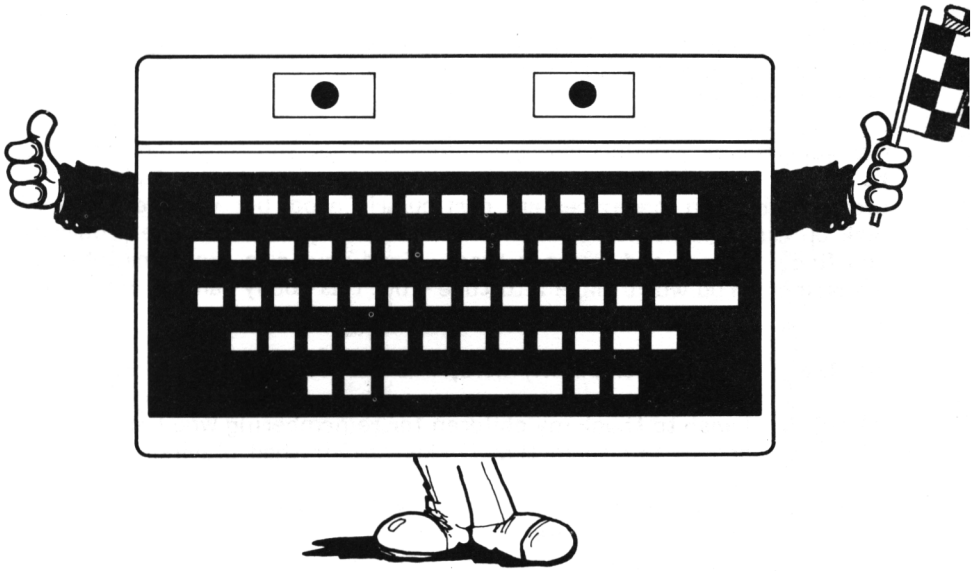
For example:

```
CSAVE"MCP",A#400,E#420
CLOAD"MCP",A#400,E#420
```

Machine code programs may be loaded into locations #400 to #420 without affecting any BASIC programs already held in user memory. If, however, machine code programs are to be loaded into the top of user memory HIMEM should be set to its new value first.

CHAPTER 17

In Conclusion



Using a home computer is easy. Some books make it look difficult. One reason for this is that many non fiction books are written by academics to demonstrate to other academics how clever they are.

I believe most things can be explained simply using straightforward language. Some technical terms are necessary, but jargon should be avoided. Jargon is a difficult weed to uproot and I apologise for any which has slipped through.

I also assume no specialised knowledge. If you have some musical training, for example, you may have found Chapter 11 simplistic. Is the beginning that far back? Yes, for most people it is! Too many authors go to great lengths to explain simple BASIC commands, but assume that their readers can solve complex mathematical equations and know entire musical symphonies off by heart.

Where do you go from here? If you have mastered this book you have left your beginner status well behind. You are an adept, and have all the knowledge required to write useful and elegant programs. You are not yet an expert – that requires practice.

For your future development I recommend that you subscribe regularly to one or more of the many microcomputer magazines. Not all the programs in such magazines are particularly good, but you are bound to pick up useful hints here and there. But don't just follow the programs you find. Change and improve them. Send your own programs into the magazines. You won't make a fortune – but this hobby can pay for itself!

```
10 REM  I wish to thank my wife, who had every reason to complain,
        but didn't.
20 REM  I wish to thank my children for remembering who I am,
30 REM  and to tell them that Daddy has finished playing with their
        computer.
40 REM  I wish to thank my parents, without whom none of this would
        have been possible.
50 REM  I wish to thank my sometime co-author John Gordon,
60 REM  and Dr Paul Johnson of ORIC P.I. for their advice and
        comments.
70 REM  I wish to thank Messrs Prentice Hall International for their
        patience.
80 REM  I wish to thank Alan and Pauline Dockree of Kelvin Word
        Processing,
90 REM  25 Bridgeway Rd, Kirkintilloch, G66
100 REM for turning my scrawled hieroglyphics into a professionally
        laid out product.
110 REM Finally I wish to thank ORIC Products International
120 REM for making such a super little computer.
130 REM That's it!
140 REM Back to the land of the living.
150 END:END:END:END:END:END:END:END:END:END:END:END
```

APPENDIX A

ASCII Codes

ASCII Code (Decimal)	ASCII Code (Hex)	Character	Alternate Character
32	# 20	Space	
33	# 21	!	
34	# 22	"	
35	# 23	#	
36	# 24	$	
37	# 25	%	
38	# 26	&	
39	# 27	'	
40	# 28	(	
41	# 29	)	
42	# 2A	*	
43	# 2B	+	
44	# 2C	,	
45	# 2D	–	
46	# 2E	.	
47	# 2F	/	
48	# 30	0	
49	# 31	1	
50	# 32	2	
51	# 33	3	
52	# 34	4	
53	# 35	5	
54	# 36	6	
55	# 37	7	
56	# 38	8	
57	# 39	9	
58	# 3A	:	
59	# 3B	;	
60	# 3C	<	
61	# 3D	=	
62	# 3E	>	
63	# 3F	?	
64	# 40	@	

ASCII Code (Decimal)	ASCII Code (Hex)	Character	Alternate Character
65	# 41	A	
66	# 42	B	
67	# 43	C	
68	# 44	D	
69	# 45	E	
70	# 46	F	
71	# 47	G	
72	# 48	H	
73	# 49	I	
74	# 4A	J	
75	# 4B	K	
76	# 4C	L	
77	# 4D	M	
78	# 4E	N	
79	# 4F	O	
80	# 50	P	
81	# 51	Q	
82	# 52	R	
83	# 53	S	
84	# 54	T	
85	# 55	U	
86	# 56	V	
87	# 57	W	
88	# 58	X	
89	# 59	Y	
90	# 5A	Z	
91	# 5B	[	
92	# 5C	/	
93	# 5D	]	
94	# 5E	↑	
95	# 5F	£	
96	# 60	©	—
97	# 61	a	—
98	# 62	b	—
99	# 63	c	—
100	# 64	d	—
101	# 65	e	—
102	# 66	f	—
103	# 67	g	—
104	# 68	h	—
105	# 69	i	—
106	# 6A	j	—

ASCII Code (Decimal)	ASCII Code (Hex)	Character	Alternate Character
107	# 6B	k	—
108	# 6C	l	—
109	# 6D	m	—
110	# 6E	n	—
111	# 6F	o	—
112	# 70	p	—
113	# 71	q	—
114	# 72	r	—
115	# 73	s	—
116	# 74	t	—
117	# 75	u	—
118	# 76	v	—
119	# 77	w	—
120	# 78	x	—
121	# 79	y	—
122	# 7A	z	—
123	# 7B	{	—
124	# 7C	|	—
125	# 7D	}	—

APPENDIX B

Microcomputer Magazines

There are a large number of magazines written for the microcomputer user and it would be tedious – and somewhat pointless – to list all of them. I have therefore listed only those magazines where you are most likely to find ORIC articles and programs.

