The BBC Micro Machine Code Porfolio

Other books for BBC Micro users

Introducing the BBC Micro
Ian Sinclair
0 00 383072 1

The BBC Micro: An Expert Guide
Mike James
0 246 12014 2

Discovering BBC Micro Machine Code
A. P. Stephenson
0 246 12160 2

Advanced Machine Code Techniques for the BBC Micro
A. P. Stephenson and D. J. Stephenson
0 246 12227 7

BBC Micro Graphics and Sound
Steve Money
0 246 12156 4

Practical Programs for the BBC Micro
Owen Bishop and Audrey Bishop
0 246 12405 9

21 Games for the BBC Micro
Mike James, S. M. Gee and Kay Ewbank
0 246 12103 3

Disk Systems for the BBC Micro
lan Sinclair
0 246 12325 7

Learning is Fun -
40 Educational Games for the BBC Micro
Vince Apps
0 246 12317 6

Advanced Programming for the BBC Micro
Mike James and S. M. Gee
0 00 383073 X

Take Off with the Electron and BBC Micro
Audrey Bishop and Owen Bishop
0 246 12356 7

Creative Animation and Graphics for the BBC Micro
Mike James
0 00 383007 1

Handbook of Procedures and Functions for the BBC Micro
Audrey Bishop and Owen Bishop
0 245 12415 6

The BBC Micro Machine Code Portfolio

Bruce Smith

DIGITALLY REMASTERED ON ACORN RISC OS COMPUTERS,
JANUARY 2007

Contents

Acknowledgements

Disks and cassettes of the programs in this book are available. Apply for details to:

> Dept AB
> Collins Professional and Technical Books
> 8 Grafton Street
> London W1X 3LA

Chapter One
Introduction

The BBC Micro Machine Code Portfolio is aimed at providing the serious machine code programmer with an interesting and useful set of assembly language routines. In all, a selection of 75 programs from my own disk-based library are included, ranging from general purpose utility aids to BASIC and machine code programming to specific utilities that could form the basis of an interesting machine code compiler. Each of the assembler routines is provided as a uniquely line numbered procedure that can be *EXECuted back into a program any time it is required. Although the programs are written making full use of BASIC II's EQU functions, a solution is provided towards the end of this chapter that will enable BASIC I users to implement these functions with the absolute minimum of fuss.

The Portfolio is not aimed at the beginner. Many of the routines assume a rudimentary knowledge of the manipulative techniques involved. For this reason, the descriptive commentary of, say, a multibyte addition program may be kept to the looping and data manipulation means used rather than the principles of the actual addition itself. This does not mean to say that the Portfolio is presented with experts in mind - far from it. My own knowledge of machine code programming was facilitated by continuous trial and error programming, and at that time no books of this sort were available to help me along the way. Enter the program and use it as described in the text; in using the program and experimenting with it the real knowledge will be gained.

The contents of each chapter have been grouped together for ease of use. Chapters 2 to 9 provide the programming utilities which can make the programmer's life a merry one. These include function key definition printers, program variable dumps, a global search and replace utility and a program compacter.

Chapter 10 provides the mathematically biased routines. Four multibyte routines handle addition, subtraction, multiplication and division. The remaining seven routines provide square root solutions and dual byte shifts.

Graphics are dealt with in Chapter 11, with just about every BASIC-type graphics command covered. This includes a useful routine using the

interpreter-based *640 multiplication table to convert an X,Y screen coordinate into an absolute screen address. The graphics routines are particularly easy to group into a BASIC driven menu to provide a simple graphics compiler (SGC).

All programs require data manipulation at some time and Chapter 2 supplies eleven procedures to sort, add, and delete items from a variety of lists and arrays. Screen interaction is important in making programs user-friendly, and Chapter 13 provides the routines to enable this to be performed with ease. Finally, Chapter 14 has grouped a few miscellaneous procedures together providing timing delays, counters and interrupt polling.

The Appendix contains a number of the Portfolio programs in a new form - as bar code listings. Using the MEP bar code reader these may be read in directly from the pages of the book itself!

The correct procedure

As mentioned, each of the assembler routines are implemented as PROCedures, with each having its own range of line numbers. The procedure contains all the necessary coding to make the routine a stand-alone one. Because a procedure must be called from a program, each program contains several lines of BASIC to call first the PROC to assemble the machine code it contains and then demonstrate the type of application possible. This serves two functions; first, it assures you that you have entered the program correctly and also shows you how it works! If you flick through a few of the programs you'll notice that the lines of the PROC are given high line numbers in steps of 1 while the assemble and test routines use low line numbers. This is quite deliberate as it keeps the two parts of the program clearly separated.

To build up a library of these programs to disk or tape, the PROCedure can be save to tape. As the low line numbers are only, in effect, test routines these can be deleted so that only the procedure remains; and it is this that should be saved. I prefer to save my PROCedures as ASCII files rather than programs as this allows them to be added easily to other programs. The *SPOOL command is used to perform this. Choosing a suitable filename, the syntax is simply:

*SPOOL NAME

If you are using the cassette filing system then you'll need to start the cassette running. If you are using disks then these will already be whirling around. So, now simply LIST the program. As the listing appears onto the screen it will also be written to the current filing

system. When the program has finished listing enter

*SPOOL

once again to complete the transfer from memory to filing medium. You will probably have noticed that the filename used in the *SPOOL command was not enclosed within the quotes normally associated with a SAVE. This is acceptable as the MOS does not expect them - though using them will have no adverse effect.

Once all programs have been treated in this way they are ready to be used in a greater scheme of things! One point - when building up a large library of procedures it is very important to catalogue them, in a book, on another disk or tape, or on the front of each tape or disk. This catalogue should depict the program's name, line numbers and function as this will be invaluable when it comes to using them at a later date.

Once a program has been saved as a spooled file it can be loaded back into memory using the MOS-based *EXEC command. To load the previously spooled file you use

*EXEC NAME

Again, the quotes around the filename are not obligatory. When the operating system encounters the file it treats each line of it as though it had been typed in at the keyboard and as the return character at the end of the line is reached the line is entered into memory. This is the main reason for using unique line numbers within procedures, as it enables several procedures to be *EXECuted into memory without fear of overwriting any program lines already there.

A demonstration

To show the flexibility of the programs within the Portfolio and their use, study the following demonstration. Suppose a short graphics routine is required that will select a MODE 4 screen and draw a dotted line from the coordinate 200,200 to 900,600. First, the three desired files to perform a MODE, MOVE and PLOT must be loaded in (these can be found in Chapter 13). Depending on the filenames you have chosen, this might take the form of executing the following commands one by one:

*EXEC mode
*EXEC move
*EXEC plot

The resultant listing forms part of Program 1.1. Next, a BASIC primer needs to be written to call each PROC and pass the relevant information through the arguments of the procedural call. First, PROCmode:

20 PROCmode (4,&A96)

The PROCedure is called passing the mode number, 4 into the variable 'action' and the assembly address, &A99, into 'addr'. Next, the graphics cursor must be moved. The problem here is that we do not really want to have to calculate the new value to be assigned to 'addr' for the code assembly; instead we can simply use the program counter itself in the form of P%. Thus line 30 becomes

30 PROCmove (200,200,P%)

The move coordinates are 200,200 and the PROCmove code is assembled from P%. Finally PROCplot can be treated in the same way to give

40 PROCplot (21,900,600,P%)

where 21 is the plot code for an absolute dotted line, 900,600 the final coordinates and P% the assembly address. Now each PROCedure will assemble its code as a subroutine call. To implement the machine code, a short procedure must be written that will call each subroutine in turn, thus:

```
JSR mode   \ set up MODE
JSR move   \ move graphics cursor
JMP plot   \ draw line and return
```

Program 1.1 lists the final program and, by way of proof of the output, Figure 1.1 lists the assembler listing produced when RUN.

