CHAPTER 3

PURE MACHINE CODE

We have seen in the last two chapters how to program in assembly code. However, this language is completely artificial; it is not one that the computer understands directly. We have to convert an assembly code into machine code, using an assembler, before we can use it. For most purposes this is ideal for us as it means we don't have to understand pure machine code. If, however, you are looking at someone else's program you may not have the original assembly code but only the machine code. In some circumstances it is easier to write in pure machine code − if you can do it! This chapter will show you how assembly code is converted into pure machine code.

A machine code program is stored in the memory as a series of consecutive bytes. Each instruction takes up either one, two or three bytes depending on what addressing mode it uses.

The first byte of an instruction tells the CPU which command is being used and also which addressing mode it is being used with.

There is a specific one-byte code for each available combination of command and addressing mode. This byte is called the OP-CODE. Because not all the addressing modes can be used with each command the total number of legal op-codes is 151. If you use a code that is not legal the computer will usually ignore it, though some illegal codes produce strange results. This is because there are some commands on the 6502 which are not documented because either they don't work properly or are totally useless.

If your computer has a 65C02 processor, which is a recent improvement on the old 6502 processor, then it will have 59 extra legal op-codes. The standard BBC Micro, however, has only the 151 standard 6502 op-codes. There is a complete list of the legal op-codes and what they do in Appendices B and C.

Addressing modes

Here is a list of all the addressing modes we can use and what data (if any) is needed for each.

Implied addressing

In this mode no explicit data is needed, so the command only uses the one byte for the op-code.

Immediate addressing

In this mode one byte of the data is needed. Thus the assembler command LDA #&7E becomes:

```
A9 7E
```

Note that the data is stored in the memory as the byte directly after the command byte.

Accumulator addressing

In this mode there is no explicit data involved. The A (for accumulator) after the command in assembly code is in fact implied by the op-code itself. The A is only for our convenience. Thus LSR A becomes:

```
4A
```

Absolute addressing

In this mode two bytes are needed after the op-code to specify the address of the byte the processor must work on. These address bytes are always stored with the low byte first, followed by the high byte. Thus STA &FE62 becomes:

```
8D 62 FE
```

Zero page addressing

If we wanted, for example, to store the accumulator at zero page address &78 using absolute addressing we would need:

```
8D 78 00
```

However, there is an addressing mode for just this sort of situation. By using ZERO PAGE ADDRESS-

ING the processor knows we are using zero page and we only have to send one byte of data to the processor. Thus the example above becomes:

```
85 78
```

This means that if we store all our frequently used variables in zero page we can access then slightly faster and with a saving of one byte of program per command. The assembler does this automatically − if it is faced with an absolute addressing command with an address in zero page, it automatically uses zero page addressing.

Absolute X addressing

In this mode the contents of the X register are added to the address before it is used. In terms of machine code the command is identical to absolute addressing but the op-code is different. Thus STA &6435,X becomes:

```
9D 35 64
```

Absolute Y addressing

This uses the Y register but is otherwise identical to absolute X addressing except that the op-code is different. For example LDA &900,Y becomes:

```
B9 00 09
```

Zero page X addressing

As with absolute addressing there is a zero page version of absolute X addressing. This is zero page X addressing and uses only one byte of data. This command automatically works in zero page. Thus LDA &78,X becomes:

```
85 7A
```

Zero page Y addressing

This is the same as zero page X addressing, only with a different op-code. For example STX &90,Y becomes:

```
96 90
```

Indirect addressing

This mode (which can only be used with the JMP instruction) has two bytes after it which together

form the address in which the processor looks for the actual address. Thus JMP (&230) becomes:

```
6C 30 02
```

Pre-indexed indirect addressing

In this mode a zero page address is specified so it only uses one byte of data. Thus ADC (&70,X) becomes:

```
61 70
```

Post-indexed indirect addressing

This again only needs a one-byte zero page address. Thus STA (&84),Y becomes:

```
91 84
```

Relative addressing

This mode is the most complicated of all. It needs one byte of data in the form of a positive or negative number. This is the number that is added to the program counter if the condition being tested is true. The program counter, of course, determines which command is being carried out or executed. The way to calculate this OFFSET, as it is called, is to subtract the address of the first byte of the command you want to branch to, from the address of the first byte of the next command after the branch command.