At the time of writing these are as follows:

The ORIC Owner (and Tansoft Gazette)
3 Club Mews
Ely
Cambs CB7 4NW

Personal Computer News
62 Oxford St
LONDON W1A 2HG (editorial address)
or
53 Frith St
LONDON W1A 2HG (subscription address)

Personal Computing Today
Subscriptions Department
513 London Road
Thornton Heath
Surrey CR4 6AR

Home Computing Weekly
Argus Specialist Publications Ltd
145 Charing Cross Road
LONDON WC2H 0EE

Personal Computer World
62 Oxford St
LONDON W1

Your Computer
IPC Electrical-Electronic Press Ltd
Quadrant House
The Quadrant
Sutton
Surrey SM2 5AS

APPENDIX C

ORIC BASIC

KEYWORD	DESCRIPTION	FORMAT
ABS	A function giving the absolute value of its argument.	ABS(N)
AND	The operation of integer bitwise logical AND between two items. Also used in IF . . . THEN statements.	X= #80 AND Y IF . . . AND . . . THEN . . .
ASC	A function returning the ASCII character value of the first character of the argument string. If the string is null then an error code is generated.	ASC(N$)
ATN	A function giving the arc tangent of its argument in radians.	ATN(A)
CALL	A statement which transfers control to a machine code routine starting at the address specified.	CALL addr
CHAR	A statement which places a character on to the HIRES screen. The top left of the character is at cursor position. X = ASCII code of character. S = 0 for standard character set and 1 for alternate character set. FB is 0 to 3 – See below.	CHAR X,S,FB
CHR$	A string function whose value is a string of length 1 containing the ASCII character specified by the least significant byte of the numeric argument.	CHR$(42)
CIRCLE	A statement which draws a circle on the HIRES screen. The centre of the circle is the current cursor position. R is the radius (1-99) FB is 0 to 3 – See below.	

FB CODES	0 Background	1 Foreground	2 Invert	3 Null (do nothing).

KEYWORD	DESCRIPTION	FORMAT
CLEAR	A statement which clears all the dynamically declared variables, including strings.	CLEAR
CLS	A statement which clears the text area on the VDU to the current text background colour and moves the cursor to the first normal text character position.	CLS
CLOAD	A command which loads a new program from a cassette tape file. Normally the old program is erased and all variables are cleared. CLOAD, S – optional 300 baud operation CLOAD, V – verify (V1·1 only) CLOAD, J – join (V1·1 only) CLOAD, A, E loads a specified area of memory.	CLOAD "PROG" CLOAD "PROG", S CLOAD "PROG", V CLOAD "PROG", S, V CLOAD "PROG", J CLOAD "PROG", S, J CLOAD "PROG", A# 500, E#600 CLOAD "PROG", A# 500, E#600, S
CONT	A command which continues the execution of a program after a break.	CONT
COS	A function giving the cosine of its radian argument.	COS(A)
CURMOV	A statement which sets the cursor to a new position on the HIRES screen. X and Y are relative to the previous position. FB is 0 to 3 (as before). The new position specified must be within the limits of the screen.	CURMOV X, Y, FB
CURSET	A statement which sets the cursor to the absolute X, Y position on the HIRES screen. X is 0 to 239 Y is 0 to 199 FB is 0 to 3	CURSET X, Y, FB

KEYWORD	DESCRIPTION	FORMAT
CSAVE	A command which causes a program to be saved in a cassette tape file. CSAVE, S – optional 300 baud save.	CSAVE "PROG" CSAVE "PROG", S CSAVE "PROG", A#400, E#420
DATA	A program object which must precede all lists of data for READ. String items require inverted commas only if they contain leading spaces.	DATA 1, 2, A, "BC"
DEEK	A function which returns a number equal to the contents of the byte at the memory address specified plus 256 times the contents of the byte at that address +1.	DEEK (addr)
DEF FN	A statement which defines a numeric function in terms of one of its variables.	DEF FNF(X) = X*X+3
DEF USR	A statement which defines the start of a machine code USR routine to generate a function.	DEF USR #410
DIM	A statement which dimensions an array. Arrays are predimensioned to 10. Arrays which are to be loaded from a tape file (V1·1) must be dimensioned using DIM.	DIM B$(15, 2)
DOKE	A statement which stores a value N in two contiguous memory locations with addresses A and A+1. INT(N/256) goes into location address A + 1 and N – INT(N/256) goes into location address A.	DOKE A, N
DRAW	A statement which draws a line on the HIRES screen from the current cursor position to that position plus X, Y. FB is 0 to 3.	DRAW X, Y, FB
ELSE	A statement delimiter which behaves as follows: When encountered as a delimiter the rest of the line is ignored. If in an IF statement the Boolean is false, the statements after ELSE will be executed.	IF . . . THEN . . . ELSE
END	A statement causing the interpreter to return to direct mode.	END

KEYWORD	DESCRIPTION	FORMAT
EXP	A function returning e to the power of the argument.	EXP(N)
EXPLODE	A statement producing a predefined sound.	EXPLODE
FALSE	A function returning 0.	FALSE
FILL	A statement which fills X character cells by Y rows on the HIRES screen with a value N. X is 1 to 40 Y is 1 to 200 N is 0 to 127	FILL Y, X, N
FN	A reserved word which returns the result of a user defined numeric function.	FNX(3)
FOR	A statement initializing a FOR . . . NEXT loop.	FOR N=1 to 6
GET	A statement which strobes the keyboard until a key is pressed.	GET A GET B$
GOSUB	A statement which transfers control to a specified line number L but allows a RETURN to the instruction directly following the GOSUB in the program.	GOSUB L
GOTO	As GOSUB, but no RETURN is allowed.	GOTO L
GRAB	A command which assigns the memory from: # 9800 to # B400 (48K) or from # 1800 to # 3400 (16K) to user RAM.	GRAB
HEX$	A function which returns a string containing the hexadecimal conversion of number N.	HEX$(N)
HIMEM	A pseudo-variable which sets the maximum address used by the interpreter.	HIMEM #97FF
HIRES	A statement which puts the machine in HIRES mode, clears the HIRES screen, sets the HIRES background to black and foreground to white, and positions the HIRES cursor at 0, 0 (top left).	HIRES
IF	A statement which sets up a test condition, which can be used to control the subsequent action of the computer.	IF . . . THEN . . .

