



A

T

M

O

S

M

A

N

U

A

L

**Ian Adamson**

# **The ORIC ATMOS Manual**

Pan Books London and Sydney

Published 1984 by Pan Books Ltd,  
Cavaye Place, London SW10 9PG  
in association with Personal Computer News

9 8 7 6 5 4 3

This revised edition produced  
for Oric Products International Ltd 1984

© Ian Adamson 1984

ISBN 0 330 28482 7

Photoset by Parker Typesetting Service, Leicester

Printed and bound in Great Britain by

Richard Clay (The Chaucer Press) Ltd, Bungay, Suffolk

This book is sold subject to the condition that it shall not,  
by way of trade or otherwise, be lent, re-sold,  
hired out or otherwise circulated without the publisher's prior consent  
in any form of binding or cover other than that  
in which it is published and without a similar condition including  
this condition being imposed on the subsequent purchaser

# Contents

Introduction	1
1 Getting the ATMOS together	3
2 The Language Lesson	7
3 Building With BASIC	23
4 Loops Beyond Compare	36
5 Down memory lane	50
6 Tapes and Data	61
7 Graphics and Colour	69
8 The Sound of Music	93
9 Oric BASIC Keywords	108
10 Introducing Machine Code	190
11 Input/Output	223

## Appendices

1 ASCII Character Codes	228
2 Escape Codes	232
3 Error Messages	234
4 Screen Grids	237
5 Memory Map	240
6 Binary/Hex/Decimal Conversion	244
7 Oric MCP-40 Printer Use	252
8 6502 Op Codes	261
9 ROM Routines and Addresses	266
10 Input/Output Circuitry	273
11 ATMOS I/O Connections	284
12 BASIC Reserved Words and Tokens	289

# Acknowledgements

The book you are holding could not have been produced without the teamwork of many people. As the notional author of this text, it falls to me to express my gratitude and my debt to them all: Andrzej Chanerley and Roger Smith for technical inputs, Ian Ritchie for musical expertise, Brian Jones for programming, printing, graphics, ethyl alcohol and corn esters supplies, Fi Inchbald for edits, optimism and balm, and Richard Kennedy for hard work, sacrifice and support in the circumstances surrounding the production of this book.

All characters and situations depicted in this book, as with all books, are fictitious, and should not be confused with any events, but this should not bother anyone who understands that 'Emptiness is not different from Form and Form is not different from Emptiness'. This book is dedicated to the memory of Pigalle and all that remains truly sacred.

Thank you.

# Introduction

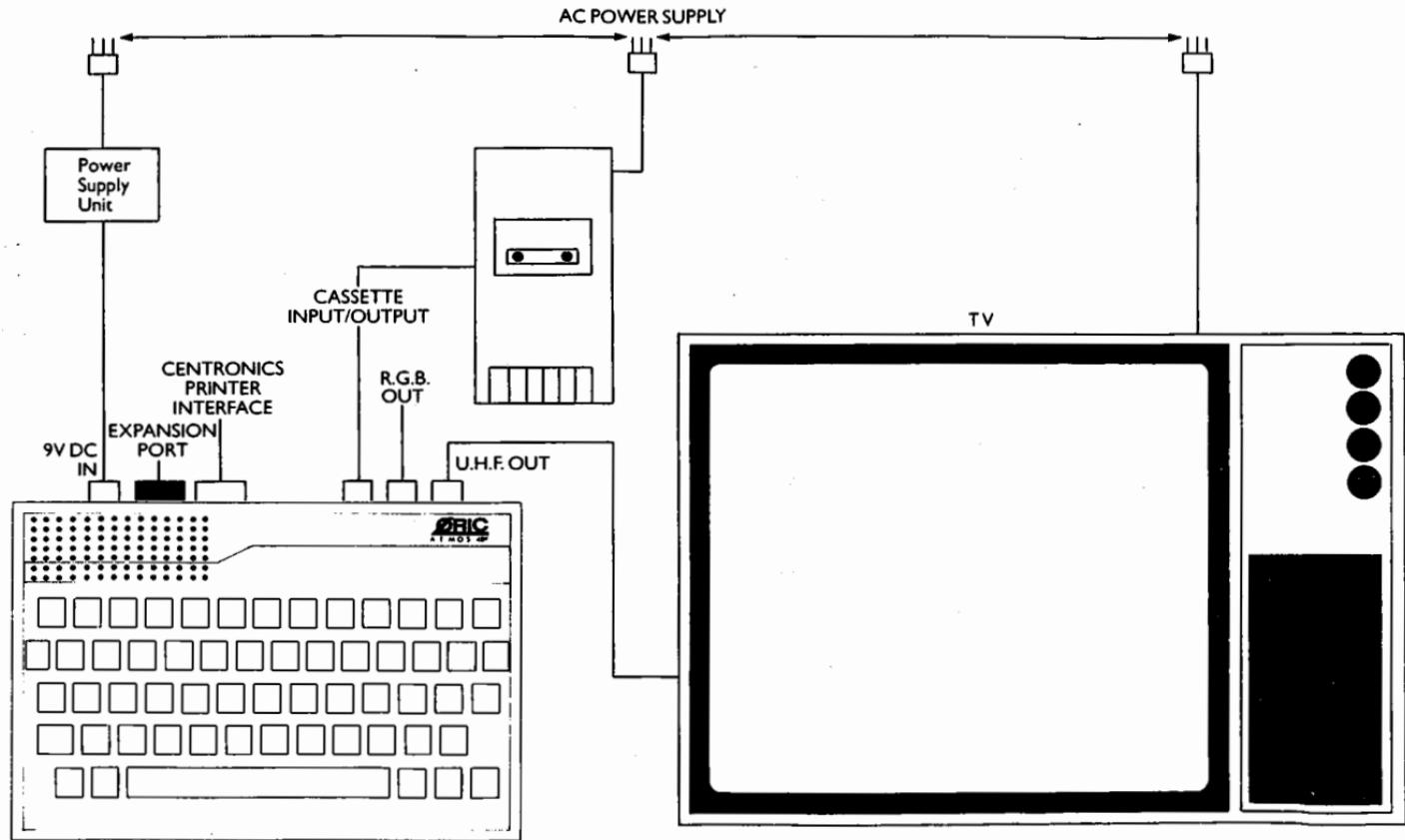
Welcome to the world of the Oric ATMOS microcomputer! If you're reading this you have become the proud owner of an ATMOS, and this handbook intends to show you how to use it to its full potential. High-resolution graphics, three channel sound, colour and many other features make the Oric ATMOS one of the most advanced microcomputers available today.

This handbook covers all you need to know to get the most from your ATMOS, learn the built-in BASIC language, get started on programming in BASIC, and discover the fascination of computing. If you're starting from scratch, please remember when the going gets a little tough that it's a complicated and powerful machine you are learning to use, and you can't expect to have it all worked out in a few hours.

Exercise some patience and application, work through the examples we give in the text, and you'll soon be hacking a keyboard with the best of them! We introduce the jargon as we come to it, but you should also realise that computing is a technical subject, and familiarity with the technical terms is a necessary part of your initiation into computing. Whilst bits and bytes and BASIC, chips and Ks and all the rest, are confusing when first encountered, they are all essential elements in discussing our subject.

This handbook first covers setting up your Oric ATMOS, and describes the facilities built into it, and some of the expansion possibilities. The first elements of the BASIC language are defined, and the elements of programming are explained as you are introduced to more of the BASIC instructions. After covering the storage and retrieval of programs on cassette tape, the complex and powerful graphics and sound facilities of the Oric are covered in separate sections. The full set of the Oric BASIC instructions and commands is then covered in an alphabetic glossary, with their formats and examples of their use, for easy reference. Chapters on machine code programming and input/output interfacing follow, and the appendices provide additional technical and reference material not suitable for the main text.

We're sure you'll find that the world your ATMOS opens up will provide you with an inexhaustible source of fun, fascination and satisfaction. Let's take our first steps towards opening up this world!



# 1 Getting the ATMOS together

Let's first run through what comes in the Oric box of goodies. You get the Oric ATMOS itself, the power supply, a cassette lead, and a TV lead. Find a convenient work surface big enough for the ATMOS, your TV, and a cassette recorder if you've got one, and you can proceed to set your system up. The components of your microcomputer system should be connected up as shown in the diagram.

The Oric power supply should be plugged into a convenient socket, close enough so that there's plenty of slack in the cable, the plug of which is fitted into the power socket of the ATMOS (at the left rear of the Oric as you face the keyboard). This supplies the 9V d.c. (direct current) power which the Oric needs. There's no on/off switch on the Oric, so it's now powered up, and two elements of our system are working. (The keyboard, which is an input device in the jargon, since we use it to input information and commands, and the processor board, which as you might imagine does all the processing of information and handles the input and output of data, are of course all built into one unit in the Oric ATMOS.)

We need an output device to know what's happening though, and this is where your TV comes in. This functions as an on-line visual display unit (VDU) output device, but we'll just call it the TV! Connect the TV lead between the Oric (using the rightmost socket) and the aerial socket of the TV. If there's a choice, use the UHF socket. Switch on the TV, and tune it to Channel 36. When you get the correct position, the screen will be displaying the message:

ORIC EXTENDED BASIC V1.1

© 1983 TANGERINE

xxxxx BYTES FREE

READY



This notice comes up on switching on the Oric. If you now take out the power supply from the Oric and re-insert it, the screen goes blank, with random white blocks on it for a few seconds (while the Oric checks through its circuits), and then clears to display the message again.

Adjust the brightness and contrast controls to get a sharp picture. The Oric itself has two controls that can be used to fine-tune the picture. They affect the colour signal, but also have some effect on the stability and steadiness of the picture, and we will deal with them here for completeness. When you start to experiment with the colour capabilities of your ATMOS you may find that colour contrast is lacking, especially between yellow and white, or that you have 'dot crawl' producing a shimmering effect, making the colour display hard on the eyes and reducing the quality of the screen resolution. If you are positive that none of this is due to a badly adjusted TV, then you should have a look underneath the Oric. You will find two small holes, fairly close together; of which one is lozenge-shaped and contains a slotted silver screw, and the other round, with a brass screw. Find a screwdriver, and leave the Oric switched on, with a multi-coloured display on the screen. Turn the Oric over and turn the screw within the lozenge-shaped hole a quarter turn or so in both directions, watching the screen as you go, until you have found the setting at which the yellow just starts to fade. Back off slightly from this position and the bias modulator is then correctly set. This should also centre the picture on the screen. Next adjust the brass screw within the round hole, adjusting it in the same way until you get a positive colour contrast, with no shimmering or rainbow effect is achieved. Remove the screwdriver from the screwhead in case it affects the signal. The controls are set for a standard test TV at the factory, and adjustment is necessary if your TV differs significantly.

Under the copyright notice the Oric displays the number of bytes of free memory. The Oric ATMOS comes with different memory sizes of 48K and 16K. A K is an abbreviation of 'kilobyte', meaning 1024 bytes (you'll find out why 1024 and not 1000 later), and a byte is a storage location in the computer memory. Out of the total memories of 49152 (48K) and 16384 (16K) bytes initially available, however, the Oric takes some on switching on for use by the system, to store information about the TV screen display, and the current state of affairs, so the number of bytes of free memory is less.

The READY on the screen indicates your ATMOS is waiting for instructions. The flashing square is the cursor. This is an indicator that shows the current position at which characters will be printed on the screen.

There are other sockets on the rear of your Oric (see Appendix 11). The R.G.B. (red, green, blue) outlet is for the connection of a colour monitor, rather than a TV. Monitors are high-quality VDUs, which provide a firmer and sharper picture quality than a TV, but are hence rather more expensive, and are generally dedicated to a computer system, but combined monitor/TV systems are available.

The next socket is for input and output to a tape recorder, and needs rather less expenditure to be put to use, especially since the cheaper cassette recorders work at least as well as more expensive ones. There are cassette recorders designed for use with computer systems, but most models will work well. The Oric can use the cassette recorder as an off-line storage device, which means that programs can be stored as an encoded sound signal on tape, and then played back in again. With your Oric ATMOS you will have received the demonstration programs supplied with the machine, and should refer to Chapter 6 on cassette handling if you can't wait to get this or other commercial software loaded into your Oric.

To use a cassette, the 3-pin DIN plugs on the cassette leads must be inserted into the Oric socket and that of the tape recorder. Most tape recorders will have a DIN socket, but if yours doesn't, alternative cassette leads are available which provide a 3-pin DIN plug for the Oric end, and two 3mm jack plugs at the other for the tape recorder EAR and MIC sockets. The Oric uses the cassette sound port to SAVE and LOAD programs, but the outputs from the sound and music facilities of the Oric are also sent as signals to this port, so that you can use the cassette lead to connect up to your hi-fi system if you wish to amplify or record the music you produce on your Oric. See Appendix 11 for the connections.

Next to the cassette port is the printer port, with its 20-pin socket. The Oric can be used with any printer with a Centronics parallel interface, providing a suitable cable is available. The Oric MCP-40 printer (which was used for all the printouts in this handbook) gives not only printed versions of programs, or hard-copy of program output, but can also produce four-colour graphics of high quality. This will be available from wherever you bought your Oric, and because of its versatility should be considered if you are going to invest in a printer. The capacity to produce hard copy output, and especially printed program listings, is extremely useful. Word processing and similar tasks need a full-size printer, but the MCP-40 is quite adequate for the hobbyist. Appendix 7 deals with the use of this printer, and Chapter 11 with use of the printer port for input/output.

The adjacent larger socket is the expansion port, for connecting additional pieces of equipment which can use this I/O (input/output) port in various ways. Such items as a joystick interface, a modem (telephone communications system) and the Oric MICRODISC Drive can be connected. Appendix 10 and Chapter 11 give details of the use of this port for the electronics enthusiast.

The alternative storage medium that is available for the Oric is the Oric MICRODISC Drive. Disc drives provide a fast, sophisticated way of storing and retrieving data from a floppy disc, but do require a considerable investment. Most of the facilities provided by a disc drive exist or can be simulated using a cassette recorder, but the disc system wins hands down on speed, flexibility and convenience.

If you find yourself spending a lot of time with your Oric and getting caught by the programming bug, you will find yourself contemplating the purchase of a drive and/or a printer (the sequence depends on what you want to do with the system), and you'll come to appreciate what amazing value the Oric is, comparing it's capacity to interface with these peripherals and its own built-in capabilities with the cost of 'add-ons'. Unfortunately that's the way the electro-mechanical cookie crumbles. For the moment, however, let's get on with exploring what we can do with the Oric's own facilities.

## 2 The Language Lesson

You have now got your Oric up and running, but apart from the message on the screen it's not doing much, is it? Well, computers are dumb beasts, as well as being very literal-minded, and your Oric is waiting to be given something to do. We need to talk in some language understood by both the Oric and ourselves, so that we can use the keyboard to input some instructions. In this chapter we're going to introduce you to the use of your Oric, and show you how to get it to do what you want. This means explaining a lot of new concepts and actions, so it's a little daunting if you are new to computing. Don't skip any of it, though, as it's all a necessary introduction to bigger and better things! Try all the things suggested in the text, as you'll only learn from experience. Experiment as much as you like, since you can't hurt the Oric with anything you enter at the keyboard.

The operations of the microchips in your Oric take place in a low-level language known as machine code. We have a section on machine code later in this handbook for those who wish to experiment with this language, but it is a difficult topic, and the Oric comes ready fitted with the BASIC language, which is much easier to get to grips with (the name is an acronym standing for Beginners All-Purpose Symbolic Instruction Code).

BASIC is a high-level language, meaning we don't have to concern ourselves with the mechanics of the complex operations inside the computer, but we can use commands and instructions to describe the operations we wish to occur. The Oric then interprets these instructions and does the job asked of it without much fuss, and very quickly, as long as we put in the instructions in the correct format.

Let's take a look at some commands first. Commands are keyed in through the keyboard, and will appear on the screen as you type the letters and symbols. The keyboard has all the standard letters and numbers, plus some special keys and BASIC symbols. The large unmarked key is the SPACE bar, flanked by the cursor control keys (with the arrows). The SHIFT keys allow you to access the upper symbol printed above the key, where this occurs. DEL stands for delete, as you might suspect. Try pressing any of the letter/symbol keys at random, with and without SHIFT. Whatever you input from the keyboard cannot hurt the Oric, although it is possible to find yourself facing a machine that has 'hung up' and won't respond to keys being pressed.

Should this occur, you have two options. The first is a bit drastic, although it won't do any harm, and simply involves pulling the plug (the d.c. supply into the back of the Oric), waiting a few seconds, and putting it back in. You'll see the power-up message return to the screen when it clears. However, there's a useful device on the underside of the Oric, inside the small square hole on the bottom of the casing. It's well protected because you don't want to press it accidentally, but the button inside is the RESET switch. This has the distinct advantage, as you'll come to appreciate later, that it resets the machine without wiping out the program that you've spent hours keying in, and can save you hours of work.

Note that the keys auto-repeat – if you hold a key down after a short delay the character will be printed repeatedly for as long as you continue to press the key. If you input a random set of characters and then press the RETURN key (which activates these commands or letters which you've been inputting), you will get an error message. This says:

### ?SYNTAX ERROR

A syntax error message means the Oric doesn't understand what has been entered, because it doesn't make sense according to the rules by which commands and instructions are interpreted.

If you enter (say by holding down one key) more than 75 characters a bell sound is given when you key in the 76th (the last character on the second line). The bell sounds again for the 77th, 78th and 79th characters. If you enter an 80th character, it is not printed on the screen, but a '/' is printed instead, and the cursor moves to the start of the next line. The Oric will only accept lines of less than 80 characters, and if this is exceeded it prints '/' and 'forgets' the line. The bell sound is the warning of this possibility.

Let's run through some direct commands. Press RETURN. If you've got characters in there, you'll get ?SYNTAX ERROR, then READY on the next line. Key in:

### PRINT "MESSAGE"

then press RETURN. Make sure you use the double quotes, not the apostrophe (which is the unshifted character on the same key). The Oric will PRINT the characters between the quotation marks on the next clear line of the screen display, so that MESSAGE appears on the screen, and then gives you the READY prompt, telling you that it's waiting to be asked to do something else. Type in:

**PRINT "4+5" (then press RETURN)**

The Oric puts 4+5 on screen. Now key in

**PRINT 4+5 (plus RETURN)**

This time the Oric gives us 9 on the screen. We'll assume you've caught on

to the use of RETURN to activate commands by now, so we won't mention it again. Because we missed out the quotes, our command was interpreted by the Oric as an instruction to PRINT the result of adding 4 and 5. Anything between quotes is known as a *string literal*, or just a *string*.

In a PRINT statement, exactly what's between the quotes is PRINTed out. Anything that's not between quotes is interpreted as a set of instructions. We can combine the two, and enter:

```
PRINT "4*3="; 4*3
```

The asterisk (\*) is used instead of the multiplication sign, to avoid confusion with the small x. The expression 4\*x is evaluated by the Oric, since it is not enclosed in quotes, whilst the string between quotes is printed just as it appears (less the quotes). This gives us a printout of:

```
4*3=12
```

You should note that your Oric gives you a space between the = sign and 1. The Oric gives positive numbers both a leading space (where the minus sign goes for negative numbers), and a following space automatically when the number is PRINTed on the screen. This ensures the separation of numbers which follow each other, and is very helpful when we need to format output to the screen.

We can deduce from the above that BASIC includes some standard English words, and arithmetic symbols (called operators), such as '+'. The full set of these arithmetic operators is:

+	Addition
-	Subtraction
*	Multiplication
/	Division
↑	Exponentiation (raising to a power)

The use of '\*' for the multiplication sign and '/' for division is standard in computing. Since your best way of understanding how the Oric deals with arithmetic is to key in numeric calculations using these operators, here's an Oric function that will save you some time. The question mark, '?', can be used as an abbreviation for PRINT. We won't use it in the text, since it can be confusing, but wherever we have, e.g., PRINT "HI", you can enter ?"HI", and the result will be the same. Try this with the example below.

Right, let's try some arithmetic. PRINT 6/2 (or ? 6/2) will give you 3, PRINT 2↑2 will give you 4, and can be read as 'two raised to the power two', or 2<sup>2</sup> (two squared). If you try to PRINT complex expressions such as 2/3+4 or 4\*3+5/3 you may find that the results are not always what you think they're going to be. This is because there are a set of rules governing the way an

expression is worked out by the Oric. To give an unambiguous meaning to a complex expression the Oric assigns a different priority to each type of operator. Highest priority is anything enclosed in brackets, then exponentiation, followed by \* and / (equal priority), and lastly + and - (equal). So if we try PRINT  $2+3*4$  (do!), then  $3*4$  is evaluated first, since \* has higher priority than +. Having worked out that  $3*4$  is 12, the Oric then adds 2, and prints up 14. We'll get the same answer if we use  $4*3+2$ , but if what we want the answer to is really 'the sum of 3 and 2, multiplied by 4' we have to use brackets. If we ask for the result of  $(3+2)*4$ , 3+2 will be worked out first, and the result multiplied by 4. The Oric, when working out an expression, first performs the operations with the highest priority, then those with the next highest, and so on. Operations with the same priority are worked out left to right, so that  $4-2+4+5$  will give you 11, while  $4-(2+4+5)$  gives -7, as the expression in the brackets is worked out first, and the result (11, which presumably you can work out without the Oric to help!) is then subtracted from 4. If we enter  $4*6/7$  it will evaluate as  $(4*6)$ , calculated first to give 24, divided by 7, with the result of 3.42857143.

The Oric gives numbers to an accuracy of 9 significant figures. It also has a trailing space after the number and a leading space. This ensures that numbers are printed separately on the screen. Try keying in PRINT "1"; "2". This PRINTS the two strings "1" and "2" with the second directly following the other (which is the function of the semi-colon in a PRINT instruction). Now try PRINT 1;2. Again the semi-colon means that the second item is to be printed directly after the first, but what you actually get on screen is:

(space)1(space)(space)2(space)

Let's just deal with a couple of other ideas, and we'll be ready to have a look at a BASIC program. If we enter PRINT "ABC" the Oric will give us ABC on screen, because we had quotes round it, meaning 'print the stuff between the quotes'. If we key in PRINT ABC, however, we will get 0. What's this? Well, because it was asked to PRINT something not between quotes, the Oric attempted to evaluate it as a number. A named variable can be used to store numbers, so the Oric tried to find a variable called ABC, failed, and gave us a zero.

You can think of a variable as a storage box with a name. The contents of the box can be altered (hence variable). The command for giving (assigning) a variable a value is LET. Key in LET ABC=93 followed by RETURN, then try PRINT ABC again. This time the Oric looks for the use of ABC as a variable name, and finds in the storage location named ABC the value 93, so it prints it. We can also proceed to reassign the value given to our variable ABC.

Try entering the commands below, printing out the value of ABC after each operation, using PRINT ABC or ?ABC. We've used a space between PRINT (or ?) and ABC, but this is not necessary. The Oric will accept PRINTABC or ?ABC just as happily. The same is true of most instructions, so that LETABC=100 will work. However, running things together in this way can make it difficult to read a printed listing of the instructions. Now try:

```
LET ABC=100
LET ABC=13+10
LET ABC=ABC+10
LET ABC=ABC*2
LET ABC=ABC/2.3
```

As you can see, we can not only give a new value to ABC, but can also use the old value of ABC in the calculation of the new value. You can think of the instruction LET ABC=ABC\*2 being interpreted as 'take the value stored in the location called ABC, multiply it by two, then put the result back into ABC'. This will also explain why if we try the command LET 100=ABC we will get the ?SYNTAX ERROR message again, because the Oric interprets the first characters after the LET as a variable name. Valid variable names must start with a capital letter, although they may be followed by any combination of capital letters and digits. The other important point about variable names is that, although they may be of any length (so that we can use meaningful variable names like HOUSEPRICE or PERCENT) only the first two letters are significant. Try PRINTing a variable ABACUS and you will find it the same as ABC, because to the Oric they are both stored under the name AB. Variable names also cannot contain any reserved words.

A reserved word is a set of letters that the Oric recognizes as a legal BASIC word. You have seen PRINT and LET, and will be introduced to others as we work through Oric BASIC. The full set of reserved words is given in Chapter 9, Oric BASIC Keywords, and these are the only words the Oric understands. It checks anything we key in, character by character, against a table of BASIC words, and transfers any such word into a different form for internal storage, whenever we press RETURN. Just to confuse the issue, LET is optional, and can be omitted, and the Oric will still know what you mean. This is because, as we stated before, anything outside quotes is assumed to be either a variable or a bit of Oric BASIC. ABC=100 is the same as LET ABC=100. Try the examples above without the use of LET.

We now need to deal with editing before we move on to programming. Up to now we've assumed that you either haven't made mistakes in keying anything in, noticed in time and were sharp enough to use the DELETE key and then re-enter the bit you got wrong, or else pressed RETURN, got the ?SYNTAX ERROR message, and tried again.

All the above options would have helped you to understand the importance of keying things in correctly. You can't afford to be sloppy with a

computer. However, the Oric has editing facilities which minimise the effort required to correct mistakes or change a small portion of a line. Well, we now come to the mysterious arrow (cursor control) keys and the CTRL key. You will have noticed that the Oric PRINTS on screen lines top to bottom, and that when it needs a new line and the screen is full it scrolls, shifting all lines upwards and losing the current top line in the process. To get rid of the current contents of the screen, enter CLS (RETURN). This command CLEARS the screen, placing the cursor in the top left of the screen. The same effect can be produced by using the CTRL key. This key is used in combination with certain other keys to produce some useful effects.

If you hold down the CTRL key and then press the 'L' key the screen will be cleared, just as if you had used CLS and then RETURN. The control key method saves some keystrokes! The other common control functions in addition to CTRL-L are:

CTRL-A	Copy character at current cursor position
CTRL-F	Keyclick on/off
CTRL-Q	Cursor on/off
CTRL-X	Current line abort
CTRL-T	Upper (CAPS) and lower case toggle

CTRL-A and CTRL-X we will deal with below. CTRL-F turns off the sound the keyboard makes when a key is pressed, and CTRL-Q makes the cursor disappear, when first used (since keyclick and cursor are on when the Oric is first switched on), and then turn keyclick and cursor back on. Such a switch is called a toggle. Up to now we've been using upper case letters only, and you should have noticed the CAPS sign in the top right of the Oric screen, on what is known as the *status line*. CTRL-T is the toggle to switch between upper and lower case letters. Try CTRL-T and you'll find that the CAPS sign disappears, and you get lower case letters. The SHIFT keys will give you capitals, whereas in CAPS mode they have no effect. The reason why CAPS is normally set ON is that BASIC COMMANDS AND INSTRUCTIONS HAVE TO BE IN CAPITALS.

OK, we now can go on to examine a BASIC program. Let's give you a program, and then examine what it does. We'll deal with editing directly after giving the program listing below, since you are likely to make some errors. Like all the listings in this handbook, this one is reproduced directly from the Oric on the Oric printer. Whenever you key anything in, be sure to pay attention to punctuation, since a semi-colon and a colon, or an apostrophe and a quote sign, are easy to confuse, and produce radically different effects. Our first program is a calculation of VAT, which is a tax of 15% added on to the sale price of goods. In this case an item is to be sold at £21, and we need to write a program that tells the Oric the sale price (which is the input data in computerspeak), then get the Oric to calculate 15% of the selling price (which is 15/100 or 0.15), and then add this amount to the

selling price. This total gives us the final price to be paid. This is the processing portion of our program, and now we need to define our output, which in this case means to PRINT the total on the screen. Breaking the task down into small steps to be performed in sequence gives us a set of operations that can be easily translated into BASIC. We then write a program like this.

```
10 REM *UAT Calculation*
20 LET SELL=21
30 LET VAT=0.15*SELL
40 LET TOTAL=SELL+VAT
50 PRINT "TOTAL=";TOTAL
```

If you look at this, you can see that the program consists of a set of program lines, numbered from 10 to 50 in steps of ten, and in sequence. Use CTRL-L to clear the screen, or enter CLS followed by RETURN, and then key in the program in exactly as listed, pressing RETURN at the end of each line. Unlike its use with direct commands, such as we were using previously, here RETURN at the end of a program line tells the Oric to store the line in memory, the cursor returns to the left of the screen to indicate that it's ready for another program line or command.

If you make an error as you type in a line, you can use CTRL-X to abort the line. This PRINTs a backslash character and returns the cursor to the left of the screen, having instructed the Oric to forget the line. Enter the line again, and repeat the process until you get it right. Remember you could use the DEL key to DELETE characters, and then type in the correct ones, but we're introducing you to CTRL-X, so use it!

Line 10 has some lower case letters in it. After you've typed 10 REM \*VAT C you will need to use CTRL-T to switch to lower case, and CAPS will disappear from the top status line of the screen. Enter the lower case letters, and then use CTRL-T again to switch back to CAPS mode.

The REM statement means that whatever follows in that line is just a REMARK or REMinder to anybody looking at the program listing, and is to be ignored by the Oric. (It still stores it in memory, though.) The asterisks are just characters put in to make the REMark stand out, and aren't multiplication signs in this case.

When you've keyed in all the lines, if you didn't get it right every time, you will have a screen that has the correct program lines split by the ones you've aborted, looking something like this (although we hope you did rather better!):

```

10 REM *UAT Calculation*
20 LET SELL-\
20 LET SELL=20
30 LET VAT=0.158\
30 LET VAT=0.15*SELL
40 LET TOTAL=SELL+VAT
50 PRINT TOTAL='T\
50 PRINT "TOTAL=";TOTAL

```

To get a listing of the program lines accepted by the Oric, type in LIST as a command, and the program will appear on screen as it has been stored in memory, and this should be the same as the first listing above. Now key in line 20 again, with a deliberate mistake.

```
20LET SEL=20
```

You now have two line 20s on screen. Use CTRL-L and then enter LIST again. The new line 20 has replaced the old. Notice that the Oric has automatically inserted a space between the 20 and LET, although we didn't have one. If we'd typed:

```
20      LET SEL=20
```

we would still only have got one space when we LISTed the program. Well, now we've got an incorrect line, and want to correct it without retyping the whole line. There is a command EDIT, which needs to be followed by the line number we want to correct. Enter EDIT 20, and when you press RETURN you'll get a blank line beneath the 'Ready' prompt displayed after the program listing, then line 20, with the cursor at the beginning of line 20. Hold down the CTRL key, and press A. The cursor moves right, and as it gets to each character the character is merged with the cursor. CTRL-A copies the character at the current cursor position into a temporary memory store (called an input buffer). Move the cursor along until it's on the equals sign. We have now 'copied' 20 LET SEL into the buffer, but not the equals sign. We need to insert an L, and we do this by moving the cursor to an empty portion of the screen. Press the up-arrow key to get the cursor above the line. Press L and it appears on screen. This is now in the buffer, following the first L. We need to follow this with =, so we use the left-arrow key and move the cursor back to cover the L we've just keyed in. Use the down-arrow key to get the cursor over the =, and then use CTRL-A again to copy everything to the end of the line. Press RETURN. You'll then have:

Ready  
EDIT20

L  
20 LET SEL=20

with the cursor below it. Use LIST again, and you'll see that the contents of the buffer (including the inserted L) have become the new line 20. You can copy anything that is on the screen, using the arrow keys to position the cursor, and then CTRL-A to copy, without using EDIT, but EDIT gives you a blank line above and below to type additional characters. If you are inserting something, you must get back to where you were before you started the insert. Try moving the cursor to the L of the LIST command you've just typed in, copy LIST with CTRL-A, then hit RETURN. The program will be LISTed again. This copy facility is very useful, as it can be used to repeat similar portions of lines. Move the cursor up level with line 40, then copy the 4. If you now press 3, the 0 of 40 at the cursor position will change to 3. Copy LET, then use the left-arrow key to move along to the S of SELL. Copy SELL, and when you've got the cursor over the + sign, press =. Again, the + will change to =. Copy VAT, then press RETURN. Now use LIST, and you'll see that there's a new line in the listing, so our program looks like this:

```
10 REM *UAT Calculation*
20 LET SELL=20
30 LET VAT=0.15*SELL
40 LET TOTAL=SELL+VAT
43 LET SELL=VAT
50 PRINT "TOTAL=" ;TOTAL
```

You can see how easy it is to make alterations and correct lines, and you'll now see why we started our program with the line numbers going up in tens. The line number may be any integer (whole number) up to 63999, so we could have numbered our program as 1, 2, 3 . . . or 9000, 9050, 9051 . . . or any sequence. If we don't leave gaps in the sequence, however, we won't be able to insert additional lines, which are often required, so we used steps of ten. You can also see that the Oric automatically inserts a line in the correct sequence, so it doesn't matter in what order we enter lines. We don't want line 43, and any line is easily deleted by just entering the line number and pressing RETURN. Do this, then LIST the program again, and you'll see that line 43 has been wiped out of the listing.

That's editing dealt with, so we'd better get back to our program. If we look it over, we can see that it should do its job. The variable SELL is set to

the value of £21, the VAT amount is correctly calculated as 0.15 of SELL, line 40 adds SELL and the calculated VAT, and line 50 should PRINT the string "TOTAL=" followed by the calculated TOTAL. We have our program, and we activate it by typing RUN (as a command) and pressing RETURN. The Oric then starts at the first line number and processes each line according to the sequence of increasing line numbers. Try it.

Well, that wasn't too successful. Unless you have introduced some errors of your own into the listing (in which case go back to the listing on screen, using LIST, check until you've found the error, and then re-enter the correct program line), you will have got a ?SYNTAX ERROR IN 40 error message. Note that error messages produced when a program is RUN give the line number at which the Oric stopped the execution of the program because it found a mistake. This doesn't always mean that the error is in the line whose number is given, since an earlier error may not have stopped the program (remember the computer can't know what we're trying to do, it just checks to see that the lines make sense, even if that sense is not what we intended), but could have produced the later error.

In this case, when we look at line 40 we find no obvious errors. LET is spelt correctly, the variable names all start with a letter and the arithmetic expression makes sense (we haven't tried to say LET SELL+VAT=TOTAL, for instance, which is meaningless to the Oric – the variable which is to be set or redefined must precede the = sign). So what's wrong? Well, we've just encountered one of the restrictions on variable names mentioned above, to whit, that which requires a variable name to not contain a BASIC reserved word. In this case it is the word TO incorporated in TOTAL. As you become more familiar with Oric BASIC you are less likely to make this mistake, but it can be baffling. If in doubt, check the list of reserved words in Appendix 12. To correct the error we must use a valid variable name. Let's use PRICE. Since this has the same number of letters as TOTAL, we can cursor up to line 40, copy the line number and LET (and the space after LET), then, with the cursor over T, type in PRICE, and then copy the rest of the line. Pressing RETURN puts the revised line into memory. It's important to remember that what's on the screen may not be what's in the memory if you are doing a lot of editing. Use LIST frequently, so you know what your Oric has stored.

Line 50 must also be revised, but we can leave the string "TOTAL=" as is, since we can have anything we want inside the quotes as this is just a collection of characters to the Oric, and it knows it doesn't have to check for BASIC words inside the quotes. So copy line 50 up to and including the semi-colon, and then type PRICE followed by RETURN. The program should now look like this after you've LISTED it again.

```

10 REM *UAT Calculation*
20 LET SELL=20
30 LET VAT=0.15*SELL
40 LET PRICE=SELL+VAT
50 PRINT "TOTAL=";PRICE

```

Press RETURN, then either take the cursor down the screen to a blank line, or use CTRL-L. Then enter RUN. When you press RETURN you'll get:

```

RUN
TOTAL = 23

```

So we've got a working program, but you might be wondering why we bothered, since it was hardly an efficient way of doing something that would have taken us a few seconds with rather cheaper materials such as a stretch of flat sand and a finger. The answer lies in the computer's ability to perform defined tasks quickly and repeatedly. Since we have now got a working program, with the necessary sequence of small operations defined which perform the desired task (such a method of producing the desired result is called an algorithm) and having translated this into BASIC instructions entered into the computer (called coding the program) and test-run the program, we may consider improvements.

The first thing we need is some way of applying the calculation to different values of SELL. We could copy line 20, changing the value of SELL, and then RUN the program again, but there's a better way. We can use the BASIC instruction INPUT, which tells the Oric to wait for an input from the user. This needs to be followed by the name of the variable in which whatever is keyed in will be stored. Enter a new line 20:20 INPUT SELL and then RUN the program. A question-mark appears, and anything keyed in appears on screen. If you enter a number, this value will be assigned to the variable SELL when you press RETURN, and the program will then continue. The single question-mark doesn't give the user much guidance on what the Oric is expecting to be entered, and INPUT allows us to provide a prompt string. The prompt is enclosed in quotes, must be followed by a semi-colon, and the variable to be input is last. Key in:

```
20 INPUT"ENTER SALE PRICE";SELL
```

RUN the program again. Try entering non-numeric values for SELL. The INPUT is checked by the Oric, and if it doesn't make sense as a number the message ?REDO FROM START is displayed, and the prompt and question mark are displayed again, as the Oric waits for more acceptable INPUT to be forthcoming.

We could do with a way to avoid entering RUN each time, and BASIC provides several ways to do this. The simplest (but not the best) is to use GOTO. This instruction is followed by a line number, and on encountering this instruction, the flow of program control (which normally follows the sequence of line numbers) is diverted to the specified line number. We want to repeat the whole sequence, starting at line 20, so we add a new line, and a REMark to comment on the action. This uses an abbreviation for REM (just as a ? can be substituted for PRINT), which is the apostrophe ('). The use of a comment for such a simple and self-explanatory instruction may appear redundant, but we're introducing with a very simple program the same instructions and techniques that apply in large programs, perhaps with hundreds of lines of code, which need REMarks to clarify the program's purpose and structure. This applies as much to the program author as anybody else, as it is surprisingly easy to find yourself looking at a program weeks or days later, and not remember the logic of the way you wrote the program. Keeping notes of the process of program development, and scattering REMs around the program, can help avoid such irritating and time-consuming problems.

```

10 REM *VAT Calculation*
20 INPUT "ENTER SALE PRICE" ;SELL
30 LET VAT=0.15*SELL
40 LET PRICE=SELL+VAT
50 PRINT "TOTAL=" ;PRICE
60 'Round again for next calculation
70 GOTO20

```

Every time the program encounters line 70, the program flow is sent to line 20, and the process repeats. This procedure can only be halted, with the program as it stands, by using CTRL-C instead of INPUTting the value of SELL. This performs a BREAK operation, halting the execution of the program. CTRL-C will halt any operations, but the program cannot be restarted, so it is mainly used to abort a program that is not performing as it should, or to get out of endless loops. Despite all the other stuff in between, our program is equivalent to:

```

10 PRINT "LOOP"
20 GOTO 10

```

There are better methods of arranging program loops, you will be glad to hear, but we'll deal with those in Chapter 4.

If you RUN the program again we can now do repeated calculations, and we're getting to the point (finally!) where you might be able to appreciate the value of computers, since your Oric will now do repeated calculations as

quickly as you INPUT data for it to work on. Of course, unless you happen to be a shopkeeper or suchlike, you may not feel that it enhances your life, but that's another matter.

For now, you're being introduced to the BASIC language, and it's as good a program as any to work with. Fun, flashing colours and electronic music have to wait until we've covered the essentials of BASIC, unfortunately.

So, for the moment, what else can we do with our admittedly somewhat tedious program? Well, currently the screen scrolls with each new PRINT statement encountered. It would be nice if each calculation started on a fresh screen. We've seen CLS as a direct command, and we can also use it in a program line. Enter a replacement for the current line 70:

```
70 CLS:GOTO 20
```

We've now got a line with two instructions, separated by a colon. This operates just as in an English sentence: First one statement is executed, and then the second. You can have as many statements on a single line as you want (up to the line length allowed), but there must be a colon after each statement.

If you RUN the program, you'll see that line 70 works, so that the screen is cleared, and the program then goes TO line 20. There are a couple of problems, though. Firstly, the screen doesn't clear before the first prompt is printed, and secondly there's no time for the result to be read – it just flashes up something, and then as quickly erases it. What do we do about this? Oric BASIC has a statement that halts program execution for a specified time: WAIT. This is followed by a number specifying how many hundredths of a second the delay will be. WAIT 100 will stop the program for 1 second, so we can try putting in a delay whilst the result of our program is displayed. Again, we'll just add it into a new line 70, which now has 3 statements in it. EDIT line 70 so that the program reads as follows:

```
10 REM *UAT Calculation*
20 INPUT"ENTER SALE PRICE";SELL
30 LET VAT=0.15*SELL
40 LET PRICE=SELL+VAT
50 PRINT "TOTAL=";PRICE
60 'Round again for next calculation
70 WAIT 500:CLS:GOTO 20
```

We now have a program that waits whilst we have the result on screen, but the delay is inflexible. We could do with a way of allowing the user to tell the Oric when a new calculation is to be performed. We've used strings thus far only as characters between quotes, but we can also have string variables,

which store strings of characters in the same way as numeric variables hold numbers. The variable names have to follow the same rules as those for numeric variables: no reserved words in the name, first character of the name a capital letter, and the rest any combination of capital letters and numbers. The variable is defined as a string variable by adding \$ to the end of the name. W\$, NAME\$, ST12\$, would all be valid string variable names. String variables can be PRINTED, INPUT and assigned using the (optional) LET statement and the = sign, in the same way as numeric variables. Again, only the first two letters of the variable name are recognised. The value assigned to a string variable is a string of characters, and not a number, and although we could give a string variable the value '123' this is still just a sequence of characters. The program below can be entered with our VAT program still in the Oric:

```

110 A$="ORIC"
120 LET A1$="BASIC"
130 LET LAST$="LESSON"
140 INPUT"ENTER YOUR NAME, PLEASE" ;NAME$
150 CLS:PRINT A$;" ";A1$;" ";LAST$
160 PRINT "FOR " ;NAME$
```

We can RUN just the portion of the program from line 110 onwards by using a different format for RUN. Entering RUN110 as a command will start the program execution at line 110. Line 110 assigns 'ORIC' to the string variable A\$, without using LET, and A1\$ and LAST\$ are assigned in lines 120 and 130, with LET used. Line 140 uses INPUT with a prompt string, for which the format is the same as that used with numeric variables. 150 clears the screen, then PRINTS the three strings in sequence, separated by spaces between quotes, and line 160 prints FOR, followed by a space, and then your name stored in NAME\$.

We could also use GOTO110 (or GOTO 110, since, as is usually the case with the Oric, the spaces don't count), to start the program from line 110. This would apparently have the same effect as RUN110, but there is a difference. Try using GOTO110 to start the program again. The difference can be shown by using GOTO160. This will PRINT out FOR, followed by the name that was INPUT last time the program was executed from line 110. The old value of NAME\$ was preserved. If you now RUN160, you will just get FOR, because the variable NAME\$ has been CLEARED of what it stored. In a program or as a command, CLEAR is used to wipe out any existing contents of variables. Strings revert to an empty or null string (not a string of spaces), and numeric variables to zero. The difference between using RUN and GOTO (line number) to start a program is that RUN does an automatic CLEAR operation before starting. RUN110 again, then enter CLEAR as a command,

and PRINT NAMES\$. You will get nothing on screen, since NAMES\$ now has nothing in it.

Having completed our digression through strings, RUN and GOTO, we can deal with the problem of our VAT program again. Eradicate lines 110 to 160, by entering just the line numbers and then RETURN. Remember this doesn't alter the screen display, so LIST the program again to check they're all non-existent. Enter the following new lines:

```
65 'First get keyboard input
66 GET A$
```

We now know what A\$ represents. GET is an instruction that tells the Oric to wait until a key is pressed, and then store a single-character string in the following string variable, corresponding to the key that was pressed ('X' if you press X, etc.). The result of line 66 is to wait until a key is pressed, before the program moves on to the next program line. The WAIT 500 instruction can now be edited out of line 70. Your program should now look like this:

```
10 REM *UAT Calculation*
20 INPUT"ENTER SALE PRICE";SELL
30 LET VAT=0.15*SELL
40 LET PRICE=SELL+VAT
50 PRINT "TOTAL=";PRICE
60 'Round again for next calculation
65 'First get keyboard input
66 GET A$
70 CLS:GOTO 20
```

RUN the program again, and confirm that GET works as stated. Notice that we don't do anything with A\$ when we use GET in this way, since we're not interested in which key was pressed, just in the fact that some key was pressed by the user of the program.

This brings up another point which should be borne in mind when writing a program. We know that a key is to be pressed because we've written the program. Anyone else needs instructions and guidance. This is referred to as making a program 'user-friendly'. Below is a listing of an improved version, with some more friendly features.

```
10 REM *UAT Calculation*
15 CLS
20 PING:INPUT"ENTER SALE PRICE";SELL
25 PRINT:PRINT"SELLING PRICE=";SELL;"Po
unds"
```

```
30 LET VAT=0.15*SELL
35 PRINT "VAT on ";SELL;"Pounds=";VAT;"P
ounds"
40 LET PRICE=SELL+VAT
50 PRINT "TOTAL=";PRICE;"Pounds"
60 'Round again for next calculation
61 'Print prompt
62 PRINT:PRINT"Press any key to make an
other entry"
65 'Get keyboard input
66 GET A$
70 CLS:GOTO 20
```

Line 15 has been added to clear the screen before anything else happens. Line 20 has had PING (one of the Oric's predefined sound commands) added to provide an audible INPUT prompt. PINGS companions are ZAP, EXPLODE and SHOOT, and are useful simple sound commands. (See the Keywords and Sound chapters for more information.) Lines 25 and 35 have been added to clarify the operations the program is performing, and the units (pounds) have been stated. Line 61 is a REMARK referring to line 62, which tells the user what he or she has to do next.

Key in this final version of our program. If you wish to experiment with saving programs on to tape, Chapter 6 has the details of how to go about this, and you might as well experiment with this program.

Make sure you understand all the bits of BASIC we've used thus far. Chapter 9 contains every Oric BASIC keyword in alphabetic order, with the format required and its definition. You might usefully check through the definitions of Chapter 9 for all the BASIC words we've encountered.

We'll proceed with this introduction to BASIC by considering the way data is held in the Oric, and the ways this can be manipulated.

# 3 Building with BASIC

We've now seen some of the Oric's BASIC instructions, and how they are put together into a program. There are three essential elements to any program, however complicated, whether we want exciting alien hordes or boring VAT figures as our end result. The Oric, or any other computer, needs data (which may be defined within the program or INPUT), plus a set of instructions as to how to process said data, and how to display the result. Before we move on to introduce more complex BASIC instructions we need to know more about the way the Oric deals with data.

We've seen the two types of data – string and numeric – that the Oric can handle. The numeric variables we're using are real or floating point variables (ordinary decimal numbers in fact). The biggest number the Oric can hold is 1.70141E+38 and the smallest is 2.93874E-39. These are expressed in exponential, E or scientific notations, which the Oric itself uses to display numbers outside the range of 99999999 and 0.01. Try the two following programs to see how the computer changes its display as the numbers become larger or smaller, and note the different results of going outside the available range.

```
10 N=2  
20 N=N*8  
30 PRINT N  
40 WAIT10:GOTO20
```

```
10 N=3  
20 N=N/2  
30 PRINT N  
40 WAIT10:GOTO20
```

Exponential format provides a way for the Oric to display very large or small numbers concisely. Numbers in exponential format are always displayed as a decimal between 1 and 9.99999999, followed by E, followed by a number between +38 and -39. The exponent value defines how many places the decimal point is shifted (to the right if the exponent is positive, and to the left if it is negative) to get the correct number. Put another way, it

defines how many times the mantissa (the decimal value before the E) is to be multiplied (E positive) or divided (E negative) by 10. Thus 1.23E+09 is 1230000000.0 and 5.67E-04 is 0.000567. The Oric used two-digit numbers for the exponent, and always includes the + sign for positive exponents, but it will accept numbers without either a leading zero or the plus sign. For example, 1.23E9 would be accepted as a valid input. Try inputting small and large numbers, in both the exponent and standard formats to acquaint yourself with the system.

The Oric can handle integer (whole number) values between 32767 and -32768, and has a separate type of variable to store such number. Integer variables are identified by appending % to a valid variable name. Thus I%, N4%, WHOLE% are all valid names for integer variables. They can be assigned non-integer values, but in this case the number will be rounded down to the next smallest integer (not to the nearest integer). Enter the following program to see this in action:

```

10 LET Bx=6
20 Bx=Bx/2.23
30 PRINT Bx
40 REALVAR=4.845
50 WHOLUARx=REALVAR
60 PRINT WHOLUARx
70 Rx=-3.4
80 PRINT Rx

```

Note especially what happens to a negative number. In rounding down negative numbers, we need to remember that -1, is smaller (less than) zero, -2 smaller than -1, etc. Hence the next smallest integer to -4.67 is -5, and so on. Integer variables are processed more quickly in calculations than floating point variables, and also take up less memory, so they should be used whenever possible if speed or economy of memory is important to a program.

The Oric also recognises hexadecimal numbers. These are numbers representing a different number system which is frequently used in computing, since it allows convenient and concise representation of binary numbers. Binary numbers are the sequences of zeros and ones used by the Oric (and all other computers) to hold all forms of data memory.

Binary representation is used because the electronic switches which are the basis of computers can only be either off or on, and each 'switch' is used to represent 1 when on and 0 when off. There's more on number systems later (see Chapters 5 and 10), but for the moment all you need to know is that the Oric will accept, both in a program, and as input, anything preceded by the hash sign, #, which is a valid hexadecimal number. Try

printing #FFFF, #1, #A, #10, as examples, and sneak a look forward if you can't wait for the explanation. The Oric has internal routines that convert all the binary numbers into outputs we humans can cope with more easily, such as decimal number (and hexadecimal, for the computer freaks), and vice versa, when we enter numeric data.

We've introduced strings and string variables, but since we've just said that all the information in your Oric is in the form of binary numbers, how come we get letters and symbols on the screen? Well, they're (binary) numbers too. Each character that appears on the screen has an associated character code, (known as its ASCII code). The full list of all the character codes is given in Appendix 1.

We now move into the area of BASIC functions.

Oric BASIC contains functions which perform various manipulations on the data it holds. Some of these allow us to convert one type of data into another. We can find the numeric code of a character using the string function ASC. The format we need to use with ASC is to place the character of which we want the code inside brackets. Try entering:

```
PRINT ASC("A")
```

You'll find 65 is PRINTed out, which is the ASCII code for A. The program below does this repeatedly for any character (SHIFTed or not) that you enter as A\$, using GET. Note that the string inside the brackets can be either a literal string in quotes, as above, or, as in the program, a string variable. All functions require what is known as an *argument* (the thing inside brackets) which they operate on, and are said to return a *result* (the ASCII code of the string argument, in this case).

```
10 PRINT "PRESS A KEY"
20 GET A$
30 PRINT "YOU PRESSED "A$
40 PRINT "THE CODE FOR "A$ "="ASC(A$)
50 GOTO 10
```

When you've had enough of pressing keys and seeing the ASCII codes PRINTed out, try stopping the program using CTRL-C. You'll find that instead of breaking into the program, you're informed that you pressed a blank, and that the code for a blank is 3. The ASCII code for CTRL-C is 3, but since this is a non-PRINTing character, we don't get anything on screen. You'll have to use the RESET button underneath your Oric to stop the program. (You will probably need a biro or something similar to get at the button inside the square hole.)

Notice that since this program does not use semi-colons as separators

between PRINT items, the PRINT line, although it works, is not as easy to read as it would be if semi-colons were used.

The reverse function to ASC is CHR\$. This takes a number between 0 and 255, which is the valid range of the argument, and returns a string containing the character whose ASCII code is that number. Having RESET your Oric, add line 45 to the program, so that it LISTS like this:

```

10 PRINT "PRESS A KEY"
20 GET A$
30 PRINT "YOU PRESSED "A$
40 PRINT "THE CODE FOR "A$ "="ASC(A$)
45 C=ASC(A$):PRINT "CHR$(";C;") GIVES "
;CHR$(C)
50 GOTO 10

```

Line 45 takes the number returned by ASC(A\$), and assigns it to the variable C. This is then inserted into the middle of the PRINT statement (which does use semi-colons) and taken as the argument for CHR\$ at the end of the line. RUN the program. Again you'll have to use the RESET button to stop the program. Notice that because the numeric value C was used, there is both a leading space and a trailing space attached to the number. The correct format for CHR\$ is CHR\$(I), where I is a number 0 to 255, with no spaces, as in the actual function at the end of the line 45

Strings and numeric values can be translated using VAL and STR\$. VAL has the format VAL(a\$), where a\$ means any string literal or variable with initial characters that the Oric can interpret as a number. Try this program to experiment with VAL:

```

10 PRINT "INPUT A NUMBER"
20 INPUT NUM$:NUM=VAL(NUM$)
30 PRINT "THE STRING ";NUM$;" CAN BE TU
RNED INTO THE NUMBER ";NUM
40 PRINT "2*";NUM;"=";2*NUM
50 GOTO 10

```

VAL will evaluate and return the result of any string interpretable as a number, up to the first non-numeric character. Try entering 23DF, +45, #F6 (hexadecimal format), #2BK (K not a valid hexadecimal character), 1E4 (exponential notation), 23.6E7, -23.6E9, and 2\*3 to illustrate the action of VAL with different strings. Because we're using INPUT in this program, you can use CTRL-C to break the program at the INPUT prompt.

The inverse function to VAL is STR\$, a string function which returns the string containing the result of evaluating a numeric expression. The expression or number is evaluated by the Oric's arithmetic routines and is

then changed into a string, so we get the same result (but as a string) as if a number had been printed on the screen. So complex expressions (possibly including other functions) can be turned into the string equivalent of their result. We can have lines like:

```
PRINT STR$(#A4/12.4E7+5*-3.4^3)
```

and, to illustrate the use of other functions in the argument of STR\$, try:

```
10 LET A$="3.567E7"
20 PRINT STR$(123/12.4*VAL(A$))
```

An important point to note about functions which involve arithmetic calculation is that they have the highest priority in evaluating an expression, above exponentiation. The result of applying a function to an argument is evaluated first, before any other calculations are worked out.

The other string function for numeric conversion is HEX\$. This returns the string form of the hexadecimal number equivalent to the argument. For example, HEX\$(418) returns #1A2. Since hexadecimal numbers can only be integers between 0 and 65535, any numbers outside that range will give a ?ILLEGAL QUANTITY ERROR, and any non-integer values will be automatically rounded.

A numeric function associated with those we've been dealing with is INT. This performs the same operation as occurs with integer variable and, as we've just noted, hexadecimal numbers, that of rounding down a number to the next smaller INTEGER. We use INT whenever we want a whole number value for a number, and the Oric does the same thing automatically for the arguments of functions that need an integer value. Try PRINT CHR\$(67.9), for example, and you'll get C appearing on screen. The Oric rounds down, and takes 67 as the value of the argument, since character codes can only be whole numbers. A valuable use of INT is in rounding numbers to a specified number of decimal places. This is based on the fact that, whereas INT(n) rounds down, INT(n+0.5) rounds to the nearest integer. Thus INT(3.7) gives 3, but INT(3.7+0.5) equals INT(4.2), which gives 4. Rounding 4.567 to two decimal places, for example, involves rounding .067 to .07. We can code a program line to do this by multiplying by 10 to the power of the number of d.p. we require, adding 0.5, using INT on this value, and then dividing by the same power of 10. We can produce a program that will round to a specified number of places.

```
10 REM *ROUNDING TO DP DECIMAL PLACES*
20 INPUT "ENTER A NUMBER ";N
30 INPUT "HOW MANY D.P. ";DP
40 LET ROUNDNUM=(INT(N*10^DP+0.5))/10^D
P
50 PRINT ROUNDNUM
```

We've seen how strings can be converted to numbers and vice versa, but what about manipulating strings as strings? Well, we can extract sections of strings, using the functions LEFT\$, RIGHT\$ and MID\$, and add strings together. String addition (called concatenation) uses the + sign, and simply joins the strings together, like this:

```

10 LET A$="Oric-1"
20 INPUT"What's your name ";NAME$
30 LET B$="BASIC Lesson"
40 LET SPACE$=" ": REM single space
50 LET A$=A$+SPACE$+B$
60 LET GREET$="Hi,"+NAME$+", glad you co
uld make it"
70 PRINT GREET$
80 LET BASIC$="For your"+SPACE$+A$
90 PRINT BASIC$
```

To illustrate string slicing here's an example of LEFT\$ in action:

```

10 A$="PORTAGE"
20 B$="ANDALUCIA"
30 C$="STARTLED"
40 D$="ARDENNES"
50 L$=LEFT$(A$,4)
60 AN$=LEFT$(B$,3)
70 PRINT L$
80 PRINT AN$
90 R$=LEFT$("BOAT",2)+LEFT$(D$,3)
100 R$=LEFT$(C$,4)+R$
110 PRINT R$
```

The argument of LEFT\$ is the string from which characters are to be extracted, and the value of the number of expression following the comma specifies how many characters, starting from the beginning of the string, are required.

All the Oric functions, along with the other BASIC keywords, are defined and illustrated in Chapter 9, to which you should refer whenever one you're unfamiliar with is used or mentioned in the rest of this handbook, and for additional information for the ones covered in this introduction to BASIC. Refer to Chapter 9 for RIGHTS and MID\$, and the final string function, LEN. This function returns the LENGTH of a string, i.e. the number of characters in the string.

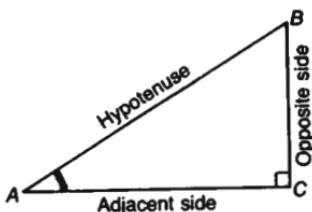
Oric BASIC has the following built-in numeric functions:

<b>ABS(n)</b>	Returns absolute magnitude of n
<b>ATN(n)</b>	Returns arctangent of n
<b>COS(n)</b>	Returns cosine of n
<b>EXP(n)</b>	Returns e raised to the power of n
<b>INT(n)</b>	Returns n truncated to integer
<b>LN(n)</b>	Returns natural (base e) logarithm of n
<b>LOG(n)</b>	Returns common (base 10) logarithm of n
<b>PI</b>	Returns value of $\pi$
<b>RND(n)</b>	Returns a random number
<b>SGN(n)</b>	Returns 0 if n zero, 1 if positive, -1 if negative.
<b>SIN(n)</b>	Returns sine of n
<b>SQR(n)</b>	Returns square root of n
<b>TAN(n)</b>	Returns tangent of n

Note that PI does not have an argument, but just returns the value of the constant  $\text{PI}(\pi)$ . Numeric functions perform difficult calculations, for which we would otherwise have to write complex programs every time we wanted to include them in calculations.

We'll take COS as an example of a numeric function. This calculates the basic trigonometric ratio of the cosine for the angle specified by the value of the expression in brackets.

In the format COS(n) the argument n may be a number, a numeric variable or an expression (which can include other numeric functions). The angle is measured in radians, and not degrees. (360 degrees =  $2\pi$  radians.) For the right-angled triangle shown below, the cosine of the angle at A will be the ratio of adjacent side/hypotenuse (AC/AB).



