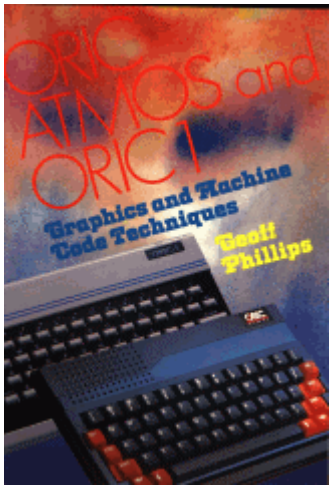

PREFACE

This book is for Atmos and Oric 1 users who want detailed information about their computer. For machine code programmers, an account of the various ROM calls is given with a full description of the methods of handling the different parts of the machine.

This book was not written to teach machine code, but to provide enough background information for existing 6502 programmers to use an Oric/Atmos.

If you are not an experienced machine code programmer, you will still find a great number of hints and tips in the book. Even if you do not understand machine code at all you will still be able to use the numerous utilities – such as Renumber, Merge and Auto.

Chapter one summarizes the hardware that makes up an Oric or Atmos computer.

Chapter two explains how BASIC works, from the way that programs and variables are stored, to creating different windows of scrolling text. A list of Oric 1 and Atmos bugs concludes the chapter.

Chapter three is about how machine code programs are entered, methods of calling your program, and how a machine code program can use the software timers. Some machine code pitfalls and tips are given at the end of the chapter, along with a real-time clock program.

Chapter four describes two important sections of Oric 1 or Atmos – the keyboard and the cassette system.

This chapter describes how individual keypresses are detected – very useful for games where several keys are used at the same time. A complete account of the cassette system is given, and after reading this chapter you will be able to write machine code programs that save and load blocks of memory, or individual bytes. A verify program is listed for Oric 1 owners.

Chapter five gives an account of how BASIC uses RAM and ROM. All important ROM and RAM addresses are printed, plus details of how the stack area is used.

Chapter six explores three important subjects – maths, HIRES and music. On the maths side, a machine code programmer will now be able to use the ROM's floating point routines. On the HIRES side, you will find out how the high-resolution graphics can be used with

different mixtures of text, and a complete account of the ROM routines for CURSET, DRAW etc. is given.

On the music side, this chapter describes how the ROM routines for MUSIC, PLAY and SOUND are used, as well as giving details of how the sound chip is accessed.

Chapter seven presents a number of fast high-resolution graphics routines. A single-point plotter is given which runs about 70 times faster than BASIC's CURSET command. A PAINT routine is listed that will fill in any shape on the high-resolution screen.

Chapter eight gives six utility programs to help BASIC programmers. These are: Renumber, Delete, Merge, Auto-Data, Trace, and ON-ERROR. Other utilities can be found throughout the book.

Chapter nine completes the book with some ambitious ideas, including a primitive form of speech synthesis, a multiprocessor and a program that allows single key entry of BASIC keywords.

Geoff Phillips

Preface to the 1998 edition.

Hi, and welcome to the on-line version of this book, first published in 1984 by McGraw Hill. I scanned this in during Summer 1998, using an OCR tool, combined with gif images for the program listing. I partially did this as a gift to the surviving Oric community, and also as an archiving exercise. I was fairly proud of this book, somewhat displeased when it was published to find it in no bookstores at all. Although it is hardly relevant today, it surely had some worth back when Oric was a going concern. Anyway, that's all water under the bridge.

My scanning skills improved somewhat during the making of this book, so you will find the later listings are sharper, bolder. In case you are interested, my trick was to use Paint shop pro: convert to large colour mode, at full size; then use Erode filter, which blackens the text; optionally do a flood fill of white; resize to about 340 across; use the sharpen more filter to highlight some of the edges; and finally reduce colours to 8 to make the image more compact. The OCR software was Cunieform. The old version I had of that was terrible with # signs, but I upgraded the version, and the newer one seemed to like them. There might be an odd escaped O - 0 problem.

Originally the book was written on a Microtan and saved onto cassette tapes. Now, those tapes contain music….

I'm hoping that much of the book remains useful to Oric users out there, emulated or real, the book has been scanned in almost without change - a few grammatical changes, for instance the changing of "data" to singular - the proof reader at Mc Graw Hill was convinced that "data" was a plural term [ok technically it is the plural of Datum, but no-one in their right mind uses this] So sentences like "The data is read" should be "The data are read" - the latter sounds horrible, to my mind "data" is a word that is always singular, like "grass". So where I've noticed this, I changed it back to how I originally typed it! I've put all the listings in as images, this means I can't introduce errors, since the listings have all come indirectly from those original programs dumped out from a running Oric. None of my code was written using an assembler, just the in-line one from Oric Mon. I do apologise for this, it seemed quite natural at the time, I don't think I'd have the patience now to work in this kind of manner, without proper cross assemblers! I can't vouch at this time if the programs are all working.

A lot has happened in the 14 or so years since the book was written. The rise of the PC, the commercialising of software, Windows, the rebirth of consoles. We've seen computers go from containing 16K of ram to typically 64 megs (4000 times), storage from 100K on floppy to 6 gigs on hard drive (60,000 times), processor power going from less than 1 MHz at 8 bit to 400 Mhz at 32 bit.

Personally, I stayed in the games industry for almost 10 years, doing conversions, and to be frank struggling towards the end to make a living out of it. Now, I'm not doing games, but working with new technology, associated with CDs, the Internet, video, and audio. I remain at heart an assembler man, but I'm learning C++ and windows, to keep up with the world.

If you want to reach me, I'm at binky@DeathsDoor.com, on ICQ at 104950, and have a home page which is linked via come.to/geffers. Email is always welcome.

CONTENTS

Page numbers are not given, because are irrelevant to the on-line version.

1 LOOKING INSIDE THE ORIC

1.1 Introduction

In this chapter we shall look at the various components of the Oric. Some of the features discussed will be further explored later in this book – the workings of the cassette system, for example.

1.2 The ROM

The Read Only Memory device contained in each Oric is responsible for supplying the BASIC interpreter program. It contains some 16K of instructions located between #C000 and #FFFF (on all machines). Since a program cannot overwrite the ROM area, the area #C000 to #FFFF is not affected by any write operations.

1.3 Use of RAM

Any BASIC program that you write is stored in the Random Access Memory located between 0 and #BFFF (or up to #3FFF for 16K users).

However, since the ROM needs a certain amount of working space, and because of other considerations, any BASIC program that you write will start at #501. The top of usable memory for your BASIC program is also going to be reduced, to at least as low as #B3FF, or #33FF for 16K machines.

If you are using high-resolution mode and have not issued a GRAB command, then the top of BASIC memory becomes #97FF (#17FF for 16K machines). This means that you have lost more than 11K! The actual layout and use of the RAM is described in more detail in Chapter 5.

1.4 Differences between machines

From the point of view of hardware, there are very few differences between machines.

There are two major categories:

1. Your Oric is either a 16K or a 48K machine.
2. Your Oric is either version 1.0 (i.e., the ORIC-1) or 1.1 (i.e., the ORIC ATMOS).

When you first power up your Oric, you will be advised of which version you are running. Chapter 2 lists the differences between the ROMs, but there is no apparent difference when looking at the hardware. Take note of your version number, so that you know which addresses apply to your particular machine.

The terms '16K machine' and '48K machine' relate to the total memory capacity. On a 16K machine, there would seem to be a gap between the end of the RAM (#3FFF) and the start of ROM (#C000). In practice, this is not the case, as the 16K of RAM is mirrored through each 16K block of addresses; e.g., location 0 is the same as locations #4000 and #8000. This is the reason why a program can still write to the screen at #BB80 on a 16K machine. Do not worry that this feature is 'accidental' and might not be true for all ORICs – the start-up routines use the mirroring to detect which machine is which.

Some very early machines have slightly different insides, but the only important difference is that the sound on these machines is much louder and can cause the break-up of a TV picture.

In this book, version 1.0 addresses are given first, followed by the version 1.1 address in brackets.

1.5 The microprocessor – 6502

The 6502 is the heart of the computer, obeying instructions held in ROM or RAM. When writing BASIC programs, the function of the 6502 is entirely invisible, but if you are going to write machine code programs, you will need to know quite a lot about this device. It is certainly worth while buying a book devoted to the subject. The programs in this book will help you to understand some aspects of machine code programming, and part of Chapter 3 gives a few guidelines on the use of some 6502 instructions.

1.6 The 6522 – VIA

The Versatile Interface Adaptor (VIA) is a microchip that belongs to the same family as the 6502 processor (hence the similar number). It is a complicated, but invaluable, device which links the Oric's 6502 to its peripherals, as well as providing two timers.

A book devoted to the 6502 will often have a chapter on the usage of the 6522; here we are only concerned with its use in connection with the Oric.

TALKING TO THE VIA

The 6522 chip is linked to page 3 of your memory map, so that whenever you read or write to an address between #300 and #30F you are enabling the VIA. These 16 addresses are normally mirrored throughout page 3 – so #380 is the same as #300 – but there is no reason to use any location between #310 and #3FF.

A quick summary of these locations follows; for more information you will need to use a book on the 6502 family of chips.

Address Description
#300 Port B in and out
#301 Port A in and out
#302 Define port B output or input (output if bits set)
#303 Define port A output or input
#304,5 Timer-1 counter
#306,7 Timer-1 latch
#308,9 Timer-2 counter/latch
#30A Shift register (not used by ORIC)
#30B Auxiliary control register
Ø30C Peripheral control register
#30D Interrupt flag register
#30E Interrupt enable register (indicates what sort of event will cause an interrupt).
#30F Read/write to port A without handshake.

CONTROL LINES ON THE 6522

The two ports can each contain one byte of information, but each bit can be separately set as input or output. In addition to the ports, there are four control lines, called CA1, CA2, CB1, and CB2. Here is a summary of how the Oric uses all of the I/O lines:

Port A – connects to the printer's 8-bit bus. It is also wired into the 8912 sound chip.

Port B – this port is easier to look at bit by bit. Starting from the right, the lowest three bits are used to supply the row when looking at the keyboard (see Sec. 1.10).

Bit 3 of port B is set to 1 when a key is pressed – more on this later.
Bit 4 is connected to the strobe line on the printer socket – when 0 the printer will expect data to be present on port A.
Bit 6 controls the relay circuit on the cassette socket.
Bit 7 connects to the cassette output circuitry.

CA1 – this line is input from the acknowledge signal on the printer port.

CA2 – this line connects to the 8912 sound chip (see Sec. 1.7).

CB1 – this line is connected to the cassette input circuitry. CB2 – when 1 the 8912 reads from port A of the 6522.

THE 6522 TIMERS

It is not often realized that the Oric has two versatile timers at its disposal. Later in this book it will be shown how easy it is to use these timers to provide a real-time clock facility – in BASIC or machine code.

The most important timer is designated timer-1 and is used mainly to count time between each interrupt. Without any supervision from the 6502, timer-1 counts down from a given 16-bit value (at location #306,7) to zero – this counter can be read from addresses #304,5.

When zero is reached, timer-1 starts counting again, using the 16-bit value stored at #306,7, and bit 6 in the interrupt flag register is set. When this happens, the 6522 will cause an interrupt signal to be sent to the 6502 processor. If the 6502 has interrupts enabled, then the appropriate interrupt handling subroutine will be called. It is very important to realize that the timer will operate regardless of the state of the 6502 – disabling interrupts does not stop the clock.

During cassette saving and loading, the VIA is set up differently, and the timers operate in a different fashion:

Timer-2, which is idle at other times, is used when receiving bits from the cassette input port in order to wait an exact amount of time.

The function of timer-1 is altered (by setting bits 6 and 7 of the auxiliary control register) so that instead of causing an interrupt bit 7 of port B is toggled and the timer is automatically set running again.

When a cassette operation is complete, the registers in the 6522 are set back to their initial values in order for the keyboard and printer to work normally.

1.7 The 8912 sound chip

All sound effects produced on the Oric are performed by the 8912 sound chip. In addition to being able to generate music, this device has one input/output port – port A – which is used to output the column number when polling the keyboard.

The 8912 is controlled by 15 eight-bit registers stored inside the chip. These are set up whenever the Oric's sound commands are executed. Here is a summary of how each register is used:

Register	Use
0,1	The lowest 12 bits give the pitch of channel A
2,3	The lowest 12 bits give the pitch of channel B
4,5	The lowest 12 bits give the pitch of channel C
6	The lowest 5 bits give the pitch of the noise channel
7	Enables: each bit has a different meaning: Bit 6: set port A as output or input
8	Bits 3,4,5: mix noise with channels A, B, and C Bits 0,1,2: enable channels A, B, and C

9 10 11,12 13	Channel A amplitude. If bit 4 is set then the music 'envelope' is used; otherwise bits 0 to 3 give the fixed volume Channel B as above Channel C as above Length of the envelope The lowest four bits give the shape of the envelope. This is different from the value you would use in the PLAY command, according to the following table: PLAY value Actual register value
	1 0,1,2,3, or 9
	2 4,5,6,7, or 15
	3 8
	4 10 or 14
	5 11
14	6 12 7 13 Register 14 is the I/0 port A.

The 8912 registers cannot be accessed directly by the 6502 – but instead via port A of the 6522 and a couple of control lines.

CB2 or the 6522 is set in order to select the 8912, and then immediately cleared (the 8912 chip will accept data as fast as you send it).

CA2 of the 6522 is either set when a register number is being passed in port A or cleared if it is data for the register. So in order to write #F7 to register 1, you would:

1. Store 1 in #30F – port A without any handshake signals.
2. Set CA2 and CB2.
3. Immediately clear CB2.
4. Put #F7 in #30F.
5. Clear CA2 and set CB2.
6. Immediately clear CB2.

There is a subroutine in the ROM to handle the above procedure – see Chapter 6 but as this is unbelievably inefficient, you will find a faster version used in the speech synthesis program of Chapter 9 (see 9.2).

1.8 Text screen

The text screen is organized as 28 rows by 40 columns of character cells. Each character cell occupies one byte of memory between #BB80 and #BFDF, but creates a display of a character 6 pixels wide by 8 pixels down.

The information for each character is retrieved by the graphics chip depending on the ASCII value of that character. Eight consecutive bytes are used for each ASCII character – one for each line of pixels. The formula for the start address of the definition of a particular character is: character value * 8 + start of character set.

The start of the character set (in TEXT mode) is #B400 for the standard character set (A to Z, 0 to 9, etc.) and #B800 for the alternate character set. Since these are in an area of RAM, it is quite a simple matter to redefine any character.

ATTRIBUTES

If the screen memory contains a control value, i.e., an ASCII value between 0 and 31, then this value is taken as an attribute, and the

character set is not referenced. This means that the first 256 bytes of both the standard and alternate character sets is wasted. Also, you may have noticed that the alternate character set overlaps with the screen!

An attribute changes the way that a particular line is interpreted by the VDU chip. Appendix C of the Oric manual (or Appendix 2 of the Atmos manual) gives the 32 possible attribute values. Some attributes – those between 8 and 15 – affect three different features of a line – double height, flashing, and character set.

At the beginning of each line, five attributes are always assumed:

1. No flashing.
2. Standard character set.
3. Paper of 0 (black).
4. Ink of 7 (white).
5. Single height.

If the character being displayed has a value between 128 and 255, then the character will be used as though 128 had been subtracted – except that whatever colours would have been displayed become inverted.

For instance, if you POKE #BB80 with 65 – i.e., the letter A – you will get a white A on a black background. If you POKE 65+128 instead, then the colours change – white (7) becomes 0 (7 – 7) and black (0) becomes white (7 – 0). This rule also works when you are setting a paper attribute: POKE #BB80,17 leaves a red square at the top left of the screen, whereas POKE #BB80,17+128 – although correctly setting the paper colour to red – creates a square which is cyan (7 – 1).

USING ESCAPE

One source of confusion lies when looking at how PRINT uses ESC (CHR$(27)) in order to set attributes. It is a good idea to totally ignore what the manual tells you about using ESCAPE when writing in machine code.

The important fact is that ESCAPE only works because BASIC is creating all the attributes for you – POKE 27 onto the text screen and nothing will happen. (Try poking it onto the high-resolution screen!)

The use of ESCAPE when using PRINT is unavoidable because this command traps any ASCII value less than 32 and treats them like control characters (changing parameters like keyclick, etc.).

The PLOT command, like POKE, does not understand escape sequences, so direct attributes must be used.

1.9 High-resolution mode

High-resolution mode moves away from using a character set, and instead causes the screen to directly reflect the contents of the video memory.

Each byte in the area #A000 to #BF3F affects 6 horizontal pixels in a matrix 240 across by 200 down. Part of the text screen remains at the bottom, at addresses #BF68 to #BFDF, although in high-resolution mode, these three lines use the character sets at #9800 to #9FFF.

This is necessary as the high-resolution screen overwrites the normal character set area. The exact details of how BASIC enters into high-resolution mode can be found in Chapter 6.

From a hardware point of view, the graphics chip switches modes when an attribute of 30 or 31 is interpreted. When an attribute of 26

or 27 is encountered, the mode is switched back to text. All the copying of character sets, etc., is carried out by software.

1.10 Keyboard

The keyboard on the Oric is a matrix of 8 columns connecting to 8 rows. By writing down one column and along one row, it is possible to examine the state of an individual key.

After every three interrupts, the system scans all the columns and rows in an attempt to find any depressed keys. The ROM subroutine only looks for one key down at a time, except that it does one extra search for the SHIFT and CONTROL keys. There is no reason, however, why a program cannot look at every key individually – this is very useful for games.

Two ports are used to poll the keyboard – port B of the 6522 and port A of the 8912.

The column is output on port A of the 8912 in the form of one bit cleared in a byte containing #FF. The row is output as a number (0 – 7) on port B of the 6522 and bit 3 of port B read back to determine whether that key is pressed.

Chapter 4 gives further details about reading from the keyboard.

1.11 Printer interface

When a byte is sent to the printer, the following occurs:

1. The byte is sent along port A of the 6522.
2. Bit 4 of port B is cleared and then set (this is the printer's strobe line).
3. The Oric waits until CA1 is pulsed by the acknowledge line on the printer. CA1 is not read directly, but causes bit 1 of the interrupt flag register to be set inside the 6522.

1.12 Cassette system

Chapter 4 explains how your programs can use the cassette system to save and load data, so this section is only concerned with some of the hardware aspects.

CASSETTE OUTPUT

The cassette output circuitry can only handle one bit at a time, creating either a high tone or a low tone depending on bit 7 of the 6522's port B.

This bit is set or reset automatically by the 6522 after timer-1 has finished counting down; the length of time to be counted depends on both the tape speed and whether the bit is 0 or 1. In order to save a whole byte, the cassette routines use a series of eight shift instructions to separate each bit.

In order for the 6522 to toggle PB7 at the end of each countdown, bits 6 and 7 of the auxiliary control register are set. Also, for no valid reason, port B is set completely to output (bit 7 is already in the output state). Because port B is zeroed in this process the printer's strobe is activated and an unwanted character is sent to the printer.

CASSETTE INPUT

The cassette circuitry connects to the CB1 line on the 6522. When this goes from low to high the CB1 flag is set in the interrupt flag register of the 6522. Timer-2 is used to count time before looking at the CB1 flag, and each bit is built up into a whole byte using a series of rotate instructions.

Finally, the cassette relay connection is activated before any of the cassette routines by setting bit 6 of port B on the 6522. This bit is cleared after all cassette operations, deactivating the relay.

2 BASIC

2.1 Introduction

An understanding of the workings of BASIC is necessary if it is required to incorporate machine code routines within BASIC programs, or if special utilities, e.g., 'Renumber', are to be written.

2.2 Memory map of BASIC

BASIC is rather greedy on the RAM – here is how it uses its memory:

#0000 – #00FF – almost all is used by BASIC – see Chapter 5.
#0100 – #010F – used when converting floating-point numbers to strings.

#0110 – #01FF – the normal 6502 stack area.

#0200 – #02FF – partially used by the non-standard parts of BASIC (e.g., DRAW and MUSIC).

#0300 – #03FF – an input/output area used by the 6522. This is not RAM.

#0400 – #04FF – not used by BASIC – reserved for use with the disk system.

#0501 – (#9C) – 1 – the BASIC program occupies memory as far as indicated by the address in locations #9C and #9D.

(#9C) – (#9E) – 1 – any simple numeric variables are stored here, along with the identification of each string variable.

(#9E) – (#A0) – 1 – numeric arrays are stored in this area, along with the identification of string arrays.

(#AO) – (#A2) – this area of memory is unused. It can be seen that pointer #A0 reaches up to meet pointer #A2 coming down.

(gA2)+1 – (#A6) – this area is used for storing both permanent and temporary strings of data. Temporary strings are only cleared when there is no more room below #A2, or when the FRE function is used.

#9800 – #9BFF – a copy of the standard character set is created here when a HIRES command is executed.

#9C00 – #9FFF – a copy of the alternate character set is moved here for use in HIRES.

#A000 – #BFDF – video memory used in HIRES mode.
#B400 – #B7FF – the standard character set when in TEXT mode.
gB800 – #BBFF – the alternate character set when in TEXT mode.
#BB80 – #BFDF – the video memory when in TEXT mode. Note that this overlaps part of the alternate character set.

BFEO – #BFFF – unused.

2.3 The format of a program

A program is stored in a completely different way from its external appearance. If you enter a simple program and then use PEEK to see what has been entered, you will not find evidence

of either the program keywords (such as SHOOT) or of line numbers.

Each line is stored in its correct place in the program in an exact way. Consider the example:

10 POKE4,3 20 END

Here is how that is translated:

#0501,2	Link address to the next line in the program – in this case, #50A. (Remember that the low byte of the address is
#503,4	always first.) Two-byte binary form of the line number, e.g., #0A.
#505	A one-byte 'token' which means 'POKE' – #B9. All BASIC keywords have a unique token value, always between t80 and #FF, as this conserves memory and makes it quicker to execute an instruction (Table 2.1, page 12, gives a list of all possible tokens).
#506	The ASCII code for '4' – 434.
#507	The ASCII code for comma – #2C.
#508	The ASCII code for '3' – #33.
#509	An end of line indicator of #00.
#50A,B	The link to the next line – #510.
#50C,D	Line number 20.
#50E	Token for END – 080.
#50F	End of line – #00.
#510,1	End of program's link field – always contains a value < 256. In other words, #511 must be zero, but #510 could be anything.
#512	Start of free space.

Table 2.1 List of all BASIC tokens

#80. END	#81. EDIT
#82. STORE	#83. RECALL
#84. TRON	#85. TROFF
#86. POP	#87. PLOT
#88. PULL	#89. LORES
#8A. DOKE	#8B. REPEAT
#8C. UNTIL	#8D. FOR
#8E. LLIST	#8F. LPRINT
#90. NEXT	#91. DATA
#92. INPUT	#93. DIM
#94. CLS	#95. READ
#96. LET	#97. GOTO
#98. RUN	#99. IF
#9A. RESTORE	#9B. GOSUB
#9C. RETURN	#9D. REM
#9E. HIMEM	#9F. GRAB
#A0. RELEASE	#A1. TEXT
#A2. HIRES	#A3. SHOOT
#A4. EXPLODE	#A5. ZAP
#A6. PING	#A7. SOUND
#A8. MUSIC	#A9. PLAY
#AA. CURSET	#AB. CURMOV
#AC. DRAW	#AD. CIRCLE
#AE. PATTERN	#AF. FILL
#B0. CHAR	#B1. PAPER
#B2. INK	#B3. STOP
#B4. ON	#B5. WAIT
#B6. CLOAD	#B7. CSAVE
#B8. DEF	#B9. POKE
#BA. PRINT	#BB. CONT
#BC. LIST	#BD. CLEAR
#BE. GET	#BF. CALL
#C0. !	#C1. NEW
#C2. TAB(	#C3. TO
#C4. FN	#C5. SPC(
#C6. @	#C7. AUTO
#C8. ELSE	#C9. THEN
#CA. NOT	#CB. STEP
#CC. +	#CD. -
#CE. *	#CF. /
#D0. ^	#D1. AND
#D2. OR	#D3. >
#D4. =	#D5. <
#D6. SGN	#D7. INT
#D8. ABS	#D9. USR
#DA. FRE	#DB. POS
#DC. HEX$	#DD. &
#DE. SQR	#DF. RND
#E0. LN	#E1. EXP
#E2. COS	#E3. SIN
#E4. TAN	#E5. ATN
#E6. PEEK	#E7. DEEK
#E8. LOG	#E9. LEN
#EA. STR$	#EB. VAL
#EC. ASC	#ED. CHR$
#EE. PI	#EF. TRUE
#F0. FALSE	#F1. KEY$
#F2. SCRN	#F3. POINT
#F4. LEFT$	#F5. RIGHT$

Note that the tokens listed are those for Vl.l ROMs. The only differences for V1.0 ROMs are:

'STORE' is 'INVERSE' and 'RECALL' is 'NORMAL'.

The use of the link address is to allow a quick method of locating a specific line. You can try this yourself by typing:

I = #501: REPEAT: J = I: I =DEEK(I): UNTIILI <256: PRINT J

which finds the highest address of your program. Since the links affect the 'LIST' command, you can have endless fun altering the links of a program so that, for instance, a program lists itself backwards!

Since the line number is always stored in a 2-byte binary format, it must be realized that there is no saving in having a line number of 5 as opposed to 50 000 – except where a GOTO or GOSUB occurs. GOTO 12345 takes up 6 bytes, but GOTO 5 only needs 2 bytes.

2.4 Pointers

As mentioned in the memory map, there are a number of pointers used by BASIC to separate a program from its variables and arrays. Not all of these are useful: #9A is the start of the BASIC pointer, but BASIC refuses to work if you move it from its normal value of #500.

The most important pointer is that at #9C which gives the address of the start of BASIC variables – or the end of the BASIC program + 1. Printing the DEEK of #9C is often more useful than the FRE command since it gives you the exact position of the end of your program. When a program is saved, this pointer is used to give the upper address limit. It follows, therefore, that by adjusting the pointer at #9C you can save more than just the BASIC program using a single CSAVE – though remember to DOKE the correct value before you do anything else after you have loaded back. The Oric assumes that #9C is always correct, adding or subtracting values as a program is altered.

When a BASIC program is loaded the upper load address is automatically stored back at #9C. Version 1.0 owners should beware of loading machine code programs in on top of BASIC programs since the #9C pointer will then point to the end of that machine code section. The solution to this is to either correct #9C or load machine code routines before loading a BASIC program. Version 1.1 owners need not worry about this particular fault.

HIMEM

The HIMEM command is often most unhelpful – especially on V1.0 machines. In cases where you cannot persuade your machine to do HIMEM correctly, simply DOKE #A6 with the value before running the program. If you wish this to be done as part of your program, you will also have to alter #A2 to the same value as #A6, otherwise strings will be placed in the wrong part of memory.

2.5 Numeric variables

All calculations are done in 'floating-point' arithmetic. This means that an expression such as '1+1' presents as much difficulty as '3.1415+9.7373'.

When you assign a value to a variable, as in 'LET A=52', this variable is stored away in a 7-byte area comprising:

Two bytes containing the identification 'A'.

Five bytes containing the floating-point representation of the number. The exact format of these 5 by'tes will be described in Chapter 6.

The identification is simply the first two characters of the variable's name, or one character followed by #00. The top bit in each of these can be set for the different types of variables – for a normal numeric variable both bits are clear.

For the fastest possible calculations, always use simple numeric variables. It must be stressed that '10 I=I+4' is slower than '10 I=I+ J'.

2.6 Integer variables

These are stored in the same amount of memory as for normal variables, but the format is different:

1. Two bytes of identification (as before) with the topmost bits set in both bytes.
2. A 2-byte binary value of the integer stored in twos-complement form with a high byte followed by a low byte (i.e., against the usual convention).
3. Three unused bytes containing zero!

The advantage of using integer variables is only where it would save the use of INT. Contrary to many magazine articles stating the opposite, there is no saving in a program that uses integer variables (but see integer arrays!).

2.7 String variables

Any string variable has two components:

1. An identification of the variable's name, occupying 2 bytes, as for numeric variables. To identify the variable as a string the second byte has the top bit set. This identification is followed by the length of the string, the address of the string, and two spare bytes.

2. The string of characters must be located somewhere in memory.

The first component is in the area between (#9C) and (#9E) – as for any numeric variable. The second component, however, can be in two distinct areas:

1. If a program assigns a definite value to a string variable, with either READ or LET, then the first component of the string points to the place in the program where the string has been entered. So, unlike some other computers, the Oric does not waste memory space by repeating the same set of characters.
2. If a string is modified in some way, e.g., LEFT$ is used, or one string is moved to another then the resultant string is placed in the string temporary space which lies between the top of available memory and the end of array space. The pointer to the next string space works backwards through memory so that new arrays can be added without the need to reorganize the strings. Since a string could be created that makes an earlier version redundant, it should be noted that the string area will eventually become full. When this happens, or when the FRE function is called, a subroutine known as 'garbage collection' is entered and all unwanted strings are removed. Garbage collection can occur at any time when a string is being created and can take several minutes to complete. The length of time that garbage collection takes is in direct proportion to the quantity of permanent strings.

2.8 Arrays

Each element of an array is stored in the same format as an equivalent single variable, but without the wasted space. For the integer arrays only 2 bytes are needed per number stored.

An array is stored in sequential order in memory, e.g., consider the array A(1,1,1). The array is stored working on a left-to-right basis:

A(0,0,0),A(1,0,0),A(0,1,0),A(1,1,0),A(0,0,1), etc.

For each array there is an overhead of at least 7 bytes in the memory area between pointer #9E and #A0

This area is made up as follows:

1. Two bytes identifying the array name – exactly as for variables, with the top bits set or cleared to indicate the type of array.
2. A 2-byte binary length which gives the exact amount of memory occupied by this array (excluding the text part of a string).
3. One byte which gives the number of dimensions.
4. For each dimension, working from right to left, there is a 2-byte number which gives the dimension plus one (remember that you can have a zero subscript when accessing part of an array). This number is stored with the high byte followed by the low byte.

2.9 READ and DATA

It is often useful to be able to use READ in a more controlled way – reading from a particular line of DATA, Some more advanced BASICs have this facility – this is often known as RESTORE N, where N is the line number from which DATA is to be read.

The READ command does not keep account of the next line number from which to read, but instead uses #B0 store the last address in memory where DATA was read. After each READ command, the line number used is stored in #AE,F so that an error message can report on the current data line (for 'OUT OF DATA', etc.). Writing to #AE,F will have no effect on READ operations.

```
04E0:  A5 00      LDA  $00
04E2:  85 33      STA  $33
04E4:  A5 01      LDA  $01
04E6:  85 34      STA  $34
04E8:  20 E4 C6   JSR  $C6E4
04EB:  A5 CE      LDA  $CE
04ED:  38         SEC
04EE:  E9 01      SBC  #$01
04F0:  85 B0      STA  $B0
04F2:  A5 CF      LDA  $CF
04F4:  E9 00      SBC  #$00
04F6:  85 B1      STA  $B1
04F8:  60         RTS
04F9:  EA         NOP
04FA:  EA         NOP
```

A RESTORE N FACILITY

Only a very short machine code program is needed to give BASIC this facility, which has been listed below in Program 2.1. Although the routine has been put at address #4EO, it will work at any spare memory location.

Version 1.1 ROM owners should change #4E8 to ‘JSR #C6B9’.

The machine code routine takes the line number stored at address 0,1, calls a ROM routine to find the address of that line, and stoics that address minus 1 at #B0 to #Bl.

USING RESTORE N

A BASIC program has been listed below (Program 2.2), for V1.0 owners, which demonstrates how to call the machine code routine. Program 2.3 is the listing for V1.1 owners – the only difference is the JSR address in machine code.

```
10 A$="A5008533A501853420E4C6A5CE38E90185B0A5CFE90085B160"
15 Z=#4E0
20 FORI=1TOLEN(A$)/2:B=VAL("#"+MID$(A$,(I-1)*2+1,2)):C=Z+I-1:POKEC,B
30 NEXT
100 INPUT"WHICH LINE?";L
110 DOKE0,L:CALL#4E0
120 FORI=1TO3:READA$:PRINTA$:NEXT
130 GOTO100
1000 DATA43,55,66,77,88,99
1010 DATATHIS IS LINE 1010
1020 DATA66,77,88,99,66,66
2000 DATALINE 2000 DATA
3000 DATA55,6,4,4
```

Program 2.2 Restore N – BASIC example for version 1.0

```
~
5 REM NEW ROM VERSION OF RESTORE X
10 A$="A5008533A501853420B9C6A5CE38E90185B0A5CFE90085B160"
15 Z=#4E0
20 FORI=1TOLEN(A$)/2:B=VAL("#"+MID$(A$,(I-1)*2+1,2)):C=Z+I-1:POKEC,B
30 NEXT
100 INPUT"WHICH LINE?";L
110 DOKE0,L:CALL#4E0
120 FORI=1TO3:READA$:PRINTA$:NEXT
130 GOTO100
1000 DATA43,55,66,77,88,99
1010 DATATHIS IS LINE 1010
1020 DATA66,77,88,99,66,66
2000 DATALINE 2000 DATA
3000 DATA55,6,4,4
```

Program 2.3 Restore N – BASIC example for version 1.1

2.10 Using RND

The RND function will start from the same sequence of numbers every time you start up an Oric, providing the argument which follows RND is positive. Although it is not made clear in the manual, when the argument is negative this starts off a new sequence of random numbers.