KEYWORD	DESCRIPTION	FORMAT
INK	A statement which controls the foreground colour of the whole screen. N is an integer from 0 to 7.	INK N
INPUT	A statement which stops the program until the ENTER key is pressed. It then reads the contents of the keyboard buffer into a specified variable. An optional screen message may be generated.	INPUT A INPUT B$ INPUT "...."; A INPUT "...."; B$
INT	A function returning the largest integer less than or equal to its argument.	INT(N)
KEY$	A function which strobes the keyboard without halting the program and returns a string containing the character corresponding to any key depressed. If no key is depressed a null string is returned.	A$ = KEY$
LEFT$	A string function which returns the left N characters of the argument string.	LEFT$(B$, N)
LEN	A function which returns the length of the argument string.	LEN(B$)
LET	Optional assignment statement.	LET K = 6
LIST	A command which lists on the screen the current program or specified lines therein.	LIST LIST 120 LIST 10-60
LLIST	As LIST except that listing is produced by the printer.	LLIST LLIST 120 LLIST 10-60
LN	A function giving the natural logarithm of its argument.	LN(X)
LOG	A function giving the base −10 logarithm of its argument.	LOG(X)
LORES	A statement which switches the machine to a LORES mode. The screen is cleared to background colour black and the foreground colour is set to white. The cursor is returned to the first (top left) character position.	LORES 0 LORES 1

KEYWORD	DESCRIPTION	FORMAT
LPRINT	A statement which prints a specified message on the printer.	LPRINT 6 LPRINT "HELLO"
MID$	A string function which returns N characters of the string starting from character M.	MID$(A$, M, N)
MUSIC	A statement which specifies a musical note or sound to be played by the PLAY command. CH is 1 to 6 and specifies channel. OCT is 0 to 6 and specifies octave. NT is 1 to 12 and specifies note. VOL is 0 to 15 and specifies volume. When VOL = 0 volume is controlled by the PLAY command.	MUSIC CH, OCT, NT, VOL
NEW	A command which initializes the interpreter for a new program to be typed in.	NEW
NEXT	The statement delimiting FOR . . . NEXT loops.	NEXT NEXT N NEXT I, J
NOT	A unary operation equivalent to unary minus.	NOT(X) NOT(A = B)
ON	A statement which causes a program jump to one of a number of locations depending on the value of a defined variable.	ON N GOSUB . . . ON N GOTO . . .
OR	The operation of bitwise integer logical OR between two items. May also be used in IF . . . THEN statements.	X = #80 OR Y IF . . . OR . . . THEN
PAPER	A statement which controls the background colour of the whole screen. N is an integer from 0 to 7.	PAPER N
PATTERN	A statement which sets the PATTERN register. N is an integer from 0 to 255.	PATTERN N
PEEK	A function which returns the contents of memory location X.	PEEK (X)
PI	A function which returns 3.14159265.	PI
PING	A statement producing a predefined sound.	PING

KEYWORD	DESCRIPTION	FORMAT
PLAY	A statement producing a sound defined by SOUND or MUSIC. TE is 0 to 7 and specifies tone channels. NE is 0 to 7 and specifies noise channels. EM is 0 to 7 and specifies envelope mode. EP is 0 to 32767 and specifies envelope period.	PLAY TE, NE, EM, EP
PLOT	A statement which places a character or string of characters on the LORES or TEXT screen. X and Y define the start of the character string. A numeric argument is interpreted as an ASCII code.	PLOT 7, 10, "A" PLOT 8, 4, A$ PLOT 9, 14, 82
POINT	A function returning –1 if the pixel on the HIRES screen specified by X and Y is background and 0 if that pixel is foreground.	POINT (X, Y)
POKE	A statement which stores the value N (0 to 255) in memory location A.	POKE A, N
POP	A statement which removes from the top of the stack the most recently stored subroutine return address.	POP
POS	A function returning the horizontal position of the LORES or TEXT cursor, In the V1·1 machine, POS(0) returns position of screen cursor. POS(1) returns position of print head.	POS (V1·0) POS (0) (V1·1) POS (1) (V1·1)
PRINT	A statement which prints a specified message on the screen.	PRINT 1 PRINT "HELLO"
PULL	A statement which removes from the top of the stack the most recent address stored by a REPEAT statement.	PULL
READ	A statement which will assign to variables values read from the DATA statements in the program.	READ A READ B$
RECALL	A statement which loads into a previously dimensioned array variables stored in a cassette tape file by the STORE command. Implemented on the V1·1 machine only.	RECALL A, "FILE" RECALL B$, "FILE" RECALL A, "FILE," S

KEYWORD	DESCRIPTION	FORMAT
RELEASE	A statement which assigns to the HIRES screen memory previously assigned to user programs by GRAB.	RELEASE
REM	A statement that causes the rest of the line to be ignored.	REM
REPEAT	A statement which is the starting point of a REPEAT . . . UNTIL loop.	REPEAT
RESTORE	A statement setting the READ pointer to the first DATA item.	RESTORE
RETURN	A statement causing a return to the statement after the most recent GOSUB statement.	RETURN
RIGHT$	A string function which returns the right N characters of the argument string.	RIGHT$ (A$, N)
RND	A function which returns a pseudo random number between 0 and 0.99999999. If X>=1 a different random number is normally returned. If X=0 the same random number is returned each time. If X<=0 the random number generator is seeded so that the same series of random numbers is generated each time.	RND (X)
RUN	A command starting a program at the program line specified, or at the lowest numbered line if none is specified. All dynamic variables including arrays are cleared.	RUN RUN N
SCRN	A function which returns the ASCII code of the character at position X, Y on the TEXT or LORES screen.	CHAR (X, Y)
SGN	A function returning −1 for a negative argument, 0 for a 0 argument and 1 for a positive argument.	SGN (X)
SHOOT	A statement producing a predefined sound.	SHOOT
SIN	A function giving the sine of its radian argument.	SIN (A)