Unless you know about trigonometry, you are unlikely to want to use the Oric's trigonometric functions very much. They are, however, useful in defining screen positions when using the Oric's High RESOLUTION graphics mode, which we'll deal with in Chapter 7.

As well as the inbuilt numeric functions of the Oric, of which we've met COS and INT, there is a way to DEFInE your own FUNctions, using the DEF FN statement. The user (that's you!) gives a definition of the required function

at the beginning of the program, with a statement of the form:

DEF FNv(z)=numeric expression

The v is the function name (a single letter A-Z), and the z is a standard numeric variable name. The expression that follows is a numeric expression which can include any other numeric functions, and uses the variable z as part of the expression. The defined function is then called in the same way as a standard function, with the name (FN followed by whatever letter was used), and a value or parameter within brackets. Let's take a look at an example:

```
5 REM*Define Function M to convert feet
to metres
10 DEF FNM(FEET)=FEET*0.3048
20 INPUT"How MANY FEET";X
30 M=FNM(X)
40 PRINT X;"FEET EQUAL";M;" METRES"
```

We've defined a function called M, with an argument FEET, such that the function M multiplies the variable FEET by the conversion factor 0.3048 to give us the equivalent number of metres. When the number of feet is input in line 20, the value is assigned to the variable x. In line 30, the variable M (which is a different thing to the function name M) is given the value FNM(X), meaning 'look for the function called M, and calculate the resulting value, using the value of X to replace the variable name used in the argument and the expression following the equals sign'. The variable in the brackets is known as a 'dummy variable', since it only serves to define a sequence of operations, and is replaced by a variable which has had its value defined in the program when the function is called using FN. At any point in the program (as long as it's after the DEF FN statement) we can use the defined functions, using any value we wish to insert in place of the dummy variable. Here's another example:

```
10 REM *Define Functions for inches to
cm, and for area of circle*
20 DEF FNC(I)=I*2.54
30 DEF FNA(R)=PI*R^2
40 INPUT"Radius of circle (in inches)";
RAD
50 CMRAD=FNC(RAD)
60 AREA=FNA(CMRAD)
70 PRINT "Area is";AREA;" sq.cm."
```

Line 20 uses the dummy variable I to define a conversion function (FNC) for inches to centimetres, and line 30 defines FNA so as to give the area of a circle ( $\pi r^2$ ) using the dummy variable R, and the inbuilt function PI. Line 50 uses the INPUT value RAD as the parameter for FNC (replacing the dummy variable I), to get the radius in centimetres as CMRAD, which is used in turn as the parameter for FNA in line 60. Since, as with the use of PI, we can use any function in the DEFINITION, lines 50 and 60 may be combined:

```

10 REM *Define Functions for inches to
cm, and for area of circle*
20 DEF FNC(I)=I*2.54
30 DEF FNA(R)=PI*R^2
40 INPUT"Radius of circle (in inches)";
RAD
50 AREA=FNA(FNC(RAD))
60 PRINT "Area is";AREA;" sq.cm."

```

In fact, a single function will take care of the whole calculation:

```

10 REM*Define function for sq.cm. area
of circle from radius in inches*
20 DEF FNA(RAD)=(RAD*2.54)^2*PI
30 INPUT"Radius of circle (in inches)";
RAD
40 AREA=FNA(RAD)
50 PRINT "Area is";AREA;" sq.cm."

```

Notice that here we've used the same name for the actual variable used in the calculation as for the dummy variable. As many functions as we require up to the maximum of 26 allowed by single letter names can be used in a program, to avoid repetitive calculations.

Before we can consider the two final methods of data storage available in Oric BASIC, we need to introduce you to loops. Repeated operations occur frequently in programs, and there are various types of loop structure that can be used. The simple GOTO loop we've met has a serious problem – it can only be stopped by breaking into the program. The Oric has a FOR . . . NEXT structure available, which we can use whenever a sequence of program statements is to be repeated a definite number of times. A FOR . . . NEXT loop is set up using a statement of the form:

FOR v=n1 TO n2 STEP n3

which defines a variable v (a standard numeric variable) which will control

the loop, and two numeric expressions which initialise the start value (n1) and define the finish value (n2) of the loop. The STEP statement may be omitted only when the STEP value (n3) is 1. The loop variable is set to the start value, and the program continues until it meets a NEXTv statement. The value of the loop variable is checked, and if it is not greater than or equal to the finish value the STEP value is added, and the program loops back to the statement following the FOR . . . statement. Notice this is the statement, not the program line following the FOR . . . statement, so that we can have a FOR . . . NEXT loop all on one line:

```
10 FOR L=1 TO 5:PRINT L*L:NEXT L
```

If the STEP value is negative, the loop variable is checked to see whether it is less than the finish value:

```
10 FOR L=10 TO 5 STEP-1:PRINT L*L:NEXT  
L
```

The facility to use the value of the variable controlling the loop in expressions within the body of the loop is very useful:

```
10 CLS  
20 FOR LOOP=0 TO 25  
30 PRINT CHR$(65+LOOP);CHR$(97+LOOP);  
40 NEXT LOOP
```

FOR . . . NEXT loops are often used in conjunction with another way of storing data within a program, using READ and DATA. A program line, or sequence of lines, starting with DATA is used to hold strings or numbers, separated by commas. These can go anywhere in the program, and the Oric starts with the first DATA item when a READ statement is encountered, which places the number or string into the variable which follows READ (which must be the appropriate type for the DATA). A pointer is then placed against the next DATA item, which will be the one used when the next READ instruction comes along. RESTORE will reset the pointer back to the start of the DATA items. The principle is shown in the next program, which uses a loop to READ and PRINT the string DATA, and then RESTORES the DATA pointer, so that the days can be READ and PRINTed again. Notice that the loop variable is not specified after the NEXT statements, which is permissible, but potentially confusing, and that string DATA items do not need quotes (although they may have them, and must if the strings are to contain leading or trailing spaces, commas, or colons).

```

10 REM*READ and DATA*
20 FOR D=0 TO 6
30 READ D$
40 PRINT D$
50 NEXT
60 REM... lots more program
70 ,
2000 DATA MONDAY,TUESDAY,WEDNESDAY,THUR
SDAY
2010 REM... more program in between
3000 DATA FRIDAY, SATURDAY,SUNDAY
4000 REM.. can do it again
4010 RESTORE
4020 FOR K=1 TO 7
4030 READ DAY$ :PRINT DAY$ :NEXT

```

The final form in which the Oric can hold strings or numbers is as arrays. These are normally set up using the DIM statement, and you should turn to the entry for DIM in Chapter 9 before going any further.

OK, you now know what an array consists of. The usefulness of arrays is not only that they allow storage of similar items of data without the definition of an individual variable for each item separately, but they are also indexed by their subscript number(s). This allows us to cross-reference and perform operations on the elements of arrays easily. Here's a short example which demonstrates the way in which arrays enable otherwise difficult operations to be programmed simply:

```

10 INPUT "HOW MANY NAMES";N$
20 DIM NAME$(N$):DIM AGE(N$)
30 CLS:SUM=0
40 FOR K=1 TO N$
50 : PRINT "ENTER NAME NO."K
60 : INPUT NAME$(K)
70 : PRINT NAME$(K) "'S AGE ?"
80 : INPUT AGE(K):SUM=SUM+AGE(K)
90 : CLS
100 NEXT K
110 FOR K=N$ TO 1 STEP -1
120 : PRINT:PRINT NAME$(K) " IS";

```

```

130 : PRINT AGE(K)" YEARS OLD"
140 NEXT K
150 PRINT:PRINT "TOTAL AGE"SUM" YEARS"
160 PRINT"AVERAGE AGE"SUM/Nx" YEARS
170 END

```

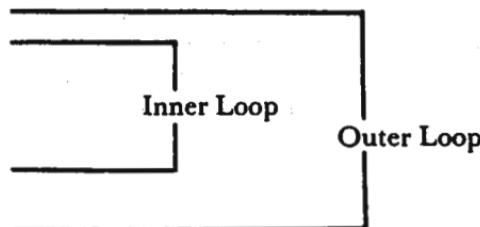
We can use the INPUT variable Nx to DIMension two arrays to the size required, and to set the FOR..NEXT loops identically. Note that humans often find it easier to consider arrays as having subscripts starting at 1, as here. This just leaves the array elements NAMES(0) and AGE(0) unused. The value of the loop variable is used to access each pair of array elements in sequence, once for INPUT and once for PRINTing out. The total age can be easily SUMmed at the same time.

Loops within loops are known as *nested* loops. We can have as many levels of nesting as we want, up to a maximum of 10. The important thing is that the loops must be correctly nested, with each loop entirely within the one outside it:

```

30 FOR A = 1 TO 6
40 FOR B = 1 TO 3
.....
80 NEXT B
.....
120 NEXT A

```



Nested loops can be used to access and initialise multi-dimensional array elements:

```

10 DIM Nx(10,3)
20 CLS
30 FOR K=1 TO 10
40 : FOR J=1 TO 3
50 : Nx(K,J)=K^J
60 : PRINT Nx(K,J)
70 : NEXT J
80 : PRINT
90 NEXT K

```

DATA can also be READ into arrays:

```
10 DIM M$(12):DIM D$(12)
20 FOR K=1 TO 12
30 : READ M$(K)
40 : READ D$(K)
50 NEXT K
60 INPUT "WHICH MONTH (1-12)"; MNTH
70 PRINT M$(MNTH); " HAS "; D$(MNTH); " DAYS"
100 DATA JANUARY,31,FEBRUARY,28,MARCH,31,APRIL,30,MAY,31,JUNE,30
110 DATA JULY,31,AUGUST,31,SEPTEMBER,30,OCTOBER,31,NOVEMBER,30,DECEMBER,31
```

The Oric stores the address of the line or statement after the FOR .. TO statement (to which program control passes back on encountering a NEXT statement) in a stack, a last-in first-out pile. This is the reason the variable name may be omitted (the Oric doesn't mind, but including the variable helps humans read a program listing), and also why loops must be correctly nested, since the Oric just looks at the address on the top of the pile. After a loop is completed, the address is removed, and a subsequent NEXT will activate a jump to the address now on the top of the pile. The stack can only hold 10 addresses, and more than 10 loops will give an ?OUT OF MEMORY ERROR message.

The same stack is used for the other type of loop available on the Oric, the REPEAT .. UNTIL structure. We'll deal with this in the NEXT chapter.

# 4 Loops beyond compare

An alert reader might have noticed that the FOR...NEXT loop structure implies that the Oric is capable of making decisions about whether one number is greater than another, since the loop variable has to be checked against the end value of the loop before the Oric can decide whether to run through the loop again or continue with the next statement.

Well, we can also do this in BASIC, and in fact this capability is crucial to programs, since decisions can be made, and the sequence of actions shunted on to different sequences of operations. Introducing REPEAT...UNTIL loops brings the question to the forefront, since REPEAT...UNTIL structures REPEAT a sequence of operations UNTIL a condition is met.

Conditions are tested by means of conditional operators:

=	Equal to
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

These operate much as you would expect for numeric values. Two things need to be remembered, however. The first is that, for negative numbers,  $-6$  will be less than  $-5$ , and so on, and the second is that the Oric, like any computer, is not totally accurate in its calculations. Where complex numerical operations have been performed to produce a number to be tested for equality with another, there is the possibility that the least significant figure of the value will be in error. In such cases it is better to take the ABSOLUTE (positive) value of the two figures, and then test that the difference is less than an acceptable amount. Refer to the ABS section in Chapter 9, and see the program below:

```
10 LET NINECUBE=9*9*9
20 IF NINECUBE=9^3 THEN PRINT "TRUE" ELSE PRINT"FALSE"
30 IF ABS(NINECUBE-9^3)<1E-7 THEN PRINT
    "TRUE" ELSE PRINT"FALSE"
```

What's the IF...THEN...ELSE statement in line 20? It tests whether the variable NINECUBE equals  $9 \uparrow 3$ , and decides whether the condition is true or false. IF the condition is true THEN it performs the operations specified after THEN, and goes on to the next program line, ignoring ELSE and anything that comes after it. IF the condition is false, THEN and the following statement(s) are ignored, and the statement(s) following ELSE are performed before moving to the next line. So we have a program structure that says: IF (condition is true) THEN (perform true task) ELSE (perform false task).

In line 20, however, the condition will evaluate as false, and PRINT out "FALSE", in line 30 the condition will evaluate as true, and "TRUE" will be PRINTED.

The ELSE part of an IF...THEN construct is optional. When there is no ELSE statement, control will pass on to the next line, with the statements following THEN having been executed if the condition was true, and ignored if the condition was false.

```
10 A=2:INPUT"ENTER A VALUE 0-5";C
20 IF C>A THEN PRINT "C>A"
30 PRINT "THIS IS ALWAYS PRINTED"
40 GOTO 10
```

If we wanted to use ELSE, we might write a line 20 as below:

```
10 A=2:INPUT"ENTER A VALUE 0-5";C
20 IF C>A THEN PRINT "C>A" ELSE PRINT "
C<A"
30 PRINT "THIS IS ALWAYS PRINTED"
40 GOTO 10
```

However, if you enter 2 for the value of C, we'll get "C<A" PRINTed out. Is this the result we want, you may well ask? Well, we are now in the realm of logic, and we must watch our complements. One or the other of the THEN... and ELSE... statements must be performed. Computers are sticklers for excluding the middle of a sorites, so it's either one thing or the other. What we need to do is recognise the true complements, since it is our mistake, not the Oric's. The opposite, or complementary condition, to  $>$  is  $\leq$ , not  $<$ . The relationships are:

$<>$	complements	=
$<$	complements	$\geq$
$>$	complements	$\leq$

So our logically correct output would be produced by the following:

```

10 A=2:INPUT"ENTER A VALUE 0-5";C
20 IF C>A THEN PRINT "C>A" ELSE PRINT "
C<=A"
30 PRINT "THIS IS ALWAYS PRINTED"
40 GOTO 10

```

With the use of an IF...THEN conditional test, we can produce the equivalent of a FOR...NEXT loop, called a counter loop. The program below sets a counter C, increments it by 1 in line 40, and checks the value in line 50, after the body of the loop. A variation of the IF...THEN format is used. If the statement after THEN is GOTO (line number), it may be replaced by IF...GOTO, omitting the THEN, or by IF...THEN (line number), omitting the GOTO. The Oric interprets all these formats identically, GOTO may similarly be omitted after ELSE, if desired.

```

10 REM Counter Loop
20 C=1
30 PRINT"COUNTER VALUE "C
40 C=C+1
50 IF C<4 GOTO 30
60 END

```

If the condition C<4 is true, the program will loop back to line 30. Notice that with this condition the loop will execute three times. If we replace the condition in line 50 with C<=4, it will then execute four times, and be an exact equivalent of a FOR C=1 TO 4 loop. Using ELSE would give us:

```

10 REM Counter Loop
20 C=1
30 PRINT"COUNTER VALUE "C
40 C=C+1
50 IF C<=4 GOTO 30 ELSE END

```

The REPEAT...UNTIL construct is started with a REPEAT statement, and the program executes the statements in the body of the loop UNTIL a condition is set. If the condition is true, the loop terminates, and the program continues with the statement after UNTIL (condition). If the condition is false, the program loops back to the statement after REPEAT. This can be used for the same sort of counter loop:

```

10 REM REPEAT..UNTIL Loop
20 C=1
30 REPEAT

```

```
40 PRINT"REPEAT "C
50 C=C+1
60 UNTIL C=4
```

However, the UNTIL condition can refer to any condition whatsoever, and is an extremely flexible structure. The following program uses a REPEAT...UNTIL loop to calculate the factorial of a number:

```
10 REM*FACTORIALS*
20 INPUT"ENTER AN INTEGER";Jx
30 Nx=Jx-1:FACT=Jx
40 REPEAT
50 : FACT=FACT*Nx
60 : Nx=Nx-1
70 UNTIL Nx=0
80 PRINT"FACTORIAL";Jx;"=";FACT
```

They can be useful in data entry routines:

```
10 REPEAT:UNTIL KEY$="G"
20 PRINT"END
```

```
10 REPEAT
20 INPUT"ENTER A NUMBER BIGGER THAN 9";
Nx
30 UNTIL Nx>9
40 REM REST OF PROGRAM
```

The Oric assesses the truth or falsity of a condition by comparison between numbers and, since it knows of nothing else, also assigns a numeric value (-1) to represent true and considers 0 as false. There are system constants TRUE and FALSE in Oric BASIC to represent these values. Whilst they only substitute for numeric values, their use can help make a program more comprehensible to humans. Take the endless loop:

```
10 REPEAT
20 REM PRINTS FOREVER
30 PRINT "ONWARDS"
40 UNTIL FALSE
```

or we could replace the condition  $N\% = 0$  with  $N\% = \text{FALSE}$  in the factorials program:

```

10 REM*FACTORIALS*
20 INPUT"ENTER AN INTEGER";Jxz
30 Nz=Jxz-1:FACT=Jxz
40 REPEAT
50 : FACT=FACT*Nz
60 : Nz=Nz-1
70 UNTIL Nz=FALSE
80 PRINT"FACTORIAL";Jxz;"=";FACT

```

It is important to note that whilst the Oric always takes 0 as FALSE, and in evaluating a conditional statement, takes -1 as TRUE, it will accept any non-zero number as TRUE when testing numeric variables.

```

5 REPEAT
10 INPUT A
20 IF A THEN PRINT"A IS NOT ZERO"ELSE P
RINT "A=0, WHICH IS FALSE"
30 UNTIL A=FALSE

```

Characters and strings may be compared in conditional tests, and we often need to compare strings to find out if they are equal. Simple comparison of strings to search lists or test input poses little problem since equality is the only thing to be tested:

```

10 PRINT"IS YOUR NAME FRED"
20 INPUT"ANSWER YES OR NO";A$
30 IF A$="YES" THEN PRINT "HI,FRED?"
40 IF A$="NO" THEN PRINT "COULDN'T YOU
LIE A LITTLE ?"
50 PRINT "WELL, WHOEVER YOU ARE, WHAT'S T
HE PASSWORD?"
60 INPUT PASS$
70 IF PASS$<>"FX123"THEN PRINT "YOU'RE
AN IMPOSTOR":END
80 PRINT "AVE FRATER"

```

However, care must be taken when comparisons are being made between strings for ordering purposes. Comparison between strings is performed by looking at each ASCII character code in turn, and whereas in comparison a single character difference just tells us what we need to know, in sorting alphabetically we have the problem of upper and lower case letters, and

possibly numbers as well. First consider a simple sorting program to put numbers into ascending order. Each item in a list (stored in the array NUM) is compared with each other item in the list, and swapped with any higher number it meets. Repeating this process for each number in the list results in the desired ordering. Note the use of the variable TEMP to transfer array elements:

```

5 REM*NUMERIC BUBBLE SORT**
10 CLS:LET J=15 'Number of items
20 DIM NUM(J)'Array for numbers
30 'Example numbers generated here"
40 FOR K=0 TO J
50 : LET NUM(K)=RND(1)*1000
60 : PRINT NUM(K)
70 NEXT K
99 '
100 REM*SORT*ROUTINE*
101 '
110 FOR M=1 TO (J-1)
120 : FOR N=M TO J
125 :      REM If correct order
           already, then skip
130 :      IF NUM(M)<NUM(N) THEN 170
135 :      REM Incorrect order, so swap
140 :      TEMP=NUM(M)
150 :      NUM(M)=NUM(N)
160 :      NUM(N)=TEMP
170 :      NEXT N
180 NEXT M
189 '
190 REM**ENDSORT**
191 '
200 REM *Print sorted list*
210 CLS:PRINT "SORTED LIST:"
220 FOR K=1TO J
230 : PRINT NUM(K)
240 NEXT K

```

The next program is a string version of the bubble sort given above. The only difference is the use of a flag to prevent unnecessary comparisons being

made when a pass has produced no swaps (elements in correct order). Key this in as listed to demonstrate the ordering sequence that strings containing assorted types of characters produce, and then use the sort routine in a program of your own (CAPS INPUT or DATA only!) to sort any alphabetic lists:

```

5 REM **ALPHABET SORT**
10 CLS
15 'Initialise string array. Replace with INPUT routine if desired
20 A$(0)="1"
30 A$(1)="B"
40 A$(2)="ab"
50 A$(3)="Ab"
60 A$(4)="zz"
70 A$(5)="aZ"
80 A$(6)="D"
90 A$(7)="d"
100 A$(8)="X"
110 A$(9)="A2"
120 A$(10)="bwert"
125 '
130 REM **PRINT LIST**
140 FOR K=0 TO 10
150 : PRINT A$(K); " ";
160 NEXT
165 '
170 REM**SORT**
180 FOR K=0 TO 9 'items-1
190 : FLAG=FALSE
200 : FOR M=K TO 10
210 : IF A$(K)<A$(M) THEN 260
220 : TEMP$=A$(M)
230 : A$(M)=A$(K)
240 : A$(K)=TEMP$
250 : LET FLAG=TRUE
260 : NEXT M
270 : IF NOT FLAG THEN K=99
280 NEXT K

```

```

285 '
290 REM**PRINT SORTED LIST**
300 PRINT:PRINT
310 FOR K=0 TO 10
320 : PRINT A$(K)
330 NEXT
340 END

```

REPEAT...UNTIL and FOR...NEXT loops may also be nested. Note the use of indents (which need colons at the beginning of the line to stop the Oric stripping off leading spaces) to help make the loop structure of the program clear.

```

10 REM Indented Loops
20 PRINT "ORIC LOOPS"
30 FOR F=1 TO 3
40 : PRINT TAB(15+F);"FOR-NEXT" F
50 : PRINT TAB(15+F);"NOW REPEAT"
60 : C=1'sets counter
70 : REPEAT
80 :     PRINT TAB(15+F+C)"REPEAT " C
90 :     C=C+1
100 :    UNTIL C>2
110 :    PRINT TAB(15+F)"OUT OF REPEAT"
120 :    PRINTTAB(15+F)"NOW NEXT F"
130 NEXT

```

There is an important programming point to note with regard to both REPEAT...UNTIL and FOR...NEXT loops. They must only be exited correctly, by satisfying the exit condition. Jumping out of a loop with a GOTO will leave the address of the loop on the stack, and if you do it repeatedly, the stack will fill up and you'll get an OUT OF MEMORY ERROR. FOR...NEXT loops can be exited by setting the loop variable to a value that will satisfy the exit test:

```

5 PRINT "ENTER DATA, ENTER -999 TO END"
10 DIM A(20):DIM D(20)
20 FOR F=1 TO 20
30 INPUT A(F)
40 IF A(F)=-999 THEN A(F)=0:F=21:GOTO 60

```

```

50 LET D(F)=A(F)*31
60 NEXT F
70 PRINT"DATA INPUT ENDED"

```

The entry of the dummy or sentinel value which terminates data entry satisfies the condition at line 40, which resets the array element to 0, sets F to 21 and then passes control to line 60. Since F is greater than 20, the loop is terminated. Such a use of the GOTO instruction, passing control forward in a program to bypass blocks of program lines is held by the purist (and, some would say, authoritarian, if not neo-fascist) programming gurus to be the only time use of GOTO is to be condoned. The middle path is advised for our readers.

REPEAT loops may be similarly terminated by the use of flags, originally set to TRUE or FALSE prior to the loop, and inverted when an exit condition is met. A GOTO may also be required if code is to be bypassed. There is, however, another option open for the programmer, although it is again of dubious virtue in the eyes of some structured programming fanatics. This is to use PULL, which takes the top address off the stack, and enables a GOTO jump out of a loop to be coded with immunity, if not satisfaction, since it must be admitted even by the most liberal of us that it's habitual use is not good practice.

```

10 REM *** PULL ***
20 A=9
30 REPEAT
40 B=A
50 REPEAT
60 PRINTB;
70 B=B-1
80 IF B<0 THEN PULL :GOTO 100
90 UNTIL B=0
100 A=A-1
110 PRINT
120 UNTIL A=-5

```

Conditional tests can involve more than one condition, combined (using the logical operators AND, OR and NOT). We can, for example, have a line which reads:

IF N<10 AND A=B THEN... .

The Oric will check the two conditions separately, and then check whether the combination of the two is true, according to the truth tables for AND, OR and NOT. AND works much as in English, in that IF condition 1 is TRUE AND condition 2 is TRUE THEN the combination of the conditions is also TRUE. If

either condition is FALSE, the combination is FALSE. The following program produces the truth table for AND.

```

10 LET L(1)=TRUE
20 LET L(2)=FALSE
30 LET L$(1)="TRUE"
40 LET L$(2)="FALSE"
50 FOR K=1 TO 2
60 : FOR H=1 TO 2
70 : PRINT L$(K)" AND "L$(H)"=";
80 : IF L(K) AND L(H) THEN PRINT
      L$(1) ELSE PRINT L$(2)
90 : PRINT
100 : NEXT H
110 NEXT K

```

Again, the program has been laid out so that structure is clear. Since the Oric uses numeric values for TRUE and FALSE, we can utilise these in numeric expressions. The modified line 80 in the program below uses the value (-1 or 0) that the Oric derives from the AND expression, plus 2, to give the appropriate array subscript (1 or 2) for L\$.

```

10 LET L(1)=TRUE
20 LET L(2)=FALSE
30 LET L$(1)="TRUE"
40 LET L$(2)="FALSE"
50 FOR K=1 TO 2
60 : FOR H=1 TO 2
70 : PRINT L$(K)" AND "L$(H)"=";
80 : PRINT L$(2+(L(K)AND L(H)))
90 : PRINT
100 : NEXT H
110 NEXT K

```

Take care with the complex brackets. The best way to check that you haven't left one out somewhere is to count left and right brackets across the expression, and check that there are the same number of each.

An expression using OR returns a value of TRUE if either of the conditions it joins is TRUE, and likewise if both are TRUE. Only if both are FALSE does it produce a FALSE result. NOT placed before a condition reverses TRUE and FALSE values.

Multiple conditions can be combined, using both OR and AND. Here's an example:

```

10 L(1)=TRUE
20 L(2)=FALSE
30 L$(1)="TRUE"
40 L$(2)="FALSE"
50 FOR K=1 TO 2
60 FOR H=1 TO 2
70 FOR J=1 TO 2
80 PRINT L$(K) AND L$(H) OR L$(J)
IS ";
90 PRINT L$(2+(L(K) AND L(H) OR L(J)))
100 PRINT
110 NEXT :NEXT :NEXT

```

As contrast to the earlier programs, notice how much more difficult it is to read this program, in the absence of LETS, indents and loop variable names. Even this is better than the sort of code often seen, which is written as compactly as possible. Here's the same program in condensed form:

```

10 L(1)=TRUE :L(2)=FALSE :L$(1)="TRUE" :L$(
2)="FALSE" :FOR K=1 TO 2
20 FOR H=1 TO 2:FOR J=1 TO 2:PRINT L$(K
)" AND "L$(H)" OR "L$(J)" IS ";
30 PRINT L$(2+(L(K) AND L(H) OR L(J)))::
PRINT:NEXT :NEXT :NEXT

```

Brackets can be used with multiple conditional expressions to ensure that the meaning you intend is understood by the Oric. Try bracketing the first two expressions, and then the second and third, running the program each time to see the different interpretation this gives.

The use of conditional GOTOS can be enhanced using an ON...GOTO statement. ON is followed by a variable or expression, which gives an integer value, defining which of the line numbers following the GOTO statement is activated:

```

10 INPUT"ENTER 1,2 OR 3,PLEASE";N
20 ON N GOTO 150,200,300
140 REM
150 PRINT "LINE 150 FROM N=1":GOTO 10
190 REM
200 PRINT "LINE 200 FROM N=2":GOTO 10
290 REM
300 PRINT "LINE 300 FROM N=3":GOTO 10

```

ON may also be used with the last program structure we are going to introduce, the GOSUB..RETURN format. The GOSUB (line number) instruction acts like a GOTO, in that it transfers the program control to the specified line number. However, unlike GOTO, the GOSUB instruction stores the current address before jumping. The address is placed on the stack and, upon encountering a RETURN statement, program control is RETURNed to the statement following the GOSUB call. The program below illustrates the principles:

```
10 REM *** GOSUB ***
20 CLS:REM GET NUMBER
30 UP=9:LO=1:GOSUB 1000
40 REM CALL MENU ROUTINE
50 X=A:GOSUB 2000
60 REM ARE YOU BORED
70 GOSUB 3000
80 IF BO THEN STOP ELSE GOTO 20
90 END
1000 REM NUMBER INPUT ROUTINE
1010 PRINT"TYPE IN AN INTEGER BETWEEN";
LO;"AND";UP:PRINT
1020 REPEAT
1030 PRINTCHR$(11)::INPUT A$
1040 A=VAL(A$)
1050 UNTIL A>=LO AND A=<UP AND A=INT(A)
1060 RETURN
2000 REM MENU & OPTION ROUTINES
2010 PRINT:PRINT SPC(16);CHR$(129); "***"
MENU***"
2020 PRINT:PRINTCHR$(130); "PRESS 1 FOR
FACTORIAL(" ;X;")"
2030 PRINT:PRINTCHR$(130); "PRESS 2 FOR
";X;" SQUARED"
2040 REM USE NUMBER ROUTINE TO SELECT
2050 UP=2:LO=1:GOSUB 1000
2060 ON A GOSUB 2100,2200
2070 RETURN
2100 REM CALCULATE FACTORIAL BY REPEATE
D ADDITION
```

```

2110 Y=X:S=1:GOSUB 2500
2120 PRINT "FACTORIAL ";X;" IS ";S
2130 RETURN
2200 REM CALCULATE SQUARE
2210 PRINT "SQUARE OF ";X;" IS ";X**X
2220 RETURN
2500 REM RECURSIVE SUBROUTINE
2510 IF Y=0 THEN RETURN
2520 S=S*Y:Y=Y-1
2530 GOSUB 2500
2540 RETURN
3000 REM ARE YOU BORED
3010 PRINT "ARE YOU BORED (Y/N)?"
3020 IF KEY$="Y" THEN BO=TRUE: GOTO 305
0 ELSE WAIT 1
3030 IF KEY$="N" THEN BO=FALSE:GOTO 305
0
3040 GOTO 3020
3050 RETURN

```

While we are on the subject of subroutines, perhaps we should take a look at why good programmers are so keen on using them. Well, firstly there are the obvious advantages of saving space by keeping what would otherwise be oft-repeated program segments in one place. This can also save programming time since you don't have to continually retype the same lines at each desired occurrence. OK, so we've seen the obvious gains to be made in terms of time and space, let's progress to the subtler, more profound advantages to be gained, in terms of philosophy and aesthetics, from using subroutines.

Many people are, in fact, so enamoured of these methods that they almost always emphasise the more intellectual aspects of structured programming, usually at great cost to the clarity of their arguments. Pragmatically, though, it is worth trying to 'structure' your thoughts, at least, and think of the task to be performed in terms of smaller, simpler subtasks. This way of thinking allows you to write short sections of code which are free of confusion and complete in their own right, although of course 'no subroutine is an island'.

When you come back to a program some time after its inception it can be very awkward to track down the exact line in which something happens if the programming task was originally conceived as an amorphous, interminable and interleaved mass. Getting back to practicalities, it doesn't

really matter whether you make these subsections of code into fully fledged subroutines or not. In fact, it is arguably better not to do so if they are only used once in the whole program. Really all that can be said about the merits or otherwise of structured programming is that you should have a clear idea of what you are doing at each stage in your program and keep unrelated tasks in different lines so that you can figure it out later. Keeping your sections or modules well separated and clearly marked with REM statements will help you develop into a less frustrated programmer.

# 5 Down memory lane

One of the great virtues of BASIC is that it allows us to state what we wish to be done and, providing we put the right instructions into a program, we don't have to worry about how it's done. However, there are instructions in BASIC which allow us to interact directly with the Oric's memory. We first need to know what's in the memory and how it's arranged before we can do anything with it, so here goes.

You first need to know something of the binary number system. This uses sequences of 0's and 1's to represent numbers in a computer's memory. Each memory location can be thought of as a bank of eight switches, which may be on (set), representing a 1, or off (unset), representing 0. Each location thus holds a sequence of eight Binary digITS or bits, and the eight together make up a byte. Within a byte the bits are numbered 0 to 7 right to left, and each bit, when set, holds a value twice that of the one to the right of it. Bit 0 can be either 0 or 1, and represents those values, whereas bit 1 represents 2 if set, and zero if unset. The value held in the sequence of eight bits is the sum of the numbers represented by the set bits. The largest value that can be held in a sequence of eight bits is 255, and the smallest, of course, is zero:

Bit	7	6	5	4	3	2	1	0
Binary	1	1	1	1	1	1	1	1
Value	128	64	32	16	8	4	2	1 = 255 decimal

The binary number 01101101, for example, would be:

Bit	7	6	5	4	3	2	1	0
Binary	0	1	1	0	1	1	0	1
Value	0	+64	+32	+0	+8	+4	+0	+1 = 109 decimal

The binary system works in powers of two (not powers of 10, as our usual decimal system does). This can be illustrated by the following program.

```
10 REM Powers of 2
20 FOR POWER=0 TO 7
30 PRINT "2 to the power";POWER;"=";2^POWER
40 NEXT POWER
```

Here's a program that converts binary to decimal, and vice versa:

```

10 REM Binary/Decimal conversion
15 '*****
16 ' MAIN PROGRAM
17 '*****
20 PRINT"PRESS B FOR BINARY TO DECIMAL"
30 PRINT "PRESS D FOR DECIMAL TO BINARY"
"
40 GET M$ : IF M$<>"D" AND M$<>"B" THEN 4
0
50 IF M$="D"THEN GOSUB 500 ELSE GOSUB 1
000
60 PRINT "ANOTHER NUMBER ? (Y/N)"
70 GET M$ : IF M$="Y"THEN CLS:GOTO20 ELSE
END
80 '*****
85 '***END MAIN*****
90 '*****
490 '
500 '*****
501 '      DECIMAL TO BINARY
502 '      SUBROUTINE
503 '*****
504 '
510 CLS:INPUT"ENTER A DECIMAL NUMBER";N
x
520 PRINT:PRINT N$;:B$=""
530 REPEAT
540 :I=INT(N$/2)
550 :    BIT=N$-2*I
560 :    IF BIT=0 THEN B$="0"+B$ELSE
        B$="1"+B$
570 :    N$=I
580 UNTIL N$=0
590 PRINT "IS "B$" IN BINARY"
600 RETURN
610 '
620 '*****ENDSUB*****

```

```

630 ,
1000 '*****
1001 '      BINARY TO DECIMAL
1002 '      SUBROUTINE
1003 '*****
1004 '
1010 CLS:INPUT"ENTER A BINARY SEQUENCE"
;B$
1020 N=0:P=0
1030 FOR J=LEN(B$)TO 1 STEP-1
1040 :     BIT$=MID$(B$,LEN(B$)-P,1)
1050 :     N=N+VAL(BIT$)*2^P
1060 :     P=P+1
1070 NEXT J
1080 PRINT B$" =DECIMAL"N
1090 RETURN
1100 '
1110 '*****ENDSUB*****

```

Notice that the program is structured with a main program module, from which the appropriate subroutine is called according to the user input after the menu has been presented. The program also shows how a listing can be made more readable by the inclusion of REM statements to break up the program into its modules, and the use of indents within loops. The decimal to binary routine uses a REPEAT...UNTIL loop to reduce the number by a power of two each time through the loop, assigning a 0 or 1 to the binary string B\$ according to whether a remainder is present or not after division by two. The loop terminates when N% = 0 and there's no more number to work with. The binary to decimal routine uses a FOR...NEXT loop to check each bit of the binary string in turn from the right, increasing the power of two (P) by which the VALUE of the bit is multiplied at each pass through the loop.

The organisation of the Oric's memory is shown diagrammatically in Appendix 5. The 48k and 16K versions have the same arrangement of memory, apart from a missing chunk in the middle of the 16K memory, between #4000 and #C000 hex (16384 and 49152 decimal). Referring to this diagram, and using your trusty Oric to convert hexadecimal (PRINT #4000, etc.), you'll see that the top 16384 locations, above #C000, are Read Only Memory (ROM) on both the 16K and the 48K machines. This is where the BASIC interpreter and arithmetic routines are held in permanent fixed memory stored in the ROM chips of the Oric.

The ROM memory cannot be altered (written to), although, as its name implies, it can be read from so that we can find out what it contains. The contents of this area of memory are machine-code instructions and data, and the process of understanding the contents of memory when read out (called disassembly, in contrast to assembly, which you can investigate in Chapter 10) is a complex one.

The rest of the Oric's memory consists of Random Access Memory, which can be both written to and read from, so that we can insert into any location a number between 0 and 255, and also discover the value contained in any location. The number stored in a location can represent part of a number, a BASIC keyword, a character code, or part of an address. If you're programming in machine code, it may also be a machine-code instruction. The interpreter stored in ROM decides what any specific number represents according to context and placement in memory.

Starting from the bottom of RAM we have five pages of memory, each of 256 locations, dedicated to specific purposes. The first page (Page Zero) contains information on the current state of affairs within the Oric needed by the 6502 Central Processing Unit (CPU) chip, such as the addresses of the start and end of the BASIC program, pointers to string variable storage, and so on.

Page 1 is a stack for the use of the arithmetic routines, storing numbers and intermediate values involved in the current calculations.

Page 2 stores run-time or system variables which hold values needed to keep track of the operation of BASIC such as cursor positions, CAPS/lower case, keyclick on/off, etc.

Page 3 is dealt with in Chapter 11 since it holds the address for Input/Output between the Oric and the external world and the addresses for transfer of data between the various dedicated chips of the Oric (see the diagram in Appendix 11).

Page 4 addresses between #0400 and #0420 are available for the user's machine-code programs, and the rest of the page is reserved for system use.

Addresses #0500 upwards are the memory locations for storage of your BASIC programs and variable values. We'll look at how this section of memory is organised below. The BASIC program grows upwards as it lengthens, taking with it the variables area which sits on top.

Locations above #9800 (#1800 for the 16K Oric) store the character sets and the screen display. The TEXT screen (which is the one that appears when we switch on and is the one which we've been using) takes much less space than the HIRES (high resolution) screen. The different screen modes will be dealt with in Chapter 7, where we approach the whole question of graphics and screen displays. However, we'd better say here that, to enable switching between the two modes, the whole area above #9800 (#1800 on the 16K) is reserved. If only TEXT output is required on screen in the course of a program the GRAB command can be used to release the area from #9800

to #B400 (#1800 to #3400 on the 16K) for use by BASIC. This is reversed by the RELEASE command when the HIRES screen is required once again. The standard and alternate (LORES 1) character sets are stored in two sequences of memory locations, one of 1024 bytes, storing the 128 characters for the standard character set, and one of 896 bytes, holding the 112 characters of the alternate set. Each character is stored as a sequence of 8 bytes, which give the bit pattern defining the pattern of dots displayed on the screen. Chapter 7 explains how this works. All characters are stored, although the non-PRINTing ones have a null number stored in their character definition bytes, just to make sure that they don't PRINT.

Characters stored in this area may be redefined by the user to form any special characters required in a program. This is accomplished by writing different values into the addresses in memory which form a specific character. The screen memory is also covered in Chapter 7. Above the screen memory storage is an area of spare memory available for machine code or Input/Output use.

The BASIC instruction to read the contents of a byte is PEEK. The format is PEEK(addr.), where addr. is a hexadecimal or decimal number specifying the address. PEEK returns the value stored in the byte as a decimal number. PRINT PEEK(1280), for example, will PRINT the value contained in the byte at memory location 1280. Precisely the same result is obtained by PRINT PEEK (#500).

POKE writes data to a byte of memory. POKE addr.,i puts the value of i (0 to 255) into the byte at the memory location specified by addr. POKE 1600,134 puts the value 134 into location 1600. The values may both be specified in hexadecimal notation: POKE #640,#56.

Integer values up to 65535 can be stored in two bytes, and the Oric uses this system for address locations which need to be stored in memory. The two bytes hold the value in the form (value of first byte) plus (256\* value of second byte). There are instructions in BASIC for reading and writing such values directly, without calculation. These are DEEK(addr.) which returns the value stored in addr. and addr.+1, and DOKE addr.,i which places the integer value i (0 to 65535) into the byte specified by addr. and the following byte.

Using these instructions, we can take a look at how a BASIC program is stored in the Oric memory. The program below will PRINT the memory address, the PEEKED value contained in the byte specified by that address, and the character corresponding to that value if the character is PRINTable. Line 40 merely breaks up the data into pages that will fit on the screen. Line 60 right justifies the number contained in the byte using SPC.

```

10 REM PROGRAM STORAGE
20 LET MEM=#500
30 CLS:FOR K=1TO200
40 IF INT(K/25)=K/25 THEN PRINT"PRESS KEY FOR MORE":GET A$
50 N=K+MEM:M=PEEK(N):L=LEN(STR$(M))
60 PRINT N;SPC(10-L);M;
70 IF M<129 AND M>32 THEN PRINT SPC(4);
CHR$(M) ELSE PRINT" "
80 NEXT K

```

The display for the first three screens produced is as follows:

1281	23
1282	5
1283	10
1284	0
1285	157
1286	32
1287	80 P
1288	82 R
1289	79 O
1290	71 G
1291	82 R
1292	65 A
1293	77 M
1294	32
1295	83 S
1296	84 T
1297	79 O
1298	82 R
1299	65 A
1300	71 G
1301	69 E
1302	0
1303	38 &
1304	5

PRESS KEY FOR MORE  
1305 20

1306	0	
1307	150	
1308	32	
1309	77	M
1310	69	E
1311	77	M
1312	212	
1313	35	#
1314	53	5
1315	48	0
1316	48	0
1317	0	
1318	54	6
1319	5	
1320	30	
1321	0	
1322	148	
1323	58	:
1324	141	
1325	32	
1326	75	K
1327	212	
1328	49	1
1329	195	
PRESS KEY FOR MORE		
1330	50	2
1331	48	0
1332	48	0
1333	0	
1334	102	f
1335	5	
1336	40	c
1337	0	
1338	153	
1339	32	
1340	215	
1341	40	c
1342	75	K

1343	207	
1344	50	2
1345	53	5
1346	41	)
1347	212	
1348	75	K
1349	207	
1350	50	2
1351	53	5
1352	32	
1353	201	
1354	32	

You might find this a little confusing, but all will be made clear! The first address PEEKed is #501 (decimal 1281), following the zero stored in address 1280, marking the start of the BASIC program area. The value stored in locations 1281 and 1282 is a pointer to the memory address containing the start of the next program line. If you enter the command PRINT DEEK(1281) as a direct command with this program in memory, you will get 1303 on screen. Looking at this location, you will see that it follows a zero in address 1302 which is used by the Oric to separate program lines. Bytes 1303 and 1304 are the first two bytes of line 20, forming the pointer to line 30. All the program lines are linked by these pointers, and the Oric can 'skip' along the program lines to find the line specified by a GOTO or GOSUB statement with maximum efficiency.

Addresses 1283 and 1284 hold a two-byte value in the same format (byte 1 + 256\*byte2) which gives the line number, in this case 10. Location 1285 holds the value 157, which is the tokenised form of REM. The Oric stores each BASIC keyword it recognises when the program line is placed in memory in a form that only occupies a single byte of memory. Appendix 12 gives a list of BASIC keywords and their token values.

After the tokenised REM we find a space (character code 32) followed by the character codes for PROGRAM, then a space, then MEMORY. The zero terminates the line. Addresses 1303 and 1304 store the pointer to the next line, and 1305/6 the line number. Although the & character is displayed, the value stored in the byte is a numeric value, not a character code, which is clear (to the Oric) from the context. Reference to Appendix 12 will tell you that the value of 150 stored at location 1307 is the token for LET. This is followed by the variable MEM in the next three locations. Address 1312 holds 212, the token for the equality operator, i.e. the equals sign '=' when stored as the arithmetic function. The Oric converts the character into the operator for insertion into memory, and then converts back to the character

form when a program is LISTED, just as it does with BASIC keywords. The tokens for the arithmetic operators are:

=	212	Equality
+	204	Addition
-	205	Subtraction
*	206	Multiplication
/	207	Division
↑	208	Exponentiation

Addresses 1313 to 1316 store #500, followed by the zero signifying the end of the line.

We'll leave it to you to work through the rest of the listing. If you want to find out the keyword corresponding to a particular value, you can use POKE:

```
10 REM
20 INPUT"ENTER TOKEN VALUE";N$.
30 POKE 1285,N$.
40 LIST
```

Address 1285, which is occupied by the token for REM, is POKED with the INPUT value. When line 40 LISTS the program, the Oric will convert the POKED value into the PRINTED form. A further illustration of the value of POKE is the simplicity with which we can change the PRINT instructions of lines 40, 60 and 70 of the memory display program into LPRINT instructions. This was how the listing of memory contents above was produced. The extra lines 90-140 added to the program, as given below, will check through the memory locations storing the program, replacing each occurrence of a PRINT token (value 186) with a LPRINT token (143). The REPEAT . . . UNTIL loop terminates when the next two locations are both zero, which is the marker for the end of the BASIC program, using DEEK.

```
90 REM CHANGE PRINT TO LPRINT
100 M=#500:C=0
110 REPEAT
120 M=M+1
130 IF PEEK(M)=186 THEN POKE M,143
140 UNTIL DEEK(M+1)=0
```

As another example of DEEK and POKE, here's a simple renumbering program, which will renumber lines in given steps, starting at a specified line. It does not renumber the destination line numbers of GOTO and GOSUB destinations. This is not too difficult to do, but requires more than one pass through the memory area.

```

10 REM**RENUMBER**
11 'Program starts line 60000.
12 'Does NOT renumber GOTO or GOSUB
13 'Destinations. Note these BEFORE
14 'Renumbering a program
60000 MEM=#501 'First line pointer
60010 INPUT"What existing line to start
renumber";BEGIN
60020 INPUT"Change this to line number"
;NLIN
60030 INPUT"Increment lines in steps of
";INC
60040 REPEAT
60050 LINE=DEEK(MEM+2):'Current line nu
mber
60060 IF LINE <BEGIN THEN 60080
60070 DOKE(MEM+2),NLIN:NLIN=NLIN+INC 'I
nsert new line number and increment
60080 MEM=DEEK(MEM) 'Get start next lin
e
60090 UNTIL DEEK(MEM+2)=60000'Do not re
number RENUMBER routine

```

Above the BASIC program are stored the variables used within the program and their values. This is the area which is reset when CLEAR is used. The variable names are stored in two bytes (the two characters of a variable name that the Oric recognises), with the type of variable identified by the modification of the ASCII code of the characters of the name. Real numeric variables are stored as unmodified character codes, integer variables have 128 added to each character code, and string variables have 128 added to the second character only. Single character names have zero (+128 if string or integer) stored in the second byte. The order in which simple variables are used in a program is determined by the sequence in which they are assigned in the program, and arrays are stored after the simple variables in the order in which they are DIMENSIONED.

Following the name of numeric variables there are five bytes which, in the case of real numeric variables, hold the number in a format where the first byte holds an exponent, and the remaining four the mantissa. The format is known as five-byte floating point format, which functions in binary much as exponential notation does in decimal. Integer variables are

stored in the first two bytes of the five, in standard two-byte form. The other bytes hold 0. String variables are stored with five bytes following the variable name. The first byte holds the length of the string, and the second and third are a pointer to the address in memory where the string of characters are held. This will be an address within the BASIC program listing if the string variable holds a literal string defined within the program. If it is a calculated string, or is redefined in the course of a program, the resultant string is stored at the top of the variables area, immediately below the character set areas, and the pointer will indicate this location. The length value stored in the first byte after the string name indicates how many bytes are to be read, starting at the address given by the pointer bytes. Two unused bytes follow the pointer.

Arrays have the same types of names as simple variables, but these are followed by a byte specifying the total number of bytes required to store the array data (including the array name). Two bytes then hold the number of dimensions in the array, followed by two bytes for each of the dimensions, specifying the number of elements for each dimension. Numeric arrays have five bytes for each element, stored in sequence, whereas integer arrays have only two bytes in each element. Each string element in an array has a length byte, and a two-byte pointer. You can access the start of the variables area and display the contents to see how the data is stored with a program similar to the one we used to probe the BASIC program storage. Set some variables, find the start of the variables area with DEEK(#9C), and you can start PEEKing. The two bytes at locations #9C and #9D are a system variable holding the address of the start of variable storage. Similarly, DEEK(#9E) will provide the end address of the variables area, and DEEK(#A2) will give you the address of the bottom of the string variables area.

That concludes our trip along the byways of the Oric memory. If you enjoyed the jaunt, you can POKE around some more with the aid of Appendix 9. Our next chapter is concerned with putting the contents of memory on to tape for safe storage, and freeing you from a continually plugged-in Oric.

# 6 Tapes and Data

As soon as you switch off your Oric's power supply any program held in memory is lost. This is because the RAM memory which stores the current program and variables is 'volatile' – i.e. when the machine is turned off, the RAM memory and CPU registers are cleared, ready for a fresh start when you next power up your Oric.

It is obviously impractical to type in a program each time you want to use it (a fifty line program will take you the best part of an hour), and thus some method of 'off-line' storage is clearly required. The cheapest and most widely used means of storing a program is on cassette tape. Now, whilst cassette storage is neither the fastest nor the most reliable medium for preserving your programs, it is considerably less expensive than disc-based systems, and your Oric ATMOS has a number of features which make cassette handling more reliable and flexible than usual.

To save Oric programs on cassette you will require both a cassette recorder and an appropriate lead to connect it to your computer. The choice of recorder is important, but this isn't to suggest that it should cost you a great deal of money. In fact it is preferable to use a cheap mono recorder (if you wish to use an existing stereo machine, make sure that you only use a single channel), and the dedicated data recorders available for around £35 from the high street chainstores are ideal for the task. Your life will be made considerably easier if you choose a machine with a tape-counter, because you can waste a great deal of time searching around for programs if you have no means of establishing exactly where they are on a tape.

Try to stick with the same recorder as far as possible, since you may run into problems when you try to load programs that were saved on another machine. (Slight differences in the alignment of playback heads can make a well-recorded cassette unusable when it is replayed on a different machine.)

The type of lead you require depends on the kind of cassette recorder you decide to use. The Oric itself needs a 3 or 7-pin DIN plug which fits into the cassette socket at the back of the machine. A 3-pin DIN to 3-pin DIN lead is supplied. The majority of data recorders use a single DIN socket which serves for both recording and playback, and the plug you'll need under these circumstances is exactly the same as for the Oric end of the connection. However, some mono cassette players have only an EAR and MIC socket, which makes matters slightly more complicated. In this case you'll

need separate jack-plugs for input and output (i.e. for playback and record), and it is wise to mark the individual plugs clearly so that you don't get them mixed up. The various types of lead are usually available from computer stores, but if you run into difficulties your local hi-fi shop can often be persuaded to make up a lead for you.

It is far better to use short-length computer tapes (C10s or C15s) from a reputable manufacturer than full-length C-90s or C-60s. Apart from the fact that it takes too long to locate a program on a C-90 (tape-counter or not), the longer (and thinner) tapes tend to stretch much more easily than short computer tapes, and are thus more likely to corrupt your recording.

The Oric has two main commands related to cassette handling: CSAVE and CLOAD. We'll first take a look at the formats and facilities that are available using these commands, before going on to discuss the other available tape file facilities of your ATMOS.

Once you have loaded your new cassette into your recorder, it's worth running it fast forward to the end, and then rewinding, to ensure that the tape is evenly and tightly wound. Most cassettes have a short plastic header that you can't record on. Make sure before you start recording your program that you have wound the tape past this header. In fact it is worth letting the tape run on for 15 seconds or so, since the majority of corrupted recordings result from damage to the relatively exposed beginning of the cassette, or stretching in the initial few inches of tape. Set your tape counter to zero and you're ready to go.

To save a program on to tape, follow the sequence below:

- 1 Check that the cassette player is plugged in.
- 2 Ensure that it is correctly connected to your Oric. The Oric end of the connection must be plugged into the cassette port DIN socket, and the other end should be plugged into the recorder's input/output socket if you are using a recorder with a DIN socket or the MIC jack-plug should be in the MIC socket if you are using a lead fitted with jack-plugs. (In the latter case, the EAR plug should be left unconnected to prevent possible feedback loops.)
- 3 Check that you have a fresh cassette in the recorder and that you have wound the tape well past the plastic header.
- 4 LIST the program that you want to CSAVE on the screen.
- 5 Key-in as a direct command:

CSAVE“FILENAME”,S

where “FILENAME” is the name of the program you wish to save. The name can be up to 16 characters long, but it is best to keep the name as short, relevant and easy to remember as possible. Any characters may be used.

You'll notice that in the above format the filename in double quotes is followed by a comma and an S. The S stands for Slow, and is an optional part of the format. The Oric CAVES and CLOADS at two different data transfer

rates. Without the ,S the Oric automatically assumes 2400 baud Fast mode, but if you add ,S the Slow mode is enabled (300 baud). A baud is a transfer measurement of bits per second of data. Whilst both formats are reliable, you should always take the precaution of making at least one copy of any valuable program in the extra safe Slow mode. Let's continue with our CSAVEing sequence:

- 6 Press the RECORD and PLAY buttons on your cassette recorder.
- 7 Press RETURN on the Oric. The message SAVING FILENAME will appear at the top of the screen, followed by the file type B for BASIC program. When the program has been CSAVED, the usual READY message will appear at the current cursor position. Stop the recorder.

If you have carefully followed the above steps, you should now have a copy of your program on cassette. In order to be certain that the program has CSAVED correctly without having to actually reload it into the Oric, you have the capacity to VERIFY a recording to ensure that the program has been correctly CSAVED, but more about that later. For now, let's deal with loading.

When you want to load from tape any program you have CSAVED, the following sequence should be followed:

- 1 Check that the cassette recorder is plugged in, that its volume is set at around the halfway point, and that the tone control is set to high.
- 2 Ensure that the recorder is correctly connected to the Oric. The computer's DIN plug connection will be the same as it was when you CSAVED the program. If you are using the supplied lead or a similar DIN-to-DIN lead, it will be connected to the recorders' input/output socket. If you are using a DIN-plug-to-jack-plug lead, the EAR jack-plug should be placed in the EAR socket and the MIC jack disconnected.
- 3 Using your tape counter, locate the start position of the program you wish to CLOAD.
- 4 Key in:

CLOAD“FILENAME”,S

The filename must be precisely correct and must include any spaces that were in the filename when the program was CSAVED, or the Oric will not recognise the program as the one you are trying to load. If you have forgotten, or are unsure of the precise filename, you can use the format:

CLOAD“,S

and the Oric will CLOAD the first program it finds on the tape. Once again the ,S has been included in the instruction format, but must only be used if the program was actually CSAVED in the Slow mode.

- 5 Press the RETURN key on your Oric.

6 Press the PLAY button on your cassette recorder. The Oric will search through the cassette until it finds the program "filename" and while it does so the message SEARCHING... appears at the top of the screen. When the program has been found the message will change to LOADING FILENAME (followed by the file type specifier B) until the CLOADING process is complete and the READY message appears. If there is a fault on the tape you may well get an ERRORS FOUND message appearing on the screen after the load finishes. If this happens don't immediately despair. It could be that the volume control on your recorder is set too high or too low, or that the tone is set at the wrong level. Before giving up hope, spend some time making small adjustments to the controls on your cassette player. If you can get the program to CLOAD after these adjustments it would be wise to make another copy of the program immediately, just to be on the safe side.

You have now learnt all that you need to know about the straightforward use of the CSAVE and CLOAD commands on the Oric. Now we'll take a look at the other cassette facilities available on the machine. If you want a program to RUN automatically as soon as it is loaded, you must add the following dimension to your CSAVE command:

CSAVE"FILENAME",AUTO,S

Once again, the **S** should only be included if you wish to CSAVE the program in the Slow mode. CSAVED in the above format, the program will RUN as soon as the CLOADING process has been satisfactorily completed. AUTO saved programs do not require any change in the CLOAD format.

As well as complete programs, it is also possible to store blocks of memory on cassette. This is particularly useful if you want to CSAVE a screen display. However, if you use this method to save screen displays, you must ensure that the Oric is in the correct mode for the display in question. To store any block of memory it is necessary to know the Address at which the block starts, and where it Ends. With this information to hand, the memory blocks can be CSAVED with the following format:

CSAVE"MEMBLOC",A#400,E#420,S

This would CSAVE (in Slow mode) the contents of RAM from locations #400 to #420. You can use this facility to CSAVE and CLOAD a memory area storing a different character set or a machine-code routine. Because the rest of the Oric's memory would be unaffected by this procedure, any BASIC program in memory would be protected from the additional routines. The areas of memory applicable to saving screens are as follows:

For the HIRES screen:

CSAVE"filename",A40960,E48000 (48K Oric)  
CSAVE"filename",A8192,E15232 (16K Oric)

For the TEXT and LORES screens:

CSAVE“filename”,A48000,E49119 (48K Oric)  
 CSAVE“filename”,A15232,E16351 (16K Oric)

The start Address and End address will accept either decimal or hexadeciml values, and again, ,S may be appended for a Slow saving rate. When CLOADED, blocks of memory are automatically loaded back to their original memory locations, so the normal CLOAD“FILENAME” is all that is required. The file type specifier is C (for code) when saving or loading memory blocks.

There are two other cassette facilities available. The most valuable of these additional commands is the VERIFY facility. When you have CSAVED a program and wish to see whether or not the process has been successful the following check can be made:

1 Rewind the cassette back to the beginning of the program you have just CSAVED.

2 Check that the volume control on your recorder is set at the correct level for CLOADING, and that the tone level is high.

3 Ensure that your lead connections are correct. The Oric's end of the lead should remain in the same socket, the other end should be in the cassette input/output socket if a DIN plug, or, if using jack-plugs, the EAR jack should be in the EAR socket (with the MIC jack unconnected).

4 Key-in the following as a direct command:

CLOAD“FILENAME”,V,S

(Once again the ,S should only be used if the program was CSAVED in the Slow mode.) The filename may be omitted if you're certain of the program's position. The computer will begin SEARCHING... in the usual way, and when it locates the program it will report VERIFYING FILENAME (plus the file type identifier). If the recording has been successful, the message 0 Verify errors detected will appear at the current cursor position. If the recording has been unsuccessful (any number other than 0 for errors) you must return to the operations outlined in step 6 of the CLOAD procedure given above, if a second try doesn't work.

The other additional CLOAD command available on the Oric is the Join facility. This enables you to Join a second program on to the end of a program already typed or CLOADED into your Oric. The format for this operation is as follows:

CLOAD“FILENAME”,J,S

All that this feature actually does is to prevent the Oric from clearing the memory as it normally does when any CLOAD format is used, and insert the new program lines sequentially into memory. If you wish to Join a second

program in this way, you must ensure that all the line numbers in the second program are higher than the highest line number in the first. If this is not the case the final product will fail to RUN, since the line numbers in the second program do not 'interlace' with those in the first, and the second program is merely inserted (above the existing program) into memory. The joined programs must fit within the available memory capacity.