For example, take the assembly code program:

```
        LDX #&80
.loop   DEX
        BNE loop
        RTS
```

If we assembled it at address &2000 onwards and then looked at it we would find the following:

```
Address byte

  2000   A2  \ LDX #
  2001   80  \ &80
```

```
2002   CA  \ DEX
2003   D0  \ BNE relative
2004   FD  \ &2002-&2005=-3=&FD
2005   60  \ RTS
```

(a branch backwards)

```
        INC &70
        BNE nocarry
        INC &71
.nocarry RTS
```

becomes:

```
Address byte
  2000   E6  \ INC
  2001   70  \ &70
  2002   D0  \ BNE
  2003   02  \ &2006-&2004=2
  2004   E6  \ INC
  2005   71  \ &71
  2006   60  \ RTS
```

(a branch forwards).

Notice that the assembler OPT command has an option so that you can see an assembled listing of the code as it is assembled. This can be enabled by using OPT3 for the second pass. Below is the second of our example programs and the print-out it produces.

```
 10 HIMEM=&2000
 20 FOR pass%=0TO3 STEP3
 30 P%=&2000
 40 [OPT pass%
 50         INC &70
 60         BNE nocarry
 70         INC &71
 80 .nocarry RTS
 90 ]
100 NEXT
```

```
2000           OPT pass%
2000 E6 70     INC &70
2002 D0 02     BNE nocarry
2004 E6 71     INC &71
2006 60        .nocarry RTS
```

A machine code monitor

Now that we have seen how to pure machine code works, we need a way of using it. On the BBC Micro there is no quick way to look at a section of memory. If we want to look at a machine code program or write one, we need a 'window' into the memory. The program we need is called a machine code monitor. It is relatively easy to write a simple monitor in about 600 bytes, so that is exactly what we will do.

In this and the following chapters we will discuss each program in detail. Each section of the program will be, in general, followed by numbers in brackets that refer to line numbers in the full listing that follows the description.

This program will work in Mode 7. We need to be able to look at a whole screen of the memory, say 200 bytes, and not just one byte at a time. The best way to do this is to display the contents of the memory in hexadecimal as a table on the screen. So as to get a large number of bytes on the screen we will need to have eight bytes on each line of this table. We need to be able to see, at a glance, the address of each byte, though we can make do with only displaying the address of the first byte of each line. So we will end up with a display looking like:

```
78A0  01 20 E3 FF 20 E3 FF A9
78A8  00 A2 07 20 E3 FF CA D0
78B0  FA A9 04 A0 C0 4C F4 FF
78B8  C9 30 90 33 C9 3A 80 06
78C0  38 E9 30 4C D1 78 C9 41
78C8  90 25 C9 47 B0 21 38 E9
78D0  37 85 75 A4 74 B1 70 0A
78D8  0A 0A 0A 05 75 91 70 81
78E0  70 20 C2 79 A9 08 20 E3
78E8  FF 20 E3 FF 4C 61 78 C9
78F0  9F F0 19 C9 9C D0 63 20
```

```
78F8  23 7A C6 74 A5 74 C9 FF
7900  74 4C 0F 79 20 23 7A A9
7910  1F 20 E3 FF A9 00 20 E3
7918  FF 20 E3 FF A9 08 20 E3
7920  FF A5 70 38 E9 08 85 70
7928  A5 71 E9 00 85 71 A5 70
7930  38 E9 60 85 72 A5 71 E9
7938  00 85 73 20 DE 79 A9 1F
7940  20 E3 FF A9 00 20 E3 FF
7948  A9 18 20 E3 FF A9 20 A2
7950  27 20 E3 FF CA D0 FA 4C
7958  61 78 C9 9E F0 1D C9 32
```

We can also make the program more pleasant to use by making the addresses (at the extreme left) yellow and the data (the contents of the memory) green. We can then highlight the byte we are currently working on by making it white.

We are going to use the cursor keys and the shifted cursor keys for control of the cursor, so the first thing we need to do is set up the cursor keys to do this.