When using this modular-cum-structured assembly approach, the use of the OPT command must be borne in mind. If the OPT command is omitted then the default value of 3 will be assumed by the assembler. In the case of the above example this was not too much of a problem, but there are occasions when it will be! For example, if assembly is performed on a conditional basis then it may be desirable to suppress it altogether using OPT 2 lest it corrupt some vital screen detail. Alternatively, a FOR . . . NEXT combination may be imperative to suppress errors during a first pass to assign forward branch labels. There is no simple way around this. A universal solution would be to include a

FOR pass=0 TO 2 STEP 2

line in all procedures. I prefer to add the OPT commands as required, but the choice is yours.

```
  10 REM *** USING THE PORTFOLIO ***
  20 PROCmode (4,&A00)
  30 PROCmove (200,200,P%)
  40 PROCplot (21,900,600,P%)
  50 PROCassemble(P%)
  60 CALL test
  70 END
  80 :
 100 DEF PROCassemble (addr)
 110 P%=addr
 120 [
 130 .test
 140 JSR mode
 150 JSR move
 160 JMP plot
 170 ]
 180 ENDPROC
 200 :
6000 DEF PROCmode (action,addr)
6001 P%=addr
6002 [
6003 .mode
6004 LDA #22
6005 JSR &FFEE
6006 LDA #action
6007 JSR &FFEE
6008 RTS
6009 ]
6010 ENDPROC
6180 DEF PROCmove(xpos,ypos,addr)
6181 P%=addr
6182 [
6183 .move
6184 LDA #25
6185 JSR &FFEE
6186 LDA #4
6187 JSR &FFEE
6188 LDA #xpos MOD 256
6189 JSR &FFEE
6190 LDA #xpos DIV 256
6191 JSR &FFEE
6192 LDA #ypos MOD 256
6193 JSR &FFEE
6194 LDA #ypos DIV 256
6195 JSR &FFEE
6196 RTS
```

```
6197 ]
6198 ENDPROC
6220 DEF PROCplot(code,xcoord,ycoord,ad
dr)
6221 P%=addr
6222 [
6223 .plot
6224 LDA #25
6225 JSR &FFEE
6226 LDA #code
6227 JSR &FFEE
6228 LDA #xcoord MOD 256
6229 JSR &FFEE
6230 LDA #xcoord DIV 256
6231 JSR &FFEE
6232 LDA #ycoord MOD 256
6233 JSR &FFEE
6234 LDA #ycoord DIV256
6235 JSR &FFEE
6236 RTS
6237 ]
6238 ENDPROC
```

Program 1.1. Spooling procedures to form a graphics program

```
>RUN
0A00
0A00 .mode
0A00 A9 16    LDA #22
0A02 20 EE FF JSR &FFEE
0A05 A9 04    LDA #action
0A07 20 EE FF JSR &FFEE
0A0A 60       RTS
0A0B
0A0B .move
0A0B A9 19    LDA #25
0A0D 20 EE FF JSR &FFEE
0A10 A9 04    LDA #4
0A12 20 EE FF JSR &FFEE
0A15 A9 C8    LDA #xpos MOD 256
0A17 20 EE FF JSR &FFEE
0A1A A9 00    LDA #xpos DIV 256
0A1C 20 EE FF JSR &FFEE
0A1F A9 C8    LDA #ypos MOD 256
0A21 20 EE FF JSR &FFEE
0A24 A9 00    LDA #ypos DIV 256
0A26 20 EE FF JSR &FFEE
0A29 60       RTS
0A2A
0A2A .plot
0A2A A9 19    LDA #25
0A2C 20 EE FF JSR &FFEE
0A2F A9 15    LDA #code
```

```
0A31 20 EE FF JSR &FFEE
0A34 A9 84    LDA #xcoord MOD 256
0A36 20 EE FF JSR &FFEE
0A39 A9 03    LDA #xcoord DIV 256
0A3B 20 EE FF JSR &FFEE
0A3E A9 58    LDA #ycoord MOD 256
0A40 20 EE FF JSR &FFEE
0A43 A9 02    LDA #ycoord DIV256
0A45 20 EE FF JSR &FFEE
0A48 60 RTS
0A49
0A49 .test
0A49 20 00 0A JSR mode
0A4C 20 0B 0A JSR move
0A4F 4C 2A 0A JMP plot
```

Fig. 1.1. Assembler listing produced by Program 1.1.

The BASIC solution

BASIC II provides several enhancements over its predecessor BASIC I. The most useful of these are the EQU functions which are implemented as pseudo-opcodes. These functions and their operations are:

EQUB : assemble specified byte
EQUW : assemble specified word (2 bytes)
EQUD : assemble specified double word (4 bytes)
EQUS : assemble specified string as ASCII characters

Numerous programs in the Portfolio take advantage of these commands, which would therefore make them inoperable on Beebs with BASIC I. These commands can be simulated quite simply using the ability of the FN command.

Program 1.2 lists the function definitions plus a suitable demonstration. Taking each definition as it appears in the program, FNequs (lines 500 to 530) uses the program counter variable P% as the string argument for the ASCII character string passed into the function via 'strings'. Before exit, P% is incremented by the length of the string.

FNequb (lines 550 to 580) takes the value 'byte%' and simply pokes it into memory at P%. The program counter is incremented by one and completes. FNequw (lines 600 to 640) is an extension and provides two pokes at the position of P%. The high and low bytes are extracted from 'word' using the MOD and DIV operators. Finally, FNequd (lines 660 to 690) uses the word indirection operator to pling its four bytes into memory.

```
 10 REM ** SIMULATING BASIC II EQU **
 20 P%=&900
 30 [
 40 LDA #255
 50 OPT FNequs("TEST",3)
 60 LDX #0
 70 OPT FNequb(6,3)
 80 LDY #&33
 90 OPT FNequw(&FFFF,3)
100 STX &70
110 OPT FNequd(&12345678,3)
120 LDX #&AA
130 RTS
140 ]
150 END
160 :
500 DEF FNequs(string$,opt)
510 $P%=string$
520 P%=P%+LEN(string$)
530 =opt
540 :
550 DEF FNequb(byte%,opt)
560 ?P%=byte%
570 P%=P%+1
580 =opt
590 :
600 DEF FNequw(word%,opt)
610 ?P%=word% MOD 256
620 P%?1=word% DIV 256
630 P%=P%+2
640 =opt
650 :
660 DEF FNequd(double%,opt)
670 !P%=double%
680 P%=P%+4
690 =opt
```

Program 1.2. Simulating the BASIC II EQU functions in BASIC I.

The assembler text (lines 40 to 130) shows how each procedure should be called. The second parameter in each of the OPT FN calls (3 throughout) simply refers to the OPT selection and this should be seeded as required by the program. To end with, Figure 1.2 shows the assembler listing provided when running this program, while the hex dump in Figure 1.3 shows that each FN has indeed performed the required task.

```
>RUN
0900
0900 A9 FF    LDA #255
0906          OPT FNequs("TEST",3)
```

```
0906 A2 00   LDX #0
0909        OPT FNequb(6,3)
0909 A0 33 LDY #&33
090D        OPT FNequw(&FFFF,3)
090D 86 70 STX &70
0913        OPT FNequd(&12345678,3)
0913 A2 AA LDX #&AA
0915 60    RTS
```

Fig. 1.2. Assembler listing produced by Program 1.2

```
900 A9
901 FF
902 54
903 45
904 53
905 54
906 A2
907 0
908 6
909 A0
90A 33
90B FF
90C FF
90D 86
90E 70
90F 78
910 56
911 34
912 12
913 A2
914 AA
915 60
```

Fig. 1.3. A hex dump of the code assembled by Program 1.2, showing that
the functions have worked.

Chapter Two
Function Key Reader

Virtually all the toolbox type of commercial ROM packages around these days include a facility for printing any resident function key definitions. Many, though, are incomplete and only deal with keys 0 through to 10, neglecting keys 11 to 15. Writing a custom-built routine to handle printing definitions present in all sixteen function keys is a relatively easy task providing a working knowledge of the function key buffer is to hand. Two programs are presented here; the first is reprinted from the April 1984 edition of Acorn User while the second is an improved version. The two differ in that the former, Program 2.1, is not capable of printing multistatement single key definitions whereas the latter, Program 2.2, is. The advantage in using the former is the saving in memory overheads as it requires only half the memory space required by the latter.

Key	Pointer Byte
0	&B00
1	&B01
2	&B02
3	&B03
4	&B04
5	&B05
6	&B06
7	&B07
8	&B08
9	&B09
10	&B0A
11	&B0B
12	&B0C
13	&B0D
14	&B0E
15	&B0F

Fig. 2.1. Function key associated bytes.