There is an additional tape file capability available. STORE and RECALL allow arrays to be saved onto tape and read back into a program, thus providing a means of passing data from one program to another. Within a program, the contents of an array can be defined, and then placed as a cassette file onto tape. Any type of array may be STORED. The format for STORE is:

STORE V, "FILENAME", S

where v is the name of the array to be stored (A\$, for example, for an array A\$(3,4), G for an array G(30), etc.).

The procedure for using the instruction is the same as that for CSAVE, and the fast save (2400 baud) will be used as default if the ,S is omitted. The message SAVING FILENAME appears on the status line, followed by a letter specifying the type of array: R for Real floating point number arrays. I for an Integer array, and S for string arrays.

RECALL will load back from tape the contents of an array previously saved on tape using STORE and the procedure is the same as when using CLOAD. The array to store the RECALLED array must have been dimensioned prior to using RECALL, or an OUT OF DATA error occurs. The array size must be the same as (or greater than) the original array and of the same type (integer, string or real). RECALL may be used within a program, or as a direct command, but if the latter, no instructions which reset the variables stored can be used (CLEAR, RUN, etc), or the stored array values will be lost.

The format for RECALL is:

RECALL V, "FILENAME", S

where v is the name of the array in which the RECALLED data is to be placed. This does not have to be the same name as that of the original array which was STORED, but must be of the same type. Try the example programs below to illustrate this. Note that whilst (in the RECALL example) the array B was DIMENSIONED as B(20), the same size as the array A(20) used in STOREing, it could be DIMENSIONED larger. Try changing line 10 in the program to read DIM B(25) and RECALL the array again.

Slow (as in the format above, using ,S) or fast (omitting ,S) speed data transmission from tape can be specified. The same speed must also have been used by the STORE instruction.

```

10 DIM A(20)
20 FOR K=0 TO 20
30 LET A(K)=2^K
40 NEXT K
50 PRINT"PRESS A KEY WHEN READY TO STOR
E"
60 GET A$
70 STORE A,"ARRAYFILE"
80 PRINT"ARRAY NOW STORED"
90 END

10 DIM B(20)
20 PRINT "SET RECORDER TO PLAY,PRESS A
KEY TO":PRINT "RECALL ARRAY DATA"
30 GET A$
40 RECALL B,"ARRAYFILE"
50 CLS
60 FOR L=0 TO 20
70 PRINT B(L)
80 NEXT L
90 END

```

### **Care of cassettes**

If you've taken the trouble to develop a program and CSAVE it on cassette, it's worth giving some thought to ensuring that your copy of the program lasts for as long as possible. It should be stressed that if a tape is left unused for as little as a month there is a danger that when you come to use it you will find the copy has been corrupted. However, there are steps which can be taken to minimise the risk:

Do not record on the first 10–15 seconds of a cassette. Most of the problems of stretching and coating loss occur in this section of the cassette.

When a cassette is not in use always ensure that it is returned to its case.

Never leave a cassette on top of a TV set or any other electrical appliance. The electromagnetic fields generated by such equipment may corrupt the signals stored on the tape.

Never touch the tape surface, and always make a point of fully rewinding the cassette after use so that only the plastic header is exposed.

New tapes should always be run through and then rewound back to the beginning before any recording is attempted. This will ensure an even tension.

Make sure that you clean the record and playback heads of your cassette recorder on a regular basis. This should really be done after every two or three hours' running time. Invest in a head demagnetiser, and demagnetise the heads of your recorder after every twelve hours' or so running time.

Once you have recorded a program, make sure that the program filename and its tape-counter reading is accurately documented on the cassette and the cassette case. Ensure that you have noted which mode the cassette was recorded in (i.e. Slow or Fast), and if the program is still under development, make it clear which version of the program has been recorded.

For the final version of any program, the cassette tabs at the top of the tape should be removed to prevent accidental erasure.

Make back-up copies of any important programs.

Rewind tapes at intervals, even if you don't CLOAD any programs from them. This helps prevent 'bleed through' of magnetisation from one section of tape to adjacently spooled sections.

#### N.B.

Certain cassette recorders may cause the Oric to give a spurious ERRORS FOUND message after CLOADING. Variation in the tape lead-in signal, due to the 'hunting' of automatic level controls when recording, can cause the data checks built in to the Oric to record an error. Despite the error message, programs will load correctly, but AUTO-run will, as usual, be inhibited. The machine code chapter contains a program which will solve this problem, should you encounter it with your recorder.

# 7 Graphics and colour

Right, by now we are sufficiently familiar with the simpler BASIC statements to look at one of the more advanced topics on microcomputers: GRAPHICS. Your Oric has particularly good colour and graphics capabilities for a micro-computer. In all there are four separate screen displays or *modes* on the Oric. These modes are TEXT, LORES 0, LORES 1 and HIRES. In this chapter we will examine each of these in turn.

## TEXT mode

This is the mode your Oric goes into when you first turn it on and, as the name implies, it is primarily intended for the display of text. How do we display text? Well, we have already met the PRINT statement that PRINTS characters or variables on to the screen but it's worthwhile taking a closer look at the details of PRINTing. The simplest form of PRINT is shown in the example below and if you type this in and then RUN it you will get the following PRINTed on your screen:

```
10 PRINT "FRED"
```

FRED

Now add another line to get the following program which, when RUN, will give the output shown below.

```
10 PRINT "FRED"  
20 PRINT "A"
```

FRED

A

It seems quite reasonable that the computer should use a new line to PRINT the "A" and in general the computer will go immediately to the start of the next line after performing a PRINT instruction. However, there are some special forms of PRINT that we can use to alter this behaviour. Add a comma

to the end of line 10 and then RUN it. This should look like the listing below and result in output as shown.

```
10 PRINT "FRED",
20 PRINT "A"
```

FRED A

Obviously the comma has affected the screen position of the information in the following line which has now moved to the end of the PRINT statement in line 10. The screen can be thought of as having five formatting fields, each eight character positions wide, across each line. If a comma is used to separate two items which are to be PRINTed, or as the final item in a PRINT statement, then the PRINT position is moved forward to the start of the next format field. This can be very convenient when we wish to PRINT out tables of figures.

There is another character which we can use as a separator in PRINT statements and this is the semi-colon, ';'. Alter line 10 again and RUN the program. This should result in the listing and output shown below.

```
10 PRINT "FRED";
20 PRINT "A"
```

FREDA

Well we seem to have altered FRED's sex this time! The semi-colon causes the PRINT position to be left wherever it was when PRINTing stopped. The example below shows the output produced by two sample PRINT statements. Note that every time the Oric has to PRINT out a number it PRINTs a space before and after the number as well.

```
10 PRINT 1,2;3
20 PRINT "A","B";"C"
```

1	2	3
A	BC	

There is another command we can use to affect the position of the PRINTing, this is done by using the TAB command. This is similar to the TAB function on a typewriter. By saying PRINT TAB(n); we can move the PRINT position to the n'th character position along the line. Try the program below which should give the output shown after the listing.

```
10 FOR I=0 TO 5
20 PRINT TAB(I); "HELLO"
30 NEXT I
```

HELLO  
HELLO  
HELLO  
HELLO  
HELLO  
HELLO

Looking at the output it seems that TAB(0), TAB(1) and TAB(2) had no discernable effect, but this is because character position 0 and character position 1 are reserved by the Oric for special codes (we'll come to these in a moment). So the PRINT position is already set to 2 when we start on a new line and, as on a typewriter, you cannot TAB backwards. If we wish to use the first two character positions, 0 and 1, we can press the CONTROL and ] keys together. This is one of the combinations the Oric recognises as a toggle which reverses the state of an internal switch (in this case the column protection switch), and since it was on it will now be turned off. If we RUN the program again we should get the output shown below.

HELLO  
HELLO  
HELLO  
HELLO  
HELLO  
HELLO

Now we come to the question, what were the first two columns reserved for in the first place? Well, if you tried the above example you've probably guessed by now. These columns affect the colour of text on the screen. When you use these columns the PRINT appears as white on black instead of the usual black on white which is like ink on a piece of paper. This is exactly what these columns are used for. The first column usually holds the PAPER colour and the second column holds the INK colour. Toggle the protection switch again with CTRL and ] so that the first two columns are protected again. Now we can try changing the INK and PAPER colours on the screen. The Oric associates each number between 0 and 7 with a particular colour and the command PAPER n or INK n will set the background or foreground colours appropriately, provided n is in the range 0-7. The colour for each number is shown in the table below.

- 0 BLACK
- 1 RED
- 2 GREEN
- 3 YELLOW
- 4 BLUE
- 5 MAGENTA
- 6 CYAN
- 7 WHITE

Try typing in PAPER 1 and press RETURN. The screen should instantly change to a display of black characters on a red piece of PAPER. Try the INK command in the same way but remember that if the INK colour is the same as the PAPER colour you won't be able to see the characters.

On the Oric all colours are controlled by special characters known as attributes. These are sometimes referred to as 'serial' attributes because they affect everything that follows on from them. So what the Oric does each time we alter a colour is to go down one of the protected columns putting a colour attribute in that position on each line, which consequently affects the rest of the line. Type in the following program and LIST it on the screen. Now RUN the program and it will show all the combinations of INK and PAPER colours available. Once again, remember that when the INK and PAPER are the same the TEXT will seem to disappear.

```

10 FOR I=0 TO 7
20 INK I
30 FOR J=0 TO 7
40 PAPER J
50 WAIT 50
60 NEXT J
70 NEXT I
80 INK 0

```

Returning to PRINTING, there is one more item we can use to affect the position at which we PRINT. This is by using the @ ('at') symbol with two numbers. The first number is like the column number in the TAB function, the second number specifies which row of the screen we are going to move to. The column position can be anywhere from 0 to 39 and the row position can be anywhere from 0 to 26. As a quick example RUN the following program (or enter it as a direct command).

```
10 PRINT @10,10;"HELLO"
```

As expected, this places a "HELLO" quarter of the way across and about a third of the way down the screen. Not very exciting you might think, but this ability to PRINT at any screen location is the basis of most games! The following example should give you some idea of why this is so.

```
10 FOR X=2 TO 30
20 PRINT @X,10;" WHIZZ!"
30 NEXT X
```

This sort of animation is possible in all directions around the screen, but we do not want to see the flashing cursor while we are PRINTing characters on the screen. To prevent this we use one of the other toggles, CTRL Q. This switches the cursor on and off and to do this in our program we used the CHR\$ function which produces characters from their code numbers. We ought to remember to turn it back on at the end of our program as well so here's a modified version of the previous listing which does this:

```
10 PRINT CHR$(17)
20 FOR X=2 TO 30
30 PRINT @X,10;" WHIZZ!"
40 NEXT X
50 PRINT CHR$(17)
```

The next program is a game using all the things we have met so far. You are a duck hunter and you must shoot your arrows at the ducks ">" that fly across the top of the screen. This program has been indented to show its structure, but the leading colons and spaces make no difference to how it RUNS.

```
10 : CLS : S=0 : PX=20
20 : PAPER 6 : PRINT CHR$(17);CHR$(6)
30 : REPEAT
40 :   PRINT @10,0;"SCORE";S
50 :   WAIT RND(1)*100
60 :   DX=2 : REPEAT
70 :     IF KEY$=CHR$(9) AND PX<37 TH
EN PX=PX+1
80 :     IF KEY$=CHR$(8) AND PX>2 THE
N PX=PX-1
90 :     PRINT @ PX,25;" ^ ";
100 :    DX=DX+1 : PRINT @ DX,3;" >"
```

```

110 :      IF KEY$="S" THEN 230 ELSE IF
KEY$<>" " THEN 200
120 :      MX=PX+1 : MY=24
130 :      REPEAT
140 :          PRINT @MX,MY;""
150 :          PRINT @MX,MY+1;" ";
160 :          MY=MY-1
170 :      UNTIL MY=3
180 :      IF MX=DX+1 THEN S=S+1 : DX=3
8
190 :      PRINT @MX,MY+1;" " : PRINT @
MX,MY;" "
200 :      UNTIL DX=38
210 :      PRINT @39,3;" ";
220 :UNTIL KEY$="S"
230 :PRINT CHR$(17);CHR$(6)

```

By now you might be wanting to know how you can have more than two colours on the screen at once, to make the ducks a different colour from the hunter, for example. This is where we come back to those serial attributes which affect the rest of the line. We can produce the attributes by typing ESCAPE followed by @,A,B,C,D,E,F or G. These give the attribute for INK 0-7 respectively. Similarly ESC P to ESC W give the attributes for PAPER. It is important to realise that these attributes are actually put in a character position on the screen and affect the rest of the line following their position. To do this from inside a program we must first PRINT the character which means ESCape (CHR\$(27)) and then the character for the letter we wish to follow the ESCape code. This can either be PRINTed directly as "Q", or using the CHR\$ function. CHR\$(64+n) gives the attribute for INK n and CHR\$(80+n) gives the attribute for PAPER n. The following program will demonstrate this for PAPER colours and you can try changing the 80 in the CHR\$ function to a 64 to see the same thing with foreground or INK colours.

```

10 FOR X=0 TO 7
20 PRINT CHR$(27);CHR$(80+X);"HELLO A
GAIN"
30 NEXT X

```

There are other special attributes which can be put on the screen in this way and these produce various effects on the rest of the line. The table below shows the characters which can follow ESCape and the attributes or effects they produce.

## ESCAPE SEQUENCES AND ATTRIBUTES

ESCAPE @	0	Black ink
ESCAPE A	1	Red ink
ESCAPE B	2	Green ink
ESCAPE C	3	Yellow ink
ESCAPE D	4	Blue ink
ESCAPE E	5	Magenta ink
ESCAPE F	6	Cyan ink
ESCAPE G	7	White ink
ESCAPE H	8	Standard text
ESCAPE I	9	Alternate text
ESCAPE J	10	Standard Double height
ESCAPE K	11	Alternate Double height
ESCAPE L	12	Standard Flashing
ESCAPE M	13	Alternate Flashing
ESCAPE N	14	Standard Double height Flashing
ESCAPE O	15	Alternate Double height Flashing
ESCAPE P	16	Black paper
ESCAPE Q	17	Red paper
ESCAPE R	18	Green paper
ESCAPE S	19	Yellow paper
ESCAPE T	20	Blue paper
ESCAPE U	21	Magenta paper
ESCAPE V	22	Cyan paper
ESCAPE W	23	White paper

The following listing contains many new features so we will go through it line by line. The first line PRINTS the character with ASCII code 12 to the screen. This is the clear screen or form feed character, so the computer takes a new sheet of paper – just a different way of doing CLS. The second line PRINTS another toggle, this one CTRL D affects double PRINTING. When this is on, anything PRINTed to the screen is repeated on the row below. The third line produces the attribute for double height and flashing. Line 40 PRINTs the message on the screen where it is PRINTed twice. Line 50 toggles the double PRINTing off again.

```

10 PRINT CHR$(12)
20 PRINT CHR$(4);
30 PRINT CHR$(27); "N";
40 PRINT "A BIG FLASHING HELLO"
50 PRINT CHR$(4)

```

Try taking out the lines toggling double printing or the line producing the attribute. The effects produced when you make these modifications should

help you understand the role each plays producing the message. Note that we have arranged to have the top line of our double height characters on line 1 and the bottom half on an even line (line 2). This is because the computer must assume that if you are using double height all the top halves of characters are on odd lines with their matching bottom half hopefully on the even line below. Try adding a line like this to start PRINTING on the wrong line and see what happens when we get it wrong.

## 15 PRINT

Here is a short program which demonstrates how useful this arrangement for finding the tops and bottoms of lines can be.

```

10 CLS
20 REPEAT
30 FOR X=1 TO 7
40 PRINT CHR$(27) ;"J" ;CHR$(27) ;CHR$(8
0+X) ;
50 PRINT"      ORIC HANDBOOK BY PAN BOO
KS"
60 NEXT
70 UNTIL KEY$<> ""

```

## LORES 0

In this mode the Oric displays a black screen on to which the standard characters can be PLOTTed in a similar way to that in which they were PRINTed at a position on the text screen. In fact the LORES 0 screen is just the same as the text screen but the attribute at the start of the screen is now just the one which selects the standard character set. The colours white on black are the default colours which are used if no attributes for these are found

The PLOT command has two co-ordinates, as for PRINT@. The first, the X co-ordinate, can be from 0-39, whilst the second, the Y co-ordinate can vary from 0-26. Using PLOT does not affect the PRINT position. The PLOT command is not as flexible as the PRINT@ command since it requires a string of characters. Thus if we wished to PLOT a number we would have to use a command like PLOT 10,10,STR\$(1).

Using PLOT we can also put individual characters on the screen and in this respect it is a very different statement to the PRINT@ command. We can use PLOT to put characters on the screen in inverse. To get inverse characters we must use the ASCII code of the character plus 128 and this cannot be done from PRINT. When a character is PRINTed in inverse the colours used are the

logical inverse colours. This does not mean that the character will appear as PAPER on INK colours but means that the character will appear as the logical inverse of INK on the logical inverse of PAPER. The logical inverse of a colour is found by taking the number of the colour away from seven. This gives us the number of the colour which will appear. Thus if the PAPER was 1 it would be PRINTed as though PAPER were (7-1) or 6 and similarly for the INK.

The following listing creates the string of inverse characters from the normal string and then uses PLOT to place them on the screen. Just to prove that PRINT doesn't allow inverse characters they are also PRINTed at the top of the screen.

```

10 CLS
20 PAPER 1 : INK 7
30 A$="HELLO"
40 FOR I=1 TO LEN(A$)
50 B$=B$+CHR$(ASC(MID$(A$, I))+128)
60 NEXT
70 PLOT 10,10,A$
80 PLOT 10,11,B$
90 PRINT A$, B$
```

An associated command is the SCRn function which returns the value stored at the SCREn location specified by X and Y (as for PLOT). This value is usually the ASCII value of the character displayed at that position, but it could also return the value of the attribute stored there if there is nothing actually displayed. The following listing demonstrates both of these ways of using SCRn:

```

10 LORES 0
20 PLOT 10,10,"A"
30 PRINT SCRn(10,10)
40 PRINT SCRn(0,0)
```

The 8 that's produced by the second use of SCRn can be explained by looking at the table of ESCape sequences and attributes we gave earlier on. This figure simply represents the attribute for the standard set. In fact it doesn't matter whether you overwrite this or not because this is, of course, the default setting anyway. The reason for these codes being put here will become apparent in the next section on the LORES 1 mode. As the above use of SCRn implies the screen is actually represented in memory by a set of 1120 memory locations (this is  $28 \times 40$ , since there are 28 lines on the screen including the top status line). These memory locations stretch from #BB80 or decimal 48000 to #BFDF or 49119 and we can address these locations

individually by using POKE. Try the following program which fills all these locations with the ASCII code of "A".

```
10 FOR I=#BB80TO#BFE0
20 POKE I,65
30 NEXT I
```

### **LORES 1**

In much the same way that LORES 0 was just TEXT mode with all the first column initialised to 8, LORES 1 initialises the first column to 9, (the attribute for the alternative character set). Thus we can print a variety of alternative characters and they will appear as combinations of 6 little squares inside one character block. Try plotting the whole alphabet on the screen. You will see that there seems to be a pattern to the arrangement of blocks. In fact there is a very logical arrangement to their patterns. To understand how these blocks are set up we must first get right down to basics.

Inside the computer all the numbers are stored in binary. For each location we have eight binary digits, each of which can be 1 or 0. In the screen locations these bits can each have a special meaning. As we have seen, the bits are worth 128, 64, 32, 16, 8, 4, 2 and 1. From the above it is plain that the highest bit (128 or b7), indicates that inverse video is to be used for that particular character's display, but you should also note that this does not affect the rest of the line! The next two bits down, b6 and b5, (worth 64 and 32 respectively), can have a special significance when considered together. If they are both zero then the location is an attribute and the remaining bits will add up to a number between 0 and 31 that can be looked up in the attribute and ESCAPE sequence table above to see what effect it has. We could also have the highest bit set making the number larger than 128, but bits 5 and 6 being zero will still indicate that it is really an attribute. Try the following program.

```
10 LORES 1
20 A$=CHR$(145)+"ABCDEFGHIJKLMNPQRSTUVWXYZ
UVWXYZ"
30 PLOT 5,10,A$
```

145 is the attribute for red PAPER with 128 added on to indicate inverse. This means that the space where the attribute lies is shown in the inverse of red, i.e. cyan, and the rest of the line is PRINTED in red. You will have noticed by now that the red does not go right to the end of the line, but reaches only as far as the last printed alternate character! Well, a quick check with SCRn should reveal the situation, in that all the other locations have been set to 16, which is the attribute for black PAPER. Now let's look at how the bits that choose the character are arranged. Remember that we said it would be an

attribute only if both b5 and b6 were 0. For the normal character set all the bits from b6 down to b0 are added up to get the code of the character to be displayed. This also happens in the alternate set, but the way that the block characters have been defined is designed to allow each block of the six to be represented by one bit. There are seven bits in the code but we must use one to keep the code from being an attribute. B5 has been chosen to always be one and the other bits are allocated as follows: The top left block is b0, top right is b1, middle left is b2, middle right is b3, bottom left is b4 and bottom right is b6. To get a character with the right blocks set, we just add up the values for the bits representing each block, then add in 32 for bit 5. In this way we can address the Oric graphics as MEDIUM RESOLUTION graphics. We can think of the screen as being 0 to 79 across and 0 to 80 blocks down. The following program shows a subroutine, which will plot any individual block on this coordinate system, in use. We must of course avoid plotting to x co-ordinates less than two since these would lie within the first character position and erase the alternate set attribute.

```

10 DIM P(2,3)
20 FOR I=1 TO 3
30 READ P(1,I),P(2,I)
40 NEXT I
50 DATA 1,2,4,8,16,64
100 LORES 1
110 Y=40
120 FOR X=2 TO 79
130 GOSUB 1000
140 NEXT X
150 FOR X=2 TO 79 STEP 0.2
160 Y=40+SIN(X/10)*35
170 GOSUB 1000
180 NEXT X
190 END
1000 REM PLOT X,Y : MEDIUM RESOLUTION
1010 CX=INT(X/2) : PX=INT(X)-CX*2+1
1020 CY=INT(Y/3) : PY=INT(Y)-CY*3+1
1030 P=#BB80+(CY+1)*40+CX
1040 Q=PEEK(P) : IF Q<32 THEN Q=32
1050 Q=Q OR P(PX,PY) : IF Q>95 THEN Q
=Q-32
1060 POKE P,Q
1070 RETURN

```

## HIRES

This is the Oric's HIGH resolution graphics mode. We can place points anywhere on the screen to a resolution of 240 dots across and 200 dots down. As with all the other modes the top left corner is  $0,0$ . There are several complex and powerful inbuilt commands for HIRES graphics on your Oric. We will start by taking a look at the two simplest commands.

The first of these is CURSET. This is the command which we use to set the start position for our graphics cursor. This command has three parameters. The first two of these are simply the x and y co-ordinates of the point in question. The third parameter is one which is used by nearly all the HIRES graphics commands, so we'll deal with it in detail here and refer back later on. This final parameter is known as the foreground/background or fb parameter. This affects the way points are plotted on the screen. The fb parameter may have any of the four values 0 to 3. The values and the way they affect the plotting commands are summarised in the table below.

### fb Effect

- 0 The points are plotted in the background colour. (This can also be thought of as un-plotting a point).
- 1 The points are plotted in the foreground colour.
- 2 The points are inverted i.e. if they were in the foreground colour before being plotted then they are changed to the background colour and vice versa.
- 3 Null. The points are not affected although the graphics cursor position will be set to the last point visited.

The next graphics command to consider is DRAW. This has three parameters like CURSET and these are the x and y parameters followed by the fb parameter. However, DRAW works in a very different way to CURSET. The x and y values specified are not the co-ordinates of a point on the screen but are relative to the current graphics position. This means that they are added to the current position to get the co-ordinates of the point to which the line will be drawn. We must be careful to ensure that this point will not be outside the screen area or the Oric will have to stop and tell us there's an error in our program. The program below demonstrates the difference between CURSET and DRAW by first plotting the points in the corners of the screen and then, after a key press, DRAWing the lines joining them up. Note that we use CURSET to set the start position before DRAWing the lines.

```
10 PAPER 0:INK 7
20 HIRES
30 CURSET 0,0,1
40 CURSET 0,199,1
```

```

50 CURSET 239,199,1
60 CURSET 239,0,1
70 GET A$
80 CURSET 0,0,3
90 DRAW 0,199,1
100 DRAW 239,0,1
110 DRAW 0,-199,1
120 DRAW -239,0,1

```

There is a third simple, but less often used, graphics command and this is CURMOV. This is the "relative movement" version of CURSET. The x and y parameters are added to (or subtracted from) the current graphics position as they are in DRAW. In fact this command is almost the same as using DRAW with the fb parameter set to 3. Try the following two versions of a program to demonstrate the different workings of CURSET and CURMOV.

```

10 PAPER 0:INK 7
20 HIRES
30 FOR I=1 TO 15
40 CURSET I,I,1
50 NEXT I

```

```

10 PAPER 0:INK 7
20 HIRES
30 FOR I=1 TO 15
40 CURMOV I,I,1
50 NEXT I

```

To demonstrate the high resolution and accuracy of plotting achievable on your Oric's screen try the following program. This should result in a TV picture which has rippling colours where the lines are close together. This is because your Oric is designed to exploit the display capability of your domestic TV to its limits and unless you have a very expensive monitor/TV this is as fine a resolution as it can handle.

```

10 PAPER 0:INK 7
20 HIRES
30 FOR I=0 TO 239 STEP 2
40 CURSET I,0,1
50 DRAW 239-I,199,1

```

```

60 CURSET I,199,1
70 DRAW -I,-199,1
80 NEXT I

```

A further feature the Oric has associated with the DRAWING of lines is the PATTERN facility which allows you to specify the type of line drawn, as dots, dashes, etc. The PATTERN command has one parameter between 0 and 255, and this defines, by its binary equivalent, a PATTERN which will be used when drawing any line of eight points in length. The default PATTERN is 255 which in binary is 11111111 so all the dots in a stretch of line are plotted. If we were to use PATTERN 85 which in binary is 01010101 we would get lines drawn with every other point left out. The program below draws a sample length of line in each of the available PATTERNS so you can see what's possible.

```

10 HIRES
20 FOR I=0 TO 31
30 FOR J=0 TO 7
40 PATTERN J*32+I
50 CURSET J*30,I*6,3
60 DRAW 24,0,1
70 NEXT J
80 NEXT I

```

The following program DRAWS a demonstration pattern but first allows you to select the PATTERN to be used for drawing the lines. This will allow you to experiment with the PATTERN command.

```

10 PAPER 0:INK 7
20 HIRES
30 INPUT "PATTERN ";P
40 PATTERN P
50 FOR I=0 TO 199 STEP 5
60 CURSET I,0,1
70 DRAW 199-I,I,1
80 CURSET I,199,1
90 DRAW -I,I-199,1
100 NEXT I

```

Now we come the advanced graphics commands available on your ATMOS. The first of these is CIRCLE and this has two parameters. The first parameter is the radius of the CIRCLE and the second is the fb parameter. The CIRCLE

will be plotted using the current graphics position as the centre. We must ensure, as with DRAW, that the CIRCLE does not go off the screen or we will cause an error. The program below demonstrates the CIRCLE command in action by picking a centre at random for the x and y co-ordinates. The minimum distance from this point to the edge of the screen is then calculated as this is the maximum radius we can use for a CIRCLE about this point. Note that 0 is not a valid value for the radius, so we don't try to plot a CIRCLE in this case. The central point of the circle has been set with CURSET using an fb parameter of 1 so that you can see where it lies.

```

10 PAPER 0:INK 7
20 HIRES : REPEAT
30 X=INT(RND(1)*239)
40 Y=INT(RND(1)*199)
50 IF Y>99 THEN DY=199-Y ELSE DY=Y
60 IF X>119 THEN DX=239-X ELSE DX=X
70 IF DX>DY THEN R=DY ELSE R=DX
80 CURSET X,Y,1
90 IF R>0 THEN CIRCLE R,1
100 UNTIL KEY$="S"

```

The next program shows a slightly more constructive use of CIRCLE with the centres not being plotted.

```

10 PAPER 0:INK 7
20 HIRES : REPEAT
30 FOR X=3 TO 50 STEP 3
40 CURSET X+50,100,3
50 CIRCLE X*2,1
60 NEXT

```

In much the same way that we read the contents of a particular screen location in the TEXT modes with the SCRn function, we can examine the status of any point on the screen with the POINT command. This has two parameters, logically, the x and y co-ordinates of the POINT to be examined. The value 1 is returned by the function if the POINT is displayed in the foreground colour and 0 is returned if it is displayed in the background colour. This is not the same as reading the contents of a whole location as we did with the SCRn command. This is because each of the 200 lines of the screen is, in fact, made up of just forty screen memory locations. The bits in these locations can indicate, just as on the TEXT screen, that an attribute is present at this location (bits 5 and 6 both 0). If this is not the case then the lower six bits represent the status of a row of 6 points along the line, where a

binary one corresponds to a foreground point and a binary 0 corresponds to a background point. This is how we get 240 or  $6 \times 40$  points along each row. Bit 7 is used to indicate that a particular block of six points is to be displayed in the logical inverse colours and in this way we can access four colours, even though we have only set one foreground colour and one background colour. We can set the attributes using PAPER and INK just as we did in TEXT mode and these commands will place an attribute in the first and second positions of each of the 200 rows of the high resolution screen. Of course this means we will not be able to plot points with x co-ordinates less than 12 because these would lie within the locations used for attributes. Now comes the question of how we access the inverse bit in the memory locations. Well, this is just one use of the FILL command for high resolution graphics. With this we can FILL a block of screen locations so many rows down, by so many bytes or locations across, with whatever number we choose. We must set the graphics cursor to the top line and left-hand corner of the block we wish to access and then issue the FILL command with the following three parameters. Parameter one is the number of rows down, number two is the number of bytes across and number three is the number which we wish to put into all of these locations. We could use this to FILL an area of the screen with a pattern of points, but in this case we will use it to set the highest bit of all the locations on the right-hand side of the screen. Note that we also set bit 6 so that the locations are not treated as attribute locations so 192 in line 40 is  $128 + 64$ .

```

10 HIRES
20 PAPER 2 : INK 4
30 CURSET 120,0,3
40 FILL 200,20,192
50 CURSET 120,100,3
60 CIRCLE 90,1

```

As we mentioned above the HIRES screen is made up of  $200 \times 40$  locations and we can access these directly by using POKE. These locations lie between #A000 and #BF40 and the following program POKEs 65 into all the locations. This time the 65 is interpreted not as the ASCII code for A but as signalling with bit 6 that it is not an attribute, and with bit 0 that the left-hand point of each line of 6 is in the foreground colour.

```

5 HIRES
10 FOR I = #A000 TO #BFE0
20 POKE I, 65
30 NEXT

```

Since we can POKE any value we like to the screen we can use this to set up a HIGH RESOLUTION colour display using all eight colours. The following program produces a sinusoidal striation of colour down the screen in this manner.

```

10 HIRES
20 FOR Y=0 TO 199
30 S=INT(SIN(Y/10)*8)
40 FOR X=0 TO 39
50 P=#A000+Y*40+X
60 Q=(S+X)-INT((S+X)/8)*8+16
70 POKE P,Q
80 NEXT
90 NEXT

```

By using this method of placing attributes on the HIRES screen we can produce more colours from the Oric. We simply mix two different colours together on alternate lines and we can make at least 36 different colours. The following program demonstrates some of the colours which can be achieved. Even more shades can be accessed by overlaying patterns of foreground colours on these striped backgrounds!

```

1 'COPYRIGHT PAN LTD
10 PAPER 0:INK 7
20 HIRES
30 FOR A=0 TO 7
40 FOR B=0 TO 7
50 FOR Y=B*20 TO B*20+20
60 P=#A000+Y*40+A*5
70 IF INT(Y/2)=Y/2 THEN Q=16+A ELSE Q
=16+B
80 POKE P,Q
90 NEXT
100 NEXT
110 NEXT

```

So far the subject of writing on the HIRES screen has been neglected. We can do this with the CHAR command which is one of the most powerful and potentially exciting commands on your Oric. This command allows us to place characters to an accuracy of one point not just the character positions, a feature only found on one, the most expensive, of the Oric's competitors.

The CHARACTERS are placed at the current graphics position with the command CHAR followed by three parameters. The first parameter is the ASCII code of the CHARACTER to be plotted, the second parameter is the character set from which the character is to be taken (0 for the standard set and 1 for the alternative set), and the third parameter is our old friend the fb parameter. This truly remarkable feature, which will allow animated displays and games of superb quality to be constructed on the ATMOS, is demonstrated by the program below which uses a subroutine to place a string of characters on the HIRES screen.

```

10 HIRES
20 PAPER 1 :INK 3
30 A$="ORIC POWER"
40 FOR W=1 TO 20
50 CURSET 30,50+W*3,3
60 GOSUB 100
70 NEXT
80 END
100 REM PRINT STRING WITH SPACING W
110 FOR I=1 TO LEN(A$)
120 CHAR ASC(MID$(A$,I)),0,1
130 DRAW W,0,3
140 NEXT I
150 RETURN

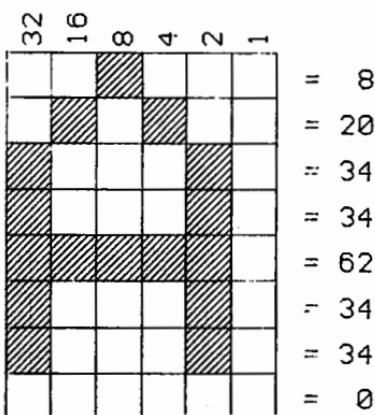
```

### Character Graphics

Having looked at the different modes available to us we now come to the most flexible of the Oric's graphics features: characters. When we mentioned games in the previous section you may have thought that there wasn't going to be much excitement in watching a string of letters zoom around the screen, no matter how accurately, but in this section we will see that we are not limited to using letters. In fact we are not limited at all in the objects we use! First, though, let's review what we know of characters.

We have used characters in both TEXT and HIRES displays and by now you might be wondering how they are made. Well the answer to this can be found by examining exactly what happens when we put a character on the HIRES screen. At this time the computer must know which points to plot in foreground colour and which are to be plotted in the background colour within the area of the character. It finds this information out by looking at a table which is kept in the memory. In this table there are stored eight pieces of data for each character. These pieces of data can be thought of as eight

binary PATTERNS each of which defines a single horizontal line of the character. In fact only the lowest six bits of the pattern are used for each line to fit in with the number of displayable pixels across each location. It will help to understand how these patterns fit together to define a character if you think of each character as being a 6 by 8 grid of blocks.



In the figure above the numbers above each block are the values of the equivalent bits in the pattern. The figures at the side of the block are the sum of the bits in that row which are on (marked by shading). This is how the Oric's normal characters are defined. In the HIRES mode this information is used to decide which points to display to produce a character. In the TEXT modes this information is encoded into the TV signal to produce the character on the display at the appropriate character position.

Unlike most computers, the Oric keeps the whole of its character defining table in RAM. This means that we can change any character to be whatever we like. This has enormous potential for games and animated displays!

When you turn your Oric on, or press the reset button, it copies the character sets, standard and alternative, from its permanent ROM memory into RAM. These tables comprise 128\*8 pieces of data for the standard set definitions, and 112\*8 for the alternate set. These tables are always stored in memory just before the screen memory. When we switch from TEXT to HIRES these tables are moved to maintain their position as being just before the screen memory. In TEXT mode they start at location #B400 (for the standard set) and location #B800 (for the alternate set). In HIRES mode the tables are moved to start at #9800 for the standard set and #9C00 for the alternate set. The position of the first of the eight pieces of data in the table for defining a character can be found by adding 8 multiplied by the ASCII code of the character in question to the start of the table. Thus the data for A is stored at #B400+8\*65

(ASCII A) which is (in decimal) 46600. Try the following program which POKEs new values into these locations and then PRINTs an A. The A should appear as an italic A because we have now redefined the way Oric draws an A. Try switching to HIRES mode and use CHAR to place an A on the screen to prove that the characters have been copied down correctly.

```
10 FOR A=0 TO 7
20 READ B
30 POKE 46600+A,B
40 NEXT
50 DATA 14,17,17,17,62,34,34,0
```

To take full advantage of this capability for changing the shapes of letters we will need to calculate many different patterns in binary. Computers are supposed to make life easier for us so let's use the Oric to do the hard work. The following listing is a CHARACTER GENERATOR program that allows you to define any shape you want and store the data in either character table. It also allows you to edit a character definition from either table. The character is displayed in large scale inside a grid and we use the full block character from the alternate set to fill in the squares. The squares are also plotted actual size on the screen as points below the grid and in this way we can directly read the binary equivalent from the screen memory (remembering that bits 6 and 7 are not part of the definition). To move the cursor use the arrow keys and to change a square press the space bar. To Save a character press S, to Edit a character press E and Quit the program use Q. The S and E options will ask you to specify which character set (0 for standard and 1 for alternate) and which character (the ASCII code) you wish to save or edit, to or from. If you wish to use the data in another program you will be able to read it off the screen where it is displayed to the right of the grid when a character is under examination.

#### Listing CHARACTER GENERATOR

```
10 HIMEM#97FF
20 GOSUB 200
30 REPEAT
40 CURSET XC,YC,3:WAIT 10:CHAR 95,1,2
50 WAIT 10:CHAR 95,1,2
```

```
60 A$=KEY$
70 UNTIL A$<> ""
80 IF A$=CHR$(9) AND XX<6 THEN XX=XX+
1 :XC=XC+6 :GOTO 30
90 IF A$=CHR$(8) AND XX>1 THEN XX=XX-
1 :XC=XC-6 :GOTO 30
100 IF A$=CHR$(10) AND YY<8 THEN YY=Y
Y+1 :YC=YC+8 :GOTO 30
110 IF A$=CHR$(11) AND YY>1 THEN YY=Y
Y-1 :YC=YC-8 :GOTO 30
120 IF A$=" " THEN GOSUB 300 :GOTO 30
130 IF A$="E" THEN GOSUB 400 :GOTO 30
140 IF A$="S" THEN GOSUB 500 :GOTO 30
150 IF A$<>"Q" THEN GOTO 30
160 STOP
200 REM INITIALISE DISPLAY
210 HIRES
220 FOR X=100 TO 136 STEP 6 :CURSET X,
30,1 :DRAW 0,64,1 :NEXT
230 FOR Y=30 TO 94 STEP 8 :CURSET 100,
Y,1 :DRAW 36,0,1 :NEXT
240 X=119 :Y=100 :M=#A000+Y*40+INT(X/6)
+1
250 XC=100 :YC=30 :XX=1 :YY=1
260 GOSUB 900
270 RETURN
300 REM CHANGE SQUARE
310 CURSET XC,YC,3 : CHAR 95,1,2
320 CURSET X+XX,Y+YY,2
330 FOR I=1 TO 8
340 P=PEEK(M+40*I) :IF P>63 THEN P=P-6
4
350 P$=MID$(STR$(P),2) :P1=180 :P2=I*8+
24 :GOSUB 600
360 NEXT
370 RETURN
400 REM EDIT CHARACTER
410 GOSUB 700
```

```
420 GOSUB 200
430 CURSET X+1,Y+1,3
440 CHAR A,S,1
450 FOR J=X+1 TO X+6
460 FOR K=Y+1 TO Y+8
470 IF POINT(J,K) THEN GOSUB 800
480 NEXT :NEXT
490 GOTO 330
500 REM SAVE CHARACTER
510 GOSUB 700
520 FOR I=1 TO 8
530 P=PEEK(M+40*I):IF P>63 THEN P=P-6
4
540 POKE (#97FF+S*#400+A*8+I),P
550 NEXT
560 GOSUB 200
570 RETURN
580 REM PRINT STRING ON SCREEN
590 CURSET P1,P2,3:FILL 8,3,64
600 CURSET P1,P2,3
610 FOR K=1 TO LEN(P$)
620 CHAR ASC(MID$(P$,K)),P3,1
630 DRAW 6,0,3
640 NEXT
650 RETURN
660 REM INPUT SET AND ASCII CODE
670 REPEAT
680 INPUT "WHICH SET";S
690 UNTIL S=1 OR S=0
700 REPEAT
710 INPUT "WHICH ASCII CODE";A
720 UNTIL A>32 AND A<128
730 RETURN
740 REM PLOT J,Kth SQUARE IN GRID
750 TX=100+6*(J-X-1)
760 TY=30+8*(K-Y-1)
770 CURSET TX,TY,3
780 CHAR 95,1,2
```

```

850 RETURN
900 REM DISPLAY CHARACTER SET
910 FOR L=32 TO 96 STEP 32:P$="""
920 FOR N=L TO L+31
930 P$=P$+CHR$(N)
940 NEXT
950 P1=20:P2=115+INT(L/3):P3=0
960 GOSUB 600
970 P1=20:P2=155+INT(L/3):P3=1
980 GOSUB 600
990 NEXT
1000 P3=0:RETURN

```

The final program in this chapter demonstrates the use of a defined character in a simple animation program of an airplane following a path specified in polar co-ordinates. This should give you some idea of the power of defined characters for your Oric and provide a fitting end to this chapter exploring the amazing graphics prowess of the ATMOS.

```

:0 HIMEM#97FF
20 FOR P=0 TO 7
30 M=#B400+8*(ASC("a")+P)
40 FOR A=0 TO 7
50 READ B
60 POKE M+A,B
70 NEXT
80 NEXT
90 REM DATA FOR a b c d e f g h
:100 DATA 30,30,6,6,38,39,38,0 →
:110 DATA 2,3,18,38,12,24,48,40 →
:120 DATA 0,57,1,63,63,16,0,0 →
:130 DATA 16,8,36,48,24,45,7,2 →
:140 DATA 0,38,39,38,6,6,30,30 →
:150 DATA 2,4,9,3,6,45,56,16 →
:160 DATA 0,39,32,63,63,2,0,0 →
:170 DATA 16,48,18,25,12,6,3,5 →
200 HIRES :PAPER 6:INK 1
210 OX=180:OY=101
220 R=80:PP=PI/100

```

```
230 REPEAT
240 FOR A=0 TO 2*PI STEP .PP
250 X=120+COS(A)*R
260 Y=100+SIN(A)*R
270 C=97+INT((A+PI/8)/(PI/4))
280 IF C=105 THEN C=97
290 CURSET OX,OY,3:FILL 8,2,64
300 CURSET X,Y,3:CHAR C,0,1
310 OX=X:OY=Y
320 NEXT
330 UNTIL KEY$<>""
```

# 8 The sound of music

The Oric sound and music facilities are extensive. For the games enthusiast who requires easy-to-use sound effects, the Oric comes with ready made ZAP, SHOOT, PING and EXPLODE. These onomatopoeic commands produce exactly the noises their names lead you to expect, and what's more they do so at impressive volume levels. Not for the Oric the insipid whimpering of some other computers' sound facilities. The Oric belts out its noises and tunes in a loud, positive voice.

The flexibility and sophistication of the other Oric sound commands are also something to shout about.

MUSIC is a command which supplies the user with a western style chromatic scale on which to base compositional programs. For sound effects there is the SOUND command which controls the noise generator necessary for simulating roaring rockets and exciting explosions. This chapter takes us out into the deep space of music on the micro. Our first stop on this journey then are the ready-made sound effect commands . . .

There are four pre-defined BASIC sound instructions on the Oric: ZAP, SHOOT, PING and EXPLODE. By simply typing in ZAP (and pressing RETURN) the sound of aliens letting fly with another blast from their laser cannon is immediately conjured up. Alternatively the one-line program:

```
10 FOR N = 1 TO 100 : SHOOT : WAIT 30 : NEXT
```

will straight away give an aural impression of a raucous evening in Dodge City. These commands are ideal game fodder and can be put to good use in your future Oric Space Inveiglers and Puc Person programs.

These simple and easy-to-use pre-defined sounds are only an introduction to the Oric's capabilities, however, and for the creative use of sound we turn to the noisy trio of SOUND, MUSIC and PLAY. In this chapter we will explain the functions and format of these instructions and explain how PLAY interrelates with the other two. Let's delve straight into the fascinating world of micro composition by looking at the Oric's flexible sound command MUSIC.

## MUSIC

MUSIC makes use of the three independent square wave generators available from the Oric's specialist sound chip as a sound source. Each tone has a separate channel, and by using MUSIC in combination with PLAY, chords and multiple part compositions are possible. For simplicity's sake we'll initially deal with Channel 1 only, since this will operate without the use of PLAY.

The syntax of the MUSIC command is:

MUSIC Channel, Octave, Note, Volume

The four values specifying Channel, Octave, Note and Volume must be separated by commas. Channel can take values of 1, 2 or 3, and defines which tone generator is in operation. For the remainder of this section we will be using Channel 1.

Octave refers to the range within which the Note parameter operates. It takes an integer value from 0 to 6, where 2 is the 'octave' directly above middle C on a piano. Octave's purpose will become more clear after looking at the Note parameter.

Note has a range of integer values from 1 to 12 inclusive. These are equivalent to the notes of a chromatic scale starting on C ( $N = 1$ ) and finishing on B ( $N = 12$ ). By using Octave and Note in combination a chromatic scale from two octaves below middle C to B four octaves above middle C is obtainable.

Volume is a self explanatory parameter and has a range from 1 (quiet) to 15 (ear-splitting).

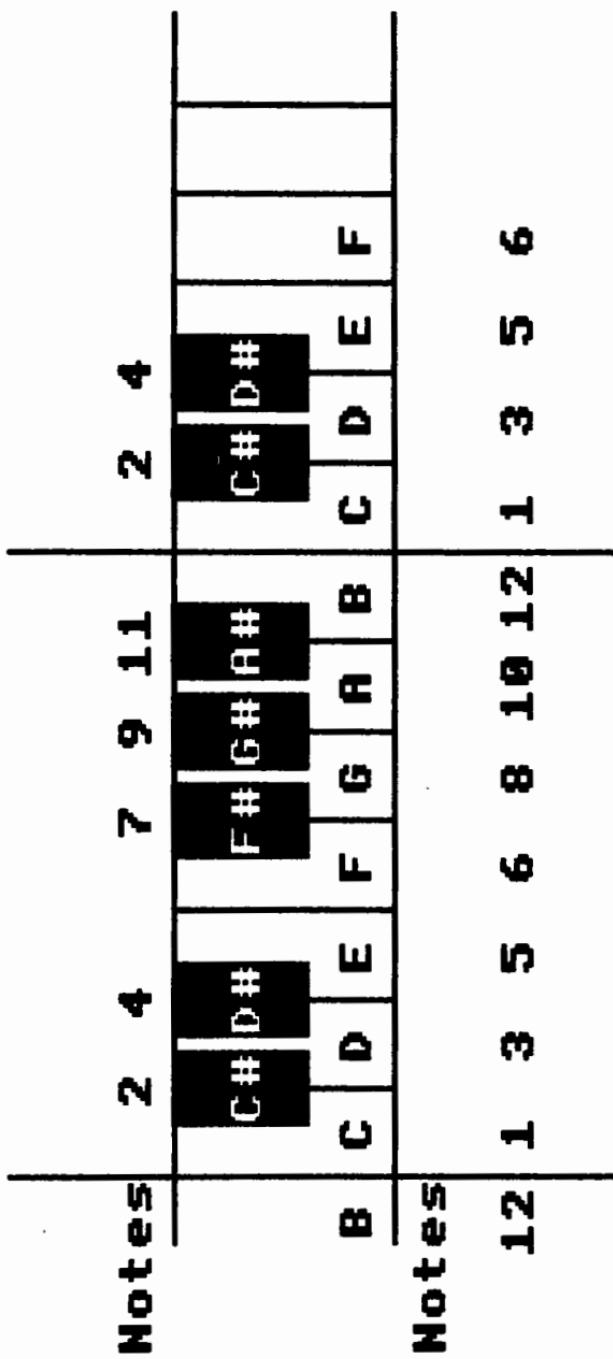
As a quick demonstration of MUSIC you can key in and RUN the program 'CHROMATIC SCALE' which plays every note in each of the seven octaves, i.e. from MUSIC 1, 0, 1, 10 (low C) to MUSIC 1, 6, 12, 10 (high B).

```

1 REM *** CHROMATIC SCALE ***
5 CLS
10 PAPER 5
20 FOR O=0 TO 6
30 FOR N=1 TO 12
40 MUSIC 1,O,N,10
50 WAIT 30
60 NEXT N
70 NEXT O
75 END

```

The elegance of the MUSIC command really becomes apparent when you relate music in the real world with MUSIC in the computer. The way Octave and Note are interdependent is an exact mirror of western musical theory. To illustrate this look at this diagram of a conventional piano keyboard:



Just as in standard music notation there are twelve symbols for the twelve notes of the chromatic scale, the Oric recognises twelve values for Note. Thus Note cannot be 13 or 14, in the same way that there are no musical notes H or I.

Using this system a major scale of C (playing all the white notes on the piano from C to C') would require:

Note = 1, 3, 5, 6, 8, 10, 12, 1

The program 'keyboard' illustrates this principle by turning the top line of keys on the Oric into a piano-type keyboard (0 being 10, '-' being 11 and '=' being 12):

1	2	3	4	5	6	7	8	9	0	-	=
C	C#	D	D#	E	F	F#	G	G#	A	A#	B

```

10 REM *** KEYBOARD ***
20 GET A$
30 A=VAL(A$)
40 IF A$="-" THEN A=11
50 IF A$="--" THEN A=12
60 IF A$="/" THEN PLAY0,0,0,0: STOP
70 IF A=0 THEN A=10
80 MUSIC 1,3,A,8
85 WAIT 20: PLAY0,0,0,0
90 GOTO 20

```

GET A\$ waits for a keypress from the keyboard while line 30 reads the value of the input into variable A. A will be the value of the number pressed, except for 0, - and = which will give respectively 10, 11 and 12. One PLAY statement is used in this program: PLAY 0, 0, 0, 0. This simply shuts off the Channel 1 tone generator when the program ends. Press the '/' key to end the program.

Due to the limitations of the Oric's (or any alpha-numeric) keyboard you are not about to be able to play 'Flight of the Bumble Bee' using the keyboard program. However, there is a way of playing technically demanding the intricate pieces of music without moving a finger (well, just the one – to press RETURN). To accomplish this we use the computer to do what it does best, remember things!

There are two ways of storing musical information in the Oric. The first of these is the DATA statement.

```

1 REM*** TUNE ***
5 READ A
10 MUSIC 1,3,A,10
25 IF A=11 THEN RESTORE

```

```

30 WAIT 25
40 GOTO5
100 DATA 3,5,6,8,10,6,10,10,9,5,9,9,8,4
,8,11

```

The program 'TUNE' uses a DATA statement to store values for Note (A). In line 5 the values are READ into the MUSIC statement and line 25 is a simple method of causing the program to repeat itself.

DATA statements have the advantage of being thrifty with memory space and so are ideal when a fanfare of some type is needed in a games program. Their disadvantage lies in the difficulty encountered in changing the DATA. It is time consuming and confusing trying to edit the melody in any way. For a more flexible storage system we must look to the second possible method of storing musical information – in arrays.

The program 'SEQUENCER' below uses the keyboard used earlier as a method of inputting Note values into a hundred note array, A(100). Line 13 sets up the array and the next part of the program is the old familiar 'KEYBOARD' program. Line 85 uses WAIT 20 followed by PLAY 0, 0, 0, 0 to give any notes the user inputs a finite length of 1/5 second. (Without this line the note would continue until another key was pressed.) Line 75 allows us to escape from inputting more notes when we have had enough and the variable G counts the number of steps in the sequence.

```

2 REM *** SEQUENCER ***
4 CLS
5 PRINT "The ORIC one hundred note sequencer"
6 PRINT"==== ===== == ===== = ===== ====="
7 PRINT
8 PRINT "Enter Notes using the top line of keys"
9 PRINT"When the last note has been entered"
10 PRINT"press ."
11 CLEAR
13 DIM A(100)
14 G=1
16 FOR N= 1 TO 100
20 GET A$
30 A(N)=VAL(A$)
40 IF A$="-" THEN A(N)=11

```

```

50 IF A$==" " THEN A(N)=12
70 IF A(N)=0 THEN A(N)=10
75 IF A$=="\\" THEN GOTO 399
80 MUSIC 1,3,A(N),8
85 WAIT 20: PLAY0,0,0,0
87 G=G+1
90 NEXT N
399 PRINT
400 CLS
405 PRINT "Set speed of sequence"
410 INPUT S
411 PRINT
412 PRINT"To Go press G."
413 PRINT"To Stop press S."
414 PRINT"To Enter a new sequence press
E."
415 PRINT"To Alter speed press A."
430 GET A$
440 IF A$=="G" THEN GOTO 500
450 IF A$=="E" THEN GOTO 2
460 IF A$=="A" THEN GOTO 399
470 IF A$=="C" THEN GOTO 1000
500 FOR N= 1 TO G
505 IF N=G THEN N=1
510 MUSIC 1,3,A(N),10
512 PLOT10,10,"No. :" +STR$(N)
513 PLOT10,12,"Note:" +STR$(A(N))
520 WAIT S
521 IF KEY$=="S" THEN GOTO 430
522 PLOT10,10,"No. : "
523 PLOT10,12,"Note: "
530 NEXTN
1000 INPUT"N";N
1015 IF N=0 THEN GOTO 400
1020 INPUT"NOTE";A(N)
1030 GOTO1000

```

Most of the rest of the program is taken up with instructions and the labelling of keys to start and stop the sequence, apart from lines 500 to 530.

At line 500 we see the use of G in allowing the sequence to cycle repeatedly, by resetting N equal to 1 if the cycle is complete, i.e. N=G. The speed of the FOR . . . NEXT loop is controlled by the line: 520 WAITS.

WAIT is an important command in music programming. The timing of notes in real music is just as important as the pitch of the notes, so in any serious music program some account must be taken of timing. In 'SEQUENCER' we can make notes last longer by simply entering them more than once. This is the method used in the commercially available sequencers incorporated in the Roland SH 101 synthesiser and the Sequential Circuits Pro-One Synth. This type of sequencer works perfectly well for repetitive and mechanical sounding melodies but is not practical when variation in phrasing is required.

We can program a more subtle compositional routine which allows both the note, pitch and the length of note to be stored. For a simple illustration of this see this next program, 'COMPOSITION':

```

1 REM *** COMPOSITION ***
10 T=1
20 CLS
30 DIML(50)
40 DIMO(50)
50 DIMA(50)
55 PRINT"TO PLAY YOUR TUNE ENTER 0 FOR
NOTE"
60 FOR N= 1 TO 50
90 PRINT
100 INPUT"NOTE 1-12:" ;A(N)
110 IF A(N)=0 THEN GOTO 500
120 INPUT"OCTAVE 1-7" ;O(N)
125 MUSIC 1,O(N),A(N),10:WAIT50:PLAY0,0
,0,0
130 INPUT"LENGTH" ;L(N)
150 NEXT
500 PLAY1,0,0,0
504 T=T+1
506 IF T=6 THEN PLAY0,0,0,0:STOP
510 FOR P=1 TO N-1
520 MUSIC1,O(P),A(P),10
530 WAIT 20*L(P)
535 IF P= N-1 THEN GOTO 500
540 NEXT P

```

This program uses the three arrays A( ), O( ) and L( ) to store the note, its octave and its length. Notes are entered through an INPUT statement so RETURN must be pressed before the note is heard. To escape from input mode to playback mode we include the line:

```
102 IF A (N) = 0 THEN GOTO 500
```

Zero must be entered in reply to the NOTE? prompt to conclude input.

This method of entering pitch and timing as numerical values has a lot in common with the very sophisticated Roland MC-4 microcomposer. This composer also expects a numerical input rather than a performance from a keyboard. The MC-4 has a number of capabilities not provided in the 'COMPOSITION' program, however! One of these is an editing facility. The ability to correct mistakes and refine program material is obviously an essential part of a useful micro-composer. This update we will leave to you but the MC-4's other advantage, polyphony, is covered by the Oric's PLAY command. (Polyphony is the playing of more than one note simultaneously.)

## PLAY

The PLAY command on the Oric has two distinct functions. The first is to enable and disable the various tone (and noise) channels allowing any of the three tone (or one noise) generators to be used simultaneously. The second is the shaping of the various envelopes. The syntax of this command is:

PLAY TChannel, NChannel, Envelope, Period

TC refers to the tone channels and NC refers to the noise channels. The PLAY command enables (i.e. turns on) any combination of the three channels (for noise or tone) using the values 1 to 7 as follows for TC and NC.

- 0 = No tone/noise channels on
- 1 = Channel 1 on
- 2 = Channel 2 on
- 3 = Channels 1 and 2 on
- 4 = Channel 3 on
- 5 = Channels 3 and 1 on
- 6 = Channels 3 and 2 on
- 7 = Channels 3, 2 and 1 on

We have been using the MUSIC command without a PLAY command in all our previous programs. This is because Channel 1 is a special case and is ON unless the command PLAY 0, 0, 0, 0 has been given. This does not apply to Channels 2 or 3 or any of the noise channels. In all programs from now on, a PLAY command is essential to the operation of the program.