We could produce ASCII codes from the cursor keys using *FX4,1 but this will not distinguish between normal and shifted cursor keys. The way round this is to set up the cursor keys as soft keys and set them to generate ASCII codes. We do this with two commands. First, *FX4,2 sets up the cursor keys as soft keys 12 to 15. The shifted cursor keys will now automatically produce ASCII codes (from &8C to &8F) but the normal cursor keys will not. So, second, we make the normal cursor keys generate ASCII codes by setting the ASCII base of the normal function keys to &90 with *FX225,144. This means that instead of producing strings the soft keys will generate ASCII codes of &90 plus the key number, so that the normal cursor keys will now generate codes &9C to &9F. So the start of our machine code routine looks like this:

```
.monitor LDA #4
         LDX #2
         LDY #0
         JSR osbyte
```

```
        LDA #&E1
        LDX #&90
        JSR osbyte

(80-140)
```

Notice that we have used OSBYTE instead of &FFF4. To do this we must define the variable OSBYTE at the beginning of the assembler program. We are also going to use the OSASCI and OSRDCH routines so we can define these at the same time. We also need a place to put the machine code monitor program. For our purposes let's put it at &7900 and move HIMEM down to leave room for it. This will cause problems if you are writing a graphics program but it is easy, in that event, to change the program to assemble the machine code elsewhere in the memory.

So the beginning of the assembler routine looks like this:

```
10 HIMEM=&7900
20 osasci=&FFE3
30 osbyte=&FFF4
40 osrdch=&FFE0
50 FORpass%=0TO2STEP2
60 P%=&7900
70 [OPTpass%
```

Now back to the machine code. Our next task is to go into Mode 7 and turn the cursor off.

```
        LDA #22
        JSR osasci
        LDA #7
        JSR osasci
        LDA #23
        JSR osasci
        LDA #1
        JSR osasci
        LDA #0
        LDX #8
.loop1  JSR osasci
        DEX
```

```
BNE loop1
```

```
(150-270)
```

Notice that the simple BASIC command VDU23,1,0;0;0;0; becomes quite complicated in machine code. The eight zeros are easier to send with a loop unlike in BASIC.

Next we must decide what memory address we are interested in displaying. It is convenient to set this to &0000 for the moment as it is easy to step through the memory to the location we are interested in. Because of the way we are going to display the table, we are going to be treating the memory as an array eight bytes across by 8192 bytes down.

It will be easier if we keep the address of the FIRST byte on the line we are looking at separate from the number of the byte (0-7) within that line.

Let's say &70 and &71 contain the address of the first byte on the line where the cursor is at present positioned and &74 contains the number of the byte on that line which we are interested in (0-7). Initially the address of the first byte on the line will be zero, so we can add to our program:

```
LDA #0
STA &70
STA &71
STA &74
```

```
(280-310)
```

Next we must display a screenful of memory − 24 lines of eight bytes each. It would be sensible to set up a routine which just prints one line (eight bytes) of memory, and use this repeatedly. However, before we can even do this we need a routine that will display the value of one byte as two hex digits. For this routine let us specify that the byte to be printed must initially be found in the accumulator. We will have to work on one nibble (half a byte) at a time, so we have to save the complete byte while we work on the first nibble in the accumulator. We can save the accumulator in &75. Next we can

mask out the least significant nibble, leaving the most significant nibble in the accumulator (this will be the left-hand digit of the hex byte). We will then have to shift it right four times so that we get a number from zero to fifteen in the low nibble of the accumulator.

```
.byte    STA &75      \ Display byte
         AND #&F0     \ in hex.
         LSR A
         LSR A
         LSR A
         LSR A

(1990-2040)
```

Next we need to display this digit on the screen. We will need to do this twice (left-hand nibble and right-hand nibble) for each byte so we need a separate routine called NIBBLE to do this.