It follows that in order to make RND truly random, you must supply it with an initial negative random seed. One of the software timers, incremented 100 times per second, can be employed here. Unless you do a WAIT command, and providing there has been some sort of user input (to delay the machine by an unknown time), you can use the third timer at #276,7. For example:

```
5  GET Z$

10 A=RND( – DEEK(#276))
```

Note that A itself is not a very random number – it will usually be a number smaller than 0.01 – but any RND afterwards should be correctly balanced between 0 and 1.

2.11 Using a printer

It is often required to make a choice as to whether to print something on a printer or on the screen. Since PRINT and LPRINT are different commands, it would seem that a program would need two separate lines to handle any one PRINT statement. Fortunately for us, the LPRINT command can be achieved by poking 255 into #2F1 and using PRINT. This will stay in force until either:

1. A proper LPRINT command has finished.
2. The program returns to command mode.
3. Address #2F1 is reset to zero.

Note that this affects all types of PRINT – even the printing of prompts on INPUT commands!

2.12 The Oric's status bytes

There are two locations in page 2 which are concerned with the status of the keyboard and the screen.

The first of these is at #20C and controls the CAPS lock function. This location is 127 when CAPS is off and 255 when on. If you put any other value into 420C, then the Oric will no longer respond correctly.

The most important status location is at #26A. The lower 6 bits of this byte each have their own meaning:

BIT 0 – cursor ON when set.
BIT 1 – screen ON when set.
BIT 2 – not used.
BIT 3 – keyboard click OFF when set.
BIT 4 – ESC has been pressed.
BIT 5 – columns 0 and 1 protected when set.

This means that you can POKE into #26A in order to turn off keyboard click, etc., rather than the unpredictable method of printing control characters.

For example, POKE #26A,10 turns off keyboard click and the cursor.

2.13 INVERSE and NORMAL

Version 1.0 owners will recognize these two commands as they crop up when listing all the tokens. Version 1.1. users have STORE and RECALL instead, but what did INVERSE and NORMAL actually do?

Although the commands do not actually work, on the V1.0 machine there are still some instructions that relate to them. The theory is that if you set the top bit when displaying a character on the screen, it is printed in 'inverse' colours – this has been explained in Chapter 1.

What remains in old ROM Orics is the code which OR's location #2F7 (the inverse flag) with any character as it is printed. Unfortunately, PRINT nearly always strips off the top bit – otherwise it would be possible to use POKE #2F7,128 to create an INVERSE facility on the old ROM Oric. You can have some fun though putting different values into #2F7 and watching PRINT go haywire!

Incidentally, the only place where PRINT does not take off the top bit (again, only for version 1.0 Orics) is where control-D double height is in force, and when the second line is printed.

2.14 Creating windows of text

The normal way of presenting 27 lines of scrolling text is by no means fixed. It is possible with just a handful of DOKE commands to make just part of the screen scroll up – leaving the rest of the screen untouched. This has many uses where part of the screen is being plotted.

Here are the DOKEs needed for version 1.0 machines:

1. DOKE #26D with the start address where scrolling is to begin minus 40.
2. POKE #26F with the number of lines which are to be scrolled.
3. You must clear the screen after doing these commands.

For version 1.1 ROMs, the procedure is:

1. DOKE #27A with the start address of the screen.
2. DOKE #278,DEEK(#27A)+40.
3. POKE #27E with the number of lines to scroll.
4. DOKE #27C, (PEEK(#27E) – 1) *40 – this is the number of characters to be scrolled up and must agree with location #27E.

The CLS command should be issued after setting up a different format for the screen.

2.15 Controlling PRINT

On version 1.1 machines the PRINT @ facility allows you to print at any place on the screen. This is also provided on 1.0 machines by way of an add-on machine code routine in the manual, but no explanation is given on how it works. If you wish to use the general PRINT subroutine in a machine code program, you will need to know a little about how PRINT works in this respect.

There are two locations which control where the next PRINT goes to: #268 – the number of lines down – and *#269* – the number of lines across. These are relative to the start of the screen as defined by #26D (version 1.0) or #27A (version 1.1). On version 1.1 machines you also have to write the address of the start of the line to #12,3.

On version 1.0 follow this example of moving to D lines down and A characters across:

100 POKE #268,D – 1:PRINT:POKE #269,A

Here is the same line for version 1.1:

100 POKE#268,D:POKE4269,A:DOKE#12,DEEK(#27A)+(D – 1) *40

To avoid large numbers of solid blocks appearing everywhere, it is recommended that you turn off the cursor before moving around the screen.

2.16 Bugs in BASIC

Most people will be aware of one or two problems with version 1.0 BASIC, the most notable example being the TAB function, which is quite useless (although the previous section should help with the problem).

In this section, we look at all the bugs and, where relevant, how they can be overcome. First of all, here are the quirks found in version 1.0 machines.

1. TAB and COMMA do not work correctly. It is best to use either SPC or, alternatively, POKE #269 with the TAB position.

2. STR$, when packing a positive number, puts the attribute ‘2’ at the front instead of a space. This often results in green numbers! The cure is to use MID$ to take off the unwanted character or to define a new STR$ function using the & function.

3. ELSE does not work under several conditions, for different reasons, so it is best to simply avoid the command altogether.

4. HIMEM is not set correctly on power-up. The solution is to always put in a HIMEM command at the start of the program, e.g., HIMEM #97FF.

5. When in high-resolution mode, the message ‘SAVING’ is still output to address #BB80 – putting one line of junk onto the screen. There is no easy cure for this problem, apart from writing your own save-to-tape routine. If you are saving a high-resolution screen, then first copy it to a free area of memory and save that part of memory.

6. When the printer is in the middle of either an LLIST or a series of LPRINTs, characters are often corrupted into ‘squiggles’. This is because the interrupt routines which read the keyboard frequently conflict with the use of the printer. The solution is to stop the clock (CALL #ED01) before printing and to start it again after printing is complete (CALL #ECC7). If you are using LLIST, then you can type:

CALL #ED01: LLIST

and then use the Reset button underneath.

7. When you use CLOAD from within a program, BASIC unkindly ends the program once the load is complete. To get around this, you could do a series of CALL instructions instead of CLOAD. Chapter 4 contains all the necessary information.

8. The function HEX$ has an unfortunate tendency to print just the hash sign for zero. This

condition should be specially tested for in your program.

9. The GET command refuses to believe that you have pressed the single quote key and instead returns an empty string (“ “). It is important that you test for this condition before using one of the functions such as ASC.

10. If a print line starts with control characters – e.g., ESC N, etc. – then the protected columns 0 and 1 are used, overwriting any PAPER and INK attributes. Always start the line with a non-attribute character, such as space,

11. The alternate character set is exactly one bit out of place! The purpose of the alternate character set, when not modified for a special use, is to provide a 'chunky' graphics capability. The format of such characters is identical to that used in the BBC's CEEFAX system, allowing a resolution of 80 chunks across by 84 chunks down. Each character cell contains six such chunks, which means that 64 graphics definitions are required to allow for all possibilities. The Oric's character set has in fact been set up for this. Characters between 32 and 95 contain all variations between a totally blank cell and a filled cell. However, in version 1.0 the entire character set must first be divided by 2 (and therefore shifted to the right) before it can be used. This can be done either with a simple BASIC loop:

```
FORI = #B900TO #BAFF: POKEI,PEEK(I) /2: NEXTI
```

or by using a short machine code routine:

```
LDY 000

LOOP: LSR B900, Y

LSR BA00, Y

DEY

BNE LOOP

RTS
```

12. If the single quote character is found at the start of a DATA item, then because of confusion with the REM facility, the rest of the DATA line is ignored. Use double quotes around any DATA items containing single quotes.

13. When loading in a machine code program, be warned that the 'end of BASIC' pointer at #9C,D is altered to reflect the end address of the machine code.

To overcome this you could either reset the value at #9C to #9D after the load or make it a rule to always load the machine code routines first.

14. In the instruction POKE N, #8, the hexadecimal sign upsets BASIC, and zero will be POKEd. Always use a decimal value or a variable instead. This fault is the reason why you will often see decimal numbers mixed with hexadecimal numbers in this book.

The DOKE command does not suffer from this fault.

15. One interesting bug is that POINT will work in text mode!

16. When loading a file, the filename is only printed when it is actually supplied within the CLOAD"" command

.

17. Although potentially useful, it is still a fault that makes the screen scroll down when the cursor is moved too high.

The following faults lie in version 1.1 ROMs:

1. ELSE fails to work should the colon character occur in quotes after the ELSE. For example: IF A=1 THEN PRINT ELSE PRINT "HELLO:".
2. One very obscure problem arises when:
   (a) The cursor has been turned off.
   (b) A character is placed at the very spot where the cursor would have been.
   (c) That character is 'inverse' – between 128 and 255.

When this happens, and providing interrupts are running, that character is forced back to 'normal' mode – losing the top bit of the character.

One solution for this problem is to force the current cursor position to a place on the screen (or even off the screen!) where it can do no harm. This is done by poking locations Ø268 and t269 as described earlier.

3. One very minor bug is that going into HIRES when in control – S mode results in BASIC writing to the wrong part of the screen. Make sure that you have enabled the screen before using the HIRES command.

3. USING MACHINE CODE

3.1 Advantages of machine code

BASIC, though easy to use, hard to misuse, and ideal for simple programs, has two serious drawbacks:

1. It is very slow to run.

2. It can often (but not always) use up a large amount of memory space.

One alternative language, FORTH, although faster than BASIC, is quite difficult to use. It is unlikely that you would ever see a program on the market which used FORTH, for the simple reason that the FORTH language would have to be sold as well.

Machine code, on the other hand, can be loaded and executed on all Oric machines. Indeed, in many cases a machine code program will be easier to convert to a different machine than its BASIC equivalent.

The speed of a computer like the Oric is not always appreciated. A simple machine code instruction takes two microseconds to complete, whereas any single BASIC command will take at least 2 milliseconds.

If you intend using machine code you will quite definitely need two things, in addition to this book:

1. A book on the programming of the 6502.

2. An assembler/disassembler program. The one used in the preparation of this book was ORICMON from Tansoft I.td. Without such a program, you will have to work out the machine code instructions by hand. An assembler allows you to enter just a three character mnemonic – such as LDA – and it works out the actual machine code values – e.g., LDA # is #A9.

A full discussion of machine code is beyond the scope of this book, but at the end of this chapter you will find some advice on the more difficult aspects of this subject. The book 6,502 *Software Design* by Leo Scanlon is particularly recommended as both a tutorial and a reference guide.

3.2 Storing machine code

A programmer has no choice as to where a program written in BASIC resides – he or she is stuck with the area #501 upwards.

A machine-code programmer has the whole of the machine available, at least in theory. If a machine code program will never return to BASIC, or use a subroutine in the ROM, then that program can be located anywhere between #400 and #B4FF, and can use the area #00 to #2FF as a scratchpad area (not forgetting to allow a certain amount of room for the stack).

The programs and subroutines in this book are of the kind that always return to BASIC, so it is important not to upset BASIC too much. This means not overwriting certain RAM areas in pages 0 and 2 and allowing BASIC to create variables and strings. You can use HIMEM to limit BASIC's memory, and can thereafter use the remaining memory for your own needs. Chapter 5 explains which areas of page 0 and page 2 RAM are used by BASIC.

If you are writing an add-on machine code program in order to manipulate a BASIC program, then you really want to put your program in a place which is unused. The most common of these are:

1. The stack area – from #110 upwards – can be used by short programs. Providing that you do not do many GOSUB, FOR, or REPEAT commands, you will be able to use up to about #1CO. The stack area is never cleared by BASIC, except during normal use.
2. From #400 to #4FF, 256 bytes are available. Be warned, however, that the Oric disk system makes use of this area.
3. The first 256 bytes of each character set are unused, so programs can be put at #B400 to #B4FF and #B800 to #B8FF (or in HIRES mode at #9800 to #98FF and #9C00 to #9CFF). Although the Reset button on the Oric causes the character set to be regenerated these areas are not affected.
4. Since the alternate character set is rarely used the entire area between #B800 and #BB7F

is available for a machine code program. This area of RAM is ideal for facilities like Renumber.

5. Another 'hidden' area lies between #BFEO and #BFFF. This area will only be overwritten if HIMEM is incorrectly set, and survives the commands 'HIRES', 'TEXT', and the Reset button.

3.3 Types of machine code program

When you write a program that is all in machine code you do not need to worry about interfering with BASIC. If your program calls the BASIC ROM for certain functions you should keep clear of the same areas of RAM that the particular subroutine uses. For instance, if using the MUSIC command keep away from the parameter area #2E0 to #2EF.

Since a machine code program can be made to autorun at the start address of the load, it makes sense to use this feature and make your program start at the earliest address.

If you are using an Assembler program, such as ORICMON, you will also have to avoid the area of RAM used by that program.

A common type of machine code program is used when a BASIC program needs an extra facility, or perhaps a machine code subroutine is used to speed up part of the program. In this case the BASIC program will often use DATA statements in order to set up the machine code. A more efficient way, for larger sections of code, is to load in a separate machine code file from tape or disk.

Another method is to put the machine code after the BASIC code and modify the #9C pointer before saving to encompass the machine code. The first instruction in the program should reset the pointers #9C, #9E, and #A0 back to the end of the program.

For example:

BASIC program #501 – #IF00
M/C program #2800 – #2E00

Before saving, DOKE #9C, #2E00. In the program:

```
1 DOKE #9C, #1F02:CLEAR
```

An example of a BASIC program creating a machine code subroutine can be found below in Sec. 3.4.

The third type of machine code program occurs where a BASIC program is being modified. Normally such a routine will be loaded separately from the BASIC, although you must remember to reset the #9C pointer on version 1.0 machines – this can often be done by the machine-code routine itself.

3.4 Creating a machine code program

Nearly all the programs in this book have been listed in terms of the assembly mnemonics and the actual machine code. In order to set up the programs you are best advised to use a machine code monitor/ assembler package. If such a facility is not available, you can quite easily use a short BASIC program to read in machine code.

Program 3.1 is an example of a program to read in a short section of code by using DATA statements.

The program itself is very useful, as it totally disables the use of control – C. This works by testing for ASCII code 3 in a routine that is patched into the slow interrupt link.

```
5 REM IGNORE CONTROL-C

10 FOR I=#BFE0 TO #BFEE:READ D:POKE I,D:NEXT I

20 DATA #8,#48,#AD,#DF,#2,#C9,#83,#D0,#3,#CE,#DF,#2,#68,#28,#40

30 IF PEEK(#D000)=166 THEN DOKE#231,#BFE0:POKE#230,76

40 IF PEEK(#D000)<>166 THEN DOKE#24B,#BFE0:POKE#24A,76
```

Program 3.1 Disable control – C

3.5 Calling a machine-code routine

A machine code program which is completely self-contained can be automatically run by using the AUTO command. Alternatively, a CALL can be used to start the program off.

Where a BASIC program calls a m/c subroutine, CALL is often used. If a CALL is to return to BASIC the subroutine must end with the RTS (#60) instruction. Do not worry about saving registers when writing such a subroutine.

CALL is also useful when entering add-on subroutines, such as 'Renumber', when it is used as an immediate command.

In addition to CALL NN, there are several alternatives:

I. USR and & functions.

2. ! – the extension command.

From the point of view of a machine code subroutine, CALL NN is much the same as!, and USR(X) is identical to &(X). One difference is in the setting-up. For the extension command '!' you DOKE the start address into #2F5, and for '&' you DOKE the address into #2FC. The USR facility uses DEFUSR in order to set up the start address.

The difference between '&' and '!' (or USR and CALL) is that & is a function that returns a value; the ! command can only take in values. The rest of this chapter will only deal with & and !, although the same considerations apply for CALL and USR.

3.6 Passing information to machine code routines

The most common method of passing small amounts of data to a machine code routine is with the DOKE and POKE commands. For small data areas, such as for addresses, use the area #0 to #B in page 0. Chapter 5 will help you in determining other areas of memory available.

The! and & keywords can both take parameters, e.g., &(A1*3), and this will be explained in Sec. 3.11.

A machine code routine could read a BASIC variable, but this would involve quite a bit of searching and conversion.

3.7 Patching into BASIC

Although BASIC is in unalterable ROM, there are several cases where it jumps out to an area of RAM. The reasons for doing this are:

1. It lets programmers patch in extra facilities.
2. It allows for add-ons, such as disks.
3. It can he more efficient to write some instructions in page 0.

Each of the patch areas has been listed below, with the address for version 1.1 ROMs given in brackets:

At #1A – a jump vector to the routine that prints 'READY'. By changing this jump to your

own routine it is possible to:

1. Trap errors.
2. Prohibit control – C.

See the ON – ERROR facility of Chapter 8.

At #E2 lies a very important subroutine. At #E2 the address at #E9,#EA is incremented. Then at #E8, the contents of the address at #E9, #EA are loaded. This provides a very fast subroutine for reading in characters from the program.

> After getting the next character, the routine jumps back into ROM. It is a very simple matter to alter the routine at #E2 in order to jump to your own subroutine. By doing this, you can look for special instructions (perhaps 'IMPLODE' and 'PONG'!). ~~~ important consideration is that you jump back into the ROM as though nothing had happened – remember to save all the registers.

#228 (4244) is the address of the 'fast' interrupt jump. By altering the jump address at #229,A (#245,6) you can provide your own interrupt handler.

#230 (#24A) is the address of the 'slow' interrupt routine. Control is passed to here at the end of the fast interrupt routine. Although 3 bytes are reserved here, there is only the single-byte instruction RTI present normally.

#228(4247) contains the jump vector for the NMI (Non-Maskable Interrupt) routine, which on the Oric connects to the 'Reset button'.

On version 1.1 only, there are a few extra jump vectors located in page 2 which are concerned with input/output:

I. #238 links to the screen output routine used by BASIC commands like PRINT.

2. #23B jumps to the subroutine which finds which key was last pressed.

3. #23E jumps to the printer output subroutine.
4. #241 contains a jump to the subroutine that prints messages on the top line of the screen. Changing this jump could be useful if you want to stop messages like 'Loading' from showing.

By far the most useful of these patches is the slow interrupt jump which allows you to make the maximum use of the system's interrupts.

3.8 Interrupts

The purpose of an interrupt is to stop a program temporarily and to enter a special subroutine in order to handle a priority condition. An interrupt on a computer will often be caused by a peripheral (such as a card-reader) announcing that it has data to transfer.

The Oric takes its interrupt line from the 6522 VIA device which is capable of causing an interrupt for a variety of reasons. Unless the Oric is loading or saving to the cassette port, the 6522 is set up to create an interrupt at exact intervals of 10 000 machine cycles – or every 10 ms. In other words, the machine is interrupted every one-hundredth of a second. (You should be warned that some BASIC instructions may cause an interrupt to be missed – e.g., PRINT.)

The length of time between interrupts is stored on the 6522's timer-1 latch at #306,7. By altering locations #306,7 you affect:

1. The repeat rate on the keyboard.
2. The flash rate of the cursor (but not the automatic flash of the VDU chip).
3. The speed of the WAIT command.
4. The speed of processing is inversely affected. This happens because the interrupts 'steal' time from the processor; the more time spent in interrupt handling, the less is available for the main task.

When an interrupt occurs, and providing that the 'fast interrupt' jump vector has not been altered, the following events take place:

1. The three software timers are decremented by one. These are 16-bit counters located in page 2 of memory and will be discussed in Sec. 3.9.

2. If the first timer has reached zero, after counting down from 3, the keyboard is scanned in a search for any keypress.

3. If the second timer has reached zero, counting down from 25, the cursor is flashed on or off.

Note that the timers being discussed are merely counters in RAM, and should not be confused with the timer-1 and timer-2 of the 6522.

When an interrupt occurs, the 6502 jumps to the address given by locations #FFFE and #FFFF. As was discussed in Sec. 3.7 this address is in page 2 of RAM, and the jump into ROM can be modified for one's own requirements.

If the fast interrupt routine does jump into ROM the last operation is to jump back to the slow interrupt location in page 2, containing the RTI instruction.

You would use the fast interrupt patch if you wanted to add some processing before the keyboard is scanned. The slow interrupt link allows you to add some processing after the keyboard has been scanned.

If you intend to modify the interrupt routines, remember:

1. Save all the registers that you use, and restore them before you finish.
2. Save any locations that might be in use by the system. For instance, if your interrupt routine calls the SOUND command you will need to save locations #2E0 to #2EF and #204 (#204 is used when checking your SOUND parameters).

At the end of your interrupt routine, you will usually either execute the RTI instruction if all interrupt processing is complete, or jump back into the normal ROM interrupt routine (to read the keyboard, etc.).

Writing interrupt routines is much more difficult than writing a normal subroutine. For one thing, testing can frequently crash the whole machine, and often a fault will not show up for a long time. Two important points are:

1. Remember to save any location that could be used by both your interrupt routine and the main program.
2. Do not assume the state of any of the processor flags. Be especially wary of the decimal flag – use CLD or SED if you are doing any addition or subtraction.

Several programs in this book modify the interrupt patches, and by understanding how these work you will be able to create your own routines.

NON-MASKABLE INTERRUPT

The Reset button on the Oric does not in fact connect to the RESET line of the 6522. Instead, it activates the Non-Maskable Interrupt (NMI) line of the 6502. Whereas a normal interrupt can be disabled, the NMI causes an unconditional jump to the address contained in locations #FFFA, #FFFB. On the Oric, this is a jump instruction in page 2 of memory which on the Oric normally leads to a 'warm-start' routine in ROM. This sets up the 6522, clears the screen, initializes the character sets, and returns to command mode in BASIC.

When writing machine-code programs it is customary to alter the appropriate address in page 2 (see Sec. 3.7) so that pressing the reset button restarts the machine code program. The button can be disabled by typing POKE DEEK (#FFFA),64.

The 'BRK' instruction causes an interrupt, but sets the BRK flag in the 6502 processor. It is used by some machine code monitors as a terminating command – just as RTS is used to return to BASIC after a CALL instruction.

Use RTS instead of BRK if your machine code monitor expects it.

3.9 Software timers

This subject was mentioned when interrupts were discussed. There are three 16-bit counters stored in RAM, maintained by the interrupt routine. The first two timers are in permanent use on the Oric: the first counts three interrupt cycles (normally 30 ms) before each keyboard read while the second counts 25 interrupts (250 ms) before flashing the cursor on or off. The third software timer is only used occasionally by the system – for WAIT, TEXT, and (in version 1.0 only) when using the HIRES command. This means that it is available for use within your own program. With very little trouble, you can time events to one-hundredth of a second.

Remember that the software timers will only be decremented when interrupts have been enabled.

Each of the three timers occupies 2 bytes, in the normal tradition of the low byte first, starting at #272. Therefore, the all-important third timer is located at #276,7. The WAIT command can

be simulated by a simple use of DOKE and DEEK into location #276, but with the advantage that the program can do further work while the third timer is counting.

Although it is a simple matter to set up this timer, there are a number of subroutines in ROM which handle each of the timers.

The A, X, and Y registers need to be set up as follows:

A – set to the timer number minus one. For instance, the third timer requires a value of two.

Y – set the Y register to the low part of the timer value.
X – set the X register to the high part of the timer value.

Here is a table of calls which relate to the software timers:

Name	Version 1.0	Version 1.1
Start 6522 Clocks	#ECC7	#EDE0
Stop 6522 clocks	#ED01	#EE1A
Update timers etc	#ED1B	#EE34
Clear all timers	#ED70	#EE8C
Read a timer into X Y	#ED81	#EE9D
Write XY into a timer	#ED8F	#EEAB
Wait for time X Y	#EDAD	#EEC9

8.10 Machine code advice

As mentioned previously, a book on machine code is essential, not only to teach the subject but as a constant guide to the 6502. This section covers some of the more error-prone areas of programming, in the hope that you may learn from my own mistakes!

BRANCHES

The following observations may be useful:

1. Any branch will depend on one bit within the processor status register. Branch instructions work in pairs, e.g., BEQ, BNE; BCS, BCC.

2. The operand in the branch instruction gives the number of bytes, forward or backward, to jump. If this number is between 0 and #7F the branch is forward in memory; otherwise the jump is to a previous location. When a backward branch is required the operand is #100 minus the number of locations that you are jumping. For example: 1200 BNE 11C2 results in an operand of (#100 – (#1202 – #11C2)) = #CO.

Any good machine code monitor will work out branch offsets for you. An assembler will allow you to enter either an absolute address or a meaningful label.

COMPARE

A newcomer to 6502 programming can become confused with the CMP instruction when testing less-than or greater-than conditions.

The compare instruction works in a similar way to subtract as regards the use of the carry flag. When a subtraction is done, the carry flag is used to indicate a borrow when the value being subtracted is greater than the accumulator. The advantage of the compare instruction is that the A, X, and Y registers are not affected.

When writing a compare instruction do a mental subtraction of the value given in the instruction from the register value (A, X, or Y). If the result is zero, the zero flag is set. If the result is positive, including zero, the carry flag is set; otherwise it is cleared.

THE BIT INSTRUCTION

BIT is probably the least used of all the instructions – CMP is often used instead.

Like the compare instruction, BIT only alters flags in the processor status register.

If you wanted to examine a number of locations, picking out one bit, then you would load the accumulator with the bits to examine and just use BIT with each address. If you used the AND instruction, you would need to keep reloading the accumulator.

BIT also traps bits 6 and 7 of the location you are examining, reflecting them in the overflow and negative flags.

Because BIT does not affect the A, X, and Y registers, you can use BIT in a sneaky way to conserve memory. Consider the program:

```
TRY1    LDA #1
        BNE CARRY-ON
TRY2    LDA #2
        BNE CARRY-ON
TRY3    LDA #3
CARRY-ON:
```

This can be replaced by:

```
TRY1:   LDA #1
        #2C
TRY2:   LDA #2
        #2C
TRY3:   LDA #3
CARRY-ON:
```

The '2C' is the opcode for the 3-byte version of BIT. Here we use the fact that BIT does not alter the accumulator in order to skip past one or two load instructions. You will find this kind of confusing programming when you disassemble the Oric's ROM.

The saving is so small as to be not worth the trouble, but it does demonstrate an interesting programming technique.

THE STACK

When using the stack remember:

1. In a subroutine you must leave the stack as you find it. This means that if you execute 5 PHA instructions, you must balance them with 5 PLA instructions. This is important because the RTS instruction will be expecting a return address on the stack.

2. To follow up the last point, here is a common mistake:

```
1000 PHP
1001 JSR 1234
1234 PLP; attempt to pass processor stack.
```

3. When saving all the registers on the stack, use a sequence such as:

```
PHP PHA TXA PHA TYA PHA
```

When you want to restore the registers, remember to reverse the order:

```
PI.A TAY PLA TAX PLA PLP
```

If you are saving an area of memory on the stack you will need to reverse the loop when loading back from the stack. For example, if this is your save routine:

```
LDX #F
A:
LDA 2EO,X
PHA
DEX
BPL A
```

then the reverse procedure is:

```
LDX #o
A:
PLA
STA 2EO,X
INX
CPX #10
BNE A
```

The stack provides the only way of examining the complete processor status register:

PHP PLA

Similarly, to set up the processor status register in one go:

```
LDA #47
PHA
PLP
```

DECIMAL INSTRUCTIONS

When a program goes unaccountably wrong always consider the state of the decimal flag. The normal state for the decimal flag is off. Many ROM subroutines will expect the decimal flag to be cleared, so remember the CLD instruction.

The decimal flag is only recognized when using either the ADD or SBC instructions, whereas INC and DEC will always work in binary.

SHIFT AND ROTATE

When using any of the shift or rotate instructions, remember:

1. There is always one bit coming away from the byte. This is always saved in the carry flag.

2. There is always one bit coming into the byte. This is either zero for shift instructions or the old carry flag for rotate instructions.

3. The rotate instructions work on 9 bits at a time. Therefore, if you rotate 0000 0001 to the right, the 1 will not appear on the left until a further rotate instruction.

CLEAR CARRY AND SET CARRY

Two simple rules apply here:

1. Clear the carry flag before doing an addition. If adding numbers longer than 8 bits, leave carry alone after the first clear carry instruction; for example:

```
CLC
LDA 0
ADC 2
 STA 0
 LDA 1
 ADC 3
 STA 1
```

INCREMENT AND DECREMENT

Important points:

1. INC and DEC take no notice of the decimal flag – they always

2. work in binary.
. INC and DEC do not either use or alter the carry flag. If you want to increment a 16-bit value, use a branch instruction, as in:

```
INC 42
BNE B
INC 43
B:
```

When decrementing numbers, you have to use a compare instruction:

```
DEC 42
LDA 42
CMP #FF
BNE C
DEC 43
 C: NOP
```

3 When using INC or DEC with several bytes, remember that you
. can only safely do one set of INC or DEC instructions at a time. The following example employs such faulty logic:

```
   INC   42
   INC   42
   BNE   A
   INC 43
A NOP
```

RETURN FROM INTERRUPT

Remember to use RTI to finish an interrupt routine. The only difference between RTI and RTS is that with RTI the 6502 saves the processor flag on the stack. This means that an interrupt routine need not save the processor status register.

SUBROUTINES

When the jump to subroutine instruction is executed, the return address is saved on the stack. This address is saved high byte followed by low byte (this follows the 6502 convention of a low address being stored in the lower location). This return address on the stack is always one less than the real return address – the 6502 adds one to the program pointer before executing each instruction.

SEI AND CLI

On the Oric an interrupt can occur at any time. If you want to disable interrupts (which will stop the keyboard from being scanned and the cursor flashing) you can use the SEI instruction. CLI (clear interrupt disable) enables interrupts again.

Note that SEI does not stop the 6522 clocks from running, but it does prevent interrupts from being generated when the clocks reach zero.

SEI should be used when your program is using the stack area in a non-standard way.

3.11 Using the! extension command

The ! command allows you to create your own BASIC command. When BASIC encounters the ! token it jumps to the address stored at #2F5,6, assuming it to be a normal subroutine.

PASSING DATA

PEEK and POKE provide one way to send data between your extension subroutine and BASIC, but a better way is to put the data after the ! command, as you would do for any other BASIC command.

The pointer #E9, #EA will be identifying the byte following the! command as you enter your subroutine. You can (and must) use this pointer to extract all the data pertaining to the command. When you exit from your subroutine #E9, #EA must be pointing to the byte following the last byte in your command.

In order to look at each character, you can call subroutines at #E2 (which increments #E9, #EA) or #E8 (which does not increment #E9,#EA). After the call the next character is passed in the accumulator. This can be used to pass over delimiters, such as commas.

USING THE FORMULA EVALUATION ROUTINE

If you want the extension command to work with expressions (such as X+ Y) as well as fixed-format data, you may need to call the ROM subroutine which evaluates an expression.

This subroutine (at #CE8B for version 1.0 ROMs or #CF17 for version 1.1 ROMs) only needs the WE9, #EA pointer to be set up. At the end of the subroutine the #E9, #EA pointer will be correctly set to the character following your expression. Note that the expression evaluated can contain the normal BASIC functions, e.g., !X*SQR(Y), but be warned that the subroutine assumes that all words have been compacted into tokens – including such things as the +,–,*, and / operators. As in BASIC, expressions must be terminated with a comma, colon, or 000 (i.e., the end of a BASIC line).

There are two possible types of answer returned:

1. A string of characters. The information about this string is stored in an area of memory pointed to by the address #DR, WD4. In this temporary area there are three bytes: length (one byte) and address of string (two bytes). When the formula results in a string, location #28 is set to OFF. Once you have finished with the string, you must release the temporary area it used by calling either #D712 (version 1.0) or gD7CD (version 1.1).
2. A floating-point number. This number is stored in the floating-point accumulator (see Chapter 6). Location 028 is set to zero to indicate a numeric result.
   If you want to convert the number into a signed 2-byte integer, you can simply call #D871 (version 1.0) or #D92C (version 1.1). This will return Y as the low byte and A as the high byte. For an example of usirv ! see Chapter 4.

3.12 Using the & extension function routine

Whereas ! can only be passed data, the % function not only expects data to be passed but also returns a value. The & facility assumes that g2FC, #2FD points to the machine code routine.

PASSING DATA

There are two types of data that can be passed – a string of characters or a number. In both cases, & must have an argument following, surrounded by parenthesis. For example, &(A$), &(4.3+S).

The formula evaluation takes place automatically on the argument, and the results are exactly the same as described in Sec. 3.11.

When a number is passed, you can either take it or leave it, but a string requires extra action.

If your subroutine has been passed a string, you must call subroutine #D7F1 (version 1.0) or #D8AC (version 1.1) in order to free up the temporary string space. This will also extract the necessary information, storing the length in the accumulator and the address of the string in #91, #92.

RETURNING DATA

Returning data will usually be the final thing that the subroutine does. Location #28 should be set to zero if you are returning a number, or #FF if the result is a string.

To return a number you simply leave that number in the floating-point accumulator at #DO to #D5 – see Chapter 6.

Returning a string is a little more complicated, since you must first allocate an area for it. This is done by putting the length (in bytes) into the accumulator and calling #D4FO (version 1.0)

or #D5AB (version 1.1). This will leave the address of the new string at #D1, #D2. Once you have put the string at this address, you must finish the subroutine with:

PLA

PLA

JMP #D539 (for version 1.0)

JMP #D5F4 (for version 1.1)

When returning a floating-point number, you exit with the usual RTS instruction.

EXAMPLE: THE INSTR FUNCTION

On some computers you will find the 'INSTR' function. This searches for a string of characters within another string, returning its position, if found.

For example, INSTR("ABCD","BC",1) is 2 (the last parameter 1 indicates the start position of the search).

The subroutine of Program 3.2 simulates the INSTR function. The function is called by a statement such as: A=&("T$,S$,N").