As an example of PLAY in action, run the program 'CHORD' which plays a C major chord of three notes until you stop the sound by hitting CTRL C.

```

90 REM*** CHORD ***
99 PLAY 7,0,0,0
100 MUSIC 1,4,1,5
110 MUSIC 2,4,5,5
120 MUSIC3,4,8,5
140 STOP

```

This program uses PLAY like this:

PLAY 7,0,0,0

All three tone channels are enabled (ON), all three noise channels are disabled (OFF). The chord could be reduced to a two-note chord by changing the PLAY command to activate only two channels:

#### Channels

(1+2)	PLAY 3,0,0,0
(3+2)	PLAY 6,0,0,0
(3+1)	PLAY 5,0,0,0

Another example of the use of PLAY is 'ANDROIDS', a 'musical piece' composed especially for this book. PLAY 3,0,0,0 plus two MUSIC statements were used, making this a duet for Channels 1 and 2.

```

5 REM *** ANDROIDS ***
10 A=1
50 REPEAT
60 LETA=A+1
90 READ X,Y
100 PLAY 3,0,0,0
110 MUSIC 1,1,X,6
120 MUSIC 2,4,Y,5
140 WAIT15
150 UNTIL A=193
160 RESTORE
170 GOTO10
1000 DATA1,8,8,8,1,8,8,8,1,8,8,8,1,8,8,
8,1,8,8,8,1,8,8,8,3,6,10,5,3,3,10,3
1010 DATA 1,10,8,10,1,10,8,10,1,8,8,8,1
,8,8,8,1,8,8,8,1,8,8,8
1020 DATA 3,3,10,5,3,6,10,6,1,10,8,10,1
,10,8,10

```

```
1030 DATA 1,8,8,8,1,8,8,1,8,8,8,1,8,8  
,8,3,3,10,5,3,3,10,3  
1040 DATA 1,1,8,1,1,1,8,1,1,1,1,8,1,1,1,8  
,1,1,1,8,1,1,1,8,1,1,1,8,1,1,1,8,1  
1050 DATA 1,8,8,8,1,8,8,8,1,8,8,8,1,8,8,  
8,1,8,8,8,1,8,8,8,3,6,10,5,3,3,10,3  
1060 DATA 1,10,8,10,1,10,8,10,1,8,8,8,1  
,8,8,8,1,8,8,8,1,8,8,8  
1070 DATA 3,3,10,5,3,6,10,6,1,10,8,10,1  
,10,8,10  
1080 DATA 1,8,8,8,1,8,8,8,1,8,8,8,1,8,8  
,8,3,3,10,5,3,3,10,3  
1085 DATA 1,1,8,1,1,1,8,1,1,1,8,1,1,1,8  
,1,1,1,8,1,1,1,8,1,1,1,8,1,1,1,8,1  
1090 DATA 3,3,10,3,3,3,3,10,3,5,5,12,5,5,  
5,12,5  
2000 DATA 6,6,1,6,6,6,1,6,1,1,8,1,1,1,8  
,1  
2010 DATA 3,3,10,3,3,3,3,10,3,5,5,12,5,5,  
5,12,5  
2020 DATA 6,6,1,6,6,6,1,6,1,1,8,1,1,1,8  
,1  
2030 DATA 3,3,10,3,3,3,3,10,3,5,5,12,5,5,  
5,12,5  
2040 DATA 6,6,1,6,6,6,1,6,1,1,8,1,1,1,8  
,1  
2050 DATA 3,3,10,3,3,3,3,10,3,5,5,12,5,5,  
5,12,5  
2060 DATA 6,6,1,6,6,6,1,6,8,8,3,8,8,8,3  
,8
```

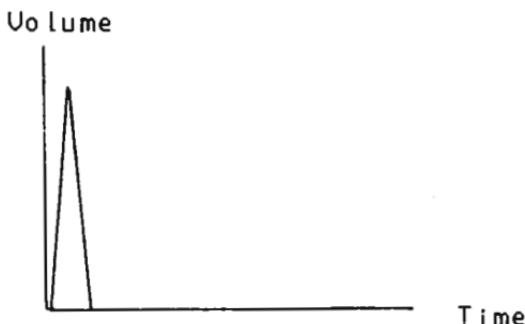
This is another example of the use of DATA statements to hold the musical information. In this case the DATA consists of pairs of numbers, the first of which is the bass line, and the second the melody.

Note that lines 1000 to 1040 are identical to lines 1050 to 1095, so you can save yourself some typing by using CTRL A (the copying function). Also lines 1090 and 2000 can be copied for lines 2010, 2020 and 2030, 2040.

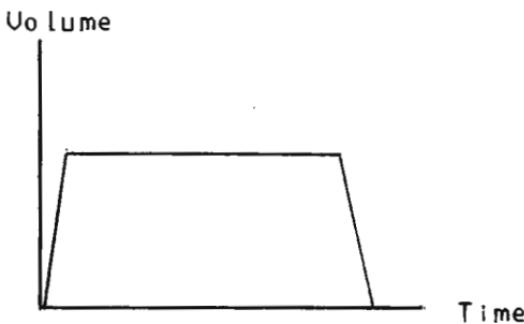
You may encounter another potential problem with DATA statements at this point, since one error in copying will change the theme for a space western into avantgarde jazz!

Getting back to PLAY, you were probably wondering what the last two parameters, Envelope and Period, are all about. Those of you familiar with synthesiser theory may already be accustomed to the concept of a volume envelope, but for the rest of you we will take a few lines to tell you what they are all about. A volume envelope is simply a graph of volume against time. As examples we can take the sound of a snare drum against, say, James Galway (playing the flute).

### SNAREDRUM



### FLUTE

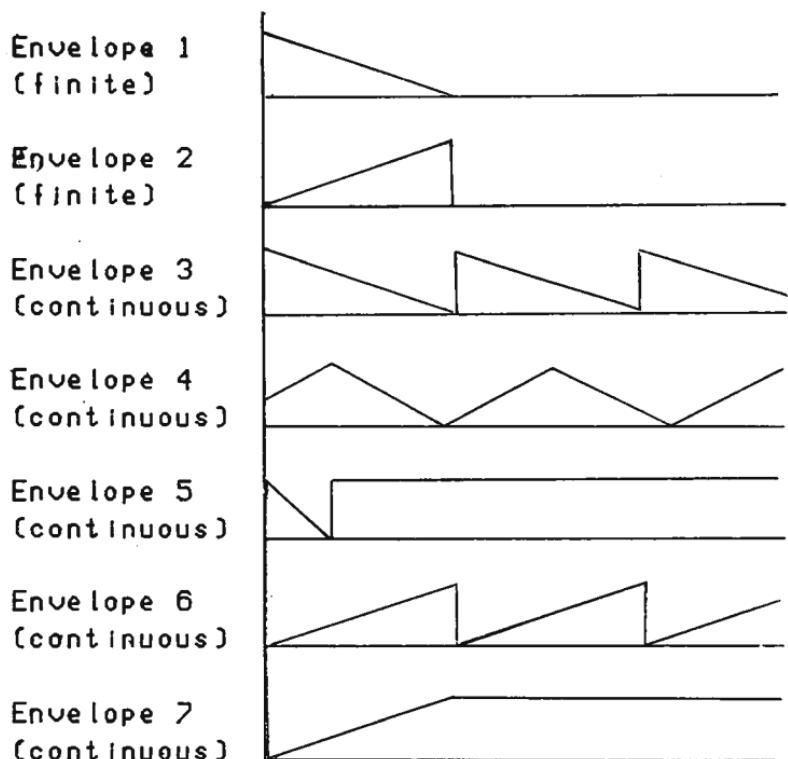


Percussion instruments such as drums and marimbas have rapid *attacks* (i.e. attain maximum volume quickly) and die away quickly, whereas flutes, violins, etc., have gentle attacks and sustain their volume for extended lengths of time (as long as Mr Galway's breath holds out, for example).

The crew down at Oric have supplied us with a selection of envelopes to choose from (seven in all) which they hope will serve any occasion. Whilst this is not entirely true, their system is certainly much easier to use than a

system of completely definable envelopes.

Graphs for each envelope value have the shapes given below. Note that 1 and 2 are *finite* (have a finish point), whilst envelopes 3 to 7 are continuous, and will sound a note until switched off. Envelope 0 exists, but is exactly the same as Envelope 1 shown here.



The envelope Period can take values from 0 to 32767 and controls how long the noise or note takes from start to end in the case of the 'finite' envelopes. For the continuous envelopes the Period controls the 'length' of the waves and is best understood by running the program 'ENVELOPE TEST' and trying some different values for the Period.

```

5 REM *** ENVELOPE TEST ***
8 CLS
10 INPUT"ENTER TONE CHANNEL 1,2 or 3";T
20 IF T<1 OR T>3 THEN 10
30 INPUT"ENTER THE ENVELOPE MODE, 0 TO
7";M
40 IF M<0 OR M>7 THEN 30

```

```

50 INPUT"ENTER THE ENVELOPE PERIOD, 0 T
0 32767";P
60 IF P<0 OR P>32767 THEN 50
70 CLS
80 PRINT"CHANNEL "T
90 PRINT"ENVELOPE MODE" M
100 PRINT ENVELOPE PERIOD" P
110 MUSIC T, 3, 4, 0
120 PLAY T, 0, M, P
130 PRINT"PRESS RETURN IF SOUND CONTINU
ES"
140 GOTO5
505 IF N=G THEN N=1

```

In line 110 of the program you will notice that the value for volume is set to 0, with MUSIC T, 3, 4, 0. Setting volume equal to 0 gives control over to the PLAY Envelope parameter. Any other value of volume will cause the Envelope parameter to be ignored.

For Envelope type 1, Periods around 1000 will give a sharp banjo-type effect, whereas Periods of approximately 32700 give a sharp attack but slow decay.

For Envelope type 6, a Period value = 19 gives an interesting machine-like effect. Hours of amusement can be had by all the family, seeing who can create the daftest sounds using the 'ENVELOPE TEST' program.

Another vital PLAY command you must always have up your sleeve is PLAY 0, 0, 0, 0, especially if your wonderful musical composition has ended in an annoying whine that you can't get rid of. Before you resort to the reset switch or pulling the plug please try the above. Another point to bear in mind is that the keyboard click will normally stop a persistent note but can also affect your program, so remember that it can be switched off (and on again) using CTRL F.

## SOUND

SOUND is basically a cruder version of MUSIC without the tone channels arranged in semitones. It has the added bonus, however, of a noise generator, making it ideal for the creation of sound effects. The SOUND command has the format:

SOUND Channel, Pitch, Volume

Channel can take values between 1 and 6. 1, 2 and 3 refer to the three tone generators (as in MUSIC). 4, 5 and 6 specify which tone channel the noise is mixed into (and comes out of). There is only one noise generator, and it

outputs noise into Channel 1, 2 or 3 according to whether 4, 5 or 6 is specified.

Pitch refers to the pitch of the tone or noise selected (i.e. the frequency). Unlike Note, Pitch is not arranged in semitones so is less useful for musical purposes. It can take values from 0 to 65536, where 65535 is very low and Pitch < 5 attracts dogs. The noise channels can also be selected for varying Pitches. This is an unusual but useful facility which is best described by reference to the program 'WAVES'.

```

180 REM*** WAVES ***
200 PLAY 0,1,0,0
205 Z=INT(RND(1)*20)
210 FOR I =0 TO 31
220 SOUND 4,I,7
230 WAIT Z
240 NEXT I
250 GOTO205

```

As you can hear, the noise gives the impression of changing pitch. (It is not mixed with tone.) When applied to NOISE, Pitch can take the values 0 to 31, or any multiple of these values ( 32 - 63, 64 - 95, etc.), for a complete 'sweep'. Programs of this types are particularly effective when linked to graphics, but we'll leave it up to you to conjure up images of waves rushing us a sandy shore!

The final parameter in SOUND is, of course, volume. Volume is identical to its namesake in MUSIC statements and can take integer values from 1 to 15. As before a zero value for volume gives the PLAY command envelope control. Just as with MUSIC, SOUND must always be used with a PLAY statement. (The exception being Channel 1 if used for tone only.) In 'WAVES' Channel 1 was selected by using PLAY 0,10,0 thus disabling the Channel 1 tone generator, but enabling Channel 1 for noise. The noise was selected in the SOUND statement by giving the Channel parameter the value of 4 (SOUND 4, I, 7). Setting the volume equal to 0 allows the seven PLAY envelopes to be used as described in connection with MUSIC.

Another function of SOUND with PLAY is multiple SOUND statements. This does not usually create chords as does MUSIC, but can be used to produce complex effects, such as the dynamic 'TEN MILE ISLAND'.

```

10 REM*** TEN MILE ISLAND ***
20 FOR P=31      TO 1 STEP -1
40 SOUND 2,P+100,15
50 SOUND 1,P*10,15
60 SOUND 3,P*2+6,10

```

```
70 SOUND 4,P,15
100 PLAY7,1,0,0
150 WAIT50
170 IF P=1 THEN EXPLODE
200 NEXT
```

This program uses four SOUND statements. The three tone statements change pitch at differing rates in different ranges while SOUND 4,P,15 selects Channel 1 noise to sweep at maximum volume level. PLAY7,1,0,0 enables all three tone generators but only one channel of noise. The China Crisis nuclear power station melt-down situation is then terminated with the old faithful EXPLODE. This brings us back to where we started in our journey through the sound of music on the Oric microcomputer. The rest is up to you. The programs given can only provide indications of what is possible with your versatile and tuneful Oric. We hope you will have many hours of enjoyment creating music and noises to amaze your friends and annoy your neighbours.

# 9 Oric BASIC keywords

In this section each of the BASIC keywords is considered along with an example of its usage. Information is provided on the format for use with each command and the following standard symbols are used throughout:

- v represents a numeric variable name
- v\$ represents a string variable name
- i, j represents integers or whole numbers
- n represents a floating-point or decimal number (which may be an integer), or the number resulting from evaluation of a numeric expression
- c represents a conditional logical expression, eg A>B or TRUE
- a\$ represents a string or string expression
- ln represents a program line number
- addr represents a memory location address

To clarify the graphics commands we will also utilise the following symbols:

- fb the foreground/background parameter (0-3). See Chapter 7 for a full explanation of the various values.
- s this is the character set descriptor which is either 0 (for the standard set) or 1 (for the alternate set)
- x, y represent x and y screen co-ordinates

Other symbols will be used as appropriate. These will be defined within the body of the keyword definition. The BASIC Token values given are the decimal values which the Oric uses to store recognised BASIC keywords in single-byte form in memory.

## ABS

**BASIC Token:** 216  
**Format:** ABS(n)

Returns the absolute magnitude of the number or expression contained in brackets. For example ABS (-19) is 19. This function can clearly be of use when it is necessary to ensure a positive solution to a calculation of two

variables. For example, in the calculation (A-B), the difference will obviously only be positive as long as A is greater than B. However, ABS(A-B) will always return a positive value.

```

10 REM *** ABS ***
20 FOR C=-500 TO 1000 STEP 150
30 PRINT ABS(C),
40 IF C<0 THEN PRINT "BC" ELSE PRINT "A
D"
50 NEXT
60 END

```

The example program above PRINTS out the date from 500BC in steps of 150. Without ABS this loop would generate negative BC years (e.g. -350BC).

```

10 REM *** ABS 2 ***
20 LET A=COS(PI)
30 IF A=-1 THEN PRINT"COS(PI)=";A
40 IF ABS(A+1)<1E-9 THEN PRINT"COS(PI)=
";A;" WITHIN COMPUTER ACCURACY"
50 PRINT"SO ";A;"=-1"

```

ABS is also useful in checking equalities since, due to inevitable inaccuracies in converting decimal to binary, and vice-versa, within the computer, rounding errors can occur. In the above program the computer calculates COS(PI) which is -1 but is not stored exactly in the computer's memory. Using ABS in line 40 checks that the calculated value is correct within the limits of conversion errors.

**Related Keywords:** SGN

## AND

**BASIC Token:** 209  
**Format:** i AND i  
           c AND c

The BASIC keyword AND is a logical condition operator which works in much the same way as the common English usage of AND. When used to combine logical expressions it will give an answer of TRUE (= -1) OR FALSE (= 0) depending, respectively, on whether both logical expressions are TRUE or not. It is often used in this way inside an IF statement to ensure that two

conditions are satisfied before a statement is executed. The first example program shows AND in use in this way where the GOTO statement at the end of line 40 will not be executed unless both A>12 and A<20.

```

10 REM *** AND ***
20 CLS:PRINT:PRINT
30 INPUT "INPUT AGE";A
40 IF A>12 AND A<20 GOTO 70
50 PRINT "YOU ARE NOT A TEENAGER"
60 END
70 PRINT "YOU ARE A TEENAGER"
80 END

```

We can also use AND to combine the binary representations of numbers in the memory. When the command is used in this way each pair of corresponding bits along the two numbers in question is treated as a combination in the same way as above, with the bit in the resultant number being set if both the bits being combined were set. Thus 12 AND 9 (00001100 AND 00001001) will give 8 (00001000).

**Related Keywords:** FALSE, NOT, OR, TRUE

## ASC

**BASIC Token:** 236  
**Format:** ASC(a\$)

Every character available on your Oric has a corresponding ASCII (pronounced as-key) code number. Thus when we create a line such as:

```
10 P$="M"
```

the Oric does not actually store the letter M, but its ASCII equivalent (which in this case is 77).

As the example program below demonstrates, ASC can also be used to monitor INPUT. Line 40 checks to see whether the first character of the string you have INPUT falls between ASCII codes 65 and 90. Since codes 65–90 correspond to the capital letters the computer will only PRINT out any INPUT which starts with a capital letter.

```

10 REM *** ASC ***
20 CLS
30 INPUT "ANY STRING";K$:CLS:PRINT:PRIN
T

```

```

40 IF ASC(K$)>64 AND ASC (K$)<91 THEN P
RINT K$
50 WAIT 100
60 GOTO10
70 NEXT
80 END

```

Since they can function as line delimiters, this type of routine will only accept " , : , " " " " or " " if they are enclosed in double quotes. In case you are interested ASCII is actually an acronym for American Standard Code for Information Interchange, and a complete list can be found in Appendix 1.

**Related Keywords:** CHR\$, HEX\$, STR\$, VAL

## ATN

**BASIC Token:** 229  
**Format:** ATN(n)

This function returns the angle of any known tangent. It should be noted that the angle returned by the Oric is expressed in radians. The value returned will always lie within the 'principal range' for TAN of -PI/2 to PI/2. Thus ATN(1) will give PI/4 or 0.785398162 for the angle whereas any angle of the form PI/4+n\*PI (where n is an integer) would also have a TAN of 1. This can be useful in graphics programs for calculating the angle of a line from the quotient of the differences of the x and y co-ordinates of its end points.

```

10 REM *** ATN ***
20 HIRES
30 INPUT "ENDPOINTS,X1,Y1,X2,Y2";X1,Y1,
X2,Y2
40 CURSET X1,Y1,3
50 DRAW X2,Y2,1:WAIT 100
60 ANGLE=ATN((Y2-Y1)/(X2-X1)):TEXT
70 PRINT"ANGLE IS";ANGLE;"RADIAN"
80 PRINT"OR";ANGLE*180/PI;"DEGREES"
90 END

```

**Related Keywords:** COS, PI, SIN, TAN

**CALL**

**BASIC Token:** 191  
**Format:** CALL addr.

This command transfers control to the machine code routine which starts at memory location addr. Return to BASIC is effected by using the machine code RTS instruction to end the routine. Using this command can result in the loss of your program if addr. is specified incorrectly and is not, in fact, the start of a machine code routine. Refer to the chapter on using machine code for a fuller appraisal of this command.

Try the following command for an example of the potential danger and power of this keyword:

CALL DEEK (#FFFC)

This will transfer control to the cold start vector, re-starting the Oric as though it had just been turned on. Similarly CALLING the address returned by DEEK (#FFFF) will cause the Oric to perform a warm start as though you had pushed the Oric's reset button. This can be handy to restore the original characters on exiting a program, or to avoid having to turn the Oric over.

**Related Keywords:** DEEK, DOKE, PEEK, POKE, USR

**CHAR**

**BASIC Token:** 176  
**Format:** CHAR n, s, fb

For Oric owners with an interest in graphics or game playing CHAR is a godsend, since it enables characters to be positioned anywhere on the HIRES screen. CHAR prints at the current point position and used in conjunction with CURMOV and CURSET allows the programmer to place text or user-defined characters to an accuracy of a single point. In the format above n (32-127) represents the character's ASCII code, s (0 or 1) is the set parameter which determines whether the standard or alternate set is to be used, and fb (0-3) is the foreground/background parameter corresponding to one of the effects listed below.

- 0 Background colour
- 1 Foreground colour
- 2 Invert colour
- 3 Null

If you try to use CHAR on anything other than the HIRES screen the computer

will throw up an error message. It should be noted that the maximum CURSET co-ordinates are x=234 and y=192.



```

1 REM***CHAR***
5 HIRES:C=0
10 FORA=0TO150STEP50
20 CURSET40+A,90,3
30 DRAW20,0,2:DRAW0,20,2
40 DRAW-20,0,2:DRAW0,-20,2
50 CURMOV7,6,3
60 CHAR49+C,0,2
70 C=C+1
80 NEXT

```

**Related Keywords:** CURMOV, CURSET, HIRES, PLOT

## CHR\$

**BASIC Token:** 237  
**Format:** CHR\$(i)

This function is valuable to Oric programmers for a number of reasons. CHR\$(i) returns the character whose code is contained in the brackets. The number must be an integer in the range 0–255. However the use of this function extends far beyond the reproduction of the standard character set, and a quick glance through the complete list of ASCII codes in Appendix 1 should reveal some of the potential of CHR\$. For example the program below uses CHR\$ to set the attributes which alter the foreground and background colours for the remainder of the PRINT statement line.

```

10 REM *** CHR$(i) ***
20 PRINT CHR$(129);
30 PRINT CHR$(150);
40 PRINT " RED ON CYAN "
50 END

```

A couple of hours' experimentation with this function used in conjunction with codes in the ranges 0–31 and 128–151 will reveal programming potential that you may never have expected from your Oric.

Owners of the Oric printer will also find CHR\$ valuable when used in conjunction with the LPRINT command. (See the control codes in Appendix 7.) Used in this way, CHR\$ enables characters outside the normal character set to be printed and offers users the opportunity to alter the print size. CHR\$ is also used with standard printers to send control codes for various printer functions.

**Related Keywords:** ASC, HEX\$, STR\$, VAL

## CIRCLE

**BASIC Token:** 173

**Format:** CIRCLE n, fb

This command draws a circle in HIRES mode whose centre is the current cursor position. Thus CIRCLE is usually used in conjunction with the CURSET command. If any part of the CIRCLE leaves the screen, the Oric will throw up an error message. The first, integer, parameter is the radius in points (1–99) and fb is 0–3. In the example program, four sets of circles are drawn at four different cursor positions. The Oric then overdraws with fb set at 0 (background).

```

10 REM *** CIRCLE ***
20 HIRES
30 FOR A=0 TO 30 STEP 10
40 FOR B=1 TO 0 STEP -1
50 CURSET 80+A,100,0
60 CIRCLE A+9,B
70 NEXT B
80 NEXT A
90 END

```

**Related Keywords:** CURMOV, CURSET, DRAW, HIRES, PATTERN

## CLEAR

**BASIC Token:** 189

**Format:** CLEAR

This command wipes the values of all variables currently in use. The example program PRINTS five variables, both numeric and string. Line 80

CLEARs the variables and lines 90 and 100 PRINT the null values (0 for numeric variables, and the empty or null string for string variables).

```

1 REM *** CLEAR ***
5 CLS
10 A=5
20 B=10
30 C=20
40 PRINTA,B,C
50 C$="OR"
60 B$="IC"
70 PRINTC$+B$
80 CLEAR
90 PRINTA,B,C
100 PRINTC$+B$
```

**Related keywords:** NEW, RUN

## CLOAD

**BASIC Token:** 182  
**Format:** CLOAD "filename" (,S)  
 CLOAD "" (,S)  
 CLOAD "filename", J (,S)  
 CLOAD "filename", V (,S)

This command enables you to load a BASIC program or machine code file from a cassette. It can take a number of formats:

CLOAD ""

If the cassette contains only one program (or you are fairly sure about the location of a particular program) this format can be used since it loads the first complete program that it finds. Like all CLOAD formats the command must be accompanied by the optional ,S if the program was CSAVED in the slow mode.

CLOAD "filename"

If you are searching for a particular program where the correct filename is known, the above format can be used in which "filename" is any name of up to sixteen characters in length. However it is advisable to keep filenames as brief and easy to remember as possible, because if you add or extract any

character, even a space, the program will fail to load. If you forget a filename you must use the first format (CLOAD "") and plough through the entire cassette until you find the program for which you are searching.

The Oric will produce messages informing you of the current status. It will display SEARCHING.. on the status line until the specified program is encountered, and then LOADING. A file name is shown whenever a program is encountered on tape. Program names are followed by B, to indicate a BASIC file, and a saved memory block by C for machine code file. Files found but not to be loaded are noted with the message FOUND.. FILENAME followed by B or C, and the correct file gives LOADING.. FILENAME (B or C). The Oric will also display an ERRORS FOUND message if errors were detected on loading, and will inhibit AUTO-run if this was specified in the CSAVE command.

If you have CSAVED a block of memory locations under a particular filename you will have had to specify the beginning and end of that block in your CSAVE instruction. However, when CLOADING such a file only the file name is required.

CLOAD can be used for two additional purposes:

#### CLOAD "filename",J

The above format allows you to join a second BASIC program to the end of a previously loaded program. However it must be stressed that all the line numbers in the second program *must* be higher than the largest line number in the first program. If this is not the case the program will not RUN because the duplicated line numbers from the second program will co-exist with their counterparts in the first.

#### CLOAD "filename",V

This final format allows you to VERIFY that your program (or memory block) has CSAVED correctly. By keying in the above statement and loading the program in the usual way, your original program remains in the Oric's memory. When the program starts to load the Oric will print VERIFYING 'FILENAME' (or whatever your program is called) at the top of the screen, and if the loading process is successful you will get a "0 verify errors detected" message at the current cursor position. This indicates that the program has been CSAVED satisfactorily and it is safe to wipe it from memory. (Although it is always wise to make more than one copy of any program.) If the program has failed to VERIFY, your original listing is still safe in the Oric's memory and you will have to try and CLOAD/VERIFY again. If after a number of attempts the Oric still fails to come up with a "0 verify errors detected" report (and you have tried adjusting the volume and tone on your cassette recorder) you will be forced to conclude that the CSAVEing has been unsuccessful, and you will have to CSAVE again.

It should be remembered that CSAVE and CLOAD operate in the fast mode

(baud rate 2400) unless the statement in which they are included ends with .S, which specifies data transfer in the slow mode (baud rate 300). If a program has been CSAVED in the slow mode it can only CLOAD in the slow mode, so that final .S must be included in the CLOAD statement.

**Related keywords:** CSAVE, RECALL, STORE

## CLS

**BASIC Token:** 148  
**Format:** CLS

This clears the screen and sets it to the current background colour. As a general rule, it is advisable to start all text-based programs with a CLS, unless you have a good reason for wanting the program to remain visible on the screen. In the example program the Oric clears the screen twice. The first time in line 20, where the screen is emptied in preparation for the display, and secondly in 70, after the screen has filled with text. (This is the same as using CTRL-L.)

```
10 REM *** CLS ***
20 PAPER 0 : INK 5 : CLS
30 FOR A=0 TO 134
40 PRINT"ORIC",
50 NEXT
60 WAIT 100
70 CLS
80 WAIT 100
90 GOTO 20
```

**Related keywords:** HIRES, LORES, TEXT

## CONT

**BASIC Token:** 187  
**Format:** CONT

This direct command is used to restart a program after a break. For example, while developing a program you will doubtlessly use CTRL-C to stop the action to examine screen displays or the status of a particular variable. In many instances you will not want to re-RUN the entire program, but allow it to continue from the point at which you stopped it. This is where CONT comes in useful and it is entered as a direct command. Note

that CONT will not allow you to restart the program if you have altered any part of the program. It must only be used as a direct command, since if included in the body of a program CONT will cause the program to crash.

**Related keywords:** RUN, STOP

## COS

**BASIC Token:** 226  
**Format:** COS(n)

Calculates the cosine of angle (n). It is important to remember that the number in brackets is expressed not in degrees but in radians. To convert radians to degrees, since  $2\pi$  radians equals  $360^\circ$ , we multiply by  $180/\pi$ . The example below contrasts COSINE and SINE curves.

```

10 REM *** COSINE AND SINE ***
20 HIRES:PRINT:PRINT:PRINT
30 INPUT "COS OR SIN";A$
40 INPUT "VALUE (1 TO 99)";U
50 CURSET 0,100,3:DRAW 239,0,1
60 FOR A=40960 TO 49079 STEP 40
70 POKE A, INT(RND(1)*2)+16
80 NEXT
90 FOR A=-PI TO PI STEP 0.02
100 IF A$="COS" THEN B=COS(A)
110 IF A$="SIN" THEN B=SIN(A)
120 CURSET A*38+120,(B*U)+99,1
130 NEXT
140 PRINT A$ " CURVE":WAIT 100
150 INPUT "RETAIN PRESENT CURVE (Y/N)";M$
160 IF M$="Y" THEN 30
170 GOTO20

```

**Related keywords:** ATN, PI, SIN, TAN

**CSAVE**

**BASIC Token:** 183  
**Format:** CSAVE "filename" (,S)  
 CSAVE "filename", A addr.,  
 E addr. (,S)

This command is used to save a program or a series of memory locations on to cassette. It can take a number of formats:

CSAVE "filename"

This is the most common construction in which "filename" is any name of up to sixteen characters in length. The use of this instruction will save the current BASIC program, which will be stored under the specified filename. Like all CSAVE commands it can be followed by an optional comma S (,S). This will save the file in the 'slow' mode. Whilst the Oric CSAVES reliably in both the slow and fast modes, it is always worth making at least one copy of any important program in the slow mode.

CSAVE "filename", AUTO

This works in exactly the same way as the first example, except that once the program has been re-loaded into the Oric, it will RUN AUTOMATICALLY.

It is also possible to save the contents of any sequential block of memory locations, using A followed by the start address (in hex or decimal), to specify the beginning of the block, and E to specify the end address. Take the example of a TEXT or HIRES screen that you might want to preserve. The CSAVE format for such an operation is as follows (48k, 16k take 32768 from addresses):

For the HIRES screen:

CSAVE "filename", A40960, E48000

For the TEXT/LORES screen:

CSAVE "filename", A48000, E49119

**Related keywords:** CLOAD, RECALL, STORE

**CURMOV**

**BASIC Token:** 171  
**Format:** CURMOV x, y, fb

This command shifts the cursor to a new position in HIRES mode. The values of x and y are not absolute values, but relative to the current cursor position. fb is 0, 1, 2, or 3. CURMOV is useful when simulating motion on the

screen, and indeed in the generation of most graphics routines. The simple example below shifts a circle across the screen and then uses the fb quantity to erase it again.

```

10 REM *** CURMOU ***
20 HIRES: INK 5: PAPER 4
30 FOR C=0 TO 100 STEP 20
40 FOR B=1 TO 0 STEP -1
50 CURSET 20,50+C,0
60 FOR A=0 TO 30
70 CIRCLE 10+A,B
80 CURMOU 5,0,0
90 NEXT A
100 NEXT B
110 NEXT C

```

**Related keywords:** CIRCLE, CURSET, DRAW, HIRES

## CURSET

**BASIC Token:** 170

**Format:** CURSET x, y, fb

This determines the absolute x, y position of the cursor in HIRES mode. To stay within range, the final position of x must be between 0 and 239, whilst y should be between 0 and 199. As always fb is an integer 0-3. Since the Oric's CIRCLE command places the centre of the circle at the current cursor position, the short program below is the perfect demonstration of this command in action.

```

10 REM *** CURSET ***
20 HIRES: INK 5:PAPER 4
30 FOR C=0 TO 100 STEP 20
40 FOR B=1 TO 0 STEP -1
50 CURSET 20,50+C,0
60 FOR A=0 TO 30
70 CIRCLE 10+A,B
80 CURMOU 5,0,0
90 NEXT A
100 NEXT B
110 NEXT C

```

**Related keywords:** CIRCLE, CURMOV, DRAW, HIRES

**DATA**

**BASIC Token:** 145  
**Format:** DATA n, n, n, . . .  
 DATA a\$, a\$, a\$, . . .

This instruction precedes and defines a list of DATA which, when used in conjunction with READ, can be READ into variables. The list can take the form of numbers, words or letters, and if you wish to preserve leading spaces the DATA must be enclosed in quotes. DATA statements can be positioned at any point in a program, regardless of where they are actually READ. If you wish to include a comma in a DATA statement as part of the DATA item it must be enclosed in quotation marks. It is essential that the DATA is matched by an appropriate variable - i.e. numeric for numbers, string for words, letters and symbols. DATA is READ strictly in the order in which it appears in a given DATA statement. Only on implementing the RESTORE command can the pointer be returned to the beginning of the DATA line. In the example program the DATA in lines 70 and 90 are READ in line 20 and PRINTED in line 30.

```

10 REM *** DATA ***
20 FOR I=1 TO 2: FOR J=1 TO 2:READ A$,
A
30 PRINT A$,A
40 NEXT
50 NEXT
60 END
70 DATA "HELLO",1,"ALL",2
90 DATA "ORIC",3,"OWNERS",4

```

**Related keywords:** READ, RESTORE

**DEEK**

**BASIC Token:** 231  
**Format:** DEEK(addr.)

This function allows us to examine the value stored in memory in the two locations at addr and addr+1. It computes automatically the value (0-65535) stored in the double byte specified. The standard format that the 6502 processor uses for storing addresses is as two bytes with the first being the least significant. What DEEK does is to take the value stored in the second, most significant, byte from addr+1 and multiply it by 256 then add the value of byte stored at location addr. In the explanation of CALL we saw

how DEEK could be used to examine addresses stored in the computer's ROM. As a further example of the use of DEEK we can use DEEK(#A6) to examine the current value of HIMEM which is not normally accessible.

**Related keywords:** CALL, DOKE, PEEK, POKE, USR

## DEF

**BASIC Token:** 184

**Format:** DEF FNv(z)= exp  
DEF USR=addr.

DEF FN is used to DEFine a numeric FUNCTION. The Oric is equipped with a number of its own predefined numeric functions, but the DEF FN command allows you to build your own defined functions into a program. Essentially DEF FN can be used to avoid reproducing a line which performs the same calculation each time that particular calculation is required. In the format line above, v (a single-letter variable name) completes the name by which the function can be recalled (using the FN command), later in the program. The z in brackets is the name which is used for the argument in the function and is known as the dummy variable. The (exp) following the equals sign represents an expression using z (the dummy variable). This is the calculation which will be carried out upon the argument passed by the FN call. In the example below the loop determines the value of A which is used as the argument in the function call.

```
10 REM *** DEF ***
20 DEF FN M(Z)=Z/6*2
30 FOR A=10 TO 100 STEP 10
40 PRINTFN M(A)
50 NEXT
```

DEF USR is used to DEFine the starting address for a USER supplied machine code routine. This is discussed in the section on USR.

```
10 REM *** DEF USR ***
20 DEF USR=5120
30 FOR I=0 TO 17
40 READ A:POKE 5120+I,A
50 NEXT
60 DATA 162,5,189,12,20,157,128,187,202
,208,247,96
70 DATA 0,#48,#45,#4C,#4C,#4F
```

```

80 REPEAT
90 DUMMY=USR(0)
100 WAIT 50
110 FOR I= 48001 TO 48006
120 POKE I,32: WAIT 10
130 NEXT
140 WAIT 50
150 UNTIL FALSE

```

**Related keywords:** FN, USR

## DIM

<b>BASIC Token:</b>	147
<b>Format:</b>	DIM v(i, j, . . .)
	DIM v%(i, j, . . .)
	DIM v\$(i, j, . . .)

The DIM instruction is used to DIMENSION arrays. Arrays allow us to store floating point numbers, integers or strings as elements in a one dimensional array (a list), or a multi-dimensional array. A simple list of six integer numbers, for example, has space allocated in memory for storage with the instruction:

DIM A%(5)

The array variable (A in this case) must be a single letter for all arrays. The array A%(5) has six elements, numbered 0 to 5, which can be assigned values with statements such as:

LET A%(3)=276

String arrays are designated by a \$ sign after the variable name, e.g. D\$(3), and floating-point numeric arrays have just the single-letter variable name, such as T(8).

The Oric will allow the use of arrays with 11 or less elements (subscripts 0 to 10), without the use of a DIM statement. This is done simply by assigning a value to one of the elements of the array. The use of an instruction such as LET N(5)=6 will automatically create an array N(10) and assign the value 6 to the sixth element N(5). If we wish to use arrays that have a greater number of elements then a DIM statement is required. Arrays are usually DIMENSIONED at the beginning of a program, and once a particular array has been created using DIM it may not be changed at any subsequent point in the program, or a REDIM'D ARRAY error is produced. The following program uses DIM to set

up the string array D\$(7), assigns a value to each element using READ and DATA, and then displays a list of the contents.

```

10 REM *** DIM ***
20 DIM D$(7)
30 FOR I=1 TO 7
40 READ D$(I)
50 NEXT
60 DATA "ORIC.", "CAL ", "FOR ", "BETI", "
DIM ", "ALPHA", "EXAMPLE "
70 FOR A=1 TO 6
80 FOR B=A TO 7
90 IF D$(A)<D$(B) THEN 110
100 T$=D$(A):D$(A)=D$(B):D$(B)=T$
110 NEXT
120 NEXT
130 FOR I=1 TO 7
140 PRINT D$(I);
150 NEXT
160 END

```

Arrays with more than one dimension may be used. In the example following, a two dimensional array of integers, N%(3,4), is DIMensioned, values assigned to the elements and then the array is printed out as a 4 column by 5 row table.

```

10 REM *** DIM ***
20 DIM Nx(3,4)
30 FOR A=0 TO 3
40 FOR B=0 TO 4
50 Nx(A,B)=A*10+B
60 NEXT
70 NEXT
80 FOR K=0 TO 4
90 FOR J=0 TO 3
100 PRINT Nx(J,K),
110 NEXT
120 PRINT
130 NEXT
140 END

```

String arrays can have each element up to the maximum string length of 255 characters. The maximum number of DIMENSIONS allowed in an array is 255 also, and since you can have multi-dimensional arrays, the practical limit to the number of array elements you can play around with is determined by your Oric's memory. Remember that arrays eat memory quickly, and do not use unnecessary arrays.

**Related keywords:** None

## DOKE

**BASIC Token:** 138  
**Format:** DOKE addr., i

This command is used to store a double byte, or integer, value (in the range 0–65535) in memory. The first parameter is the address of the first byte of the two bytes in which we wish to store the value, and the second integer parameter in the format is the value to be stored. The format in which it is stored is the normal 6502 lo-byte, hi-byte representation (see machine code chapter). If we wished to store 770 in the memory we could use the command:

DOKE 30000,770

This would have the effect of storing 2 in location 30000 and 3 in location 30001 since 770 is  $3 \times 256 + 2$ .

**Related keywords:** CALL, DEEK, PEEK, POKE, USR

## DRAW

**BASIC Token:** 172  
**Format:** DRAW x, y, fb

This is one of the Oric's graphics commands which can only be used in the HIRES mode. DRAW is used in conjunction with CURMOV and CURSET, and DRAWS a solid line from the current cursor position to a point x points along the x-axis, and y points along the y-axis. Thus the x and y co-ordinates specified in a DRAW statement are not absolute (i.e. they do not represent the literal co-ordinates of the Oric's screen), but are relative to the current cursor position. Thus x must be in the range 0–199 and y in the range 0–239. Note that if your DRAW statement takes the line off the screen an error message will be produced. As usual, the fb code falls between 0 and 3 (see individual entry for CHAR).

Whilst DRAW normally produces a solid line on the screen, by coupling it

with PATTERN the line can be converted into dots and dashes of a specified density. (See PATTERN).

For a dramatic demonstration of DRAW in action, see the example program for HIRES.

```

1 REM *** DRAW ***
5 HIRES:PAPER6:INK1
10 FORA=10TO230STEP10
20 CURSETA,0,0
30 DRAW0,199,1
35 WAIT20:PING
40 NEXT
50 FORY=10TO190STEP10
60 CURSET0,Y,0
70 DRAW239,0,1
75 WAIT20:PING
80 NEXT

```

**Related keywords:** CIRCLE, CURMOV, CURSET, HIRES, PATTERN

## EDIT

**BASIC Token:** 129

**Format:** EDIT In

This instruction enables you to bring the line specified by In down below the current cursor position for editing. Whilst it is possible to edit a particular line on screen after you have LISTED it, since any insertions must be keyed-in on lines above or below the line which is being edited, it is considerably easier to be sure of what you are adding when the line is free of the rest of the program.

In either instance, the cursor movement keys are used to position the cursor at the beginning of the line to be altered and CTRL-A to copy those sections of line that you wish to preserve. The insertion of characters is fully explained in Chapter 2, but essentially it involves keying-in the required characters in the correct position on the line above or below the edited line, reversing the movement of the cursor to re-enter the line at the desired point, and then copying any other correct portions until all is finished, when RETURN is pressed.

**Related keywords:** LIST

**END**

**BASIC Token:** 128  
**Format:** END

This command can be used to stop a program if you want to avoid the "BREAK IN ln" message which the Oric gives when the STOP command is used. Whilst it is usually placed in the last statement of a program, there are a number of circumstances in which its use can allow a program to crash with grace when incorrect data is input. The example program demonstrates END being used for this purpose.

```
10 REM *** END ***
20 REPEAT
30 INPUT "NUMBER";NUM
40 IF NUM<=0 THEN END
50 PRINT "NATURAL LOG OF" ;NUM;"IS" ;LN(NU
M)
60 UNTIL FALSE
```

**Related keywords:** STOP

**EXP**

**BASIC Token:** 225  
**Format:** EXP(n)

This is another of the Oric's many mathematical functions. EXP(n) returns e (=2.7183) to the power of n. It is often used in conjunction with LN (log<sub>e</sub>), since EXP(LN(n)) gives the antilog value.

```
10 REM *** EXP ***
20 HIRES
30 CURSET 30,30,0
40 DRAW 0,150,1:DRAW 190,0,1
50 FOR N=0 TO 5 STEP .025
60 X=N*40+30
70 Y=180-EXP(N)
80 CURSET X,Y,1
90 NEXT
100 END
```

**Related keywords:** LN, LOG

**EXPLODE**

**BASIC Token:** 164  
**Format:** EXPLODE

This is one of a number of pre-defined sounds available on the Oric. They are primarily of value in arcade-type games, but can also be used as prompts in more serious programs. Creative use of delays (WAIT) can also be used to extend their potential. WAIT delays must be used if repeated EXPLODES are required.

```

10 REM *** EXPLODE ***
20 HIRES: PAPER 3 : INK 4
30 FOR B=0 TO 95 STEP 4
40 CURSET 120, 2+B, 0
50 GOSUB 200
60 CURSET 120, 190-B, 0
70 GOSUB 200
80 NEXT
90 EXPLODE
100 FOR K=1 TO 10
110 PAPER 4:INK 3
120 WAIT K
130 PAPER 3:INK 4
140 WAIT K
150 NEXT
160 WAIT 200
170 GOTO 20
180 END
200 FOR I=1 TO 3
210 CHAR 100,1,1:CURMOU 6,0,0
220 NEXT
230 RETURN

```

**Related keywords:** PING, SHOOT, ZAP

**FALSE**

**BASIC Token:** 240  
**Format:** FALSE

If you use your Oric to deal with data, there are certain circumstances under which it will take a decision as to whether something is TRUE or FALSE.

```

1 REM *** FALSE ***
10 FOR A=11 TO 20
20 IF A<10 THEN PRINT TRUE ELSE PRINT F
ALSE
30 NEXT
40 END

```

The result of the above program is unspectacular, but in the Oric's terms quite understandable. Since the 10 in line 20 can never be equal to A (as the loop starts at 11), the Oric simply prints out ten zeros. This is because 0 is the computer's representation of FALSE. Thus whilst it is possible to replace FALSE with zero in a program, in a complex jungle of code this function can serve to clarify a listing. Thus:

```

10 REM *** FALSE 2 ***
20 A=10:B=1
30 REPEAT
40 PRINT (A>B):B=B+1
50 UNTIL FALSE

```

will print out nine -1's (which is the Oric's representation of TRUE), before the statement in line 40 that A is greater than B becomes untrue when A=B (i.e. 10=10). From this point on the Oric will churn out an endless line of zeros. So whilst the FALSE in line 50 could be replaced by 0 without making any difference to the running of the program, it should be clear that the use of FALSE considerably clarifies what is actually going on in the program.

**Related keywords:** TRUE

## FILL

**BASIC Token:** 175  
**Format:** FILL i<sub>1</sub>, i<sub>2</sub>, i<sub>3</sub>

This is another of the Oric's graphics commands. It enables you to FILL specific locations on the HIRES screen with a particular value. The Oric's screen is composed of 200 lines with 40 locations per line. Following the above command format, FILL fills i<sub>1</sub> lines of i<sub>2</sub> bytes with the value of i<sub>3</sub> at the current cursor position. Thus i<sub>1</sub> is in the range 1 to 200, i<sub>2</sub> is in the range 1 to 40 and i<sub>3</sub> must be between 0 and 255.

FILL is usually used to colour different sections of the Oric's screen with specific colours. Its role in the suite of graphics commands available on the

Oric is fully explained in Chapter 7, whilst the example program under the individual entry for HIRES shows FILL in action.

```

1 REM *** FILL ***
5 PAPER0
10 HIRES:PRINTCHR$(17)
20 REPEAT
30 X=INT(RND(1)*231)
40 Y=INT(RND(1)*175+7)
50 A=INT(RND(1)*(230-X)/6+1)
60 D=INT(RND(1)*(182-Y)+8)
70 CURSETEX+6,Y-7,0:FILL D+7,A,16
80 CURSETEX,Y,0:FILL D,A,(17+RND(1)*7)
90 UNTIL FALSE

```

**Related keywords:** CURSET, CURMOV, HIRES

## FN

**BASIC Token:** 196  
**Format:** FN v(n)

FN followed by a variable name can only be used in a program if it has been preceded by DEF FN, since it refers to a function which has been defined earlier in the program. For a full description of user-defined functions refer to Chapter 4 and the individual entry for DEF. The following example shows FN in action, converting radians into degrees.

```

10 REM *** FN ***
20 DEF FNDEG(R)=R*180/PI
30 REPEAT
40 INPUT "RADIANs";A
50 PRINT "=";FNDEG(A);"DEGREES"
60 PRINT
70 UNTIL A=999
80 END

```

**Related keywords:** DEF

# FOR . . . TO . . . (STEP) NEXT

**BASIC Token:** 141 . . . 195 . . . (203)

...

144

**Format:** FOR v=n TO n (STEP n )  
NEXT v

Much of a computer's activity revolves around the repetition of simple tasks. In BASIC the construction of loops is one of the means by which such repetition can be effectively implemented. FOR . . . NEXT loops are the commonest implementations of such a construction.

A FOR . . . NEXT loop forces the computer to execute the statements contained within it a specified number of times. For example:

```
10 FOR A=1 TO 3
20 PRINT A
30 NEXT A
```

The above loop will PRINT out the numbers 1, 2, 3. Let's follow it through and see how it works. First time through the loop the computer takes the variable A and assigns it the value 1 which it then PRINTS in line 20. It then continues until it reaches the word NEXT, and then returns to line 10. This process is then repeated, with the value of A being incremented by one with each pass of the loop until A=3. At this point the looping is complete and the program moves on and executes the line following the NEXT statement.

In the above example, the variable is incremented in steps of 1, which will always be the case unless the program specifies otherwise. You can use the optional word STEP to alter the STEP size.

```
10 FOR X=5 TO 20 STEP 5
20 PRINT X
30 NEXT
```

The above loop would PRINT out 5, 10, 15, 20. You may have noticed that in line 30 the name of the variable has been omitted following NEXT. Oric BASIC permits such an omission, but for the sake of clarity it is usually wise to leave the variable name in, especially if your program contains loops within loops.

It is possible for a STEP size to be negative. Thus:

```
10 FOR A=20 TO 5 STEP -5
20 PRINT A
30 NEXT A
```

is an acceptable construction. The STEP may also be non-integer. We can use:

```
10 FOR F=1 TO 2 STEP .2
20 PRINT F
30 NEXT F
```

The start, finish and STEP values may all be non-integer, variables, or calculated expressions.

```
5 A=56 : C=PI/2
10 FOR X=2*4 TO A/3 STEP C
20 PRINT X
30 NEXT X
```

**Related keywords:** REPEAT, UNTIL

## FRE

**BASIC Token:** 218  
**Format:** FRE(0)  
**FRE””**

This instruction has two formats for two distinct functions. In the command formats above, the first example returns the number of memory bytes still available at any given moment in a program's development.

The second format example forces what is known as "garbage collection". This is simply a means of tidying up the storage of strings (starting from just below HIMEM and working down). This is useful if you are reassigning strings which quite conceivably will be shorter when they take their new value. However, whilst their value has been re-established, the amount of memory given over to the new and shorter string will remain the same (so the Oric is effectively storing redundant spaces). If the new strings are longer, they will have to be stored elsewhere, leaving the entire space originally allocated to the old value empty. By using FRE"” you effectively shunt the stored strings together and do away with the wasted space.

The example program below demonstrates this "garbage collection" at work:

```
10 REM *** FRE ***
20 PRINTFRE(0)
30 A$="A":B$="B"
40 REPEAT
50 A$=A$+A$:B$=B$+A$
60 UNTIL LEN(A$)=128
```

```

70 PRINTFRE(0)
80 PRINT"NOW WE TIDY UP"
90 A$="" :B$=""
100 PRINTFRE("")
```

**Related keywords:** None

## GET

**BASIC Token:** 190  
**Format:** GET v\$  
 GET v

'Press any key to continue' (or a variation on it), is one of the most common instructions facing users of interactive programs. Well, if the program is written in Oric BASIC it's more than likely that GET (followed by a string variable) is holding up the program until you are ready to go on. It has some similarities with INPUT. GET stops the program and will not allow the Oric to continue until a key is pressed. In the statement:

```
100 GET A$
```

the character of the first key pressed will now be stored in A\$. However, unlike INPUT, GET does not require RETURN, but automatically continues to the next line of the program as soon as a key is pressed. Thus GET A\$ will only allow the value of a single character to be stored in A\$. GET followed by a numeric variable name can be used to GET a single digit from the keyboard, but will give an error if a non-numeric key is pressed.

As this example program demonstrates, GET is useful in menu-driven programs which only require single key entry:

```

1 REM***GET***
5 HIRES :C=RND(1)*6+1 :INKC :PAPER0
10 FORA=10 TO 50 STEP 10
20 CURSET50+(A*2),96,3
30 CIRCLE10+A,2
40 NEXT
50 PRINT"PRESS ANY KEY TO RUN AGAIN"
60 GETA$
70 GOTO5
```

GET differs from its sister command, KEYS, in that whilst the former actually stops the program until a key is pressed, the latter scans for input but passes on to the next line whether or not a key is pressed.

**Related keywords:** INPUT, KEYS

**GOSUB**

**BASIC Token:** 155  
**Format:** GOSUB ln  
 GOSUB v

This is one of the most valuable commands in BASIC. GOSUB is similar to GOTO in as much as it forces the Oric to execute instructions from a specified line number instead of following the program's line numbers in sequence. However, unlike GOTO this instruction stores a return address in the computer's memory stack. This means that GOSUB ln will cause the program to jump to line number ln and execute all subsequent operations until a RETURN statement is encountered. At this point the program will RETURN to the statement following the line containing the original GOSUB. The line number in the GOSUB expression may also be a variable or the result of evaluating an expression. The parts of a program which are accessed (or 'called') by a GOSUB statement are known as subroutines.

The destination of RETURN can only be altered with the use of the POP command (see POP in this section).

An intelligent use of subroutines can greatly enhance the efficiency and clarity of a program.

```

1 REM***GOSUB***
10 CLS:C=0
20 PRINT"ENTER A NUMBER <22"
30 INPUTN
40 IFN>30THENgosub250
50 IFC=1THEN10
60 GOSUB100
70 PRINTF:GOTO310
100 REM***SUBROUTINE 1***
110 IFN<>1THEN140
120 F=1
130 GOTO180
140 N=N-1
150 GOSUB100
160 F=F*(N+1)
170 N=N+1
180 RETURN
190 REM***ENDSUB***
200 GOTO310
250 REM***SUBROUTINE 2***
260 CLS:C=1

```

```
270 PRINT"OBEDY INSTRUCTIONS! TRY AGAIN.
```

```
"  
280 WAIT300  
290 RETURN  
300 REM***ENDSUB***  
310 END
```

**Related keywords:** GOTO, ON, RETURN

## GOTO

**BASIC Token:** 151  
**Format:** GOTO ln  
 GOTO v

Normally the control sequence of a BASIC program is via the numbered statements – ie from the lowest to the highest. The use of GOTO interrupts this sequence and directs control to the line number specified.

It is traditional to stress that GOTO should be used with care, as it is a particularly powerful command. GOTO is more often than not used to ‘patch’ a poorly structured program. Under such circumstances programs become difficult to follow and errors tough to trace. In short, if you find that you are using a large number of GOTOS then your program is almost certainly badly conceived.

Oric BASIC allows the line number to which the GOTO directs control to be a variable. For example:

```
60 GOTO A  
70 GOTO A+B
```

GOTO can also be used as a direct command. For example, GOTO 90 executes the program from line 90, retaining any values previously assigned to variables. In this respect it differs from RUN 90 which clears the variables before execution.

```
10 REM***GOTO***  
20 CLS:GOTO 40  
30 END  
40 GOTO 1000  
50 PRINT"PROGRAM " ;  
60 GOTO90  
70 PRINT"A POOR " ;  
80 GOTO 200
```

```

90 PRINT"DEMONSTRATES ";
100 GOTO 70
200 PRINT"USE   ";
210 GOTO 1020
300 PRINT"GOTO!";
310 GOTO 30
1000 PRINT:PRINT:PRINT"THIS ";
1010 GOTO 50
1020 PRINT"OF ";
1030 GOTO 300

```

**Related keywords:** GOSUB, ON

## GRAB

**BASIC Token:** 159  
**Format:** GRAB

A large proportion of the Oric's memory area is reserved for the HIRES screen. If you are developing a lengthy BASIC program which does not require use of the HIRES, it is possible to regain the HIRES dedicated memory area with the use of GRAB, which is normally entered as a direct command. (GRAB can also be used in the body of a program when extra memory may be needed for the storage of arrays.)

Once this command has been activated you will be unable to utilise the Oric's HIRES screen until the section of memory in question (bytes #9800 to #B400 on the 48K Oric and #1800 to #3400 on the 16K machine) has been turned over to the HIRES mode once again. This can be achieved by activating RELEASE which is entered as a direct command.

```

1 REM***GRAB***
5 CLS
10 PRINT"How much memory?" : PRINTFRE(0
)
20 PRINT"Now we grab some more space"
30 GRAB
50 PRINTFRE(0)
60 PRINT "Now we'll get rid of it again
"
70 RELEASE
80 PRINTFRE(0)
90 PRINT"See ?"

```

**Related keywords:** HIRES, RELEASE

**HEX\$**

**BASIC Token:** 220  
**Format:** HEX\$(i)

The hexadecimal number system is one much favoured by computers. Unlike the decimal system with which the majority of humans are familiar which is based on 10's, the hexadecimal system operates with a base of 16 digits:

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

A–F in the hexadecimal system are equal to 10–15 in decimal. HEX\$(i) allows us to convert the decimal integer in brackets into its hexadecimal (hex for short) equivalent. Thus HEX\$(15) will return #F (the hash sign simply stands for hexadecimal). The decimal number in brackets can be any whole number between 0 (#0) and 65535 (#FFFF). The Oric will throw up a 'BAD SUBSCRIPT' message if you try to convert negative numbers or fractions.

Whilst it is unlikely that you will use this function unless you are fairly well acquainted with the hexadecimal system, it is worth understanding how to convert a hex number into its decimal equivalent. Since the decimal system uses the position of each digit to represent powers of 10 (eg.  $522 = (5 \cdot 10^2) + (2 \cdot 10^1) + (2 \cdot 1)$ ), you will not be surprised to learn that the hexadecimal system follows much the same principle except that the position of the digits in a hex number indicate powers of 16. Thus:

$$46E = (4 \cdot 16^2) + (6 \cdot 16^1) + (14 \cdot 1) = \text{decimal } 1134$$

**Related keywords:** STR\$, VAL

**HIMEM**

**BASIC Token:** 158  
**Format:** HIMEM addr

This command gives you the capacity to reserve a portion of your computer's memory to store machine code programs.

Normally, the Oric splits up the available memory space as efficiently as possible to store your programs, their variables and whatever memory space is required for graphics displays. Under these conditions the user has no control over the manner in which the machine divides up its memory. However, when using machine code subroutines the programmer must allocate a section of memory exclusively for the storage of the machine code so that it will be unaffected by the storage of the BASIC code determined by the computer.

Thus HIMEM sets the address of the highest memory location available to that portion of a program written in Oric BASIC. HIMEM should be set at the very beginning of a program, and the command should always be used with a great deal of care.

```

10 REM *** HIMEM ***
20 PRINT "HIMEM IS CURRENTLY";DEEK(#A6)
30 GRAB
40 PRINT "AFTER GRAB IT IS";DEEK(#A6)
50 RELEASE
60 PRINT "AFTER RELEASE IT'S";DEEK(#A6)
70 HIMEM 3000
80 PRINT "OR WE CAN SET IT TO ANY VALUE
I.E."
90 PRINT DEEK(#A6)

```

**Related keywords:** GRAB, RELEASE

## HIRES

**BASIC Token:** 162

**Format:** HIRES

This command converts the Oric's TEXT or LORES screens into the high resolution screen. It is this screen which most effectively utilises the Oric's sophisticated graphics facilities. When the command is activated the first twenty-four lines of the screen change into a black background of  $240 \times 200$  points. The bottom three lines remain in the normal TEXT mode which the computer uses for messages in the usual way.

Like all graphics screens, HIRES consumes a great deal of the Oric's memory, which can be reclaimed for a BASIC program with the use of the GRAB command. However, if this facility has been used you will be unable to operate in the HIRES mode until you use RELEASE to make the memory available for HIRES once again.

When you are developing a program which uses HIRES you will be unable to LIST or EDIT the program in the normal way since it will not print on the HIRES screen. Each time you RUN the program you will have to return to the TEXT screen before you can examine your listing.

The commands listed below up and including FILL will only operate when the Oric is in the HIRES mode, whilst the program which follows puts a number of them to good use:

```

10 HIRES
20 CURSET 14,10,3
30 DRAW 0,180,1:DRAW 220,0,1
40 CURMOU -210,-10,3:DRAW -19,19,1
50 FOR B=1 TO 6
60 READ H: GOSUB 100
70 NEXT B
80 GOSUB 400:END
90 REM DRAW BAR FOR CHART
100 X=(B*5+2)*6-5: CURSET X,190,3
110 DRAW 0,-H,1:DRAW 10,-10,1:DRAW 12,0
,1:DRAW -10,10,1
120 CURSET X+23,180-H,3:DRAW 0,H+10,1
140 CURSET X,190-H,3
150 FILL H,2,B+16
160 CURSETX+13,190-H,3:FILL H,1,16
200 RETURN
300 DATA 100,75,55,124,150,155
400 REM PRINT ON TEXT WITH CHAR
410 A$="SALES OF ORIC COMPANION"
420 FOR I= 1 TO LEN(A$)
430 CURSET I*6+40,10,3
440 CHAR ASC(MID$(A$,I)),0,1
450 NEXT
460 PATTERN 51:CURSET 40,20,3:DRAW 150,
0,1
470 RETURN

```

**Related keywords:** CHAR, CIRCLE, CURMOV, CURSET, DRAW, FILL, LORES, TEXT, PAPER, INK

## IF . . . THEN . . . (ELSE)

**BASIC Token:** 153, 201, 200

**Format:** IF c THEN statements ELSE statement

The IF . . . THEN . . . (ELSE) format is what is known as a decision structure. It is used to test conditions and control the Oric's subsequent actions. The

**ELSE** is enclosed in brackets above (although not when it is actually used in a program), because it is an optional element in this format. The "statements" can be any BASIC instruction sequence provided it will fit into the Oric's maximum line length. **IF** the condition c is **TRUE**, the statements following **THEN** are executed. Control then passes to the next line, unless **ELSE** has been used, in which case the statements following **ELSE** are executed.