Having called this routine we must reload the accumulator with the original byte and this time mask out the most significant nibble, leaving the right-hand nibble in the least significant nibble of the accumulator, and call the nibble routine again. However, note there is little point in calling NIBBLE again as, once it is called, the BYTE routine will have finished so the next command would be RTS. We may as well 'fall through' straight to NIBBLE and let the RTS at its end (supplied by the OS subroutine OSASCI) do the job. This means that we have to place NIBBLE directly in place of the second JSR NIBBLE command. This leaves us with:

```
         JSR nibble
         LDA &75
         AND #&F
.nibble  ...

(2050-2080)
```

The nibble routine must add 48 (ASCII code for 0) to the number in the accumulator before printing it using OSASCI. However, if the number is 10 or more

(decimal) then we need first to add a further seven to bring it to the corresponding ASCII codes for the characters A, B, C, D, E and F.

```
.nibble  CLC
         ADC #48
         CMP #58
         BCC print
         ADC #7
.print   JSR osasci
         RTS
```

Notice that because the last-but-one command of NIBBLE is a JSR we can instead just jump to OSASCI and the RTS command at its end will save us from needing an extra RTS at the end of NIBBLE. Thus the end of NIBBLE becomes:

```
.print   JMP osasci
```

```
(2080-2140)
```

We now have the byte display routine; so, next, we need to write the line display routine. For this we need the address of the first byte on the line. This may not necessarily be the line the cursor is on, so we can't use &70 and &71. Instead we can specify that the address of the first byte on the line must be stored at &72 and &73. The routine must first of all print a 'yellow' teletext code for the address. Then it must print the two-byte address stored in &72 and &73 (remember that &73 is the high byte).

```
.line    LDA #&83
         JSR osasci
         LDA &73
         JSR byte
         LDA &72
         JSR byte
```

```
(2160-2210)
```

Next we need a space to separate the address from the data.

```
        LDA #32
        JSR osasci
```

(2220-2230)

We will next use POST-INDEXED INDIRECT ADDRESSING to load the byte to be displayed into the accumulator; so we need to set Y to zero for the first byte. Then, for each byte, we can print a 'green' teletext code to separate the byte from the previous one; then load then byte into the accumulator and display it; then increment Y; and repeat the process until all eight bytes that make up the line have been displayed. After that we only need a carriage return to complete this routine. As before, we can save ourselves from putting an RTS at the end by jumping to the OSASCI routine.

```
        LDY #0
.loop3  LDA #&82
        JSR osasci
        LDA (&72),Y
        JSR byte
        INY
        CPY #8
        BNE loop3
        LDA #13
        JMP osasci
```

(2240-2330)

Having written a routine for displaying a line, we can now go back to the main routine and display a whole screenful of data. For this overall display, it would be best if the byte we are currently examining or altering always appears half-way down the screen as then we can see what we have done and what is coming. For this reason the line the cursor is on will always be the thirteenth line down. Thus to print the block of memory above the cursor we need to subtract 96 (12 times 8) from the contents of &70 and &71. This we can put into &72 and &73 ready to display a line. We also need to start at the top of

the screen (later we will jump back to this point so the change of mode is not sufficient).

```
.display LDA #30     \ Display sect of
         JSR osasci  \ memory as table 8
         LDA &&0     \ bytes of 24
         SEC         \ lines.
         SBC #96
         STA &72
         LDA &71
         SBC #0
         STA &73

(320-400)
```

We are going to print 24 lines in one go, so we can use the X register to count down from 24 to 1. We also need to add 8 to the contents of &72 and &73 to move the address forward by eight bytes after each line.

```
         LDX #24
.loop2   JSR line
         LDA &72
         CLC
         ADC #8
         STA &72
         LDA &73
         ADC #0
         STA &73
         DEX
         BNE loop2

(410-510)
```