The function key buffer is located in page &B of block zero RAM occupying the bytes &B99 to &BFF inclusive. With the exception of the first seventeen bytes, all of this is used to hold the key definitions in ASCII format; commands are not tokenised. These first seventeen bytes, &B99 to &B 10, are the key pointer bytes and Figure 2.1 details the bytes associated with the individual keys. Monitoring each of these bytes as key definitions are entered, modified and deleted gives an insight into their purpose. Figure 2.2 is a hex dump of the start of the buffer after a hard break, either when you have switched on or the CTRL-BREAK sequence is carried out. At this stage the buffer contains nothing but &10 in every byte.

```
0B00 10 10 10 10 10 10 10 10 ........
0B08 10 10 10 10 10 10 10 10 ........
0B10 10 10 10 10 10 10 10 10 ........
0B18 10 10 10 10 10 10 10 10 ........
0B20 10 10 10 10 10 10 10 10 ........
0B28 10 10 10 10 10 10 10 10 ........
0B30 10 10 10 10 10 10 10 10 ........
0038 10 10 10 10 10 10 10 10 ........
```

Fig 2.2. Key buffer after switch-on.

Figure 2.3 depicts the same area of the key buffer after a short definition has been entered into f0 thus:

   *KEY 0 CLS |M

```
0B00 10 14 14 14 14 14 14 14 ........
0B08 14 14 14 14 14 14 14 14 ........
0B10 14 43 4C 53 0D 10 10 10 .CLS....
0B18 10 10 10 10 10 10 10 10 ........
0B20 10 10 10 10 10 10 10 10 ........
0B28 10 10 10 10 10 10 10 10 ........
0B30 10 10 10 10 10 10 10 10 ........
0B38 10 10 10 10 10 10 10 10 ........
```

Fig 2.3. Key buffer after executing *KEY0 CLS|M.

The dump shows that the ASCII string CLS is present but that the return sequence '|M' has been replaced with the more conventional ASCII return character &0D. It is also obvious from the dump that the key pointer bytes have altered. The first byte at &B00 is, we know from Figure 2.1, associated with *KEY9, and this byte still contains &10 or 16 decimal. Counting sixteen bytes from this location we arrive at the

first character in the *KEY 0 definition. The remaining key pointer bytes
now all contain &14 or 20 decimal; counting 20 bytes from the *KEY 0
pointer brings us to the last byte; of the *KEY 0 definition, the carriage
return character at &B14.

Figure 2.4 shows the buffer after a further key has been defined, thus:

   *KEY 9 AUTO|M

```
0B00 10 19 19 19 19 19 19 19 ........
0B08 19 14 19 19 19 19 19 19 ........
0B10 19 43 4C 53 0D 41 55 54 .CLS.AUT
0B18 4F 0D 10 10 10 10 10 10 O.......
0B20 10 10 10 10 10 10 10 10 ........
0B28 10 10 10 10 10 10 10 10 ........
0B30 10 10 10 10 10 10 10 10 ........
0B38 10 10 10 10 10 10 10 10 ........
```

Fig. 2.4. Key buffer after executing *KEY9 AUTO|M.

The ASCII characters of the new definition are entered into the buffer
immediately after the last definition. The key pointer byte for *KEY 9 at
&B09 still contains & 14 while the *KEY9 byte remains at &10. All the
other key pointer bytes have been updated to hold &19 or 25 decimal.
Starting from &B00 and counting 25 bytes brings us to &B19, the last
byte defined in the buffer.

It is worth looking at what happens in the buffer if a function key is
redefined. Figure 2.5 shows the effect of placing a longer definition into
*KEY 0 than was already present, thus:

   *KEY 0 VDU 7|M

What has happened now is that the previous *KEY0 definition has been
deleted, the remaining definition(s) shuffled up to the front of the buffer
and the new *KEY0 definition added onto the end. Each of the key
pointer bytes have been adjusted to point to the correct location.

```
0B00 15 1B 1B 1B 1B 1B 1B 1B ........
0B08 1B 10 1B 1B 1B 1B 1B 1B ........
0B10 1B 41 55 54 4F 0D 56 44 .AUTO.VD
0B18 55 20 37 0D 10 10 10 10 U 7.....
0B20 10 10 10 10 10 10 10 10 ........
0B28 10 10 10 10 10 10 10 10 ........
0B30 10 10 10 10 10 10 10 10 ........
0B38 10 10 10 10 10 10 10 10 ........
```

Fig. 2.5. Key buffer after redefining f0 as *KEY0 VDU 7|M

*KEY 9 was defined as AUTO and the pointer byte at &B09 now holds &10 giving the offset from &B00 to the start of the definition. *KEY 0 which is now tacked onto the end of the *KEY 9 definition has had its pointer offset reset to &15 or 21 decimal. The remaining pointer bytes have also been adjusted to all give the correct offset to the last used byte in the buffer, &1B or 27 decimal, which when added to &B00 gives &B1B.

The key pointer area contains an extra 'general' byte at &B10 that we have not yet mentioned. This byte is, in fact, the TOP pointer in the buffer and always holds the byte offset into the buffer of the last used location. The MOS uses this byte to test if a *KEY definition is present when a function key has been defined. If the key pointer byte and the TOP pointer byte are the same, the MOS inserts the definition directly after the last definition (as pointed to by the pointer and TOP bytes). If, on the other hand, the pointer byte and TOP byte are different, the MOS knows that a definition is already present for the function key just defined and that it must do some reshuffling of the bytes in the buffer.

Program 2.1

Program 2.1 is the first of the function key definition printer programs. Called PROCkeys1, it occupies 63 lines between 1000 and 1063. All processor registers are used and the object code occupies 126 bytes anywhere in memory as specified by the variable 'addr'.

```
  10 REM *** FUNCTION KEY PRINTER ***
  20 REM *(C) Bruce Smith & Acorn User*
  30 oswrch=&FFEE
  40 osasci=&FFE3
  50 PROCkeys1 (&C00)
  60 *KEY0 CALL &C00|M
  70 END
  80 :
1000 DEF PROCkeys1(addr)
1001 LOCAL key, pointer
1002 key=&B00
1003 pointer=&B10
1004 FOR pass=0 TO 3 STEP3
1005 P%=addr
1006 [OPT pass
1007 LDX #0
1008            .main_loop
```

```
1009              TXA
1010              ASL A
1011              TAX
1012              JSR print_word_key
1013              LDA number_table,X
1014              JSR oswrch
1015              LDA number_table+1,X
1016              JSR oswrch
1017              TXA
1018              LSR A
1019              TAX
1020              LDA #&20
1021              JSR oswrch
1022              LDA key,X
1023              CMP pointer
1024              BNE over
1025              JMP update
1026 .over
1027              TAY
1028              INY
1029 .next_character
1030              LDA key,Y
1031              CMP #13
1032              BEQ carriage_return
1033              JSR oswrch
1034              INY
1035              BNE next_character
1036 .carriage_return
1037              LDA #ASC"|"
1038              JSR oswrch
1039              LDA #ASC"M"
1040              JSR oswrch
1041 .update
1042              LDA#13
1043              JSR osasci
1044              INX
1045              CPX#16
1046              BNEmain_loop
1047              RTS
1048 .print_word_key
1049              LDY#6
1050 .next_letter
1051              LDA spell_key,Y
1052              JSR oswrch
1053              DEY
1054              BNE next_letter
1055              RTS
1056 .number_table
1057              EQUS" 0 1 2 3 4 5 6 7 "
1058              EQUS"8 9101112131415"
1059 .spell_key
1060              EQUS " YEK* "
1061 ]
```

```
1062 NEXT
1063 ENDPROC
```

Program 2.1. PROCkeys1 - a simple function key lister

Converting the hard facts of Function Key Buffer operation into some suitable machine code is relatively easy and the main areas of coding are straightforward. From this we can see the main routines of the code which, in everyday terms, are a follows:

(a) Print the string '*KEY' followed by the current key number (therefore we need a key counter!).
(b) Obtain the key pointer and if the same as the TOP pointer move onto the next function key.
(c) Else increment key pointer and print the definition string until the RETURN character is found.
(d) Print |M and do a RETURN.
(e) Increment the function key counter.
(f) Repeat the whole process until all sixteen keys have been printed.