This decision structure is one of the most powerful elements of BASIC and its use is most satisfactorily clarified by example.

```

1 REM***IF/THEN/ELSE
5 CLS
10 INPUT"ENTER YOUR NAME";N$
20 CLS
30 INPUT"ARE YOU MALE OR FEMALE(M/F)";S
$
40 CLS
50 PRINT"HELLO ";
60 IFS$="M"THEN PRINT "MR ";N$ELSE PRIN
T"MRS ";N$
70 END

```

**GOSUB** must be used in standard format with an **IF . . . THEN . . . ELSE** construct. **GOTO** may be omitted, and just the line number given, and **GOTO** may replace **THEN**.

**Related keywords:** AND, GOTO, NOT, OR, ON

## INK

**BASIC Token:** 178  
**Format:** INK i

This instruction operates in both the high and low resolution modes. It sets the foreground colour of the entire screen according to the value of i, whose possible values are listed in the colour table below:

- 0 BLACK
- 1 RED
- 2 GREEN
- 3 YELLOW
- 4 BLUE
- 5 MAGENTA
- 6 CYAN
- 7 WHITE

**INK** changes the colour of everything ‘written’ on the screen and cannot be used to print individual characters in different colours (see the graphics chapter and the individual entry for **CHR\$** for an explanation of how this can be done).

```

1 REM***INK***
10 HIRES
20 A=0:B=0
30 FORU=0TO25
40 B=B+1 :A=A+3
50 IFB>7THENB=0
60 INKB
70 CURSET110,100,3
80 CIRCLEA,1
90 NEXTU
100 GOTO20

```

**Related keywords:** LORES, HIRES, PAPER, TEXT, CHR\$

## INPUT

**BASIC Token:** 146

**Format:** INPUT v, v\$, . . .

INPUT “prompt”, v, v\$, . . .

This allows the computer to receive information from the outside world. The command stops the execution of a program and the Oric will not continue until the user has INPUT a word/letter or number. There are a number of ways in which the command can be formatted. For example:

```
10 INPUT N$
```

will stop the program after placing a question mark prompt on the screen. The user must now key-in the appropriate INPUT and press the RETURN key. However, it is unlikely that anyone other than the programmer will understand exactly what form of data is required by the program, so obviously some sort of explanatory message is required. This can be achieved in one of two ways:

```
10 PRINT @ 13,10; “WHAT IS YOUR NAME”
20 INPUT N$
```

or else:

```
10 INPUT “WHAT IS YOUR NAME”;N$
```

The only advantage of the first construction is that it enables you to position the message anywhere on the screen (provided you own an Oric V1.1), whereas the second example PRINTS the prompt AT the current cursor position.

**Related keywords:** GET, KEYS

## INT

**BASIC Token:** 215  
**Format:** INT(n)

The purpose of INT is to convert any number which has a decimal part into a whole number. It should be noted that the number which is given by the use of INT is always less than the actual value supplied. For example:

```
10 Z=INT(47.677)
20 PRINT Z
```

will PRINT out 47. However, you need to be careful when dealing with negative values. For example:

```
10 X=INT(-10.56)
20 PRINT X
```

will, of course, PRINT -11.

**Related keywords:** None

## KEY\$

**BASIC Token:** 241  
**Format:** v\$=KEY\$

KEY\$ is one of the means by which the Oric receives information from the outside world. Like GET it seeks a keyboard response, but will allow the program to continue whether or not a key is pressed.

Since KEY\$ contains the value of whatever key is being pressed, it is valuable in arcade-type games (e.g. allowing the cursor keys to be used to control 'movement' on the screen). As the example below demonstrates, it is also useful when a particular response is required from the user of the program:

```
1 REM***KEY$***
5 CLS :X=2
10 PRINT"USE 'Z' TO MOVE LEFT, 'M' FOR RIGHT & 'S' TO STOP"
```

```

20 REPEAT
30 U$=KEY$
40 IFU$="Z"THENX=X-3:PLOTX+3,10,""
50 IFU$="M"THENX=X+3:PLOTX-3,10,""
60 IFX<2THENX=2
70 IFX>35THENX=35
80 PLOTX,10,"<*>""
90 UNTILU$="S"
100 PRINT"EXAMPLE TERMINATED"

```

**Related keywords:** GET, INPUT

## LEFT\$

**BASIC Token:** 244  
**Format:** v\$=LEFT\$(a\$, i)

Like MID\$ and RIGHTS\$ this command allows specific characters to be extracted from a string. It takes the following format:

LEFT\$(a\$,i)

in which a\$ is any previously defined string variable and i is equal to the number of consecutive characters that you want extracted from the lefthand side of that string. Thus:

```

10 REM *** LEFT$ ***
20 A$="ORIC LEFT$"
30 FOR I=1 TO LEN(A$)
40 B$=LEFT$(A$, I)
50 PRINTB$
60 NEXT I
70 END

```

will extract and PRINT all the possible LEFT\$ from "ORIC LEFT\$". If it is in excess of the number of characters in the string the Oric will simply return the entire string.

**Related keywords:** RIGHT\$, MID\$, LEN

**LEN**

**BASIC Token:** 233  
**Format:** LEN(a\$)

LEN simply returns the number of characters in a particular string. For example:

```
10 J=LEN ("NUMBER")
20 PRINT J
```

would PRINT out 6, because there are six letters in the string 'NUMBER'.

```
10 H=LEN ("PRINT OUT")
20 PRINT H
```

would PRINT out 9, because this time the string is made up of eight letters and a space.

This function is useful when, for example, you wish to format a screen display which is to contain strings INPUT by the user of the program whose length is, of course, unknown. LEN can also be used in a FOR . . . NEXT construction, when a task must be performed for each letter of a string. Thus:

```
10 FOR A=1 TO LEN(X$)
```

is an acceptable construction. The empty or null string, "", has a LENGTH of 0.

**Related keywords:** MID\$, RIGHT\$

**LET**

**BASIC Token:** 150  
**Format:** LET: v=n  
 LET v\$=a\$

Throughout this book we have tried to stress that it is often not enough to produce programs that RUN, but that they should also be readable to an outsider. In Oric BASIC, LET is an optional function, but its use can serve to clarify a listing.

LET is used in the assignment of variables. For example:

```
10 LET A=10
```

This means that in a given memory location which shall henceforth be known as A, the value 10 will be held. What we have created is a curious beast known as a constant variable – a variable whose value remains the

same. This said, we can alter the value of variable A with a construction like this:

**20 LET A=A+10**

This may seem to be nonsense in mathematical terms, but as far as BASIC is concerned we are simply saying that to the memory location A which initially held the value of 10 we must now add a further 10. Thus A now holds the value of 20 ( $10 + 10$ ). Oric BASIC, however, does allow us to omit LET from a statement:

**30 A=A+10**

but, to labour the point, its inclusion does make for a clear and readable listing.

**Related keywords:** None

## LIST

**BASIC Token:** 188  
**Format:** LIST i-i

This command is important for every stage of a program's development, since it enables the programmer to LIST a line, a series of lines or the entire program. Let's look at the various formats:

**LIST**

On its own, LIST will PRINT out the entire program on to the screen. Since any reasonable sized program will be considerably more than a single screenful of text, you will want to stop each section of program as it scrolls up the screen. To do this you simply use the space bar at the bottom of the keyboard. This will stop the scrolling which can be reactivated by pressing the space bar again once you have examined the relevant section.

**LIST 40**

This will PRINT line 40 on to the screen.

**LIST 40 - 80**

This will PRINT lines 40-80 (inclusive) on to the screen.

**Related keywords:** EDIT, LLIST

**LLIST**

**BASIC Token:** 142  
**Format:** LLIST i-i

This behaves in exactly the same way as the LIST command, except that instead of PRINTing the specified lines on to the screen it lists them on to the printer.

**Related keywords:** LPRINT, LIST

**LN**

**BASIC Token:** 224  
**Format:** LN(n)

This is one of the Oric's valuable mathematical functions. It returns what are known as natural logarithms or, more properly, logarithms to the base e.

LN(n) is the natural logarithm of n. The antilog is EXP(LN(n)). Natural log. operations can be used, if appropriate, as with common logs. For example:

EXP(LN(x)+LN(y))

gives the product of x and y.

**Related keywords:** EXP, LOG

**LOG**

**BASIC Token:** 232  
**Format:** LOG(n)

This is a mathematical function which calculates the common logarithm to the base 10. The n in the format must be greater than 0 otherwise LOG(n) would be an imaginary number. See any calculus text for details of such numbers, if you're interested, but for now just note that the Oric can't handle them.

```

1 REM***LOG***
5 CLS
10 INPUT"ENTER A NUMBER (1-10)";N
20 FORA=1TO10
30 C=RND(1)*9
40 PRINT"THE LOG OF ";N*A
50 PRINT" IS ";LOG(N*A)
60 NEXT

```

**Related keywords:** EXP, LN

# LORES

**BASIC Token:** 137  
**Format:** LORES 0  
 LORES 1

This command allows access to two of the Oric's four screens. It has two formats (LORES 0 and LORES 1), one for each screen, although both of the low-resolution screens have exactly the same format: twenty-seven lines of forty characters.

LORES 0 is similar to the TEXT screen except that it generates a black screen upon which white characters can be printed. The normal use of a PRINT statement will cause the entire screen area to scroll into the TEXT screen. This means that characters must be positioned by PRINT@ or PLOT.

LORES 1 operates exactly the same way except that it prints the alternate character set to the screen. The example program demonstrates the use of both screens.

```

1 REM***LORES***
10 LORES0:C=0
20 PLOT2,24,"THE CHARACTER SET IN LORES
0"
30 FORA=32TO126
40 X=RND(1)*36+1:Y=RND(1)*22+1
50 C$=CHR$(A)
60 PLOTX,Y,C$
70 WAIT50
80 NEXT
90 IFC=1THENWAIT500:CLS:END
100 PRINT: PRINT"AND NOW THE SET IN LOR
ES 1  ":WAIT300
110 GOSUB130
120 GOTO30
130 CLS:LORES1:C=1
140 RETURN

```

**Related keywords:** TEXT, HIRES

**LPRINT****BASIC Token:** 143**Format:** LPRINT a\$

LPRINT i

This command works exactly the same as a PRINT statement except that it is used to send the print items to a printer. However, it is not possible to use an LPRINT @ statement. Used with the control codes listed in Appendix 1, LPRINT can be used to regulate the output to a variety of printers. It is possible to alter the line length of the printer output by using POKE #256,i where i is the required line length.

**Related keywords:** CHR\$, LLIST, PRINT

**MID\$****BASIC Token:** 246**Format:** MID\$(a\$, i<sub>1</sub>, i<sub>2</sub>)

Like RIGHT\$ and LEFT\$, this function is used to extract specified characters from a predefined string. The name of the command is actually somewhat misleading, since it enables programmers to extract characters from any part of the string (not just the middle). The extracted characters constitute what is known as a substring, which starts at character i<sub>1</sub> and is i<sub>2</sub> characters in length. If i<sub>2</sub> characters starting at i<sub>1</sub> is greater than the length of the string, the program will simply print out the entire string from the point specified by i<sub>1</sub>. If no such point as i<sub>1</sub> exists within a string, then nothing is returned. Without i<sub>2</sub> the command will simply return the rest of the string from the point specified by i<sub>1</sub>.

In the example program below, MID\$ is used to get rid of the leading spaces in a numerical print out.

```

10 REM *** MID$ ***
20 A$=" ORIC "
30 IF ASC(A$)=32 THEN A$=MID$(A$,2): GO TO 30
40 IF ASC(RIGHT$(A$,1))=32 THEN A$=MID$(A$,1,LEN(A$)-1): GOTO 40
50 PRINT "*" ;A$;"*"

```

**Related keywords:** LEFT\$, RIGHTS\$

## MUSIC

**BASIC Token:** 168  
**Format:** MUSIC c, o, n, v  
**c=channel 0 - 3**  
**o=octave 0 - 7**  
**n=note 1-12**  
**v=volume 0 - 15**

This is one of the Oric's three major sound commands, and as its name implies it is used to generate musical output. The utilisation of the Oric's sound commands is relatively complex, and thus it is important that if you intend to take full advantage of these sophisticated facilities you should refer to the sound chapter. However it is possible here to outline the parameters of the command.

There are four variables involved in a MUSIC statement and the command itself is usually coupled with PLAY (which creates the "shape" of the sound and determines the number of sound channels in operation at any one time). As laid out in the command format above, c sets the tone channel to be used, whilst o determines which of the eight available octaves is to be used (0-7). The third variable, n, determines which of the twelve notes in a given octave will be generated (1-12). The last parameter, v, determines the volume of the sound produced (0-15).

Unless you have a good reason for wanting the last note of your musical masterpiece to drone on for eternity, you must close down all the tone channels by using the statement PLAY 0,0,0,0 to silence the computer.

```

1 REM***MUSIC***
10 FOR0=0TO6
20 FORN=1TO12
30 MUSIC1,0,N,10
40 PLAY3,0,7,0
50 WAIT30

```

```

60 NEXTN
70 NEXTO
80 PLAY0,0,0,0
90 EXPLODE

```

**Related keywords:** PLAY, SOUND

## NEW

**BASIC Token:** 193  
**Format:** NEW

This command deletes the current BASIC program and variables from the Oric's memory. It is advisable to enter NEW before starting any new program to ensure that it is unaffected by any superfluous instructions from a previous program.

**Related keywords:** RUN

## NOT

**BASIC Token:** 202  
**Format:** NOT i  
 NOT c

The keyword NOT is a logical condition operator which works in much the same way as the common English usage of the word. It reverses the value of TRUE (-1) or FALSE (0) which the Oric assigns to a logical expression. For example it can be used as part of an IF... THEN statement:

IF NOT (condition) THEN (action)

It is easy to see how this sort of construction could play a valuable role in a games program. For example:

IF NOT DEAD THEN GOTO (next stage of game)

where DEAD is a flag which determines whether or NOT the program should enter the endgame sequence.

The condition can also take the format NOT i or NOT c in which i is an integer and c is a logical expression. Thus it can take its place in the following type of constructions:

IF NOT X THEN X=6

If X was 0 (FALSE) the condition NOT X is TRUE, and X will be set to 6. NOT 7 would give the answer 248 since, for the purposes of operation on integers

we would consider the individual bits. If a bit was set in the original integer then it will NOT be set in the answer and vice versa. Thus 7 (00000111) will become 248 (11111000).

```

1 REM***NOT***
5 CLS :A=1
10 INPUT"THINK OF AN INTEGER";N
20 REPEAT
25 A=A+1
30 IFNOT(A=N)THENPRINTA
40 WAIT20:CLS
50 UNTIL A=N
60 PRINTA:EXPLODE

```

**Related keywords:** AND, IF, OR

## ON

**BASIC Token:** 180  
**Format:** ON i GOTO ln<sub>1</sub>, ln<sub>2</sub> . . .  
                   ON i GOSUB ln<sub>1</sub>, ln<sub>2</sub> . . .

This command must be paired with either GOTO or GOSUB and facilitates multiple branching in a program. It is commonly used as a control structure in menu-driven programs in which the user is given a number of options whose consequences are handled by different parts of the program. For example, consider the program lines

```

1 REM***ON***
5 CLS
10 INPUT"PLEASE ENTER 1,2 OR 3";N
20 ON N GOTO 100,200,300
50 REM
100 PRINT"LINE 100 FROM N=1":GOTO10
150 REM
200 PRINT"LINE 200 FROM N=2":GOTO10
250 REM
300 PRINT"LINE 300 FROM N=3":GOTO10

```

If the user enters 3 in line 10, the program will GOTO the third line number (300) specified in line 20. If N=2 then the program will jump to the second line number, and so on.

If N is greater than the number of line numbers specified then the program will simply continue on to the statement which follows the one containing the ON...GOTO statement. However, negative INPUT will throw up an error report. Either way, INPUT checks are clearly required. Non-integer values are automatically rounded.

ON...GOSUB works in exactly the same way, except that since the program is jumping to a subroutine a RETURN statement is required, at which the program will jump back to the line following the ON...GOSUB statement.

**Related keywords:** GOSUB, GOTO, RETURN

## OR

**BASIC Token:** 210  
**Format:** i OR i  
 c OR c

This is another of the Oric's logical condition operators. In order to understand how this condition operates, it is worth examining the consequences of this keyword in a tabular form:

TRUE OR TRUE=TRUE
TRUE OR FALSE=TRUE
FALSE OR TRUE=TRUE
FALSE OR FALSE=FALSE

It should be clear that the OR operator has a similar effect to the 'either...or...' construction in English whereby an answer of TRUE (-1) is returned if either of the two conditions in question is correct.

OR can also be used as a "bit-wise" operator on two integers in a similar manner to AND, with each pair of corresponding bits being considered jointly. Thus 13 (00001101) OR 24 (00010100) is 29 (00011101).

Test this with the following command:

PRINT 13 OR 24

which should return 29.

When used as part of an IF...THEN construction, OR has the following type of format:

IF (A=0) OR (B=1) THEN 300

If either condition is TRUE, the program will jump to line 300.

```

1 REM***ORIC***
5 CLS
10 INPUT"ENTER AGE OF HUSBAND";HA
20 INPUT"ENTER AGE OF WIFE";WA
30 INPUT"ENTER ANNUAL INCOME OF HUSBAND
";IH
40 INPUT"ENTER ANNUAL INCOME OF WIFE";I
W
50 IF(HA>21 AND IH>=5000) OR (WA>21 AND
IW>=5000)THEN100
60 PRINT"NOT ELIGIBLE FOR LOAN"
70 END
100 PRINT"LOAN AVAILABLE"

```

**Related keywords:** AND, NOT, FALSE, TRUE

## PAPER

**BASIC Token:** 177  
**Format:** PAPER i

This sets the background colour of the Oric's entire screen. It operates in both high and low resolution modes, but cannot be used to set the background of individual sections of screen. PAPER must be coupled with i, giving one of the colour codes:

- 0 BLACK
- 1 RED
- 2 GREEN
- 3 YELLOW
- 4 BLUE
- 5 MAGENTA
- 6 CYAN
- 7 WHITE

To set the background colour of individual sections of screen see the individual entries for CHR\$ and FILL and the graphics chapter.

```

1 REM***PAPER***
10 HIRES:INK0
20 A=0:B=0
30 FORU=0TO25
40 A=A+1:B=B+3

```

```

50 IFA>7THEN A=0
60 PAPER A:WAIT 20
70 CURSET 110, 100, 3
80 CIRCLE B, 1
90 NEXT U
100 GOT O20

```

**Related keywords:** INK

## PATTERN

**BASIC Token:** 174

**Format:** PATTERN i

This is another of the Oric's HIRES graphics commands. Since it is used in conjunction with either DRAW or CIRCLE it can only be used in the high resolution mode.

Normally both DRAW and CIRCLE create a solid line display. However, PATTERN can be introduced to break the lines created by these commands into dots or dashes, the pattern of which is determined by the integer i following PATTERN, in the range 0-255. The example program below demonstrates the full range of PATTERN's influence:

```

10 REM***PATTERN***
20 HIRES : INK 0 : PAPER 1
30 FOR A=1 TO 65
40 PATTERN 170-A
50 DRAW 230, 0, 1
60 CURMOV -230, 3, 0
70 NEXT

```

**Related keywords:** HIRES, DRAW, CIRCLE

## PEEK

**BASIC Token:** 230

**Format:** PEEK (addr.)

PEEK examines the memory location specified by addr. and returns the value contained in that location. The value returned will be between 0 and 255. For example, PEEK (#256) will return the current printer line length on your Oric (80). The value of this function becomes clear when it is realised that by POKEing this location (see individual entry for POKE), with a value

other than 80 will alter the line length. For a fuller explanation of the PEEK function see Chapter 5.

```

1 REM***PEEK***
10 REPEAT
20 PRINTCHR$(20)
30 GOSUB100
40 UNTILFALSE
50 END
100 IF PEEK(48039)=83THENPRINT"CAPS ON"
ELSEPRINT"CAPS OFF"
110 RETURN

```

**Related keywords:** CALL, DEEK, DOKE, POKE, USR

## PI

**BASIC Token:** 238  
**Format:** PI

The trigonometric constant. Returns the value of 3.14159265. In the short example program below PI is used in the calculation of the area of a circle.

```

1 REM***PI***
5 CLS
10 FORU=1TO5
20 R=RND(1)*30
30 A=PI*(R^2)
40 PRINT"IF CIRCLE RADIUS="R;
50 PRINT"THEN ITS AREA IS"A
60 PRINT:PRINT
70 NEXT

```

## PING

**BASIC Token:** 166  
**Format:** PING

This is another of the Oric's pre-defined sounds which can be used as a prompt or as a feature in games programs. Multiple PINGS can only be used effectively when coupled with the WAIT command. It can be sampled by

using CTRL-G (or by attempting to key more than eighty characters into a single statement).

```

1 REM***PING***
100 PAPER1:INK0:CLS:PRINT
110 PRINTCHR$(140)"CHOOSE ONE OF THE FO
LLOWING:"
120 PRINT:PRINT:PRINT
130 PRINTCHR$(147)"1 DIARY":PING
140 WAIT30
150 PRINTCHR$(148)"2 HOROSCOPE":PING
155 WAIT50
160 PRINTCHR$(146)"3 CALENDAR":PING

```

**Related keywords:** EXPLODE, SHOOT, ZAP

## PLAY

**BASIC Token:** 169

**Format:** PLAY t, s, e, d

t=tone channel 0-7

s=sound channel 0-7

e=envelope 0-7

d=duration 0-32767

This is one of the Oric's rather complex sound commands which, once you get used to them, offer a music/sound potential well in advance of the machine's competitors. However, the commands are a little difficult to grasp, particularly if you're not musically minded, so it is well worth spending some time working your way through Chapter 7. For the present we will restrict ourselves to clarifying the format of the command.

Your Oric is endowed with three sound/tone channels and PLAY is the command which determines the combination of these channels. In terms of the command format at the top of the page, t (tone) and s (sound) determine which channels are activated (0-7). The effects of the combination of channels is really only comprehensible after a little experimentation, but the following chart will be of some use for future reference. The column on the left represents the value of t or s, whilst the righthand column tells you which combination of channels are activated by this value.

t or s	channel combination
0	no channels
1	1
2	2
3	1 and 2
4	3
5	1 and 3
6	2 and 3
7	1, 2 and 3

PLAY's third parameter, e, is probably the most difficult to understand, since it is the integer which determines the "shape" of the sound which the Oric produces. Once again the sound chapter should be consulted for a full explanation of this feature. However, variable e (0-7) when set at 1 or 2 produces what are known as sound envelopes of a fixed length, whilst all other settings generate continuous sounds of various types. The example program below will hopefully clarify the effect of the different PLAY settings. Finally d (0-32767) sets the duration of the sound envelope.

When using either MUSIC or SOUND at some point in the course of your program you will want to turn off the sound channels, which can be achieved by the statement PLAY 0,0,0,0.

```

1 REM***PLAY***
5 CLS
10 CLS:PRINT:PRINT
20 INPUT"ENTER NUMBER (0-7) FOR SOUND S
HAPE";E
30 IFE<0 OR E>7THEN20
40 INPUT"ENTER A NUMBER (0-65535) FOR D
URATION";D
50 PRINT"THIS IS SOUND ENVELOPE ";E;" W
ITH A ";D;" DURATION"
60 SOUND1,1500,0
70 PLAY1,0,E,D
80 PRINT"PRESS ANY KEY TO STOP PLAY"
90 GETA$:GOTO5

```

**Related keywords:** MUSIC, SOUND

**PLOT****BASIC Token:** 135**Format:** PLOT x, y, n

PLOT x, y, a\$

This command is used to position characters on the low resolution screens.

In the first of the command formats given above, n must be a numeric expression that will return the appropriate ASCII character represented by that expression. Since this includes not just the standard printed character set but also the attribute characters (see Chapter 7 and Appendix 1), PLOT can be used in the production of interesting screen displays. For example:

PLOT 11, 11, 12: PLOT 13, 11, "ORIC"

will produce a flashing "ORIC" on your screen. Experimentation should reveal the potential of the command used in this way, and the graphics chapter offers a full explanation of how the attribute characters can most effectively be exploited.

If PLOT is used in the second format, it will position the string expression on the screen. For both numeric and string expressions the co-ordinates are in the ranges 0-39 (x) and 0-26 (y). With x=0 and y=0, the PLOT position is the top lefthand character position of the screen. The command will operate in the TEXT, LORES 0 and LORES 1 modes, but will not function on the HIRES screen.

```

1 REM***PLOT***
5 CLS:LORES0
10 PLOT2,2,"PLOTTING IN LORES 0 MODE"
20 FORA=1TO23
30 X=RND(1)*33+4:Y=RND(1)*22+3
40 PLOTX,Y,A:PLOTX+2,Y,"ORIC"
50 WAIT50:NEXT

```

**Related keywords:** CHAR, PRINT

**POINT****BASIC Token:** 243**Format:** POINT (x,y)

This command is used to test whether the point on the HIRES screen specified by the co-ordinates which follow it (x in the range 0 to 239, y between 0 and 199) holds a background or foreground colour. If the point contains a background colour, POINT returns 0, and if it contains a foreground colour POINT returns -1.

POINT is often used in games programs to detect collisions, and the example program below offers a simple demonstration of the command in this role.

```

1 REM***POINT***
5 HIRES:PAPER4:INK0
10 FORB=5TO175
20 CURSET200,B,2:CHAR124,0,2
30 NEXTB
40 A=5
50 REPEAT
60 A=A+8
70 CURSETA,90,0
80 CHAR127,0,1:WAITS
90 CHAR127,0,0
100 C=POINT(A+11,90)
120 UNTILC=-1
130 EXPLODE

```

**Related keywords:** HIRES

**POKE****BASIC Token:** 185**Format:** POKE addr, i

This instruction allows programmers to place the value of i into the memory location represented by the address addr. The address must fall into the range 0-65535, and since any given location is made up of eight binary digits the value of i falls into the range 0-255. Both addr. and the value of i may be specified in either decimal or hexadecimal notation.

POKE enables you to enter machine-code routines and tinker around with

your Oric's system variables, and, although the effects of an aimless POKE may initially appear somewhat traumatic, as far as it is possible to tell no permanent harm can be done to your machine.

As an example of the power of this command, try the following program to alter the status line display.

```
10 REM *** POKE ***
20 FOR A=1 TO 8
30 POKE 46599+A,A
40 WAIT 100
50 NEXT
60 WAIT 1000
70 CALL DEEK(#FFFA)
```

Or this one to display an "A" in the (normally inaccessible) top left-hand corner of the screen:

**POKE 48000,65**

**Related keywords:** DEEK, DOKE, PEEK

## **POP**

**BASIC Token:** 134  
**Format:** POP

This instruction operates in exactly the same way for GOSUB . . . RETURN as PULL does for a REPEAT . . . UNTIL loop. In many respects POP should be considered as a last option command, since it forces a program to jump out of a nested subroutine, which should never be necessary in a properly planned and structured program.

Normally when a subroutine is called the RETURN address is stored in the area of memory known as the stack. With each new subroutine a further address is added to the stack and as each subroutine encounters a RETURN it retrieves its return address on a 'last in first out' basis. POP is the only legitimate means of circumventing this operation.

Imagine a program which calls subroutine 1 which in turn calls subroutine 2. When a specific set of conditions have been met in subroutine 2 under certain circumstances it may be necessary to jump back to the main body of the program rather than return to subroutine 1. POP enables you to do this, since it bypasses the first return address on the Oric's stack and in this case retrieves the return address of subroutine 1, which will return control to the main body of the program. The example program below demonstrates such a procedure in action.

```

10 REM *** POP ***
20 FOR I=-5 TO 5:PRINT I,
30 GOSUB 100
40 PRINT:NEXT
50 END
100 GOSUB 200
110 PRINT X
120 RETURN
200 IF I>0 THEN X=LOG(I) ELSE POP
210 RETURN

```

**POS**

**BASIC Token:** 219  
**Format:** POS(0)  
 POS(1)

This instruction returns the current horizontal cursor position when used to detect such a position generated by a straightforward PRINT statement. It cannot be used to return positions generated by PLOT or PRINT@. POS (0) returns the cursor position on the screen, POS (1) returns the horizontal position of the printhead when LPRINTing.

```

1 REM***POS***
10 DIM A(4,2)
15 REPEAT
20 FOR K=1 TO 4
30 FOR J= 1TO 2
40 A(K,J)=K*RND(1)+J
50 NEXT
60 NEXT :CLS
70 FOR K=1 TO 3:PRINT :NEXT
80 PRINT A(1,1);
90 REPEAT:PRINT" ";:UNTIL POS(0)=15
100 PRINTA(1,2)

```

**Related keywords:** PRINT

**PRINT**

**BASIC Token:** 186  
**Format:** PRINT plist  
 PRINT@ x,y; plist  
 plist=list of print items

PRINT instructions affect the current cursor position on screen, which defines the point from which any printing will start. PRINT (which may be abbreviated to ?) is followed by a list of items to be displayed on screen. PRINT@ x,y; defines a print co-ordinate position of the TEXT and LORES screens.

PRINT may be used on its own, with no PRINT items following, and a blank line is displayed on screen. The PRINT list can consist of numbers, numeric variables, expressions, strings and string expressions which may follow directly after each other, or be separated by commas and semi-colons. The PRINT functions TAB and SPC can also be included (see their individual entries in this section).

Semi-colons after PRINT items leave the PRINT position set directly after the last character printed, so that the next item to be printed will follow on. If the semi-colon is at the end of a PRINT list, any subsequent PRINT statement will continue PRINTing on the same line. When there is no ambiguity in the sequence of PRINT items the semi-colon may be omitted, and each item will be placed on screen directly after the preceding (but remember that numbers have leading and trailing spaces). Thus it is possible to have a line PRINT AX"TIMES"23, with no semi-colons between the items, as the Oric will interpret the items correctly, whilst we cannot say PRINT AX12, with the intention of displaying the value stored in variable AX, then 1, then 2, since the Oric will interpret it as an instruction to PRINT the value of a variable called AX12. Since the effect is the same, it is advisable to use a semi-colon unless you are certain the Oric will not misinterpret the sequence you place in the line.

A comma between PRINT items causes the cursor position to be moved to the start of the next PRINT field. The Oric screen is divided up into five fields, each eight PRINT positions wide, and a comma moves the cursor at least one space to the right, and then to the beginning of the next field, thus ensuring at least one space between each item formatted this way. Multiple commas may be used to shift over more than one field to the right.

Control characters and attributes may be placed in PRINT statements using CHR\$(i), where i is an ASCII character code. The same instruction may be used to place other characters on the TEXT and LORES screen, but control

codes and attributes are non-printing, that is they do not PRINT anything on the screen, but occupy a character position in the screen memory map, so that a space appears on screen.

Many examples of the use of PRINT statements appear throughout this handbook. The program below demonstrates a variety of PRINT formats.

```

10 REM *** PRINT ***
20 B=2
30 A$="TO BE OR NOT TO BE"
40 PRINT 1,2,3
50 PRINT A$
60 PRINT 2*B OR NOT 2*B
70 PRINT
80 PRINT 1;2;3

```

PRINT@ instructions have the form PRINT@ x,y, where x is the column number (0 to 39) across the screen left to right, and y is the row number (0 to 27) of PRINT lines down the screen. The two values are separated by a comma, and must be followed by a semi-colon before the list of items to be PRINTED, which follows the same formats as for PRINT. Variables or calculated expressions may be used for x and y, and they will be rounded down if they are non-integer. An ILLEGAL QUANTITY error will occur if the values are outside the correct range.

The example program draws a circle, using SIN and COS to calculate the x and y positions for PRINT@.

```

10 REM *** PRINT @ ***
20 R=13:XC=20:YC=13:CLS
30 FOR A=0 TO 2*PI STEP PI/50
40 X=XC+COS(A)*R
50 Y=YC+SIN(A)*R
60 PRINT @X,Y;"O"
70 NEXT

```

**Related keywords:** LPRINT, SPC, TAB

**PULL**      **BASIC Token:** 136  
**Format:** PULL

This instruction operates in exactly the same way for a REPEAT . . . UNTIL loop as POP does for GOSUB . . . RETURN. If the truth be told, both POP and

PULL must be regarded as "patching" commands. If your program has been properly planned and structured, it should never be necessary to jump out of either a loop or a subroutine. However, no one is perfect, and PULL does give you the chance to extricate yourself from a tricky situation.

Let's assume you have programmed a REPEAT loop to print out the numbers between -5 and 10, along with a countdown of each number to zero (see example program below). We use PULL to exit the inner loop in the case of negative numbers since decrementing will obviously never make them zero. (Of course this is a somewhat artificial example, since it would obviously be simpler to use the IF statement to avoid entering the REPEAT loop.)

```
10 REM *** PULL ***
20 A=9
30 REPEAT
40 : B=A
50 : REPEAT
60 : PRINT B;
70 : B=B-1
80 : IF B<0 THEN PULL:GOTO 100
90 : UNTIL B=0
100 : A=A-1
110 : PRINT
120 UNTIL A=-5
```

```

9 8 7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
7 6 5 4 3 2 1
6 5 4 3 2 1
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
0
-1
-2
-3
-4

```

**Related keywords:** POP, REPEAT, UNTIL

## READ

**BASIC Token:** 149

**Format:** READ v,v,...  
READ v\$,v\$,...

This is always used with DATA statements, and it is important to ensure that READ is followed by an appropriate variable (v in the case of numeric data and v\$ when READING string DATA). The command READS the DATA statements sequentially, and must never be programmed to READ more DATA than is actually contained within the statements (this will result in an OUT OF DATA error message). Thus the positioning of a READ statement is critical, whilst DATA statements can be placed at any point in a program.

When DATA statements have been exhausted, the computer's internal "pointer" can be returned to the beginning of the DATA line with the use of the RESTORE command.

```

1 REM***READ***
5 CLS
10 FORA=1 TO5
20 READU,U$
30 PLOTU,10,U$
40 NEXT
50 DATA5,MAN,9,WOMAN,15,CHILD,21,DOG,25
,HOUSE

```

**Related keywords:** DATA, READ, RESTORE

**RECALL****BASIC Token:** 131**Format:** RECALL v , "filename" (,S)  
RECALL v\$, "filename" (,S)  
RECALL v%, "filename" (,S)

This instruction is the partner to STORE, which stores an array in a cassette tape file. RECALL will load back from tape the contents of an array previously saved on tape using STORE, and the procedure is the same as when using CLOAD. The array to store the RECALLED array must have been dimensioned prior to using RECALL, or an OUT OF DATA error occurs. The array variable must be the same type as that of the original array (integer, string or real) and identical size or greater.

Slow speed data transmission from tape can be specified by adding a comma followed by an S ( ,S) to the command. This must also have been used by the STORE instruction. See STORE for an example program.

**Related keywords:** CLOAD, CSAVE, DIM, STORE

**RELEASE****BASIC Token:** 160**Format:** RELEASE

When the GRAB command has been used to liberate that area of memory allocated for the HIRES screen for the development of a lengthy BASIC program, this screen cannot be used again until the Oric's memory organisation is returned to normal by the use of RELEASE. This instruction reallocates the bytes #9800 to #B400 (on the 48K Oric) or bytes #1800 to #3400 (on the 16K machine).

**Related keywords:** HIRES, RELEASE, GRAB

**REM****BASIC Token:** 157**Format:** REM

REM statements are essential tools for all serious programmers whatever their experience, although their inclusion or otherwise has no effect on the way a program actually works. REM is short for REMARK or REMINDER and a REM statement enables the programmer to add a running commentary to a program which will explain the function of particular sections of code. The

Oric ignores everything following a REM statement for the duration of the line in which it is included.

It is arguable that very short programs will always be self-explanatory, and thus not require REM statements. However, we recommend that you get into the habit of using REMs from the very beginning, even if you are just keying in a few lines of code. There is always the tendency to consider any program in progress as self-explanatory while you are actually working on it, but when you come to try to sort it out later you will save yourself hours if you have taken the trouble to include even a schematic commentary of REMs. The instruction can be replaced by an apostrophe if you get tired of keying in REM each time you need a reminder.

```
10 REM *** REM ***
20 ' THIS WORKS AS A REM AS WELL
30 REM USE CHR$(96) TO GET ' IN LINES
```

**Related keywords:** None

## REPEAT

**BASIC Token:** 139  
**Format:** REPEAT

When coupled with UNTIL this command creates a loop which forces the Oric to repeat a series of instructions until a specified condition has been met. Unlike FOR...NEXT loops there is no counter to be incremented, and if one is required it must be programmed into the loop (as in the example below).

If you commit the cardinal sin of jumping out of a REPEAT...UNTIL loop with a GOTO statement (a practice much frowned upon by all but the most free-thinking programmers), it is imperative that you return to the loop or eventually suffer the consequences of a corrupted stack. The only way in which a REPEAT...UNTIL loop may be exited before completion is by utilising the PULL command (see individual entry in this section).

```
1 REM***REPEAT/UNTIL***
5 HIRES:INK4:PAPER0
10 A=0
20 REPEAT
40 CURSET55+A, 110-A, 3
50 DRAW2+A, 0, 1:DRAW0,A+A, 1
55 A=A+1
60 UNTIL A>60
```

**Related keywords:** FOR, NEXT, PULL, UNTIL

**RESTORE**

**BASIC Token:** 154  
**Format:** RESTORE

There are occasions when DATA must be READ more than once in the course of a single program. However, each time an individual item of DATA is READ by the Oric the computer's internal 'pointer' is moved along the statement until the DATA is exhausted. In order that the pointer is returned to the beginning of the statement for the information to be READ again, RESTORE must be used if an 'OUT OF DATA' error report is to be avoided. Oric BASIC does not permit restoration to a specific line number.

```

1 REM***RESTORE***
5 HIRES:PAPER1:INK0
10 C=1
20 FORA=1 TO5
25 IFC=0 THEN WAIT20 : SHOOT
30 READU
40 CURSETU, 40, 3
50 FORB=1 TO18 STEP3
60 CIRCLEB,C
70 NEXTB
80 NEXTA
90 IFC=0 THEN END
100 RESTORE
110 C=0
120 GOTO20
130 DATA28,73,118,163,208

```

**Related keywords:** DATA, READ

**RETURN**

**BASIC Token:** 156  
**Format:** RETURN

This instruction must be used as the final statement of any subroutine. It tells the Oric that the computer has reached the end of a subroutine and must RETURN to the line following the statement containing the original GOSUB instructions.

The destination of RETURN can only be altered with the use of the POP command.

```

1 REM***RETURN***
10 CLS
20 PRINT"KEY IN RADIUS OF CIRCLE"
30 INPUTR
40 C=2*PI*R :Z=C
50 GOSUB200
60 CLS
70 PRINT"CIRCUMFERENCE IS ";Z
80 A=PI*(R^2):Z=A
90 GOSUB200
100 PRINT"AREA IS ";Z
110 GOTO300
200 REM**SUBROUTINE TO CORRECT TO
    TWO DECIMAL PLACES**
210 Z=INT(100*(Z+.005))
220 Z=Z/100
230 RETURN
240 REM**END OF SUBROUTINE**
300 END

```

**Related keywords:** GOSUB, POP

## RIGHT\$

**BASIC Token:** 245

**Format:** RIGHT\$(a\$,i)

Like MID\$ and LEFT\$ this command extracts specific characters from a string. It takes the following format:

RIGHT\$(a\$,i)

in which a\$ is any pre-defined string variable and i is equal to the number of consecutive characters that you want extracted from the right-hand side of that string. Thus:

```

10 REM *** RIGHT$ ***
20 A$="ORIC RIGHT$"
30 FOR I=1 TO LEN(A$)
40 B$=RIGHT$(A$,I)
50 PRINT SPC(LEN(A$)-LEN(B$));B$
60 NEXT I
70 END

```

will extract the specified number of characters right to left from "ORIC RIGHT\$". If i is in excess of the total number of characters in the specified string the Oric will simply return the entire string.

**Related keywords:** LEFT\$, MID\$

## RND

**BASIC Token:** 223

**Format:** RND(i)

Without RND, a lot of computer games would be extremely predictable, since it provides the Oric's built-in random factor. The random number generator is only a pseudo-random function, in that it produces a sequence of numbers between 0 (which it may equal) and 1 (which it never quite gets to).

RND performs different functions according to the value of the parameter within the brackets. The normal usage is RND(1), to give a random number between 0 and 1. This is not of great value in itself, as perhaps you will see if you enter PRINT RND(1) as a direct command a few times. The random numbers we wish to generate need to be within a certain range, so that we can use RND to simulate the throw of a dice or an x co-ordinate between 3 and 27, or suchlike.

To do this, we multiply the result of RND(1) by the appropriate factor, which will produce a number between, for example, 0 and 5.999999 if we multiplied by 6. Adding 1 and then taking the integer value will give us the result of a die throw, and the program below shows two equivalent methods of rounding down. The first uses INT, and the second merely assigns the result of RND(1)\*6+1 to the integer variable N%, which automatically rounds down.

```

1 REM *** ROUND AND RND ***
10 LET A=RND(1)*6+1
20 LET N% =A
30 LET R=INT(A)
40 PRINTR
50 PRINTN%

```

The result of RND(0) is more predictable. It returns the value of the last random number generated.

```

1 REM ***NOT SO RND ***
10 FOR R=1 TO 50
20 PRINTRND(0)
30 NEXT

```

Generating the same sequence of random numbers can sometimes be useful. The number sequence is entered at a specific point by a specific negative value for the RND parameter. The example program demonstrates that setting the seed for the generator by RND(-4) will produce the same sequence of numbers.

```

1 REM***RND SEED***
10 FOR K=1TO2:PRINT
20 SEED=RND(-4)
30 FOR F=1 TO 6
40 PRINTRND(1)
50 NEXTF
60 NEXTK

```

**Related keywords:** None

## RUN

**BASIC Token:** 152  
**Format:** RUN ln  
 RUN

Used in isolation as a direct command RUN commences execution of the current BASIC program in the Oric's memory. RUN LN, where ln is a line number, causes the computer to commence execution of the program from the specified line. If the program in question contains no such line the Oric will throw up an error message (UNDEFINED STATEMENT ERROR). RUN can also be used within the body of a program, as the example below demonstrates:

```

1 REM***RUN***
10 A$="PRESS ANY KEY TO STOP":GOSUB70
20 CLS:PAPER1
30 X=RND(1)*32+1:Y=RND(1)*20+1
40 PLOTX,Y,"ORIC":WAIT50
50 U$=KEY$
60 IF U$ THEN END ELSE RUN
70 Z=LEN(A$):FORL=1TOZ
75 REM**PRINT MID$(A$,L,1)
80 POKE47999+L,ASC(MID$(A$,L,1))
90 NEXTL
100 RETURN

```

**Related keywords:** CONT, END, STOP

**SCRN**

**BASIC Token:** 242  
**Format:** SCRN(x,y)

The SCRN function returns the ASCII code of the character at the screen position defined by the co-ordinates x (column number 0 -38) and y (row number 0 -26). SCRN works only in LORES and TEXT modes. Numeric expressions are valid for x and y, but the values must lie in the correct range, or an ILLEGAL QUANTITY error occurs.

The simple examples below illustrate the use of SCRN. The first, uses PRINT@, and the second uses PLOT. The hash sign is PRINTED at location 5,5 on the screen, by each method and line 20 uses SCRN to first PRINT the ASCII code for #, then the same SCRN expression is used with CHR\$ to print out the actual character found in the screen memory location corresponding to column x, row y.

```
5 CLS
10 PRINT@5,5;"#"
20 PRINT SCRN(5,5),CHR$(SCRN(5,5))
```

```
5 CLS
10 LET A$="#"
15 PLOT5,5,A$
20 PRINT SCRN(5,5),CHR$(SCRN(5,5))
```

SCRN also works with redefined characters, returning the ASCII code of the character which has been re-defined. The program below uses an asterisk as a missile fired up the screen, guided left and right with the appropriate cursor control keys, until one of the targets (formed by the re-defined ! character) printed across the screen is hit (when SCRN(X,Y)=33, the code for "!").

```
10 REM *** SCRN ***
20 CLS: FOR I=1 TO 8
30 READ A:POKE 46343+I,A
40 NEXT:POKE #24E,1:POKE#24F,1
50 PRINT @2,0;"PRESS SPACE TO FIRE, ARRO
WS TO MOVE"
60 FOR I=1 TO 10:PRINT @INT(RND(1)*37)+2,1;"!";
70 IF I=10 THEN 10:END
```

```

70 NEXT:PRINTCHR$(17);CHR$(6)
80 REPEAT
90 REPEAT:UNTIL KEY$="" OR KEY$=CHR$(1
3):IF KEY$=CHR$(13) THEN GOTO 200
100 X=20:Y=26
110 PRINT @X,Y;"*";:SHOOT
120 REPEAT:OX=X
130 IF KEY$=CHR$(8)AND X>2 THEN X=X-1
135 WAIT 1
140 IF KEY$=CHR$(9)AND X<39 THEN X=X+1
150 Y=Y-1
160 PRINT @OX,Y+1;" ";
170 IF SCRNC(X,Y)=33 THEN PRINT @X,Y;" "
:EXPLODE:Y=1:GOTO 190
180 PRINT @X,Y;"*"
190 UNTIL Y=1:PRINT @X,Y;" "
200 UNTIL KEY$=CHR$(13)
210 POKE#24E,32:POKE#24F,4:PRINTCHR$(17
);CHR$(6)
220 END
400 DATA 30,33,45,63,45,33,30,0

```

**Related keywords:** ASC, PLOT, PRINT“

## SGN

**BASIC Token:** 214  
**Format:** SGN(n)

SGN is a numeric function which gives the sign or signum of a number, returning -1 if the expression within the brackets evaluates as a negative number, 0 if the number is zero, and 1 if it gives a positive value.

```

10 REM **SGN Function**
20 INPUT"Enter a number";N
30 PRINT"Your number was ";
40 ON SGN(N)+2 GOSUB100,200,300
50 PRINT"Hit a key to try another numbe
r"
60 GET M$

```

```

70 GOTO20
100 PRINT"NEGATIVE"
110 RETURN
200 PRINT"Zero"
210 RETURN
300 PRINT"POSITIVE"
310 RETURN

```

The example program uses SGN to test the number INPUT, and uses this value plus 2 to give 1, 2 or 3 for use with the ON...GOTO instruction in line 40, which passes control to the relevant subroutine to print out the sign of the number.

**Related keywords:** ABS

## SHOOT

**BASIC Token:** 163  
**Format:** SHOOT

Shoot is a pre-defined sound command that produces a gunshot noise. In a program, SHOOT merely requires a line like:

10 SHOOT

Multiple SHOOT instructions must, as with all pre-defined sound commands except ZAP, use WAIT to give sufficient time for the noise to execute and avoid the sounds running into each other. Try the following:

```

10 FOR A=1TO6
20 SHOOT
30 NEXT

```

This produces a sound like a single SHOOT instruction because the loop finishes before the first sound has finished. Add the line:

```

10 FOR A=1TO6
20 SHOOT
25 WAIT25
30 NEXT

```

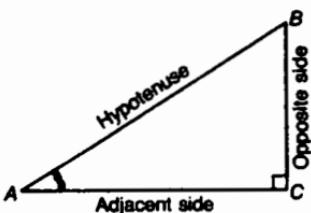
and you'll get a six-gun firing distinct rounds.

**Related keywords:** EXPLODE, PING, ZAP

**SIN**

**BASIC Token:** 227  
**Format:** SIN(n)

SIN is a trigonometric function returning the Sine of the angle given by the value of n, with n expressed in radians. The function calculates the basic trigonometric ratio, which for a right triangle as shown in the diagram gives the Sine of the angle at A (SIN(A)) as the length of the opposite side divided by the length of the hypotenuse, i.e. BC/AB.



The numeric expression evaluated by the SIN function must be expressed in radians, and not degrees. As the angular measure goes from 0 to 360 degrees round the circle, the measure in radians goes from 0 to  $2\pi$  radians. Conversion of degrees to radians and vice versa is simple, using the Oric's built in PI function:

$$\begin{aligned} x \text{ degrees} &= x * \text{PI}/180 \text{ radians} \\ x \text{ radians} &= x / \text{PI} * 180 \text{ degrees} \end{aligned}$$

The example program prints out the values of the SINE of the angles from 0 to 360 in steps of ten degrees. Line 30 converts the degrees to radians, and line 40 prints the angle and the result of applying SIN to the value in radians.

```

10 REM **SIN Function**
20 FOR ANGLE=0 TO 360 STEP 10
30 LET RADIANS=ANGLE*PI/180
40 PRINT ANGLE,SIN(RADIANS)
50 NEXT

```

If you look at the result closely, you will see that the values vary between 0 and 1, without quite reaching either, due to the inaccuracies in the multiple calculations involved. Change line 40 to the following, which rounds the result to 4 decimal places, to see a clearer picture of the SIN values:

```

10 REM **SIN Function**
20 FOR ANGLE=0 TO 360 STEP 10
30 LET RADIANS=ANGLE*PI/180
40 PRINT ANGLE,(INT(SIN(RADIANS)*1E4+0.
5)/1E4)
50 NEXT

```

The next program uses TAB in conjunction with SIN to calculate a position for printing an asterisk, producing a SIN curve as the screen scrolls. Since the value given by SIN varies between 0 and 1, the expression 18\*SIN(T) in line 20 gives a value between -18 and +18, which is added to the TAB position to vary the placing across the screen.

```

10 FOR T=0 TO 8*PI STEP PI/8
20 PRINT TAB(20+18*SIN(T));"*"
30 NEXT

```

**Related keywords** ATN, COS, PI, TAN

## SOUND

**BASIC Token:** 167

**Format:** SOUND c,p,v

c=channel 0-6

p=pitch 0-65535

v=volume 0-15

SOUND produces, as you might imagine, a defined sound from the Oric's dedicated sound chip. The Channel defines which of the three tone channels (which are channels 1, 2 and 3), or noise channels (4, 5 and 6) is activated. SOUND may be used without a PLAY command to activate channels, and uses tone channel 1 in this case. Pitch controls the tone of the sound produced, defining the frequency of the tone produced. The useful range is 0-2000, but the example program below will enable you to judge for yourself. Volume has a range from 0, which activates the envelope parameter of a PLAY command, and produces nil volume without a PLAY instruction, through 1 (quiet) to 15 (loud!).

```

10 FOR P=0 TO 32767
20 SOUND 1,P,2
40 NEXT

```

The listings below give some idea of the flexibility of the SOUND command. Whilst the basic structure of the program is the same in each case, very different sounds are produced. The first program has a PLAY command which activates noise channel 1, followed by the SOUND instructions, using WAIT to set the length. Channel is set to 4 (=noise channel 1), the pitch varies according to the value of the loop variable P, and two different volumes are used.

```

1 REM SOUND EXAMPLE
5 FOR P=1 TO 3
10 PLAY 0,1,1,250
20 SOUND 4,8+P,6
30 WAITS
40 SOUND 4,3+P,3
50 WAIT 20
60 PLAY 0,0,0,0
70 NEXT P
80 GOTO 5

```

The last parameter of the PLAY command is not activated whilst the volume of the SOUND command is other than 0. Change line 20 to read SOUND 4, 8+P,0 and line 40 to SOUND 3+P,0 to hear what happens when the envelope value is activated. (The same applies to the next two examples.) In the next program, only the sound channel value is changed, and it is set to 1, which is a tone channel.

```

1 REM SOUND EXAMPLE 2
5 FOR P=1 TO 3
10 PLAY 0,1,1,250
20 SOUND 1,8+P,6
30 WAITS
40 SOUND 1,3+P,3
50 WAIT 20
60 PLAY 0,0,0,0
70 NEXT P
80 GOTO 5

```

The final listing merely deletes line 10, and uses SOUND without a PLAY instruction, although PLAY 0,0,0 is used to turn off the sound.

```

1 REM SOUND EXAMPLE 3
5 FOR P=1 TO 3
20 SOUND 1,8+P,6
30 WAITS
40 SOUND 1,3+P,3
50 WAIT 20
55 REM DELETE LINE 60 FOR YET ANOTHER SO
UND ?
60 PLAY 0,0,0,0
70 NEXT P
80 GOTO 5

```

**Related keywords:** MUSIC, PLAY, WAIT

## SPC

**BASIC Token:** 197  
**Format:** SPC(n)

SPC is a PRINT operator which places n spaces on the screen. If n is not an integer, the value is rounded down. Expressions may be used, and SPC is useful in formatting, since string functions may be used inside the brackets. SPC is used in PRINT statements to place spaces before or between the other PRINT items.

```

10 REM**SPC Demo**
20 FOR F=1TO10
30 PRINT "*";SPC(F);"*";F;"spaces"
40 NEXT

```

The program above uses a loop value to define the number of places to be placed between the asterisks, and helpfully tells you how many it has placed to boot. The SPC function is useful as an alternative to the use of TAB and is flexible in use, as the next example shows. The value produced within the FOR...NEXT loop is converted to the string form, using STR\$ in line 30, and LEN is used with this string to give a calculated number of spaces in line 40. This produces a display with all the numbers aligned.

```

1 REM Calculated SPC Value
10 FOR F=1 TO 10
20 LET Nx=F^2*39
30 LET N$=STR$(Nx)
40 PRINT SPC(20-LEN(N$));Nx
50 NEXT F

```

**Related keywords:** PRINT, LPRINT, TAB

## SQR

**BASIC Token:** 222  
**Format:** SQR(n)

SQR is a function which calculates the square root of the value represented by the numeric expression n, which must be positive ( $>0$ ), or an ILLEGAL QUANTITY ERROR results.

```

10 REM SQR function demo
20 FOR N=30 TO 20 STEP-1
30 PRINT N,SQR(N)
40 NEXT N

```

The program produces a display of the numbers from 30 to 20 with their square roots.

**Related keywords:** None

## STOP

**BASIC Token:** 179  
**Format:** STOP

The instruction STOP halts program execution, displaying the message BREAK IN ln where ln is the number of the line containing the STOP instruction. The program may be re-started with the command CONT, followed by RETURN. It is similar to the instruction END, but END terminates the program, which may not be re-started, as the example program illustrates.

```

10 REM STOP
20 FOR X=1 TO 3
30 PRINT "X=";X
40 PRINT "PROGRAM HALTED. ENTER CONT TO RESTART."

```

```

50 STOP
60 NEXT X
70 END
80 PRINT"THIS LINE NOT PRINTED, SINCE CON
T DOES NOT WORK WITH END"

```

**Related keywords:** END, WAIT

## STORE

**BASIC Token:** 130

**Format:** STORE v ,“filename” ,(S)  
 STORE v\$,“filename” ,(S)  
 STORE v%,“filename” ,(S)

STORE is used to save the contents of an array as a cassette file on tape. The array must have been previously DIMENSIONED, either via a DIM instruction or implicitly by using an array element to set a default 11 element array, else an OUT OF DATA error will occur. The array may be a floating point array, e.g. A(20), an integer array, e.g. A%(20), or a string array such as A\$(20). The array variable must be specified as A, A\$, or A%, to identify the array correctly. Arrays with multiple dimensions are allowed. The “filename” can be anything you like up to 16 characters.

The same procedure is used as with CSAVE, and the default baud rate of 2400 baud (fast save) can likewise be altered to the slow rate of 300 baud by appending S after a comma ,(S) to the end of the command. The message SAVING FILENAME (or whatever you've called the array) appears on the status line, followed by a letter specifying the type of array: R for arrays containing Real floating point numbers; I for an Integer array, and S for String arrays. Try the following program to see STORE and RECALL in action. A\$(20) is DIMENSIONED and loaded with demonstration strings. The array is then STORED, and the variables cleared. The program will then RECALL the array and PRINT sample values.

```

1 REM *** STORE/RECALL ***
10 DIMA$(20)
20 FOR F=0 TO 20
30 LETA$(F)="NUMBER "+STR$(F)
40 NEXT
50 PRINT"START CASSETTE ON RECORD, PRESS
A KEY":GET A$
60 STORE A$,"ARRAY",S

```

```

70 CLEAR
80 DIM A$(20)
90 PRINT"REWIND CASSETTE, SET TO PLAY, PRESS ANY KEY":GET A$
100 RECALL A$,"ARRAY",S
110 PRINTA$(9)
120 PRINTA$(10)
130 END

```

**Related keywords:** CLOAD, CSAVE, DIM, RECALL

## STR\$

**BASIC Token:** 234  
**Format:** STR\$(n)

This is a string function, used to transform a numeric value into a string form. It is thus the opposite of VAL, which turns a string of numeric characters into a number. Exponential and hexadecimal numbers may be used, and they will be converted to standard notation (as would normally appear on screen) before being turned into a string. The string has a first character that holds the sign if the number is negative, but is left as a space if the number is positive.

```

1 REM **STR$ **
10 LET N=12.34
20 CLS:PRINT STR$(N):WAIT 15
30 PLOT 10,1,STR$(N):WAIT 45
40 PLOT 0,1,STR$(N) :WAIT 15
50 PLOT 10,8,STR$(N):WAIT 15
60 LET X=-23.5
70 PRINT
80 PRINT STR$(X):WAIT 15
90 PLOT 9,2,STR$(X)
100 PRINT "HEXADECIMAL #A3 PRINTS AS "S
TR$(#A3)
110 PRINT"1.345E-4PRINTS AS "STR$(1.345
E-4)
120 PRINT"1.23E2 PRINTS AS "STR$(1.23E2
)

```

The program uses PLOT to place characters, but could equally well have used PRINT@.

STR\$ is also useful in placing numbers on to the HIRES screen, as only string characters can be placed using CHAR. The next program illustrates this, using STR\$ to get the string form of a number, then taking each character in turn, finding the character code with ASC, and placing it on the screen. The same technique is used to strip off the first character if the number is not negative.

```
10 REM *** STR$ ***
20 HIRES
30 FOR A=1 TO 10 STEP .5
40 A$=STR$(A)
50 GOSUB 100
60 NEXT
70 END
100 REM PUT STRING ON HIRES SCREEN
110 FOR B=1 TO LEN(A$)
120 CURSET B*6+A*6,A*16,0
130 CHAR ASC(MID$(A$,B)),0,1
140 NEXT
150 RETURN
```

**Related keywords:** VAL

# TAB

**BASIC Token:** 194  
**Format:** PRINT TAB(n)

TAB is used in a PRINT statement to place PRINT items at a particular column position. The value of n defines the column to which the PRINT position will be advanced. The next item to be PRINTed will follow directly on, if nothing or a semi-colon is placed after the TAB(n) statement, or in the next PRINT field if a comma is used. Numbers are PRINTed with leading spaces if not negative. Columns are numbered 0 to 39 across the screen, and the following program will illustrate the effect of the protected columns.

```
20 FOR F=0 TO 15
30 PRINT TAB(F);F
40 NEXT F
50 REM NOW USE CTRL ] AND TRY AGAIN
```

There can be more than one TAB statement in a PRINT instruction. The next example illustrates this.