String S$ is searched for within string T$, starting at position N. The quotes are used since R can only take one parameter; this means that you can only use simple variables (such as A$) in the actual statement.

The listing will work unchanged for version 1.0 owners, but users of version 1.1 ROMs should make the following adjustments:

```
9800  JSR
D8AC
981D  JSR CF17
982D  JSR CF17
983D  JSR CF17
9840  JSR
D92C
987B  JSR D499
```

To use INSTR, you must first type DOKE #2FC, #9800.

```
9800:  20 F1 D7    JSR  $D7F1
9803:  A0 09       LDY  #$09
9805:  B9 33 00    LDA  $0033,Y
9808:  48          PHA
9809:  88          DEY
980A:  10 F9       BPL  $9805
980C:  A5 E9       LDA  $E9
980E:  48          PHA
980F:  A5 EA       LDA  $EA
9811:  48          PHA
9812:  A0 01       LDY  #$01
9814:  B1 D3       LDA  ($D3),Y
9816:  85 E9       STA  $E9
9818:  C8          INY
9819:  B1 D3       LDA  ($D3),Y
981B:  85 EA       STA  $EA
981D:  20 8B CE    JSR  $CE8B
9820:  A0 02       LDY  #$02
9822:  B1 D3       LDA  ($D3),Y
9824:  99 35 00    STA  $0035,Y
9827:  88          DEY
9828:  10 F8       BPL  $9822
982A:  20 E2 00    JSR  $00E2
982D:  20 8B CE    JSR  $CE8B
9830:  A0 02       LDY  #$02
9832:  B1 D3       LDA  ($D3),Y
9834:  99 38 00    STA  $0038,Y
9837:  88          DEY
9838:  10 F8       BPL  $9832
983A:  20 E2 00    JSR  $00E2
983D:  20 8B CE    JSR  $CE8B
9840:  20 71 D8    JSR  $D871
9843:  38          SEC
```

```
9844:  A5 35      LDA  $35
9846:  E5 38      SBC  $38
9848:  85 35      STA  $35
984A:  E6 35      INC  $35
984C:  C6 33      DEC  $33
984E:  A5 33      LDA  $33
9850:  85 3B      STA  $3B
9852:  C5 35      CMP  $35
9854:  B0 21      BCS  $9877
9856:  A9 00      LDA  #$00
9858:  85 3C      STA  $3C
985A:  A4 3B      LDY  $3B
985C:  B1 36      LDA  ($36),Y
985E:  A4 3C      LDY  $3C
9860:  D1 39      CMP  ($39),Y
9862:  D0 0F      BNE  $9873
9864:  E6 3B      INC  $3B
9866:  E6 3C      INC  $3C
9868:  A5 3C      LDA  $3C
986A:  C5 38      CMP  $38
986C:  D0 EC      BNE  $985A
986E:  A4 33      LDY  $33
9870:  C8         INY
9871:  D0 06      BNE  $9879
9873:  E6 33      INC  $33
9875:  D0 D7      BNE  $984E
9877:  A0 00      LDY  #$00
9879:  A9 00      LDA  #$00
987B:  20 ED D3   JSR  $D3ED
987E:  68         PLA
987F:  85 EA      STA  $EA
9881:  68         PLA
9882:  85 E9      STA  $E9
9884:  A0 09      LDY  #$09
9886:  68         PLA
9887:  99 33 00   STA  $0033,Y
988A:  88         DEY
988B:  10 F9      BPL  $9886
988D:  60         RTS
988E:  EA         NOP
988F:  EA         NOP
9890:  EA         NOP
```

Program 3.2 INSTR

-

3.13 A real-time clock

Program 3.3 is a short program to give your programs a clock that can return the current time of day.

```
0410:  48          PHA
0411:  18          CLC
0412:  F8          SED
0413:  AD C4 02    LDA  $02C4
0416:  69 01       ADC  #$01
0418:  8D C4 02    STA  $02C4
041B:  D8          CLD
041C:  AD C5 02    LDA  $02C5
041F:  69 00       ADC  #$00
0421:  8D C5 02    STA  $02C5
0424:  C9 3C       CMP  #$3C
0426:  D0 32       BNE  $045A
0428:  A9 00       LDA  #$00
042A:  8D C5 02    STA  $02C5
042D:  18          CLC
042E:  AD C6 02    LDA  $02C6
0431:  69 01       ADC  #$01
0433:  8D C6 02    STA  $02C6
0436:  C9 3C       CMP  #$3C
0438:  D0 20       BNE  $045A
043A:  A9 00       LDA  #$00
043C:  8D C6 02    STA  $02C6
043F:  18          CLC
0440:  AD C7 02    LDA  $02C7
0443:  69 01       ADC  #$01
0445:  8D C7 02    STA  $02C7
0448:  C9 18       CMP  #$18
044A:  D0 0E       BNE  $045A
044C:  A9 00       LDA  #$00
044E:  8D C7 02    STA  $02C7
0451:  18          CLC
0452:  AD C8 02    LDA  $02C8
0455:  69 01       ADC  #$01
0457:  8D C8 02    STA  $02C8
045A:  68          PLA
045B:  40          RTI
045C:  A2 04       LDX  #$04
045E:  A9 00       LDA  #$00
0460:  9D C4 02    STA  $02C4,X
0463:  CA          DEX
0464:  10 FA       BPL  $0460
0466:  A2 02       LDX  #$02
0468:  BD 72 04    LDA  $0472,X
046B:  9D 30 02    STA  $0230,X
046E:  CA          DEX
046F:  10 F7       BPL  $0468
0471:  60          RTS
0472:  4C 10 04    JMP  $0410
```

Program 3.3 Clock

The time can be set up (and read back) using PEEK and POKE from the following locations:

#2C5 Seconds

#2C6 Minutes

#2C7 Hours

#2C8 Days

Location #2C4 is used to store one-hundredth second intervals – but this is not in a suitable form for reading.

Owners of version 1.1 ROMs should change the instruction at #46B to STA 24A,X. To start the clock CALL#45C.

ACCURACY

The clock will stay fairly accurate, except when certain commands are used. The most serious problems will arise when doing any tape saving or loading. A minor loss of time can happen during any sound command and when scrolling occurs.

3.14 Relocater program

To complete this chapter, here is a program that allows you to move a machine code program to a different address (Program 3.4). All 3-byte instructions are modified, where necessary, reflecting the new start address.

Since a program may reference locations near to itself, but not actually part of the program, the relocater needs five addresses:

#70, #71 Start address of whole area.

#*72#,*73 End address of whole area.

#78, 079 Start address of actual program.

#7A, #7B End address of actual program.

#7C, #7D New start address of program.

The routine can only cope with instructions – it cannot handle data. If your program has imbedded data, you will have to use the utility in stages.

For example, to move the instruction: 1000 INC 1234 to #2000 (assuming that 1234 is a location that will now become 2234), you would need to set up the following addresses:

#70, #71 – 00 10
#72, #73 – 34 12
#78, #79 – 00 10
#7A, #7B – 02 10
#7C, #7D – 00 20

The routine is entered from address #440 and does not have any calls to the ROM.

```
0440:  D8          CLD
0441:  38          SEC
0442:  A5 7C       LDA  $7C
0444:  E5 78       SBC  $78
0446:  85 76       STA  $76
0448:  A5 7D       LDA  $7D
044A:  E5 79       SBC  $79
044C:  85 77       STA  $77
044E:  A0 00       LDY  #$00
0450:  B1 78       LDA  ($78),Y
0452:  91 7C       STA  ($7C),Y
0454:  29 0F       AND  #$0F
0456:  C9 0D       CMP  #$0D
0458:  F0 29       BEQ  $0483
045A:  C9 0E       CMP  #$0E
045C:  F0 25       BEQ  $0483
045E:  B1 78       LDA  ($78),Y
0460:  A2 10       LDX  #$10
0462:  DD 20 04    CMP  $0420,X
0465:  F0 1C       BEQ  $0483
0467:  CA          DEX
0468:  10 F8       BPL  $0462
046A:  A2 0D       LDX  #$0D
046C:  DD 31 04    CMP  $0431,X
046F:  F0 56       BEQ  $04C7
0471:  CA          DEX
0472:  10 F8       BPL  $046C
0474:  29 0F       AND  #$0F
0476:  C9 08       CMP  #$08
0478:  F0 4D       BEQ  $04C7
047A:  C8          INY
047B:  B1 78       LDA  ($78),Y
047D:  91 7C       STA  ($7C),Y
047F:  D0 46       BNE  $04C7
0481:  F0 44       BEQ  $04C7
0483:  A0 02       LDY  #$02
0485:  B1 78       LDA  ($78),Y
0487:  C5 71       CMP  $71
0489:  F0 04       BEQ  $048F
048B:  90 2F       BCC  $04BC
048D:  B0 07       BCS  $0496
048F:  88          DEY
0490:  B1 78       LDA  ($78),Y
0492:  C5 70       CMP  $70
0494:  90 26       BCC  $04BC
0496:  A0 02       LDY  #$02
0498:  A5 73       LDA  $73
```

(listing continues)

```
049A:  D1 78     CMP  ($78),Y
049C:  F0 04     BEQ  $04A2
049E:  90 1C     BCC  $04BC
04A0:  B0 07     BCS  $04A9
04A2:  88        DEY
04A3:  A5 72     LDA  $72
04A5:  D1 78     CMP  ($78),Y
04A7:  90 13     BCC  $04BC
04A9:  A0 01     LDY  #$01
04AB:  B1 78     LDA  ($78),Y
04AD:  18        CLC
04AE:  65 76     ADC  $76
04B0:  91 7C     STA  ($7C),Y
04B2:  C8        INY
04B3:  B1 78     LDA  ($78),Y
04B5:  65 77     ADC  $77
04B7:  91 7C     STA  ($7C),Y
04B9:  38        SEC
04BA:  B0 0B     BCS  $04C7
04BC:  A0 01     LDY  #$01
04BE:  B1 78     LDA  ($78),Y
04C0:  91 7C     STA  ($7C),Y
04C2:  C8        INY
04C3:  B1 78     LDA  ($78),Y
04C5:  91 7C     STA  ($7C),Y
04C7:  C8        INY
04C8:  98        TYA
04C9:  18        CLC
04CA:  65 78     ADC  $78
04CC:  85 78     STA  $78
04CE:  90 02     BCC  $04D2
04D0:  E6 79     INC  $79
04D2:  18        CLC
04D3:  98        TYA
04D4:  65 7C     ADC  $7C
04D6:  85 7C     STA  $7C
04D8:  90 02     BCC  $04DC
04DA:  E6 7D     INC  $7D
04DC:  A5 79     LDA  $79
04DE:  C5 7B     CMP  $7B
04E0:  F0 04     BEQ  $04E6
04E2:  B0 08     BCS  $04EC
04E4:  90 07     BCC  $04ED
04E6:  A5 78     LDA  $78
04E8:  C5 7A     CMP  $7A
04EA:  90 01     BCC  $04ED
04EC:  60        RTS
04ED:  4C 4E 04  JMP  $044E
04F0:  EA        NOP
```

```
04F1:  EA          NOP
04F2:  EA          NOP
04F3:  EA          NOP
04F4:  EA          NOP
04F5:  EA          NOP
04F6:  EA          NOP
04F7:  EA          NOP
04F8:  EA          NOP
0420: 79 39 D9 EC CC 59 4C 6C
0428: 20 B9 AC BC 19 F9 99 8C
0430: 00 0A 00 B8 CA 4A EA 2A
0438: 6A 40 60 AA BA 8A 9A 00
```

Program 3.4 Relocater program (#420 – #43F and #440 – #4F0)

4.THE KEYBOARD AND CASSETTE SYSTEM

4.1 Keyboard

HARDWARE

The hardware which enables the keyboard to work has already been described in Chapter 1. To summarize, the keyboard is scanned every 30 ms using port A of the 8912 and port B of the 6522. This is done by writing to each column and row in the keyboard matrix – identifying just one key at a time. At any moment there may be any number of keys pressed, but although the automatic scanning routine only looks for one key (or two, if you count the shift and control keys) it is still possible to look for multiple keypresses.

USEFUL LOCATIONS

The keyboard routines in ROM leave behind a number of useful locations.

The most important address is #2DF which contains the ASCII value of the last keypress. This value is OR'ed with #80 by the keyboard routine to indicate that the keypress has not been processed.

'I'his location is subject to delays when the same key is pressed twice because of the autorepeat feature, so often you will want a faster access to the keyboard. Location 4208 is set to a unique value when a key is pressed, but there is no direct correspondence between this value and the ASCII sequence – you will need to use a good deal of trial and error. The value here is a combination of two 3-bit column and row numbers.

For example, when 'A' is pressed (in both upper-case and lowercase) you will find that location 4208 contains #AE.

The two shift keys and the control key are not recorded in location #208, but instead at 4209. This makes it possible to differentiate between the left and right shift keys – useful for games, etc.

USEFUL ROM ADDRESSES

When fast key action is not required, a machine code program can quickly get the ASCII code of the last keypress with one of two calls:

1. To read a key without waiting, returning the ASCII code in the accumulator, call subroutine #E905 (version 1.0) or #EB78 (version 1.1). This is identical to using KEY$ in BASIC.
2. To wait for a key to be pressed (i.e., like GET in BASIC), call either #C5F8 (version 1.0) or #C5E8 (version 1.1).

INDEPENDENT KEYPRESS ROUTINE

The normal method of detecting keypresses is slow and inefficient, since the whole keyboard must be scanned 33 times a second and interrupts must be running for this to happen.

More importantly, the limitation of being able to read only one key at a time can be a real hurdle when writing a game program.

Program 4.1 shows a short subroutine that examines only one key and sets the zero flag to reflect the state of the key. In other words, the zero flag is set when the key is not pressed and clear when the key is pressed.

```
4000:  08        PHP
4001:  78        SEI
4002:  48        PHA
4003:  A9 0E     LDA  #$0E
4005:  20 35 F5  JSR  $F535
4008:  68        PLA
4009:  09 B8     ORA  #$B8
400B:  8D 00 03  STA  $0300
400E:  A2 04     LDX  #$04
4010:  CA        DEX
4011:  D0 FD     BNE  $4010
4013:  AD 00 03  LDA  $0300
4016:  29 08     AND  #$08
4018:  AA        TAX
4019:  28        PLP
401A:  8A        TXA
401B:  60        RTS
```

Program 4.1 Read keyboard subroutine

This subroutine can be used for any number of keys simultaneously. It requires two registers to be set up: the accumulator should contain the row number (0 – 7) and the X register should be set to the column number. The column number is one bit cleared in a byte containing OFF, i.e., #7F, #BF, #DF, #EF, #F7, #FB, g FD, or #FE. As with location #208, the required values do not fall in a recognizable pattern – Table 4.1 gives the A and X values for each possible key. Version 1.1 users must change the instruction at #4005 to JSR #F590.

Table 4.1 Keypress values

Key required	Accumulat	X register
1 2 3	or	DF BF
4 5 6	0 2 0	7F F7
7 8 9	2 0 2	FB FD
0 - =	0 7 3	FE FE
\ ESC Q	7 3 7	FD FB
	3 1 1	F7 7F
W E R		BF DF
T Y U	6 6 1	BF
I 0 P	1 6 5	
	5 5 5	7F F7
[ ] DEL		FB FD
CTRL A S	5 5 5	FE FE
D F G	2 6 6	FD FB
H J K	1 1 6	F7
	6 1 3	
L ; “		7F BF
RETURN	7 3 3	DF EF
SHIFT (LEFT)	7	DF BF
	4	7F F7
Z X C		FB FD
VBN	2 0 2	FE FE
M comma period	0 2 0	
/ SHIFT (RIGHT)	2 4 4	FD FB
LEFT ARROW	7 7	7F DF
DOWN ARROW	4	EF
SPACE	4	
UP ARROW	4	DF BF

RIGHT ARROW	4 4	7F F7 FB FD FE FD FB F7 EF DF BF FE F7 7F

4.2 Cassette input/output

This section will describe the various ways in which the cassette system can be used.

There are three programs described in this part of the chapter, each giving an extra facility that can be used from BASIC.

The routines in ROM that allow cassette I/0 are neatly structured so that saving and loading can be done either:

1. As a complete section of memory.
2. One byte at a time.
3. One bit at a time.

The third option is not used in this chapter; most applications are only concerned with whole bytes.

However, Sec. 9. I – speech synthesis – shows how bits can be read from the cassette hardware.

Saving and loading bytes is often more useful than saving a large area since you can have a free hand as to the exact format of your data on tape.

This is one subject where the two versions of ROM differ greatly: both the subroutine addresses and the usage of page 0 and page 2 are altered.

Generally, version 1.1. uses page 2 to store filenames and flags, whereas version 1.0 uses the BASIC input buffer area – g3F to #67.

4.3 Saving an area of memory

The sequence of events when saving a block of memory (remember that a BASIC program is just a block of memory) is:

1. Disable interrupts and change the 6522 into cassette mode.
2. Print the message ‘SAVING’ and the filename on the top line of the screen.
3. Save a header record, composed of:
   (a) 259 occurrences of #16 (this is the actual ‘header’).
   (b) The value #24 to indicate the start of the record.
   (c) For version 1.0 – #5E to #66 – or for version 1.1 – #2A0 to #2B0. This information is saved backwards and includes the start and end addresses and other flags.
   (d) A filename, ending with #0 – this is either #35 onwards, for version 1.0, or #27F onwards, for version 1.1.
4. Save the block of memory, byte by byte.
5. Re-enable interrupts and reset the 6522 back to its normal mode.

LOCATIONS USED WHEN SAVING

From the previous paragraph, you will notice that all the important information is saved as a 9-byte block of data. Here is how version 1.0 uses its flags and buffers:

#5F, #60 Start address

#61, #62 End address.

#63 Autoload flag – set to zero if no autoload required.
#64 Machine code of BASIC – set to zero for BASIC.
#67 Speed. Zero means fast, one means slow.
#35 – #44 Filename, terminated by #00.

In version 1.1, the same flags are stored as follows:

*#2A9, #*2AA Start address

#2AB, #2AC End address.

#2AD Autoload flag – zero means no autoload.

#2AE Machine code flag – set to zero if BASIC.
#24D Speed. Zero means fast, one means slow,
#27F – #28E Filename, terminated with zero.

Although the addresses of these flags differ between ROM versions they are in an identical format. This allows programs saved to tape by one type of machine to work on a different type.

Note that the machine code and autoload flags will only be recognized by the CLOAD"" command – if you use the subroutines as described in this chapter, they will be ignored.

Another point is that the speed flag is used by the routine that saves individual bytes. If unchanged, the speed will remain the same as the previous cassette operation.

SUBROUTINES REQUIRED

In order to save a block of memory, having set up the speed, start address, etc., you must call a series of subroutines:

1. For version 1.0:

   JSR E6CA (interrupts off)
   JSR E57B (save)
   JSR E804 (interrupts on)

2. For version 1.1:

   JSR E76A (interrupts off)
   JSR E585 (print 'saving')
   JSR E607 (save header record)
   JSR E62E (save area of memory)
   JSR E93D (interrupts on)

4.4 Loading an area of memory

Loading back data is basically the reverse of saving, except that:

1. On version 1.1, the loading program may be just verifying the tape against memory.
2. The filename has to be matched against each filename on tape.

The sequence of events when loading (or verifying) is:

1. Disable interrupts and alter the 6522 ready for cassette I/0.
2. Print the message 'searching'

3. Lock onto the file header, until a sequence of three #16s is detected.
4. Wait until #24 is detected and then read in the header record.
5. Store the filename coming in.
6. If the filename on the tape is different from the required name then go back to sequence 3. (Version 1.1 also prints 'FOUND XX'.)
7. Change message to 'loading' (or 'verifying').
8. Load or verify the file on tape.
9. Re-enable interrupts, etc.

LOCATIONS USED

See Sec. 4.3 for the important locations – these are the same when loading. When loading, it is not necessary to provide the information that will be loaded in from the header record. The essential details are:

Version 1.0*: #67* – the tape speed (zero when fast, one when slow).

#35 – #44 – the filename, terminated by #00.

Version 1.1*:* #24D – tape speed (zero when fast, one when slow).

#27F – #28E – the filename, terminated by #00.
#25B – the verify flag – set to zero for load, one for verify.
#25A – the join flag – set to zero for a normal load.

On version 1.1, the count of verify errors is stored at #25C,D. On both versions an error flag is kept at #2B1 – this indicates errors in loading any byte. Location Ø2B1 will contain zero when there are no errors.

Note that when you use a series of subroutines as described in this chapter, you will not get messages such as 'errors found' or the count of verify errors.

One improvement made in version 1.1 is that location #21F is checked before any message is displayed on the top line – this prevents the HIRES screen from being overwritten.

The filename on tape is stored at #49 to #56 (version 1.0) or #293 to #2A2 (version 1.1).

SUBROUTINES REQUIRED

In order to load a tape file, call the following subroutines: 1. Version 1.0:

JSR E6CA (disable interrupts, etc.)
JSR E4A8 (search and load)
JSR E804 (enable interrupts, etc.)

2. Version 1.1:

JSR E76A (disable interrupts, etc.)
JSR E57D (print 'searching' message)
JSR E4AC (find file)
JSR E59B (print 'loading')
JSR E4E0 (load file, or verify)
JSR E93D (enable interrupts)

Note that these subroutines are not exactly the same as a CLOAD command. As mentioned before, no error messages are printed and, in addition to this, the program will not autorun.

On version 1.1, the routine that prints a message on the top line is patched via a jump at #241. This may be (carefully!) altered in order to add your own processing at either the 'search' or the 'load' phase.

Yet a further important difference between the two ROM versions exists when a BASIC program is loaded. On version 1.1 a subroutine is called which relinks all the lines in the program. This prevents problems arising when the links have been corrupted during loading, and allows the 'join' facility to create an executable program. This is not done on

version 1.0, so be warned that if you deliberately upset the links (one reason would be to stop 'LIST' from working) you will find that version 1.1 ROMs correct your vandalism! If you are mixing machine code with BASIC, be sure to end your BASIC program with a link between #00 and #FF (see Chapter 2), or you may find some of your machine code gets corrupted.

SUMMARY OF ROM SUBROUTINES

In order for you to save and load data, byte by byte, here is a list of all important addresses. Note that in order to do any cassette I/0, you must first call the subroutine which disables interrupts. For version 1.0 this is WE6CA and for version 1.1 it is #E76A. When you have finished your cassette I/0, you should call #E804 (version 1.0) or #E93D (version 1.1).

1. Version 1.0:

Clear top line #E563
Print message on top line (Addressed by A=low,Y=high, X=start position) #F436
Find header #E696
Read one byte into accumulator #E630
Output header #E6BA
Output byte from the accumulator #E5C6

2. Version 1.1:

Clear top line #E5F5
Print message on top line (addressed by A=low, Y=high, at position X) #F865
Find header #E735
Read one byte into accumulator #E6C9
Output header #E75A
Output byte from accumulator #E65E

Note that the 'header' referred to above is just the sequence of 259 lots of ' #16' – not the header record.

The following examples will help with an understanding of the tape subroutines.

4.5 A verify facility for version 1.0

```
B400:  20 CA E6    JSR  $E6CA
B403:  20 63 E5    JSR  $E563
B406:  A9 03       LDA  #$03
B408:  A0 E5       LDY  #$E5
B40A:  20 76 E5    JSR  $E576
B40D:  20 96 E6    JSR  $E696
B410:  20 30 E6    JSR  $E630
B413:  C9 24       CMP  #$24
B415:  D0 F9       BNE  $B410
B417:  A2 09       LDX  #$09
B419:  20 30 E6    JSR  $E630
B41C:  95 5D       STA  $5D,X
B41E:  CA          DEX
B41F:  D0 F8       BNE  $B419
B421:  20 30 E6    JSR  $E630
B424:  F0 05       BEQ  $B42B
B426:  95 49       STA  $49,X
B428:  E8          INX
B429:  D0 F6       BNE  $B421
B42B:  20 63 E5    JSR  $E563
B42E:  A9 70       LDA  #$70
B430:  A0 B4       LDY  #$B4
B432:  20 76 E5    JSR  $E576
B435:  A5 5F       LDA  $5F
B437:  A4 60       LDY  $60
B439:  85 33       STA  $33
B43B:  84 34       STY  $34
B43D:  A0 00       LDY  #$00
B43F:  20 30 E6    JSR  $E630
B442:  D1 33       CMP  ($33),Y
B444:  F0 16       BEQ  $B45C
B446:  A5 33       LDA  $33
B448:  85 00       STA  $00
B44A:  A5 34       LDA  $34
B44C:  85 01       STA  $01
B44E:  20 63 E5    JSR  $E563
B451:  A9 7A       LDA  #$7A
B453:  A0 B4       LDY  #$B4
B455:  20 76 E5    JSR  $E576
B458:  4C 64 B4    JMP  $B464
B45B:  60          RTS
B45C:  20 54 E5    JSR  $E554
B45F:  90 DE       BCC  $B43F
B461:  20 63 E5    JSR  $E563
```

Program 4.2 *(continues)*

```
B464:  20 07 E8     JSR  $E807
B467:  A5 61        LDA  $61
B469:  85 9C        STA  $9C
B46B:  A5 62        LDA  $62
B46D:  85 9D        STA  $9D
B46F:  60           RTS
B470: V 56 E 45 R 52 I 49
B474: F 46 Y 59 I 49 N 4E
B478: G 47 . 00 E 45 R 52
B47C: R 52 O 4F R 52 S 53
B480: . 00
```

Program 4.2 Verify for version 1.0 (#B400 – #B46F and #B470 – #B480)

Owners of version 1.1 will not need this routine (Program 4.2), as VERIFY is one of the features of the updated ROM.

To use this program, POKE #67 with zero or one (fast or slow tape speed) and CALL #B400.

If any differences are found, the 'errors found' message is printed, and the program finishes. This will happen immediately after the error is found, unlike the version 1.1 verify routine which waits until the end. Another difference is that the program here leaves the address of the error at #0,1, so that further investigation is possible.

Note that you are able to load this program on top of an existing BASIC program because although the end-of-BASIC pointer (#9C) is corrupted, the actual verify routine will subsequently correct it.

You will notice an unfamiliar address – #E807. This is the same as #E804 (which enables interrupts, etc.), except that the subroutine is called at a later address in order to prevent the top line being cleared.

4.6 CLOAD with an exit

One irritation when loading a program is that there is no easy way to stop a CLOAD. Control-C does not work, of course (the keyboard is not scanned during cassette I/0), and the only resort is the 'Reset button',

While there is simply not enough time between loading each byte to scan the whole keyboard, it is possible to examine one particular key.

The following program loads the next program it finds, but will exit if 'I' is pressed. We use the keyboard routine discussed in the first part of this chapter, but since the 6522 is in cassette mode we must make a temporary alteration to port B. Before looking for a keypress, one bit in port B is set to be input; after looking at the key, port B is set back to output.

There are two different versions of the program, according to what version of ROM is used.

VERSION 1.0 PROGRAM

To run this program (Program 4.3), set location #67 to the tape speed (zero for fast, one for slow) and call #B49B.

```
B400:  A9 F7       LDA  #$F7
B402:  8D 02 03    STA  $0302
B405:  78          SEI
B406:  A2 FD       LDX  #$FD
B408:  A9 0E       LDA  #$0E
B40A:  20 35 F5    JSR  $F535
B40D:  A9 FD       LDA  #$FD
B40F:  8D 00 03    STA  $0300
B412:  A0 04       LDY  #$04
B414:  88          DEY
B415:  D0 FD       BNE  $B414
B417:  AD 00 03    LDA  $0300
B41A:  29 08       AND  #$08
B41C:  08          PHP
B41D:  A9 FF       LDA  #$FF
B41F:  8D 02 03    STA  $0302
B422:  28          PLP
B423:  60          RTS
B424:  86 36       STX  $36
B426:  84 37       STY  $37
B428:  20 30 E6    JSR  $E630
B42B:  48          PHA
B42C:  20 00 B4    JSR  $B400
B42F:  F0 02       BEQ  $B433
B431:  68          PLA
B432:  68          PLA
B433:  A6 36       LDX  $36
B435:  A4 37       LDY  $37
B437:  68          PLA
B438:  60          RTS
B439:  20 CA E6    JSR  $E6CA
B43C:  A9 00       LDA  #$00
B43E:  85 35       STA  $35
B440:  20 63 E5    JSR  $E563
B443:  A9 03       LDA  #$03
B445:  A0 E5       LDY  #$E5
B447:  20 76 E5    JSR  $E576
B44A:  20 96 E6    JSR  $E696
B44D:  20 24 B4    JSR  $B424
B450:  C9 24       CMP  #$24
B452:  D0 F9       BNE  $B44D
B454:  A2 09       LDX  #$09
```

Program 4.3 *(continues)*

```
B456:  20 24 B4    JSR  $B424
B459:  95 5D       STA  $5D,X
B45B:  CA          DEX
B45C:  D0 F8       BNE  $B456
B45E:  20 24 B4    JSR  $B424
B461:  F0 05       BEQ  $B468
B463:  95 49       STA  $49,X
B465:  E8          INX
B466:  D0 F6       BNE  $B45E
B468:  95 49       STA  $49,X
B46A:  20 F0 E6    JSR  $E6F0
B46D:  8A          TXA
B46E:  D0 D0       BNE  $B440
B470:  20 63 E5    JSR  $E563
B473:  A9 12       LDA  #$12
B475:  A0 E5       LDY  #$E5
B477:  20 76 E5    JSR  $E576
B47A:  20 6E E5    JSR  $E56E
B47D:  EA          NOP
B47E:  EA          NOP
B47F:  E5 EA       SBC  $EA
B481:  EA          NOP
B482:  EA          NOP
B483:  A5 5F       LDA  $5F
B485:  A4 60       LDY  $60
B487:  85 33       STA  $33
B489:  84 34       STY  $34
B48B:  A0 00       LDY  #$00
B48D:  20 24 B4    JSR  $B424
B490:  EA          NOP
B491:  B0 43       BCS  $B4D6
B493:  91 33       STA  ($33),Y
B495:  20 54 E5    JSR  $E554
B498:  90 F3       BCC  $B48D
B49A:  60          RTS
B49B:  20 39 B4    JSR  $B439
B49E:  20 04 E8    JSR  $E804
B4A1:  60          RTS
B4A2:  EA          NOP
B4A3:  EA          NOP
B4A4:  EA          NOP
B4A5:  EA          NOP
B4A6:  EA          NOP
B4A7:  EA          NOP
B4A8:  EA          NOP
B4A9:  EA          NOP
B4AA:  EA          NOP
B4AB:  EA          NOP
```

Program 4.3 Version 1.0 CLOAD with exit

VERSION 1.1 PROGRAM

To run this program (Program 4.4), set location #24D to the tape speed (zero for fast, one for slow) and call #B4A9.

```
B400:  A9 F7       LDA  #$F7
B402:  8D 02 03    STA  $0302
B405:  78          SEI
B406:  A2 FD       LDX  #$FD
B408:  A9 0E       LDA  #$0E
B40A:  20 90 F5    JSR  $F590
B40D:  A9 FD       LDA  #$FD
B40F:  8D 00 03    STA  $0300
B412:  A0 04       LDY  #$04
B414:  88          DEY
B415:  D0 FD       BNE  $B414
B417:  AD 00 03    LDA  $0300
B41A:  29 08       AND  #$08
B41C:  08          PHP
B41D:  A9 FF       LDA  #$FF
B41F:  8D 02 03    STA  $0302
B422:  28          PLP
B423:  60          RTS
B424:  86 36       STX  $36
B426:  84 37       STY  $37
B428:  20 C9 E6    JSR  $E6C9
B42B:  48          PHA
B42C:  20 00 B4    JSR  $B400
B42F:  F0 02       BEQ  $B433
B431:  68          PLA
B432:  68          PLA
B433:  A6 36       LDX  $36
B435:  A4 37       LDY  $37
B437:  68          PLA
B438:  60          RTS
B439:  A9 00       LDA  #$00
B43B:  8D 7F 02    STA  $027F
B43E:  8D 5B 02    STA  $025B
B441:  20 6A E7    JSR  $E76A
B444:  20 7D E5    JSR  $E57D
B447:  20 35 E7    JSR  $E735
B44A:  20 24 B4    JSR  $B424
B44D:  C9 24       CMP  #$24
B44F:  D0 F9       BNE  $B44A
B451:  A2 09       LDX  #$09
B453:  20 24 B4    JSR  $B424
B456:  9D A7 02    STA  $02A7,X
B459:  CA          DEX
B45A:  D0 F7       BNE  $B453
B45C:  20 24 B4    JSR  $B424
```

Program 4.4 *(continues)*

```
B45F:  F0 0D      BEQ  $B46E
B461:  9D 93 02   STA  $0293,X
B464:  E8         INX
B465:  E0 10      CPX  #$10
B467:  D0 F3      BNE  $B45C
B469:  20 24 B4   JSR  $B424
B46C:  D0 FB      BNE  $B469
B46E:  9D 93 02   STA  $0293,X
B471:  20 94 E5   JSR  $E594
B474:  20 90 E7   JSR  $E790
B477:  8A         TXA
B478:  D0 CD      BNE  $B447
B47A:  EA         NOP
B47B:  20 9B E5   JSR  $E59B
B47E:  AD A9 02   LDA  $02A9
B481:  AC AA 02   LDY  $02AA
B484:  85 33      STA  $33
B486:  84 34      STY  $34
B488:  A0 00      LDY  #$00
B48A:  20 24 B4   JSR  $B424
B48D:  AE 5B 02   LDX  $025B
B490:  D0 05      BNE  $B497
B492:  91 33      STA  ($33),Y
B494:  4C A3 B4   JMP  $B4A3
B497:  D1 33      CMP  ($33),Y
B499:  F0 08      BEQ  $B4A3
B49B:  EE 5C 02   INC  $025C
B49E:  D0 03      BNE  $B4A3
B4A0:  EE 5D 02   INC  $025D
B4A3:  20 6C E5   JSR  $E56C
B4A6:  90 E2      BCC  $B48A
B4A8:  60         RTS
B4A9:  20 39 B4   JSR  $B439
B4AC:  20 3D E9   JSR  $E93D
B4AF:  60         RTS
B4B0:  EA         NOP
B4B1:  EA         NOP
B4B2:  EA         NOP
B4B3:  EA         NOP
B4B4:  EA         NOP
B4B5:  EA         NOP
B4B6:  EA         NOP
B4B7:  EA         NOP
B4B8:  EA         NOP
B4B9:  EA         NOP
```

Program 4.4 Version 1.1 CLOAD with exit

4.7 Data saving and loading

The version 1.1 ROM gives the facility to save and recall complete data arrays. A similar routine is available to version 1.0 owners, published in *Oric Owner* magazine.