The X register is used to keep a count of the current key being investigated so initially this is set to zero (line 1007). The register is also used as an index into the key number look-up table (lines 1056 to 1058) where each of the ASCII codes {key number occupies two bytes. To ensure that the correct offset is located, the key count must be multiplied by two using an arithmetic shift left (lines 1009 to 10ll). The word '*KEY' and the current key number are printed using the subroutine calls of lines 1012 to 1016.
   A definition present test is performed in line 1022 and 1023 and a branch over executed (line 1024) if one is found. A simple index, extract and print routine is used to print the definition string (lines 1036 to 1040) before the key count in the index register is updated (lines 1041 to 1047).

Program 2.2

Program 2.2 is, in essence, the same as Program 2.1 but extra coding has been incorporated into the listing to test for multiple definitions and control codes. The procedure is called PROCkeys2 and assumes line numbers 1070 to 1189 inclusive. The source code generates 246 bytes of hex assembled at 'addr'.

```
  10 REM *FUNCTION KEY DEFINITIONS V2*
  20 PROCkeys2(&A00)
  30 *KEY0 CALL&A00|M
  40 END
  50 :
1070 DEF PROCkeys2 (addr)
1071 FOR pass=0 TO 3 STEP3
1072 P%=addr
1073 [
1074 OPT pass
1075 .entry
1076           LDA #0
1077           STA key
1078           STA offset
1079 .mainloop
1080           JSR &FFE7
1081           JSR printwordkey
1082           LDX key
1083           LDA numbertable,X
1084           INX
1085           JSR &FFEE
1086           LDA numbertable,X
1087           JSR &FFEE
1088           INX
1089           STX key
1090           LDA #32
1091           JSR &FFEE
1092           LDX offset
1093           LDA &B00,X
1094           STA keystart
1095           INC keystart
1096           LDA &B10
1097           STA endpointer
1098           LDX #&F
1099 .keyend
1100           LDA &B00,X
1101           CMP endpointer
1102           BCS nexttry
1103           CMP keystart
1104           BCC nexttry
1105           STA endpointer
1106 .nexttry
1107           DEX
1108           BPL keyend
1109           LDA endpointer
1110           CMP keystart
1111           BCC nextkey
1112           LDX keystart
1113 .printdef
1114           LDA &B00,X
1115           CMP #128
1116           BCC asciichr
1117           PHA
```

```
1118              LDA #ASC"|"
1119              JSR &FFEE
1120              LDA #ASC"!"
1121              JSR &FFEE
1122              PLA
1123              AND #&7F
1124 .asciichr
1125              CMP #32
1126              BCS notcontrol
1127              PHA
1128              LDA #ASC"|"
1129              JSR &FFEE
1130              PLA
1131              CLC
1132              ADC #64
1133              JSR &FFEE
1134              JMP nextcharacter
1135 .notcontrol
1136              CMP #127
1137              BNE over
1138              LDA #ASC"|"
1139              JSR &FFEE
1140              LDA #ASC"?"
1141              JSR &FFEE
1142              JMP nextcharacter
1143 .over
1144              CMP#124
1145              BNE not
1146              LDA #ASC"|"
1147              JSR &FFEE
1148              JSR &FFEE
1149              JMP nextcharacter
1150 .not
1151              JSR &FFEE
1152 .nextcharacter
1153              CPX endpointer
1154              BEQ nextkey
1155              INX
1156              JMP printdef
1157 .nextkey
1158              INC offset
1159              LDA offset
1160              CMP #16
1161              BNE notfinished
1162              JSR &FFE7
1163              RTS
1164 .notfinished
1165              JMP mainloop
1166              .printwordkey
1167              LDY #6
1168              .nextletter
1169              LDA spellkey,Y
```

```
1170            JSR &FFEE
1171            DEY
1172            BNE nextletter
1173            RTS
1174 .numbertable
1175            EQUS" 0 1 2 3 4 5 6 7 "
1176            EQUS"8 9101112131415"
1177 .spellkey
1178            EQUS" YEK* "
1179 .key
1180            EQUB 0
1181 .keystart
1182            EQUB 0
1183 .endpointer
1184            EQUB 0
1185 .offset
1186            EQUB 0
1187 ]
1188 NEXT
1189 ENDPROC
```

Program 2.2. PROCkeys2 - the complete function key lister.

The definition printing routine (line 1113) begins by testing the
definition for a character code greater than 128 (entered previously with
the '|!' sequence). If one is present this sequence is printed; either way,
control progresses to line 1124 where a control character (less than &32)
is tested for. If a control character is found, the '|' character is printed
followed by the ASCII code of the control character representation,
obtained by adding &40 to it. Thus CTRL-L is printed as '|L' .

The end_pointer bytes are used to keep track of the length of the
entered key definition as previously calculated by the key_end coding
(lines 1099 to 1112), and the print_def loop continues until the entire
function key definition is printed. The main_loop is executed sixteen
times to print all the function key definitions. The final output of this
program is shown in Figure 2.6.

```
*KEY 0 CALL&A00|M
*KEY 1 CLS|M
*KEY 2 *GREPL|M
*KEY 3 LIST|M
*KEY 4 *ASSFORM|M
*KEY 5 *INSPECT
*KEY 6 *BASFORM|M
*KEY 7 FORN=&70 TO &7F:P.?N:N.|M
*KEY 8 *EXMON|M
*KEY 9 *BASIC|M
*KEY 10 OLD|MLIST|M
```

```
*KEY 11
*KEY 12
*KEY 13
*KEY 14
*KEY 15
```

Fig. 2.6. Typical output of Program 2.2.

Program fact sheets

Function key printers

Program 2.1

Proc title	: PROCkeysl
Line numbers	: 1000 to 1063
Variables required	: addr
Length	: 126 bytes
Zero page requirements	: none
Registers changed	: A, X, Y

Program 2.2

Procedure title	: PROCkeys2
Line numbers	: 1070 to 1189
Variables required	: addr
Length	: 246 bytes
Zero page requirements	: none
Registers changed	: A, X, Y