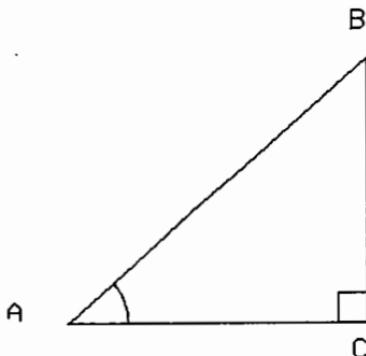
```
10 PRINT TAB(10)10;TAB(20);20
20 PRINT TAB(10);10;TAB(20),-20
30 PRINT TAB(10)"X"TAB(20),"Y"
```

**Related keywords:** LPRINT, PRINT, POS, SPC

**TAN**

**BASIC Token:** 228  
**Format:** TAN(n)

TAN is a trigonometric function which returns the tangent of the angle given by the numeric expression n. TAN gives a result which is equivalent to SIN(n)/COS(n). In the right triangle pictured below this is the ratio of Opposite side/Adjacent side. The value of n must be expressed in radians. See SIN for the conversion of degrees to radians and vice versa.



The first example program merely displays a table of the values of TAN for angles from 0 to 360 degrees (2\*PI radians).

```

10 REM *** TAN ***
20 DEF FNRC(DEG)=DEG*PI/180
30 FOR D=0 TO 360
40 P=D/20:Pz=D/20
50 PRINT D,TAN(FNRC(D))
60 IF P=Pz THEN WAIT 100
70 NEXT

```

The second example calculates the distance from B to C (imagine it's across a river whose width is unknown) from a knowledge of the distance AC (along the riverbank) and the angle at A.

```

10 REM *** TAN ***
20 CLS:PRINT"ANGLE AT A IS ";
30 READ A:PRINT A
40 PRINT"DISTANCE ALONG AC IS ";
50 READ AC:PRINT AC
60 PRINT:PRINT" SINCE THE TANGENT OF A IS "
S DEFINED AS"
70 PRINT"BC/AC WE CAN SEE THAT BC IS TAN(A)*AC"
80 PRINT:PRINT" THUS BC IS ";TAN(A)*AC
100 END
200 REM ANGLE IS IN RADIANS
210 DATA 0.73,20

```

**Related keywords:** ATN, COS, SIN, PI

## TEXT

**BASIC Token:** 161  
**Format:** TEXT

TEXT is an instruction that places the Oric in the standard text mode, as at initial switch-on, with the 40 column by 27 row screen, for display of the standard character set (and user defined characters). Use of LORES or HIRES sets alternative screen modes which remain until countermanded by CLS and TEXT respectively.

The LORES screen will scroll upwards, leaving a TEXT screen, with repeated PRINTings, however.

**Related keywords:** HIRES, LORES

## TROFF

**BASIC Token:** 133  
**Format:** TROFF

TROFF, standing for TRace OFF, turns off the trace facility by which program line numbers are displayed on screen as the lines are executed by the BASIC

interpreter. See TRON, which activates the trace facility.

**Related keywords:** TRON

## TRON

**BASIC Token:** 132  
**Format:** TRON

TRON (meaning TRACE ON) enables the aid to debugging and tracing problems in programs, which prints the line number of each program line in square brackets [ ] when the BASIC interpreter encounters and executes each line. TRON and TROFF enable the programmer to see the sequence in which the lines are executed, as well as the results of the execution which would normally be displayed, when inserted (temporarily) into the program listing. In the example program below, there is a problem of an endless loop. The TRON statement prints out the line numbers, and we can see the flow of control.

```
10 REM *** TRON ***
20 CLS
30 TRON
40 FOR A=1 TO 10
50 PRINT A
60 IF A=6 THEN A=1
70 NEXT
80 END
```

Inserting a line 65 TROFF would give us a display of line numbers for only the first cycle through the FOR...NEXT loop.

**Related keywords:** TROFF

## TRUE

**BASIC Token:** 239  
**Format:** TRUE

TRUE is a system constant built in to the Oric which returns the value of -1, which is used to represent the result of evaluating a conditional expression as TRUE. The value used by the Oric for FALSE is 0, and the FALSE variable holds this value. The two variables are used in conjunction to make a program clearer, primarily in conjunction with the use of flags, which are

commonly set to be equal to one of two values, and represent conditions which may easily be tested. TRUE must be used with some care if the NOT operator is to be used, since the Oric takes any non-zero value for a numeric expression to be true, but whereas NOT TRUE=FALSE and NOT FALSE=TRUE, the conditional test IF AB THEN . . . will evaluate as TRUE if variable AB is anything other than zero, but if AB were, say, 34, then NOT AB would return -23. This is not the same as FALSE in the Oric's eyes!

The program illustrates the use of TRUE and FALSE to control a REPEAT . . . UNTIL loop.

```
5 FLAG=FALSE
10 PRINT"INPUT A TWO-LETTER WORD"
20 REPEAT
30 INPUT A$
40 IF LEN(A$)=2THEN FLAG=TRUE
50 UNTIL FLAG=TRUE
60 PRINT"SO YOU CAN READ INSTRUCTIONS!"
```

**Related keywords:** FALSE

## UNTIL

**BASIC Token:** 140  
**Format:** UNTIL c

UNTIL forms part of the REPEAT . . . UNTIL loop structure. When an UNTIL statement is encountered, the conditional expression is evaluated. If it is found to be TRUE, program control will pass to the next statement, and the loop will be exited. If the condition is FALSE, control will be passed back to the statement following the REPEAT which initialised the loop. A ?BAD UNTIL ERROR will be produced if no corresponding REPEAT is found.

**Related keywords:** FALSE, REPEAT, TRUE

## USR

**BASIC Token:** 217  
**Format:** DEF USR = addr  
 USR(i)

This function allows access to machine code routines in the course of a BASIC program. In the command format above, the first example DEFINES the start address of the machine code routine as addr.

In the second format, the routine is called by `USR(i)` and places the value of `i` in the floating point accumulator. Once the machine code routine is completed, `USR` returns to the main body of the program and the result must either be printed (`PRINT USR(0)`) or assigned to a variable (`A=USR(0)`). If there are no values to be passed to a machine code routine, then the routine should be invoked by `CALL`. See Chapter 10 for an introduction to machine code programming.

**Related keywords:** CALL, DEF

## VAL

**BASIC Token:** 235

**Format:** VAL(a\$)

This is a string function which returns the numeric value of the characters given by the string expression within brackets. The first character of the string to be evaluated must begin with a space, a minus sign, a hash sign or a number, else zero is returned. After these characters, or the first number, the string is evaluated up to the first non-numeric character, as the decimal equivalent if a hexadecimal number (starting with #) is found in the string. Exponential notation is also handled, and the number will be held in the same form as it will print on the screen, rather than in the precise form it had within the string. To see this, try entering, say, 1.23E2 into the program below, along with any other numeric forms. Any non-numeric characters found in the string after the Oric has found characters it can interpret as a number will be ignored.

```

10 REM *** VAL ***
20 CLS
30 REPEAT
40 INPUT "ANY STRING PLEASE ";A$
50 PRINT "THAT STARTS WITH THE NUMBER";
60 PRINT VAL(A$)
70 PRINT
80 UNTIL A$="STOP"
90 END

```

**Related keywords:** ASC, STR\$

**WAIT**

**BASIC Token:** 181  
**Format:** WAIT n

This instruction gives a delay of n one-hundredths of a second before program execution continues. WAIT is crucial to the sound commands of the Oric, for the purpose of controlling the duration of the results of PLAY, SOUND and MUSIC instructions. It may also be used to introduce pauses into programs, for slowing down screen displays, but it should be noted that no keyboard input will interrupt a WAITing period. For delays until the user presses a key, or inputs data, GET and INPUT must be used.

**Related keywords:** MUSIC, SOUND, PLAY

**ZAP**

**BASIC Token:** 165  
**Format:** ZAP

This is one of the Oric's pre-defined sounds, producing a noise like a futuristic weapon report for use in games. Unlike its Oric comrades, ZAP does not need a delay whilst the sound is produced, and it may be used repeatedly without the WAIT instruction required by PING, SHOOT and EXPLODE.

```
10 FOR F=1TO6
20 PAPER F
30 ZAP
40 NEXT
```

**Related keywords:** EXPLODE, SHOOT, PING

# 10 Introducing machine code

Assuming you have had a go at writing some programs in Oric BASIC, you may be wondering if you could speed things up a little so that your Invaders move more swiftly and smoothly across the screen, and your programs generally run faster. You can – but first let us consider why the animated effects sometimes appear slow or jerky and noticeable delays occur in complex programs.

BASIC is an *interpretive* language (although it can also be *compiled* if a suitable program and floppy disk system are available), which means that the instructions you code (the program) are held almost exactly as keyed in. This means that the BASIC interpreter has to examine each statement at RUN time, decide what the statement is trying to do, and then call appropriate machine code routines to action the statement. Although the machine code routines are themselves very fast and efficient, several routines may need to be involved for a single statement. Even more important, each statement must be *parsed*, i.e. scanned for operators such as '+', '−', 'IF', etc, and then rearranged into a form suitable for linear execution in the specific sequence the Oric requires (conforming to the priority sequence) and finally passed on to the machine code routines. This is the interpretive process, and it occurs for every statement, each time the statement is encountered. It is this process which slows things down. Therefore, if we code directly in machine code and develop our own routines for specific functions, we get code which is very much faster, and it is even possible that we may have to introduce delays deliberately to slow things down! Ever tried shooting down an Invader which crosses the screen in one-tenth of a second?

However, coding a program in machine code has its disadvantages too. It is difficult to learn, it is easier to make mistakes when using machine code, and it is more difficult to debug – no helpful messages like 'SYNTAX ERROR'! But it is also interesting and rewarding and, provided that a little care is taken, is well within the reach of the hobbyist as well as the serious programmer.

To use machine code efficiently and correctly it is necessary first to learn something about the 6502 microprocessor itself, but before we can do this we need to know a little about number systems, or ways of representing the manipulating numeric information. Knowledge of the way in which charac-

ters (such as the alphabet) are represented is also necessary. We will therefore deal with these now before we get on to the instruction codes themselves. (Do not skip or skim the following paragraphs – they could save you hours of debugging!)

### Number systems – decimal

Thanks to Mother Nature we use a *decimal* number system for most of our arithmetic needs – counting starts on our fingers and thumbs. Thanks to the Arabs, who gave us the zero and decimal point, we have been able to develop methods for multiplication and division, without which life might be a bit tedious. Let's look first at the decimal number system.

When we count we usually start at one, although in a count-down we count backwards to zero. 5 . . 4 . . 3 . . 2 . . 1 . . ZERO – WE HAVE LIFT-OFF! The zero is important and should really start any count (upward). When we're counting upwards and reach number nine we then revert to zero again, but now add a leading digit (one for the first time, two for the second, etc. – 0, 1, 2, 3, 4, . . . 9, 10). When we reach 99, we add another leading digit and revert to zero – 100.

We are really adding one to the previous number until the last digit reaches 9. Then we reset the last digit to zero and carry one, adding it to the second-to-last digit. If this is already 9 we reset it to zero and carry one to the next digit, and so on. Finally, adding a number greater than one to another number is simply a matter of adding one more than once. For example, 9+5 can be written as 9+1+1+1+1+1. We do not normally do this because we have memorised the sums of all possible combinations of the digits 0 to 9, and the process of addition has become automatic.

So why state the obvious? Well, patience is a virtue and automatons are not creative, so we have to stretch our automatic assumptions. Computers generally work in a non-decimal number system. Ever wondered why 1K in computing is 1024 instead of 1000 – after all, a kilogram (or kg) is 1000 grams? You are about to find out, because the next number system is *binary*.

### Binary

The binary system uses only two digits: zero and one (0 and 1). In computing we always write zero as '0' to distinguish it from the letter 'O', and this is a practice which you would do well to follow. In the decimal system we had a carry of one whenever we added one to a nine, the nine at that position then reverting to zero. In the binary system our carry of one occurs when we add one to one, and the digit-position reverts to zero. So,  $1+1=10$ . This is the first rule. The second is even easier:  $0+1$  (or  $1+0$ ) = 1. The third is obvious:  $0+0=0$ .

Try this for practice:  $10101+01101=?$

If you did not get an answer of 100010 you need to reread the previous paragraph.

Now let us compare the decimal digits 0 to 15 with the binary equivalents.

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

Being an astute reader you will have noticed two things – that the binary numbers were written as a sequence of four binary digits, and that a third column labelled ‘hexadecimal’ had been included. You may even have noticed that this third column looked like decimal until decimal value 10, and that it then became alphabetic. This is because writing long strings of ones and zeros can be pretty tedious, and can lead to errors, so we represent groups of four binary digits by a single digit. Since there are no single-digit numbers greater than 9 we ‘borrow’ the first six characters of the alphabet – A, B, C, D, E, and F – to represent the numbers 10 to 15. This makes life much easier – for example, the decimal number 1024 in binary is 010000000000, and in hexadecimal is 400. We arrive at this value by grouping the binary digits, from the right, in fours, and then converting each group to its hexadecimal (or hex, for short) equivalent. Thus:

0100 0000 0000 binary  
4 0 0 hex

Very clever, you may think, but what does all this have to do with programming in machine code? Everything, but bear with us for a while – there are another pair of terms to be introduced first. They are both conventions. First, we refer to *binary digits* as *bits*. We can say therefore, that a hexadecimal (or hex) digit represents four bits. Secondly, the main unit of computer memory is the *byte*, which consists of eight bits. Therefore, the contents of a byte can be expressed as two hex digits. For example:

$$\begin{aligned}\text{decimal } 17 &= \text{binary } 00010001 \\ &= \text{hex } 11\end{aligned}$$

The highest binary value that can be contained in a byte (eight bits) is expressed:

$$\begin{aligned}1111 \quad 1111 \\ &= F \quad F \quad \text{in hex} \\ &= 255 \quad \text{in decimal}\end{aligned}$$

A byte can therefore represent 256 values, i.e. 0 to 255 decimal. The bits in a byte are numbered 0 to 7, right to left, when we need to refer to them individually.

Converting numbers between binary or hex and decimal is likely to get tiresome for large numbers unless we understand a little more about number systems. Basically, the decimal system is to the base 10, and the value of each digit (working from the right) is the product of the digit and 10, to an increasing power. A power is the number of times a number is multiplied by itself, and a power of zero always gives 1 (except for zero to the power of zero which is zero).

For example, decimal 123 can be written as:

$$\begin{aligned}(1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0) \\ \text{or} \\ (1 \times 10 \times 10) + (2 \times 10) + (3 \times 1) \\ \text{or} \\ (1 \times 100) + (2 \times 10) + (3 \times 1) \\ \text{or} \\ 100 + 20 + 3 = 123\end{aligned}$$

The small numbers above the 10's are the powers.

In binary the same principle is used, except that the base used is two. We can therefore write decimal 15 in binary as:

$$\begin{aligned}1111 &= (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) \\ &= (1 \times 2 \times 2 \times 2) + (1 \times 2 \times 2) + (1 \times 2) + (1 \times 1) \\ &= (1 \times 8) + (1 \times 4) + (1 \times 2) + (1 \times 1) \\ &= 8 + 4 + 2 + 1 \\ &= 15 \text{ Decimal}\end{aligned}$$

Hexadecimal uses base 16 (there are sixteen digits in the system, 0–15). So, hex 21 can be written as:

$$\begin{aligned}\text{hex } 21 &= (2 \times 16^1) + (1 \times 16^0) \\ &= (2 \times 16) + (1 \times 1) \\ &= 32 + 1 \\ &= 33 \text{ decimal.}\end{aligned}$$

The above examples show conversion of binary and hexadecimal to decimal. To convert from decimal into binary or hex we divide by the appropriate base, repeatedly, and then string together the remainder digits, from the final operation backward:

$$\left. \begin{array}{l} \text{decimal 15 } 15/2=7, \text{ remainder 1} \\ \quad 7/2=3, \text{ remainder 1} \\ \quad 3/2=1, \text{ remainder 1} \\ \quad 1/2=0, \text{ remainder 1} \end{array} \right\} = 1111 \text{ binary}$$

$$\left. \begin{array}{l} \text{decimal 11 } 11/2=5, \text{ remainder 1} \\ \quad 5/2=2, \text{ remainder 1} \\ \quad 2/2=1, \text{ remainder 0} \\ \quad 1/2=0, \text{ remainder 1} \end{array} \right\} = 1011$$

Similarly, for conversion from decimal to hexadecimal we divide by decimal 16:

$$\left. \begin{array}{l} \text{decimal 43 } 43/16=2, \text{ RY 11=hex B} \\ \quad 2/16=0, \text{ RY 2=hex 2} \end{array} \right\} = 2B$$

$$\left. \begin{array}{l} \text{decimal 107 } 107/16=6, \text{ RY 11=hex B} \\ \quad 6/16=0, \text{ RY 6=hex 6} \end{array} \right\} = 6B$$

### Negative numbers

So far we have considered only arithmetic which results in a positive answer. We also need to consider negative results. How are they represented and identified?

The convention in binary systems is to indicate a negative number by setting the high (leftmost) bit to a 1. This itself is not enough – another convention is also required to allow the addition of positive and negative numbers and obtain a correct result. For example, let us add  $-4$  and  $+5$ , using the high order bit in the former number to indicate negative status or sign.

$-4 = \text{binary}$	$10000100$
$+5 = \text{binary}$	$00000101$
binary addition gives	$10001001$

This is obviously incorrect. To achieve correct results we must represent negative numbers by subjecting the absolute value to a process called *two's complementation*. This simply entails a reversal of bit values (i.e. swapping 0 to 1 and vice versa) and adding 1. Let us try it with 4:

4 decimal =	$00000100$
'Flipping' 0's and 1's:	$11111011$
Adding 1 gives	$11111100$

Now adding +5	00000101
Result:	00000001

The result is nine bits: 100000001. The leftmost bit ‘falls off’ the end and is a carry, which can be ignored. The remaining bits gives a result +1, which is the correct sum of +5 and -4. To interpret a negative number we subtract 1 and then ‘flip’ the bits.

For example:

$$\begin{array}{r}
 11111111 \\
 - \quad \quad \quad 1 \\
 \hline
 11111110
 \end{array}$$

Flip = 00000001 = 1 and so 11111111 = -1 decimal.

Now consider the example of 127+1:

$$\begin{array}{r}
 127 = \quad 01111111 \text{ -- the sign bit is } 0 \text{ (positive)} \\
 + \quad 1 = \quad \underline{\quad 00000001} \\
 \hline
 10000000 \text{ -- the sign bit is now } 1!
 \end{array}$$

This *overflow* from bit 6 to bit 7 which appears to change the sign of a result is an error condition which must be tested and catered for in the arithmetic. In practice it is only the most significant bit of the total number that needs to be tested for an error – in a 16 bit number it is bit 15. It is therefore essential to decide the magnitude of the largest number to be handled when deciding how many bytes are to be allocated for number storage. Overflow on low order bytes in a multi-byte number may be ignored and only carry testing is required.

### Binary coded decimal or BCD

So far we have been working in binary. It is also possible to represent decimal numbers in binary, and perform arithmetic with this form of representation.

Since the largest decimal digit is 9, and can be represented in 4 bits – 1001 – it is possible to store 2 decimal digits in an eight-bit byte. For example 99=10011001. This is known as packed decimal, and the representation of decimal numbers in this fashion is known as *binary coded decimal*, or BCD for short.

When performing arithmetic on packed BCD it is necessary to test for 9s overflow – the generation of binary combinatorics greater than 9 – and adjust the digit concerned, carrying 1 to the next digit.

Tens' complement numbers can be calculated by subtracting the positive value of a number from a string of 9s which represent the maximum value of numbers to be represented, then adding 1. For example, if the largest number we intend to work with is 999, the tens' complement of 2 (i.e. -2) will be given by:

$$\begin{array}{r}
 999 \\
 - 2 \\
 \hline
 997 \\
 + 1 \\
 \hline
 998
 \end{array}$$

The sum of (say)  $763 - 2$  will therefore be:

$$\begin{array}{r}
 763 \\
 + 998 \\
 \hline
 1761 = 761 + \text{Carry 1 (ignored)}
 \end{array}$$

Fortunately, this process will not be necessary when writing 6502 machine code, since this chip has a decimal arithmetic capability! However, the above has been included for completeness. Well, that's the complexity of computer numbers clarified (we hope), so let us take a look at addressing.

## Addressing

When we write in BASIC each statement or group of statements has an associated line number. This is for the GOTOS and GOSUBS and can be considered as a program *address* for those commands which cause a transfer of control from the normal sequence of processing. If our program was purely sequential we would not need GOTOS or GOSUBS, and we would not require line numbers either. Some high-level languages other than BASIC only need line numbers as labels for jumps.

When we use machine code we do not use statement numbers (as addresses, anyway) and we need some way of pointing the machine code equivalents of GOTOS and GOSUBS at the appropriate part of the program. In addition, we must decide where our variables (and constants) are to be stored in memory, whereas in BASIC this is done for us automatically. When we come to the machine-code instructions we will find that many of these will require an address (of data, or of the next instruction), and we must supply these in a form that is machine executable. This means in binary, though preferably represented as hex digits for our convenience, since this is what the computer understands.

The 6502 chip has sixteen address lines, i.e. a memory address contains sixteen bits (though a special case, of eight bit addresses, also exists). This means that the highest memory location (or byte) is 1111 1111 1111 1111, or hex FFFF, or decimal 65535 (which is generally expressed as 64K). If we did not use hex notation (of four hex digits) we would need to write strings of sixteen bits for each and every instruction which addressed memory! Even the simplest program would take a very long time to code and enter, and would be extremely error-prone. So, dear programmer, if you have

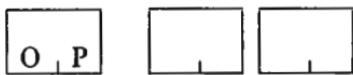
skimmed lightly over the paragraphs which precede this, it might be a good idea if you went back to the beginning and re-read this section. Also try some examples of your own, and check your answers against the tables at the back of this handbook to verify your results. It will be time well spent and will soon make you proficient in number system conversions. Even if you don't want to explore the complexities of machine code, an understanding of binary and hex will stand you in good stead in the world of computers.

### Machine code instructions

Instructions at the machine-code level have two main components – the operation code (or op code) and the operand(s).

The operation code, in the case of the 6502 (and most other microprocessors) consists of a single byte. The binary value of this byte specifies which operation (e.g. addition or subtraction) is to be performed. (For convenience, each op code is given a *mnemonic*, which means it's supposed to be easier to remember than the hex op code.) A list of op codes, and their associated mnemonics, will be found in Appendix 8.

The second part of an instruction, the operand, varies according to the nature of the instruction, and may consist of one or two bytes, or be implied by the op code itself (i.e. may not be actually stated as an operand.) In the latter case the op code specifies the complete operation. When physical operands do exist they contain either an address (one or two bytes) or data (one byte).



Op Code

Data (1 byte)

or

Address (2 bytes)

With one-byte op codes, it is possible to have 256 different instructions (values 0-255). However, the actual number of instructions required for satisfactory programming is considerably less, and the 6502 uses the spare values to modify the basic instruction set to include different addressing modes. As a result, each instruction type may have several different op codes, each one using a different mode of addressing. Before we examine these modes, however, we need to familiarise ourselves with the concepts of *registers*, *pages*, *indexing*, and *indirect addressing*.

One principle of machine-code operations should be pointed out briefly here. The operations of the 6502 are timed by a clock, which regulates the operation cycles of the CPU. A cycle allows one operation to take place. The first cycle in executing an operation fetches the op code, then the second

and third bytes of the operand are found in the subsequent cycles, if they are present. The next cycle will execute the operation, unless indirect addressing is being used, which will require more cycles. Different operations will thus occupy different numbers of cycles.

## Registers

A register is a memory location within the 6502 microprocessor chip which is used to hold data required by the instruction being executed. Because the registers reside within the chip itself, they are not directly addressable except by using the instructions which operate on them. The 6502 has six registers, one of which has sixteen bits (the PC or program counter register), the others (A, X, Y, S, and P registers) being eight-bit registers. Registers may also be used as temporary stores (to pass data between routines), or as counters. However, each of the registers mentioned above has a specific purpose which will be explained shortly. Let us first examine the other main concepts.

## Pages

It was mentioned earlier that the 6502, as a result of having sixteen address lines, can directly address  $2^{16}$  or 65536 locations in main memory. This is accomplished via instructions with two-byte addresses (value 0000 to FFFF hex). However, one-byte address instructions are also supported, which execute faster, and five of the six registers are eight bits (one byte) long. This implies that some instructions can address only 256 bytes of memory at the very start of RAM! In fact, a special addressing mode exists which allows short-address instructions to access the first 256 bytes of memory very much faster. This mode is called *zero-page* addressing, which introduces the concept of pages of memory.

Because of the internal architecture of the 6502, the 64K of RAM may be considered to consist of pages of 256 bytes each, with page zero and page one having special functions. For completeness, it should be said that instructions which cause a page boundary to be crossed take one extra cycle to execute.

The limitations of register length will now be discussed, introducing the concept of indexing.

## Indexing

Indexing is a means of dynamic address modification. In other words, an instruction address is changed at the time of execution. This facility is necessary to access consecutive items in tables, or arrays, for example, which would otherwise require separate instructions for each table item to

be accessed. The subscript facility for BASIC arrays is the high-level language equivalent.

You might recall that two of the registers mentioned earlier were the X and Y registers. It would be more appropriate to call them the X and Y index registers, since they are primarily designed for indexing. In brief, some of the op codes specify that the address of the data to be operated on is to be calculated (by the chip) by adding the contents of the X or Y registers to the specified address operand. To access multiple items in a table, therefore, one needs only to modify the contents of the relevant index register. Indexing, therefore, is yet another addressing mode.

### **Indirect addressing**

Indirection is a very powerful facility which allows an instruction address-operand to be selected (rather than computed or modified) from a number of possible addresses. The principle is that the address following the op code is used to point at a location in memory which contains the required address. This second address is used to access the data required by the instruction.

## **6502 Registers**

Since a large number of instructions use the registers mentioned earlier, we will first examine these registers.

### **A-Register (accumulator)**

The A-register is also called the accumulator, because it is used to accumulate results derived from arithmetic operations. Accumulator-based arithmetic is very fast because the operations are carried out within the hardware register. However, before any accumulator operation is performed, the accumulator must first be loaded with data from memory. Similarly, after the operation, the modified data is stored back into memory.

### **X and Y registers (indices)**

These are primarily index registers, and are used for dynamic address modification. When used as indices they must first be loaded or initialised. There are specific instructions to modify the contents of these registers, and other instructions which specify them as indices.

### S-register (stack pointer)

The S-register is used to locate the next available location in a special area of memory called the *stack*. The stack is effectively a sequential list of items, and is used for subroutine calls and exits. This concept will be dealt with more thoroughly in conjunction with subroutine control instructions. The stack register always addresses page 1 of RAM. For this reason page 1 should not be used by the programmer. There are specific instructions which use the stack register to modify the contents of page 1.

### P-register (processor status)

This register is really a set of eight one-bit flags, which provide information on CPU status following the execution of instruction. The flags are set and reset automatically by instruction execution. The values of these flags can be tested, and a range of instructions has been provided for this purpose.

The flags within the P-register are as follows:

Bit 7: N flag – set if arithmetic result is Negative.

Bit 6: V flag – is the overflow indicator which is set when a carry occurs from accumulator bit 6 to bit 7.

Bit 5: Not used.

Bit 4: B-flag – set when the BRK (Break) command is executed.

Bit 3: D-flag – set to 1 when processor is operating in Decimal mode: 0 for binary mode.

Bit 2: I-flag – Interrupt mark. Set by interrupts, or instruction to inhibit further interrupts. Cleared by specific instruction.

Bit 1: Z-flag – set to 1 when the result of arithmetic or data transfer is Zero.

Bit 0: C-flag – indicates a ‘Carry’ or ‘borrow’ on arithmetic, or the presence of a bit shifted or rotated out of an address or register.

Some bits within this register can be directly set or reset by the programmer.

### PC-register (program counter)

This is the only sixteen-bit register, and contains the address of the instruction following the one currently being executed. Although not directly accessible by the programmer, it can be stored into the stack and examined in the stack area. The PC-register is used by the CPU (Central Processing Unit) in the microprocessor to fetch the next instruction to be executed, and is automatically incremented as each instruction/address byte is fetched. It is also modified by Branch and Jump instructions, whenever these instructions change the sequence of program flow.

## 6502 Addressing modes

We have now covered some of the concepts of addressing, and move on to the actual modes supported by the 6502.

### Implied

An implied address is one where the address is specified by the op code itself. On the 6502, instructions which operate directly on specific registers are considered to have implied addresses (for example **INX** and **INY**, meaning increment X and Y register respectively), and are therefore single-byte instructions.

### Immediate

An immediate address is the address following the op code. Instructions operating in immediate mode contain a single byte of data in the byte following the op code. This data byte is called a literal. Instructions of this kind occupy two bytes.

### Zero page

A zero-page address is one which specifies a memory location within page zero (the first 256 bytes of memory). A zero-page instruction will therefore be two bytes long.

### Indexed

This mode of addressing uses index registers X and Y to modify the address following the op code. This address may be one-byte (page zero) or two-bytes (anywhere in memory). Instructions may therefore be of two or three bytes.

### Absolute

Absolute addresses are two-byte addresses unmodified by indexing. Instructions using this form of addressing are therefore three bytes long.

### Indirect

Pure indirect addressing uses a two-byte address after the op code to access a two-byte data address elsewhere in memory. Both addresses are unmodified by indexing.

Only one instruction in the 6502 instruction set uses this form of

addressing – the **JMP** (Jump) instruction, which is therefore *three* bytes in length.

### Indirect-indexed

The 6502 supports two forms of indirect-indexed addressing, both of which are used with index registers and are restricted to addressing page zero of memory.

The first form uses index register X and will be referred to hereafter as indirect-X addressing. It is used to access tables in page zero *only*. This form is actually better described as indexed-indirect, since the index register X is added to the zero page address to find the location which contains the first byte of the indirection address. This is the two-byte address which is used in accessing the memory.

The second form uses the index register Y and will be called indirect-Y addressing. This form allows access to specific entries within tables held anywhere in main memory. The final address (of data) is computed by adding the contents of Y to the sixteen-bit address pointed to by the page zero address following the op code. If the sixteen-bit address held in page zero points at the start of a table in main memory, then index register Y can be used to point at any specific entry within 256 bytes of the start of the table (Y is eight bits long – remember!). Since the op code is always followed by a page zero address, indirect-indexed address instructions are two bytes long.

### Relative

Relative addresses are based on the value held in the program counter, and are used by a group of instructions called *conditional branches*. These instructions are two bytes long and specify a test to be made on the P-register flags. The address following the op code is a one-byte address which specifies the relative address of the next instruction to be executed if the test is satisfied. Because the address is only one byte long the range of addressing is limited to 256. Since most loops are generally fairly short the address byte is allowed to contain both *positive* and *negative* (or forward and backward) branch addresses. This is accomplished by using the high-order bit of the address as a sign bit, allowing a forward branch of up to +127 bytes, and a backward branch of up to -128 bytes (using 2's complement arithmetic). Since branch instructions are two bytes long this results in actual branch ranges of -126 (-128+2) and +129 (+127+2), relative to the address of the branch instruction.

### Instruction classification by addressing mode

We have seen that several addressing modes exist for 6502 instructions, and

it has also been said that multiple op codes exist for each type of instruction. We will now expand on these modes and give each an abbreviated name so that, when the instructions are to be used, a quick glance at the table at the end of this handbook (Appendix 8) will make it easy to see which addressing modes (and op codes) are available for each instruction.

Mode	Length	Description of mode
I	1	Implied (implied operands)
IM	2	Immediate (literal follows op code)
ZP	2	Zero page (one-byte address)
ZX	2	Zero page, indexed by X
ZY	2	Zero page, indexed by Y
IX	2	Indirect X
IY	2	Indirect Y
RA	2	Relative address (Branch <i>only</i> )
AB	3	Absolute, no indexing
AX	3	Absolute, indexed by X
AY	3	Absolute, indexed by Y
IN	3	Indirect, no indexing (JMP <i>only</i> )

The following layout illustrates the format of the various instruction modes. Each box represents one byte.

## Instruction formats

I (Implied)	Op Code			
IM (Immediate)	Op Code	Literal		
ZP (Zero page)	Op Code	ZP Addr.		
ZX (ZP indexed by X)	Op Code	ZP Addr.		
ZY (ZP indexed by Y)	Op Code	ZP Addr.		
IX (ZP indirect X)	Op Code	ZP Addr.		
IY (ZP indirect Y)	Op Code	ZP Addr.		
RA (Relative)	Op Code	Rel. Addr.		
AB (Absolute)	Op Code	2-Byte	Address	
AX (Abs. indexed by X)	Op Code	2-Byte	Address	
AY (Abs. indexed by Y)	Op Code	2-Byte	Address	
IN (Pure indirect)	Op Code	2-Byte	Ind. Addr.	

A further classification of instructions can be made by considering the effect

on registers and storage locations when they are executed. The final classification can be made on function.

In the following pages, which describe the instructions, we will group them initially by function under the headings of *manipulation*, *test*, *arithmetic*, *logic*, and *control*. Within each group we will deal with sub-groups of instructions similarly. The table in Appendix 8 can then be used for quick reference to the op codes for each instruction mnemonic by addressing mode. We'll now take a look at the 6502 instruction set.

### Data manipulation instructions

By 'manipulation' we mean instructions that move data about, from memory to a register or vice versa, and between or within registers. The 6502 does not have instructions capable of moving data directly from one set of memory locations to another, so all data must be moved *via* a register. Since the registers that are available for this purpose are only one byte wide, we need to move data one byte at a time. After BASIC, where we have numeric variables and strings, this may appear both tiresome and slow. However, each move takes only a few microseconds, and program loops can be used to eliminate repetitive coding.

#### (a) Load instructions

The following instructions load the appropriate register with a single byte of data retrieved from the memory location specified by the address following the op code.

<b>LDA</b>	Load A (accumulator)
<b>LDX</b>	Load X (index register X)
<b>LDY</b>	Load Y (index register Y)

#### (b) Store instructions

The following instructions place the contents of the register at the memory location given by the address – the converse of load.

<b>STA</b>	Store A (accumulator)
<b>STX</b>	Store X (index register X)
<b>STY</b>	Store Y (index register Y)

To move data from one location to another (say hex 1000 to hex 1001) we would use the sequence:

<b>LDA</b>	\$1000
<b>STA</b>	\$1001

The \$ sign is used here to specify a hex address (in assembler). This is one of

the conventions we will adopt from now on. Note that the index registers can also be used for this purpose (when not in use as indices, of course).

### (c) Register transfer instructions

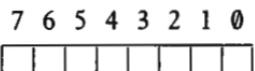
This group of instructions is used to transfer data between registers directly, i.e. without using a memory location as an intermediate store. Since the operation is entirely between registers, and can be fully specified by the mnemonic (and op codes), no address is required, and the instructions are one-byte long, and fast in execution.

<b>TAX</b>	Transfer A to X
<b>TAY</b>	Transfer A to Y
<b>TXA</b>	Transfer X to A
<b>TYA</b>	Transfer Y to A
<b>TSX</b>	Transfer S to X
<b>TXS</b>	Transfer X to S

Note that there are no instructions to transfer between Y and S or between X and Y.

### (d) Shift instructions

These four instructions manipulate data within the accumulator or a single byte of memory. If we consider either as a string of eight bits:



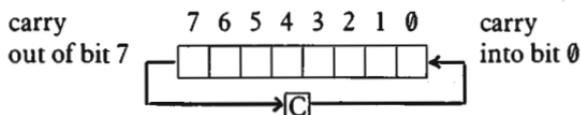
then the string is moved left or right by one bit position so that the endmost bit (7 or 0, depending on the instruction) falls off the end into the *carry* bit in the *P-register*. The vacated bit position is filled with a zero or from the previous contents of the carry bit (again, depending on the instruction).

<b>ASL</b>	Arithmetic shift left.
<b>LSR</b>	Logical shift right.
<b>ROR</b>	Rotate right.
<b>ROL</b>	Rotate left.

The accumulator is the only register which can have its contents shifted directly. Note that both types of shift do not take negative numbers into account. Bit 7 takes the value of 0 in **LSR** (which indicates a positive number) and the previous value of bit 6 for **ASL** (which may be either 0 or 1, positive or negative sign).

The difference between the shift (**ASL**, **LSR**) instructions and the two rotate instructions is that the latter are effectively nine-bit rotations, since the previous contents of the carry bit is shifted into the vacated bit position.

**ROL** is illustrated below as a diagram to indicate this.



There are four more instructions which could be classified as manipulative, but these have been included under 'control' because they are concerned with stack manipulation.

### Test instructions

This subset of instructions make true programming possible, since they allow different routines to be invoked as a result of programmed decisions. The BASIC equivalent is the 'IF... THEN GOTO...' and they can therefore be classified as conditional branches. There are however exceptions to this classification. The compare instructions test but do not include a branch operation; instead, this instruction sets flags in the P-register, which are used by the conditional branch instructions.

It should be stated immediately that most (but not all) 6502 instructions affect the flags in the P-register. You should refer to the section on P-register flags now so that the following will be more easily understood. The op code table in Appendix 8 shows which flags are affected by each instruction, and can therefore be used to select the appropriate conditional branch instructions when programming.

#### (a) Conditional branches

	P-register
<b>BCC</b> Branch on carry clear (carry=0)	C
<b>BCS</b> Branch on carry set (carry=1)	C
<b>BEQ</b> Branch if equal to zero (Z flag=1)	Z
<b>BMI</b> Branch on minus (N flag=1)	N
<b>BNE</b> Branch on not equal to zero (Z flag = 0)	Z
<b>BPL</b> Branch on plus (N flag = 0)	N
<b>BVC</b> Branch on overflow clear (V flag = 0)	V
<b>BVS</b> Branch on overflow set (V flag = 1)	V

All the above conditional branch instructions use relative addresses only, i.e. one-byte addresses with range +129 to -126 from current instruction byte.

#### (b) Compare instructions

**CMP** Compare data with contents of accumulator.

**CPX** Compare data with contents of X index register.

**CPY** Compare data with contents of Y index register.

These instructions set the P-register flags Z and C in the following way, which can then be tested with conditional branch instructions, as noted:

Register < data: C = 0: Test with **BCC**

Register = data: Z = 1: Test with **BEQ**

Register >= data: C = 1: Test with **BCS**

Register > data: Z = 1 and C = 1: Test with **BEQ** followed by **BCS**

### (c) Bit compare

**BIT** Compare and set bits in Status

The **BIT** instruction performs a comparison between storage and the accumulator, and sets the Z flag to 1 if they are equal. Bits 7 and 6 of the memory location are transferred to the Status register (N and V flags respectively). The accumulator is unchanged.

## Arithmetic instructions

There are only two types of arithmetic instructions implemented on the 6502. These are add and subtract operations. Multiplication and division must be handled by sub-routines, and these can of course be affected by repeated addition or subtraction.

### (a) Arithmetic instructions

**ADC** Add with carry

**INC** Increment memory (add 1)

**INX** Increment register X (add 1)

**INY** Increment register Y (add 1)

**SBC** Subtract with carry

**DEC** Decrement memory (subtract 1)

**DEX** Decrement register (subtract 1)

**DEY** Decrement register Y (subtract 1)

The **ADC** and **SBC** need special explanation as both use the current value of the carry flag as part of the operation. The **ADC** adds the data at the specified address, plus the carry value, to the accumulator. The **SBC** subtracts the data minus the inverse of the carry flag (1 if carry = 0, 0 if carry = 1) from the accumulator. When starting a series of linked arithmetic operations the carry flag must be cleared before the first **ADC** and set before the first subtract. This is achieved by the following instructions:

**CLC** Clear carry flag.  
**SEC** Set carry flag.

The remaining instructions are relatively simple, and do not involve the carry flag. Increment increases the value of the register or memory by 1, decrement decreases it. Note the absence of direct accumulator increment/decrement instructions.

### (b) Decimal mode

Two other instructions which are related to arithmetic, but do not actually perform arithmetic operations are:

**SED** Set decimal mode

**CLD** Clear decimal mode (set binary mode)

Decimal? Yes, the 6502 can also perform decimal arithmetic, as you should perhaps have realised from the description of the P-register flags earlier on. Decimal arithmetic is performed on data formatted as 'packed' decimal, i.e. two binary coded decimal digits (0 to 9) per byte. It is most important, if switching between decimal and binary arithmetic, to *set* decimal mode before using decimal arithmetic, and to *clear* decimal mode before attempting binary arithmetic. Try adding two **BCD** numbers in pure binary and check the result (make both numbers greater than 5).

## Logic instructions

There are three logic instructions which operate at the bit level. They are used to test combinations of bit patterns and set a flag to indicate the result. The result is always either TRUE or FALSE, and to help you understand how they operate we will use a diagram called a truth table. The instructions are:

**AND** Logical AND of data and accumulator

**ORA** Inclusive OR of data and accumulator

**EOR** Exclusive OR of data and accumulator

### AND

Let us first examine **AND** with the following truth table. The symbol  $\wedge$  is used to represent **AND**.

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

A is an accumulator bit and B is the corresponding data bit. A  $\wedge$  B is spoken 'A AND B', and is the logical result of the operation. A zero result is FALSE, and a 1 is TRUE.

It can be seen from the table that the only combination of accumulator and data bit values that yields a TRUE result is the last – when both bits are 1. All other combinations yield 0 – a FALSE result.

The accumulator bits are affected and contain the results for A  $\wedge$  B. This instruction is generally used to turn off (i.e. set to 0) specific bits in a byte.

Let us assume that in the accumulator bit 0 was being used as a programmed flag (or switch) and we wanted to zeroise it without affecting any other bits. We would achieve this by ANDing the accumulator with binary 1111110 (hex FE).

Bits: 7 6 5 4 3 2 1 0

1	1	0	1	0	0	0	1
---	---	---	---	---	---	---	---

Accumulator before operation

AND

1	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---

Data (mask)

Result

1	1	0	1	0	0	0	0
---	---	---	---	---	---	---	---

Accumulator after operation

Check the result at each bit position against the truth table. Note that *only* bit 0 has been modified. We use the term 'mask' to describe data which is being used solely to modify the contents of the accumulator.

## ORA

The ORA instruction is used to turn bits on (i.e. set them to 1). Look at the following truth table for ORA:

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

In this case if either (or both) bits in A and B are 1, the result is 1. Using the same examples as before, we now wish to set bit 0 on again, i.e. back to 1.

Bits: 7 6 5 4 3 2 1 0

1	1	0	1	0	0	0	0
---	---	---	---	---	---	---	---

Accumulator before operation

ORA

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

Data (mask)

Result

1	1	0	1	0	0	0	1
---	---	---	---	---	---	---	---

Accumulator after operation

The symbol  $\vee$  represents OR. We can refer to the operation as 'accumulator  $\vee$  mask'.

### EOR

The **EOR** or exclusive or provides a TRUE (i.e. 1) result if one and only one of the two corresponding bits is 1, i.e. if one bit is 1 and the corresponding bit is 0. Here's the truth table:

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	0

$\vee$  is the symbol used for exclusive OR. This instruction is used to turn bits on if they are off, and off if they are on, i.e. to change the state of a bit, for use, say, as a 'toggle' switch.

Bits: 7 6 5 4 3 2 1 0

1	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

Accumulator before operation

### EOR

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

Data (mask)

Result	0	1	0	1	0	1	0	1
--------	---	---	---	---	---	---	---	---

Accumulator after operation

In this example we have modified several bits in one operation. This can also be done with **AND** and **ORA**. Try **EOR** on the accumulator above with a mask of **00000000** (hex **00**).

### Control instructions

These instructions, with the exception of **NOP**, **CLI** and **SEI**, are used in modifying the sequence of a program, and in altering the flow of control.

<b>NOP</b>	No OPeration (do nothing)
<b>JMP</b>	JuMP or unconditional branch
<b>JSR</b>	Jump to SubRoutine
<b>RTS</b>	ReTurn from Subroutine
<b>PHA</b>	PusH A onto stack
<b>PHP</b>	PusH P onto stack
<b>PLA</b>	PulL A from stack
<b>PLP</b>	PulL P from stack
<b>BRK</b>	BReaK
<b>RTI</b>	ReTurn from Interrupt
<b>SEI</b>	SEt Interrupt disable flag
<b>CLI</b>	CLear Interrupt disable flag

(a) NOP

This is a one-byte instruction which is used to delay the execution of a routine (by two cycles) or to patch-out instructions, if you wish to block a section of code. As you might have gathered it simply does nothing, and is extremely useful. When used to patch-out instructions, it must be used to replace every byte of the instruction, so that two or three NOP instructions may be required.

(b) JMP

This is the machine-code equivalent of a BASIC GOTO. The sequence of control is transferred directly to the specified address.

(c) JSR and RTS

These are extremely important instructions in the 6502 instruction set, since **JSR** causes a jump to a routine elsewhere in the program, and the **RTS** instruction provides a return to the instruction following the jump. They are directly equivalent to the BASIC GOSUB and RETURN instructions. This is made possible by the way the Stack operates, and we should now discuss the Stack.

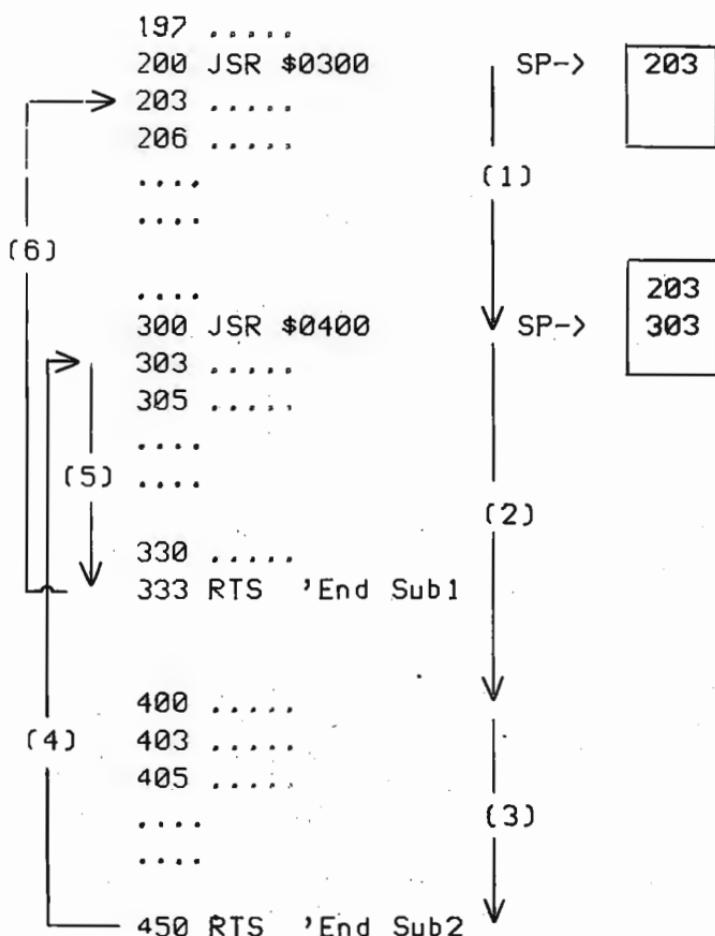
The Stack is a structure composed of things piled one on top of another. The things in this case are addresses, taken from the Program Counter (PC) Register. Why do we need a Stack, and how do we use it? Let's consider an example. Overleaf is a possible control flow in a machine-code program.

Let us consider the following example. A **JSR** instruction at location \$0200 calls a subroutine at location \$0300. At the time of the call the PC-Register contains \$0203. The address of the instruction following the **JSR**. The 6502 will place this address in the stack and then jump to location \$0300. At location \$0300 another **JSR** is executed, to a subroutine at \$0400. The PC8 contents (\$0303) now have to be saved, and are placed after the first address in the stack.

Subroutine \$0400 executes to completion, and terminates with an **RTS** instruction. This causes the last address placed in the Stack to be loaded into the program counter (\$0303) and a jump to the PC address to be taken to the instruction following the second **JSR**. Similarly, when subroutine \$0300 terminates with an **RTS**, the PC is loaded with \$0203 and execution continues from this address.

Only one question remains – how is the right address selected from the Stack? The answer lies in the Stack Pointer Register (s) which always points to the next available Stack location, and is automatically up-dated (forwards) by the **JSR** and (upwards) by the **RTS** instructions.

Finally, the 6502 Stack resides in page 1 of memory, i.e. hex 100 to hex 1FF. Since the Stack Pointer is an 8-bit register the leading 1 in both addresses is implied – that is, the 6502 will always assume an extra bit before the address. Also, the 6502 Stack is upside down, i.e. the first available



Stack location is at the bottom (High Address) of the Stack, and the Stack Pointer is decremented, on each **JSR**, by 2 (Addresses stored are absolute, 16-bit, from the PCR).

The example given above demonstrated two nested **JSR**'s, with one **JSR** issued from within the first call (**JSR'ed**) routine. Since the Stack is 256 bytes long (1 page), and the PC two bytes long, 128 levels of nesting are possible. In practice, it is most unlikely that this degree of nesting will be used, so page 1 locations below the agreed low address may be used by the program if required.

### Push and Pull instructions

These four instructions place and retrieve Accumulator and P-Register (Status bits) on and from the Stack. It is sometimes necessary to save and restore the P-Register status flags so they are as they were before a subroutine call. The **PHP** instruction will place the P-Register contents on the Stack and update the Stack Pointer. The **PLP** will restore the P-Register from the Stack address indicated by the Stack Pointer, and then update the Stack Pointer.

The Accumulator contents may be similarly saved and restored with the **PHA** and **PLA** instructions. Remember that the **RTS** takes the last two bytes from the top of the Stack, and ensure that Pushes and Pulls are ordered so that the **RTS** does not get erroneous information by trying to branch to the contents of the saved Accumulator or P-Register.

### Interrupts

A complete discourse of Interrupts is beyond the scope of this chapter, and we will therefore skim very lightly over the subject. The reader is referred to any of several available reference texts on the 6502, if he feels the need to explore this area any further.

Briefly, the 6502 has two types of interrupts: hardware and software. When an interrupt occurs the contents of the PC-Register and the P-Register are deposited on the Stack, and a Jump is taken to an indirect address stored in high memory. These addresses are known as Vectors, and the process as Vectoring. The subroutine at the final address services the interrupt, and then returns to the point at which normal execution was interrupted via an **RTI** instruction which restores the P-Register and PC-Register from the Stack and updates the Stack Pointer.

The **BRK** instruction is a software interrupt instruction, which causes a Jump to the indirect address at locations FFFE, FFFF. These addresses are normally in Oric ROM (Read Only Memory) and therefore not directly modifiable. We will leave **BRK** at this point, and return to it later.

The last two instructions are the **SEI** and **CLT**. These instructions are

used to Set or Clear the Interrupt disable flag in the P-Register. When set, all interrupts are disabled, i.e. ignored, with the exception of one type of hardware interrupt – the **NMI**, appropriately, Non-Maskable Interrupt. Disabling of interrupts is necessary when it is essential to complete a process in a specific time, usually very small, because the process is itself time-critical.

### Machine Code programming conventions and rules

We have now completed the 6502 instruction set, and are ready to start programming. A few conventions will assist us.

We introduced the \$ earlier on to indicate hex data. We now introduce the # to indicate a literal, i.e. an Immediate operand, and % to denote binary.

Machine-code programming has two stages — the symbolic and the code-level stages. For example:

LDA #\$00	in symbolic – Load hex Literal 00 to Accumulator
A9 00	is code level (LDA immediate=hexA9)

We normally develop the program at symbolic level, and then translate it to code. This is called *assembly*, and, if done manually, hand assembly.

### IMPORTANT

Up to this point we have written all two-byte hex addresses as \$hh11, where hh is the High Order byte and 11 the Low Order byte. This is satisfactory and desirable at the symbolic level. However, the 6502 expects 2-byte addresses in reverse sequence and machine code must be written this way. Thus we code an address as \$llhh. For example:

LDA \$103E (Load Accumulator from Absolute Address 103E)

would be coded as:

AD 3E 10 (LDA Absolute Op Code = AD)

This must be remembered when translating from symbolic to code forms. Now let us start programming, using some useful machine-code routines as examples.

### Machine code examples

Most programs will use some arithmetic, and it therefore seems appropriate to commence with examples of arithmetic routines. We'll start with the addition of two single byte binary numbers, located at hex 500 and hex 501,

the result to be stored in hex 502, assuming that the sum can be stored in a single byte.

CLC	Clear Carry Flag
LDA \$500	Load First number into Accumulator
ADC \$501	Add second number into Accumulator
STA \$502	Store result

Easy, wasn't it? Note the use of CLC before the ADC.

Now let us try it again, this time with two 3-byte binary numbers, located at \$501 – \$503, \$504 – \$506, storing the result at \$507 – \$509, and assuming, as before, that the result will fit into 3 bytes. We could repeat the last three instructions for each byte to be added, but this is inefficient. We will use indexing instead. We will also assume, for this example, that the numbers are being stored in memory with the high-order bit first (i.e. lowest address), in the conventional manner. It could be stored the other way around (low-order byte first) in the same way as two byte addresses – this is up to the programmer. Here is the program:

#### Address Instruction:

600	CLC	Clear Carry flag
601	LDX #\$03	Load X with 03 (literal)
603	LDA \$500, X	Load From Address \$500 + X
606	STA \$503, X	Add from Address \$503 + X
609	STA \$506, X	Store result Address \$506 + X
60C	DEX	Decrement X by 1
60D	BNE * -12	Go back to \$603 if X<>0

Both programs shown have been written in a symbolic form – they will need translation into machine code before they can be used. The # indicates that the following characters constitute a literal operand, which will be stored as part of the instruction, in immediate address form. The use of "X" indicates that the address preceding is to be indexed by register X – i.e. the final address for data is the sum of the first address plus the contents of register X. The "\*" in the last instruction means the contents of the PRogram Counter after the instruction has been fetched and decoded, but before it has been executed. Now let us look at the machine code.

#### Address Mode

600	18	Implied
601	A2 03	Immediate
603	BD 00 50	Absolute, X
606	7D 03 50	Absolute, X
609	9D 06 50	Absolute, X
60C	CA	Implied
60D	D0 F4	Relative

To understand the logic of the process, we could consider the problem expressed in diagrammatic form, using the flow-charting method. Such a flow-chart should be constructed before any attempt at coding, as it simplifies the visualisation of logic requirements, and therefore the programming task. It is surprising how much time this can save. Refer to any general text on programming if you are interested.

Note the use of index decrementing to address data bytes in ascending order. The following may make it clearer:

Address	Index X	Final Address
500	3	503
500	2	502
500	1	501

If the number \$12345 was stored in locations \$501 – \$503, the low order byte (\$45) would be taken first, then the middle byte (\$23), then the high order byte (\$1), which is the correct order of addition.

If we stored numbers in the same way as addresses (compare the instructions at location \$603 in the symbolic and machine code examples), we would need to increment (increase by 1) the register – number \$12345 would be stored as \$54321 – and test (with a compare) for an index value of 3. The method shown saves one instruction and is easier to follow. Appendix 8 contains a table of instructions or Op codes, in each addressing mode.

Subtraction routines can be written in the same way, using the addition routines as examples. Remember to use SEC (Set Carry) in place of the CLC, though. An alternative method is to use 2's-complement and perform subtraction by the addition of a negative number.

Multiplication and division can also be performed by repetitive addition. However, it is more efficient to use shifted addition methods. You should refer to one of the standard texts such as Leventhal or Zaks for suitable routines.

Since this is an Oric companion, we will next consider routines that are designed for this machine specifically. Screen handling is a good area to start with so we will develop a routine to move a character around the screen, using keyboard control. There are two ways of doing this – by directly accessing the screen area, and using Oric's own routines by calling them from our machine-code program. Keyboard access will use the latter method for simplicity.

This routine will use the Oric arrow keys to move a character around the screen in the direction of the arrow key. We will restrict character movement to rows 2 – 25 and columns 2 to 39. (Avoiding the first two lines and the attribute columns).

The TEXT screen consists of 27 rows of 40 columns, although the leftmost

columns are reserved for colours and are not usually used in TEXT or LORES modes.

The screen occupies memory locations \$BB80 to \$BFE0 (48K machines) or \$3B80 to \$3FE0 (16K machines). The program will assume a 48K machine. Users of 16K machines must subtract hex 8000 from all screen addresses to use the routine.

```

100 HIMEM 32767:REM $7FFF
101 :
110 FOR X=0 TO 113
120 : READ CODE
130 : POKE 32768+X, CODE
140 NEXT X
150 :
160 CLS:PRINT "CURSOR KEYS TO MOVE OR S
PACE TO END"
170 POKE 0,#D0:POKE 1,#BB:REM SCREEN
180 POKE 2,#02:POKE 3,#02:REM COL/ROW
190 POKE #24E,1:POKE #24F,1:REM KBD
200 :
210 FOR X=1 TO 10
220 : CALL #8000
230 NEXT X
240 IF KEY$<>" " THEN GOTO 210
250 POKE #24E,32:POKE #24F,4
260 STOP
1000 REM OP ADDR      SYMBOLIC ADDR
1010 DATA #D8          'CLD      8000
1020 DATA #A9,#20       'LDA #$20   1
1030 DATA #A4,#02       'LDY COL    3
1040 DATA #91,#00       'STA ($00),Y 5
1050 DATA #20,#3B,#02    'JSR GTORKB 7
1060 DATA #10,#51       'BPL DISPLAY A
1070 DATA #A4,#02       'LDY COL
1080 DATA #C9,#08       'CMP #$08
1090 DATA #90,#4B       'BCC DISPLAY
1100 DATA #F0,#0D       'BEQ LEFT
1110 DATA #C9,#0B       'CMP #$11
1120 DATA #F0,#19       'BEQ UP

```

```
1130 DATA #B0,#43      'BCS DISPLAY
1140 DATA #C9,#09      'CMP #$09
1150 DATA #F0,#0B      'BEQ RIGHT
1160 DATA #4C,#41,#80   'JMP DOWN
1170 DATA #C0,#02      'LEFT CPY #$02
1180 DATA #F0,#38      'BEQ DISPLAY
1190 DATA #88          'DEY
1200 DATA #4C,#5D,#80   'JMP DISPLAY
1210 DATA #C0,#27      'RIGHTCPY $$27 (39)
1220 DATA #F0,#30      'BEQ DISPLAY
1230 DATA #C8          'INY
1240 DATA #4C,#5D,#80   'JMP DISPLAY
1250 DATA #A6,#03      'LDX ROW
1260 DATA #E0,#02      'CPX #$02
1270 DATA #F0,#26      'BEQ DISPLAY
1275 DATA #C6,#03      'DEC ROW
1280 DATA #EA,EA        'NOP,NOP
1290 DATA #20,#64,#80   'JSR SUB
1300 DATA #4C,#5D,#80   'JMP DISPLAY
1310 DATA #A6,#03      'DOWN LDX ROW
1320 DATA #E0,#1B      'CPY #$1B
1330 DATA #F0,#18      'BEQ DISPLAY
1335 DATA #E6,#03      'INC ROW
1340 DATA #A9,#28      'LDA $$28 (40)
1350 DATA #20,#51,#80   'JSR ADD
1360 DATA #4C,#5D,#80   'JMP DISPLAY
1370 DATA #18          'ADD CLC
1380 DATA #65,#00      'ADC $00
1390 DATA #85,#00      'STA $00
1400 DATA #A9,#00      'LDA $$00
1410 DATA #65,#01      'ADC $01
1420 DATA #85,#01      'STA $01
1430 DATA #60          'RTS
1440 DATA #A9,#58      'DISPLAYLDA $$58 "X"
1450 DATA #91,#00      'STA ($00),Y
1460 DATA #84,#02      'STY COL
1470 DATA #60          'RTS
1480 DATA #38          'SUB SEC
```

```

1490 DATA #A5, #00      'LDA $00
1500 DATA #E9, #28      'SBC #$28
1510 DATA #85, #00      'STA $00
1520 DATA #A5, #01      'LDA $01
1530 DATA #E9, #00      'SBC #$00
1540 DATA #85, #01      'STA $01
1550 DATA #60           'RTS

```

The following is an explanation of the program:

Line 100 sets the upper limit of memory for BASIC programs. In this example we have allowed about 6K for our machine-code programs – in fact only 114 bytes are used and the figure was chosen to allow the program to start at hex 8000. In practice, where machine-code routines are being used with BASIC programs it is preferable to subtract the size of the machine-code routines from the current HIMEM position, this being obtained by a PRINT DEEK(#A6), and then using HIMEM as shown to set the limit for BASIC.