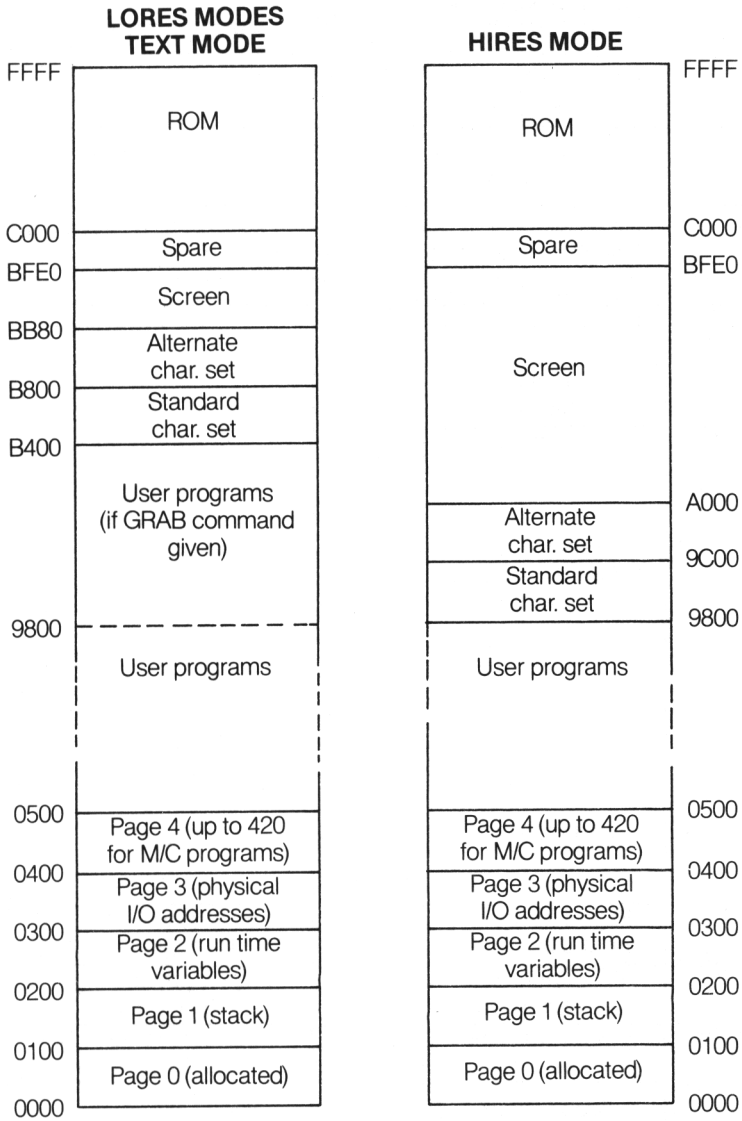
KEYWORD	DESCRIPTION	FORMAT
SOUND	A statement specifying the sound enabled by the PLAY statement. C is 1 to 6 and specifies channel. P is 1 to 4096 and specifies period. V is 0 to 15 and specifies volume. When V = 0 volume is controlled by PLAY.	SOUND C, P, V
SPC	A function which when used with PRINT or LPRINT causes a specified number of spaces to be printed.	SPC (N)
SQR	A function returning the square root of its argument.	SQR (X)
STEP	This function specifies step sizes other than 1 when used as part of the FOR . . . NEXT loop.	FOR . . . TO . . . STEP N
STOP	A statement which stops a program, which may then be restarted with CONT.	STOP
STORE	A command storing variables held in a predefined array into a cassette tape file. (V1·1 only). ,S – optional 300 baud STORE.	STORE A, "FILE" STORE B$, "FILE" STORE A, "FILE", S
STR$	A function which converts its numeric argument into the equivalent string representation.	STR$ (N)
TAB	A function which, when used with PRINT or LPRINT moves the start of printing N spaces from the left (correctly implemented on V1·1 only).	TAB (N)
TAN	A function returning the tangent of its radian argument.	TAN (X)
TEXT	A command switching the machine to TEXT mode.	TEXT
THEN	A keyword used with IF to decide a course of action.	IF . . . THEN . . .
TROFF	A statement switching off the trace function.	TROFF
TRON	A statement switching on the trace function.	TRON

KEYWORD	DESCRIPTION	FORMAT
TRUE	A function returning –1	TRUE
USR	A function passing its argument to a floating point subroutine and returning the value generated by that routine.	DEF USR = addr USR (N)
VAL	A function returning the numerical value of its argument string.	VAL (B$)
WAIT	A statement causing the program to pause for a specified number of ten millisecond intervals.	WAIT N
ZAP	A statement producing a predefined sound.	ZAP