Chapter Three
Program Information

Two programming utilities are provided in this chapter. Program 3.1 lists the status of the various BASIC pseudo-variables in addition to displaying the length of the program currently under development and the number of bytes remaining available for use. To complement this, Program 3.2 when called will list every variable currently defined within a BASIC program (except the resident integer variables A% to Z%), and this includes assembler labels. This is particularly useful in long programs when it is difficult to keep a mental track of the variable names you have already chosen and thus avoids the infuriating situation that can occur when you use the same variable name twice and wonder why the program will just not work as it should!

```
  10 REM *** PROGRAM INFORMATION ***
  20 himem=HIMEM
  30 himem=himem-&200
  40 HIMEM=himem
  50 PROCinfo (&70,HIMEM)
  60 *KEY0 CALL HIMEM|M
  70 END
  80 :
1200          DEF PROCinfo (current,addr)
1201          FOR pass=0 TO 3 STEP3
1202          P%=addr
1203          [
1204          OPT pass
1205          LDX #title MOD 256
1206          LDY #title DIV 256
1207          JSR print_message
1208 .do_page
1209          LDX #message1 MOD 256
1210          LDY #message1 DIV 256
1211          JSR print_message
1212          LDA &18
1213          JSR hex_out
1214          LDA #0
1215          JSR hex_out
1216          JSR &FFE7
1217 .do_top
1218          LDX #message4 MOD 256
```

```
1219              LDY #message4 DIV 256
1220              JSR print_message
1221              LDA &13
1222              JSR hex_out
1223              LDA &12
1224              JSR hex_out
1225              JSR &FFE7
1226 .do_himem
1227              JSR &FFE7
1228              LDX #message2 MOD 256
1229              LDY #message2 DIV 256
1230              JSR print_message
1231              LDA &7
1232              JSR hex_out
1233              LDA &6
1234              JSR hex_out
1235              JSR &FFE7
1236 .do_lomem
1237              LDX #message3 MOD 256
1238              LDY #message3 DIV 256
1239              JSR print_message
1240              LDA &1
1241              JSR hex_out
1242              LDA &0
1243              JSR hex_out
1244              JSR &FFE7
1245              JSR &FFE7
1246 .do_size
1247              LDX #message5 MOD 256
1248              LDY #message5 DIV 256
1249              JSR print_message
1250              SEC
1251              LDA &13
1252              SBC &18
1253              JSR hex_out
1254              LDA &12
1255              JSR hex_out
1256              LDX #bytes MOD 256
1257              LDY #bytes DIV 256
1258              JSR print_message
1259 .do_next_free
1260              LDX #message6 MOD 256
1261              LDY #message6 DIV 256
1262              JSR print_message
1263              LDA &3
1264              JSR hex_out
1265              LDA &2
1266              JSR hex_out
1267              JSR &FFE7
1268              JSR &FFE7
1269 .memory_left
1270              LDX #message7 MOD 256
```

```
1271              LDY #message7 DIV 256
1272              JSR print_message
1273              SEC
1274              LDA &6
1275              SBC &2
1276              STA store
1277              LDA &7
1278              SBC &3
1279              JSR hex_out
1280              LDA store
1281              JSR hex_out
1282              LDX #bytes MOD256
1283              LDY #bytes DIV 256
1284              JSR print_message
1285              RTS
1286 .print_message
1287              STX current
1288              STY current+1
1289              LDY #0
1290 .loop
1291              LDA (current),Y
1292              BMI all_done
1293              JSR &FFE3
1294              INY
1295              BNE loop
1296 .all_done
1297              RTS
1298 .hex_out
1299              PHA
1300              LSR A
1301              LSR A
1302              LSR A
1303              LSR A
1304              SED
1305              CLC
1306              ADC #&90
1307              ADC #&40
1308              CLD
1309              JSR &FFEE
1310              PLA
1311              AND #15
1312              SED
1313              CLC
1314              ADC #&90
1315              ADC #&40
1316              CLD
1317              JMP &FFEE
1318 .title EQUB 12
1319              EQUS" Program"
1320              EQUS" Information Service"
1321              EQUD &0D0D0D0D
1322              EQUB 255
1323 .message1
```

```
1324            EQUS"PAGE : &"
1325            EQUB 255
1326 .message2
1327            EQUS"HIMEM : &"
1328            EQUB 255
1329 .message3
1330            EQUS"LOMEM : &"
1331            EQUB 255
1332 .message4
1333            EQUS"TOP : &"
1334            EQUB 255
1335 .message5
1336            EQUS"Program Size=&"
1337            EQUB 255
1338 .message6
1339            EQUS"Next Free Location=&"
1340            EQUB 255
1341 .message7
1342            EQUS"Memory Remaining=&"
1343            EQUB 255
1344 .bytes
1345            EQUS" bytes"
1346            EQUD &0D0D
1347            EQUB 255
1348 .store
1349            EQUB 0
1350            ]
1351            NEXT
1352            ENDPROC
```

Program 3.1. PROCinfo - provides details on system pseudo-variables.

Program status

PROCinfo is the assembler procedure to generate the source code for the info program. I have given the name 'info' to the procedure simply because it is more representative to the program's function. I would suggest, however, that it is saved to tape or disk with the filename STATUS. This is because INFO is generally recognised as a disk filing system command and therefore the command *INFO could not be used to load and run the program from disk whereas *STATUS would be acceptable. When executed, the program prints the hexadecimal values of the following:

PAGE
HIMEM
LOMEM

TOP
Program size
Next free location
Memory remaining

All the information required to calculate each of these values can be found in zero page. Figure 3.1 lists the byte allocation for the first couple of dozen locations.

The assembler is quite straightforward and is split into easy-to-handle segments. The screen information title is first printed onto the screen using the print_message subroutine (lines 1286 to 1297). The address of the string to be printed is transferred to the subroutine via the index registers. On return, the relevant data is extracted from zero page and printed in hexadecimal format using a fairly standard hex to ASCII print routine, 'hex_out' (lines 1298 and 1317).

&00 - &01	: LOMEM
&02 - &03	: VARTOP (top of variables)
&04 - &05	: Basic Stack Pointer
&06 - &07	: HIMEM
&08 - &09	: ERL
&0A	: Text pointer index
&9B - &9C	: Text pointer
&0D - &11	: RND seed
&12 - &13	: TOP
&16 - &17	: Error vector
&18	: PAGE byte

Fig. 3.1. Assignment of first 24 zero page bytes.

The values of PAGE, TOP, HIMEM, LOMEM and the next free location can be obtained directly from the BASIC workspace. The other values must be calculated, which generally involves a simple two-byte subtraction. Program size is calculated by subtracting TOP from PAGE and the amount of memory remaining by subtracting the top of variables (termed VARTOP by me!) from HIMEM. The actual value of VARTOP is not displayed by the program but could be simply added if so required.

The 'hex_out' routine works four bits at a time. Taking the high nibble first (as this is the first printed working left to right) and moving this into the low nibble, the conversion is performed using decimal addition with the decimal flag set with SED. The decimal addition of &90 converts the binary values 0 to 9 into the range &90 to &99 with the carry flag set.

The addition of a further &40 converts these values to the range &30 to &39 with the carry set, which corresponds to the correct ASCII codes for the values 0 to 9. If the original nibble held &A to &F, adding &90 gives values in the range &0 to &5 (remember we are working with decimal addition). Addition of a further &40 with the carry set gives a final result in the range &41 to &46, the ASCII codes for A to F. The low nibble is treated in the same manner to produce the second digit before the decimal flag is cleared. Using STATUS is straightforward: just perform a CALL to the assembly address. The BASIC primer generates the 372 bytes of code above a lowered HIMEM and can be called using function key 0. Figure 3.2 shows a typical output of the machine code.

```
     Program Information Service

PAGE  : &1C00
TOP   : &2559

HIMEM : &7A00
LOMEM : &2559

Program Size=&0959 bytes

Next Free Location=&26C7

Memory Remaining=&5339 bytes
```

Fig. 3.2. Typical of output produced by Program 3.2.

Variable lister

PROCvars generates a useful variable lister that occupies a compact 103 bytes of memory, the cassette / RS 423 buffer in the demonstration. Five bytes of workspace are required in addition, and two bytes of these must be in zero page to facilitate indirect addressing.

```
  10 REM *** LIST ALL PROGRAM VARIABLES
***
  20 PROCvars(&70,&71,&73,&A00)
  30 *KEY 1 CALL&A00|M
  40 END
  50 :
1400 DEF PROCvars(asc,varpointer,varstr
ing,addr)
1401 FOR pass=0 TO 3 STEP 3
1402 P%=addr
```

```
1403 [ OPT pass
1404             .variables
1405             LDA #12
1406             JSR &FFEE
1407             LDA #14
1408             JSR &FFEE
1409             LDA #65
1410             STA asc
1411             LDA #&82
1412             STA varpointer
1413             LDA #4
1414             STA varpointer+1
1415 .loop
1416             LDY #1
1417             LDA (varpointer),Y
1418             BEQ update
1419             STA varstring+1
1420             LDY #0
1421             LDA (varpointer),Y
1422             STA varstring
1423 .next_var
1424             LDA #13
1425             JSR &FFE3
1426             LDA asc
1427             JSR &FFE3
1428             LDY #2
1429 .print_loop
1430             LDA ( varstring),Y
1431             BEQ end_print
1432             JSR&FFE3
1433             INY
1434             JMP print_loop
1435 .end_print
1436             LDY #1
1437             LDA ( varstring),Y
1438             BEQ update
1439             TAX
1440             DEY
1441             LDA (varstring),Y
1442             STA varstring
1443             STX varstring+1
1444             JMP next_var
1445 .update
1446             LDA #2
1447             CLC
1448             ADC varpointer
1449             CMP #&F6
1450             BEQ finished
1451             STA varpointer
1452             INC asc
1453             JMP loop
1454 .finished
1455             LDA #13
```

```
1456              JSR &FFE3
1457              LDA #15
1458              JSR &FFE3
1459              RTS
1460              ]
1461              NEXT
1462              ENDPROC
```

Program 3.2. PROCvars - lists all program variables.