We are now at the stage where we need a cursor to appear. We are going to highlight the byte we are interested in by making it white. To do this we need to put a ‘white’ teletext code before it and a ‘green’ teletext code after it. As we are going to want to remove this cursor again, it would be sensible to use a subroutine to position the text cursor where we are going to place the ‘white’ byte. We shall call this routine CURSOR1. We know that the cursor will

always be on line 12 so we only have to calculate how far across it will be. As each byte uses up three screen characters for its display we need to multiply the byte number by three. To do this we need to load the accumulator with the contents of &74, shift the accumulator left one bit to multiply it by two, and then add the contents of &74 to the accumulator to make three times the original number. We then have to add six to the accumulator to shift the cursor right past the address at the beginning of the line. We can use VDU31,x,y to move the text cursor on the screen. So the routine looks like:

```
.cursor1 LDA #31     \ Move text cursor
         JSR osasci  \ to position
         LDA &74     \ for editing
         ASL A       \ cursor
         CLC
         ADC &74
         CLC
         ADC #6
         JSR osasci
         LDA #12
         JMP osasci

(2350-2450)
```

We can now go back to the main routine. Firstly we need to call CURSOR1 and then we need to print a 'white'. As we have used 'green' codes to separate the bytes we don't need to put another one in. The text cursor is then on the first nibble of the byte.

```
.start   JSR cursor1
         LDA #&87
         JSR osasci
```

We now need to get a key from the keyboard using OSRDCH. If ESCAPE has been pressed we must acknowledge it with OSBYTE &7E and we then want to turn the cursor back on with VDU23,1,1,0;0;0; reset the cursor keys to their normal functions with *FX4,0 and move the cursor to the bottom of the screen using VDU31, all before

returning to BASIC.

```
.key    JSR osrdch
        BCC key1
        LDA #&7E
        JSR osbyte
        LDA #23
        JSR osasci
        LDA #1
        JSR osasci
        JSR osasci
        LDA #0
        LDX #7
.loop4  JSR osasci
        DEX
        BNE loop4    \ Note that X must now
        LDA #4       \ be zero, so we don't
        LDY #0       \ need to set it for
        JSR osbyte   \ the OSBYTE call
        LDA #31
        JSR osasci
        LDA #0
        JSR osasci
        LDA #24
        JMP osasci
```

(550-770)

Having checked for the ESCAPE key we need to see if the byte under the cursor is being altered. If the key pressed is either 0 to 9 or A to F then we must alter the byte accordingly. Firstly we can check if the code is less than 48. If so, then we must check for other keys.

```
.key1   CMP #48
        BCC key2
```

(780-790)

Next we can look to see if the code is less than 58. If so, then a number key has been pressed and we need to subtract 48 to get the value of the new nibble.

```
        CMP #58
        BCS letter
        SEC
        SBC #48
        JMP hex
```

(800-840)

If not, we then want to look for a letter. If the code is less than 65 we are not interested and must wait for another key. If it is larger than or equal to 71 then we must look to see if it is a cursor key. Otherwise we want to subtract 55 to get the value of the new nibble.

```
.letter CMP #65
        BCC key
        CMP #71
        BCS key2
        SEC
        SBC #55
```

(850-900)

We now have to decide what to do with the nibble. Probably the best way to input the byte is for each new nibble to shift the old byte left one nibble. Thus the high nibble is lost, the low nibble becomes the high nibble and the nibble typed in replaces the low nibble. To do this we have to temporarily store the nibble we have typed in while we shift the memory byte left four bits. Then we can OR in the new nibble and store the result back in memory.

```
.hex    STA &75
        LDY &74
        LDA (&70),Y
        ASL A
        ASL A
        ASL A
        ASL A
        ORA &75
        STA (&70),Y
```

```
        LDA (&70),Y
```

(910-990)

This program will allow us to look into the ROMs but if we try to store an alteration back into a ROM nothing will happen. To make sure that the user doesn't think something has happened it would be sensible to load the byte back again before storing it on the screen. This way, if the byte hasn't been altered then the displayed byte on the screen won't change. As we have left the text cursor at the first nibble on the screen we can just call the subroutine BYTE to display the byte, moving the text cursor back again and go back to waiting for the next key.

```
        JSR byte
        LDA #8
        JSR osasci
        JSR osasci
        JMP key
```

(1000-1050)

If the key pressed is not a number or a letter we must check whether it is a cursor key and act accordingly. We shall check first of all a cursor-up. If this has been pressed we shall branch to the relevant routine. Otherwise we shall check for a cursor-left.

```
.key2   CMP #&9F
        BEQ up
        CMP #&9C
        BNE key3
```