APPENDIX D

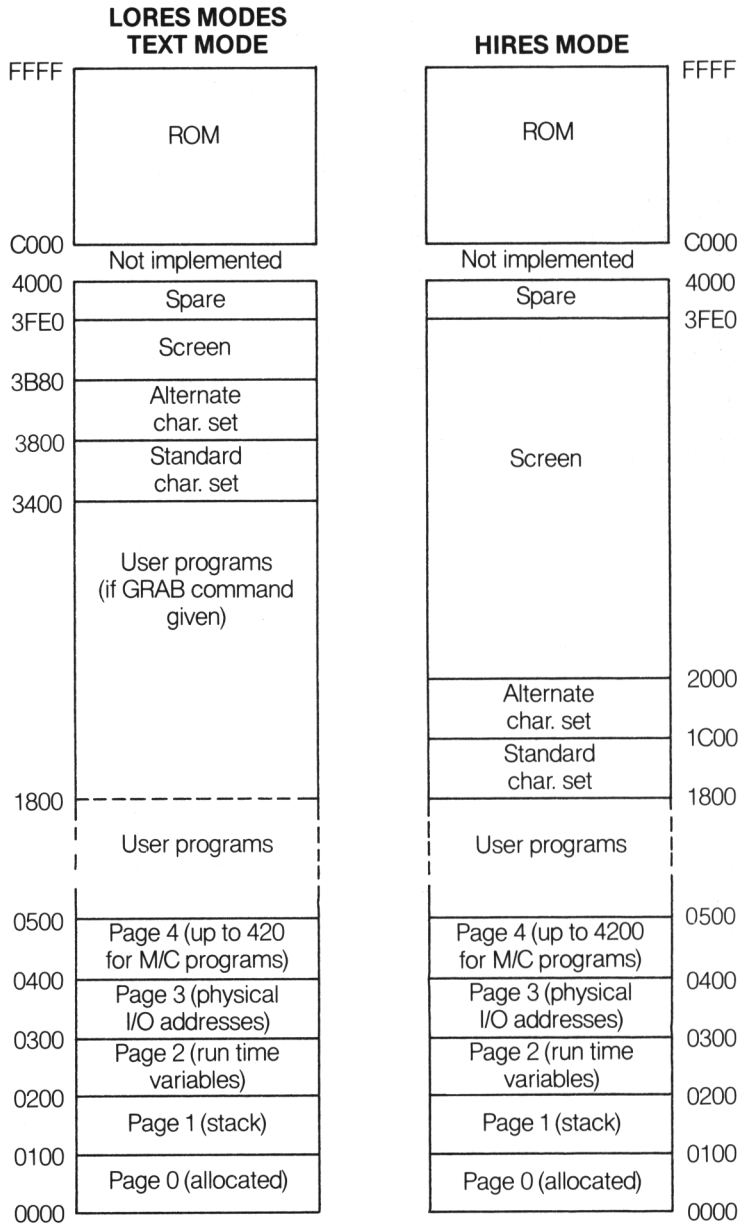
Memory Maps

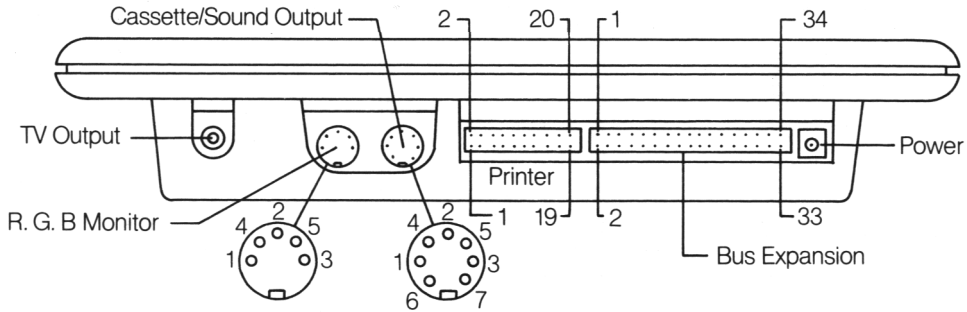
48k MEMORY MAP



All numbers are hexadecimal

16k MEMORY MAP





BUS EXPANSION

MAP	1	2	ROMDIS	STB	1	2	GND	1 – RED	1 Tape Output
02	3	4	RESET	DO	3	4	GND	2 – GREEN	2 GND
I/O	5	6	I/O Control	D1	5	6	GND	3 – BLUE	3 Tape Input
R/W	7	8	IRQ	D2	7	8	GND	4 – SYNC	4 Sound
D2	9	10	D0	D3	9	10	GND	5 – GND	5 Sound
A3	11	12	D1	D4	11	12	GND		6 Relay Contact (Motor Control)
A0	13	14	D6	D5	13	14	GND		7 Relay Contact (Motor Control)
A1	15	16	D3	D6	15	16	GND		
A2	17	18	D4	D7	17	18	GND		
D5	19	20	A4	ACK	19	20	GND		
A5	21	22	D7						
A6	23	24	A15						
A7	25	26	A14						
A8	27	28	A13						
A9	29	30	A12						
A10	31	32	A11						
+ 5V	33	34	GND						

PRINTER

R.G.B. CASSETTE/SOUND

APPENDIX F

Error Codes

Error codes are displayed with a line number when they occur within a program.

1) **CAN'T CONTINUE**
An attempt has been made to continue a program after a line has been added or deleted following a STOP instruction or a CTRLC key entry.

2) **DISP TYPE MISMATCH**
An instruction has been used in the wrong mode (e.g. CHAR in TEXT mode).

3) **DIVISION BY ZERO**
Self-explanatory.

4) **FORMULA TOO COMPLEX**
More than two **IF/THEN** statements have been put in the same line.

5) **ILLEGAL DIRECT**
A statement which can only be used in a program has been used as a direct command from the keyboard (e.g. DATA).

6) **ILLEGAL QUANTITY**
ORIC has been asked to calculate the incalculable (e.g. SQR (–1).

7) **NEXT WITHOUT FOR**
Self-explanatory.

8) **OUT OF DATA**
An attempt has been made to read more DATA items than there are defined in the program.

9) **OUT OF MEMORY**
Self-explanatory, but might also be caused by more than 16 nested program loops or sub-routines.

10) **OVERFLOW**
A number larger than $1.70141*10^{38}$ has occurred during a calculation.

11) **REDIM'D ARRAY**
An attempt has been made to dimension a previously dimensioned array.

12) **RETURN WITHOUT GOSUB**
Self-explanatory.

13) **STRING TOO LONG**
A string more than 254 characters in length has been specified.

14) **BAD SUBSCRIPT**
An attempt has been made to reference an array element that does not exist.

15) **SYNTAX ERROR**
A statement has been misspelled or incorrectly punctuated.

16) TYPE MISMATCH
An attempt has been made to assign a number to a string variable or vice versa.

17) UNDEF'D STATEMENT
An attempt has been made to jump to a non-existent line number.

18) UNDEF'D FUNCTION
An attempt has been made to use a function that has not been previously defined.

19) REDO FROM START
A non-numeric character has been entered when the form of the INPUT instruction requires a number.

20) BAD UNTIL
The program encounters an **UNTIL** without previously encountering a **REPEAT.**

APPENDIX G

6502 Assembly Language Instruction Set

ADC Add Memory to Accumulator with Carry
AND "AND" Memory with Accumulator
ASL Shift Left One Bit (Memory or Accumulator)

BCC Branch on Carry Clear
BCS Branch on Carry Set
BEQ Branch on Result Zero
BIT Test Bits in Memory with Accumulator
BMI Branch on Result Minus
BNE Branch on Result not Zero
BPL Branch on Result Plus
BRK Force Break
BVC Branch on Overflow Clear
BVS Branch on Overflow Set

CLC Clear Carry Flag
CLD Clear Decimal Mode
CLI Clear Interrupt Disable Bit
CLV Clear Overflow Flag
CMP Compare Memory and Accumulator
CPX Compare Memory and Index X
CPY Compare Memory and Index Y

DEC Decrement Memory by One
DEX Decrement Index X by One
DEY Decrement Index Y by One

EOR "Exclusive-Or" Memory with Accumulator

INC Increment Memory by One
INX Increment Index X by One
INY Increment Index Y by One

JMP Jump to New Location
JSR Jump to New Location Saving Return Address

LDA Load Accumulator with Memory
LDX Load Index X with Memory
LDY Load Index Y with Memory
LSR Shift Right One Bit (Memory or Accumulator)

NOP No Operation

ORA "OR" Memory with Accumulator

PHA Push Accumulator on Stack
PHP Push Processor Status on Stack
PLA Pull Accumulator from Stack
PLP Pull Processor Status from Stack

ROL Rotate One Bit Left (Memory or Accumulator)
ROR Rotate One Bit Right (Memory or Accumulator)
RTI Return from Interrupt
RTS Return from Subroutine

SBC Subtract Memory from Accumulator with Borrow
SEC Set Carry Flag
SED Set Decimal Mode
SEI Set Interrupt Disable Status
STA Store Accumulator in Memory
STX Store Index X in Memory
STY Store Index Y in Memory

TAX Transfer Accumulator to Index X
TAY Transfer Accumulator to Index Y
TSX Transfer Stack Pointer to Index X
TXA Transfer Index X to Accumulator
TXS Transfer Index X to Stack Pointer
TYA Transfer Index Y to Accumulator

ADC

Add memory to accumulator with carry

Operation: A + M + C—A, C

N Z C I D V
/ / / – – /

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	ADC	#Oper	69	2	2
Zero Page	ADC	Oper	65	2	3
Zero Page, X	ADC	Oper, X	75	2	4
Absolute	ADC	Oper	6D	3	4
Absolute, X	ADC	Oper, X	7D	3	4*
Absolute, Y	ADC	Oper, Y	79	3	4*
(Indirect, X)	ADC	(Oper, X)	61	2	6
(Indirect), Y	ADC	(Oper), Y	71	2	5*

*Add 1 if page boundary is crossed.

AND

AND memory with accumulator

Operation: A ∧ M—A

N Z C I D V
/ / – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	AND	#Oper	29	2	2
Zero Page	AND	Oper	25	2	3
Zero Page, X	AND	Oper, X	35	2	4
Absolute	AND	Oper	2D	3	4
Absolute, X	AND	Oper, X	3D	3	4*
Absolute, Y	AND	Oper, Y	39	3	4*
(Indirect, X)	AND	(Oper, X)	21	2	6
(Indirect), Y	AND	(Oper), Y	31	2	5*

*Add 1 if page boundary is crossed.