An understanding of variable storage is essential to follow the program's operation. In addition to the resident integer variables there are basically two other types of variable. One of these variables is postfixed with a % sign to signify that it is also an integer, while a variable without the % defines that it is a floating point variable. When a program is run, the BASIC interpreter extracts each variable from the program and places it in a fixed format above the main program and below TOP. The format is as follows:

(a) A two-byte address which points to the next variable starting with the same leiter. If none are present these bytes contain zero.
(b) The variable name in ASCII format excluding the first letter of the variable, e.g. START is stored as TART.
(c) A zero byte to mark the end of the variable name.
(d) The binary representation of the value assigned to that variable. This is stored in four bytes for an integer variable and five bytes for a floating point variable.

We can see from item (a) that it is quite easy to move from one variable to another, starting with the same letter, simply by extracting the address pointer from each variable 'definition' in turn. However, we need to know exactly where the first variable is located and Acorn have provided, by design, a variable pointer table on Page 4 in block zero RAM. Figure 3.3 details the locations holding the pointers for the characters A to Z and a to z. If both locations for a particular character contain zero then no variable beginning with that letter is present. 28 The BBC Micro Machine Code Portfolio

Character	LSB address	MSB address
A	&482	&483
B	&484	&485
C	&486	&487
D	&488	&489

E	&48A	&48B
F	&48C	&48D
G	&48E	&48F
H	&490	&491
I	&492	&493
J	&494	&495
K	&496	&497
L	&498	&499
M	&49A	&49B
N	&49C	&49D
O	&49E	&49F
P	&4A9	&4A1
Q	&4A2	&4A3
R	&4A4	&4A5
S	&4A6	&4A7
T	&4A8	&4A9
U	&4AA	&4AB
V	&4AC	&4AD
W	&4AE	&4AF
X	&4B9	&4B1
Y	&4B2	&4B3
Z	&4B4	&4B5
a	&4C2	&4C3
b	&4C4	&4C5
c	&4C6	&4C7
d	&4C8	&4C9
e	&4CA	&4CB
f	&4CC	&4CD
g	&4CE	&4CF
h	&4D0	&4D1
i	&4D2	&4D3
j	&4D4	&4D5
k	&4D6	&4D7
l	&4D8	&4D9
m	&4DA	&4DB
n	&4DC	&4DD
o	&4DE	&4DF
p	&4E9	&4E1
q	&4E2	&4E3
r	&4E4	&4E5
s	&4E6	&4E7
t	&4E8	&4E9
u	&4EA	&4EB
v	&4EC	&4ED

w	&4EE	&4EF
x	&4F0	&4F1
y	&4F2	&4F3
z	&4F4	&4F5

Fig. 3.3. Variable start pointers

Program lowdown

Figure 3.4 flowcharts the program's operation. The first ten lines of assembler clear the screen, place it into paged mode, save the ASCII code for A in 'asc' and seed the variable pointer table start address, &482, into a zero page vector.

The main program loop is entered at line 1415 and commences by extracting the most significant byte from the pointer table. For a variable to be present, this bye must be non-zero as no variables can be placed in zero page. If it is zero a branch to update is performed, otherwise the low byte address is accessed and seeded into a second vector, pointer.

Lines 1423 to 1427 print a carriage return followed by the first character of the variable saved in asc. Using post-indexed indirect addressing, the print_loop (lines 1429 to 1434) extract each variable character from the program workspace, printing each until the zero terminating byte is encountered.

The linking address from the beginning of the variable definition is then sought. If this is zero a branch to update is performed, otherwise the link address is placed into the pointer vector and the next variable name printed.
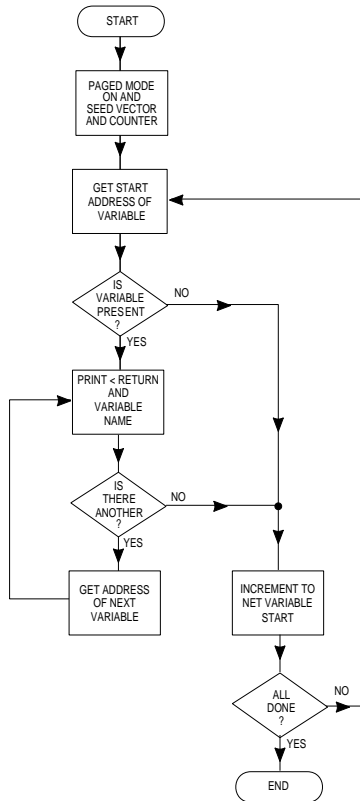
Fig. 3.4. PROCvars flowchart.

The update routine (lines 1445 to 1453) first increments the var_pointer vector by two to move onto the next character associated bytes, and increments the character value, asc, by one. The program terminates when the last location in the variable pointer table is reached (line 1449 and 1450). Finally, Figure 3.5 illustrates a typical output ofthe program, listing the variables in the program itself!

```
asc
addr
end_print
finished
loop
next_var
pass
```

```
print_loop
update
varpointer
varstring
variables
```

Fig. 3.5. Typical output produced by Program 3.3.

Program fact sheets

Program 3.1

Procedure title	: PROCinfo
Line numbers	: 1200 to 1352
Variables required	: current,addr
Length	: 372 bytes
Zero page requirements	: 2 bytes (anywhere in memory)
Registers changed	: A, X, Y

Program 3.2

Procedure title	: PROCvars
Line numbers	: 1400 to 1462
Length	: 103 bytes
Zero page requirements	: 5 bytes, four forming vectors
Registers changed	: A, X, Y

Chapter Four
Program Formatters

BASIC's LISTO command allows a limited amount of control in producing formatted listings, inserting spaces to indent loops and structures as required. The two programs presented in this chapter provide an extended formatting option for either BASIC or assembler programs; indeed, the Assembler Formatter was used to produce the clear listing within this book, inserting ten spaces between line number and mnemonic but leaving labels un-indented and clearly separated from the listing.

```
 10 REM * A Basic Formatted Listing *
 20 FOR loop=0 TO 100
 30 PRINT loop : NEXT loop
 40 INPUT "A number" N%
 50 IF N%=10 PRINT"Correct" ELSE PRINT
"wrong"
 60 REPEAT : INPUT "Code" C$
 70 FOR wait=0 TO 1000 : NEXT wait
 80 UNTIL C$="END"

>LIST

 10 REM * A Basic Formatted Listing *
 20 FOR loop=0 TO 100
 30   PRINT loop
     : NEXT loop
 40 INPUT "A number" N%
 50 IF N%=10 PRINT"Correct"
     ELSE PRINT "wrong"
 60 REPEAT
     : INPUT "Code" C$
 70  FOR wait=0 TO 1000
     : NEXT wait
 80   UNTIL C$="END"
```

Fig. 4.1. A BASIC listing with and without the BASIC formatter

The BASIC formatter splits multistatement lines by issuing a carriage return each time it encounters a colon. It also splits IF. . .THEN. . .ELSE structures in addition to indenting them along with

REPEAT . . . UNTIL and FOR. . .NEXT loops. Figure 4. 1 shows the type of listing the BASIC Formatter is capable of. Now for the programs!

The BASIC Formatter (Program 4.1)

The basic_format procedure assembles its machine code into Page 9 of block zero RAM. This area has a number of uses (in addition to housing our machine code) and is more normally associated with ENVELOPEs 5-16, the speech buffer, cassette and RS 423 buffer. The routine has two entry points - &900 and &928 in this case - and function keys I and 2 have been programmed to call these locations. These two entries simply turn the formatter on and off respectively.