(1060-1090)

Having established that the cursor-left key has been pressed we need to remove the highlight cursor. We will have to do this several times so we need a subroutine which we shall call CURSOR. This must first call CURSOR1 to position the text cursor and

then rub over the highlight teletext code with a 'green' code.

```
.cursor  JSR cursor1
         LDA #&82
         JMP osasci

(2470-2490)
```

So our cursor-left routine can now remove the cursor. Next it must decrement &74 to move the address of the byte being looked at, back by one. If this is now 255 then we must set it to seven (the end of the line) and do a cursor up. Otherwise we can go back to the start of the main routine. Notice that as this last branch is more than 128 bytes we have to use the 'skip and jump' technique.

```
         JSR cursor
         DEC &74
         LDA &74
         CMP #255
         BEQ skip1
         JMP start
.skip1   LDA #7
         STA &74
         JMP up1

(1100-1180)
```

While we are at it, we can write the UP routine as well. This will be the same as UP1 but with a call to CURSOR in front of it.

```
.up      JSR cursor
.up1     ...

(1190)
```

Next we have to scroll the screen down one line. We can do this by first moving the cursor to the top of the screen using VDU30 and doing a cursor-up.

```
.up1     LDA #30
```

```
JSR osasci
LDA #11
JSR osasci
```

(1200-1230)

Next we have to subtract eight from &70 and &71 to move the cursor line back by one.

```
LDA &70
SEC
SBC #8
STA &70
LDA &71
SBC #0
STA &71
```

(1240-1300)

Next we must call LINE with the address of the top line in &72 and &73. This will be the contents of &70 and &71 minus 96.

```
LDA &70
SEC
SBC #96
STA &72
LDA &71
SBC #0
STA &73
JSR line
```

(1310-1380)

Lastly we need to clear the bottom line of the screen by moving to the bottom line and printing 31 spaces.

```
LDA #31
JSR osasci
LDA #0
JSR osasci
LDA #24
JSR osasci
```

```
          LDA #32
          LDX #31
.loop%    JSR osasci
          DEX
          BNE loop5
          JMP start
```

(1390-1500)

Next we must check for cursor-down. If this has been pressed then we must jump to the relevant routine.

```
.key3     CMP #&9E
          BEQ down
```

(1510-1520)

Otherwise we must check for cursor-right. If you are typing a long program it will be annoying to have to find the cursor-right key between typing in each byte, so we shall allow both the cursor-right key and the space bar to do the same job.

```
          CMP #&9D
          BEQ right
          CMP #32
          BNE key4
```

(1510-1520)

The RIGHT routine must first remove the old cursor and then increment &74 to move the cursor right by one byte. If the contents of &74 now equal eight then we must set it back to zero and jump to the cursor-down routine. Otherwise, we must use the 'skip and jump' technique to branch back to START.

```
.right    JSR cursor
          INC &74
          LDA &74
          CMP #8
          BEQ skip2
```

```
        JMP start
.skip2  LDA #0
        STA &74
        JMP down1

(1570-1650)
```

As with the UP routine, we can insert the DOWN routine here.

```
.down   JSR cursor
.down1  ...

(1660)
```

Next we have to scroll the screen up a line. We can do this by moving to the bottom of the screen and doing a cursor-down. However, at the same time we can print the new bottom line. This is because we are leaving a blank line at the bottom of the screen. By printing the new line in this blank space the carriage return at the end will scroll the screen for us. First we need to move to this blank line.

```
.down1  LDA #31
        JSR osasci
        LDA #0
        JSR osasci
        LDA #24
        JSR osasci

(1670-1720)
```

Now we add eight to the address of the cursor line.

```
        LDA &70
        CLC
        ADC #8
        STA &70
        LDA &71
        ADC #0
        STA &71

(1730-1790)
```

Next we must store the address of the new bottom line in &72 and &73 and call LINE. Then we can jump back to START.

```
        LDA &70
        CLC
        ADC #88
        STA &72
        LDA &71
        ADC #0
        STA &73
        JSR line
        JMP start
```

(1800-1880)