ASL — Shift Left One Bit (Memory or Accumulator)

Operation: C—76543210—0

N Z C I D V
/ / / – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Accumulator	ASL	A	0A	1	2
Zero Page	ASL	Oper	06	2	5
Zero Page, X	ASL	Oper, X	16	2	6
Absolute	ASL	Oper	0E	3	6
Absolute, X	ASL	Oper, X	1E	3	7

BCC — Branch on Carry Clear

Operation: Branch on C = 0

N Z C I D V
– – – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BCC	Oper	90	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to different page.

BCS — Branch on carry set

Operation: Branch on C = 1

N Z C I D V
– – – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BCS	Oper	B0	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to next page.

BEQ — Branch on result zero

Operation: Branch on Z = 1

N Z C I D V
– – – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BEQ	Oper	F0	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to next page.

BIT

Test bits in memory with accumulator

Operation: a ∧ M, M_7 —N, M_6 —V

N	Z	C	I	D	V
M_7	/	–	–	–	M_6

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Zero Page	BIT	Oper	24	2	3
Absolute	BIT	Oper	2C	3	4

BMI

Branch on result minus

Operation: Branch on N = 1

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BMI	Oper	30	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to different page.

BNE

Branch on result not zero

Operation: Branch on Z = O

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BNE	Oper	D0	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to different page.

BPL

Branch on result plus

Operation: Branch on N = 0

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BPL	Oper	10	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to different page.

BRK

Force Break

N Z C I D V
– – – * – –

Operation: Forced Interrupt PC + 2 P

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	BRK	00	1	7

*A BRK command cannot be masked by setting I.

BVC

Branch on overflow clear

N Z C I D V
– – – – – –

Operation: Branch on V = O

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BVC	Oper	50	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to different page.

BVS

Branch on overflow set

N Z C I D V
– – – – – –

Operation: Branch on V = 1

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Relative	BVS	Oper	70	2	2*

*Add 1 if branch occurs to same page.
*Add 2 if branch occurs to different page.

CLC

Clear carry flag

N Z C I D V
– – 0 – – –

Operation: 0 – C

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	CLC	18	1	2*

CLD

Clear decimal mode

Operation: 0 – D

N	Z	C	I	D	V
–	–	–	–	0	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	CLD	D8	1	2

CLI

Clear interrupt disable bit

Operation: 0—I

N	Z	C	I	D	V
–	–	–	0	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	CLI	58	1	2

CLV

Clear overflow flag

Operation: 0—V

N	Z	C	I	D	V
–	–	–	–	–	0

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	CLV	B8	1	2

CMP

Compare memory and accumulator

Operation: A—M

N	Z	C	I	D	V
/	/	/	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	CMP	#Oper	C9	2	2
Zero Page	CMP	Oper	C5	2	3
Zero Page, X	CMP	Oper, X	D5	2	4
Absolute	CMP	Oper	CD	3	4
Absolute, X	CMP	Oper, X	DD	3	4*
Absolute, Y	CMP	Oper, Y	D9	3	4*
(Indirect, X)	CMP	(Oper, X)	C1	2	6
(Indirect), Y	CMP	(Oper), Y	D1	2	5*

*Add 1 if page boundary is crossed.

CPX

Compare memory and index X

Operation: X—M

N Z C I D V
/ / / – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	CPX	#Oper	E0	2	2
Zero Page	CPX	Oper	E4	2	3
Absolute	CPX	Oper	EC	3	4

CPY

Compare memory and index Y

Operation: Y—M

N Z C I D V
/ / / – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	CPY	#Oper	C0	2	2
Zero Page	CPY	Oper	C4	2	3
Absolute	CPY	Oper	CC	3	4

DEC

Decrement memory by one

Operation: M-1—M

N Z C I D V
/ / – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Zero Page	DEC	Oper	C6	2	5
Zero Page, X	DEC	Oper, X	D6	2	6
Absolute	DEC	Oper	CE	3	6
Absolute, X	DEC	Oper, X	DE	3	7

DEX

Decrement index X by one

Operation: X-1X

N Z C I D V
/ / – – – –

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	DEX	CA	1	2

DEY

Decrement index Y by one

Operation: Y-1—Y

N Z C I D V
/ / – – – –

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	DEY	88	1	2

EOR

Exclusive—Or memory with accumulator

Operation: A-M—A

N Z C I D V
/ / – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	EOR	#Oper	49	2	2
Zero Page	EOR	Oper	45	2	3
Zero Page, X	EOR	Oper, X	55	2	4
Absolute	EOR	Oper	4D	3	4
Absolute, X	EOR	Oper, X	5D	3	4*
Absolute, Y	EOR	Oper, Y	59	3	4*
(Indirect, X)	EOR	(Oper, X)	41	2	6
(Indirect), Y	EOR	(Oper), Y	51	2	5*

*Add 1 if page boundary is crossed.

INC

Increment memory by one

Operation: M + 1—M

N Z C I D V
/ / – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Zero Page	INC	Oper	E6	2	5
Zero Page, X	INC	Oper, X	F6	2	6
Absolute	INC	Oper	EE	3	6
Absolute, X	INC	Oper, X	FE	3	7

INX

Increment index X by one

Operation: X + 1—X

N Z C I D V
/ / – – – –

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	INX	E8	1	2

INY

Increment index Y by one

Operation: Y+1—Y

N Z C I D V
/ / – – – –

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	INY	C8	1	2

JMP

Jump to new location

Operation: (PC + 1)—PCL
(PC + 2)—PCH

N Z C I D V
– – – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Absolute	JMP	Oper	4C	3	3
Indirect	JMP	(Oper)	6C	3	5

JSR

Jump to new location saving return address

Operation: PC + 2, (PC + 1)—PCL
(PC + 2)—PCH

N Z C I D V
– – – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Absolute	JSR	Oper	20	3	6

LDA

Load accumulator with memory

Operation: M—A

N Z C I D V
/ / – – – –

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	LDA	#Oper	A9	2	2
Zero Page	LDA	Oper	A5	2	3
Zero Page, X	LDA	Oper, X	B5	2	4
Absolute	LDA	Oper	AD	3	4
Absolute, X	LDA	Oper, X	BD	3	4*
Absolute, Y	LDA	Oper, Y	B9	3	4*
(Indirect, X)	LDA	(Oper, X)	A1	2	6
(Indirect), Y	LDA	(Oper), Y	B1	2	5*

*Add 1 if page boundary is crossed.

LDX

Load index X with memory

Operation: M—X

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	LDX	#Oper	A2	2	2
Zero Page	LDX	Oper	A6	2	3
Zero Page, Y	LDX	Oper, Y	B6	2	4
Absolute	LDX	Oper	AE	3	4
Absolute, Y	LDX	Oper, Y	BE	3	4*

*Add 1 when page boundary is crossed.