However, dealing with whole arrays is not always convenient, and one annoying feature is the long headers saved before each record. As has been discussed earlier, 259 bytes are saved to form the header. The purpose of this is to allow time between the cassette recorder stopping and starting, but at slow speed this amounts to 5 seconds!

The length of these headers has been greatly reduced in the following subroutines, and also depends on the tape speed. It is assumed that you are not using the cassette relay – if you are, then you may need to increase the length of the header at #B307.

The following routines can be accessed from BASIC via the '!' and '&' extension commands.

SAVING DATA

Data are saved using the! command. A short header is written first, followed by the actual data. For example, ! "START" would write a record containing the word 'START' onto tape.

When a number is written out, it is saved as a floating-point number. A string is saved with the length first, followed by each character of the string.

LOADING DATA

To load back data you use the & function. This returns the next record from tape. For example, A$ = & (0) (the argument in brackets can be any numeric expression) would read the next string on tape into A$.

When loading data, it is important that the correct type of data is recognized; if you get the type wrong, you will get a TYPE MISMATCH error.

At fast speed, you must not put too much processing between the retrieval of each record – unless a similar delay was incurred when the data were saved.

There is a short header recorded with each record on tape – this gives a small amount of leeway between each record, but it is advisable to disconnect the REMOTE jack socket as the cassette should not be made to start and stop continually.

EXAMPLE

The following BASIC program (Program 4.5) shows how to use the data saving routines. Remember that it is only a guide – you can save and recall both strings and numbers.

Note the two DOKE commands in line 30 – these set up the addresses for the extension commands to work.

```
10 REM DATA SAVING EXAMPLE
20 REM SET UP EXTENSION COMMAND ADDRESSES
30 DOKE#2F5,#B300:DOKE#2FC,#B35A
40 HIMEM#97FF
50 DIMA$(100)
60 FORZ=1TO100:A$(Z)=STR$(Z*Z):NEXTZ
70 REM SAVE A HEADER
80 !"START"
90 FORZ=1TO100
100 'A$(Z)
110 NEXTZ
120 CLS:PRINT"LOADING BACK"
130 REPEAT:UNTIL&(0)="START"
140 FORZ=1TO100:IFA$(Z)=&(0)THENNEXTZ:END
150 PRINT"ERROR"
```

Program 4.5 BASIC example of data saving

PROGRAM LISTINGS

There are two versions, one for each version of ROM. You will probably want to code the routines into DATA statements, so that they can become part of a BASIC program. You may

find Chapter 3 useful in understanding how the subroutines work.

1. The program for version 1.0 ROMs is listed in Program 4.6.

```
B300:  20 8B CE    JSR  $CE8B
B303:  20 CA E6    JSR  $E6CA
B306:  A9 08       LDA  #$08
B308:  A6 67       LDX  $67
B30A:  EA          NOP
```

Program 4.6 *(continues)*

```
B30B:  D0 02     BNE  $B30F
B30D:  0A        ASL
B30E:  0A        ASL
B30F:  AA        TAX
B310:  A9 16     LDA  #$16
B312:  20 C6 E5  JSR  $E5C6
B315:  CA        DEX
B316:  D0 F8     BNE  $B310
B318:  A9 24     LDA  #$24
B31A:  20 C6 E5  JSR  $E5C6
B31D:  20 24 B3  JSR  $B324
B320:  20 04 E8  JSR  $E804
B323:  60        RTS
B324:  A5 28     LDA  $28
B326:  20 C6 E5  JSR  $E5C6
B329:  A5 28     LDA  $28
B32B:  D0 0C     BNE  $B339
B32D:  A0 05     LDY  #$05
B32F:  B9 D0 00  LDA  $00D0,Y
B332:  20 C6 E5  JSR  $E5C6
B335:  88        DEY
B336:  10 F7     BPL  $B32F
B338:  60        RTS
B339:  A0 02     LDY  #$02
B33B:  B1 D3     LDA  ($D3),Y
B33D:  99 D0 00  STA  $00D0,Y
B340:  88        DEY
B341:  10 F8     BPL  $B33B
B343:  A5 D0     LDA  $D0
B345:  20 C6 E5  JSR  $E5C6
B348:  A0 00     LDY  #$00
B34A:  C4 D0     CPY  $D0
B34C:  F0 08     BEQ  $B356
```

```
B34E:  B1 D1      LDA  ($D1),Y
B350:  20 C6 E5   JSR  $E5C6
B353:  C8         INY
B354:  D0 F4      BNE  $B34A
B356:  20 12 D7   JSR  $D712
B359:  60         RTS
B35A:  20 CA E6   JSR  $E6CA
B35D:  20 96 E6   JSR  $E696
B360:  20 30 E6   JSR  $E630
B363:  C9 24      CMP  #$24
B365:  D0 F9      BNE  $B360
B367:  20 30 E6   JSR  $E630
B36A:  85 28      STA  $28
B36C:  D0 0F      BNE  $B37D
B36E:  A0 05      LDY  #$05
B370:  20 30 E6   JSR  $E630
B373:  99 D0 00   STA  $00D0,Y
B376:  88         DEY
B377:  10 F7      BPL  $B370
B379:  20 04 E8   JSR  $E804
B37C:  60         RTS
B37D:  20 30 E6   JSR  $E630
B380:  85 D0      STA  $D0
B382:  20 F0 D4   JSR  $D4F0
B385:  A0 00      LDY  #$00
B387:  C4 D0      CPY  $D0
B389:  F0 08      BEQ  $B393
B38B:  20 30 E6   JSR  $E630
B38E:  91 D1      STA  ($D1),Y
B390:  C8         INY
B391:  D0 F4      BNE  $B387
B393:  20 04 E8   JSR  $E804
B396:  68         PLA

B397:  68         PLA
B398:  4C 39 D5   JMP  $D539
B39B:  EA         NOP
```

Program 4.6 Version 1.0 data saving

2. The program for version 1.1 ROMs is listed in Program 4.7.

```
B300:  20 17 CF    JSR  $CF17
B303:  20 6A E7    JSR  $E76A
B306:  A9 08       LDA  #$08
B308:  AE 4D 02    LDX  $024D
B30B:  D0 02       BNE  $B30F
B30D:  0A          ASL
B30E:  0A          ASL
B30F:  AA          TAX
B310:  A9 16       LDA  #$16
B312:  20 5E E6    JSR  $E65E
B315:  CA          DEX
B316:  D0 F8       BNE  $B310
B318:  A9 24       LDA  #$24
B31A:  20 5E E6    JSR  $E65E
B31D:  20 24 B3    JSR  $B324
B320:  20 3D E9    JSR  $E93D
B323:  60          RTS
B324:  A5 28       LDA  $28
B326:  20 5E E6    JSR  $E65E
B329:  A5 28       LDA  $28
B32B:  D0 0C       BNE  $B339
B32D:  A0 05       LDY  #$05
B32F:  B9 D0 00    LDA  $00D0,Y
B332:  20 5E E6    JSR  $E65E
B335:  88          DEY
B336:  10 F7       BPL  $B32F
B338:  60          RTS
```

Program 4.7 *(continues)*

```
B339:  A0 02     LDY  #$02
B33B:  B1 D3     LDA  ($D3),Y
B33D:  99 D0 00  STA  $00D0,Y
B340:  88        DEY
B341:  10 F8     BPL  $B33B
B343:  A5 D0     LDA  $D0
B345:  20 5E E6  JSR  $E65E
B348:  A0 00     LDY  #$00
B34A:  C4 D0     CPY  $D0
B34C:  F0 08     BEQ  $B356
B34E:  B1 D1     LDA  ($D1),Y
B350:  20 5E E6  JSR  $E65E
B353:  C8        INY
B354:  D0 F4     BNE  $B34A
B356:  20 CD D7  JSR  $D7CD
B359:  60        RTS
B35A:  20 6A E7  JSR  $E76A
B35D:  20 35 E7  JSR  $E735
B360:  20 C9 E6  JSR  $E6C9
B363:  C9 24     CMP  #$24
B365:  D0 F9     BNE  $B360
B367:  20 C9 E6  JSR  $E6C9
B36A:  85 28     STA  $28
B36C:  D0 0F     BNE  $B37D
B36E:  A0 05     LDY  #$05
B370:  20 C9 E6  JSR  $E6C9
B373:  99 D0 00  STA  $00D0,Y
B376:  88        DEY
B377:  10 F7     BPL  $B370
B379:  20 3D E9  JSR  $E93D
B37C:  60        RTS
B37D:  20 C9 E6  JSR  $E6C9
B380:  85 D0     STA  $D0
```

```
B382:  20 AB D5    JSR  $D5AB
B385:  A0 00       LDY  #$00
B387:  C4 D0       CPY  $D0
B389:  F0 08       BEQ  $B393
B38B:  20 C9 E6    JSR  $E6C9
B38E:  91 D1       STA  ($D1),Y
B390:  C8          INY
B391:  D0 F4       BNE  $B387
B393:  20 3D E9    JSR  $E93D
B396:  68          PLA
B397:  68          PLA
B398:  4C F4 D5    JMP  $D5F4
B39B:  EA          NOP
B39C:  EA          NOP
```

Program 4.7 Version 1.1 data saving

EXPLANATION OF SAVING DATA – THE ! COMMAND

Note that the entry addresses are the same for version 1.0 and version 1.1 ROMs, although many of the subroutine calls are different.

1. At #B300, the first step is to call the formula evaluation subroutine. This reads in whatever follows the ! command.
2. At # B303, the 6522 is set up for cassette handling and interrupts are disabled.
3. Depending on the tape speed, either 8 (for slow speed) or 32 (for fast speed) bytes of header are written to tape.
4. To indicate an end to the header, #24 is written to tape.
5. At #B324, the first byte saved is the type indicator at #28. This will have been set to #0 if a number followed the! command or #FF if a string was processed.
6. For a number, the floating-point accumulator at #DO to #D5 is saved on tape, in reverse order.
7. For a string, the length is output, followed by each byte of the string itself.
8. In order to release the temporary string created by the formula evaluation routine a special ROM subroutine is called at #B356
9. At #B320, the subroutine to reset the 6522 is called, restarting clocks, enabling interrupts, etc.

EXPLANATION OF LOADING DATA – THE *'&'* COMMAND

1. At #B35A, the 6522 is switched into cassette mode, with interrupts disabled, etc
2. Following this, a subroutine is called in order to latch on to the small header coming in. Once it has established that it is reading the header, it will return.
3. At #B360, the routine waits until the header finishes; one- #24 has been received, the actual data is loaded.
4. The recall of data follows the same order as the save subroutine, so the first item encountered is the type of data flag. This is stored into Ø28 and is used to indicate which type of data is subsequently read.
5. For a number, the data are stored back into the floating-point accumulator at #DO to gD5.
6. For strings, the length is read, and then a subroutine (called at #B382) allocates the required amount of string space. The address of this area is stored at #D1,#D2 by this ROM sub- routine.

7. Finally, the 6522 is reset, interrupts are enabled again, and the subroutine comes to an end with either: (a) the RTS instruction (for a number) (b) a jump to a special ROM address, after removing the top return address on the stack. This jump (at #B398) is different for each ROM version.

4.8 Conclusions

The intention of this chapter was to show how versatile the Oric's tape system can be. You are not limited to saving and loading in a fixed way, but can devise your own file organization on tape. Also, you will see that it is possible to do extra processing between reading each byte. While your program is loading, why not make the colours on your screen slowly change, or move a message around? There are other tape routines in this book – see the Merge program in Chapter 8 and the speech synthesis idea in Chapter 9.

5 THE ORIC ROM IN DETAIL

5.1 Introduction

The purpose of this chapter is to provide a list of all important ROM addresses. In addition to this, the use of the first three pages of memory is analysed in depth.

Many of the important subroutines are explained elsewhere in this book so only brief descriptions are provided here.

Where a ROM address is given you will find the version 1.0 presented first, followed by the version 1.1 address in parenthesis.

5.2 Use of page 0 memory

Any unspecified locations can be assumed to be used by the ROM but to be of no significance.

#00 – #0B – Unused by BASIC.

#10 – #11 – Address of current HIRES position.

#12 – #13 – Address of start of current line (in TEXT mode).

#14 – #15 – Used by the 8912 register load subroutine.

#18 – #19 – Used to point to the start of error messages.

#1A – #C – Contains a jump to the routine which prints ‘Ready’.

#1F – #20 – Address of last PLOT position.

#21 – #23 – Contains the DEF USR jump.

#28 – The type of data which resulted from formula evaluation; 0 means numeric, #FF means a string.

#29 – A flag that indicates whether the last variable used was an integer.

#2A – A flag which is used in several places.

#33 – #34 – Various uses, but often used to store a line number that is being located.

#35 – #83 – The BASIC input buffer. This is used to store anything that is typed, including immediate commands and INPUT data (which explains why an immediate command cannot use INPUT). This area is also used in version 1.0 during cassette operations – see Chapter 4.

#86 – Address of last temporary string.

#88 – #90 – A table of temporary strings.

#9A – #9B – Start of BASIC pointer.

#9C – #9D – Start of variables pointer.

#9E – #9F – Start of arrays pointer.

#A0 – #A1 – End of arrays pointer.

#A2 – #A3 – Pointer to next free string space.

#A6 – #A7 – Highest available memory location available to BASIC.

#A8 – #A9 – Current line number (read-only).

#AA – #AB - The current line number – saved for error messages.

#AC – #AD- Address of the start of the current instruction – 1.

#AE – #AF – Current DATA line number – used only when printing error messy ges. Altering this location does not change the READ sequence.

#BO – #B1 – The address of the next DATA item – 1. It is this address that one must modify in order to change the data accessed by the next READ command.

#B4 – #B5 – The identity of the last variable used.

#D0 – #D5 – The main floating-point accumulator.

#D8 – #DD – The second accumulator.

#E2 – #E7 – The get-character routine. This part increments the pointer at #E9, #EA and drops into address #E8.

#E8 – #F9 – The second part of the get-character routine. This section loads the next character, according to location #E9,#EA.

#FA – #FE – The current random number is stored here as a floating-point number.

5.3 Use of page 1

#100 –#10F – Used as a temporary area containing the ASCII string of characters whenever a floating-point number is converted. This is used for commands like PRINT and STR$.

#110 – #1FF – Used as a normal 6502 stack area (although it is occasionally pruned by non-standard methods).

USES OF THE STACK

Obviously the ROM makes extensive use of the stack for JSR commands, etc., but some BASIC commands can generate extra entries on the stack:

1. The 'FOR' command generates 18 bytes on the stack. From low address to high address these are:

   1 byte containing the 'FOR' token – #8D.

   2 bytes pointing to the FOR variable.

   5 bytes containing the STEP value (as a floating-point number).

   1 byte indicating the sign of the step (either 1 or #FF).

   5 bytes containing the upper limit in floating-point format. Note that the lower limit will have been stored in the variable being used, so this need not be saved.

   2 bytes containing the line number of the FOR command.

   2 bytes containir.g the address of the statement which follows the FOR instruction.

2. The 'GOSUB' command is somewhat more economical: it only needs 5 bytes. From low to high addresses these are:

   1 byte equal to #9B"'pthe token for 'GOSUB').

   2 bytes giving the line number of the GOSUB command.

   2 bytes giving the address of the character which follows the GOSUB command.

3. 'REPEAT' follows a similar format (again, these are listed from low address to high address):

   1 byte containing the token #8B – 'REPEAT'.

   2 bytes equal to the line number of the REPEAT.

   2 bytes giving the address of the byte which follows the REPEAT command.

4. The formula evaluation subroutine keeps intermediate results on the stack. This is done because arithmetic operations have to be done in a strict order of priority. From most important to least

   important, the operators are:

   (a) Parenthesis – e.g., 4*(3+6).

(b) Exponentiation – e.g., 3^6+4.

(c) Negate – e.g., – B*4.

(d) Multiply and divide – e.g., 3*4/3+6.

(e) Add and subtract.

(f) NOT.

(g) AND.

(h) OR.

From low address to high address, the following information is

stored on the stack:

2 bytes indicating the importance of the operation.

6 bytes containing the floating-point number.

2 bytes containing the address of the appropriate maths routine.

Because these commands use the stack you must ensure that the stack area is not filled. It is difficult to upset GOSUB and REPEAT, since you cannot specify which line RETURN and UNTIL apply to, but often a FOR ... NEXT loop can remain open.

For example, if you used the following lines in your program, you might leave the FOR information on the stack:

```
100 FORI=1 TO 10
110 IF X$(I) =”4” THEN 140
 120 NEXT I
```

If this type of logic were repeated in other places you would soon be dealing with out-of-memory errors. Remember that a FOR... NEXT loop needs 18 bytes of stack!

If this is likely to be a problem, you will need to add a line in order to clear the unwanted FOR... NEXT information:

```
100 FOR I=1 TO 100
110 IF X$(I) =”4” THEN FOR I=1 TO 1:NEXT I:GOTO 140
120 NEXT I
```

Remember also that if you use up most of the stack for FOR and GOSUB commands, you may eventually hit an out-of-memory error when a complex formula is encountered.

5.4 Use of page 2

The difference between the use of #00 to #FF and #200 to #2FF is that the latter is mostly used by the newer Oric commands, while page 0 is almost fully utilised by the standard parts of Microsoft BASIC.

Only those locations which are of any interest (or are unused) are mentioned. There are often differences in the use of some locations by the two ROM versions.

#204 – Used when checking the range of values used in sound and graphics commands.

#208 – Contains details of the last ordinary key pressed (but not the ASCII code).

#209 – Contains details about the last shift or control key pressed.

#20C – Caps lock. This contains either #7F or #FF – no other value will work!

#212 – Contains the HIRES FB flag to indicate whether to draw, erase, or invert.

#213 – The pattern register is stored here. You can POKE values here instead of using

the PATTERN command.

#215 – The graphics cursor mask.
#219 – The HIRES cursor X value.
#21A – The HIRES cursor Y value (do not forget to alter #10 and #11 as well).

#21F – A graphics flag; 0 is TEXT, 1 is graphics.

#220 – Memory size indicator; 0 means 48K, 1 means 16K.

#221 – #227 – Unused.

#228 – #22A – (version 1.0) Jump vector to the fast interrupt routine.

#22B – #22D – (version 1.0) Jump vector to the non-maskable interrupt routine.

#230 – #232 – (version 1.0) Jump vector to the slow interrupt routine.

#228 – #232 – (version 1.1) Unused.

#233 – #237 – Unused.

#238 – #260 – (version 1.0) Unused.

#238 – #23A – (version 1.1) Jump vector to VDU output routine.

#23B – #23D – (version 1.1) Jump vector to the KEY$ routine.

#23E – #240 – (version 1.1) Jump vector to the printer output subroutine.

#241 – #243 – (version 1.1) Jump vector to the routine that prints on the top line of the screen.

#244 – #246 – (version 1.1) Jump vector to the fast interrupt routine.

#247 – #249 – (version 1.1) Jump vector to the NMI routine.

#24A – #24C – (version 1.1) Jump vector to the slow interrupt routine.

#24D – #25D – (version 1.1) Various uses in cassette I/0 – see Chapter 4.

#268 -- Cursor position down the screen, relative to the start address of the screen.

#269 -- Cursor position across the screen.

#26A -- Oric status byte. Each bit relates to one aspect: from high bit to low bit – unused, double-height,

protected-columns, ESC pressed, keyclick, unused, screen-on, cursor-off.

#26B – Text screen paper colour.

#26C – Text screen ink colour.

#26D – #26E – (version 1.0) Start of the TEXT screen – #28.

#26F – (version 1.0) Number of lines to scroll.

#272 – #273 – Timer 1 (used for reading the keyboard). W274 – t275 – Timer 2 (used by the flashing cursor).

#276 – #277 – Timer 3 (used by WAIT, HIRES, and TEXT). #278 – #2BF – (version 1.0) Unused.

#278 – #279 – (version 1.1) Start of TEXT screen + #28.

#27A – #27B – (version 1.1) Start of the text screen.

#27C – #27D – (version 1.1) Number of bytes to scroll – (#27E – 1) * #28

#27E – (version 1.1) Number of lines to scroll.

#27F – #2BF – (version 1.1) Used by the tape routines – (Chap. 4).

#2CO – Graphics enable: 0 means TEXT with GRAB, 1 means TEXT with RELEASE, and 3 means HIRES.

#2C1 – #2C2 – Highest address with graphics enabled + 1.

#2C4 – #2DE – Unused.

#2DF – The AS'CII code for the last key pressed (with top bit set).

#2E0 – #2EF – The parameter area for graphics and sound commands.

#2F1 – Print flag – set to 128 to make all print go to the printer, 0 for it to go to the screen.

#2F2 – A flag set by the EDIT command.

#2F4 – The trace flag.

#2F5 – Address of the ! extension command.

#2F7 – (version 1.0 only). The inverse flag. Try out different values!

#2F9 – #2FA – Unused.

#2FB – #2FD – A jump command to the & extension.

#2FE – #2FF – Unused.

5.5 Summary of ROM addresses

Version 1.0 ROM addresses are given first, followed by the equivalent version 1.1 address in parenthesis. All addresses are hexadecimal. Where there is no equivalent version 1.1 address, ' –' has been specified.

C000 (C000)	Jump to cold start.
C003 (C003)	Jump to warm start.
C006 (C006)	Addresses of subroutines to handle each token (-1)
C0EA (COEA)	BASIC tokens. The last character of each has its top bit set.
C0EA (C3C6)	Search the stack until a 'FOR' entry is found.
C3F8 (C3F4)	A block move.
C43B (C437)	Check stack for free space.
C448 (C444)	Check an address against the top of memory
C475 (C471)	Warm start entry (does not clear program).
C483 (C47C)	Input and process a line.
C56F (C55F)	Recreate links between each line.
C59C (C58C)	Input a line.
C5F8 (C5E8)	Wait for a keypress and return the ASCII code.
C60A (C5FA)	Translate a line into tokens.
C6E4 (C6B9)	Find the address of a given line.
C719 (C6EE)	The NEW command.
C733 (C708)	The RUN command.
C738 (C70D)	The CLEAR command.
C751 (C726)	Reset the stack.
C773 (C748)	The LIST command.
C824 (C7FD)	The LLIST command.
C832 (C809)	The LPRINT command.
C841 (C855)	The FOR command.
C8AD (C8C1)	Process a BASIC statement.
C91F (C952)	The RESTORE command.
CBED (CCB0)	Print 'READY'.
CE8B (CF17)	The formula evaluation subroutine.

D3ED (D499) Integer to floating point.

D4F0 (D5AB) Allocate string space.

D539 (D5F4) Set up a new string.

D595 (D650) Garbage collection subroutine.

D7F1 (D8AC) Calculate the length of a string and clear temporary strings.

D871 (D92C) Floating point to integer.

DA79 (DB04) Add 0.5 to accumulator.

DA80 (DB0B) Subtract accumulator from memory.

DA97 (DB22) Add accumulator to memory.

DC79 (DCAF) Calculate LOG

DCB7 (DCED) Multiply the accumulator by memory.

DD4D (DD51) Move memory to the second accumulator.

DDA3 (DDA7) Multiply the accumulator by 10.

DDBF (DDC3) Divide the accumulator by 10.

DDEO (DDE4) Divide memory by the accumulator.

DDE5 (DDE9) Divide the second accumulator by the main accumulator.

DE73 (DE77) Move memory to the main accumulator.

DEA5 (DEAD) Move the accumulator to memory.

DECD (DED5) Move the second accumulator to the main accumulator.

DEDD (DEE5) Move the main accumulator to the second accumulator.

DF12 (DF21) Calculate SGN.

DF31 (DF49) Calculate ABS.

DFA5 (DFBD) Calculate INT.

DFCF (DFE7) Input a floating-point number from a string of ASCII characters.

E0D1 (E0D5) Output a floating-point number into a string of ASCII characters.

E22A (E22E) Calculate the square root.

E231 (E235) Raise the second accumulator to the power of a number in memory.

E26D (E271) Negate the main accumulator.

E34B (E34F) Calculate RND.

E387(E38B) Calculate COS.

E38E (E392) Calculate SIN.

E3D7 (E3DB) Calculate TAN.

E43B (E43F) Calculate ATN.

E4A8 (----) Load a file from tape (see Chapter 4 for version 1.1).

E554 (E56C) Test for the end of a load from tape.

E563 (E5F5) Clear the top line.

E576 (E5EA) Print message at far left of top line.

E57B (----) Save a file on tape (for version 1.1, see Chapter 4).

E5C6 (E65E) Output one byte to tape.

E630 (E6C9) Read byte from tape.

E696 (E735) Latch onto tape header.

E6BA (E75A) Output a tape header.

E6CA (E76A) Change the 6522 ready for cassette I/0.

E6F0 (E790) Compare filenames.

E70E (----) Print authors' names.

E804 (E93D) Reset the 6522 after the completion of tape I/0.

E905 (EB78) Read a key without waiting.

E9A9 (EC21) Switch to text mode.

E9BB (ECC3) Switch to high-resolution mode.

EC03 (EE22) Entry for interrupt handler.

ECC7 (EDEO) Start clock.

ED01 (EE1A) Stop clock.

ED09 (EE22) Poll timers.

ED1B (EE34) Service timers.

ED70 (EE8C) Clear timers.

ED81 (EE9D) Read timer.

ED8F (EEAB) Set timer.

EDAD (EEC9) Wait for a given time.

F02D (FOC8) CURSET

F064 (FOFD) CURMOV

F079 (F110) DRAW

F0A5 (F12D) CHAR

F141 (F1C8) POINT

F17F (F204) PAPER

F18B (F210) INK

F1E5 (F268) FILL

F2E5 (F37F) CIRCLE

F412 (FA9F) PING

F415 (FAB5) SHOOT

F418 (FACB) EXPLODE

F41B (FAE1) ZAP

F41E (FB40) SOUND

F421 (FBD0) PLAY

F424 (FC18) MUSIC

F409 (F77C) Output character from X register to screen.

F430 (F8B2) Entry point for non-maskable interrupt (NMI)

F436 (F865) Output message to top line at position X.

F4C8 (F523) Poll keyboard.

F535 (F590) Write to the 8912 chip.

F57B (F5C1) Output character in A to the printer.

F89B (F8D0) Set up the ASCII character set.

FA6C (FA86) Load up all the 8912 registers.

FAFA (FB14) The high-pitched click.

FB10 (FB2A) The low-pitched click.

6 MATHS, HIRES, AND MUSIC

6.1 Introduction

This chapter is concerned with the ROM subroutines which deal with arithmetic calculations, high-resolution graphics, and the sound facilities.

These three subjects have been grouped together because the ROM subroutines are all quite complex and their use can save a lot of programming work and memory space.

6.2 Maths

If a machine code program needs to do any arithmetic (such as multiplication or division), it will normally require a specially written set of subroutines. With the exception of single-byte add and subtract, the 6502 cannot directly do any arithmetic.

The BASIC interpreter contains a large number of useful subroutines to handle all of its mathematics. Often a few subroutine calls can save you hundreds of bytes of memory. It must be pointed out, though, that calling such subroutines is a little risky. Should an error arise (such as a division by zero error) you will be rudely dumped back to BASIC. Also, you may find that the routines are not fast enough for your needs – especially functions like TAN and LOG.

The maths routines make good use of page 0 – see Chapter 5.

FLOATING POINT

All calculations are done in 'floating point', In BASIC, numbers can be stored in either a floating-point variable (e.g., B) or an integer variable (e.g., C%). A variable such as B can contain any number up to an accuracy of nine digits, with a decimal point that 'floats' up and down the number. An integer variable can only contain a number between – 32768 and +32767, without a decimal point. Although in theory this would seem to be faster to process, the ROM can only manipulate numbers in floating-point form, so converts any integers that are used.

When floating-point numbers are stored in memory (e.g., for variables and array elements), they occupy 5 bytes of memory. This is made up as follows:

Byte 1: the exponent of the number.

Bytes 2 to 5: the mantissa of the number (most significant bit to least significant bit).

The number is translated into binary, and then the decimal point altered so that it is to the left of the most significant digit (which in binary is always going to be 1). The exponent represents the number of decimal places that the decimal point has been moved, so if the mantissa is M and the exponent is E, then the value of the number is O.M. * 2" E.

There are three considerations:

1. When the exponent is positive (meaning that the number is 1 or greater), the exponent will be #80 upwards. When the exponent is negative (meaning that the number is 0 – 0.99999) the exponent is subtracted from #80.

For example, an exponent of – 4 is #7C; an exponent of +4 is #84.

2. Since the leftmost bit of the mantissa is always going to be 1, this bit is assumed and replaced with a bit that represents the sign of the number (0 is positive, 1 negative).

3. When the number is zero the exponent is set to #00.

This can be quite difficult to follow, so here are a few examples of how numbers are stored. Do not worry if you do not understand floating point fully – it does not prevent you from using the subroutines!

EXAMPLES OF FLOATING POINT

1. +4 is Exponent: #83 Mantissa #00 #00 #00 #00. The most significant bit has been

replaced by the positive sign. The exponent is #83 because the number 100.0 is stored as 0.1.

2. –6 is Exponent: #83 Mantissa #CO #00 #00 #00. In this case, the lost bit at the front of the number has been replaced with '1' because the mantissa is negative.

INTERNAL FLOATING-POINT NUMBERS

All calculations involve two operands, and since intermediate numbers need to be stored somewhere. there exist two floating-point accumulators. These are similar in format to the floating-point numbers stored in memory, except that the sign of the mantissa does not overwrite the highest bit in the mantissa. To save time, the sign is stored as a sixth byte, with its top bit cleared for positive numbers and set for negative numbers.

A mantissa of zero is still represented by a zero exponent.

The two accumulators are known as ACC1 and ACC2 in the remainder of this chapter. Unless stated to the contrary, ACC1 is used to receive the result of any calculation, with the exception of some of the transfer commands. As discussed in Chapters 3 and 4, ACC1 is used by the extension commands ! and & when passing numeric data, as well as in the formula evaluation subroutine.

LOCATION OF NUMBERS

ACC1 is stored between #DO and #D5, as described above. ACC2 follows ACC1 at #D8 to #DD.

When a floating-point number is turned into a string of ASCII characters, this string is always stored between #100 and #10F. The reverse procedure, however, uses the pointer #E9, #EA to indicate the start address. Remember also that version 1.0 ROMs have a bug that puts the attribute *02* (instead of #20) at the front of the number.

When the routines refer to a number in memory, two of the 6502 registers are used to point to the start of this area.

ROM ROUTINES FOR MATHS

As usual, version 1.0 ROM addresses are given first, followed by the equivalent version 1.1 address in brackets.

Movement of data

Convert integer in Y(low) and A(high) to ACC1. #D3ED (#D499).

Convert ACC1 to integer in Y (low) and A (high). #D871 (#D92C).

Move from memory location A (low), Y (high) to ACC2. #DD4D (#DD51).

Move from memory location A (low), Y (high) to ACC1. #DE73 (#DE77).

Move ACC1 to memory location X (low), Y (high). #DEA5 (#DEAD).

Move ACC2 to ACC1. #DECD (#DED5).

Move ACC1 to ACC2. #DEDD (#DEE5).

Input ACC1 from an ASCII string (as in the VAL function). You must call subroutine #E8 first, then call #DFCF (#DFE7). The string should be terminated by a comma, colon, or #00.

Output ACC1 into an ASCII string, as in the STR$ function (the string is stored at #100 upwards, ending with #00). #E0D1 (#EOD5).

Arithmetic

Add a half to ACC1. #DA79 (#DB04).

Calculate a number in memory (A low, Y high) minus ACC1. #DA80 (#DBOB).

Add a number in memory (A low, Y high) to ACC1. #DA97 (#DB22).

Multiply a number in memory (A low, Y high) by ACC1. #DCB7 (#DCED).

Multiply ACC1 by ten. #DDA3 (#DDA7).

Divide ACC1 by ten. #DDBF (#DDC3).

Divide a number in memory by ACC1. #DDE0 (#DDE4).

Divide ACC2 by ACC1. #DDE5 (#DDE9).

Raise ACC2 to the power of a number in memory. #E231 (#E235).

Multiply by – l. #E26D (#E271).