We now have all we need. However, if you want to look at the contents of address &7F00 you will have to scroll through &0000 to &7F00 one line at a time and this will be slightly tedious. To solve this problem we shall make shifted cursor-up and shifted cursor-down move a page at a time through the memory. This we can do by incremented or decremented the high byte of the address of the cursor (line &71) and jumping back to DISPLAY.

```
.key4   CMP #&8E
        BNE key5
        INC &71
        JMP display
.key5   CMP #&8F
        BNE key6
        DEC &71
        JMP display
```

(1890-1960)

Finally, if the key that has been pressed is not one we are interested in then we must go back and wait for another key to be pressed.

```
.key6   JMP key
```

This program is a very simple one. There are several ROMs available that have more sophisticated monitors in them. BBC MONITOR (ROM based) from BBC Publications (1985) is one such. Another package from the same publishers is TOOLBOX 2 by Ian Trackman (1985), which comprises a book and software tape. It has a particularly interesting implementation of a monitor among its many utilities, and complements this book.

Of course, you can improve our monitor program. However, it will provide a useful tool for those people who can't be bothered to write a better version or buy a ROM. It will also give you valuable experience in how assembly code programs work.

Here, then, is the complete listing of the monitor program. You might like to use the command *SAVE MONITOR 7900 +20B 7900 to save the machine code once it is assembled. Then the monitor can be used by typing *MONITOR. This will take up less room on a disc or tape and will be faster to load. However, you should keep a copy of the source assembly code program so that you can assemble the program into different places, if necessary, by changing P% at line 60.