Lines 110 to 140 constitute a loop which extracts the machine code from the DATA statements and loads it to the specified addresses. This routine is called a LOADER, and is the conventional way of using BASIC to load machine-code routines.

Line 160 clears the screen.

Line 170 sets the screen start address (\$BBD0 in our example) into known Page 0 locations (hex 0,+1). Note the “backward” orientation of the address.

Line 180 sets column and row minima in Page 0 locations 2 and 3.

Line 190 sets the keyboard delay and repeat rates to the fastest speeds. These are system constants for the Oric operating system, and more of these will be found in the Appendix 9.

Lines 210 to 230 form a loop to call our machine-code subroutine ten times. Note that # in BASIC means Hexadecimal, but is used in Assembler or Symbolic code as a literal sign. Line 240 tests for a space-bar character, which we use in this routine to go back to our main program. When found, line 250 resets the keyboard delay and repeat to the original values of 32 and 4, and halts execution of the program.

We now come to the machine-code routine itself. The routine has been coded with one instruction per DATA statement, for clarity, and the symbolic code (or Assembly Language code) has been included in the same line as a comment.

Lines 1010 to 1160 test the incoming keyboard character for cursor control codes (i.e. the arrow codes), and take a branch to the appropriate routines (labelled LEFT, RIGHT, UP and DOWN) when found, exiting to the DISPLAY routine at the end of the program if any other character is entered.

The first few instructions set the arithmetic mode (**CLD**), and clear the previously displayed character ("X") from the screen. Line 1040 uses indirect Y addressing to store a blank over the previous "X" by indexing the screen pointer in hex 00+01 by the column position in hex 02. Line 1050 calls the system keyboard routine (details in Appendix 9) to read the keyboard and return the input character in register A.

Line 1010 reloads the column position in register Y for subsequent use.

Line 1170 to 1240 are the LEFT and RIGHT routines, which decrement or increment the column position by 1, after testing for the minimum and maximum column values.

Line 1250 to 1360 are the UP and DOWN routines. These modify the Row address and test for minimum and maximum allowable values. If within limits the routines call the SUBTRACT and ADD routines with a **JSR** instruction (lines 1370 to 1430 and lines 1480 to 1550 respectively), which modify the screen address in locations 00+01 by a value of decimal 40 (characters per line). On return from these routines a **JMP** is taken to the final routine in the program, DISPLAY.

Lines 1440 to 1470 constitute the DISPLAY routine. This routine loads register A with ASCII "X", then puts it on the screen – via an indirect – Y **STA** instruction, where Y (the column) modifies the screen address (in hex 00, 01). The (updated) value of Y is then re-stored in hex 02 and the program exits via an **RTS** to the CALLING BASIC program at line 230.

Note line 1280 – two **NOP** instructions have been used to patch out another instruction. If **NOP** was not available all instruction addresses below this statement would have had to have been modified by 2 – very time-consuming. Note the use of relative addressing in the Branch instructions – try working them out using the symbolic addresses as a guide.

Now key in the program and use the arrow keys to move the "X" around the screen.

## CALL

The previous machine-code program was invoked by the **CALL** command in BASIC. This is the simplest method of entering a machine-code program and involves nothing more than setting the address of the program in the **CALL** statement, in either decimal or hex (preceded by #). There are other ways of invoking machine-code logic, but the main advantage of the **CALL** is that several machine-code routines can be easily accessed, by re-using the **CALL** with different addresses.

## !(SHRIEK)

This operator allows new BASIC commands (written in machine code) to be defined and used. The address of the machine-code routine must first be

placed in locations \$2F5 and \$2F6 via a DOKE instruction. The command is then executed by entering ! followed by any parameters needed by the machine-code routine. The system will store the parameters in the input buffer when the ! is encountered. To access the parameters it is necessary to access the input buffer, which occupies locations hex 35 to hex 84. The logic required to do this must include all syntax checking and parameter validation, and can therefore be fairly complex. To avoid some of this complexity it is possible to use a system routine located in Page Zero hex E2, which returns the contents of the input buffer in the A Register one character at a time. JSR \$00E2 will accomplish this.

A third method is to use the address of the input buffer in locations \$E9 and \$EA, indexed by (say) Register Y, to obtain each character in turn, incrementing Y each time, using an indirect - Y LDA instruction.

If multiple ! commands are required, remember to precede each with DOKE #2F5, address-of-routine.

### & (AMPERSAND)

The ampersand symbol (&) may be used to define additional functions for use within a BASIC program. The function definition must be written in machine code, and the start address of the definition routine DOKED into location #2FC before the function is called. The syntax requires an argument within brackets following &, and this value is placed in the floating point accumulator, for use by the routine. The following example provides a function which returns the current cursor row:

```

10 REM** DEFINE & TO GIVE CURRENT
20 REM** ROW OF CURSOR
30 FOR I=0 TO 5
40 READ D$:POKE #400+I,D$
50 NEXT
60 DOKE #2FC,#400 'Start address
65 REM load Y with CURROW value
70 DATA #AC,#68,#02
75 REM enter BASIC routine to get
single byte into F.P. Accumulator
80 DATA #4C,#B6,#D4

```

This provides a means of accessing the cursor row (CURROW) system variable, with the floating point accumulator holding the returned value. & (0) will return this value from within BASIC. The floating point accumulator occupies six bytes from #D0 to #D5. Numbers are stored with the first byte holding the exponent value plus 128. Four bytes hold the mantissa,

high byte being #D1, and the most significant bit representing  $2 \uparrow -1$ , and the least significant bit of #D4 holding  $2 \uparrow -32$ . #D5 is a sign byte, with 0 representing positive, and #FF negative. The value is thus held as mantissa \*  $2 \uparrow$  exponent. Zero has a unique representation, with the exponent set to zero.

The following program will correct the problem with some cassette recorders referred to in Chapter 6. The program loads into locations #281 to #2BF, avoiding #2A9 to #2B0. When CLOADED it copies another program into locations #221 to #22A, and changes the fast interrupt vector (at #245 and #246) to point to it. This zeroes the error flag at #2B1 each time an interrupt occurs. Unless auto-search is inhibited the following file on tape is then loaded.

```

10 REM**Cassette Loading Bogue Error
15 REM**Rectification Program
20 FOR D=0 TO 39
30 READ N$:POKE #281+D,N$
40 NEXT
50 FOR D=0 TO 13
60 READ N$:POKE #2B1+D,N$
70 NEXT
80 PRINT "SET CASSETTE TO RECORD, PRESS
A KEY. PROGRAM NAME IS '*' :GET A$"
90 CSAVE "*",A#281,E#2BF,AUTO 'add ,S fo
r slow save
100 DATA #08,#78,#AD,#F9,#FF
110 DATA #C9,#01,#D0,#28
120 DATA #AD,#B6,#E4,#C9,#A2
130 DATA #D0,#15,#A0,#09
140 DATA #B9,#B5,#02,#99,#21,#02
150 DATA #88,#10,#F7,#A9,#21
160 DATA #8D,#45,#02,#A9,#02
170 DATA #8D,#46,#02,#4C,#67,#E8
180 :
190 DATA #00,#4C,#B6,#E7,#48
200 DATA #A9,#00,#8D,#B1,#02,#68
210 DATA #4C,#22,#EE
220 REM**Inhibit auto-search for next
      **file by POKEing #60
230 REM**into locations #2B2 and #2A6

```

# 11 Input/Output

The Oric has flexible input/output facilities, which enable the enthusiast with some experience of simple electronics to control and monitor external events using fairly straightforward interfacing techniques.

The memory map given in Appendix 5 provides the information necessary to perform such interfacing. There are two usable areas:

1. Top of Page 3 memory, locations from #03FF downwards.
2. Bottom of Page 3 memory #0300 to #030F which in fact contain the 16 registers, output ports, etc. of the 6522 VIA (Versatile Interface Adaptor) chip inside the Oric. Of the 16 pin-outs, 8 provide the eight bits of the printer interface and are also multiplexed to the sound chip. Certain of the remaining 8 connect to the keyboard interface. This chip is responsible for a multitude of activities and hence must be used with care.

The area specified in 1 above can be configured for interfacing by using the expansion bus, which gives access to the necessary control pin-outs. The memory locations in 2 above can be interfaced using the printer port.

Any memory location in the spare memory area between #BFE0 and #BFFF could be used as output or input by means of the appropriate POKE or PEEK command. External devices configured to any location would be simply seen as a memory address which can be read from or written to.

This is RAM area which is disabled by taking the MAP pin low (MAP).

It should be pointed out that  $\overline{MAP}$  will disable the internal ROM, should it be accessed at the same time. This feature is used by the Oric MICRODISC drives, and the complexity of I/O using this area, as well as a possible clash with Oric peripherals, makes this area unsuitable for our purposes. Although we have confined our discussion to the area between #BFE0 and #BFFF any RAM area, except #0300 to #030F, could be disabled by  $\overline{MAP}$  and replaced by external memory or other suitably interfaced devices.

### **Top of Page 3 memory: #03FF downwards**

This area is probably the easiest to interface and uses the  $\overline{I/O}$  pin as an output signal low and, providing the top of Page 3 is being used, i.e. locations above #030F, also uses the  $\overline{I/O\ CONTROL}$  signal which, as before, needs to be pulled low ( $\overline{I/O\ CONTROL}$ ), in order to disable the internal 6522 VIA chip.

The  $\overline{I/O}$  pin automatically goes low whenever an address in the range #0300 to #03FF is being accessed using a POKE or PEEK command, but addresses #0300 to #030F are used by the internal 6522 VIA, for output to the printer and sound chip, so that disabling the 6522, when used in this range, by means of the  $\overline{I/O\ CONTROL}$  input pin defeats the purpose of addressing these locations. Hence,  $\overline{I/O\ CONTROL}$  should be applied in the range #0310 to #03FF only for the purpose of interfacing to external devices. These locations should be utilised from location #03FF downwards for user I/O since Oric peripherals will be designed to use addresses #0300 upwards. In this way any conflict will be postponed until the last possible moment!

As before the  $R/\overline{W}$  (READ/ $\overline{WRITE}$ ) pin is available for gating/decoding, being low on POKE (i.e. writing to) and high on PEEK (i.e. reading from). Typical circuit examples are given in Appendix 10.

### **Bottom of Page 3 memory: #0300 to #030F**

This printer interface area is in some respects the trickiest area to use, but also the most challenging. These memory locations actually control the functions of the 6522 VIA internal to the Oric, which, given its design, are many and varied.

Firstly, the memory locations #0300 to #030F are the 16 controlling registers of the VIA. To go through them all is not necessary (thankfully), but interested readers are referred to data sheets from the manufacturers or supplier.

We need to know the functions of the first four locations (i.e. 4 of the 6522 internal registers) and the last one, #0300, #0301, #0302, #0303 and #030F. The 6522 chip has two sets of 8 data lines, called port A and port B.

These data lines are read from or written into through #0300 for port B and #0301 for port A, ORB and ORA respectively. However, initially it is necessary to specify whether the data lines are to be read from (i.e. used as input) or written into (i.e. used as output). This is done by writing to #0302 for port B and #0303 for port A, DDRB and DDRA respectively. These stand for Data Direction Registers A or B, Direction in this case meaning input or output. The 7 lines of port B (B0, B1, B2, B3, B5, B6, B7) are configured internally, whilst B4 is the STROBE pin on the printer interface.

All 8 lines of port A (A0, A1, A2, A3, A4, A5, A6, A7) are brought to the outside and form the data pins of the printer interface (pins 3, 5, 7, 9, 11, 13, 15, 17 respectively).

Before using the printer port for output the following must be noted. Usually the 6522 port A is a latching output port, i.e. once the data is output to port A it stops there. However, the Oric's internal 6522 is used for many functions and other data transfers occur which use port A, since this port is also multiplexed to the sound chip, which in turn is connected to the keyboard interface. Therefore simply POKEing a data byte to port A, will *not* latch this data, and latching has to be done externally. There is a STROBE pin on the printer socket, which is in fact line B4 of port B, used by the VIA to tell a printer that data is ready. This pin provides a negative pulse (it is normally high) and can be used as the latching pulse to latch data on to the outputs of an external device at the appropriate time. In addition there is also the I/O pin on the expansion bus, which will go low when the VIA is accessed, since the VIA is mapped on to Page 3 of memory. This too can be used as a latching pulse, producing, as with B4, a high-to-low transition. Typical latches which can be used are shown in Appendix 10.

The software to perform an output via port A of the internal Oric 6522 can be as follows.

Firstly the LPRINT command, LPRINT CHR\$(i), will output the binary pattern corresponding to the value of i. If i=1, then the bit pattern will look like:

0 0 0 0 0 0 1
D 7 . . . . D 0

and pin 3 of the printer interface socket will go high (5V).

If i=4, the bit pattern will look like:

0 0 0 0 0 1 0 0
D 7 . . . . D 0

with pin 7 of the printer socket going high, and so on.

This command is actually a machine-code subroutine which outputs data and a STROBE signal at the appropriate time, when data is on the lines.

If however, the POKE command is used then data has to be POKEED to location #0301, as already mentioned, through which it transfers to port A.

Thus we need to POKE #0301, i , where i lies between 0 and 255.

The problem is the latching pulse. B0 to B7 are controlled through ORB, i.e. #0300. B4 (the STROBE) is *normally* high, so outputting a 0 to B4 will produce the STROBE:

POKE #0300, i

where i =

	x	x	x	0	x	x	x
	D	7	.	.	.	.	D
	0						

For example, with i as 175 we will have B4 equal to 0. The sequence then becomes:

POKE #0301, i (output data)  
POKE #0300, 175 (latch)

However, it appears necessary to interpose a time delay between the STROBE and the output latch to provide reliable operation. As an alternative to the above sequence, the POKE #0301, i statement accesses Page 3, therefore  $\overline{I/O}$  will go low at the appropriate time, and this can be used as a latching pulse.

For an input into the printer port A, the 6522 must first be told that port A is now to receive and not transmit. Latching is not a problem here, but, again, either the  $\overline{I/O}$  or  $R/W$  pin could be used to enable a buffer chip (tri-state) during a read operation. Port A of the 6522 is programmed for output on power-on, and therefore it is necessary to program for input. This can be done by POKEing 0 into location #0303, as this location actually maps the Data Direction Register for port A (DDRA), as previously mentioned, so:

POKE #0303,0

actually sets all 8 bits of port A for input:

DDRA

	0	0	0	0	0	0	0
	D	7	.	.	.	.	D
	0						

while:

POKE #0303,3

sets bits D0 and D1 for output, and D2 to D7 for input, of port A:

DDRA 3:  
D7 . . . . . D0

	0	0	0	0	0	1	1
	input pins	output					

so the sequence of software is:

```
10 POKE#0303,0
15 REM set all bits of port A as inputs
20 PRINT PEEK(#0301)
25 REM read port A
30 POKE#0303,255
35 REM set the ORIC 6522 back as an output device
```

Line 30 is essential, otherwise the user will find that after 10 and 20 without 30, the keyboard will be disabled.

Initially, before any data is input, PRINT PEEK(#0301) will give 255, since all data lines of the 6522 are pulled high by pull-up resistors.

Information on the techniques needed for addressing and decoding data, and sample interfacing circuitry, is given in Appendix 10 and 11.

# Appendix 1

## ASCII Character Codes

### Codes 0-31

CODE	CHARACTER	Control Code
0	Null	
1	Copy	CTRL-A
2		
3	Break	CTRL-C
4	Double line printing	CTRL-D
5		
6	Keyclick	CTRL-F
7	Bell (PING) CTRL-G	
8	Backspace (Cursor left)	CTRL-H
9	Cursor right	CTRL-I
10	Line feed (Cursor down)	CTRL-J
11	Cursor up	CTRL-K
12	Clear screen	CTRL-L
13	RETURN	CTRL-M
14	Clear line	CTRL-N
15	Disable screen	CTRL-O
16		
17	Cursor	CTRL-Q
18		
19	Screen	CTRL-S
20	Caps (upper case)	CTRL-T
21		
22		
23		
24	Cancel line	CTRL-X
25		
26		
27	ESC (Escape)	
28		
29		
30		
31		
32	Space	

**Codes 33–127**

The alternate characters are produced in the LORES 1 mode. LORES 0 uses standard characters.

STANDARD CODE	CHARACTER	ALTERNATE CHARACTER	STANDARD CODE	CHARACTER	ALTERNATE CHARACTER
33	!	33	71	G	71
34	"	34	72	H	72
35	#	35	73	I	73
36	\$	36	74	J	74
37	%	37	75	K	75
38	&	38	76	L	76
39	,	39	77	M	77
40	(	40	78	N	78
41	)	41	79	O	79
42	*	42	80	P	80
43	+	43	81	Q	81
44	,	44	82	R	82
45	-	45	83	S	83
46	.	46	84	T	84
47	/	47	85	U	85
48	0	48	86	V	86
49	1	49	87	W	87
50	2	50	88	X	88
51	3	51	89	Y	89
52	4	52	90	Z	90
53	5	53	91	[	91
54	6	54	92	]	92
55	7	55	93	↑	93
56	8	56	94	£	94
57	9	57	95	©	95
58	:	58	96	a	96
59	;	59	97	b	97
60	<	60	98	c	98
61	=	61	99	d	99
62	>	62	100	e	100
63	?	63	101	f	101
64	((	64	102	g	102
65	A	65	103	h	103
66	B	66	104	i	104
67	C	67	105	j	105
68	D	68	106	k	106
69	E	69	107	l	107
70	F	70	108		108

CODE	STANDARD CHARACTER	ALTERNATE CHARACTER		CODE	STANDARD CHARACTER	ALTERNATE CHARACTER
109	m	109	■	119	w	
110	n	110	■	120	x	
111	o	111	■	121	y	
112	p			122	z	
113	q			123	{	
114	r			124		
115	s			125	}	
116	t			126		
117	u			127	DEL	127 DEL
118	v					

### Codes 128–151

When you issue a PRINT statement with a character whose ASCII code is greater than 128, PRINT will strip the most significant bit (with 128) from the character and place the remaining code directly on the screen. Note that these ‘stripped’ codes are not treated as control toggles, but directly entered as attributes. These codes can only be PRINTed using CHR\$(i) and hence only operate on the low resolution screens. (If you try using these codes on the HIRES screen the Oric will return an illegal quantity error report.) For example:

```
100 PRINT CHR$(132);CHR$(145);“ATTRIBUTE”
```

This statement will PRINT the string “ATTRIBUTE” in blue letters on a red background. Be warned that codes 138, 139, 142, 149, which generate double height characters, require two identical program lines to produce the desired effect.

### CODE RESULT

- 128 Returns black foreground (text/graphics)
- 129 Returns red foreground (text/graphics)
- 130 Returns green foreground (text/graphics)
- 131 Returns yellow foreground (text/graphics)
- 132 Returns blue foreground (text/graphics)
- 133 Returns magenta foreground (text/graphics)
- 134 Returns cyan foreground (text/graphics)
- 135 Returns white foreground (text/graphics)
- 136 Returns black foreground (text/graphics)
- 137 Returns graphics character
- 138 Returns double-height characters (text – see note above)
- 139 Returns double-height characters (graphics – see note above)

- 140 Returns flashing characters (text)
- 141 Returns flashing characters (graphics)
- 142 Returns flashing double-height characters (text)
- 143 Returns flashing double-height characters (graphics)
- 144 Returns black background
- 145 Returns red background
- 146 Returns green background
- 147 Returns yellow background
- 148 Returns blue background
- 149 Returns magenta background
- 150 Returns cyan background
- 151 Returns white background

# Appendix 2

## Escape Codes

The escape codes available on the Oric are as given below. They insert attributes into a character position of the screen display, and may be entered directly from the keyboard, using the ESC key followed by the character, or placed in a program using PRINT CHR\$(27), followed by the character in a string form (within quotes or using CHR\$).

ESCAPE @	0	Black ink
ESCAPE A	1	Red ink
ESCAPE B	2	Green ink
ESCAPE C	3	Yellow ink
ESCAPE D	4	Blue ink
ESCAPE E	5	Magenta ink
ESCAPE F	6	Cyan ink
ESCAPE G	7	White ink
ESCAPE H	8	Standard text
ESCAPE I	9	Alternate text
ESCAPE J	10	Standard Double height
ESCAPE K	11	Alternate Double height
ESCAPE L	12	Standard Flashing
ESCAPE M	13	Alternate Flashing
ESCAPE N	14	Standard Double height Flashing
ESCAPE O	15	Alternate Double height Flashing
ESCAPE P	16	Black paper
ESCAPE Q	17	Red paper
ESCAPE R	18	Green paper
ESCAPE S	19	Yellow paper
ESCAPE T	20	Blue paper
ESCAPE U	21	Magenta paper
ESCAPE V	22	Cyan paper
ESCAPE W	23	White paper

The following escape codes are concerned with screen synchronisation:

ESCAPE X      TEXT 60Hz  
ESCAPE Y      TEXT 60Hz

ESCAPE Z	TEXT 50Hz
ESCAPE {	TEXT 50Hz
ESCAPE ;	GRAPHICS 60Hz
ESCAPE }	GRAPHICS 60Hz
ESCAPE ~	GRAPHICS 50Hz
ESCAPE -	GRAPHICS 50Hz

The above codes enable the screen display to be co-ordinated with the frequency of the a.c. current supply which drives a monitor or TV used by the Oric for display, ensuring that the signals put out by the Oric provide the correct frame rates. The UK mains supply frequency is 50Hz, and that of the US and Europe is 60Hz.

# Appendix 3

## Error Messages

Error messages are produced by the Oric whenever a program halts, due to either errors in the program syntax or structure, or machine limitations. The Oric doesn't make mistakes, but there are restrictions on what it can do. The error message is followed by the number of the program line at which the error was found, if it occurs during the RUNNING of a program. This provides an invaluable aid to debugging programs, but it should be noted that the error may have a cause earlier in the program. Thus a message ?ILLEGAL QUANTITY ERROR IN 130 indicates that an expression in line 130 could not be processed because of an incorrect parameter value. The value may have been generated earlier in the program, and the error only halts the program when the incorrect value is used. TRON and TROFF may be used to trace program execution (with suitable PRINT statements to output variable values if required) when this type of problem is encountered, in conjunction with STOP instructions to provide breakpoints in the program execution. The Oric's error messages and their explanations are as given below.

?BAD SUBSCRIPT ERROR: The program tried to refer to an array element that did not exist. With default arrays, this means outside the range 0–10, and in the case of DIMensioned arrays, outside the dimensioned range, as, for example, referring to A(20,3) when a DIM A(19,3) statement was used.

?BAD UNTIL ERROR: The program encountered an UNTIL statement without having a corresponding REPEAT statement stored as the beginning of the loop.

?CAN'T CONTINUE ERROR: After a CTRL-C or STOP instruction has halted a program, CONT may only be used if no changes have been made to the program. Attempting to use CONT results in this error if any changes have been made.

?DISP TYPE MISMATCH ERROR: Instructions valid only in a single screen mode were used with the incorrect mode, such as DRAW or CHAR used when in TEXT or LORES mode, or using PLOT and PRINT when in HIRES.

?DIVISION BY ZERO ERROR: An expression involved the (impossible) task of

dividing by zero. Watch out for undefined variables, which return the value of zero.

?FORMULA TOO COMPLEX ERROR: The number of intermediate values which the Oric needs to store when interpreting and evaluating an expression can exceed the storage available, and this error results. Break down the expression into smaller sections for evaluation.

?ILLEGAL DIRECT ERROR: A statement only allowed within a program line (such as INPUT or GET) was used as a direct command.

?ILLEGAL QUANTITY ERROR: A parameter in an expression was outside the valid range. Check the keyword definitions for valid ranges, the values returned by computed expressions used as parameters, any use of INT, and the rounding processes the Oric does automatically, when adjusting computed values to integers. The error is also generated when an integer value greater than 32767 or less than -32768 is assigned to an integer variable.

?NEXT WITHOUT FOR ERROR: The program encountered a NEXT statement when no corresponding FOR . . . TO statement was stored. Either the FOR . . . TO was omitted, or the flow of control in the program jumped into a loop.

?OUT OF DATA ERROR: The program was instructed to READ non-existent DATA items, i.e. too many READ statements and/or too few DATA items.

?OUT OF MEMORY ERROR: The Oric has memory allocated to store various data, such as the BASIC program, variables, and screen. If the combined requirements of these areas exceeds the available memory, this error is produced. Note that HIMEM restricts the area available to the BASIC program and variables if it is lowered. FRE can be used to close up the space given over to string storage, which becomes larger than is required by the volume of data if many string operations are performed in a program, as strings are shuffled around. If more than 24 subroutines, or REPEAT . . . UNTIL loops, or more than 10 FOR . . . NEXT loops, are nested in a program the stack space allocated for storage of return line numbers is exceeded, and, since a particular area of memory is full, we get the same error message.

?OVERFLOW ERROR: If the Oric generates a number too large for it to handle in the course of a calculation then the number cannot be stored in the 5-byte representation used by the Oric, and this error is generated. The largest value the Oric can hold is approximately 1.7E38 and the smallest 2.93E-39.

?REDIM'D ARRAY ERROR: Any array previously dimensioned, either as a default array (11 elements per dimension) or with a DIM statement, cannot be re-DIMensioned in the course of a program.

?REDO FROM START: In response to an INPUT statement requiring a numeric

data entry, non-numeric data was entered. Control returns to the INPUT statement to allow re-entry of data.

?RETURN WITHOUT GOSUB ERROR: The program encountered a RETURN statement without having previously processed a corresponding GOSUB instruction.

?STRING TOO LONG ERROR: Maximum length of a string is 255 characters.

?SYNTAX ERROR: The instruction being interpreted has an incorrect format. This may be either punctuation (incorrect or missing) or misspelt keywords.

?TYPE MISMATCH ERROR: This error occurs when a string is assigned to a numeric variable or function, and vice versa.

?UNDEF'D STATEMENT ERROR: The program attempted to transfer control, in response to a GOTO, GOSUB or THEN statement, to a line number that did not exist.

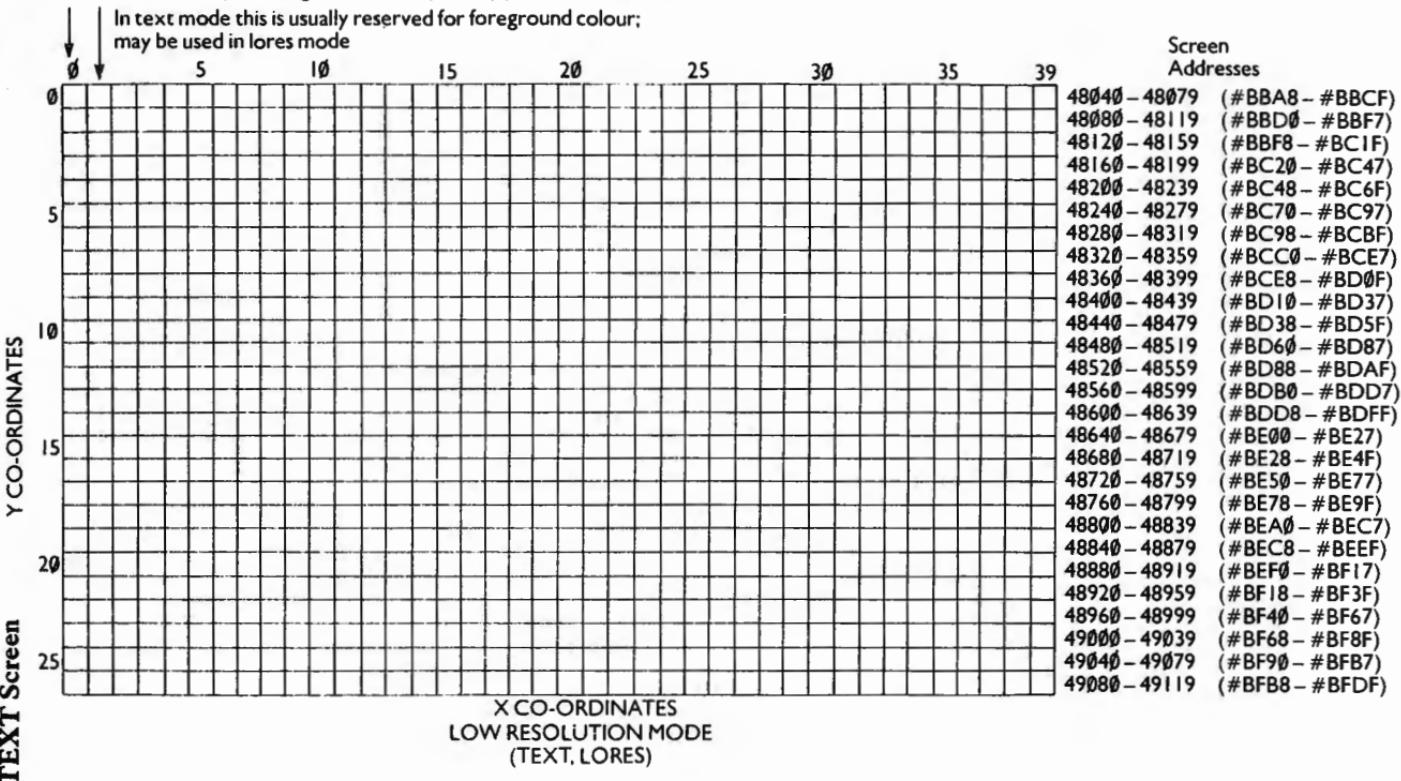
?UNDEF'D FUNCTION ERROR: An instruction to evaluate a user-defined function was encountered in the program where it had not been previously specified with a DEF FN statement.

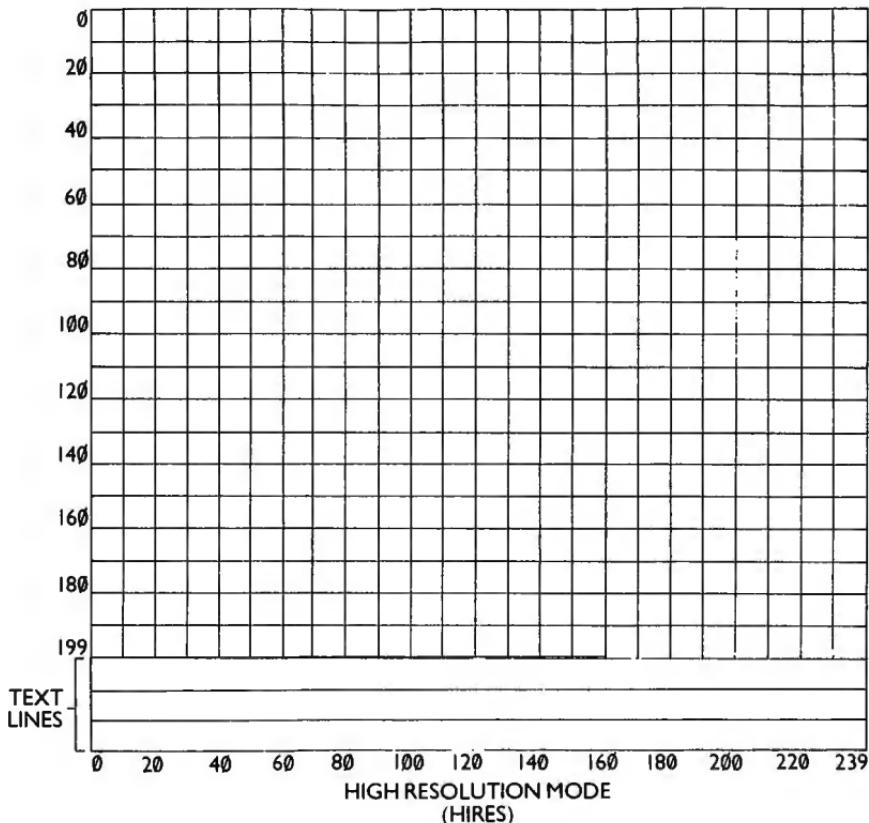
# Appendix 4

## Screen grids

Reserved column (for background colour) usually protected in both text and lores

In text mode this is usually reserved for foreground colour;  
may be used in lores mode



**HIRE S Screen****Screen Addresses**

40960 – 40999	(#A000 – #A027)
41000 – 41039	(#A028 – #A04F)
41040 – 41079	(#A050 – #A077)
41080 – 41119	(#A078 – #A09F)
41120 – 41159	(#A0A0 – #A0C7)
41160 – 41199	(#A0C8 – #A0EF)
41200 – 41239	(#A0F0 – #A117)
.	.
.	.
.	.
.	.
.	.
.	.
.	.
.	.
48880 – 48919	(#BEF0 – #BF17)
48920 – 48959	(#BF18 – #BF3F)

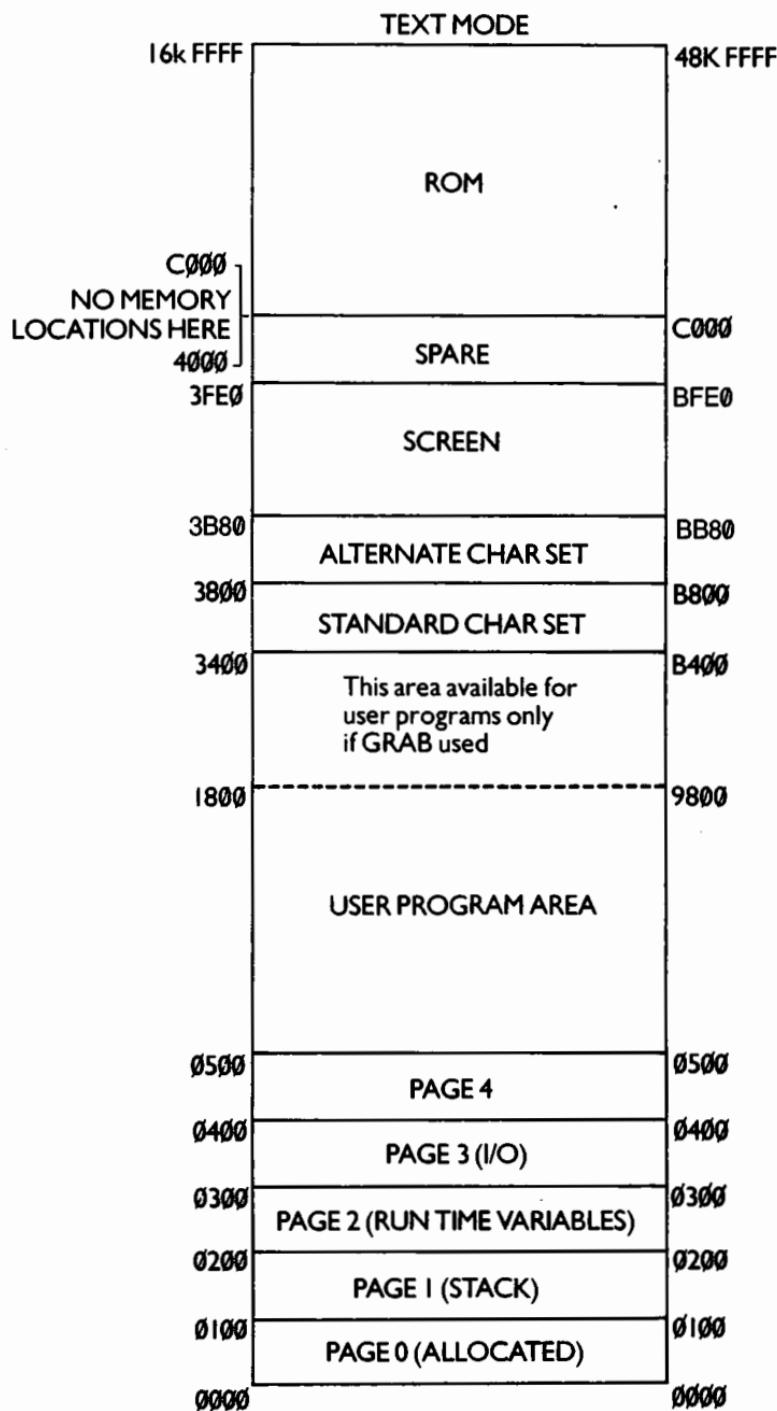
---

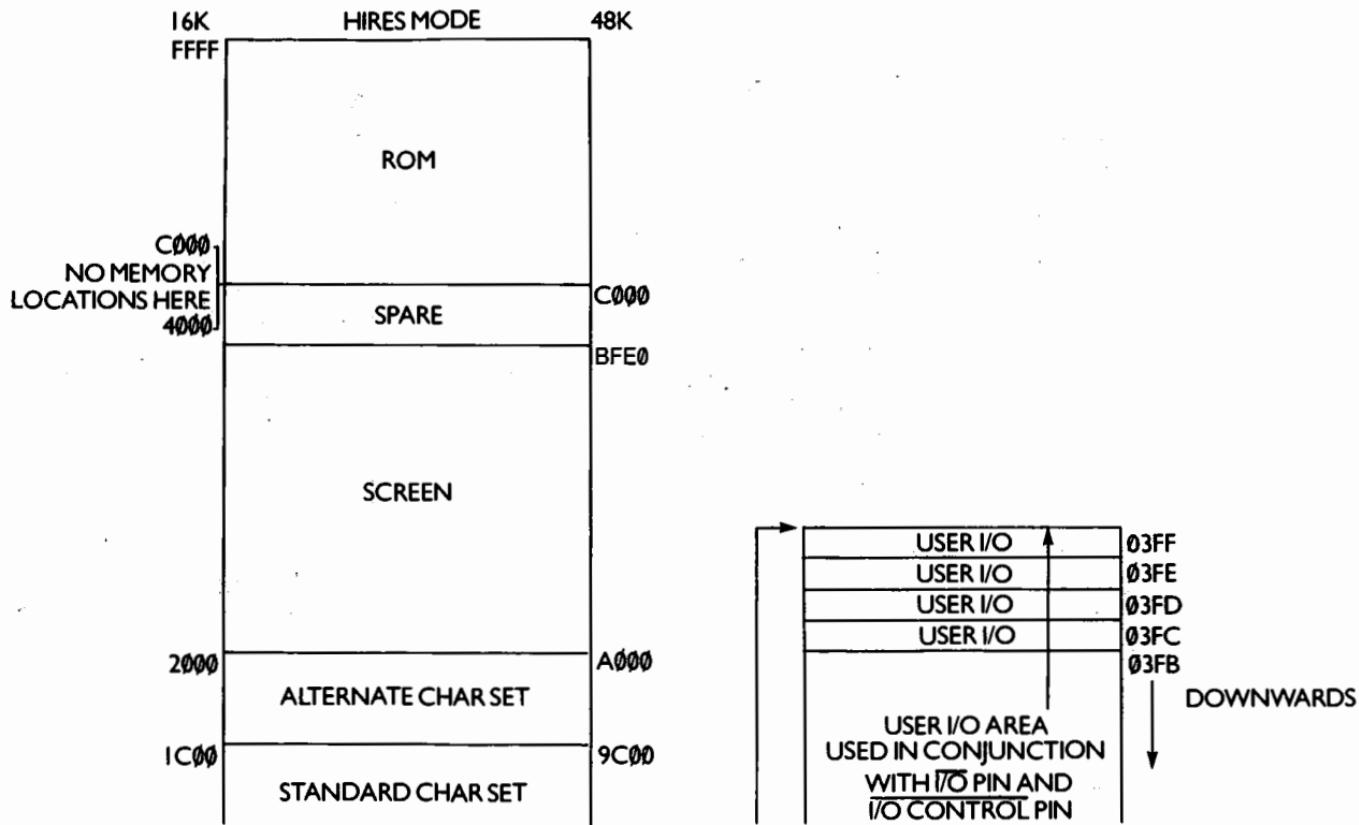
(TEXT LINES AS FOR LAST  
THREE LINES OF TEXT MODE)

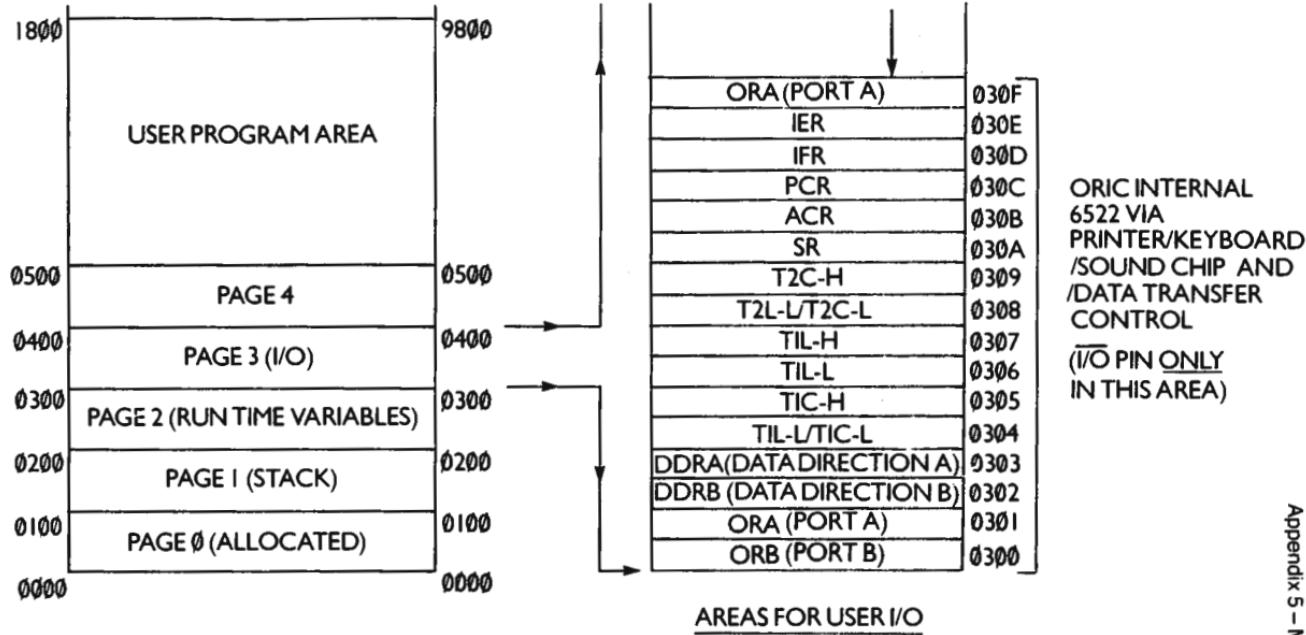
# Appendix 5

## Memory map

The memory maps given here are for TEXT and HIRES modes. All addresses are given in hexadecimal. The HIRES map has user Input/Output locations marked. These are the same in TEXT mode. The 16K RAM Oric addresses are given on the left and the 48K addresses on the right.







# Appendix 6

## Binary/hex/decimal conversions

This appendix gives, firstly, a table of the equivalent hexadecimal and binary numbers for decimal values up to 255. A table is then provided for hex/decimal/hex conversion. Any hex number is easily converted to decimal with a simple PRINT #xxxx instruction on your Oric, and decimal to hex conversion is also provided via HEX\$.

Decimal	Hex	Binary
0	#00	00000000
1	#01	00000001
2	#02	00000010
3	#03	00000011
4	#04	00000100
5	#05	00000101
6	#06	00000110
7	#07	00000111
8	#08	00001000
9	#09	00001001
10	#0A	00001010
11	#0B	00001011
12	#0C	00001100
13	#0D	00001101
14	#0E	00001110
15	#0F	00001111
16	#10	00010000
17	#11	00010001
18	#12	00010010
19	#13	00010011
20	#14	00010100
21	#15	00010101
22	#16	00010110
23	#17	00010111
24	#18	00011000
25	#19	00011001
26	#1A	00011010
27	#1B	00011011

Decimal	Hex	Binary
28	#1C	00011100
29	#1D	00011101
30	#1E	00011110
31	#1F	00011111
32	#20	00100000
33	#21	00100001
34	#22	00100010
35	#23	00100011
36	#24	00100100
37	#25	00100101
38	#26	00100110
39	#27	00100111
40	#28	00101000
41	#29	00101001
42	#2A	00101010
43	#2B	00101011
44	#2C	00101100
45	#2D	00101101
46	#2E	00101110
47	#2F	00101111
48	#30	00110000
49	#31	00110001
50	#32	00110010
51	#33	00110011
52	#34	00110100
53	#35	00110101
54	#36	00110110
55	#37	00110111
56	#38	00111000
57	#39	00111001
58	#3A	00111010
59	#3B	00111011
60	#3C	00111100
61	#3D	00111101
62	#3E	00111110
63	#3F	00111111
64	#40	01000000
65	#41	01000001
66	#42	01000010
67	#43	01000011
68	#44	01000100
69	#45	01000101

Decimal	Hex	Binary
70	#46	01000110
71	#47	01000111
72	#48	01001000
73	#49	01001001
74	#4A	01001010
75	#4B	01001011
76	#4C	01001100
77	#4D	01001101
78	#4E	01001110
79	#4F	01001111
80	#50	01010000
81	#51	01010001
82	#52	01010010
83	#53	01010011
84	#54	01010100
85	#55	01010101
86	#56	01010110
87	#57	01010111
88	#58	01011000
89	#59	01011001
90	#5A	01011010
91	#5B	01011011
92	#5C	01011100
93	#5D	01011101
94	#5E	01011110
95	#5F	01011111
96	#60	01100000
97	#61	01100001
98	#62	01100010
99	#63	01100011
100	#64	01100100
101	#65	01100101
102	#66	01100110
103	#67	01100111
104	#68	01101000
105	#69	01101001
106	#6A	01101010
107	#6B	01101011
108	#6C	01101100
109	#6D	01101101
110	#6E	01101110
111	#6F	01101111

Decimal	Hex	Binary
112	#70	01110000
113	#71	01110001
114	#72	01110010
115	#73	01110011
116	#74	01110100
117	#75	01110101
118	#76	01110110
119	#77	01110111
120	#78	01111000
121	#79	01111001
122	#7A	01111010
123	#7B	01111011
124	#7C	01111100
125	#7D	01111101
126	#7E	01111110
127	#7F	01111111
128	#80	10000000
129	#81	10000001
130	#82	10000010
131	#83	10000011
132	#84	10000100
133	#85	10000101
134	#86	10000110
135	#87	10000111
136	#88	10001000
137	#89	10001001
138	#8A	10001010
139	#8B	10001011
140	#8C	10001100
141	#8D	10001101
142	#8E	10001110
143	#8F	10001111
144	#90	10010000
145	#91	10010001
146	#92	10010010
147	#93	10010011
148	#94	10010100
149	#95	10010101
150	#96	10010110
151	#97	10010111
152	#98	10011000
153	#99	10011001

Decimal	Hex	Binary
154	#9A	10011010
155	#9B	10011011
156	#9C	10011100
157	#9D	10011101
158	#9E	10011110
159	#9F	10011111
160	#A0	10100000
161	#A1	10100001
162	#A2	10100010
163	#A3	10100011
164	#A4	10100100
165	#A5	10100101
166	#A6	10100110
167	#A7	10100111
168	#A8	10101000
169	#A9	10101001
170	#AA	10101010
171	#AB	10101011
172	#AC	10101100
173	#AD	10101101
174	#AE	10101110
175	#AF	10101111
176	#B0	10110000
177	#B1	10110001
178	#B2	10110010
179	#B3	10110011
180	#B4	10110100
181	#B5	10110101
182	#B6	10110110
183	#B7	10110111
184	#B8	10111000
185	#B9	10111001
186	#BA	10111010
187	#BB	10111011
188	#BC	10111100
189	#BD	10111101
190	#BE	10111110
191	#BF	10111111
192	#C0	11000000
193	#C1	11000001
194	#C2	11000010
195	#C3	11000011

Decimal	Hex	Binary
196	#C4	11000100
197	#C5	11000101
198	#C6	11000110
199	#C7	11000111
200	#C8	11001000
201	#C9	11001001
202	#CA	11001010
203	#CB	11001011
204	#CC	11001100
205	#CD	11001101
206	#CE	11001110
207	#CF	11001111
208	#D0	11010000
209	#D1	11010001
210	#D2	11010010
211	#D3	11010011
212	#D4	11010100
213	#D5	11010101
214	#D6	11010110
215	#D7	11010111
216	#D8	11011000
217	#D9	11011001
218	#DA	11011010
219	#DB	11011011
220	#DC	11011100
221	#DD	11011101
222	#DE	11011110
223	#DF	11011111
224	#E0	11100000
225	#E1	11100001
226	#E2	11100010
227	#E3	11100011
228	#E4	11100100
229	#E5	11100101
230	#E6	11100110
231	#E7	11100111
232	#E8	11101000
233	#E9	11101001
234	#EA	11101010
235	#EB	11101011
236	#EC	11101100
237	#ED	11101101

Decimal	Hex	Binary
238	#EE	11101110
239	#EF	11101111
240	#F0	11110000
241	#F1	11110001
242	#F2	11110010
243	#F3	11110011
244	#F4	11110100
245	#F5	11110101
246	#F6	11110110
247	#F7	11110111
248	#F8	11111000
249	#F9	11111001
250	#FA	11111010
251	#FB	11111011
252	#FC	11111100
253	#FD	11111101
254	#FE	11111110
255	#FF	11111111

**Decimal/Hex/Decimal Conversion Table**

3	2	1		0			
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
1	4096	1	256	1	16	1	1
2	8192	2	512	2	32	2	2
3	12288	3	768	3	48	3	3
4	16384	4	1024	4	64	4	4
5	20480	5	1280	5	80	5	5
6	24576	6	1536	6	96	6	6
7	28672	7	1792	7	112	7	7
8	32768	8	2048	8	128	8	8
9	36864	9	2304	9	144	9	9
A	40960	A	2560	A	160	A	10
B	45056	B	2816	B	176	B	11
C	49152	C	3076	C	192	C	12
D	53248	D	3328	D	208	D	13
E	56344	E	3584	E	224	E	14
F	61440	F	3840	F	240	F	15

To use this table for decimal to hexadecimal conversion, take the decimal number, for example 49120. Find the largest number in the table that is less

than the number to be converted. For our example, this is 45056, in column 3 of the table. The hex digit corresponding is B, which gives us our first hexadecimal digit. Next take the value of this away from the original number.  $49120 - 45056$  is 4064. The nearest smaller number (3840) is found in column 2. This gives us F as our next hex digit. (If the next smallest number is in column 1, we give a zero value as our next hex digit.)

Repeating the procedure,  $4064 - 3840$  gives a remainder of 224, which is hex E in column 1. There is no remainder, so our last digit is 0. Decimal  $49120 = \text{hexBFE0}$ .

Hex to decimal conversion is simple. Merely add the decimal values corresponding to the hex digits. #BFE0, for example, gives us  $45056 + 3840 + 224 + 0 = 49120$ .

# Appendix 7

## Oric MCP-40 printer use

In this appendix we will look at the operation of the Oric printer/plotter. This sophisticated device allows us to produce high-quality graphical output which is comparable with printouts produced by devices costing over a thousand pounds.

The printer owes much of its versatility to the remarkably advanced 4-colour penholder which can be positioned to an accuracy of less than 1/100th of an inch. We will examine in detail the operation of the printer as a pen plotter for drawing pictures, but first we will consider the action as a colour printer.

We can send characters to the printer using the LPRINT and LLIST commands which are almost identical in effect to the PRINT and LIST commands. There are, however, some control codes which we can send to the printer to access its plotter functions and colours. The four control codes we can use are summarised in the table below:

Character	Effect
8	Backspace
10	Line Feed
11	Reverse Line Feed
29	Rotate Pen holder

Note that use of character 29 only allows the programmer to rotate the pen holder one position, and not to choose a specific colour. This means that you'll have to keep careful track of the current colour if you use this method.

There are two other control codes which we can use to control the printer: CHR\$(18) tells the printer to enter the plotter graphics mode and CHR\$(17) tells the printer to return to text mode.

From now on we will consider the use of the printer as a graphic plotter. In this mode it has a much wider range of control characters. Once we have entered the plotter mode any of the alphabetic characters in the list below will be taken as the start of a command sequence. The parameters are simply LPRINTed to the printer following the appropriate command character and may be either string literals or the results of evaluating variables or calculations. Some of the commands are accompanied by example pro-

grams which use these commands to produce sample diagrams.

Character	Parameters	Effect
A		Exit plotter mode.
C	n	Change pens to number n(0–3). (colours are usually arranged Black, Blue, Green and Red.)
D	x,y	Draw from current position to (x,y). More than one set of coordinates may be included.
H		Move pen to plotter origin.
I		Reset origin to current position.
J	x,y	Draw relative to current position.
L	n	Choose type of line which is drawn (see diagram). Parameter n may be in the range 0–15.

```

10 LPRINT CHR$(18)
20 FOR I=0 TO 15
30 LPRINT "I"
40 LPRINT "L";I
50 LPRINT "P";"LINE TYPE:";I
60 LPRINT "D400,0"
70 LPRINT "M0,-20"
80 NEXT I
90 LPRINT CHR$(17)
100 END

```

LINE TYPE: 0	-----
LINE TYPE: 1	-----
LINE TYPE: 2	.....
LINE TYPE: 3	.....
LINE TYPE: 4	.....
LINE TYPE: 5	.....
LINE TYPE: 6	.....
LINE TYPE: 7	.....
LINE TYPE: 8	.....
LINE TYPE: 9	.....
LINE TYPE: 10	.....
LINE TYPE: 11	.....
LINE TYPE: 12	.....

LINE TYPE: 13    - - - - -  
 LINE TYPE: 14    - - - - -  
 LINE TYPE: 15    - - - - -

M            x,y        Move pen to (x,y).  
 P            a\$         Print the characters in a\$ up to the next carriage return.  
 Q            n         Set print orientation. The value of n (0–3) chooses normal, down, backwards (upside down) or up, as in the diagram below.

```

10 LPRINT CHR$(18)
20 LPRINT "M240,0"
30 FOR N=0 TO 3
40 LPRINT "Q";N
50 LPRINT "PBRIAN WAS HERE"
60 NEXT
70 LPRINT CHR$(17)
80 END

```

BRIAN WAS HERE  
 BRIAN WAS HERE  
 BRIAN WAS HERE  
 BRIAN WAS HERE  
 BRIAN WAS HERE

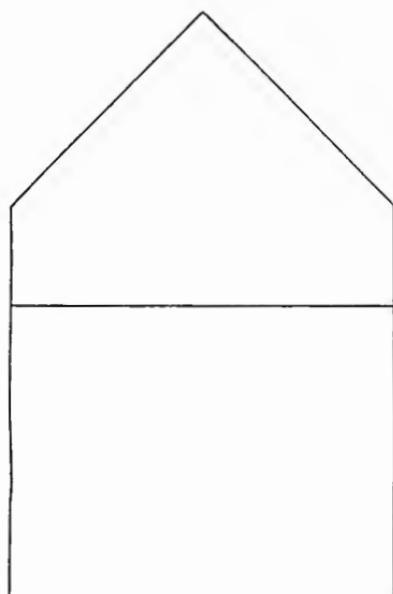
R            x,y        Move pen relative to the current position.  
 S            n         Size of characters for printing. n can be 0 to 63. 0 gives 80 characters per line, 63 gives 1 character per line.

```

10 LPRINT CHR$(18); "I"
20 FOR N=0 TO 4
30 LPRINT "S";N
40 LPRINT "PHELLO"
50 NEXT
60 LPRINT "M0,-500"
70 LPRINT "S63"
80 LPRINT "PA"
90 LPRINT "S1"
100 LPRINT CHR$(17)
110 END

```

~~HELLO~~HELLOHELLOHELLOHELLO

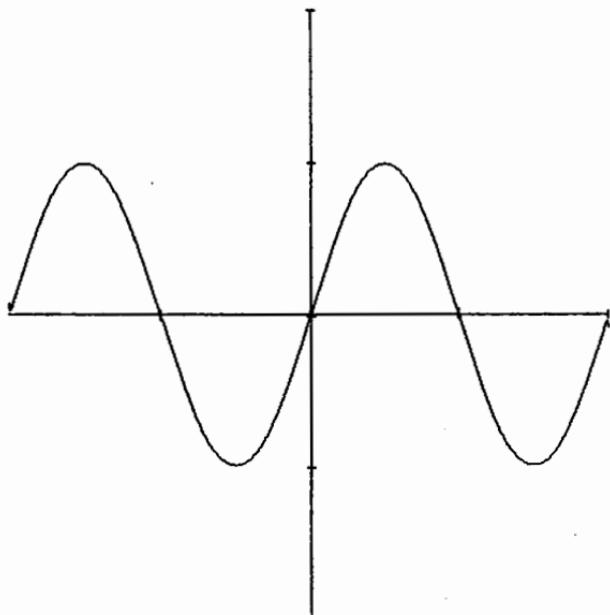


X

a,d,n

Axis drawing command. The parameters are:  
 a axis direction, (0 for y,1 for x)  
 d distance between tick marks  
 n number of marks along axis  
 See the example below for the effects of these parameters.

```
10 LPRINT CHR$(18); "I"
20 LPRINT "M200,-200"
30 LPRINT "X0,100,4"
40 LPRINT "M0,0"
50 LPRINT "X1,100,4"
60 LPRINT "M0,0"
70 FOR X=0 TO 400
80 A=(X-200)*PI/100
90 Y=SIN(A)*100
100 LPRINT "D";X;",";Y
110 NEXT
120 LPRINT CHR$(17)
130 END
```

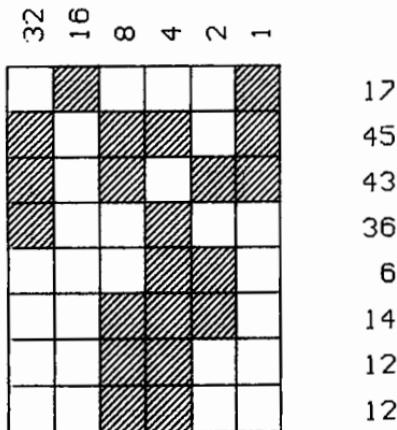


We will now have a look at a simple utility program for drawing diagrams of user-defined characters. This will provide an example of the potential for serious use that is inherent in the Oric MCP-40's high-resolution graphical abilities. You may have noticed that some diagrams in this book were drawn on the Oric printer, although its limited width prevents it producing large display graphics.

```
10 LPRINT CHR$(18)
20 LPRINT "M0,-480"
30 LPRINT "I"
40 FOR I=0 TO 8
50 LPRINT "M90,";I*30+30
60 LPRINT "D220,";I*30+30:IF I>6 THEN90
70 LPRINT "M";I*30+90;" ,30"
80 LPRINT "D";I*30+90;" ,270"
90 NEXT
100 LPRINT "Q3"
110 FOR I=5 TO 0 STEP-1
120 X=270-I*30-5:Y=275
130 LPRINT "M";X;" ,Y
140 LPRINT "P";2^I
150 NEXT:LPRINT"Q0"
160 FOR I=1 TO 8
170 READ P:Q=P
180 FOR J=5 TO 0 STEP -1
190 PP=INT(2^J+.1)
200 IF P>=PP THEN P=P-PP:GOSUB 400
210 NEXT J
220 X=290:Y=270-I*30+5
230 LPRINT "M";X;" ,";Y
240 LPRINT "P";
250 IF Q<100 THEN LPRINT " ";
260 IF Q<10 THEN LPRINT " ";
270 LPRINT " ";Q
280 NEXT
290 LPRINT CHR$(17)
300 END
400 X=30*(8-J):Y=270-30*I
410 FOR A=0 TO 30 STEP 5
420 LPRINT "M";X;" ,";Y+A
430 LPRINT "D";X+30-A;" ,";Y+30
440 NEXT
450 FOR A=5 TO 30 STEP 5
460 LPRINT "M";X+A;" ,";Y
470 LPRINT "D";X+30;" ,";Y+30-A
```

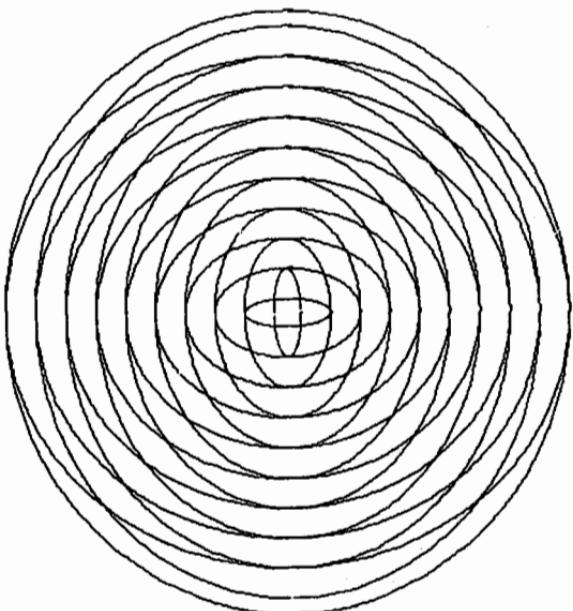
```
480 NEXT
490 RETURN
500 DATA 17, 45, 43, 36, 6, 14, 12, 12
```

Here is a sample output from this program, showing how it translates the decimal data defining the character into binary, and thence into graphical form.