The 'on' entry point (line 1485) first prints the formatter on message before storing the current value of LISTO, found at &1F, in a byte above the program. Its maximum value of 7 is then inserted. The WRCHV vector contents are extracted and saved and the WRCHV pointed to the *format' entry point at line 1521. The 'off' entry, line 1506, simply reverses these procedures. Line 1518 could be changed if required to make the formatter clear the LISTO option each time it is switched off by replacing it with

LDA #0

```
  10 REM *** LISTING FORMATTER ***
  20 PROCbasic_format(&900)
  30 *KEY0 CALL &900|M
  40 *KEY1 CALL &928|M
  50 END
  60 :
1480 DEF PROCbasic_format(addr)
1481 interpreter=&E0A4
1482 FOR pass=0 TO 3 STEP3
1483 P%=addr
1484 [OPT pass
1485 .on
1486          LDX #&00
1487 .next_character
1488          LDA message,X
1489          JSR &FFE3
1490          INX
1491          CMP#13
1492          BNE next_character
1493          LDA &1F
```

```
1494               STA listo
1495               LDX #&07
1496               STX &1F
1497               LDA &20E
1498               STA address
1499               LDA &20F
1500               STA address+1
1501               LDA #format MOD 256
1502               STA &20E
1503               LDA #format DIV 256
1504               STA &20F
1505               RTS
1506 .off
1507               LDX #&00
1508 .next_character
1509               LDA message2,X
1510               JSR &FFE3
1511               INX
1512               CMP #13
1513               BNE next_character
1514               LDA address
1515               STA &20E
1516               LDA address+1
1517               STA &20F
1518               LDA listo
1519               STA &1F
1520               RTS
1521 .format
1522               PHA
1523               CMP #ASC(":")
1524               BNE no_colon
1525               JSR output
1526               LDA #&00
1527               STA byte
1528               STA byte+1
1529               BEQ not_else
1530 .no_colon
1531               LDA #&01
1532               CMP &1E
1533               BNE not_same
1534               LDA #&00
1535               STA byte+2
1536               STA byte+3
1537               STA byte+4
1538 .not_same
1539               CPY #&00
1540               BEQ carry_on
1541 .not_else
1542               PLA
1543               JMP interpreter
1544 .carry_on
1545               LDA &37
1546               CMP #&E7
```

```
1547            BNE not_if
1548            INC byte+2
1549 .not_if
1550            CMP #&8B
1551            BNE not_else
1552            INC byte+3
1553            JSR output
1554            JSR interpreter
1555            JMP not_else
1556 .output
1557            LDA #&0A
1558            JSR interpreter
1559            LDA #&0D
1560            JSR interpreter
1561            CLC
1562            LDA &3B
1563            ADC &3C
1564            ADC byte+2
1565            TAX
1566            INX
1567            LDA #&20
1568            JSR interpreter
1569            JSR interpreter
1570            JSR interpreter
1571 .more_spaces
1572            JSR interpreter
1573            JSR interpreter
1574            DEX
1575            BNE more_spaces
1576            RTS
1577 .message
1578            EQUS" Formatter on!"
1579            EQUB 7
1580            EQUB 13
1581 .message2
1582            EQUS " Formatter off!"
1583            EQUB 7
1584            EQUB 13
1585 .byte
1586            EQUS" "
1587 .address
1588            EQUS" "
1589 .listo EQUB 0
1590            ]
1591            NEXT pass
1592            ENDPROC
```

Program 4.1. PROCbasic_format - neatly formats a BASIC listing.

On entry into 'format' ,through the reset WRCHV the accumulator contains the character to be written. This is tested to see if it is a colon. If this test fails a branch to 'no_colon' is performed. Assuming a colon is present, the 'output' routine at line 1556 is called to perform a line feed and carriage return and a series of spaces printed. The output routine uses a direct jump into the MOS to do the printing. This is necessary as we have intercepted the normal WRCHV address.

Incidently, disassembling from this address, &E9A4, provides an interesting insight into how the Beeb programs the CRTC to display characters. As a machine code programmer you must be in possession of a suitable disassembler, so have a look! But I digress, so back to the program description. On return from the output call (line 1526) the 'byte' locations, which act as counters, are cleared and a forced branch to 'not_else' is performed, which prints the character to the screen (line 1541).

Routing around the rest of the code takes place if any of the indenting loop commands already mentioned are identified by intercepting the count value held at &1E and used by LISTO. Special treatment of the IF statement is required to ensure that any subsequent ELSE is generated both on a new line and further indented - this is because ELSE is normally ignored by LISTO. These two commands are identified by their token values before the 'interpreter' call hands them over to the BASIC detokenising routine for expansion. The codes and the entry points are as follows:

    IF   (= &E7) entry at line 1546
    ELSE   (= &8B) entry at line 1550

The byte at &37 is used by the BASIC interpreter to hold the current command token (line 1545).

The Assembler Formatter (Program 4.2)

This utility is probably the one I use most of all along with the global search and replace utility presented in Chapter 7. I find that the neatest way to present assembler listings is in the manner used throughout this book; the mnemonics are indented and clearly and follow through. The most obvious way to perform this task is simply to tap in spaces as required at the keyboard as the program is entered, but this is boring, time-consuming and extremely wasteful of memory which can be of a premium in the hi-res graphics modes. Thus, Program 4.2 was conceived.

```
  10 REM *** ASSEMBLER FORMATTER ***
  20 PROCass_format (&C00)
  30 *KEY0 CALL &C00|M
  40 *KEY1 CALL &C29|M
  50 END
  60 :
1600 DEF PROCass_format (addr)
1601 oswrch=?&20E+(?&20F*256)
1602 FOR pass=0 TO 3 STEP 3
1603 P%=addr
1604 [
1605          OPT pass
1606 .on
1607          LDX #0
1608 .nextchr
1609          LDA message,X
1610          JSR &FFE3
1611          INX
1612          CMP #13
1613          BNE nextchr
1614          LDA#0
1615          STA byte +1
1616          LDA &20E
1617          STA byte+2
1618          LDA &20F
1619          STA byte+3
1620          LDA #assembler MOD 256
1621          STA &20E
1622          LDA #assembler DIV 256
1623          STA &20F
1624          RTS
1625 .off
1626          LDX #0
1627 .nextchr
1628          LDA message2,X
1629          JSR &FFE3
1630          INX
1631          CMP #13
1632          BNE nextchr
1633          LDA byte+2
1634          STA &20E
1635          LDA byte+3
1636          STA &20F
1637          RTS
1638 .assembler
1639          STA byte
1640          PHP
1641          TXA
1642          PHA
1643          LDA byte+1
1644          BNE testshut
```

```
1645              LDA byte
1646              CMP #ASC("[")
1647              BNE return
1648              STA byte+1
1649 .return
1650              LDA byte
1651              JSR oswrch
1652              PLA
1653              TAX
1654              PLP
1655              LDA byte
1656              RTS
1657 .testshut
1658              LDA byte
1659              CMP #93
1660              BNE testcr
1661              LDA #0
1662              STA byte+1
1663              BEQ return
1664 .testcr
1665              CMP #13
1666              BNE testlabel
1667              LDA #0
1668              STA byte+4
1669              BEQ return
1670 .testlabel
1671              CMP #ASC(".")
1672              BEQ signal
1673              CMP #ASC(":")
1674              BCC return
1675              LDA byte+4
1676              BNE return
1677              LDX #10
1678              LDA #32
1679 .spaces
1680              JSR oswrch
1681              DEX
1682              BNE spaces
1683              LDA #1
1684 .signal
1685              STA byte+4
1686              JMP return
1687 .message
1688              EQUS"Assembler "
1689              EQUS"Formatter On!"
1690              EQUW &0D07
1691 .message2
1692              EQUS"Assembler "
1693              EQUS"Formatter Off!"
1694              EQUW &0D07
1695 .byte
1696              EQUS" "
1697              ]
```

```
1698              NEXT
1699              ENDPROC
```

Program 4.3. PROCass_format - makes an assembler listing more readable.

Like its BASIC predecessor, the Assembler Formatter has two entry points to turn the utility on and off. The 'on' entry point is at line 1606 which outputs the 'Assembler Formatter On' message before saving and redirecting the contents of the WRCHV to the 'assembler' entry point at line 1638. The 'off' routine, entered at line 1625 performs the reverse operation.

When the formatter is on, all output produced by the Beeb is channelled through the 'assembler' routine via WRCHV. After preserving the program status (lines 1639 to 1642) the accumulator's contents are tested to see if they contain the '[' code to indicate the start of assembler (line 1646). If this test succeeds, the code is stored at 'byte+1' .As you may have noticed, the code immediately before this tested this particular location to see if it were non-zero which would denote an already open assembler listing. This test routine would therefore be jumped over to the test_shut routine (line 1657). This section of code first tests to see if the close bracket, end of assembler mark, has been found in which case the 'byte' values are reset and the normal oswrch output pursued.