You do not need to type in the comments on the right-hand side of the listing.

```
 10 HIMEM=&7900
 20 osasci=&FFE3
 30 osbyte=&FFF4
 40 osrdch=&FFE0
 50 FORpass%=0TO2STEP2
 60 P%=&7900
 70 [OPTpass%
 80 .monitor LDA #4      \ Main program
 90          LDX #2      \ initialisation.
100          LDY #0
110          JSR osbyte
120          LDA #&E1
130          LDX #&90
```

```
140          JSR osbyte
150          LDA #22
160          JSR osasci
170          LDA #7
180          JSR osasci
190          LDA #23
200          JSR osasci
210          LDA #1
220          JSR osasci
230          LDA #0
240          LDX #8
250 .loop1   JSR osasci
260          DEX
270          BNE loop1
280          LDA #0
290          STA &70
300          STA &71
310          STA &74
320 .display LDA #30     \ Display sect of
330          JSR osasci  \ memory as table 8
340          LDA &70     \ bytes of 24
350          SEC         \ lines.
360          SBC #96
370          STA &72
380          LDA &71
390          SBC #0
400          STA &73
410          LDX #24
420 .loop2   JSR line
430          LDA &72
440          CLC
450          ADC #8
460          STA &72
470          LDA &73
480          ADC #0
490          STA &73
500          DEX
510          BNE loop2
520 .start   JSR cursor1 \ Start checking
530          LDA #&87    \ keys
540          JSR osasci
550 .key     JSR osrdch
560          BCC key1    \ Check for
570          LDA #&7E    \ ESCAPE.
```

```
 580          JSR osbyte
 590          LDA #23
 600          JSR osasci
 610          LDA #1
 620          JSR osasci
 630          JSR osasci
 640          LDA #0
 650          LDX #7
 660 .loop4   JSR osasci
 670          DEX
 680          BNE loop4
 690          LDA #4
 700          LDY #0
 710          JSR osbyte
 720          LDA #31
 730          JSR osasci
 740          LDA #0
 750          JSR osasci
 760          LDA #24
 770          JMP osasci
 780 .key1    CMP #48     \ Check for byte
 790          BCC key2    \ being altered.
 800          CMP #58
 810          BCS letter
 820          SEC
 830          SBC #48
 840          JMP hex
 850 .letter  CMP #65
 860          BCC key
 870          CMP #71
 880          BCS key2
 890          SEC
 900          SBC #55
 910 .hex     STA &75
 920          LDY &74
 930          LDA (&70),Y
 940          ASL A
 950          ASL A
 960          ASL A
 970          ASL A
 980          ORA &75
 990          STA (&70),Y
1000          LDA (&70),Y
1010          JSR byte
```

```
1020          LDA #8
1030          JSR osasci
1040          JSR osasci
1050          JMP key
1060 .key2    CMP #&9F    \ Check for
1070          BEQ up      \ cursor-up.
1080          CMP #&9C    \ Check for
1090          BNE key3    \ cursor-left.
1100          JSR cursor
1110          DEC &74
1120          LDA &74
1130          CMP #255
1140          BEQ skip1
1150          JMP start
1160 .skip1   LDA #7
1170          STA &74
1180          JMP up1
1190 .up      JSR cursor
1200 .up1     LDA #30
1210          JSR osasci
1220          LDA #11
1230          JSR osasci
1240          LDA &70
1250          SEC
1260          SBC #8
1270          STA &70
1280          LDA &71
1290          SBC #0
1300          STA &71
1310          LDA &70
1320          SEC
1330          SBC #96
1340          STA &72
1350          LDA &71
1360          SBC #0
1370          STA &73
1380          JSR line
1390          LDA #31
1400          JSR osasci
1410          LDA #0
1420          JSR osasci
1430          LDA #24
1440          JSR osasci
1450          LDA #32
```

```
1460          LDX #31
1470 .loop5   JSR osasci
1480          DEX
1490          BNE loop5
1500          JMP start
1510 .key3    CMP #&9E    \ Check for
1520          BEQ down    \ cursor-down.
1530          CMP #&9D    \ Check for
1540          BEQ right   \ cursor-right
1550          CMP #32
1560          BNE key4
1570 .right   JSR cursor
1580          INC &74
1590          LDA &74
1600          CMP #8
1610          BEQ skip2
1620          JMP start
1630 .skip2   LDA #0
1640          STA &74
1650          JMP down1
1660 .down    JSR cursor
1670 .down1   LDA #31
1680          JSR osasci
1690          LDA #0
1700          JSR osasci
1710          LDA #24
1720          JSR osasci
1730          LDA &70
1740          CLC
1750          ADC #8
1760          STA &70
1770          LDA &71
1780          ADC #0
1790          STA &71
1800          LDA &70
1810          CLC
1820          ADC #88
1830          STA &72
1840          LDA &71
1850          ADC #0
1860          STA &73
1870          JSR line
1880          JMP start
1890 .key4    CMP #&8E    \ Check for
```

```
1900          BNE key5
1910          INC &71      \ cursor-down.
1920          JMP display
1930 .key5    CMP #&8F     \ Check for
1940          BNE key6     \ shifted
1950          DEC &71      \ cursor-up.
1960          JMP display
1970 .key6    JMP key
1980
1990 .byte    STA &75      \ Display byte
2000          AND #&F0     \ in hex.
2010          LSR A
2020          LSR A
2030          LSR A
2040          LSR A
2050          JSR nibble
2060          LDA &75
2070          AND #&F
2080 .nibble  CLC          \ Display nibble
2090          ADC #48      \ in hex.
2100          CMP #58
2110          BCC print
2120          CLC
2130          ADC #7
2140 .print   JMP osasci
2150
2160 .line    LDA #&83     \ Display line
2170          JSR osasci   \ of table.
2180          LDA &73
2190          JSR byte
2200          LDA &72
2210          JSR byte
2220          LDA #32
2230          JSR osasci
2240          LDY #0
2250 .loop3   LDA #&82
2260          JSR osasci
2270          LDA (&72),Y
2280          JSR byte
2290          INY
2300          CPY #8
2310          BNE loop3
2320          LDA #13
2330          JMP osasci
```

```
2340
2350 .cursor1 LDA #31     \ Move text cursor
2360          JSR osasci  \ to position
2370          LDA &74     \ for editing
2380          ASL A       \ cursor
2390          CLC
2400          ADC &74
2410          CLC
2420          ADC #6
2430          JSR osasci
2440          LDA #12
2450          JMP osasci
2460
2470 .cursor  JSR cursor1 \ Remove editing
2480          LDA #&82    \ cursor.
2490          JMP osasci
2500 ]
2510 NEXT
2520 CALL monitor
```