LDY

Load index Y with memory

Operation: M—Y

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	LDY	#Oper	A0	2	2
Zero Page	LDY	Oper	A4	2	3
Zero Page, X	LDY	Oper, X	B4	2	4
Absolute	LDY	Oper	AC	3	4
Absolute, X	LDY	Oper, X	BC	3	4*

*Add 1 when page boundary is crossed.

LSR

Shift right one bit (memory or accumulator)

Operation: 0—7 6 5 4 3 2 1 0—C

N	Z	C	I	D	V
0	/	/	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Accumulator	LSR	A	4A	1	2
Zero Page	LSR	Oper	46	2	5
Zero Page, X	LSR	Oper, X	56	2	6
Absolute	LSR	Oper	4E	3	6
Absolute, X	LSR	Oper, X	5E	3	7

NOP

No operation

Operation: No operation (2 cycles)

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	NOP	EA	1	2

ORA

OR memory with accumulator

Operation: AVM—A

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	ORA	#Oper	09	2	2
Zero Page	ORA	Oper	05	2	3
Zero Page, X	ORA	Oper, X	15	2	4
Absolute	ORA	Oper	0D	3	4
Absolute, X	ORA	Oper, X	1D	3	4*
Absolute, Y	ORA	Oper, Y	19	3	4*
(Indirect, X)	ORA	(Oper, X)	01	2	6
(Indirect), Y	ORA	(Oper), Y	11	2	5*

*Add 1 on page crossing.

PHA

Push accumulator on stack

Operation: A|

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	PHA	48	1	3

PHP

Push processor status on stack

Operation: P|

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	PHP	08	1	3

PLA

Pull accumulator from stack

Operation: A↓

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	PLA	68	1	4

PLP

Pull processor status from stack

Operation: P↓

N	Z	C	I	D	V
From Stack					

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	PLP	28	1	4

ROL

Rotate one bit left (memory or accumulator)

Operation: M or A: 7 6 5 4 3 2 1 0—C

N	Z	C	I	D	V
/	/	/	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Accumulator	ROL	A	2A	1	2
Zero Page	ROL	Oper	26	2	5
Zero Page, X	ROL	Oper, X	36	2	6
Absolute	ROL	Oper	2E	3	6
Absolute, X	ROL	Oper, X	3E	3	7

ROR

Rotate one bit right (memory or accumulator)

Operation: M or A: C—7 6 5 4 3 2 1 0

N	Z	C	I	D	V
/	/	/	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Accumulator	ROR	A	6A	1	2
Zero Page	ROR	Oper	66	2	5
Zero Page, X	ROR	Oper, X	76	2	6
Absolute	ROR	Oper	6E	3	6
Absolute, X	ROR	Oper, X	7E	3	7

RTI

Return from interrupt

Operation: P↑PC↑

N Z C I D V
From Stack

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	RTI	40	1	6

RTS

Return from subroutine

Operation: PC↑, PC + 1—PC

N Z C I D V
– – – – – –

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	RTS	60	1	6

SBC

Subtract memory from accumulator with borrow

Operation: A—M—C—A
Note: C = Borrow

N Z C I D V
/ / / – – /

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Immediate	SBC	# Oper	E9	2	2
Zero Page	SBC	Oper	E5	2	3
Zero Page, X	SBC	Oper, X	F5	2	4
Absolute	SBC	Oper	ED	3	4
Absolute, X	SBC	Oper, X	FD	3	4*
Absolute, Y	SBC	Oper, Y	F9	3	4*
(Indirect, X)	SBC	(Oper, X)	E1	2	6
(Indirect), Y	SBC	(Oper), Y	F1	2	5*

*Add 1 when page boundary is crossed.

SEC

Set carry flag

Operation: 1—C

N Z C I D V
– – 1 – – –

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	SEC	38	1	2

SED

Set decimal mode

Operation: 1—D

N	Z	C	I	D	V
–	–	–	–	1	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	SED	F8	1	2

SEI

Set interrupt disable status

Operation: 1—I

N	Z	C	I	D	V
–	–	–	1	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	SEI	78	1	2

STA

Store accumulator in memory

Operation: A—M

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Zero Page	STA	Oper	85	2	3
Zero Page, X	STA	Oper, X	95	2	4
Absolute	STA	Oper	8D	3	4
Absolute, X	STA	Oper, X	9D	3	5
Absolute, Y	STA	Oper, Y	99	3	5
(Indirect, X)	STA	(Oper, X)	81	2	6
(Indirect), Y	STA	(Oper), Y	91	2	6

STX

Store index X in memory

Operation: X—M

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Zero Page	STX	Oper	86	2	3
Zero Page, Y	STX	Oper, Y	96	2	4
Absolute	STX	Oper	8E	3	4

STY

Store index Y in memory

Operation: Y—M

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*		*OP Code*	*No. Bytes*	*No. Cycles*
Zero Page	STY	Oper	84	2	3
Zero Page, X	STY	Oper, X	94	2	4
Absolute	STY	Oper	8C	3	4

TAX

Transfer accumulator to index X

Operation: A—X

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	TAX	AA	1	2

TAY

Transfer accumulator to index Y

Operation: A—Y

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	TAY	A8	1	2

TSX

Transfer stack pointer to index X

Operation: S—X

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	TSX	BA	1	2

TXA Transfer index X to accumulator

Operation: X—A

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	TXA	8A	1	2

TXS Transfer index X to stack pointer

Operation: X—S

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	TXS	9A	1	2

TYA Transfer index Y to accumulator

Operation: Y—A

N	Z	C	I	D	V
/	/	–	–	–	–

Addressing mode	*Assembly language form*	*OP Code*	*No. Bytes*	*No. Cycles*
Implied	TYA	98	1	2