Mathematical functions – as used in BASIC

LOG	**(ACC1)**	**#DC79**	**#DCAF**
SGN	**(ACC1)**	**#DF12**	**#DF21**
ABS	**(ACC1)**	**#DF31**	**#DF49**
INT	**(ACC1)**	**#DFA5**	**#DFBD**
SQR	**(ACC1)**	**#E22A**	**#E22E**
RND	**(ACC1)**	**#E34B**	**#E34F**
COS	**(ACC1)**	**#E387**	**#E38B**
SIN	**(ACC1)**	**#E38E**	**#E392**
TAN	**(ACC1)**	**#E3D7**	**#E3DB**
ATN	**(ACC1)**	**#E43B**	**#E43F**

6.3 High-resolution graphics

HOW HIRES WORKS

The switch from TEXT to HIRES mode is often assumed to be a fixed procedure. In fact it is possible to mix HIRES and TEXT in combinations other than the usual 200 by 240 pixels followed by three low-resolution lines.

The standard HIRES effect is obtained by clearing down the area of memory between #A000 and #BF3F and writing a special attribute – #1E to the last text screen position. All the other processes, such as copying the character sets and spacing out the text lines, are just cosmetic.

The way of mixing HIRES and TEXT is a little complicated and can be best thought of as a third graphics mode – SEMI-HIRES. This third mode can be entered while in TEXT mode, but to BASIC, you remain in TEXT mode. Because of this, the bottom half of the screen cannot be used for the HIRES area, since this would then overwrite the character sets and conflict with the text screen.

What happens in SEMI-HIRES mode is that when a code of #1E is found in the text area, the VDU switches the rest of the screen into HIRES mode, using the HIRES memory. This continues until the attribute #lA is encountered in the HIRES memory. Figure 6.1 explains how each character square shown on the screen relates to two different addresses. For instance, the top left character cell is either #BB80 on the text screen or #A000 on the high-resolution screen.

In SEMI-HIRES mode, you will normally only use up to #B3FF for the high-resolution part; anything below that should be in TEXT mode. If you do go below that, you will wipe out part of the character sets and therefore not be able to display characters on any text areas. In the proper HIRES mode, it does not matter that we overwrite the character sets, since we are only presenting text on the bottom three lines. The last three lines on a screen that is in HIRES mode always use the copied character set at #9800 to #9FFF – this does not apply in SEMI-HIRES mode.

MIXING HIRES AND TEXT

Looking at Fig. 6.1, you will see that on the left side are the addresses which relate to the HIRES screen and on the right are addresses which relate to the text screen. It is important to think in terms of character cells when considering what will happen to the screen. To change

part of the screen into HIRES, you only need the one character position to contain the special attribute – #1E. However, when switching back to TEXT, you should remember that there are eight lines which are now in HIRES. If you only switch one line back to TEXT, then the following lines in that character cell will still be in HIRES – only the rest of that one line will have been altered back in to TEXT mode. Therefore, to switch the rest of the screen to TEXT, you need to have that TEXT attribute on the last line of the eight which correspond to a particular character cell. If you want to change modes in the middle of a line, you will need one TEXT attribute placed after each of the high-resolution lines. Do not worry if you cannot follow this – the examples will clarify the issue.

For instance, if we wish the top eight lines of the screen (where CAPS is displayed) to be in HIRES mode, we POKE #BB80,30 and POKE #A13F,26. Now you will find that the text area #BB80 to #BBA7 has been replaced by a HIRES area (#A000 to #A13F). There are two important points to note:

1. The very first location in the high-resolution area (in our last example this is #A000) cannot be used.

2. The last location must be left alone, since it is responsible for switching back into TEXT mode.

In BASIC, the system will still think that it is in TEXT mode – little does it know what you have done to its screen!

This means that it will give you a 'DISP TYPE MISMATCH ERROR' if you try any graphics commands. This is easily overcome by ORing location #2CO with 1. Obviously you should be careful that the HIRES area #A000 to #A13F is not being used by BASIC – a HIMEM #9FFF will do the trick! After this, you will be able to use all the HIRES commands as normal. Unless you have previously entered HIRES mode properly, the cursor position and PATTERN register will be undefined.

Remember not to draw over the bottom part of the screen!

The SEMI-HIRES mode has the advantage of letting you have a screen composed of half text and half graphics. In addition you will recover at least 2K of memory space (#9800 to #9FFF).

To create such a set-up, you would:

1. POKE #BB80,30 (switch to HIRES).

2. POKE #B3FF,26 (end of HIRES area). Note that #B3FF is the address of the lowest line within the required character block.

3. POKE #2CO,PEEK(#2CO) OR 1 (to allow HIRES commands).

4. For version 1.0: POKE#26F,12:DOKE#26D,#BDD8; for version 1.1: POKE #27E,12: DOKE #27A, #BE00: DOKE #278, #BE28: DOKE #27C,440.

These POKEs and DOKEs make sure that only the bottom half of the screen scrolls.

5. Clear the screen.

It is advisable to do all these commands in one go. You will notice that the HIRES screen contains vertical lines. This can be cleared by using the FILL command (filling the screen with #40). Alternatively, if you do a HIRES command beforehand, this will not be necessary.

When using this HIRES area, remember to leave location #B3FF well alone! Your first HIRES commands should set the pattern and cursor positions.

MIXING HIRES AND TEXT ON ONE LINE

Here is something quite remarkable! Type the following as a one line command:

```
FOR K=OT07:J=K*40

POKE #A022+ J,K+1:

POKE #A023+ J,26: NEXT:

POKE #BBA1,30
```

The CAPS sign should burst into colour!

In this way, part of a text line can be switched to HIRES, and back again, and the attributes in that HIRES area affect the rest of the text on that line.

This feature opens up all sorts of possibilities. For instance, it is now possible to flash just part of one character on the text screen.

HIRES AND INTERRUPTS

On version 1.0 ROMs the TEXT and HIRES commands use the third software timer to wait for two interrupts after storing the HIRES or TEXT attribute at #BFDF. This means that interrupts must be running normally at the time you use the commands. On version 1.1 ROMs this applies only to the TEXT command.

MACHINE CODE SUBROUTINES

In order to perform the BASIC HIRES instructions (such as CIRCLE, DRAW, and CURSET), a machine code program must first set up a number of parameters in the area #2E1 to #2EF. These are always in the same format as the actual BASIC command and must be stored as 2-byte integer values at #2E1 upward. For instance, 'CIRCLE 20,1' would require: #2E1: #14; #2E2: #0; #2E3: #1; #2E4: #0.

Consult the BASIC handbook for the format of each command.

The range of the parameters you pass will be checked, as it would be in BASIC, and location #2EO is set to 1 if there are any errors.

As usual, the first address is for version 1.0 ROMs and the address in brackets is for version 1.1 ROMs:

HIRES #E9BB	(#EC33) – no parameters.
TEXT #E9A9	(#EC21) – no parameters.
PAPER #F17F	(#F204) – #2E1,2: paper colour.
INK #F18B	(#F210) – #2E1,2: ink colour.
CURSET #F02D	(#F0C8) – #2E1,2: X; #2E3,4: Y; #2E5,6: FB code
DRAW #F079	(#F110) – parameters as for CURSET.
POINT #F141	(#F1C8) – #2E1,2: X; #2E3,4: Y. Returns #FF or #00 in #2E1, depending on whether the point is set or cleared.
CURMOV #F064	(#F0FD) – see CURSET.
CHAR #F0A5	(#F12D) – #2E1,2:ASCII code; #2E3,4: character set; #2E5,6:FB code.
FILL #F1E5	(#F268) – #2E1,2: No. of rows; #2E3,4:No. of cells; #2E5,6: value.
CIRCLE #F2E5	(#F37F) – #2E1,2:radius; #2E3,4: FB code.

PATTERN – No call is needed, simply POKE #213 with the required pattern.

You will find that these subroutines are only slightly quicker than the equivalent BASIC command. Chapter 7 explains some faster methods of using high resolution.

6.4 Sound and music

ROM ROUTINES

All of the BASIC commands for sound and music can be easily accessed from machine code. The same method of supplying parameters is used as for the graphics commands.

PLAY	#F421 (#FBDO)	– #2E1,2: tone enable; #2E3,4: noise enable; #2E5,6: envelope; #2E7,8: envelope period.
MUSIC	#F424 (#FC18)	#2E1,2: channel; #2E3,4: octave; #2E5,6: note; #2E7,8: volume.
SOUND	#F41E (#FB40)	#2E1,2: channel; #2E3,4: period; #2E5,6: volume.

ZAP	#F41B (#FAE1).
EXPLODE	#F418 (#FACB).
PING	#F412 (#FA9F).
SHOOT	#F415 (#FA9B)
KEYCLICK-1	#FAFA (#FB14)
KEYCLICK-2	#FB10 (#FB2A)

In machine code a program can also directly access the sound chip. Chapter 1 describes this device and gives all the details about the registers.

In order to write to the 8912 sound chip, you must call a subroutine at #F535 (#F590) for every register that you need to change. This is done by putting the register number (0 to #E) in the accumulator and loading the data in the X register.

Please note that the envelope shape that you put in register #D is different from that used in the PLAY command. Refer to Chapter 1 for details of which values relate to which envelope.

Since any one musical effect may require the setting of up to 14 registers, the ROM conveniently provides a subroutine to do just that. The routine starts at #FA6C (#FA86) and assumes that the X and Y registers point to the start of a 14-byte table (X is low, Y is high, and the table refers to registers 0 to #D).

Often it is useful to call this loader in order to get the type of sound required, and then change individual registers to alter pitch, volume, etc.

The subroutine that loads up a register with a value suffers from being rather inefficient. You will find a better version in the speech synthesis program of Chapter 9.

FASTER HIGH-RESOLUTION GRAPHICS

7.1 Objectives

Chapter 6 dealt with the subject of high-resolution graphics when using the routines contained in the ROM. This chapter will present you with a much faster set of routines to be incorporated into your own programs. Depending on what your program is doing, you could just use one or more of the subroutines, or just make use of the concepts involved.

These special subroutines occupy RAM between #1200 and #17FF, with other extensions and examples put elsewhere. The relocating program in Chapter 4 can be used to move the high-resolution routines to a place suited to your needs.

Here is a summary of the high-resolution routines, to whet your appetite!

1. Plot character cell. This is an extremely fast routine for putting a character cell on the screen.

2. Test routines to look for collisions between a character cell and other objects.

3. A fast equivalent to CURSET and POINT.
4. An easy-to-use routine for drawing larger, odd-shaped objects, with colours.
5. A colouring facility for the character plot routine.
6. A paint facility to fill in irregular shapes. This can be used from within BASIC.
7. A compactor routine for the high-resolution screen. This makes it possible to store a picture in a compressed form.

7.2 The theory behind the fast plotting routine

THE USE OF TABLES

The normal way of plotting points on the screen involves two things:

1. Finding the correct address for a given X, Y position.
2. Determining the bit position within the byte at that address.

When you are plotting a whole 6 by 8 character, you would then usually shift that character a number of times, depending on the bit position. If you are just dealing with one pixel, you use the bit position (as a number 0 to 5) in order to reference a table containing the numbers: #20, #10, #8, #4, #2, and #1. One of these numbers is then either ANDed or ORed with the contents of the address in order to set, clear, or test the bit.

The usual way of calculating all this information is to:

1. Multiply the Y co-ordinate by 40 and add #A000.
2. Add this to the X co-ordinate divided by 6, to give the address of the pixel.

3. Use the remainder from this last division to give the bit position.

What really slows up this procedure is the division by 6. Division by 4 or 8 can be done with the use of simple shifts right, but in order to divide by 6 you must use a rather cumbersome divide routine. Multiplying by 40 is also a difficult task in machine code.

The method used here gives an increased speed of about 10 times in comparison to using the ROM routines. When compared with using BASIC, the acceleration is seventy-fold!

The secret is that a 1K table is generated and used, and this provides all the addresses and bit positions very rapidly. This table sits at 1400 to 17FF (but is easily relocated) and is made up of four 256-byte tables, as follows:

1. The high value of the address at the start of every HIRES line.
2. The low value of this address.
3. The number of character cell positions across the line, for each possible value of X.
4. The bit position within the byte for each possible value of X, multiplied by 16. (This will be explained shortly.)

It can be seen that given a particular (X, Y) coordinate, the correct address can be quickly located.

If you want to draw a character at a given point (the top left position of the character) you will

then have to shift each line of that character a number of times – as given by part four of the table. Since this is rather long-winded, we save time by shifting the character from one to five times and saving all the possibilities in a table (#60 bytes long). The location of this table is supplied by the calling program, since there could be any number of these character tables. These short tables are organized as 6 sets of 16 bytes – one per number of shifts required. Each of these consists of two sets of 8 bytes, corresponding to the patterns that make up the character you are plotting. You need 2 bytes since the character, when shifted, falls between one character cell and the next. Most of the routines in this chapter will rely on the 1K table, but apart from that there are three different types of graphics, as follows:

1. Graphics routines that draw a previously analysed character.
2. Graphics routines that draw a shape that is not in the form of character cells, e.g., 12 dots across by 4 dots down.
3. Graphics routines to handle individual pixels, for use in routines like PAINT.

DRAWING AN ANALYSED CHARACTER

Once you have worked out all the possible shifted values of a character, it is quite simple to display that character, since the last part of the 1K table gives the offset into the #60 byte table (which is why the values were multiplied by 16 beforehand).

In order to display a character, and subsequently remove it, you only need to use the one routine, employing the exclusive-OR function.

In other words, if you exclusive-OR the letter ‘A’ onto a blank screen, the ‘A’ will appear, until you re-do the exclusive-OR, when the ‘A’ disappears. This saves having separate routines to draw and remove a character.

The added advantage is that if you exclusive-OR on top of an existing pattern, you keep that pattern intact after the second time you call the routine.

One disadvantage is that you must be careful not to exclusive-OR over the top of an attribute or the screen could go haywire!

```
1200:  A2 00       LDX  #$00
1202:  A0 00       LDY  #$00
1204:  86 80       STX  $80
1206:  8A          TXA
1207:  0A          ASL
1208:  0A          ASL
1209:  0A          ASL
120A:  0A          ASL
120B:  99 00 15    STA  $1500,Y
120E:  A5 80       LDA  $80
1210:  99 00 14    STA  $1400,Y
1213:  C8          INY
1214:  F0 0C       BEQ  $1222
1216:  E8          INX
1217:  E0 06       CPX  #$06
1219:  D0 EB       BNE  $1206
121B:  E6 80       INC  $80
121D:  A2 00       LDX  #$00
```

```
121F:  F0 E5        BEQ  $1206
1221:  EA           NOP
1222:  A9 A0        LDA  #$A0
1224:  85 81        STA  $81
1226:  A9 00        LDA  #$00
1228:  A8           TAY
1229:  85 80        STA  $80
122B:  A5 80        LDA  $80
122D:  99 00 16     STA  $1600,Y
1230:  A5 81        LDA  $81
1232:  99 00 17     STA  $1700,Y
1235:  A5 80        LDA  $80
1237:  18           CLC
1238:  69 28        ADC  #$28
123A:  85 80        STA  $80
123C:  90 02        BCC  $1240
123E:  E6 81        INC  $81
1240:  C8           INY
1241:  D0 E8        BNE  $122B
1243:  60           RTS
```

Program 7.1 is the first part of our graphics routines, which sets up the tables.

Following the table set-up subroutine, which you call once, we have the routine (Program 7.2) you call for *each* character which you will want to display eventually.

To call this routine, you must set up the X register to the appropriate ASCII value, A to the low part of a spare #60 byte address, and the Y register to the high value of this #60 byte table. The routine uses the normal character set area at #9800, though you may choose to alter this to the alternate area at #9C00.

```
1244:  85 8C        STA  $8C
1246:  84 8D        STY  $8D
1248:  A9 00        LDA  #$00
124A:  85 80        STA  $80
124C:  A9 00        LDA  #$00
124E:  85 85        STA  $85
1250:  86 84        STX  $84
1252:  06 84        ASL  $84
1254:  26 85        ROL  $85
1256:  06 84        ASL  $84
1258:  26 85        ROL  $85
```

```
125A:  06 84     ASL  $84
125C:  26 85     ROL  $85
125E:  18        CLC
125F:  A5 85     LDA  $85
1261:  69 98     ADC  #$98
1263:  85 85     STA  $85
1265:  A0 00     LDY  #$00
1267:  B1 84     LDA  ($84),Y
1269:  85 87     STA  $87
126B:  A9 00     LDA  #$00
126D:  85 88     STA  $88
126F:  A6 80     LDX  $80
1271:  F0 0B     BEQ  $127E
1273:  46 87     LSR  $87
1275:  66 88     ROR  $88
1277:  CA        DEX
1278:  D0 F9     BNE  $1273
127A:  46 88     LSR  $88
127C:  46 88     LSR  $88
127E:  A5 87     LDA  $87
1280:  91 8C     STA  ($8C),Y
1282:  98        TYA
1283:  48        PHA
1284:  09 08     ORA  #$08
1286:  A8        TAY
1287:  A5 88     LDA  $88
1289:  91 8C     STA  ($8C),Y
128B:  68        PLA
128C:  A8        TAY
128D:  C8        INY
128E:  C0 08     CPY  #$08
1290:  D0 D5     BNE  $1267
1292:  18        CLC
1293:  A5 8C     LDA  $8C
1295:  69 10     ADC  #$10
1297:  85 8C     STA  $8C
1299:  90 02     BCC  $129D
129B:  E6 8D     INC  $8D
129D:  E6 80     INC  $80
129F:  A5 80     LDA  $80
12A1:  C9 06     CMP  #$06
12A3:  D0 C0     BNE  $1265
12A5:  60        RTS
```

Program 7.2 Create character table (#1244–#12A5)

The subroutine of Program 7.3 is called by the character drawing and testing routines, and calculates the address of an X, Y co-ordinate.

```
12A6:  A4 8B       LDY  $8B
12A8:  B9 00 16    LDA  $1600,Y
12AB:  85 82       STA  $82
12AD:  B9 00 17    LDA  $1700,Y
12B0:  85 83       STA  $83
12B2:  A4 8A       LDY  $8A
12B4:  B9 00 14    LDA  $1400,Y
12B7:  18          CLC
12B8:  65 82       ADC  $82
12BA:  85 82       STA  $82
12BC:  90 02       BCC  $12C0
12BE:  E6 83       INC  $83
12C0:  B9 00 15    LDA  $1500,Y
12C3:  18          CLC
12C4:  65 8C       ADC  $8C
12C6:  85 8C       STA  $8C
12C8:  90 02       BCC  $12CC
12CA:  E6 8D       INC  $8D
12CC:  60          RTS
```

Program 7.3 Calculate address (#12A6–#12CC)

And now to the subroutine which you call when you want to display something (Program 7.4).

The routine exclusive-Ors the character stored in the table pointed to by the A and Y registers onto the screen at the X, Y co-ordinate given by addresses 8A and 8B respectively. Register A is the low part of the table's address and register Y is the high value. For example:

```
LDA #45          X co-ordinate
STA  $8A
LDA #60          Y co-ordinate
STA  $8B
LDA #40          F40 is address of the table.
LDY #0F
```

```
12CD:  85 8C       STA  $8C
12CF:  84 8D       STY  $8D
12D1:  20 A6 12    JSR  $12A6
12D4:  A0 00       LDY  #$00
12D6:  B1 82       LDA  ($82),Y
12D8:  51 8C       EOR  ($8C),Y
12DA:  91 82       STA  ($82),Y
12DC:  98          TYA
```

```
12DD:  48        PHA
12DE:  09 08     ORA  #$08
12E0:  A8        TAY
12E1:  B1 8C     LDA  ($8C),Y
12E3:  AA        TAX
12E4:  68        PLA
12E5:  A8        TAY
12E6:  C8        INY
12E7:  8A        TXA
12E8:  51 82     EOR  ($82),Y
12EA:  91 82     STA  ($82),Y
12EC:  18        CLC
12ED:  A5 82     LDA  $82
12EF:  69 27     ADC  #$27
12F1:  85 82     STA  $82
12F3:  90 02     BCC  $12F7
12F5:  E6 83     INC  $83
12F7:  C0 08     CPY  #$08
12F9:  D0 DB     BNE  $12D6
12FB:  60        RTS
```

Program 7.4 Display character cell (#12CD–#12FB)

7.3 Collisions

The last routine can be used to put a character on and take a character off the screen, but one important facility is to be able to test for collisions – in games, etc.

There are two main cases to consider:

1. When an object is prohibited from running into other objects, including any screen border.

2. When an object is being shot at, by some other moving character that is using exclusive-OR, and it is possible that a given character has been 'destroyed'. An example of this is where a laser base is destroyed by a rain of missiles.

In the first case, we need a routine that looks at a given area, and if there is room for your character, it returns with the zero flag set. The second subroutine examines the area where your character was last seen, and returns the zero flag set if your character is still in one piece.

The timings for calling these two routines are quite different:

1. The first routine is looking for a clear area, so call it before drawing on the character.

2. The second routine must be called after all screen objects have been moved and drawn. The assumption here is that after one character has been drawn, another may overlap it and thus wipe part of it out.

To call either of these routines you must set up the A and Y registers, as in the previous subroutine, with 8A and 8B containing the X and Y positions of the character on the screen.

```
12FC:  85 8C       STA  $8C
12FE:  84 8D       STY  $8D
1300:  20 A6 12    JSR  $12A6
1303:  A0 00       LDY  #$00
1305:  B1 82       LDA  ($82),Y
1307:  31 8C       AND  ($8C),Y
1309:  D0 1F       BNE  $132A
130B:  98          TYA
130C:  48          PHA
130D:  09 08       ORA  #$08
130F:  A8          TAY
1310:  B1 8C       LDA  ($8C),Y
1312:  AA          TAX
1313:  68          PLA
1314:  A8          TAY
1315:  C8          INY
1316:  8A          TXA
1317:  31 82       AND  ($82),Y
1319:  D0 0F       BNE  $132A
131B:  18          CLC
131C:  A5 82       LDA  $82
131E:  69 27       ADC  #$27
1320:  85 82       STA  $82
1322:  90 02       BCC  $1326
1324:  E6 83       INC  $83
1326:  C0 08       CPY  #$08
1328:  D0 DB       BNE  $1305
132A:  60          RTS
```

Program 7.5 Test for collisions (#12FC–#132A)

The first subroutine is given in Program 7.5.

The second routine,which tests to see if a character is a whole is listed in 7.6

```
132B:  85 8C        STA  $8C
132D:  84 8D        STY  $8D
132F:  20 A6 12     JSR  $12A6
1332:  A0 00        LDY  #$00
1334:  B1 82        LDA  ($82),Y
1336:  31 8C        AND  ($8C),Y
1338:  D1 8C        CMP  ($8C),Y
133A:  D0 23        BNE  $135F
133C:  98           TYA
133D:  48           PHA
133E:  09 08        ORA  #$08
1340:  A8           TAY
1341:  B1 8C        LDA  ($8C),Y
1343:  85 8E        STA  $8E
1345:  68           PLA
1346:  A8           TAY
1347:  C8           INY
1348:  A5 8E        LDA  $8E
134A:  31 82        AND  ($82),Y
134C:  C5 8E        CMP  $8E
134E:  D0 0F        BNE  $135F
1350:  18           CLC
1351:  A5 82        LDA  $82
1353:  69 27        ADC  #$27
1355:  85 82        STA  $82
1357:  90 02        BCC  $135B
1359:  E6 83        INC  $83
135B:  C0 08        CPY  #$08
135D:  D0 D5        BNE  $1334
135F:  60           RTS
```

Program 7.6 Test for destroyed character (#132B–#135F)

7.4 Fast single-point plotter

This short routine (Program 7.7) uses the 1K table set up at 1400 to 17FF to provide an extremely fast method of dealing with individual pixels. It takes registers X and Y which give the co-ordinates and returns an address at #82 and a bit position in the accumulator. This bit position is in the form of one bit set in the byte – ready to be ORed with the contents of the address.

```
1360:  18           CLC
1361:  B9 00 16     LDA  $1600,Y
1364:  7D 00 14     ADC  $1400,X
1367:  85 82        STA  $82
```

Program 7.7 *(continues)*

```
1369:  A9 00        LDA  #$00
136B:  79 00 17     ADC  $1700,Y
136E:  85 83        STA  $83
1370:  BD 00 15     LDA  $1500,X
1373:  4A           LSR
1374:  4A           LSR
1375:  4A           LSR
1376:  4A           LSR
1377:  A8           TAY
1378:  B9 9E 13     LDA  $139E,Y
137B:  A0 00        LDY  #$00
137D:  60           RTS
```

Program 7.7 Fast pixel-addressing subroutine (#1360–#137D)

This subroutine also uses a short table of the bit positions at 139E:

139E: 20 10 08 04 02 01

The next subroutines (Programs 7.8 to 7.11) use a fast plotting routine, as follows:

1. Set (OR) a dot – #137E.
2. Remove (AND after inverting) a dot – 01386.
3. Alter dot (exclusive-OR) – #1390.
4. Test dot – return the zero flag set if dot is clear – #1398.

To call any of these subroutines, you only need to set registers X and Y to the correct horizontal and vertical values.

```
137E:  20 60 13    JSR  $1360
1381:  11 82       ORA  ($82),Y
1383:  91 82       STA  ($82),Y
1385:  60          RTS
```

Program 7.8 Set dot (#137E–#1385)

```
1386:  20 60 13    JSR  $1360
1389:  49 FF       EOR  #$FF
138B:  31 82       AND  ($82),Y
138D:  91 82       STA  ($82),Y
138F:  60          RTS
```

Program 7.9 Clear dot (#1386–#138F)

```
1390:  20 60 13    JSR  $1360
1393:  51 82       EOR  ($82),Y
1395:  91 82       STA  ($82),Y
1397:  60          RTS
```

Program 7.10 Alter dot (#1390–#1397)

```
1398:  20 60 13    JSR  $1360
139B:  31 82       AND  ($82),Y
139D:  60          RTS
```

Program 7.11 Test dot (#1398–#139D)

COLOURING THE SCREEN

For the fastest possible graphics, it is advisable to colour the screen with preset paper and ink attributes to the left of your graphics area. If it is important to colour a character, then the following routines in Program 7.12 can be used.

For each character you need a 16-byte area, half filled with the attribute for each of the eight lines. The remaining 8 bytes are used to store the contents of the screen before it is overwritten. Normally you will store a set of INK colours (i.e., numbers between 0 and 7), but remember that it is possible to specify a PAPER colour, or perhaps even a flashing attribute.

The routine is called with X and Y registers set to the screen position (top left) where the colours are to be stored. Also, #8C, #8D should be set to the address of the 16-byte area that is being used for this character.

```
1187:  20 60 13     JSR  $1360
118A:  B1 82        LDA  ($82),Y
118C:  48           PHA
118D:  98           TYA
118E:  AA           TAX
118F:  09 08        ORA  #$08
1191:  A8           TAY
1192:  68           PLA
1193:  91 8C        STA  ($8C),Y
1195:  8A           TXA
1196:  A8           TAY
1197:  B1 82        LDA  ($82),Y
```

Program 7.12 *(continues)*

```
1199:  C9 40        CMP  #$40
119B:  D0 04        BNE  $11A1
119D:  B1 8C        LDA  ($8C),Y
119F:  91 82        STA  ($82),Y
11A1:  18           CLC
11A2:  A5 82        LDA  $82
11A4:  69 27        ADC  #$27
11A6:  85 82        STA  $82
11A8:  90 02        BCC  $11AC
11AA:  E6 83        INC  $83
11AC:  C8           INY
11AD:  C0 08        CPY  #$08
11AF:  D0 D9        BNE  $118A
11B1:  60           RTS
```

Program 7.12 Colour character subroutine (#1187–#11B1)

Having drawn the colours, and perhaps worked out a new position for the character, you must remove the attributes, restoring the screen to its former glory.

```
11B2:  20 60 13     JSR  $1360
11B5:  A5 8C        LDA  $8C
11B7:  09 08        ORA  #$08
11B9:  85 8C        STA  $8C
11BB:  B1 8C        LDA  ($8C),Y
11BD:  91 82        STA  ($82),Y
11BF:  18           CLC
11C0:  A5 82        LDA  $82
11C2:  69 27        ADC  #$27
11C4:  85 82        STA  $82
11C6:  90 02        BCC  $11CA
11C8:  E6 83        INC  $83
11CA:  C8           INY
11CB:  C0 08        CPY  #$08
11CD:  D0 EC        BNE  $11BB
11CF:  60           RTS
```

Program 7.13 Remove colours (#11B2–#11CF)

This is done by the subroutine at #11B2, listed in Program 7.13.
If you intend to colour moving objects by using these two subroutines, follow this order of events:

1. Draw all the objects, using the exclusive-OR character facility provided.
2. Fill in the colour attributes, where required.
3. Delay as necessary, and work out new positions of objects, etc.
4. Remove colours, restoring parts of the screen.
5. Use exclusive-OR at the old positions to remove objects.

This sequence is important because you may get into trouble using exclusive-OR over attributes, since the attributes may be altered to one of the ‘nasty’ control codes and cause the picture to break up.

7.6 Drawing larger shapes

Although the character drawing subroutines are quite fast, it can be quite awkward to have to work out graphics in terms of 6 by 8 character cells. The following routine (Program 7.14), though still using the special graphics table, moves away from using character cells, and lets you draw an irregular shape, complete with colour.

All you have to do is provide the subroutine with the address of your object, plus details of its height (in pixels) and width (in character cells).

You must set up this information as follows:

1. Store the graphics shape in a free area of memory. The area must be pointed to by #8C, #8D.

2. A second free area is required, equal in size to the first, pointed to by #80, #81.

3. The data for the object must be stored line by line, with 1 byte for

each 6 pixels, or an attribute, and the number of bytes across

should be stored at #8E.

4. The number of lines down is required at location 8F.

5. Finally, you must load up the X and Y registers with the appropriate screen position (top left of the object).

With the parameters stored in exactly the same way, you call the routine at #115E in order to remove your artwork (Program 7.15).

```
10F2:  20 60 13    JSR  $1360
10F5:  BD 00 15    LDA  $1500,X
10F8:  4A          LSR
10F9:  4A          LSR
10FA:  4A          LSR
```

Program 7.14 *(continues)*

```
10FB:  4A          LSR
10FC:  85 8A       STA  $8A
10FE:  A5 8F       LDA  $8F
1100:  85 85       STA  $85
1102:  A0 00       LDY  #$00
1104:  A9 00       LDA  #$00
1106:  85 84       STA  $84
1108:  A9 00       LDA  #$00
110A:  85 86       STA  $86
110C:  A6 8A       LDX  $8A
110E:  B1 82       LDA  ($82),Y
1110:  91 80       STA  ($80),Y
1112:  B1 8C       LDA  ($8C),Y
1114:  C9 40       CMP  #$40
1116:  90 19       BCC  $1131
1118:  29 BF       AND  #$BF
111A:  E0 00       CPX  #$00
111C:  F0 07       BEQ  $1125
111E:  18          CLC
111F:  6A          ROR
1120:  66 86       ROR  $86
1122:  CA          DEX
1123:  D0 FA       BNE  $111F
1125:  05 84       ORA  $84
1127:  46 86       LSR  $86
1129:  46 86       LSR  $86
112B:  A6 86       LDX  $86
112D:  86 84       STX  $84
112F:  09 40       ORA  #$40
1131:  91 82       STA  ($82),Y
1133:  C8          INY
1134:  C4 8E       CPY  $8E
1136:  D0 D0       BNE  $1108
1138:  18          CLC
1139:  A5 82       LDA  $82
113B:  69 28       ADC  #$28
113D:  85 82       STA  $82
113F:  90 02       BCC  $1143
1141:  E6 83       INC  $83
1143:  18          CLC
1144:  A5 80       LDA  $80
1146:  65 8E       ADC  $8E
1148:  85 80       STA  $80
114A:  90 02       BCC  $114E
114C:  E6 81       INC  $81
114E:  18          CLC
114F:  A5 8C       LDA  $8C
1151:  65 8E       ADC  $8E
1153:  85 8C       STA  $8C

1155:  90 02       BCC  $1159
1157:  E6 8D       INC  $8D
1159:  C6 85       DEC  $85
115B:  D0 A5       BNE  $1102
115D:  60          RTS
```

Program 7.14 Draw large shape (#10F2–#115D)

```
115E:  20 60 13    JSR  $1360
1161:  A6 8F       LDX  $8F
1163:  A4 8E       LDY  $8E
1165:  88          DEY
1166:  B1 80       LDA  ($80),Y
1168:  91 82       STA  ($82),Y
116A:  88          DEY
116B:  10 F9       BPL  $1166
116D:  18          CLC
116E:  A5 80       LDA  $80
1170:  65 8E       ADC  $8E
1172:  85 80       STA  $80
1174:  90 02       BCC  $1178
1176:  E6 81       INC  $81
1178:  18          CLC
1179:  A5 82       LDA  $82
117B:  69 28       ADC  #$28
117D:  85 82       STA  $82
117F:  90 02       BCC  $1183
1181:  E6 83       INC  $83
1183:  CA          DEX
1184:  D0 DD       BNE  $1163
1186:  60          RTS
```

Program 7.15 Remove large shape (#115E–#1186)

The method used to show graphics in these last two routines is simply to overwrite parts of the screen, and not use exclusive-OR. This will occasionally be more convenient, since you will then not have to worry about drawing over the top of other attributes. However, I have not provided any method of detecting collisions when using this method.

7.7 Examples

EXAMPLE 1 – A DEMONSTRATION OF THE CHARACTER DRAW FACILITY

This first example is based on the character cell type of graphics. It moves a multicoloured square of AB and CD back and forth.