We hope that this brief introduction to the wonders of the Oric's printer will have inspired you with as much enthusiasm as we have for this versatile and remarkable little machine. Just a few last words to say that the MCP-40's manual is clearly written and that, with the aid of these few examples, you should have little difficulty in using it effectively with your Oric. The printer has a standard Centronics interface and can also be connected to other computers if desired. Now we will leave it to the printer to have the final word in this section.

```
10 LPRINT CHR$(18); "I"
20 LPRINT "M240,-240"
30 LPRINT "I"
40 LPRINT "M0,0"
50 A=10 :B=30
60 REPEAT
70 GOSUB 200
80 B=A:A=A+20
90 GOSUB 200
100 B=B+40
110 UNTIL B=210
120 B=200:GOSUB 200
130 LPRINT "M100,-250"
140 LPRINT "PBYE?"
150 LPRINT CHR$(17)
160 END
200 LPRINT "M";A;",";0"
210 FOR M=0 TO 2*PI STEP PI/100
220 X=A*COS(M):Y=B*SIN(M)
230 LPRINT "D";X;",";Y
240 NEXT
250 LPRINT "D";A;",";0"
260 RETURN
```



BYE!

# Appendix 8

## 6502 OP codes

The tables presented in this appendix give the 6502 Op Codes, with the hex codes for each mnemonic categorised by address mode.

The notation is as follows:

<b>R1</b>	Register Changed
<b>R2</b>	Sending Register
<b>A</b>	Accumulator
<b>X</b>	Register X
<b>Y</b>	Register Y
<b>S</b>	Stack Register
<b>PC</b>	Program Counter
<b>P</b>	Processor Register (Status Flags)
<b>M</b>	Memory Location

The P Register flags are:

Bit 0	C	Carry
1	Z	Zero
2	I	Interrupt
3	D	Decimal
4	B	Break
5	*	(not used)
6	V	Overflow
7	N	Negative

MNEMONIC	DESCRIPTION	R1	R2	IMPLIED	IMMED	ZERO PAGE	Z-PAGE X
ADC	ADD WITH CARRY	A			69	65	75
AND	LOGICAL AND	A			29	25	35
ASL	ARITHMETIC SHIFT LEFT	A/M		0A(A)		06	16
BCC	BRANCH ON CARRY CLEAR						
BCS	BRANCH ON CARRY SET						
BEQ	BRANCH IF EQUAL TO ZERO						
BIT	COMPARE BITS WITH ACCUMULATOR					24	
BMI	BRANCH ON MINUS						
BNE	BRANCH ON NOT EQUAL TO ZERO						
BPL	BRANCH ON PLUS						
BRK	BREAK	S		00			
BVC	BRANCH ON OVERFLOW CLEAR						
BVS	BRANCH ON OVERFLOW SET						
CLC	CLEAR CARRY			18			
CLD	CLEAR DECIMAL			D8			
CLI	CLEAR INTERRUPT MASK			58			
CLV	CLEAR OVERFLOW FLAG			B8			
CMP	COMPARE TO ACCUMULATOR					C9	
CPX	COMPARE TO REG-X					E0	E4
CPY	COMPARE TO REG-Y					C0	C4
DEC	DECREMENT MEMORY					C6	D6
DEX	DECREMENT REG-X	X		CA			
DEY	DECREMENT REG-Y	Y		88			
EOR	EXCLUSIVE OR ACCUMULATOR	A			49	45	55
INC	INCREMENT MEMORY					E6	F6
INX	INCREMENT REG-X	X		E8			
INY	INCREMENT REG-Y	Y		C8			
JMP	JUMP TO ADDRESS	PC					

Z-PAGE Y	INDIRECT X	INDIRECT Y	RELATIVE ADDRESS	ABSOLUTE ADDRESS	ABSOLUTE X	ABSOLUTE Y	INDIRECT	P-REGISTER							
								X	V	*	B	D	I	Z	C
	61 21	71 31		6D 2D 0E	7D 3D 1E	79 39		x x						x x	
			90 B0					x						x	
			F0	2C					M7 M6					x	
			30 D0 10												
			50 70										x		0
													0		0
	C1	D1		CD EC CC	DD	D9		x						x x	
				CE	DE			x						x	
	41	51		4D EE	5D FE	59		x						x	
				4C				x x						x x	
									6C						

MNEMONIC	DESCRIPTION	R1	R2	IMPLIED	IMMED	ZERO PAGE	Z-PAGE X
JSR	JUMP TO SUBROUTINE	PC/S					
LDA	LOAD ACCUMULATOR	A			A9	A5	B5
LDX	LOAD REG-X	X			A2	A6	
LDY	LOAD REG-Y	Y			A0	A4	B4
LSR	LOGICAL SHIFT RIGHT	A/M	4A(A)			46	56
NOP	NO OPERATION			EA			
ORA	INCLUSIVE OR ACCUMULATOR	A			09	05	15
PHA	PUSH A ONTO STACK	S	A	48			
PHP	PUSH P-REGISTER ONTO STACK	S	P	08			
PLA	PULL ACCUMULATOR FROM STACK	A		68			
PLP	PULL P-REGISTER FROM STACK	P		28			
ROL	ROTATE LEFT ONE BIT	A/M		2A(A)		26	36
ROR	ROTATE RIGHT ONE BIT	A/M		6A(A)		66	76
RTI	RETURN FROM INTERRUPT	P/ PC/S		40			
RTS	RETURN FROM SUBROUTINE	PC/S		60			
SBC	SUBTRACT WITH CARRY	A			E9	E5	F5
SEC	SET CARRY	P		38			
SED	SET DECIMAL	P		F8			
SEI	SET INTERRUPT MASK (DISABLE)	P		78			
STA	STORE ACCUMULATOR	M	A			85	95
STX	STORE REG-X	M	X			86	
STY	STORE REG-Y	M	Y			84	94
TAX	TRANSFER A TO X	X	A	AA			
TAY	TRANSFER A TO Y	Y	A	A8			
TSX	TRANSFER S TO X	X	S	BA			
TXA	TRANSFER X TO A	A	X	8A			
TXS	TRANSFER X TO S	S	X	9A			
TYA	TRANSFER Y TO A	A	Y	98			



# Appendix 9

## ROM Routines and Addresses

These are routines within the Oric ROM which can be CALLED from BASIC to produce the effects described. They can also be used directly from USER routines in machine code (see Chapter 10).

For most of these routines it will be sufficient to load the appropriate 6502 registers before JSRing to the routine in question. However, for the graphics and sound routines, parameters must be passed by storing them in the area of memory starting at #2E0. Parameters are represented as 16 bit 2's complement numbers. The parameter area address will be referred to as PARAMS. On return, PARAMS + 0 is set to 1 if an error occurred. CALLING routines should set PARAMS + 0 to 0 before CALLING. All addresses are given in hexadecimal. All routines corrupt registers A, X and Y unless otherwise specified.

### VDU

address: F77C

Prints the character on the screen and moves cursor to the right.

Call parameters: X = character to print  
Return parameters: none  
Registers corrupted: none

See Appendix 3 for Control Codes

### STOUT

address: F865

Prints a message on the status line on the screen (locations 48000 to 48039).

Call parameters: A = message address (low)  
Y = message address (high)  
X = horizontal position to start  
Return parameters: X = next position

Message is terminated with a zero.

**GTORKB****address: EB78**

Characters are returned at the current repeat rate which is determined by locations 24E and 24F (hex). Location 24E is the delay before a key starts repeating in units of 30ms. Location 24F is the time between successive characters when the keyboard is repeating in units of 30ms. To obtain characters at maximum rate set locations 24E and 24F to one.

Call parameters: none

Return parameters: A = ASCII character

signflag + ve no character

signflag - ve valid character

Registers corrupted: none

**PRTCHR****address: F5C1**

Prints a character to the printer.

Call parameters: A = ASCII character

Registers corrupted: A

**OUTLED****address: E75A**

Outputs leader (9 characters of ASCII Code 16, SYN) to tape at current speed.

Call parameters: none

Return parameters: none

Note: For all cassette routines the current speed is governed by location 24D. 0 is fast, greater than 0 is slow.

**GETSYN****address: E735**

Reads bytes from tape until in sync with tape.

Call parameters: none

Return parameters: none

Registers corrupted: A,X

**OUTBYT** address: E65E

Outputs a byte to the tape at current speed.

Call parameters: A = output character

Return parameter: none

Register corrupted: A

**RDBYTE** address: E6C9

Reads a byte from the cassette at current speed.

Call parameters: none

Return parameters: A = input character

Registers corrupted: A

**CURSET** address: F0C8

Call parameters: PARAMS + 1: x value

PARAMS + 3: y value

PARAMS + 5: fb value

Return parameters: PARAMS set to 1 if out of range error

**CURMOV** address: F0FD

Call parameters: PARAMS + 1: x value

PARAMS + 3: y value

PARAMS + 5: fb value

Return parameters: PARAMS set to 1 if out of range error

**DRAW** address: F110

Call parameters: PARAMS + 1: x value

PARAMS + 3: y value

PARAMS + 5: fb value

Return parameters: PARAMS set to 1 if out of range error

Registers corrupted: A,X

**CHAR** address: F12D

Call parameters: PARAMS + 1: ASCII character (in LSB)

PARAMS + 3: character set (0 standard, 1 alternate)

PARAMS + 5: fb value

Return parameters: PARAMS set to 1 if out of range error

**CIRCLE****address: F37F**

Call parameters: PARAMS + 1: radius  
 PARAMS + 3: fb value

Return parameters: PARAMS set to 1 if out of range error

**PATRN (PATTERN)****address: F11D**

Call parameters: PARAMS + 1: pattern value

Return parameters: PARAMS set to 1 if out of range error

Register corrupted: X

**POINT****address: F1C8**

Call parameters: PARAMS + 1: x value  
 PARAMS + 3: y value

Return parameters: PARAMS set to 1 if out of range error

PARAMS + 1: 0=background  
 1=foreground

**FILL****address: F268**

Call parameters: PARAMS + 1: no. of rows  
 PARAMS + 3: no. of cells  
 PARAMS + 5: value

Return parameters: PARAMS set to 1 if out of range error

**PAPER****address: F204**

Call parameters: PARAMS + 1: colour

Return parameters: PARAMS set to 1 if out of range error

**INK****address: F210**

Call parameters: PARAMS + 1: colour

Return parameters: PARAMS set to 1 if out of range error

**PING****address: FA9F**

Accessed by address.

NB: All sound routines use the same parameter passing routines as the graphic routines.

<b>SHOOT</b>	<b>address: FAB5</b>
	Accessed by address.
<b>EXPLD</b>	<b>address: FACB</b>
	Accessed by address.
<b>ZAP</b>	<b>address: FAE1</b>
	Accessed by address.
<b>KBEEP</b>	<b>address: FB14</b>
	Produces keyboard click. Accessed by address.
<b>CONTBP</b>	<b>address: FB2A</b>
	Gives CTRL key click. Accessed by address.
<b>SOUND</b>	<b>address: FB40</b>
Call parameters:	PARAMS + 1: channel PARAMS + 3: period PARAMS + 5: volume
Return parameters:	PARAMS set to 1 if out of range error
<b>MUSIC</b>	<b>address: FC18</b>
Call parameters:	PARAMS + 1: channel (1-3) PARAMS + 3: octave (0-7) PARAMS + 5: note (1-12) PARAMS + 7: volume (0-15)
Return parameters:	PARAMS set to 1 if out of range error
<b>PLAY</b>	<b>address: FBD0</b>
Call parameters:	PARAMS + 1: tone channel PARAMS + 3: noise channel PARAMS + 5: envelope mode PARAMS + 7: envelope period
Return parameters:	PARAMS set to 1 if out of range error

**W8912****address: F590**

Writes the data in X to the 8912 register specified in A. The routine ensures that the keyboard port is kept enabled. Register OE should not be addressed, as this is the external port used by the keyboard.

Call parameters: A=8912 register no.

X=output data

Return parameters: none

### Oric RAM locations page zero/page two

#### PAGE 0

LINWID	#0031	Line width for terminal. (V1.0 printer width)
TXTTAB	#009A-#009B	Start of BASIC text
VARTAB	#009C-#009D	Start of variables
ARYTAB	#009E-#009F	Start of arrays
STREND	#00A0-#00A1	End of variables, lo-men
MEMSIZ	#00A6-#00A7	Top of FRE memory, HIMEM
CHRGET	#00E2-#00E7	Code to increment TXTPTR
CHRGOT	#00E8	LDA instruction
TXTPTR	#00E9-#00EA	Pointer to current character being interpreted
SKPSPC	#00EB-#00EE	If space then CHRGET
QNUM	#00EF-#00F8	Set carry if 0-9 and zero flag if CHR\$(0)
CHRRTS	#00F9	Return instruction

#### PAGE 2

KEYAD	#0208	Key address if key press
KBSTAT	#0209	#A4=left shift, #A7=right
CAPLCK	#020C	#80=CAPS
PAT	#0213	PATTERN register for CIRCLE/DRAW
CURX	#0219	Graphics cursor X and
CURY	#021A	Y coordinates
GRA	#021F	1=HIRES,0=TEXT/LORES
SXTNK	#0220	1=16K memory, else 48K
XVDU	#238	jump to VDU routine (VDU)
XGETKY	#23B	jump to key routine (GTORKB)
XPRTCP	#23E	jump to printer output routine (PRTCHR)
XSTOUT	#241	jump to status line output routine (STOUT)

INTFS	#244	jump to interrupt handler
NMIJP	#247	jump to NMI routine
INTSL	#24A	return from interrupt handler (normally RTI but may be patched to a jump)
TSPEED	#24D	0 - fast, <>0 slow
KBDLY	#24E	delay for keyboard auto repeat
KBRPT	#24F	repeat rate for keyboard repeat
PWIDTH	#256	printer width (normally set to 80)
VWIDTH	#257	screen width (normally set to 40)
CURROW	#268	cursor row position
MODE 0	#26A	bits in this byte define the current state of various functions
		BIT      FUNCTION
	7	Spare
	6	Spare
	5	1=Protect columns 0 and 1 on screen
	4	1=Last character printed was ESC
	3	1=Key click off
	2	Spare
	1	1=VDU on
	0	1=Cursor on
BGND	#026B	Background (PAPER) colour +16
FGND	#026C	Foreground (INK) colour
CURON	#0270	Cursor on/off flag
CUR INV	#0271	Cursor invert flag
TIMER1	#0272-#0273	Keyboard
TIMER2	#0274-#0275	Cursor
TIMER3	#0276-#0277	Spare/WAIT Sixteen-bit timer decrementing every 1/100th of second. Continues past 0
VDUL2	#0278-#0279	Addr. of second line on screen
VDUL1	#027A-#027B	Addr. of first line on screen
VDUCH	#027C-#027D	No. of char's to scroll normally 26 lines*40
NOROWS	#027E	No. of rows on screen
ICHAR	#02DF	Current key pressed
PARAMS	#02E0	Parameter block transfer buffer for graphics and sound

# Appendix 10

# Input/Output Circuitry

The electronics enthusiast wishing to interface his Oric to control or monitor external events might be new to computing, and it is hoped that the notes given here will enable him or her to experiment in applying some custom made silicon chips in example circuits, and explain how to uniquely specify memory locations for external Input/Output purposes.

## **Addressing and decoding memory locations**

Figure 10.1 shows a 6502 CPU, the Central Processor Unit which is the 'brains' of the Oric microcomputer. It has 8 data lines (D<sub>0</sub> to D<sub>7</sub>) and 16 address lines, A<sub>0</sub> to A<sub>15</sub>. Referring to figure 10.2, we can see that there are 65536 ways of arranging the 16 address lines in either a low state (0) or high state (1). These states are actually voltage levels, 0V for low and +5V for high. Hence, the 6502 CPU has the capability of talking to 65536 memory locations, i.e. a 64K memory microcomputer.

These address lines are configured to either ROM chips, RAM chips or other memory mapped devices (see the memory map, Appendix 5). The eight data-lines are used to pass 8-bit instructions or raw data between the CPU and the ROM/RAM memory and peripherals. Referring to figure 10.3, we can see that there are 255 ways of arranging the 8-bit data lines. How, then, can we use these address lines to decode memory locations? Chapter 11 tells us that we could use either the 'spare' area of memory or Page 3 memory for suitable locations. The spare area is between #BFE0 and #BFFF.

Looking at the conversion table given in Appendix 6, we can convert straight from hexadecimal to decimal. From left to right:

$$\begin{array}{r} \# \quad B \quad F \quad F \quad F \\ 45056 + 3840 + 240 + 15 = \text{location 49151} \end{array}$$

We could of course use the Oric for these conversions, with a simple PRINT #BFFF, or whatever, but we're concerned here with the values of each 4-bit sequence (called a nibble) coded by each hex symbol.

Similarly, referring to the memory map again, we see that Page 3 of the memory occupies locations #0300 to #03FF (768 to 1023 decimal). How can these decimal and hexadecimal numbers be used to provide us with information about the state of the address lines A0 to A15? Take location #BFFF as an example. If we refer to figure 10.4, which gives the bit patterns for each hex nibble, we can see that hex B=1011 binary, and F=1111 binary, so that the bit pattern, or pattern of states (0 or 1) on the address lines will be:

hex	#	B	F	F	F	
bit pattern		1 0 1 1	1 1 1 1	1 1 1 1	1 1 1 1	
address line	A15	...	...	...	...	A0

Every time the instruction:

POKE #BFFF,xxx

or:

POKE 49120,xxx

is used, the state of the address lines will be as shown, giving all address lines set high (1) except A14, which is low (0).

Similarly, for an example Page 3 location POKE #03FF,xxx will give the following bit pattern:

hex	#	0	3	F	F	
bit pattern		0 0 0 0	0 0 1 1	1 1 1 1	1 1 1 1	
address line	A15	...	...	...	...	A0

Lines 10, 11, 12, 13, 14, 15 are all low in this case.

The different state of these lines for each address means that each memory location can be uniquely decoded and used to provide an enable (low) signal to activate an input chip, whenever a certain memory location is specified by the POKE statement.

### Decoders and decoding

There are many ways of providing an enable (low) signal, and we shall describe and illustrate examples to illustrate general principles. Basically, most input and output chips are enabled (rendered active) by a low (0) signal, and what is necessary is that this low signal, applied to an output or input chip, should occur only when the correct address is on the address lines.

This  $\overline{CE}$  chip-enable signal can be used to activate a chip such as the 74LS374 6-bit output latch shown in figure 10.6. The  $\overline{CE}$  signal can activate the same chip (or others such as the 74LS244 and 74LS245) during an input, except that this time input data is latched on to the data-bus of the Oric only if the internal 6522 in the Oric has been disabled by pulling  $\overline{I/O}$  CONTROL low at the same time, so that clashes with external and internal data do not occur. The  $\overline{I/O}$  CONTROL chip is activated when in Page 3 as explained in Chapter 11.

Figure 10.5 shows a possible decoding method for the top of Page 3 using a 74LS30 type 8 line decoder.  $\overline{IO}$  is pulled low whenever page three is accessed so we can use this to save ourselves the bother of decoding the top eight bits of the address.

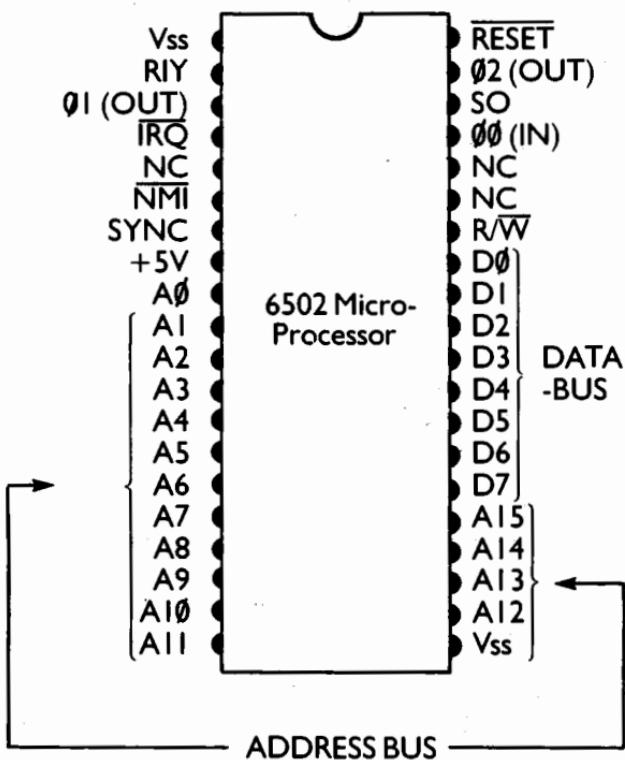
All connections are made via the Oric's 34-way expansion-bus using an IDC female connector, obtainable from RS components, for example. All the chips necessary for construction can also be obtained from RS or other similar suppliers.

Figure 10.6 shows a circuit using the 6522 VIA, which provides a very flexible way of obtaining input/output to external devices (which is why the Oric itself uses one!). See Chapter 11. The VIA has 16 registers, and these can be mapped on to the Oric memory by using, for example, A0, A1, A2 and A3, since there are 16 ways of arranging these lines in either a 1 or 0 state. The significant registers are ORB, ORA, DDRA and DDRB. The VIA is programmed for input by inputting a control word to DDRA for port A and DDRB for port B. If we memory-map DDRA on to #03E3 as shown, then POKE#03E3,255 will make port A an output. POKE#03E1,XX will then output data to port A. Similarly, POKE#03E3,0 will make port A an input, and PRINT PEEK(#03E1) will read the input data.

Decoding is easier, since only 12 address lines need to be decoded and  $4 \times 17154$ , 4-16 line decoders will produce 4 unique output lines which, when gated, will produce a chip-enable signal ( $\overline{CE}$ ) for the VIA as well as the necessary disable signals (depending on which area of memory is being used).

When using the printer interface for input/output, no lines need to be pulled low, so  $\overline{I/O}$  CONTROL,  $\overline{MAP}$  and  $\overline{ROMDIS}$  are left alone. A 74LS235 transceiver chip is suitable since it can be used for both input and output along the same data lines, but power is still only obtainable from the 34-way connector unless an external supply is used.

Fig. 10.1: The 6502 Micro-Processor



\*these control lines are available at the expansion bus, and can be utilized for user designed peripherals.

D<sub>0</sub>–D<sub>7</sub> : DATA-BUS

A<sub>0</sub>–A<sub>15</sub> : ADDRESS-BUS

\*R/W : READ/WRITE

\*IRQ : INTERRUPT REQUEST, ACTIVE IF  
THE "I" REGISTER IS AT "0" & IRQ IS PULLED LOW.

\*RESET : INITIALIZES THE 6502 FROM A  
POWER DOWN CONDITION.

BIT	LEVEL POWER	DECIMAL
-----	-------------	---------

$A_0$	I	$2^0$	1
$A_1$	I	$2^1$	2
$A_2$	I	$2^2$	4
$A_3$	I	$2^3$	8
$A_4$	I	$2^4$	16
$A_5$	I	$2^5$	32
$A_6$	I	$2^6$	64
$A_7$	I	$2^7$	128
$A_8$	I	$2^8$	256
$A_9$	I	$2^9$	512
$A_{10}$	I	$2^{10}$	1024
$A_{11}$	I	$2^{11}$	2048
$A_{12}$	I	$2^{12}$	4096
$A_{13}$	I	$2^{13}$	8192
$A_{14}$	I	$2^{14}$	16384
$A_{15}$	I	$2^{15}$	32768

TOTAL =  $65,535 + 1 = 65,536$  MEMORY LOCATIONS

ALL LINES AT 0 ; MEMORY  
LOCATION 0000

ADDRESS LINES  
SPECIFYING MEMORY

## DATA-LINES

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	BIT
1	1	1	1	1	1	1	1	BINARY LEVEL
2 <sup>7</sup>	2 <sup>6</sup>	2 <sup>5</sup>	2 <sup>4</sup>	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>	POWER OF 2
255 =	128	64	32	16	8	4	2	DECIMAL

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

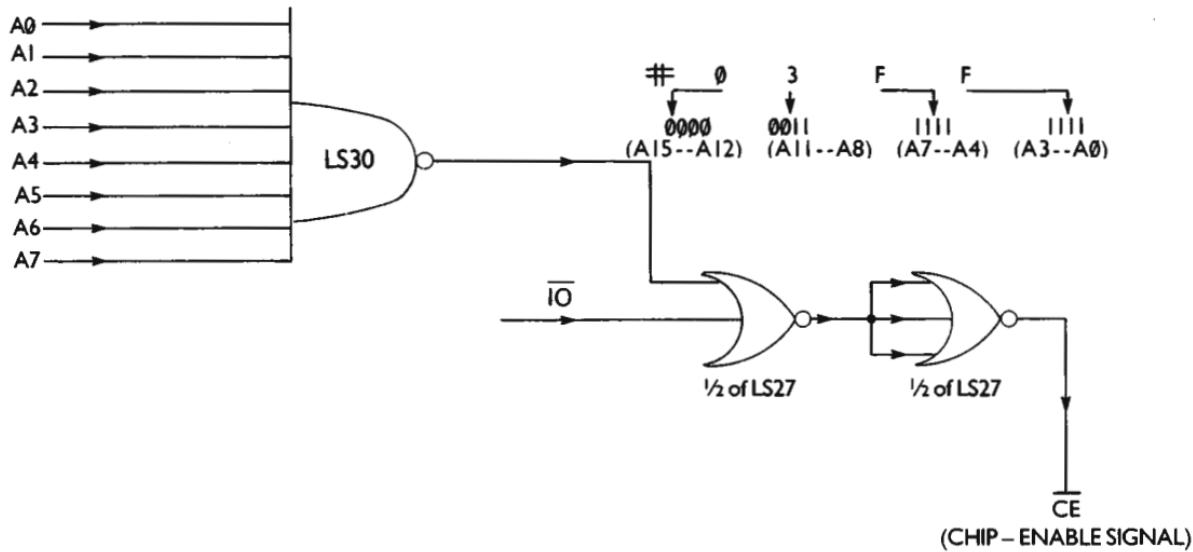
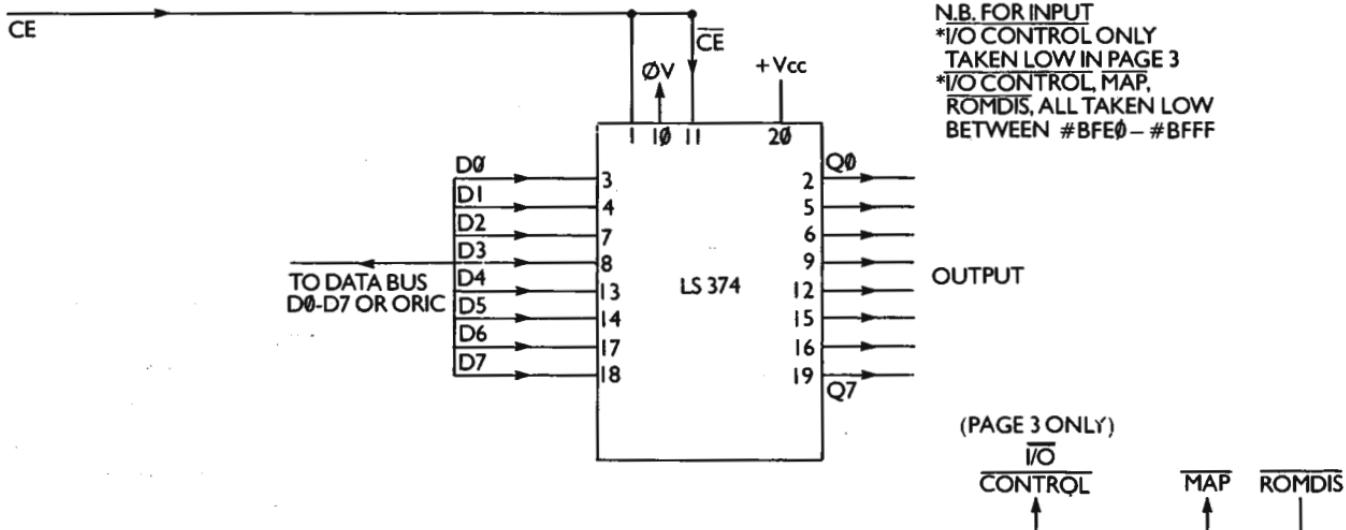
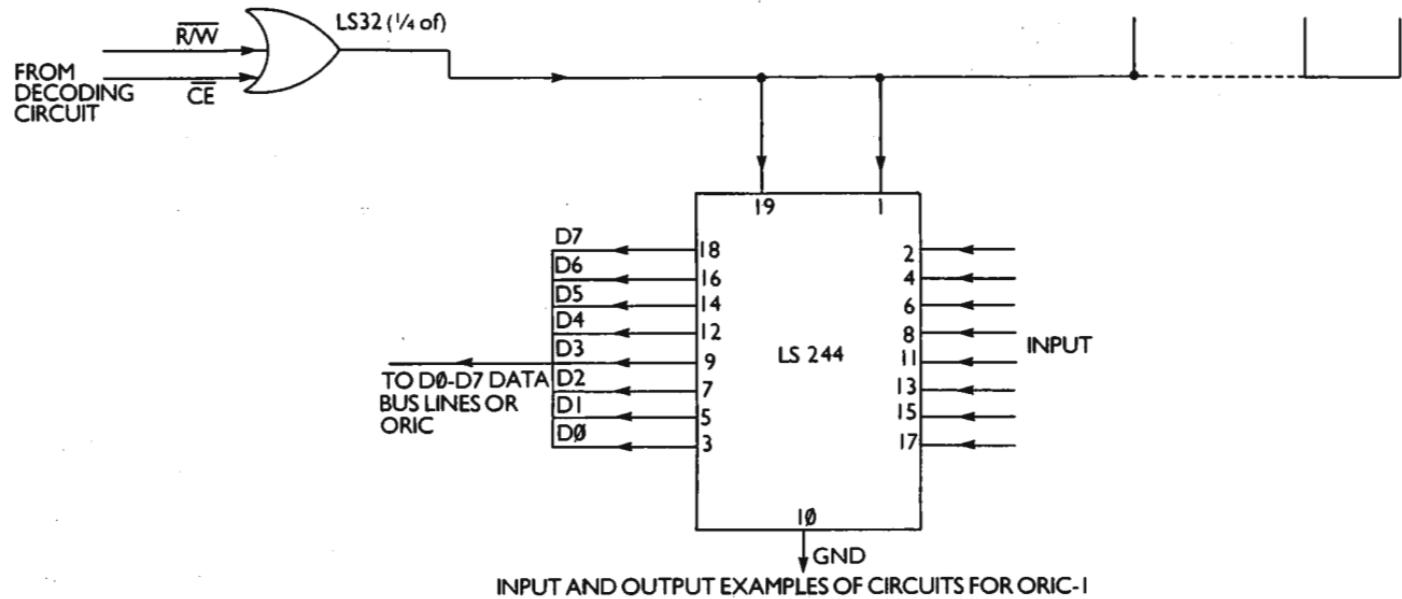
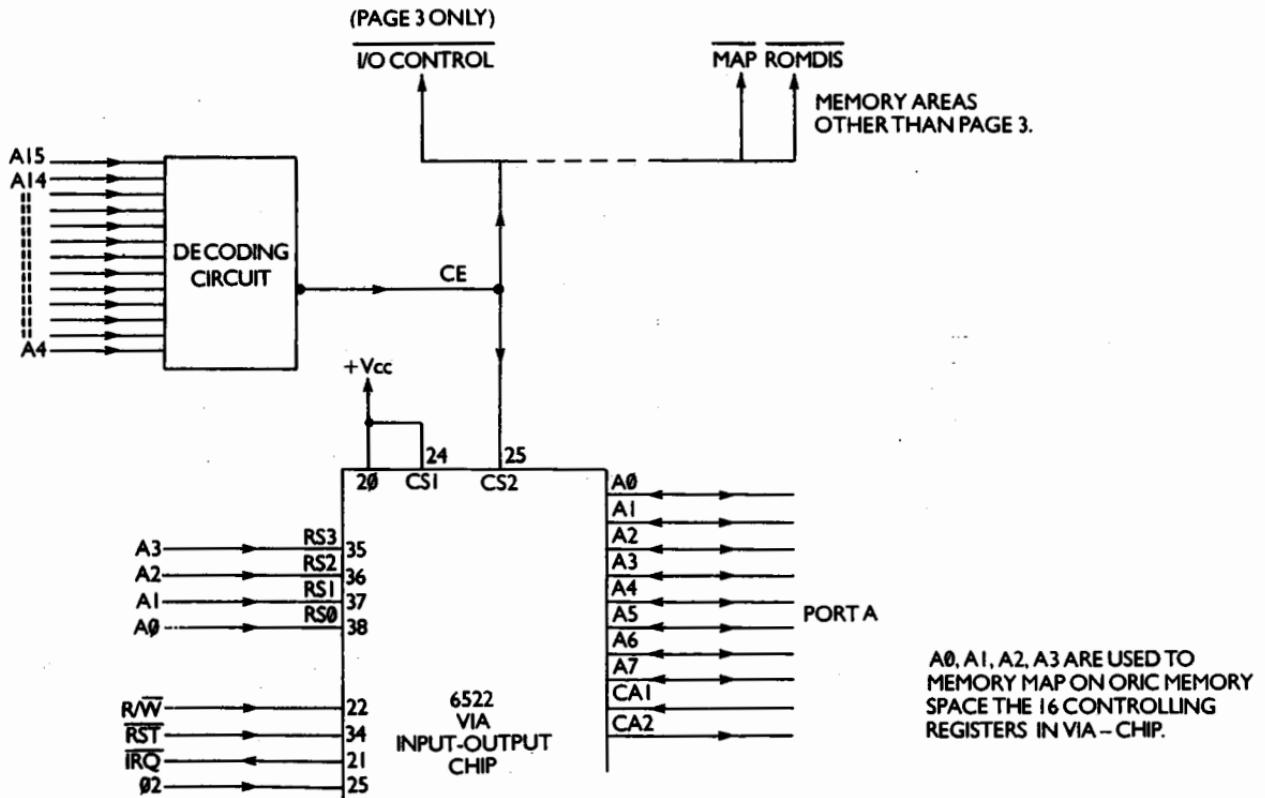


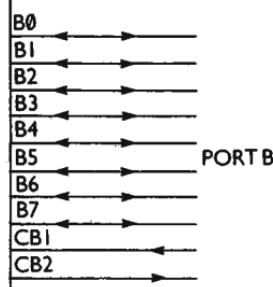
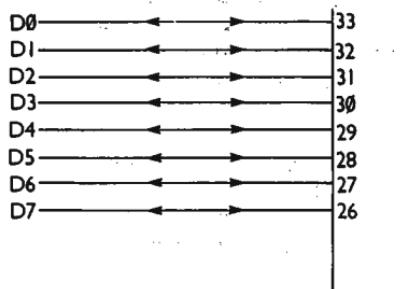
Fig. 10.5  
ABSOLUTE DECODING OF  
LOCATION 03FF (TOP OF PAGE 3)

FROM DECODING CIRCUIT









POSSIBLE DECODERS

4x 74154, 4-16 DECODER  
5 x 74LS 138, 3-8 DECODER

USING CUSTOM MADE CHIP FOR  
EXTERNAL I/O. THE  
6522 VERSATILE - INTERFACE -  
ADAPTOR (VIA).

FOR EXAMPLE IN PAGE 3: ON  
ORIC MEMORY MAP ie. 03E0 TO 03Ef

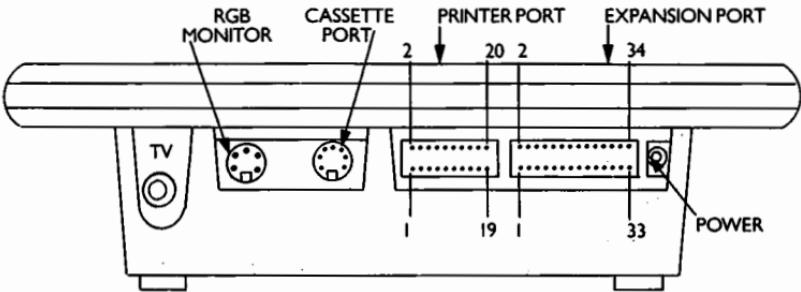
A15 ←                            A3 ←                    A0  
0000            0011            1110            XXXX  
(0)              (3)              (E)

03EF	ORA (PORT A)
03EF	IER
03ED	IFR
03EC	PCR
03EB	ACR
03EA	SR
03E9	T3C-H
03E8	T2L-L/T2C-L
03E7	T1L-H
03E6	T1L-L
03E5	T1C-H
03E4	TK-L/T1C-L
03E3	DDRA
03E2	DDRB
03E1	ORA (PORT A)
03E0	ORB (PORT B)

CONTROLLING  
REGISTERS OF  
VIA, MEMORY MAPPED  
ONTO PAGE 3 MEMORY  
SPACE

# Appendix 11

## ATMOS I/O Connections



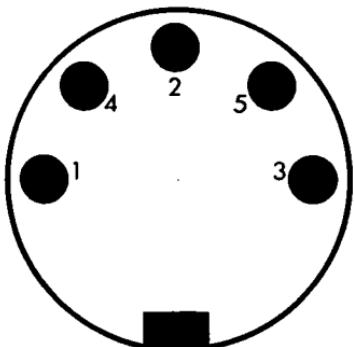
EXPANSION SOCKET (MALE)

2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	
ROMDIS	RESET	I/O	CONT.	IRQ	Dd	D1	D6	D3	D4	A4	D7	A15	A14	A13	A12	A11	GND
1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	

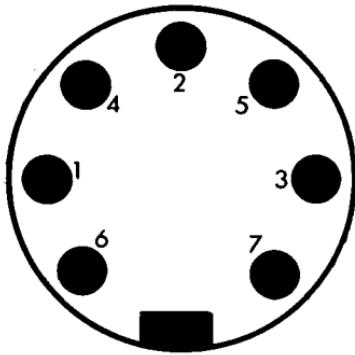
PRINTER SOCKET (MALE)

2	4	6	8	10	12	14	16	18	20
STB	D0	DI	D2	D3	D4	D5	D6	D7	ACK

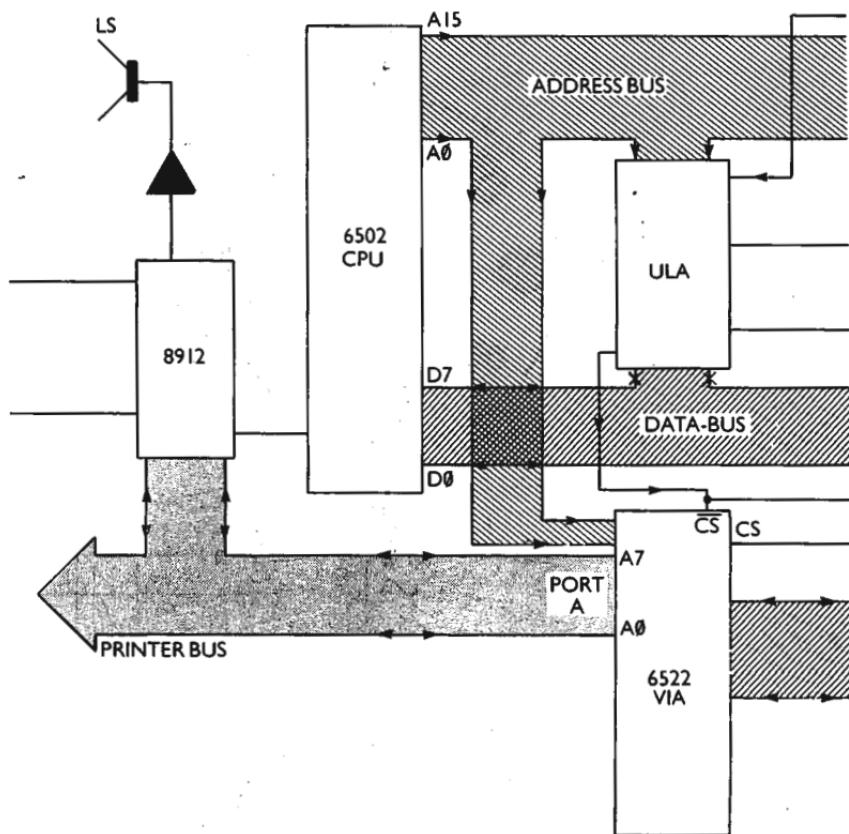
PIN-OUTS FOR PRINTER AND EXPANSION SOCKETS

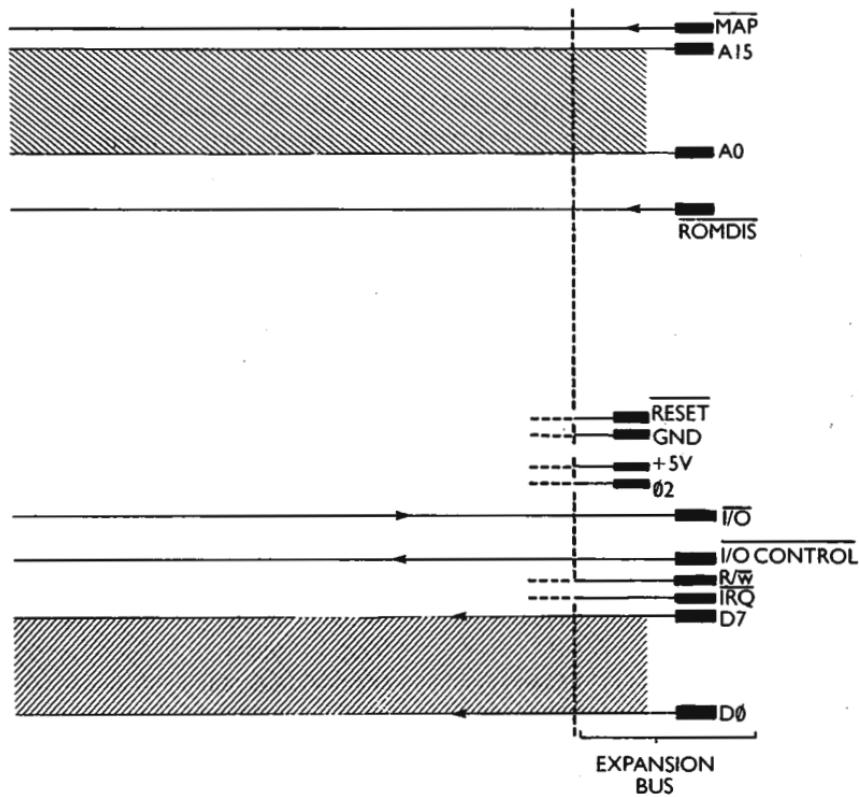


**RGB OUTPUT**  
1. RED  
2. GREEN  
3. BLUE  
4. SYNC  
5. GROUND



**CASSETTE & SOUND**  
1. TAPE OUT  
2. GROUND  
3. TAPE IN  
4. SOUND  
5. RELAYS  
6. RELAYS  
7. RELAYS

**Block Schematic-Chip Layout**



BLOCK DIAGRAM  
SHOWING EXPANSION BUS PIN-OUT

CS

# Appendix 12

## BASIC Reserved Words and Tokens

Here in a convenient form are presented the BASIC keywords and the tokens which are used to store Oric BASIC keywords in a single byte of memory. These are reserved words and may not be used in variable names.

ABS	216	FOR	141	PAPER	177
AND	209	FRE	218	PATTERN	174
ASC	236	GET	190	PEEK	230
ATN	229	GO	247	PI	238
AUTO	199	GOSUB	155	PING	166
CALL	191	GOTO	151	PLAY	169
CHAR	11	GRAB	159	PLOT	135
CHR\$	237	HEX\$	220	POINT	243
CIRCLE	173	HIMEM	158	POKE	185
CLEAR	189	HIRES	162	POP	134
CLOAD	182	IF	153	POS	219
CLS	148	INK	178	PRINT	186
CONT	187	INPUT	146	PULL	136
COS	226	INT	215	READ	149
CSAVE	183	KEY\$	241	RECALL	131
CURMOV	171	LEFT\$	244	RELEASE	160
CURSET	170	LEN	233	REM	157
DATA	145	LET	150	REPEAT	139
DEEK	231	LIST	188	RESTORE	154
DEF	184	LLIST	142	RETURN	156
DIM	147	LN	224	RIGHT\$	245
DOKE	138	LOG	232	RND	223
DRAW	172	LORES	137	RUN	152
EDIT	129	LPRINT	143	SCRN	242
ELSE	200	MID\$	146	SGN	214
END	128	MUSIC	168	SHOOT	163
EXP	225	NEW	193	SIN	227
EXPLODE	164	NEXT	144	SOUND	167
FALSE	240	NOT	202	SPC	197
FILL	175	ON	180		
FN	196	OR	210		

SQR	222	TAN	228	TRUE	239
STEP	203	TEXT	161	UNTIL	140
STOP	179	THEN	201	USR	217
STORE	130	TO	195	VAL	235
STR\$	334	TROFF	133	WAIT	181
TAB	194	TRON	132	ZAP	165

# Index

- @ 72, 75
- ABS function 29, 36, 108
- Addressing 54, 121
  - machine code, 196, 201, 214, 273
- Algorithm 17
- AND 44, 109
- Animation 73, 91, 190
- Apostrophe 18
- Argument, of function 25
- Arithmetic operators 9, 27, 57
- Arrays 33, 41, 59, 60, 97, 123
  - storing and recalling 66, 180
- ASC 25, 110
- ASCII codes 25, 40, 76, 78, 86, 228
- ATN 29, 111
- Attributes 72, 74, 78, 83
  - ASCII codes 230
  - Escape sequences 74, 78, 232
- AUTO 64, 119
- BASIC**
  - interpreter 190
  - language 6
  - program storage 53, 57ff, 65
  - reserved words 11, 16, 288
  - tokens 57
- Baud rates 63, 117
- BCD (Binary Coded Decimal) Notation 195
- Binary
  - notation 20, 50, 78, 82, 84, 191
  - conversion tables 244
- Bits 50, 78, 84, 192
- Bubble sort 41
- Bytes 50, 54, 59, 192
- CALL** 112, 220
- CAPS mode 12, 13
- Cassette recorder
  - choice of 5, 61
  - connecting up 5, 62, 63, 65
- Cassette storage 61ff
  - auto-run programs 64, 119
- joining programs 65, 116
- loading programs 63, 115
- saving programs 62, 119
- storing/recalling arrays 66, 180
- storing/recalling memory blocks 64, 119
- verifying a saved program 65, 116
- Cassettes**
  - care of 67
  - choice of 62
- CHAR** 85, 112
- Character**
  - alternate set 54, 78
  - codes 25, 57, 59, 228
  - comparison 40
  - double height 75
  - flashing 75
  - graphics 86
  - non-PRINTing 25
  - on HIRES screen 86
  - sets in memory 53, 87
- CHR\$** 26, 74, 113
- Chromatic scale** 94
- CIRCLE** 82, 114
- CLEAR** 20, 59, 114
- CLOAD** 62, 115
- CLS** 12, 117
- Colon** 19
- Colour** 71
  - attributes 72, 74
  - escape sequences 75, 232
  - inverse colours 76
  - memory locations 84
  - TV tuning 4
- Comma** 70
- Commands** 7
- Complements** 37
- Concatenation** 28
- Conditions** 36, 40, 44, 46
- CONT** 117
- Control characters** 12, 71, 73, 228
- COS** 29, 118
- CSAVE** 62, 119

- CTRL key 12
- CURMOV 81, 119
- CURSET 80, 120
- Cursor 4, 12
  - control keys 7, 12
  - graphics 80
- DATA 32, 96, 121, 165
- Decimal notation 50, 191
  - conversion tables 244
- Decoding 274
- DEEK function 54, 58, 60, 121
- DEF 122
- DEF FN 29, 122
- DEF USR 122
- DEDelete key 7
- DIM 33, 123
- Disc Drives 5
- Documentation 18
- DOKE 54, 58, 125
- Dollar sign 20
- Double size characters 75
- DRAW 80, 125
- EDIT 14, 126
- Editing programs 12
- ELSE 37, 139
- END 127
- Envelope period 104
- Error messages 235
- ESCape sequences 74, 77, 78
- Exclamation mark (Shriek) 220
- EXP 29, 127
- Expansion port 5
- EXPLODE 93, 128
- Exponential notation 23, 59
- FALSE 39, 44, 128
- FILL 84, 129
- Flags 41, 44
- Flashing characters 75
- FN 29, 130
- Foreground/background parameter 80
- FOR . . . NEXT loops 31, 131, 167
- FRE 132
- Functions
  - numeric 29
  - string 25
- GET 21, 133
- GOSUB 47, 134, 151
- GOTO 18, 43, 46, 151
- GRAB 53, 136
- Graphics 69ff
  - characters 86
  - high resolution 80
  - low resolution 78
  - medium resolution 79
  - Oric MCP-40 printer 252
  - text 69
- HEX 27, 137
- Hexadecimal notation 24, 27, 137, 192
  - conversion tables 245
- High resolution graphics 80
- HIMEM 122, 137
- HIRES mode 53, 80, 138
  - screen grid 239
- IF . . . THEN . . . (ELSE) 37, 139
- ILLEGAL QUANTITY ERROR 27
- Indexing 198
- INK 71, 140
- INPUT 17, 26, 141
- Input/Output
  - circuitry 273
  - connections 284
  - memory locations 223
- INT 27, 29, 142
- Interrupts 148, 213
- Join facility 65, 115
- KEY\$ 142
- Keyboard 7
- Keyclick toggle 12
- Keywords 11, 108ff, 288
- Languages 7
- LEFT\$ 28, 143
- LEN 28, 144
- LET 10, 11, 144
- Line numbers 13, 15, 18, 57, 134, 135, 171
- LIST 14, 145
- LLIST 146
- LN 29, 127, 146
- LOG 29, 146
- Logical operators 44, 109, 150, 152
- Logical values 39
- Loops 18, 31, 43
  - counter 38
  - FOR . . . NEXT 31, 131
  - nested 34, 43
  - REPEAT . . . UNTIL 36, 38, 167
- LORES 0 mode 76, 147
  - screen grid 238

- LORES 1 mode 54, 78, 147  
 alternate character set 54, 78  
 screen grid 238  
 Low resolution graphics 78  
**LPRINT** 58, 148
- Machine code** 190ff  
 addressing 122, 196, 201, 214  
 conventions 214  
 instructions 197, 202, 261  
 number systems 191  
 registers 199  
**ROM routines & addresses** 267  
 storage 137  
**Mantissa** 24, 59  
**Medium resolution graphics** 79  
**Memory** 4, 50ff  
 area available for programs 132  
 characters in 53, 86  
 Input/output areas 223  
 locations 50, 52, 83  
 map 240  
 pages 53  
**ROM routines & addresses** 266  
 screen 53, 64, 77  
 variables storage 59  
**MID\$** 28, 148  
**Monitors** 4  
**MUSIC** 94, 149  
 envelopes 103
- Negative numbers** 24, 36, 194  
**NEW** 150  
**NEXT** 31, 35, 131  
**Noise channels** 100, 105  
**NOT** 44, 150  
**Numbers** 9, 10, 23, 36  
 binary 24, 50  
 Binary Coded Decimal (BCD) 195  
 decimal 191  
 hexadecimal 24, 27, 137, 192  
 integer 24, 54  
 negative 24, 36  
 range 23, 27  
 real 23, 59  
**Numeric functions** 29
- ON** 46, 151  
**Operands** 197  
**Operation codes** 197  
 6502 Op codes table 261  
**Operators**  
 arithmetic 9, 27
- conditional 38  
 order of priorities 10, 27  
 string 28  
**OR** 44, 152  
 Oric MCP-40 printer 5, 252  
**OUT OF DATA ERROR** 165  
**OUT OF MEMORY ERROR** 35, 43  
 Output, see **Input/Output**
- Pages 53, 198  
**PAPER** 71, 153  
**Parsing** 190  
**PATTERN** 82, 154  
**PEEK** 54, 154  
 Percent sign 24  
**PI,  $\pi$ , constant** 29, 155  
**PING** 22, 93, 155  
**PLAY** 96, 100, 156  
**PLOT** 76, 158  
**POINT** 83, 159  
**POKE** 54, 78, 84, 159  
**POP** 160  
**POS** 161  
**PRINT @** 72, 162  
**PRINT** 9, 162  
 abbreviation 9  
 text formatting 25, 69, 178, 183  
**Printer** 5, 146, 148, 152  
**Priority** 10, 27  
**Prompt string** 17  
**PULL** 44, 163
- RAM** 52, 87  
**READ** 32, 165  
**RECALL** 66, 166  
**REDIM'D ARRAY ERROR** 123  
**REDO FROM START message** 17  
**Registers** 198, 199  
**Relative co-ordinates** 80  
**RELEASE** 54  
**REM** 13  
 abbreviation 18  
**REPEAT . . . UNTIL** 36, 38, 44, 163, 167  
**Reserved words** 11, 16  
 Reset button 8, 25  
**RESTORE** 121, 165, 168  
**Result, of function** 25  
**RETURN** 47, 168  
**RIGHT\$** 28, 169  
**RND** 29, 170  
**ROM** 52 routines and addresses 266  
**Rounding** 24, 27  
**RUN** 16, 20, 171

- Scientific notation, see Exponential notation
- Screen  
 attributes 72, 74  
 characters 85  
 colour 71  
 co-ordinates 76, 77, 79, 80, 83  
 display modes 53, 69ff  
 grids 238  
 memory locations 53, 64, 77, 84  
 saving on tape 65, 119
- SCRN 77, 172
- Semi-colon 70
- SGN function 29, 173
- SHIFT key 7
- SHOOT 93, 174
- SIN 29, 174
- Sorting 40
- SOUND 105, 177
- Sound  
 channels 96, 105  
 envelopes 100, 103
- SPACE key 7
- SPC 178
- SQR 29, 179
- Stack 35, 47, 167
- Status line 12, 77
- STEP 13, 131
- STOP 179
- STORE 66, 180
- String 9  
 numeric 26  
 slicing 28  
 variable 20, 59, 125, 132
- STR\$ 26, 76, 181
- Subroutines 48, 52, 134, 160
- SYNTAX ERROR 7, 11, 16
- TAB 70, 183
- TAN 29, 184
- Ten's complement 195
- TEXT mode 53, 69, 185  
 screen grid 238
- Text formatting 25, 69, 178, 183
- THEN 37, 139
- TO 31, 131
- Toggles 12, 71, 73, 75
- Tokens 57, 58, 288
- Tone channels 100, 105
- TROFF 185
- TRON 186
- TRUE 39, 44, 186
- TV 3, 81  
 tuning 4
- Two's complement 194
- UNTIL 187
- USR 122, 187
- VAL 26, 188
- Variables  
 assigning values 10, 17, 20  
 dummy 29, 44  
 floating point 59  
 garbage collection 132  
 integer 24, 59  
 loop 32, 35  
 names 11, 16, 20, 30  
 numeric 10, 23, 40  
 storage 59  
 string 20, 59, 60
- Verify facility 65, 116
- WAIT 19, 189  
 with MUSIC 97, 99
- ZAP 93, 189
- Zero page 53, 198