00 – BRK
01 – ORA—(Indirect, X)
02 – Future Expansion
03 – Future Expansion
04 – Future Expansion
05 – ORA—Zero Page
06 – ASL—Zero Page
07 – Future Expansion
08 – PHP
09 – ORA—Immediate
0A – ASL—Accumulator
0B – Future Expansion
0C – Future Expansion
0D – ORA—Absolute
0E – ASL—Absolute
0F – Future Expansion
10 – BPL
11 – ORA—(Indirect), Y
12 – Future Expansion
13 – Future Expansion
14 – Future Expansion
15 – ORA—Zero Page, X
16 – ASL—Zero Page, X
17 – Future Expansion
18 – CLC
19 – ORA—Absolute, Y
1A – Future Expansion
1B – Future Expansion
1C – Future Expansion
1D – ORA—Absolute, X
1E – ASL—Absolute, X
1F – Future Expansion
20 – JSR
21 – AND—(Indirect, X)
22 – Future Expansion
23 – Future Expansion
24 – BIT—Zero Page
25 – AND—Zero Page
26 – ROL—Zero Page
27 – Future Expansion
28 – PLP
29 – AND—Immediate
2A – ROL—Accumulator
2B – Future Expansion
2C – BIT—Absolute
2D – AND—Absolute
2E – ROL—Absolute
2F – Future Expansion
30 – BMI
31 – AND—(Indirect), Y
32 – Future Expansion
33 – Future Expansion
34 – Future Expansion
35 – AND—Zero Page, X
36 – ROL—Zero Page, X
37 – Future Expansion
38 – SEC
39 – AND—Absolute, Y
3A – Future Expansion
3B – Future Expansion
3C – Future Expansion
3D – AND—Absolute, X
3E – ROL—Absolute, X
3F – Future Expansion
40 – RTI
41 – EOR—(Indirect, X)
42 – Future Expansion
43 – Future Expansion
44 – Future Expansion
45 – EOR—Zero Page
46 – LSR—Zero Page
47 – Future Expansion
48 – PHA
49 – EOR—Immediate
4A – LSR—Accumulator
4B – Future Expansion
4C – JMP—Absolute
4D – EOR—Absolute
4E – LSR—Absolute
4F – Future Expansion
50 – BVC
51 – EOR—(Indirect), Y
52 – Future Expansion
53 – Future Expansion
54 – Future Expansion
55 – EOR—Zero Page, X
56 – LSR—Zero Page, X
57 – Future Expansion
58 – CLI
59 – EOR—Absolute, Y
5A – Future Expansion
5B – Future Expansion
5C – Future Expansion
5D – EOR—Absolute, X
5E – LSR—Absolute, X
5F – Future Expansion
60 – RTS
61 – ADC—(Indirect, X)
62 – Future Expansion
63 – Future Expansion
64 – Future Expansion
65 – ADC—Zero Page
66 – ROR—Zero Page
67 – Future Expansion
68 – PLA
69 – ADC—Immediate
6A – ROR—Accumulator
6B – Future Expansion
6C – JMP—Indirect
6D – ADC—Absolute
6E – ROR—Absolute
6F – Future Expansion
70 – BVS
71 – ADC—(Indirect), Y
72 – Future Expansion
73 – Future Expansion
74 – Future Expansion
75 – ADC—Zero Page, X
76 – ROR—Zero Page, X
77 – Future Expansion
78 – SEI
79 – ADC—Absolute, Y
7A – Future Expansion
7B – Future Expansion
7C – Future Expansion
7D – ADC—Absolute, X
7E – ROR—Absolute, X
7F – Future Expansion
89 – Future Expansion
81 – STA—(Indirect, X)
82 – Future Expansion
83 – Future Expansion
84 – STY—Zero Page
85 – STA—Zero Page
86 – STX—Zero Page
87 – Future Expansion
88 – DEY
89 – Future Expansion
8A – TXA
8B – Future Expansion
8C – STY—Absolute

8D – STA—Absolute
8E – STX—Absolute
8F – Future Expansion
90 – BCC
91 – STA—(Indirect), Y
92 – Future Expansion
93 – Future Expansion
94 – STY—Zero Page, X
95 – STA—Zero Page, X
96 – STX—Zero Page, Y
97 – Future Expansion
98 – TYA
99 – STA—Absolute, Y
9A – TXS
9B – Future Expansion
9C – Future Expansion
9D – STA—Absolute, X
9E – Future Expansion
9F – Future Expansion
A0 – LDY—Immediate
A1 – LDA—(Indirect, X)
A2 – LDX—Immediate
A3 – Future Expansion
A4 – LDY—Zero Page
A5 – LDA—Zero Page
A6 – LDX—Zero Page
A7 – Future Expansion
A8 – TAY
A9 – LDA—Immediate
AA – TAX
AB – Future Expansion
AC – LDY—Absolute
AD – LDA—Absolute
AE – LDX—Absolute
AF – Future Expansion
B0 – BCS
B1 – LDA—(Indirect), Y
B2 – Future Expansion
B3 – Future Expansion
B4 – LDT—Zero Page, X
B5 – LDA—Zero Page, X
B6 – LDX—Zero Page, Y
B7 – Future Expansion
B8 – CLV
B9 – LDA—Absolute, Y
BA – TSX
BB – Future Expansion
BC – LDY—Absolute, X
BD – LDA—Absolute, X
BE – LDX—Absolute, Y
BF – Future Expansion
C0 – CPY—Immediate
C1 – CMP—(Indirect, X)
C2 – Future Expansion
C3 – Future Expansion
C4 – CPY—Zero Page
C5 – CMP—Zero Page
C6 – DEC—Zero Page
C7 – Future Expansion
C8 – INY
C9 – CMP—Immediate
CA – DEX
CB – Future Expansion
CC – CPY—Absolute
CD – CMP—Absolute
CE – DEC—Absolute
CF – Future Expansion
D0 – BNE
D1 – CMP—(Indirect), Y
D2 – Future Expansion
D3 – Future Expansion
D4 – Future Expansion
D5 – CMP—Zero Page, X
D6 – DEC—Zero Page, X
D7 – Future Expansion
D8 – CLD
D9 – CMP—Absolute, Y
DA – Future Expansion
DB – Future Expansion
DC – Future Expansion
DD – CMP—Absolute, X
DE – DEC—Absolute, X
DF – Future Expansion
E0 – CPX—Immediate
E1 – SBC—(Indirect, X)
E2 – Future Expansion
E3 – Future Expansion
E4 – CPX—Zero Page
E5 – SBC—Zero Page
E6 – INC—Zero Page
E7 – Future Expansion
E8 – INX
E9 – SBC—Immediate
EA – NOP
EB – Future Expansion
EC – CPX—Absolute
ED – SBC—Absolute
EE – INC—Absolute
EF – Future Expansion
F0 – BEQ
F1 – SBC—(Indirect), Y
F2 – Future Expansion
F3 – Future Expansion
F4 – Future Expansion
F5 – SBC—Zero Page, X
F6 – INC—Zero Page, X
F7 – Future Expansion
F8 – SED
F9 – SBC—Absolute, Y
FA – Future Expansion
FB – Future Expansion
FC – Future Expansion
FD – SBC—Absolute, X
FE – INC—Absolute, X
FF – Future Expansion

Index

PRENTICE HALL INTERNATIONAL PERSONAL COMPUTER BOOK

This book puts you in control of your ORIC–1 microcomputer right from the start. No previous knowledge of computers is needed.

Clear step-by-step explanations show you:

- **how to understand and use the keyboard**
- **how to handle words and numbers**
- **how to display information on the screen**
- **how to use the sound and graphics features**
- **how to define your own graphics symbols**
- **how to use a cassette recorder to store programs and data.**

Later chapters cover slightly more advanced topics:

- **machine code and assembly language**
- **file handling**
- **using the printer.**

There are detailed, helpful appendices on BASIC commands, error codes, ASCII codes, the 6502 assembly language instruction set, and other subjects.
This book covers the old and the new versions of the ORIC–1, both V1.0 and V1.1.

Ian McLean is a chartered engineer with seventeen years' experience in the electronics industry and in education.

0-13-477332-2

£7.95

cover design: Diana Allen

UNDERSTANDING ORIC McLEAN

Prentice / Hall International