Of course, if you modified the character definitions for A to D, then you would see some other graphics pattern crossing the screen.

```
0D00:  20 00 12    JSR  $1200
0D03:  A2 41       LDX  #$41
0D05:  A9 00       LDA  #$00
0D07:  A0 0E       LDY  #$0E
0D09:  20 44 12    JSR  $1244
0D0C:  A2 42       LDX  #$42
0D0E:  A9 60       LDA  #$60
0D10:  A0 0E       LDY  #$0E
0D12:  20 44 12    JSR  $1244
0D15:  A2 43       LDX  #$43
0D17:  A9 C0       LDA  #$C0
0D19:  A0 0E       LDY  #$0E
0D1B:  20 44 12    JSR  $1244
0D1E:  A2 44       LDX  #$44
0D20:  A9 20       LDA  #$20
0D22:  A0 0F       LDY  #$0F
0D24:  20 44 12    JSR  $1244
0D27:  A9 10       LDA  #$10
0D29:  85 90       STA  $90
0D2B:  85 91       STA  $91
0D2D:  A9 01       LDA  #$01
0D2F:  48          PHA
0D30:  A6 90       LDX  $90
0D32:  A4 91       LDY  $91
0D34:  86 BA       STX  $BA
0D36:  84 BB       STY  $BB
0D38:  A9 00       LDA  #$00
0D3A:  A0 0E       LDY  #$0E
0D3C:  20 CD 12    JSR  $12CD
0D3F:  18          CLC
0D40:  A5 BA       LDA  $BA
0D42:  69 06       ADC  #$06
0D44:  85 BA       STA  $BA
0D46:  A9 60       LDA  #$60
0D48:  A0 0E       LDY  #$0E
0D4A:  20 CD 12    JSR  $12CD
0D4D:  18          CLC
0D4E:  A5 91       LDA  $91
```

```
0D50:  69 08     ADC  #$08
0D52:  85 8B     STA  $8B
0D54:  A9 20     LDA  #$20
0D56:  A0 0F     LDY  #$0F
0D58:  20 CD 12  JSR  $12CD
0D5B:  A6 90     LDX  $90
0D5D:  86 8A     STX  $8A
0D5F:  A9 C0     LDA  #$C0
0D61:  A0 0E     LDY  #$0E
0D63:  20 CD 12  JSR  $12CD
0D66:  A5 90     LDA  $90
0D68:  38        SEC
0D69:  E9 06     SBC  #$06
0D6B:  AA        TAX
0D6C:  A4 91     LDY  $91
0D6E:  A9 80     LDA  #$80
0D70:  85 8C     STA  $8C
0D72:  A9 0F     LDA  #$0F
0D74:  85 8D     STA  $8D
0D76:  68        PLA
0D77:  48        PHA
0D78:  F0 05     BEQ  $0D7F
0D7A:  20 87 11  JSR  $1187
0D7D:  F0 03     BEQ  $0D82
0D7F:  20 B2 11  JSR  $11B2
0D82:  18        CLC
0D83:  A5 91     LDA  $91
0D85:  69 08     ADC  #$08
0D87:  A8        TAY
0D88:  38        SEC
0D89:  A5 90     LDA  $90
0D8B:  E9 06     SBC  #$06
0D8D:  AA        TAX
0D8E:  A9 90     LDA  #$90
0D90:  85 8C     STA  $8C
0D92:  A9 0F     LDA  #$0F
0D94:  85 8D     STA  $8D
0D96:  68        PLA
0D97:  48        PHA
0D98:  F0 05     BEQ  $0D9F
0D9A:  20 87 11  JSR  $1187
0D9D:  F0 03     BEQ  $0DA2
0D9F:  20 B2 11  JSR  $11B2
0DA2:  68        PLA
0DA3:  49 01     EOR  #$01
0DA5:  D0 08     BNE  $0DAF
0DA7:  A2 FF     LDX  #$FF
0DA9:  CA        DEX
0DAA:  D0 FD     BNE  $0DA9
```

```
0DAC:  F0 81      BEQ  $0D2F
0DAE:  EA         NOP
0DAF:  E6 90      INC  $90
0DB1:  A5 90      LDA  $90
0DB3:  C9 80      CMP  #$80
0DB5:  D0 0D      BNE  $0DC4
0DB7:  A9 10      LDA  #$10
0DB9:  85 90      STA  $90
0DBB:  E6 91      INC  $91
0DBD:  A5 91      LDA  $91
0DBF:  C9 80      CMP  #$80
0DC1:  D0 01      BNE  $0DC4
0DC3:  00         BRK
0DC4:  4C 2D 0D   JMP  $0D2D
```

Program 7.16 Example 1 (#D00–#DC6)

This program (7.16) assumes that BRK will return you to whatever machine code monitor you are using. You should change this to RTS (instruction code #60) if you are just using BASIC. Please note that these routines assume that you are already in HIRES mode – if you are in a machine code routine, call subroutine #E9BB (or V1.1 ROM #EC33) in order to enter HIRES. The call to get back into TEXT mode is #E9A9 (#EC21 Vl. 1 ROM). To run the program call #D00.

The colours for Example 1 are as follows:

F80: 1 1 1 1 2 2 2 2

F90: 6 6 6 6 3 3 3 3

EXAMPLE 2 – USING THE NON-CHARACTER GRAPHICS ROUTINES

```
2000:  A9 40      LDA  #$40
2002:  85 8C      STA  $8C
2004:  A9 20      LDA  #$20
2006:  85 8D      STA  $8D
2008:  85 81      STA  $81
200A:  A9 80      LDA  #$80
200C:  85 80      STA  $80
200E:  A9 05      LDA  #$05
2010:  85 8E      STA  $8E
2012:  A9 09      LDA  #$09
2014:  85 8F      STA  $8F
2016:  A4 90      LDY  $90
2018:  A6 91      LDX  $91
201A:  20 C0 20   JSR  $20C0
201D:  A5 93      LDA  $93
```

```
201F:  F0 0C       BEQ  $202D
2021:  E6 90       INC  $90
2023:  A5 90       LDA  $90
2025:  C9 B0       CMP  #$B0
2027:  D0 D7       BNE  $2000
2029:  C6 93       DEC  $93
202B:  F0 D3       BEQ  $2000
202D:  C6 90       DEC  $90
202F:  A5 90       LDA  $90
2031:  C9 10       CMP  #$10
2033:  D0 CB       BNE  $2000
2035:  E6 93       INC  $93
2037:  D0 C7       BNE  $2000
2039:  00          BRK
203A:  EA          NOP
20C0:  8A          TXA
20C1:  48          PHA
20C2:  98          TYA
20C3:  48          PHA
20C4:  A5 80       LDA  $80
20C6:  48          PHA
20C7:  A5 81       LDA  $81
20C9:  48          PHA
20CA:  20 F2 10    JSR  $10F2
20CD:  A0 10       LDY  #$10
20CF:  CA          DEX
20D0:  D0 FD       BNE  $20CF
20D2:  88          DEY
20D3:  D0 FA       BNE  $20CF
20D5:  68          PLA
20D6:  85 81       STA  $81
20D8:  68          PLA
20D9:  85 80       STA  $80
20DB:  68          PLA
20DC:  A8          TAY
20DD:  68          PLA
20DE:  AA          TAX
20DF:  20 5E 11    JSR  $115E
20E2:  60          RTS
```

Program 7.17 Example 2 (02000 – #2039 and #20CO – #20E2)

This example (Program 7.17) demonstrates how easy it is to move any shape around, using the second drawing method. The flickering is due to the constant drawing and clearing of the object. One way round this would be to leave the object on the screen and not clear it off. Providing that your object has been defined with at least one blank pixel on all sides, you will automatically wipe out the previous creation when moving in any direction. Of course, this will clear anything that your object crosses, but the graphics animation is smoother and twice as fast as before.

The shape for Example 2 in the area #2040 to #206C is as follows:

7.8 PAINT subroutines

I wonder if you thought that the FILL command would paint an area of the screen when you first bought your Oric! Unfortunately there is no easy way to shade in anything more complicated than a rectangle, so here is the highlight of the graphics routines, a super-fast PAINT subroutine.

THEORY

The paint facility here uses the 1K graphics table, created by calling 1200, and the set and test dot subroutines at 137E and 1398. However, so that the routine can be called from BASIC, the subroutine has been designed to do all necessary calls, and saves all zero page locations that it overwrites.

The theory behind a PAINT subroutine assumes that the shape is completely enclosed and that a starting point is supplied somewhere inside. The general approach is to move away from this starting point, going in all directions in turn. From each new point that is not yet filled, another set of directions is remembered, and in this way the whole shape is eventually painted.

To 'remember' each point that needs painting, a stack is used, so that we explore all avenues until a complete dead-end is reached and then back-track through all other possibilities – like you might do when mapping a maze.

The problem with this is that you have only a limited stack to use. In order for PAINT to work within this constraint, it must constantly prune unwanted values off the stack. Even so, this PAINT routine is probably the fastest you will ever see on the Oric.

Rather than try to explain in detail how the machine code routine works., Program 7.18 is a BASIC equivalent to the PAINT subroutine listed below in Program 7.19.

```
5 REM BASIC VERSION OF PAINT
10 DIMA(100):S=100
15 INPUTX,Y
20 RF=0:S=S-1:A(S)=255:S=S-1:A(S)=255:GOTO35
30 Y=A(S):S=S+1:X=A(S):S=S+1
35 IFX=255THEN END
40 IFRF=0THENUF=TRUE:DF=TRUE
45 T=S:R=T
46 IFA(R)=255THEN50
47 IFA(R)=YANDA(R+1)=XTHENR=R-1:FORK=RTOTSTEP-1:A(K+2)=A(K):NEXT:S=
S+2:GOTO50
48 R=R+2:GOTO46
50 CURSETX,Y,1
60 IFUFANDPOINT(X,Y-1)=0THENS=S-1:A(S)=X:S=S-1:A(S)=Y-1
70 UF=POINT(X,Y-1)
80 IFDFANDPOINT(X,Y+1)=0THENS=S-1:A(S)=X:S=S-1:A(S)=Y+1
90 DF=POINT(X,Y+1)
100 RF=0:IFPOINT(X-1,Y)=0THENS=S-1:A(S)=X-1:S=S-1:A(S)=Y:RF=TRUE
120 IFPOINT(X+1,Y)=0THENS=S-1:A(S)=X+1:S=S-1:A(S)=Y:RF=TRUE
130 GOTO30
```

Program 7.18 BASIC paint program

This program runs very slowly because, for every point plotted, the routine must look in the four surrounding positions. Here is a summary of what is happening:

1. The flag RF (right flag) is set to true whenever it is possible to move either left or right.
2. The flags UF and DF (up and down flags) are set to the state of the pixels above and below the current dot position. Before doing this, the subroutine looks for an empty pixel above or below, and if the up or down flag is set as well, that position is put on the stack as a point to investigate. These flags are used in order to stop the stack from being saturated with unnecessary values. Since all dots along a line are investigated, it is not necessary to look at all the dots above and below, since any one of them will scan its own horizontal brothers.
3. As each point is set, the stack is examined for any outstanding references to that point, and these are removed.

Program 7.19 gives the listing of the machine code PAINT subroutine.

```
1000:  78          SEI
1001:  A2 0F       LDX  #$0F
1003:  B5 80       LDA  $80,X
```

Program 7.19 *(continues)*

```
1005:  9D E2 10    STA  $10E2,X
1008:  CA          DEX
1009:  10 F8       BPL  $1003
100B:  D8          CLD
100C:  20 00 12    JSR  $1200
100F:  A6 00       LDX  $00
1011:  A4 01       LDY  $01
1013:  A9 00       LDA  #$00
1015:  85 8C       STA  $8C
1017:  A9 FF       LDA  #$FF
1019:  48          PHA
101A:  48          PHA
101B:  D0 04       BNE  $1021
101D:  68          PLA
101E:  A8          TAY
101F:  68          PLA
1020:  AA          TAX
1021:  E0 FF       CPX  #$FF
1023:  F0 5F       BEQ  $1084
1025:  A5 8C       LDA  $8C
1027:  D0 06       BNE  $102F
1029:  A9 01       LDA  #$01
102B:  85 8D       STA  $8D
102D:  85 8E       STA  $8E
102F:  20 A2 10    JSR  $10A2
1032:  88          DEY
1033:  20 98 10    JSR  $1098
1036:  85 8F       STA  $8F
1038:  D0 08       BNE  $1042
103A:  A5 8D       LDA  $8D
103C:  F0 04       BEQ  $1042
103E:  8A          TXA
103F:  48          PHA
1040:  98          TYA
1041:  48          PHA
1042:  A5 8F       LDA  $8F
1044:  85 8D       STA  $8D
1046:  C8          INY
1047:  C8          INY
1048:  20 98 10    JSR  $1098
104B:  85 8F       STA  $8F
104D:  D0 08       BNE  $1057
104F:  A5 8E       LDA  $8E
1051:  F0 04       BEQ  $1057
1053:  8A          TXA
1054:  48          PHA
1055:  98          TYA
1056:  48          PHA
1057:  A5 8F       LDA  $8F
```

```
1059:  85 8E        STA  $8E
105B:  88           DEY
105C:  CA           DEX
105D:  20 98 10     JSR  $1098
1060:  D0 0A        BNE  $106C
1062:  8A           TXA
1063:  48           PHA
1064:  98           TYA
1065:  48           PHA
1066:  A9 01        LDA  #$01
1068:  85 8C        STA  $8C
106A:  D0 04        BNE  $,1070
106C:  A9 00        LDA  #$00
106E:  85 8C        STA  $8C
1070:  E8           INX
1071:  E8           INX
1072:  20 98 10     JSR  $1098
1075:  D0 08        BNE  $107F
1077:  8A           TXA
1078:  48           PHA
1079:  98           TYA
107A:  48           PHA
107B:  A9 01        LDA  #$01
107D:  85 8C        STA  $8C
107F:  CA           DEX
1080:  A9 00        LDA  #$00
1082:  F0 99        BEQ  $101D
1084:  A2 0F        LDX  #$0F
1086:  BD E2 10     LDA  $10E2,
1089:  95 80        STA  $80,X
108B:  CA           DEX
108C:  10 F8        BPL  $1086
108E:  58           CLI
108F:  60           RTS
1090:  84 80        STY  $80
1092:  20 7E 13     JSR  $137E
1095:  A4 80        LDY  $80
1097:  60           RTS
1098:  84 80        STY  $80
109A:  20 98 13     JSR  $1398
109D:  08           PHP
109E:  A4 80        LDY  $80
10A0:  28           PLP
10A1:  60           RTS
10A2:  86 82        STX  $82
10A4:  84 83        STY  $83
10A6:  BA           TSX
10A7:  86 8F        STX  $8F
10A9:  E8           INX
```

```
10AA:  E8          INX
10AB:  E8          INX
10AC:  BD 00 01    LDA  $0100,X
10AF:  C9 FF       CMP  #$FF
10B1:  F0 25       BEQ  $10D8
10B3:  C5 83       CMP  $83
10B5:  D0 1C       BNE  $10D3
10B7:  BD 01 01    LDA  $0101,X
10BA:  C5 82       CMP  $82
10BC:  D0 15       BNE  $10D3
10BE:  CA          DEX
10BF:  E4 8F       CPX  $8F
10C1:  F0 09       BEQ  $10CC
10C3:  BD 00 01    LDA  $0100,X
10C6:  9D 02 01    STA  $0102,X
10C9:  38          SEC
10CA:  B0 F2       BCS  $10BE
10CC:  E6 8F       INC  $8F
10CE:  E6 8F       INC  $8F
10D0:  38          SEC
10D1:  B0 05       BCS  $10D8
10D3:  E8          INX
10D4:  E8          INX
10D5:  38          SEC
10D6:  B0 D4       BCS  $10AC
10D8:  A6 8F       LDX  $8F
10DA:  9A          TXS
10DB:  A6 82       LDX  $82
10DD:  A4 83       LDY  $83
10DF:  4C 90 10    JMP  $1090
```

Program 7.19 PAINT subroutine (#1000–#10E1)

USING THE PAINT FACILITY

The routine is called at #1000 after setting X (at #0) and Y (at #1) to a point inside the shape.

It assumes that high-resolution mode has been selected, though of course this is not of importance to the program itself, since it is simply processing an area of memory.

PAINT is great fun to watch!

7.9 High-resolution compactor subroutine

When displaying a high-resolution screen, you would normally need nearly 8K of RAM. This may strike you as being wasteful, especially when most of the time you are just looking at blank or filled areas. Below is a routine that compacts a high-resolution screen (or any other data) into anything from 1K to 8K, according to the complexity of the picture. You do not have to save the whole of the screen in any case, so this routine can be used with the split-screen facility discussed in Chapter 6.

The compactor and the companion expander routine do not use any of the other subroutines in this chapter, so can be used on their own. One obvious use is the creation of front pages when loading a program: instead of loading a complete 8K picture, you only need to load maybe 3 or 4K.

COMPACTOR CONSIDERATIONS

These routines make two assumptions:

1. The value #0F is not used. (It has no meaning in HIRES mode anyway.)

2. Most of the screen contains characters with an ASCII value between 0 and 127. The routine does cope with the occasional ‘inverse’ byte, though with less efficiency than ‘normal’ characters.

USING THE COMPACTOR

You need to set up start and end addresses as follows:

1. The start address of the area to be condensed should be entered at #82, #88.

2. The end address of this area should be stored at #84, #85.

3. The start address of the resultant data must be stored at #86, #87.

After using the compactor subroutine, the highest address of the compacted data is left at #88, #89.

The compactor subroutine is intended for use within a machine code program. If you wish to use it from BASIC you will have to save locations #80 to #8F in the same way as the PAINT routine does.

The expander routine, which reverses the compacting process, uses the same addresses, but does not need the end address (#82) to be specified.

```
0C00:  A5 82     LDA  $82
0C02:  48        PHA
0C03:  A5 83     LDA  $83
```

Program 7.20 *(continues)*

```
0C05:  48        PHA
0C06:  A5 86     LDA  $86
0C08:  85 88     STA  $88
0C0A:  A5 87     LDA  $87
0C0C:  85 89     STA  $89
0C0E:  A0 01     LDY  #$01
0C10:  B1 84     LDA  ($84),Y
0C12:  48        PHA
0C13:  A9 0F     LDA  #$0F
0C15:  91 84     STA  ($84),Y
0C17:  A0 00     LDY  #$00
0C19:  B1 82     LDA  ($82),Y
0C1B:  85 8A     STA  $8A
0C1D:  C8        INY
0C1E:  C0 7F     CPY  #$7F
0C20:  F0 06     BEQ  $0C28
0C22:  B1 82     LDA  ($82),Y
0C24:  C5 8A     CMP  $8A
0C26:  F0 F5     BEQ  $0C1D
0C28:  A5 8A     LDA  $8A
0C2A:  30 0E     BMI  $0C3A
0C2C:  C0 01     CPY  #$01
0C2E:  D0 0A     BNE  $0C3A
0C30:  A5 8A     LDA  $8A
0C32:  A2 00     LDX  #$00
0C34:  81 88     STA  ($88,X)
0C36:  A9 01     LDA  #$01
0C38:  D0 11     BNE  $0C4B
0C3A:  98        TYA
0C3B:  AA        TAX
0C3C:  09 80     ORA  #$80
0C3E:  A0 00     LDY  #$00
0C40:  91 88     STA  ($88),Y
0C42:  C8        INY
0C43:  A5 8A     LDA  $8A
0C45:  91 88     STA  ($88),Y
0C47:  8A        TXA
0C48:  A8        TAY
0C49:  A9 02     LDA  #$02
0C4B:  18        CLC
0C4C:  65 88     ADC  $88
0C4E:  85 88     STA  $88
0C50:  90 02     BCC  $0C54
0C52:  E6 89     INC  $89
0C54:  18        CLC
0C55:  98        TYA
0C56:  65 82     ADC  $82
0C58:  85 82     STA  $82
0C5A:  90 02     BCC  $0C5E
```

```
0C5C:  E6 83     INC  $83
0C5E:  A5 8A     LDA  $8A
0C60:  C9 0F     CMP  #$0F
0C62:  D0 B3     BNE  $0C17
0C64:  68        PLA
0C65:  A0 01     LDY  #$01
0C67:  91 84     STA  ($84),Y
0C69:  68        PLA
0C6A:  85 83     STA  $83
0C6C:  68        PLA
0C6D:  85 82     STA  $82
0C6F:  60        RTS
```

Program 7.20 Compactor routine (#C00–#C6F)

Program 7.20 gives the compactor subroutine, which starts at #C00

Finally, Program 7.21 gives the expander routine, at #C75.

```
0C75:  A5 82     LDA  $82
0C77:  85 84     STA  $84
0C79:  A5 83     LDA  $83
0C7B:  85 85     STA  $85
0C7D:  A5 86     LDA  $86
0C7F:  48        PHA
0C80:  A5 87     LDA  $87
0C82:  48        PHA
0C83:  A0 00     LDY  #$00
0C85:  B1 86     LDA  ($86),Y
0C87:  30 06     BMI  $0C8F
0C89:  91 84     STA  ($84),Y
0C8B:  A9 01     LDA  #$01
0C8D:  D0 15     BNE  $0CA4
0C8F:  29 7F     AND  #$7F
0C91:  AA        TAX
0C92:  C8        INY
0C93:  B1 86     LDA  ($86),Y
0C95:  48        PHA
0C96:  8A        TXA
0C97:  A8        TAY
0C98:  68        PLA
0C99:  88        DEY
0C9A:  91 84     STA  ($84),Y
0C9C:  88        DEY
```

Program 7.21 *(continues)*

```
0C9D:  10 FB     BPL  $0C9A
0C9F:  8A        TXA
0CA0:  A8        TAY
0CA1:  88        DEY
0CA2:  A9 02     LDA  #$02
0CA4:  18        CLC
0CA5:  65 86     ADC  $86
0CA7:  85 86     STA  $86
0CA9:  90 02     BCC  $0CAD
0CAB:  E6 87     INC  $87
0CAD:  38        SEC
0CAE:  98        TYA
0CAF:  65 84     ADC  $84
0CB1:  85 84     STA  $84
0CB3:  90 02     BCC  $0CB7
0CB5:  E6 85     INC  $85
0CB7:  A0 00     LDY  #$00
0CB9:  B1 86     LDA  ($86),Y
0CBB:  C9 0F     CMP  #$0F
0CBD:  D0 C6     BNE  $0C85
0CBF:  68        PLA
0CC0:  85 87     STA  $87
0CC2:  68        PLA
0CC3:  85 86     STA  $86
0CC5:  60        RTS
```

Program 7.21 Expander routine (#C75–#CC5)

7.10 Conclusions

It is not intended that you use all of these routines every time that you want to do some high-resolution graphics – indeed the ROM routines may prove to be fast enough for your needs. It may well be, though, that you require several of the routines for a game involving fast-moving graphics, and these could be relocated into a spare memory address using the relocater program in Chapter 3.

Even if you do not want to use the movement routines, the paint facility is invaluable in BASIC, though once again you may need to relocate it to a higher address if you have a 48K machine.

The use of tables to speed up graphics should be remembered for other applications where memory can be sacrificed in return for a faster response.

The compactor routine can certainly be improved upon. It depends largely on the picture being stored, but you may find it better to scan vertically rather than horizontally. A completely different approach would be to analyse the screen in terms of pixels, keeping a count of the number of dots alternately set or clear.

Additional note (August 1998)

A lot of water has passed under the bridge since this was written,15 years of programming, including much games programming on other machines has taught me that some of the methods in this chapter aren't quite optimal. At the time, I clearly had only a passing knowledge of TV's workings, i.e. raster and flybacks. I don't think I would really use XOR these days, far better to restore previous sprite, save background, draw sprite. Having said that, I still come across the XOR trick, for instance in a C++ MFC book, which suggested its use for drawing lines and boxes. As for the suggestion that multiplying by 40 is hard, well in fact it's just a a few shifts and adds.

8 USEFUL UTILITIES

8.1 Introduction

This chapter presents six utilities to help you write programs in BASIC. You may have seen other versions of some of the routines (such as Delete and Renumber), but the routines here are generally shorter and faster.

The programs can be entered by using a machine code monitor (such as Tansoft's ORICMON program) or by using a simple BASIC loader, as described in Chapter 3. Once the machine code is in memory, you should save it on tape so that it can be loaded independently of any other program.

Chapter 3 contains a relocater routine should you need to move the programs to a different address.

Some of the programs are considerably dependent on the ROM, and often you will find two listings printed – one per version of ROM. Where there are only a small number of differences, a listing is given for version 1.0, with the changes that are required for version 1.1.

8.2 Renumber routine

This is quite a lengthy program, occupying about 600 bytes. Its purpose is to resequence a BASIC program so that the line numbers increase in even steps. This is very useful when you need to insert new lines into a program.

The utility is located in the alternate character set area, between #B800 and #BA5C. Remember that pressing the Reset button will wipe out the program!

To renumber your program you must first DOKE 0 with the starting line number and then DOKE 2 with the increment. Finally, you should CALL #BA1E in order to start the process. For large programs, prepare to wait a couple of minutes. For example, DOKE 0,10:DOKE 2,10:CALL#BA1E would renumber the program starting at 10, in steps of 10.

ALL GOTO, GOSUB, THEN, and ELSE statements are converted to fit in with the new steps.

Version 1.0 owners should note that the Renumber program can be loaded after the BASIC program to be renumbered, since the #9C end-of-BASIC pointer is corrected.

HOW IT WORKS

Renumber is the most complicated program in this book. The theory is as follows:

1. Use the two link bytes to store the new line numbers. The old numbers must be kept for cross-reference purposes.
2. Go through the program looking for 'GOTO', 'GOSUB', 'THEN', or 'ELSE'.
3. When one of these tokens is found, look up the line number that follows and replace it with the new line number stored in the link field.
4. At the end of the program, recalculate the links.

CHANGES FOR VERSION 1.1

The ROM is called in eight places; however, for version 1.1 these addresses are different:

```
#B94E:      JSR   (input a number into ACC1 from ASCII)
DFE7     #B951:   (turn ACC1 into an integer)
JSR         D92C  (turn integer into floating point)
#B968: JSR  D499  (create program links)
#BA1A:      JSR   (disable interrupts)
C55F     #BA1E:   (enable interrupts)
JSR         E76A  (add memory to ACC1)
```

#BA30: JSR (convert ACC1 into ASCII)
E93D #BA44:
JSR DB22
#BA47: JSR
E0D5

PROGRAM LISTING

The program (8.1) needs this short table: #BA58: #91, #00, #00, #00, #00.

This is the floating-point representation of 65536 which is used to handle line numbers beyond 32767.

```
B800:  A2 00     LDX  #$00
B802:  A5 35     LDA  $35
B804:  48        PHA
B805:  A5 36     LDA  $36
B807:  48        PHA
B808:  A0 00     LDY  #$00
B80A:  B1 35     LDA  ($35),Y
```

```
B80C:  C9 3A      CMP  #$3A
B80E:  B0 07      BCS  $B817
B810:  C9 30      CMP  #$30
B812:  90 03      BCC  $B817
B814:  C8         INY
B815:  D0 F3      BNE  $B80A
B817:  B1 35      LDA  ($35),Y
B819:  81 35      STA  ($35,X)
B81B:  A5 9C      LDA  $9C
B81D:  C5 35      CMP  $35
B81F:  D0 06      BNE  $B827
B821:  A5 9D      LDA  $9D
B823:  C5 36      CMP  $36
B825:  F0 08      BEQ  $B82F
B827:  E6 35      INC  $35
B829:  D0 02      BNE  $B82D
B82B:  E6 36      INC  $36
B82D:  D0 E8      BNE  $B817
B82F:  84 37      STY  $37
B831:  38         SEC
B832:  A5 9C      LDA  $9C
B834:  E5 37      SBC  $37
B836:  85 9C      STA  $9C
B838:  A5 9D      LDA  $9D
B83A:  E9 00      SBC  #$00
B83C:  85 9D      STA  $9D
B83E:  68         PLA
B83F:  85 36      STA  $36
B841:  68         PLA
B842:  85 35      STA  $35
B844:  60         RTS
B845:  EA         NOP
B846:  EA         NOP
B847:  EA         NOP
B848:  A0 00      LDY  #$00
B84A:  B9 01 01   LDA  $0101,Y
B84D:  F0 03      BEQ  $B852
B84F:  C8         INY
B850:  D0 F8      BNE  $B84A
B852:  A5 9C      LDA  $9C
B854:  85 38      STA  $38
B856:  A5 9D      LDA  $9D
B858:  85 39      STA  $39
B85A:  A2 00      LDX  #$00
B85C:  A1 38      LDA  ($38,X)
B85E:  91 38      STA  ($38),Y
B860:  A5 38      LDA  $38
B862:  C5 35      CMP  $35
B864:  D0 06      BNE  $B86C
```

```
B866:  A5 39        LDA  $39
B868:  C5 36        CMP  $36
B86A:  F0 0F        BEQ  $B87B
B86C:  38           SEC
B86D:  A5 38        LDA  $38
B86F:  E9 01        SBC  #$01
B871:  85 38        STA  $38
B873:  A5 39        LDA  $39
B875:  E9 00        SBC  #$00
B877:  85 39        STA  $39
B879:  D0 E1        BNE  $B85C
B87B:  84 37        STY  $37
B87D:  88           DEY
B87E:  30 08        BMI  $B888
B880:  B9 01 01     LDA  $0101,Y
B883:  91 38        STA  ($38),Y
B885:  88           DEY
B886:  10 F8        BPL  $B880
B888:  18           CLC
B889:  A5 37        LDA  $37
B88B:  65 9C        ADC  $9C
B88D:  85 9C        STA  $9C
B88F:  A5 9D        LDA  $9D
B891:  69 00        ADC  #$00
B893:  85 9D        STA  $9D
B895:  60           RTS
B896:  D8           CLD
B897:  A9 01        LDA  #$01
B899:  85 35        STA  $35
B89B:  A9 05        LDA  #$05
B89D:  85 36        STA  $36
B89F:  A0 01        LDY  #$01
B8A1:  B1 35        LDA  ($35),Y
B8A3:  C9 FF        CMP  #$FF
B8A5:  F0 32        BEQ  $B8D9
B8A7:  A0 02        LDY  #$02
B8A9:  B1 35        LDA  ($35),Y
B8AB:  C5 D4        CMP  $D4
B8AD:  D0 07        BNE  $B8B6
B8AF:  C8           INY
B8B0:  B1 35        LDA  ($35),Y
B8B2:  C5 D3        CMP  $D3
B8B4:  F0 18        BEQ  $B8CE
B8B6:  A0 04        LDY  #$04
B8B8:  B1 35        LDA  ($35),Y
B8BA:  F0 03        BEQ  $B8BF
B8BC:  C8           INY
B8BD:  D0 F9        BNE  $B8B8
B8BF:  C8           INY
```

```
B8C0:  98        TYA
B8C1:  18        CLC
B8C2:  65 35     ADC  $35
B8C4:  85 35     STA  $35
B8C6:  A5 36     LDA  $36
B8C8:  69 00     ADC  #$00
B8CA:  85 36     STA  $36
B8CC:  D0 D1     BNE  $B89F
B8CE:  A0 00     LDY  #$00
B8D0:  B1 35     LDA  ($35),Y
B8D2:  85 D3     STA  $D3
B8D4:  C8        INY
B8D5:  B1 35     LDA  ($35),Y
B8D7:  85 D4     STA  $D4
B8D9:  60        RTS
B8DA:  A9 01     LDA  #$01
B8DC:  85 35     STA  $35
B8DE:  A9 05     LDA  #$05
B8E0:  85 36     STA  $36
B8E2:  A0 01     LDY  #$01
B8E4:  B1 35     LDA  ($35),Y
B8E6:  F0 22     BEQ  $B90A
B8E8:  48        PHA
B8E9:  A5 01     LDA  $01
B8EB:  91 35     STA  ($35),Y
B8ED:  88        DEY
B8EE:  B1 35     LDA  ($35),Y
B8F0:  48        PHA
B8F1:  A5 00     LDA  $00
B8F3:  91 35     STA  ($35),Y
B8F5:  18        CLC
B8F6:  A5 00     LDA  $00
B8F8:  65 02     ADC  $02
B8FA:  85 00     STA  $00
B8FC:  A5 01     LDA  $01
B8FE:  65 03     ADC  $03
B900:  85 01     STA  $01
B902:  68        PLA
B903:  85 35     STA  $35
B905:  68        PLA
B906:  85 36     STA  $36
B908:  D0 D8     BNE  $B8E2
B90A:  A9 FF     LDA  #$FF
B90C:  91 35     STA  ($35),Y
B90E:  18        CLC
B90F:  A5 35     LDA  $35
B911:  69 02     ADC  #$02
B913:  85 9C     STA  $9C
B915:  A5 36     LDA  $36
```

```
B917:  69 00      ADC  #$00
B919:  85 9D      STA  $9D
B91B:  60         RTS
B91C:  20 34 BA   JSR  $BA34
B91F:  18         CLC
B920:  98         TYA
B921:  65 3A      ADC  $3A
B923:  85 E9      STA  $E9
B925:  A5 3B      LDA  $3B
B927:  69 00      ADC  #$00
B929:  85 EA      STA  $EA
B92B:  98         TYA
B92C:  48         PHA
B92D:  A0 00      LDY  #$00
B92F:  B1 E9      LDA  ($E9),Y
B931:  C9 3A      CMP  #$3A
B933:  B0 07      BCS  $B93C
B935:  C9 30      CMP  #$30
B937:  90 03      BCC  $B93C
B939:  C8         INY
B93A:  D0 F3      BNE  $B92F
B93C:  B1 E9      LDA  ($E9),Y
B93E:  48         PHA
B93F:  EA         NOP
B940:  EA         NOP
B941:  EA         NOP
B942:  EA         NOP
B943:  A5 E9      LDA  $E9
B945:  48         PHA
B946:  A5 EA      LDA  $EA
B948:  48         PHA
B949:  98         TYA
B94A:  48         PHA
B94B:  20 E8 00   JSR  $00E8
B94E:  20 CF DF   JSR  $DFCF
B951:  20 71 D8   JSR  $D871
B954:  20 96 B8   JSR  $B896
B957:  A5 D3      LDA  $D3
B959:  05 D4      ORA  $D4
B95B:  D0 07      BNE  $B964
B95D:  A9 00      LDA  #$00
B95F:  8D 01 01   STA  $0101
B962:  F0 0A      BEQ  $B96E
B964:  A4 D3      LDY  $D3
B966:  A5 D4      LDA  $D4
B968:  20 ED D3   JSR  $D3ED
B96B:  20 3C BA   JSR  $BA3C
B96E:  68         PLA
B96F:  A8         TAY
```

```
B970:  68          PLA
B971:  85 EA       STA  $EA
B973:  68          PLA
B974:  85 E9       STA  $E9
B976:  68          PLA
B977:  91 E9       STA  ($E9),Y
B979:  A5 E9       LDA  $E9
B97B:  85 35       STA  $35
B97D:  A5 EA       LDA  $EA
B97F:  85 36       STA  $36
B981:  20 00 B8    JSR  $B800
B984:  20 48 B8    JSR  $B848
B987:  68          PLA
B988:  A8          TAY
B989:  B1 3A       LDA  ($3A),Y
B98B:  C9 3A       CMP  #$3A
B98D:  B0 07       BCS  $B996
B98F:  C9 30       CMP  #$30
B991:  90 03       BCC  $B996
B993:  C8          INY
B994:  D0 F3       BNE  $B989
B996:  B1 3A       LDA  ($3A),Y
B998:  C9 2C       CMP  #$2C
B99A:  F0 80       BEQ  $B91C
B99C:  60          RTS
B99D:  EA          NOP
B99E:  EA          NOP
B99F:  D8          CLD
B9A0:  20 DA B8    JSR  $B8DA
B9A3:  A9 01       LDA  #$01
B9A5:  85 3A       STA  $3A
B9A7:  A9 05       LDA  #$05
B9A9:  85 3B       STA  $3B
B9AB:  A0 01       LDY  #$01
B9AD:  B1 3A       LDA  ($3A),Y
B9AF:  C9 FF       CMP  #$FF
B9B1:  F0 2C       BEQ  $B9DF
B9B3:  A0 04       LDY  #$04
B9B5:  B1 3A       LDA  ($3A),Y
B9B7:  F0 18       BEQ  $B9D1
B9B9:  C9 97       CMP  #$97
B9BB:  F0 0C       BEQ  $B9C9
B9BD:  C9 C9       CMP  #$C9
B9BF:  F0 08       BEQ  $B9C9
B9C1:  C9 C8       CMP  #$C8
B9C3:  F0 04       BEQ  $B9C9
B9C5:  C9 9B       CMP  #$9B
B9C7:  D0 05       BNE  $B9CE
B9C9:  20 1C B9    JSR  $B91C
```

```
B9CC:  D0 E7      BNE  $B9B5
B9CE:  C8         INY
B9CF:  D0 E4      BNE  $B9B5
B9D1:  C8         INY
B9D2:  98         TYA
B9D3:  18         CLC
B9D4:  65 3A      ADC  $3A
B9D6:  85 3A      STA  $3A
B9D8:  90 02      BCC  $B9DC
B9DA:  E6 3B      INC  $3B
B9DC:  38         SEC
B9DD:  B0 CC      BCS  $B9AB
B9DF:  A9 01      LDA  #$01
B9E1:  85 35      STA  $35
B9E3:  A9 05      LDA  #$05
B9E5:  85 36      STA  $36
B9E7:  A0 01      LDY  #$01
B9E9:  B1 35      LDA  ($35),Y
B9EB:  C9 FF      CMP  #$FF
B9ED:  F0 27      BEQ  $BA16
B9EF:  A0 03      LDY  #$03
B9F1:  91 35      STA  ($35),Y
B9F3:  A0 00      LDY  #$00
B9F5:  B1 35      LDA  ($35),Y
B9F7:  A0 02      LDY  #$02
B9F9:  91 35      STA  ($35),Y
B9FB:  A0 01      LDY  #$01
B9FD:  98         TYA
B9FE:  91 35      STA  ($35),Y
BA00:  A0 04      LDY  #$04
BA02:  B1 35      LDA  ($35),Y
BA04:  F0 03      BEQ  $BA09
BA06:  C8         INY
BA07:  D0 F9      BNE  $BA02
BA09:  98         TYA
BA0A:  38         SEC
BA0B:  65 35      ADC  $35
BA0D:  85 35      STA  $35
BA0F:  90 02      BCC  $BA13
BA11:  E6 36      INC  $36
BA13:  38         SEC
BA14:  B0 D1      BCS  $B9E7
BA16:  A9 00      LDA  #$00
BA18:  91 35      STA  ($35),Y
BA1A:  20 6F C5   JSR  $C56F
BA1D:  60         RTS
BA1E:  20 CA E6   JSR  $E6CA
BA21:  A5 E9      LDA  $E9
BA23:  48         PHA
```

```
BA24:  A5 EA      LDA  $EA
BA26:  48         PHA
BA27:  20 9F B9   JSR  $B99F
BA2A:  68         PLA
BA2B:  85 EA      STA  $EA
BA2D:  68         PLA
BA2E:  85 E9      STA  $E9
BA30:  20 04 E8   JSR  $E804
BA33:  60         RTS
BA34:  C8         INY
BA35:  B1 3A      LDÀ  ($3A),Y
BA37:  C9 20      CMP  #$20
BA39:  F0 F9      BEQ  $BA34
BA3B:  60         RTS
BA3C:  A5 D5      LDA  $D5
BA3E:  10 07      BPL  $BA47
BA40:  A9 58      LDA  #$58
BA42:  A0 BA      LDY  #$BA
BA44:  20 97 DA   JSR  $DA97
BA47:  20 D1 E0   JSR  $E0D1
BA4A:  60         RTS
```

8.3 Delete utility

It is often useful to be able to extract part of a program, but normally this would involve much typing in order to remove the unwanted lines. Here is a short routine that will delete any given section of a program.

To run the program, DOKE 0 with the lowest line number and DOKE 2 with the highest line number. When you are ready to delete part of the program, type CALL#420. For example, DOKE0,100: DOKE2,200:CALL#420 would delete lines 100 to 200 (inclusive).

Owners of version 1.0 ROMs should load the DELETE program before loading the BASIC program that is to be modified, otherwise the #9C end-of-BASIC pointer will be incorrect.

Version 1.1 ROM owners must make these three changes to the listed routine:

#429: JSR C6B9 (find address of a given line number)

#441: JSR C6B9

#462: JSR C55F (create program links)

HOW THE DELETE UTILITY WORKS

This program first finds the address of the earliest line to delete, storing it at 0,1.

Then it finds the address of the line following the section that is to be deleted. It is then a simple matter to move down the program, from the second address to the first. Finally, the program is re-linked, and the #9C end-of-BASIC pointer corrected.

PROGRAM LISTING

This is given in Program 8.2.

```
0420:  D8          CLD
0421:  A5 00       LDA  $00
0423:  85 33       STA  $33
0425:  A5 01       LDA  $01
0427:  85 34       STA  $34
0429:  20 E4 C6    JSR  $C6E4
042C:  A5 CE       LDA  $CE
042E:  85 00       STA  $00
0430:  A5 CF       LDA  $CF
0432:  85 01       STA  $01
0434:  18          CLC
0435:  A5 02       LDA  $02
0437:  69 01       ADC  #$01
0439:  85 33       STA  $33
043B:  A5 03       LDA  $03
043D:  69 00       ADC  #$00
043F:  85 34       STA  $34
0441:  20 E4 C6    JSR  $C6E4
0444:  A0 00       LDY  #$00
0446:  B1 CE       LDA  ($CE),Y
0448:  91 00       STA  ($00),Y
044A:  E6 00       INC  $00
044C:  D0 02       BNE  $0450
044E:  E6 01       INC  $01
0450:  E6 CE       INC  $CE
0452:  D0 02       BNE  $0456
0454:  E6 CF       INC  $CF
0456:  A5 CE       LDA  $CE
```

```
0458:  C5 9C      CMP  $9C
045A:  D0 EA      BNE  $0446
045C:  A5 CF      LDA  $CF
045E:  C5 9D      CMP  $9D
0460:  D0 E4      BNE  $0446
0462:  20 6F C5   JSR  $C56F
0465:  18         CLC
0466:  A5 91      LDA  $91
0468:  69 02      ADC  #$02
046A:  85 9C      STA  $9C
046C:  85 9E      STA  $9E
046E:  A5 92      LDA  $92
0470:  69 00      ADC  #$00
0472:  85 9D      STA  $9D
0474:  85 9F      STA  $9F
0476:  60         RTS
0477:  EA         NOP
```

Program 8.2 Delete

8.4 Merge program facility

This is an invaluable routine, often used in connection with the previous two subroutines when copying parts of BASIC programs around.

This program is much more sophisticated than the 'join' facility of version 1.1 ROMs, as it can interleave two programs correctly and also replace duplicated lines.

To use Merge:

1. Load up the Merge machine code routine.

2. Load up the first BASIC program.

3. Type in CALL #B8AE.

4. Play back the tape containing the new section to be introduced. Any lines with the same line number will act as replacements.

If the tape speed of the first BASIC program is different from the second, you will need to alter the tape-speed flag. This is either 0 fast) or 1 (slow) and is stored at #67 (version 1.0) or #24D (version 1.1).

The Merge routine will take a maximum of 3 minutes to complete, depending on the size of the first program.

There must be room in memory to store both the original program andthe program that is merged.

HOW MERGE WORKS

It is not possible to insert lines into a program directly as they arrive from tape – there is not enough time between bytes. The method used here is to move the existing program up to the end of memory and then load in the new lines as a normal program. Then the Merge routine can input each line of the program stored at the end of memory into the correct place. When this process is finished, the #9C end-of-BASIC pointer is recalculated and the program is re-linked.

PROGRAM LISTINGS

Because the tape handling routines are greatly different between the two ROM versions, there are two program listings (Programs 8.3 and 8.4).

```
B800:  A9 05     LDA  #$05
B802:  85 38     STA  $38
B804:  A9 01     LDA  #$01
B806:  85 37     STA  $37
B808:  A0 03     LDY  #$03
B80A:  B1 37     LDA  ($37),Y
B80C:  C5 36     CMP  $36
B80E:  F0 24     BEQ  $B834
B810:  B0 2F     BCS  $B841
B812:  A0 03     LDY  #$03
B814:  C8        INY
B815:  B1 37     LDA  ($37),Y
B817:  D0 FB     BNE  $B814
B819:  C8        INY
B81A:  98        TYA
B81B:  18        CLC
B81C:  65 37     ADC  $37
B81E:  85 37     STA  $37
B820:  90 02     BCC  $B824
B822:  E6 38     INC  $38
B824:  A0 00     LDY  #$00
B826:  B1 37     LDA  ($37),Y
B828:  D0 DE     BNE  $B808
B82A:  C8        INY
B82B:  B1 37     LDA  ($37),Y
B82D:  D0 D9     BNE  $B808
B82F:  18        CLC
B830:  60        RTS
B831:  EA        NOP
B832:  EA        NOP
B833:  EA        NOP
```

```
B834:  88        DEY
B835:  B1 37     LDA  ($37),Y
B837:  C5 35     CMP  $35
B839:  F0 04     BEQ  $B83F
B83B:  B0 04     BCS  $B841
B83D:  90 D3     BCC  $B812
B83F:  38        SEC
B840:  60        RTS
B841:  18        CLC
B842:  60        RTS
B843:  EA        NOP
B844:  EA        NOP
B845:  EA        NOP
B846:  EA        NOP
B847:  EA        NOP
B848:  A2 00     LDX  #$00
B84A:  A5 9C     LDA  $9C
B84C:  85 39     STA  $39
B84E:  A5 9D     LDA  $9D
B850:  85 3A     STA  $3A
B852:  A1 39     LDA  ($39,X)
B854:  91 39     STA  ($39),Y
B856:  A5 39     LDA  $39
B858:  C5 37     CMP  $37
B85A:  D0 06     BNE  $B862
B85C:  A5 3A     LDA  $3A
B85E:  C5 38     CMP  $38
B860:  F0 0F     BEQ  $B871
B862:  38        SEC
B863:  A5 39     LDA  $39
B865:  E9 01     SBC  #$01
B867:  85 39     STA  $39
B869:  A5 3A     LDA  $3A
B86B:  E9 00     SBC  #$00
B86D:  85 3A     STA  $3A
B86F:  D0 E1     BNE  $B852
B871:  98        TYA
B872:  18        CLC
B873:  65 9C     ADC  $9C
B875:  85 9C     STA  $9C
B877:  A5 9D     LDA  $9D
B879:  69 00     ADC  #$00
B87B:  85 9D     STA  $9D
B87D:  60        RTS
B87E:  EA        NOP
B87F:  EA        NOP
B880:  08        PHP
B881:  A0 04     LDY  #$04
B883:  B1 00     LDA  ($00),Y
```

```
B885:  F0 03        BEQ  $B88A
B887:  C8           INY
B888:  D0 F9        BNE  $B883
B88A:  C8           INY
B88B:  28           PLP
B88C:  98           TYA
B88D:  B0 0E        BCS  $B89D
B88F:  20 48 B8     JSR  $B848
B892:  98           TYA
B893:  48           PHA
B894:  88           DEY
B895:  B1 00        LDA  ($00),Y
B897:  91 37        STA  ($37),Y
B899:  88           DEY
B89A:  10 F9        BPL  $B895
B89C:  68           PLA
B89D:  18           CLC
B89E:  65 00        ADC  $00
B8A0:  85 00        STA  $00
B8A2:  A5 01        LDA  $01
B8A4:  69 00        ADC  #$00
B8A6:  85 01        STA  $01
B8A8:  60           RTS
B8A9:  EA           NOP
B8AA:  EA           NOP
B8AB:  EA           NOP
B8AC:  EA           NOP
B8AD:  EA           NOP
B8AE:  38           SEC
B8AF:  A5 9C        LDA  $9C
B8B1:  E9 04        SBC  #$04
B8B3:  85 35        STA  $35
B8B5:  A5 9D        LDA  $9D
B8B7:  E9 05        SBC  #$05
B8B9:  85 36        STA  $36
B8BB:  A9 FF        LDA  #$FF
B8BD:  E5 35        SBC  $35
B8BF:  85 00        STA  $00
B8C1:  48           PHA
B8C2:  A9 B3        LDA  #$B3
B8C4:  E5 36        SBC  $36
B8C6:  85 01        STA  $01
B8C8:  48           PHA
B8C9:  A9 01        LDA  #$01
B8CB:  85 02        STA  $02
B8CD:  A9 05        LDA  #$05
B8CF:  85 03        STA  $03
B8D1:  A0 00        LDY  #$00
B8D3:  B1 02        LDA  ($02),Y
```

```
B8D5:  91 00       STA  ($00),Y
B8D7:  A5 01       LDA  $01
B8D9:  C9 B4       CMP  #$B4
B8DB:  F0 10       BEQ  $B8ED
B8DD:  E6 00       INC  $00
B8DF:  D0 02       BNE  $B8E3
B8E1:  E6 01       INC  $01
B8E3:  E6 02       INC  $02
B8E5:  D0 02       BNE  $B8E9
B8E7:  E6 03       INC  $03
B8E9:  A9 00       LDA  #$00
B8EB:  F0 E6       BEQ  $B8D3
B8ED:  A9 00       LDA  #$00
B8EF:  85 35       STA  $35
B8F1:  EA          NOP
B8F2:  EA          NOP
B8F3:  EA          NOP
B8F4:  EA          NOP
B8F5:  EA          NOP
B8F6:  EA          NOP
B8F7:  EA          NOP
B8F8:  20 CA E6    JSR  $E6CA
B8FB:  20 A8 E4    JSR  $E4A8
B8FE:  20 04 E8    JSR  $E804
B901:  EA          NOP
B902:  EA          NOP
B903:  6A          ROR
B904:  EA          NOP
B905:  EA          NOP
B906:  EA          NOP
B907:  EA          NOP
B908:  EA          NOP
B909:  EA          NOP
B90A:  A5 61       LDA  $61
B90C:  EA          NOP
B90D:  85 9C       STA  $9C
B90F:  A5 62       LDA  $62
B911:  EA          NOP
B912:  85 9D       STA  $9D
B914:  20 41 B9    JSR  $B941
B917:  68          PLA
B918:  85 01       STA  $01
B91A:  68          PLA
B91B:  85 00       STA  $00
B91D:  A5 01       LDA  $01
B91F:  C9 B4       CMP  #$B4
B921:  F0 17       BEQ  $B93A
```

B923: EA

B924: EA

```
B925:  A0 02       LDY  #$02
B927:  B1 00       LDA  ($00),Y
B929:  85 35       STA  $35
B92B:  C8          INY
B92C:  B1 00       LDA  ($00),Y
B92E:  85 36       STA  $36
B930:  20 00 B8    JSR  $B800
B933:  20 80 B8    JSR  $B880
B936:  A9 00       LDA  #$00
B938:  F0 E3       BEQ  $B91D
B93A:  20 6F C5    JSR  $C56F
B93D:  20 FA FA    JSR  $FAFA
B940:  60          RTS
B941:  38          SEC
B942:  A5 9C       LDA  $9C
B944:  E9 02       SBC  #$02
B946:  85 02       STA  $02
B948:  A5 9D       LDA  $9D
B94A:  E9 00       SBC  #$00
B94C:  85 03       STA  $03
B94E:  A0 00       LDY  #$00
B950:  A9 00       LDA  #$00
B952:  91 02       STA  ($02),Y
B954:  C8          INY
B955:  91 02       STA  ($02),Y
B957:  60          RTS
B958:  EA          NOP
B959:  EA          NOP
B95A:  EA          NOP
B95B:  EA          NOP
B95C:  EA          NOP
B95D:  EA          NOP
```

Program 8.3 Merge (version 1.0 ROMs)

```
B800:  A9 05     LDA  #$05
B802:  85 38     STA  $38
B804:  A9 01     LDA  #$01
B806:  85 37     STA  $37
B808:  A0 03     LDY  #$03
B80A:  B1 37     LDA  ($37),Y
B80C:  C5 36     CMP  $36
B80E:  F0 24     BEQ  $B834
B810:  B0 2F     BCS  $B841
B812:  A0 03     LDY  #$03
B814:  C8        INY
B815:  B1 37     LDA  ($37),Y
B817:  D0 FB     BNE  $B814
B819:  C8        INY
```

```
B81A:  98        TYA
B81B:  18        CLC
B81C:  65 37     ADC  $37
B81E:  85 37     STA  $37
B820:  90 02     BCC  $B824
B822:  E6 38     INC  $38
B824:  A0 00     LDY  #$00
B826:  B1 37     LDA  ($37),Y
B828:  D0 DE     BNE  $B808
B82A:  C8        INY
B82B:  B1 37     LDA  ($37),Y
B82D:  D0 D9     BNE  $B808
B82F:  18        CLC
B830:  60        RTS
B831:  EA        NOP
B832:  EA        NOP
B833:  EA        NOP
B834:  88        DEY
B835:  B1 37     LDA  ($37),Y
B837:  C5 35     CMP  $35
B839:  F0 04     BEQ  $B83F
B83B:  B0 04     BCS  $B841
B83D:  90 D3     BCC  $B812
B83F:  38        SEC
B840:  60        RTS
B841:  18        CLC
B842:  60        RTS
B843:  EA        NOP
B844:  EA        NOP
B845:  EA        NOP
B846:  EA        NOP
B847:  EA        NOP
B848:  A2 00     LDX  #$00
B84A:  A5 9C     LDA  $9C
B84C:  85 39     STA  $39
B84E:  A5 9D     LDA  $9D
B850:  85 3A     STA  $3A
B852:  A1 39     LDA  ($39,X)
B854:  91 39     STA  ($39),Y
B856:  A5 39     LDA  $39
B858:  C5 37     CMP  $37
B85A:  D0 06     BNE  $B862
B85C:  A5 3A     LDA  $3A
B85E:  C5 38     CMP  $38
B860:  F0 0F     BEQ  $B871
B862:  38        SEC
B863:  A5 39     LDA  $39
B865:  E9 01     SBC  #$01
B867:  85 39     STA  $39
```

```
B869:  A5 3A      LDA  $3A
B86B:  E9 00      SBC  #$00
B86D:  85 3A      STA  $3A
B86F:  D0 E1      BNE  $B852
B871:  98         TYA
B872:  18         CLC
B873:  65 9C      ADC  $9C
B875:  85 9C      STA  $9C
B877:  A5 9D      LDA  $9D
B879:  69 00      ADC  #$00
B87B:  85 9D      STA  $9D
B87D:  60         RTS
B87E:  EA         NOP
B87F:  EA         NOP
B880:  08         PHP
B881:  A0 04      LDY  #$04
B883:  B1 00      LDA  ($00),Y
B885:  F0 03      BEQ  $B88A
B887:  C8         INY
B888:  D0 F9      BNE  $B883
B88A:  C8         INY
B88B:  28         PLP
B88C:  98         TYA
B88D:  B0 0E      BCS  $B89D
B88F:  20 48 B8   JSR  $B848
B892:  98         TYA
B893:  48         PHA
B894:  88         DEY
B895:  B1 00      LDA  ($00),Y
B897:  91 37      STA  ($37),Y
B899:  88         DEY
B89A:  10 F9      BPL  $B895
B89C:  68         PLA
B89D:  18         CLC
B89E:  65 00      ADC  $00
B8A0:  85 00      STA  $00
B8A2:  A5 01      LDA  $01
B8A4:  69 00      ADC  #$00
B8A6:  85 01      STA  $01
B8A8:  60         RTS
B8A9:  EA         NOP
B8AA:  EA         NOP
B8AB:  EA         NOP
B8AC:  EA         NOP
B8AD:  EA         NOP
B8AE:  38         SEC
B8AF:  A5 9C      LDA  $9C
B8B1:  E9 04      SBC  #$04
B8B3:  85 35      STA  $35
```

```
B8B5:  A5 9D      LDA  $9D
B8B7:  E9 05      SBC  #$05
B8B9:  85 36      STA  $36
B8BB:  A9 FF      LDA  #$FF
B8BD:  E5 35      SBC  $35
B8BF:  85 00      STA  $00
B8C1:  48         PHA
B8C2:  A9 B3      LDA  #$B3
B8C4:  E5 36      SBC  $36
B8C6:  85 01      STA  $01
B8C8:  48         PHA
B8C9:  A9 01      LDA  #$01
B8CB:  85 02      STA  $02
B8CD:  A9 05      LDA  #$05
B8CF:  85 03      STA  $03
B8D1:  A0 00      LDY  #$00
B8D3:  B1 02      LDA  ($02),Y
B8D5:  91 00      STA  ($00),Y
B8D7:  A5 01      LDA  $01
B8D9:  C9 B4      CMP  #$B4
B8DB:  F0 10      BEQ  $B8ED
B8DD:  E6 00      INC  $00
B8DF:  D0 02      BNE  $B8E3
B8E1:  E6 01      INC  $01
B8E3:  E6 02      INC  $02
B8E5:  D0 02      BNE  $B8E9
B8E7:  E6 03      INC  $03
B8E9:  A9 00      LDA  #$00
B8EB:  F0 E6      BEQ  $B8D3
B8ED:  A9 00      LDA  #$00
B8EF:  8D 7F 02   STA  $027F
B8F2:  8D 5B 02   STA  $025B
B8F5:  8D 5A 02   STA  $025A
B8F8:  20 6A E7   JSR  $E76A
B8FB:  20 7D E5   JSR  $E57D
B8FE:  20 AC E4   JSR  $E4AC
B901:  20 9B E5   JSR  $E59B
B904:  20 E0 E4   JSR  $E4E0
B907:  20 3D E9   JSR  $E93D
B90A:  AD AB 02   LDA  $02AB
B90D:  85 9C      STA  $9C
B90F:  AD AC 02   LDA  $02AC
B912:  85 9D      STA  $9D
B914:  20 41 B9   JSR  $B941
B917:  68         PLA
B918:  85 01      STA  $01
B91A:  68         PLA
B91B:  85 00      STA  $00
B91D:  A5 01      LDA  $01
```

```
B91F:  C9 B4      CMP  #$B4
B921:  F0 17      BEQ  $B93A
B923:  EA         NOP
B924:  EA         NOP
B925:  A0 02      LDY  #$02
B927:  B1 00      LDA  ($00),Y
B929:  85 35      STA  $35
B92B:  C8         INY
B92C:  B1 00      LDA  ($00),Y
B92E:  85 36      STA  $36
B930:  20 00 B8   JSR  $B800
B933:  20 80 B8   JSR  $B880
B936:  A9 00      LDA  #$00
B938:  F0 E3      BEQ  $B91D
B93A:  20 5F C5   JSR  $C55F
B93D:  20 14 FB   JSR  $FB14
B940:  60         RTS
B941:  38         SEC
B942:  A5 9C      LDA  $9C
B944:  E9 02      SBC  #$02
B946:  85 02      STA  $02
B948:  A5 9D      LDA  $9D
B94A:  E9 00      SBC  #$00
B94C:  85 03      STA  $03
B94E:  A0 00      LDY  #$00
B950:  A9 00      LDA  #$00
B952:  91 02      STA  ($02),Y
B954:  C8         INY
B955:  91 02      STA  ($02),Y
B957:  60         RTS
B958:  EA         NOP
B959:  EA         NOP
B95A:  EA         NOP
B95B:  EA         NOP
```

Program 8.4 Merge (version 1.1 ROMs)

8.5 AUTO DATA feature

This utility is designed to save time when typing long programs. As it stands, the program types the next line number (in sequence) followed by the command 'DATA', every time that you press RETURN.

This can be easily changed to any other automatic command – such as PRINT – or just the line number alone.

On version 1.0, remember to load the machine code program before you load the BASIC program, or the end-of-BASIC pointer will be incorrectly set up.

To start the AUTO feature, CALL #4A1. To stop the AUTO temporarily (to do an immediate command, such as CSAVE), you can use CONTROL-X. To turn off AUTO, you need to do two DOKE commands in immediate mode: For version 1.0 ROMs, do: DOKE #229,#EC03:POKE #230,64. For version 1.1 ROMs, do: DOKE #245,#EE22:POKE #24A,64.

Before you call the routine, you must DOKE 0 with the starting line number and DOKE 2 with the

line increment.

This routine can only handle line numbers up to 32767. You will also find that the first digit of the line number will be lost whenever the 'READY' message appears.

HOW AUTO DATA WORKS

The routine is called every time that an interrupt occurs (normally 100 times per second) before the keyboard is scanned. When the last key pressed was RETURN, the AUTO routine feeds in the next line number and the word 'DATA' (not as a token!). To the system, it is as if these keys have been pressed. You will notice a small delay is made between the end of one line and the start of the next. This is done because problems arose when characters were sent at full speed and the line was corrupted as it was stored in memory.

It is at #453 that the word 'DATA' is moved into a temporary buffer, but this can be altered or removed if required.

If you change location #454 to #0, the subroutine will only generate a line number.

PROGRAM LISTINGS

There are two program listings, one for each ROM version (Programs 8.5 and 8.6).

```
0410:  08          PHP
0411:  48          PHA
0412:  8A          TXA
0413:  48          PHA
0414:  98          TYA
0415:  48          PHA
0416:  AD DF 02    LDA  $02DF
0419:  30 0E       BMI  $0429
041B:  AD 00 04    LDA  $0400
041E:  C9 88       CMP  #$88
0420:  F0 57       BEQ  $0479
0422:  C9 66       CMP  #$66
0424:  D0 03       BNE  $0429
0426:  4C BB 04    JMP  $04BB
```

```
0429:  68          PLA
042A:  A8          TAY
042B:  68          PLA
042C:  AA          TAX
042D:  68          PLA
042E:  28          PLP
042F:  4C 03 EC    JMP  $EC03
0432:  A9 88       LDA  #$88
0434:  8D 00 04    STA  $0400
0437:  A9 01       LDA  #$01
0439:  8D 01 04    STA  $0401
043C:  A4 00       LDY  $00
043E:  A5 01       LDA  $01
0440:  20 ED D3    JSR  $D3ED
0443:  20 D1 E0    JSR  $E0D1
0446:  A0 00       LDY  #$00
0448:  B9 00 01    LDA  $0100,Y
044B:  F0 06       BEQ  $0453
044D:  99 02 04    STA  $0402,Y
0450:  C8          INY
0451:  D0 F5       BNE  $0448
0453:  A9 44       LDA  #$44
0455:  99 02 04    STA  $0402,Y
0458:  A9 41       LDA  #$41
045A:  99 03 04    STA  $0403,Y
045D:  99 05 04    STA  $0405,Y
0460:  A9 54       LDA  #$54
0462:  99 04 04    STA  $0404,Y
0465:  A9 00       LDA  #$00
0467:  99 06 04    STA  $0406,Y
046A:  18          CLC
046B:  A5 00       LDA  $00
046D:  65 02       ADC  $02
046F:  85 00       STA  $00
0471:  A5 01       LDA  $01
0473:  65 03       ADC  $03
0475:  85 01       STA  $01
0477:  EA          NOP
0478:  EA          NOP
0479:  AC 01 04    LDY  $0401
047C:  EE 01 04    INC  $0401
047F:  B9 02 04    LDA  $0402,Y
0482:  F0 07       BEQ  $048B
0484:  09 80       ORA  #$80
0486:  8D DF 02    STA  $02DF
0489:  D0 9E       BNE  $0429
048B:  8D 00 04    STA  $0400
048E:  F0 99       BEQ  $0429
0490:  08          PHP
```

```
0491:  48          PHA
0492:  AD DF 02    LDA  $02DF
0495:  C9 8D       CMP  #$8D
0497:  D0 05       BNE  $049E
0499:  A9 66       LDA  #$66
049B:  8D 00 04    STA  $0400
049E:  68          PLA
049F:  28          PLP
04A0:  40          RTI
04A1:  A9 4C       LDA  #$4C
04A3:  8D 30 02    STA  $0230
04A6:  A9 90       LDA  #$90
04A8:  8D 31 02    STA  $0231
04AB:  A9 04       LDA  #$04
04AD:  8D 32 02    STA  $0232
04B0:  A9 10       LDA  #$10
04B2:  8D 29 02    STA  $0229
04B5:  A9 04       LDA  #$04
04B7:  8D 2A 02    STA  $022A
04BA:  60          RTS
04BB:  AD 0F 04    LDA  $040F
04BE:  29 01       AND  #$01
04C0:  8D 0F 04    STA  $040F
04C3:  CE 0F 04    DEC  $040F
04C6:  D0 03       BNE  $04CB
04C8:  4C 32 04    JMP  $0432
04CB:  4C 29 04    JMP  $0429
```

Program 8.5 AUTO DATA utility (version 1.0 ROMs)

```
0410:  08          PHP
0411:  48          PHA
0412:  8A          TXA
0413:  48          PHA
0414:  98          TYA
0415:  48          PHA
0416:  AD DF 02    LDA  $02DF
0419:  30 0E       BMI  $0429
041B:  AD 00 04    LDA  $0400
041E:  C9 BB       CMP  #$BB
0420:  F0 57       BEQ  $0479
0422:  C9 66       CMP  #$66
0424:  D0 03       BNE  $0429
0426:  4C BB 04    JMP  $04BB
0429:  68          PLA
042A:  A8          TAY
042B:  68          PLA
042C:  AA          TAX
042D:  68          PLA
```

Program 8.6 *(continues)*

```
042E:  28        PLP
042F:  4C 22 EE  JMP  $EE22
0432:  A9 88     LDA  #$88
0434:  8D 00 04  STA  $0400
0437:  A9 01     LDA  #$01
0439:  8D 01 04  STA  $0401
043C:  A4 00     LDY  $00
043E:  A5 01     LDA  $01
0440:  20 99 D4  JSR  $D499
0443:  20 D5 E0  JSR  $E0D5
0446:  A0 00     LDY  #$00
0448:  B9 00 01  LDA  $0100,Y
044B:  F0 06     BEQ  $0453
044D:  99 02 04  STA  $0402,Y
0450:  C8        INY
0451:  D0 F5     BNE  $0448
0453:  A9 44     LDA  #$44
0455:  99 02 04  STA  $0402,Y
0458:  A9 41     LDA  #$41
045A:  99 03 04  STA  $0403,Y
045D:  99 05 04  STA  $0405,Y
0460:  A9 54     LDA  #$54
0462:  99 04 04  STA  $0404,Y
0465:  A9 00     LDA  #$00
0467:  99 06 04  STA  $0406,Y
046A:  18        CLC
046B:  A5 00     LDA  $00
046D:  65 02     ADC  $02
046F:  85 00     STA  $0C
0471:  A5 01     LDA  $01
0473:  65 03     ADC  $03
0475:  85 01     STA  $01
0477:  EA        NOP
0478:  EA        NOP
0479:  AC 01 04  LDY  $0401
047C:  EE 01 04  INC  $0401
047F:  B9 02 04  LDA  $0402,Y
0482:  F0 07     BEQ  $048B
0484:  09 80     ORA  #$80
0486:  8D DF 02  STA  $02DF
0489:  D0 9E     BNE  $0429
048B:  8D 00 04  STA  $0400
048E:  F0 99     BEQ  $0429
0490:  08        PHP
0491:  48        PHA
0492:  AD DF 02  LDA  $02DF
0495:  C9 8D     CMP  #$8D
0497:  D0 05     BNE  $049E
0499:  A9 66     LDA  #$66
```

```
049B:  8D 00 04    STA  $0400
049E:  68          PLA
049F:  28          PLP
04A0:  40          RTI
04A1:  A9 4C       LDA  #$4C
04A3:  8D 4A 02    STA  $024A
04A6:  A9 90       LDA  #$90
04A8:  8D 4B 02    STA  $024B
04AB:  A9 04       LDA  #$04
04AD:  8D 4C 02    STA  $024C
04B0:  A9 10       LDA  #$10
04B2:  8D 45 02    STA  $0245
04B5:  A9 04       LDA  #$04
04B7:  8D 46 02    STA  $0246
04BA:  60          RTS
04BB:  AD 0F 04    LDA  $040F
04BE:  29 01       AND  #$01
04C0:  8D 0F 04    STA  $040F
04C3:  CE 0F 04    DEC  $040F
04C6:  D0 03       BNE  $04CB
04C8:  4C 32 04    JMP  $0432
04CB:  4C 29 04    JMP  $0429
```

Program 8.6 AUTO DATA utility (version 1.1 ROMs)

8.6 Trace utility

This program helps a BASIC program to be debugged by constantly displaying the current line number as the program runs.

This is often useful in determining what exactly a program is doing. If a program should crash, or go into a tight loop, then this will be immediately noticeable.

USING THE PROGRAM

The Trace program should be loaded from tape first, followed by the program to be traced. On version 1.0 this order is important since the end-of-BASIC pointer (#9C) must reflect the end of the BASIC program.

To start the trace, type CALL #495 – a very large number should appear in the top left corner. When you run your BASIC program, this number will change to show the line number currently being executed.

HOW IT WORKS

The program is called by the slow interrupt vector, but only updates the line number when it changes. Locations #A8, #A9 contain the current line number in integer form, so this must be converted to decimal and displayed. This could be done with ROM subroutines, but you must remember that we are in the middle of an interrupt call; it is important not to disturb any page 0 and page 2 locations that might be in use.

In the Trace program, we use a standard binary-to-decimal technique which involves the subtraction of the powers of 10.

The Trace program demonstrates how it is possible for the Oric to do two tasks at the same time. The demonstration program for the Oric Atmos uses interrupts in order to play music while the main BASIC program runs. Chapter 9 shows how it is possible to run two BASIC programs concurrently – again using interrupts.

PROGRAM LISTING

First of all, there is a table of 11 bytes at #4A1:

#4A1: #10, #27, #E8, #03, #64, #00, #OA, #00, #4C, #22, #04

This table contains the binary value for each power of 10. At the end of the table is the jump that overwrites the page 2 slow interrupt vector. Owners of version 1.1 ROMs should DOKE #49B with #24A because of the different interrupt patch address.

The program listing is given in Program 8.7.

```
0422:  48          PHA
0423:  8A          TXA
0424:  48          PHA
0425:  98          TYA
0426:  48          PHA
0427:  78          SEI
0428:  D8          CLD
0429:  A5 A8       LDA  $A8
042B:  C5 02       CMP  $02
042D:  D0 0E       BNE  $043D
042F:  A5 A9       LDA  $A9
0431:  C5 03       CMP  $03
0433:  F0 5A       BEQ  $048F
0435:  A5 A8       LDA  $A8
0437:  85 02       STA  $02
0439:  A5 A9       LDA  $A9
043B:  85 03       STA  $03
043D:  A9 00       LDA  #$00
043F:  8D 20 04    STA  $0420
0442:  A5 A8       LDA  $A8
0444:  85 00       STA  $00
0446:  A5 A9       LDA  $A9
0448:  85 01       STA  $01
```

```
044A:  A0 00     LDY  #$00
044C:  A2 00     LDX  #$00
044E:  A5 00     LDA  $00
0450:  38        SEC
0451:  F9 A1 04  SBC  $04A1,Y
0454:  85 00     STA  $00
0456:  A5 01     LDA  $01
0458:  C8        INY
0459:  F9 A1 04  SBC  $04A1,Y
045C:  90 07     BCC  $0465
045E:  85 01     STA  $01
0460:  E8        INX
0461:  88        DEY
0462:  38        SEC
0463:  B0 E9     BCS  $044E
0465:  88        DEY
0466:  A5 00     LDA  $00
0468:  79 A1 04  ADC  $04A1,Y
046B:  85 00     STA  $00
046D:  8A        TXA
046E:  09 30     ORA  #$30
0470:  8E 21 04  STX  $0421
0473:  AE 20 04  LDX  $0420
0476:  9D 80 BB  STA  $BB80,X
0479:  EE 20 04  INC  $0420
047C:  AE 21 04  LDX  $0421
047F:  C8        INY
0480:  C8        INY
0481:  C0 08     CPY  #$08
0483:  90 C7     BCC  $044C
0485:  A5 00     LDA  $00
0487:  09 30     ORA  #$30
0489:  AE 20 04  LDX  $0420
048C:  9D 80 BB  STA  $BB80,X
048F:  68        PLA
0490:  A8        TAY
0491:  68        PLA
0492:  AA        TAX
0493:  68        PLA
0494:  40        RTI
0495:  A0 02     LDY  #$02
0497:  B9 A9 04  LDA  $04A9,Y
049A:  99 30 02  STA  $0230,Y
049D:  88        DEY
049E:  10 F7     BPL  $0497
04A0:  60        RTS
04A1:  EA        NOP
04A2:  EA        NOP
04A3:  EA        NOP
```

8.7 On-error GOTO feature

When a BASIC program stops, it always returns to command mode. This can be undesirable, especially on the production version of a complicated program, where obscure bugs may still be lurking. Also, it is often a nice touch to detect control-C, and not just crash the machine, but instead jump back into the program.

This short utility traps any attempt to return to command mode and forces the computer to re-enter the program at line 500, without loss of variables.

Be warned that using this routine can be a little annoying to yourself, especially when you find yourself stuck in your own program!

HOW IT WORKS

When BASIC finishes a program, or a command, it prints 'Ready'. This is not done directly, but instead through a jump command at #1A to #IC. This means that the jump can be modified for our own purposes. Often the address at #1B is changed so that the machine simply jumps to the start-up routine – wiping everything out. If you want to do this incidentally, type DOKE #1B,DEEK(#FFFC).

Here we change the vector to jump to #B1DO (the program can be easily relocated to another address if you wish), so you must DOKE #1B, #B1DO.

The routine does exactly what would have normally happened; then we force 'GOTO500' into BASIC's input buffer, as though it had been typed, which persuades the machine to re-enter the program. The GOTO500 can easily he changed to any other command. Note that when the program is re-entered, all GOSUBs have effectively been POPped, so RETURN will produce an error message – and unless you are very careful, you will end up with unceasing display of that error message, since there is now a fault in the error handler!

HOW TO USE THE PROGRAM

Version 1.1 owners will need to change #B1F0 to JMP #C4BD. The on-error feature is switched on by DOKE #1B, #B1D0 and off by either DOKE #1B,#CBED (version 1.0) or DOKE #1B,#CCB0 (version 1.1). You can quite easily change the line number that is jumped to by altering #B1D8 to #B1E3.

Note that should a BASIC error occur, you will still get the error message printed before the program continues. This is one occasion where control-S can be used in order to inhibit the printing of error messages. The screen will still scroll if the cursor is within the bottom four lines, regardless of control-S.

PROGRAM LISTING

```
B1D0:   68          PLA
B1D1:   4E F1 02    LSR   $02F1
B1D4:   A9 97       LDA   #$97
B1D6:   85 35       STA   $35
B1D8:   A9 35       LDA   #$35
B1DA:   85 36       STA   $36
B1DC:   A9 30       LDA   #$30
B1DE:   85 37       STA   $37
B1E0:   A9 30       LDA   #$30
B1E2:   85 38       STA   $38
B1E4:   A2 04       LDX   #$04
B1E6:   A0 00       LDY   #$00
B1E8:   84 39       STY   $39
B1EA:   A2 34       LDX   #$34
B1EC:   A9 13       LDA   #$13
B1EE:   85 30       STA   $30
B1F0:   4C CD C4    JMP   $C4CD
```

Program 8.8 On-error GOTO facility

9. STRETCHING THE ORIC TO ITS LIMITS

9.1 Introduction

This chapter presents a few ideas that are more interesting than practical. It is hoped that these last few programs will encourage further experimentation – perhaps to improve the methods used.

9.2 Speech synthesis program

The first thing to be said here is that you should not expect too much of this program!

Speech synthesis is normally done with the help of a special add-on piece of hardware. The two programs below show that a limited form of speech synthesis is possible on an unexpanded Oric. The speech produced is frequently unintelligible, requires about 2K per second of speech, but can add a touch of magic to a dull program.

The program here fills up about 15K of memory in around 7 to 10 seconds (depending on the content of the message).

USING THE PROGRAM

There are two very short programs (it is just the data that is bulky!).

The first reads from the cassette port and produces a stream of data in memory. The second reverses the process, but puts out the speech through the loudspeaker via the sound chip.

The best way to create a message (at least when you first experiment) is to set up the cassette recorder so that as you record, the signal goes directly to the Oric. On many cassette recorders, this is done by disconnecting the recording jack, so that the internal microphone is used, but leaving in the earphone jack. If your cassette recorder cannot do this, or has a five-pin connector, you will have to record the message on the cassette recorder and then play it back.

For best results speak loudly, clearly, slowly, and very near to the microphone. If you are recording on to tape, play your voice back at a very high volume. You will find that music will not come out in a recognizable form, although pure tones (such as whistling) come out clearly, but much faster.

If you have difficulties at first, try different levels of playback, and above all remember to speak S-L-O-W-L- Y!

Words containing the letters T, S, and D will sound better than letters such as P, L, and R.

To record a message, type CALL #420 once all the connections have been made, and start talking immediately! After about 10 seconds of constant speech the program should return to you. If not, then something has gone wrong – the Reset button should get you out of trouble.

When you are ready to hear the Oric's interpretation of your message, type in:

PLAY 1,0,0,0:SOUND1,1,1:CALL #480.

Prepare to be disappointed for the first few attempts!

THE THEORY BEHIND THE PROGRAM

The first program at #420 works as follows:

l. Interrupts are disabled – we need full use of the machine and the cassette also needs to be used.

2. Locations 2,3 are used to point to the next address where data is stored.

3. At #42D, the cassette input bit is cleared by reading from port B.

4. When #30D contains #52, then one bit has been received from the cassette input port; otherwise counter X is increased – measuring the gaps between input bits.

5. When a bit is received, or the counter reaches 255, the value of the counter is stored at the next address as pointed to by (2,3)

6. When the pointer at (2,3) reaches #3400, interrupts are enabled again and the program returns to BASIC.

Obviously, you can change the lower and upper limits of the data area to suit your needs. Once the data has been input, you can edit it – for example, to remove any delay at the front of the message. The data can then be stored on tape, or incorporated into a larger program.

The second program has to work in reverse of the first, turning the series of counts into a series of clicks. Providing that these clicks are separated by the same time interval as the gaps between each bit in the original signal, you should get an approximation of the speech. The main problem encountered when developing the idea was that the ROM subroutine which writes to the 8912 sound chip is incredibly inefficient. From #4AD to #4C6 you will find a considerably faster routine to write value X into register A.

The second program alternates between sending a volume of 7 and a volume of 13 to channel A. The SOUND 1,1,1 command will have set up a frequency that is beyond both human hearing and the capabilities of the loudspeaker, so the basic sound signal does not show up during pauses.

Here is how the program at #480 functions:

1. Disables interrupts to get the maximum use of the machine.

2. Sets up the pointer (2,3) to the start of the data area.

3. Delays depending on the next byte of data. The NOP instructions act as a fine tune to get the best results. Two-millionths of a second can make all the difference to this program!

4. Sets the volume to either 7 or 13, and writes this to register 8 of the 8912 chip.

5. When the pointer (2,3) reaches #3400, enables interrupts and returns to BASIC.

CHANGES FOR VERSION 1.1 ROMS

Four changes are required if you own a version 1.1 ROM:

#420 JSR #E76A

#451 JSR #E93D

#480 JSR #E76A

#4D2 JSR #E93D

PROGRAM LISTINGS

There are two – Programs 9.1 and 9.2

```
0420:  20 CA E6    JSR  $E6CA
0423:  A9 00       LDA  #$00
0425:  85 02       STA  $02
0427:  A9 06       LDA  #$06
0429:  85 03       STA  $03
042B:  A0 00       LDY  #$00
042D:  AD 00 03    LDA  $0300
0430:  A2 00       LDX  #$00
0432:  A9 52       LDA  #$52
0434:  CD 0D 03    CMP  $030D
0437:  F0 05       BEQ  $043E
0439:  E8          INX
043A:  D0 F8       BNE  $0434
043C:  A2 FF       LDX  #$FF

043E:  8A          TXA
043F:  91 02       STA  ($02),Y
0441:  A2 01       LDX  #$01
0443:  CA          DEX
0444:  D0 FD       BNE  $0443
0446:  C8          INY
0447:  D0 E4       BNE  $042D
0449:  E6 03       INC  $03
044B:  A5 03       LDA  $03
044D:  C9 34       CMP  #$34
044F:  D0 DC       BNE  $042D
0451:  20 04 E8    JSR  $E804
0454:  60          RTS
```

Program 9.1 Speech input (#420–#454)

Program 9.2 follows.

```
0480:  20 CA E6    JSR  $E6CA
0483:  A9 00       LDA  #$00
0485:  85 02       STA  $02
0487:  A9 06       LDA  #$06
0489:  85 03       STA  $03
048B:  A0 00       LDY  #$00
048D:  B1 02       LDA  ($02),Y
048F:  AA          TAX
0490:  CA          DEX
0491:  EA          NOP
0492:  EA          NOP
0493:  EA          NOP
0494:  CA          DEX
0495:  D0 FD       BNE  $0494
0497:  C9 FF       CMP  #$FF
0499:  B0 2C       BCS  $04C7
049B:  EA          NOP
049C:  A5 04       LDA  $04
049E:  C9 07       CMP  #$07
04A0:  D0 04       BNE  $04A6
04A2:  A9 0D       LDA  #$0D
04A4:  D0 02       BNE  $04A8
04A6:  A9 07       LDA  #$07
04A8:  85 04       STA  $04
04AA:  AA          TAX
04AB:  A9 08       LDA  #$08
04AD:  8D 0F 03    STA  $030F
04B0:  A9 FE       LDA  #$FE
04B2:  8D 0C 03    STA  $030C
04B5:  29 DD       AND  #$DD
04B7:  8D 0C 03    STA  $030C

04BA:  8E 0F 03    STX  $030F
04BD:  09 20       ORA  #$20
04BF:  8D 0C 03    STA  $030C
04C2:  29 DD       AND  #$DD
04C4:  8D 0C 03    STA  $030C
04C7:  C8          INY
04C8:  D0 C3       BNE  $048D
04CA:  E6 03       INC  $03
04CC:  A5 03       LDA  $03
04CE:  C9 34       CMP  #$34
04D0:  D0 BB       BNE  $048D
04D2:  20 04 E8    JSR  $E804
04D5:  60          RTS
04D6:  EA          NOP
```

9.3 Extra 6502 op-codes

Out of the 256 possible instruction codes, about 100 would appear to be unused. However, if you try to execute any of these, one of three things can happen:

1. The machine crashes.

2. The instruction acts like a NOP, and alters nothing.

3. The instruction obeys a combination of instructions.

The first of these is very puzzling – it appears that the 6502 itself halts, refusing to obey any more commands until it is reset (not by the NMI button underneath the Oric). These instructions, which we might give the mnemonic KILL, have instruction codes ending in #2 – e.g., #22 – except for the valid instruction #A2.

The second category is not very important, except that, in doing nothing, the instructions are still useful in protecting a program from being understood! A disassembler program will usually be unable to cope with any unknown instructions and will often be misled into passing over real instructions in your program!

Just as real instructions can take 1, 2, or 3 bytes, so can our new 'NOP' instructions:

One-byte NOP instructions: #1A, #3A, #5A, #7A, #DA, #EA, #FA.

Two-byte NOP instructions: #64, #74, #D4, #F4.

Three-byte NOP instructions: #0C, #1C, #3C, #5C, #7C, #9C, #DC, #FC.

The third category are the most interesting instructions – hybrid op-codes. Some instructions on the 6502 do two operations at once. These are instruction codes ending in #3, #7, and #F. You may find that instructions ending in #B also do a combination of things, but not to any fixed pattern.

What happens to these instructions is that they execute two instructions in quick succession. For op-codes ending in 3, combine that op-code with an ending of 6, followed by the same op-code with an ending of 1. For example, #23 is #26 and #21, or:

ROL NN

AND (NN,X)

This is not a particularly useful combination, yet interesting nonetheless.

Similarly, op-codes that end in 7 are combinations of 6 and 5. For instance, #27 is #26, #25, or:

ROL NN

AND NN

Finally, op-codes that end in 4F are combinations of #E and #D. So #2F is the same as #2E and #2D, or:

ROL NNNN

AND NNNN

IMPORTANT NOTE

There is no guarantee that the hidden op-codes act in the same way on all 6502 microprocessors. It is fairly likely that all Oric machines behave in the same way, but it is still a risky business to rely upon any undocumented instruction.

9.4 Multitasking in BASIC

Fundamentally, a computer such as the Oric can only execute one instruction at a time. This is done at such a speed that a computer can appear to run two or more programs concurrently.

This happens on the Oric every hundredth of a second, in order to handle interrupts, and the impression is given that two things are happening at once – the cursor flashing on and off is an example.

As some of the programs in this book have shown, it is quite possible to use interrupts to run a small machine code program as a background task. It is more of a problem to be able to cope with two BASIC programs running simultaneously, and such is the purpose of the routine in this section.

THEORY

The major problem with switching between two BASIC programs is that they need their own versions of page 0, stack, and page 2 memory. Since copying 1500 bytes of data is a time-consuming task, even for machine code, we can only afford to interchange the running of the two programs about every twelfth of a second. Any less than that and we would be spending too little time on the actual programs; any more and the interchange would become more noticeable.

The program uses #8100 to #83FF to store the first three pages of the BASIC program 1 and #8400 to #86FF for BASIC program 2.

The multitasking is called by the slow interrupt vector, i.e., every hundredth of a second. If the counter at #87FF is not either 0 or 8, the routine simply returns; otherwise it switches from its current place in the program to the other position, moving about all the important locations that BASIC uses. Instead of having two different BASIC programs which are swapped in and out, this utility works by allowing one BASIC program to have two independent sections running. All BASIC statements will work – including CALL.

USING THE PROGRAM

A special part of the routine starts the procedure, by setting the counter at #87FF to 255. This gives the 'first' program a chance to split off into a different section. It will be about two seconds before the machine will switch to the 'second' program. The BASIC program example will demonstrate this (Program 9.3).

```
1 REM EXAMPLE OF SPLIT PROGRAM
5 CLS:KK=0
10 DOKE#87AF,#877F:DOKE#231,#87AE:POKE#230,76
20 REM SPLIT OFF FIRST PROGRAM
25 FORX=1TO99:NEXT
30 IFKK=0THENKK=1:GOTO6000
35 REM PROGRAM B GETS TO HERE
40 FORZ=1TO20000:PLOT10,10,STR$(Z)
50 NEXT
5999 REM PROGRAM A GETS HERE
6000 FORX=1TO20000:PLOT20,20,STR$(X)
6005 ZAP:FORT=1TO100:NEXT
6010 NEXT
```

Program 9.3 BASIC example of multi-tasking

Firstly, #87AF is DOKEd with #877F – this makes the very first interrupt go to the special routine at #877F, instead of the normal address at #8700.

Then the slow interrupt patch is entered – at #231 for version 1.0 (as listed) or at #24B for version 1.1 – please use the appropriate address. Finally, location #230 (version 1.0) or #24A (version 1.1) is POKEd with 76.

Once this last instruction is complete and the first interrupt occurs (which will happen some time during the FOR... NEXT loop), the current BASIC circumstances are saved as the starting point for program 2.

Program 1 will then continue until 247 interrupts have passed, and has ample time to switch to line 6000, preventing the second program from following.

When the second program does get to line 30, it will find KK equal to 1, and will drop through to line 40.

PROGRAMMING LIMITATIONS

Since only the first three pages of memory are being switched, both the BASIC program and its variables are being shared between the programs. Once the program has separated into two paths, you will get into trouble if you try to set up variables in each section since they keep their own account of the end of variables, strings, etc. It is a good idea to have one section creating the variables and the other only using variables set up before the multitasking began.

Although you can have a lot of fun experimenting with this idea (try pressing control-C!), there are many pitfalls, and its practical use may be limited.

Note that the machine code areas ought to be protected by a HIMEM #80FF command.

To stop the programs multi-tasking, cancel one of the programs (the other will carry on while you are typing!) and enter either POKE #230,64 (for version 1.0) or POKE #24A,64 (for version 1.1).

PROGRAM LISTING

```
8700:  78        SEI
8701:  48        PHA
8702:  8A        TXA
8703:  48        PHA
8704:  98        TYA
8705:  48        PHA
```

```
8706:  AD FF 87    LDA  $87FF
8709:  F0 37       BEQ  $8742
870B:  C9 08       CMP  #$08
870D:  D0 67       BNE  $8776
870F:  BA          TSX
8710:  8E FB 87    STX  $87FB
8713:  A2 00       LDX  #$00
8715:  B5 00       LDA  $00,X
8717:  9D 00 81    STA  $8100,X
871A:  BD 00 01    LDA  $0100,X
871D:  9D 00 82    STA  $8200,X
8720:  BD 00 02    LDA  $0200,X
8723:  9D 00 83    STA  $8300,X
8726:  BD 00 84    LDA  $8400,X
8729:  95 00       STA  $00,X
872B:  BD 00 85    LDA  $8500,X
872E:  9D 00 01    STA  $0100,X
8731:  BD 00 86    LDA  $8600,X
8734:  9D 00 02    STA  $0200,X
8737:  CA          DEX
8738:  D0 DB       BNE  $8715
873A:  AE FC 87    LDX  $87FC
873D:  9A          TXS
873E:  A9 00       LDA  #$00
8740:  F0 34       BEQ  $8776
8742:  BA          TSX
8743:  8E FC 87    STX  $87FC
8746:  A2 00       LDX  #$00
8748:  B5 00       LDA  $00,X
874A:  9D 00 84    STA  $8400,X
874D:  BD 00 01    LDA  $0100,X
8750:  9D 00 85    STA  $8500,X
8753:  BD 00 02    LDA  $0200,X
8756:  9D 00 86    STA  $8600,X
8759:  BD 00 81    LDA  $8100,X
875C:  95 00       STA  $00,X
875E:  BD 00 82    LDA  $8200,X
8761:  9D 00 01    STA  $0100,X
8764:  BD 00 83    LDA  $8300,X
8767:  9D 00 02    STA  $0200,X
876A:  CA          DEX
876B:  D0 DB       BNE  $8748
876D:  AE FB 87    LDX  $87FB
8770:  9A          TXS
8771:  A9 10       LDA  #$10
8773:  8D FF 87    STA  $87FF
8776:  CE FF 87    DEC  $87FF
8779:  68          PLA
877A:  A8          TAY
```

```
877B:  68          PLA
877C:  AA          TAX
877D:  68          PLA
877E:  40          RTI
877F:  48          PHA
8780:  8A          TXA
8781:  48          PHA
8782:  98          TYA
8783:  48          PHA
8784:  A9 FF       LDA  #$FF
8786:  8D FF 87    STA  $87FF
8789:  A9 00       LDA  #$00
878B:  8D AF 87    STA  $87AF
878E:  A2 00       LDX  #$00
8790:  B5 00       LDA  $00,X
8792:  9D 00 84    STA  $8400,X
8795:  BD 00 01    LDA  $0100,X
8798:  9D 00 85    STA  $8500,X
879B:  BD 00 02    LDA  $0200,X
879E:  9D 00 86    STA  $8600,X
87A1:  CA          DEX
87A2:  D0 EC       BNE  $8790
87A4:  BA          TSX
87A5:  8E FC 87    STX  $87FC
87A8:  68          PLA
87A9:  A8          TAY
87AA:  68          PLA
87AB:  AA          TAX
87AC:  68          PLA
87AD:  40          RTI
87AE:  4C 00 87    JMP  $8700
87B1:  EA          NOP
87B2:  EA          NOP
87B3:  EA          NOP
87B4:  EA          NOP
```

Program 9.4 Multi-tasking routine

9.5 Single-key facility

Since the first appearance of cheap computers, there have evolved two methods of entering programs.

The first Sinclair computers up to the ZX Spectrum use a single-key system, in which every key, when combined with different shifts, generates a complete BASIC word. For example, pressing 'R' could result in 'RANDOMIZE' appearing.

On the Oric, and almost all of the more expensive computers, each command must be entered letter by letter. The reason for this is that BASIC is not

necessarily the only language available, and the BASIC commands would be meaningless to FORTH, Assembler, etc.

The program in this section gives the capability of single-key command entry. Although intended for use with BASIC, you could quite easily change the table of commands to work with other languages.

USING THE PROGRAM

The program occupies the first two pages of the alternate character set, and so will only be dislodged by a HIRES command, or the Reset button.

To run the program, type CALL #B894.

Owners of version 1.1 ROMs should also change #B89A to #49, instead of #2F.

While the single-key program is running, you can carry on typing commands in full by switching to lower case (use control- T). Lower case will be turned into upper case when commands are entered, and lower case only is applied when quotes are used. This in itself is a useful tool when entering a lot of PRINT or DATA statements.

When a capital letter is entered outside of quotes, a command is inserted. For example, 'N' might produce the word 'NEXT'. These commands are generated from a table at #B900. This table contains the ASCII codes required for each character between #40 and #5A. Each ASCII string must be terminated by #00. This will be clarified if you examine the table of single-key commands, Table 9.1.

Table 9.1

```
B900: 40 00 41 53 43 28 00 43
B908: 53 41 56 45 00 43 4C 4F
B910: 41 44 00 44 41 54 41 00
B918: 45 4C 53 45 00 46 4F 52
B920: 00 47 4F 54 4F 00 48 49
B928: 52 45 53 00 49 4E 50 55
B930: 54 00 47 4F 53 55 42 00
B938: 4B 45 59 24 00 4C 49 53
B940: 54 00 4D 55 53 49 43 00
B948: 4E 45 58 54 00 50 45 45
B950: 4B 00 50 4C 41 59 00 43
B958: 48 52 24 28 00 52 45 54
B960: 55 52 4E 00 53 54 52 24
B968: 28 00 54 4F 00 55 4E 54
B970: 49 4C 00 56 41 4C 28 00
B978: 57 41 49 54 00 45 58 50
B980: 4C 4F 44 45 00 52 45 50
B988: 45 41 54 00 5A 41 50 00
```

The single-key facility can be stopped by changing either #230 (version 1.0) or #24A (version 1.1) to 64, using the POKE statement.

HOW IT WORKS

The program patches into the ever-popular slow interrupt vector so that it can alter any keypress found in #2DF.

If a lower-case letter is entered and no quote has been found, it is converted to

upper case with a simple AND #DF instruction. If it is upper case, then the appropriate word is located in the table, and that word is fed out to #2DF, character by character, as part of the interrupt routine. A similar technique was used by the AUTO DATA program of Chapter 8.

PROGRAM LISTING

The program occupies #B800 to #B8AA but could be easily relocated (Program 9.5).

```
B800:  78          SEI
B801:  48          PHA
B802:  98          TYA
B803:  48          PHA
B804:  8A          TXA
B805:  48          PHA
B806:  A5 00       LDA  $00
B808:  F0 19       BEQ  $B823
B80A:  AD DF 02    LDA  $02DF
B80D:  D0 73       BNE  $B882
B80F:  A4 00       LDY  $00
B811:  B9 00 B9    LDA  $B900,Y
B814:  F0 09       BEQ  $B81F
B816:  09 80       ORA  #$80
B818:  8D DF 02    STA  $02DF
B81B:  E6 00       INC  $00
B81D:  D0 5B       BNE  $B87A
B81F:  85 00       STA  $00
B821:  F0 5F       BEQ  $B882
B823:  AD DF 02    LDA  $02DF
B826:  F0 5A       BEQ  $B882
B828:  29 7F       AND  #$7F
B82A:  C9 22       CMP  #$22
B82C:  D0 0F       BNE  $B83D
B82E:  AD FF B8    LDA  $B8FF
B831:  D0 05       BNE  $B838
B833:  EE FF B8    INC  $B8FF
B836:  D0 42       BNE  $B87A
```

```
B838:  CE FF B8    DEC  $B8FF
B83B:  F0 3D       BEQ  $B87A
B83D:  AE FF B8    LDX  $B8FF
B840:  D0 38       BNE  $B87A
B842:  C9 61       CMP  #$61
B844:  90 08       BCC  $B84E
B846:  C9 7B       CMP  #$7B
B848:  B0 04       BCS  $B84E
B84A:  29 DF       AND  #$DF
B84C:  D0 0A       BNE  $B858
B84E:  C9 41       CMP  #$41
B850:  90 09       BCC  $B85B
B852:  C9 5B       CMP  #$5B
B854:  B0 05       BCS  $B85B
B856:  09 80       ORA  #$80
B858:  8D DF 02    STA  $02DF
B85B:  C9 C1       CMP  #$C1
B85D:  90 1B       BCC  $B87A
B85F:  C9 DB       CMP  #$DB
B861:  B0 17       BCS  $B87A
B863:  29 1F       AND  #$1F
B865:  EA          NOP
B866:  A8          TAY
B867:  A2 00       LDX  #$00
B869:  98          TYA
B86A:  F0 0A       BEQ  $B876
B86C:  E8          INX
B86D:  BD 00 B9    LDA  $B900,X
B870:  D0 FA       BNE  $B86C
B872:  88          DEY
B873:  D0 F7       BNE  $B86C
B875:  E8          INX
B876:  86 00       STX  $00
B878:  D0 95       BNE  $B80F
B87A:  AD DF 02    LDA  $02DF
B87D:  09 80       ORA  #$80
B87F:  8D DF 02    STA  $02DF
B882:  AD DF 02    LDA  $02DF
B885:  C9 8D       CMP  #$8D
B887:  D0 05       BNE  $B88E
B889:  A9 00       LDA  #$00
B88B:  8D FF B8    STA  $B8FF
B88E:  68          PLA
B88F:  AA          TAX
B890:  68          PLA
B891:  A8          TAY
B892:  68          PLA
B893:  40          RTI
B894:  A0 03       LDY  #$03
```

```
B896:  B9 A7 B8    LDA  $B8A7,Y
B899:  99 2F 02    STA  $022F,Y
B89C:  88          DEY
B89D:  D0 F7       BNE  $B896
B89F:  8C FF B8    STY  $B8FF
B8A2:  A9 2B       LDA  #$2B
B8A4:  8D 07 03    STA  $0307
B8A7:  60          RTS
B8A8:  4C 00 B8    JMP  $B800
B8AB:  EA          NOP
B8AC:  EA          NOP
B8AD:  EA          NOP
B8AE:  EA          NOP
```

Program 9.5 Single-key utility

9.6 Silence routine

The last program in this book can be used to shut up even the noisiest program!

It works by altering the slow interrupt vector so that every hundredth of a second all sound channels are disabled.

The routine will work for most programs, failing in cases where the interrupts are tampered with. Most sound commands (including the keyclick) will generate a very soft click, although some sound effects (such as PING and EXPLODE) will present part of their noise before being silenced.

USING THE PROGRAM

Version 1.1 ROM owners should change the address at #42E to #F590, instead of #F535.

Load up the silence routine first, and type

DOKE#231,#420:POKE#230,76 for version 1.0 ROMs or
DOKE#24B,#420:POKE#24A,76 for version 1.1 ROMs

The silence routine should now be in service – try typing ZAP – and you can now load in the program to be silenced.

The silence routine can be finished by typing POKE #230,64 for version 1.0 ROMs or POKE #24A,64 for version 1.1 ROMs.

PROGRAM LISTING

This is given in Program 9.6.

```
0420:  48          PHA
0421:  8A          TXA
0422:  48          PHA
0423:  98          TYA
0424:  48          PHA
0425:  AD 0F 02    LDA  $020F
0428:  48          PHA
0429:  A2 3F       LDX  #$3F
042B:  A9 07       LDA  #$07
042D:  20 35 F5    JSR  $F535
0430:  68          PLA
0431:  8D 0F 02    STA  $020F
0434:  68          PLA
0435:  A8          TAY
0436:  68          PLA
0437:  AA          TAX
0438:  68          PLA
0439:  40          RTI
```

Afterthoughts (August 1998)

Hehe, this chapter shows its age doesn't it. I'm afraid my ignorance of some of the basic concepts of digital audio in 1983 show up here, the proper way of playing back sound effects is of course to use analogue to digital, store the data, and then the fun begins when we try to playback. There is a way on this rather limited sound chip (when compared with today's luxurious sound blasters) to playback PCM data (i.e. wave files), which is to use the volume controls on the three channels to simulate the wave's amplitude - you'd probably need a 'scope to do this properly.

I think my explanation of the extra op-codes is probably inaccurate, it is not so much that two instructions are executed, rather that the logic inside the ALU of the 6502 decodes the instruction, and the logic simply has the effect of following a combination of instructions.

I wonder if any of the emulators for 6502 based machines work in the same way…

I like my bias against ZX Spectrums in this chapter, heh, I did go on to program the Spectrums of this world, but I have to confess to having used an Amstrad CPC to do all the programming, I just couldn't stand that wretched keyboard.