If a carriage return is not present (lines 1664 to 1669) the 'test_label' routine is invoked. If the label start character, a full-stop, is present the fact is signalled in 'byte-4' and the routine completed; a delimiting colon is treated in a similar manner. If neither of these characters is encountered, the X register is loaded with the number of padding spaces to be printed (line 1677). I chose to use ten though you can adjust this to your own taste. The 'spaces' loop is entered and exited on completion of printing the ten spaces. Mnemonics will subsequently be printed from this ten spaces in position, while labels are printed as usual.

Program fact sheets

Program 4.1

Procedure title	: PROCbasic_format
Variables required	: addr
Line numbers	: 1480 to 1592
Length	: 227 bytes
Zero page requirements	: none
Registers changed	: none

Program 4.2

Procedure title	: PROCass_format
Variables required	: addr
Line numbers	: 1600 to 1699
Length	: 214 bytes
Zero page requirements	: none

Chapter Five
The Screen

If you are interested in the graphics capabilities of the BBC Micro there will no doubt be occasions when you wish to save the graphics design you have created so that it cal; be recalled at a later date. Programs 5.1 and 5.2 will facilitate this using the OSFILE call to perform these tasks rapidly in machine code. The third program in this chapter, Program 5.3, provides a printer screen dump program that will work on the Epson, Star and compatible printers.

Save Screen Memory (Program 5.1)

The OSFILE routine is entered in the MOS at &FFDD. Like the majority of the operating system calls it expects to be pointed in the direction of a parameter block via an address held within the index registers. The parameter block needs to contain all the information required by the call to operate. Figure 5.1 details the OSFILE parameter block.

XY+0 to XY+1	: Filename address. Filename must be terminated by RETURN.
XY+2 to XY+5	: File load address, stored low byte first.
XY+6 to XY+9	: Run address of file, stored low byte first.
XY+10 to XY+13	: Data start address to be saved.
XY+14 to XY+17	: Data end address.

Fig. 5.1. The OSFILE parameter block.

The OSFILE call can perform up to eight different tasks depending upon the value in the accumulator when the call is effected and these are detailed in Figure 5.2. The call code we are interested in here is with the accumulator holding 0.

---

0   : Save block of memory as detailed in parameter block.
1   : Write information in parameter block to catalogue entry.
2   : Write load address only for existing file.
3   : Write the run address only for an existing file.
4   : Write file attributes only for an existing file.
5   : Read a file's catalogue information to parameter block.
6   : Delete file named in parameter block.
255 : Load the file detailed in the parameter block.

Fig. 5.2. The OSFILE call codes.

Program 5.1 is relatively straightforward, but the amount of screen memory to be saved will vary depending on the currently selected screen mode. For example, MODEs 0, 1 and 2 utilise a full 20K from &3000 while MODEs 4 and 5 require 10K from &5800, and the amazingly versatile MODE 7 needs just a meagre 1K from &7C00.

```
10 REM *** SAVE SCREEN MEMORY ***
20 PROCsavescreen (&C00)
30 inc=5
40 X=640 : Y=512
50 MODE 4
60 FOR loop=1 TO 50
70 MOVE X,Y
80 DRAW X+inc,Y
90 DRAW X+inc,Y+inc
100 DRAW X,Y+inc
110 DRAW X,Y
120 X=X-20 : Y=Y-20
130 inc=inc+40
140 NEXT
150 CALL &C00
160 END
170 :
1700 DEF PROCsavescreen (addr)
1701 FOR pass=0 TO 3 STEP 3
1702 P%=addr
1703 [
1704 OPT pass
1705 .save_screen
1706          LDA #135
1707          JSR &FFF4
1708          TYA
1709          BEQ dump1
1710          CMP#3
1711          BCC dump1
1712          CMP#4
1713          BEQ dump2
```

```
1714              CMP#5
1715              BEQ dump2
1716              CMP #7
1717              BEQ teletext
1718 .error
1719              LDY #0
1720 .loop
1721              LDA message,Y
1722              BEQ finished
1723              JSR &FFE3
1724              INY
1725              BNE loop
1726 .finished
1727              RTS
1728 .dump1
1729              LDA #&30
1730              STA paramblk+3
1731              STA paramblk+7
1732              STA paramblk+&0B
1733              LDA #0
1734              JMP osfile
1735 .dump2
1736              LDA #&58
1737              STA paramblk+3
1738              STA paramblk+7
1739              STA paramblk+&0B
1740              LDA#0
1741              JMP osfile
1742 .teletext
1743              LDA #&7C
1744              STA paramblk+3
1745              STA paramblk+7
1746              STA paramblk+&0B
1747              LDA #0
1748              JMP osfile
1749 .osfile
1750              LDX #paramblk MOD 256
1751              LDY #paramblk DIV 256
1752              JMP &FFDD
1753 .filename
1754              EQUS"SSAVED"
1755              EQUB 13
1756 .paramblk
1757              EQUB filename MOD 256
1758              EQUB filename DIV 256
1759              EQUD&3000
1760              EQUD 0
1761              EQUD&3000
1762              EQUD&7FFF
1763 .message
1764              EQUB 7
1765              EQUS"Not a graphics Mode"
```

```
1766             EQUB 13
1767             EQUB 0
1768             ]
1769             NEXT
1770             ENDPROC
```

Program 5.1. PROCsavescreen - saves the screen memory to tape or disk

The program acts 'intelligently' in this respect by obtaining the current screen mode from the Y register after an *FX135 call (lines 1706 to 1708). If, after the comparison of line 1710, the carry is clear a MODE of less than 3 is indicated and the branch to 'dump1' performed. If a mode value of 4 or 5 is determined, 'dump2' is sought while a branch to 'teletext' is executed if 7 is returned. Note that the 'error' loop is entered if the screen is in MODE 3 or MODE 6; this prints out the 'Not a graphics Mode' message from line 1765 and the routine is exited.

Each of these sections of code simply seed the first page number of the current graphics MODE into the correct places within the parameter block. If the graphics MODE was MODE 1 then the branch to 'dump1' would seed the value &30 into the three bytes at paramblk+3, paramblk+7 and paramblk+&0B, prior to loading the accumulator with 0 and jumping to 'osfile' at line 1749. Here the address of 'paramblk' is loaded into the index registers and a JMP to OSFILE at &FFDD performed.

The parameter block is located at the top of the calling machine code, lines 1756 to 1762 and the EQU functions used to prime the static contents. The filename is stored at 'filename' (line 1753) and I have chosen to use SSAVED, but this can be changed to suit your own needs, of course.

The BASIC test routine simply draws a succession of squares in MODE 4 before using the machine code to save the screen's contents.

The following program, Program 5.2, can be used to reload screen memory.

Load Screen Memory (Program 5.2)

The load screen memory program is essentially the same program as its saving counterpart. The main difference is that the accumulator is loaded with 255 to indicate a load operation to the MOS.

```
10 REM *** LOAD SCREEN MEMORY ***
20 PROCloadscreen (&C00)
30 MODE4
40 CALL &C00
```

```
  50 END
  60 :
1800 DEF PROCloadscreen (addr)
1801 FOR pass=0 TO 3 STEP 3
1802 P%=addr
1803 [
1804 OPT pass
1805 .load_screen
1806            LDA #135
1807            JSR &FFF4
1808            TYA
1809            BEQ dump1
1810            CMP#3
1811            BCC dump1
1812            CMP#4
1813            BEQ dump2
1814            CMP#5
1815            BEQ dump2
1816            CMP #7
1817            BEQ teletext
1818 .error
1819            LDY #0
1820 .loop
1821            LDA message,Y
1822            BEQ finished
1823            JSR &FFE3
1824            INY
1825            BNE loop
1826 .finished
1827            RTS
1828 .dump1
1829            LDA #&30
1830            STA paramblk+3
1831            STA paramblk+7
1832            STA paramblk+&0B
1833            LDA #255
1834            JMP osfile
1835 .dump2
1836            LDA #&58
1837            STA paramblk+3
1838            STA paramblk+7
1839            STA paramblk+&0B
1840            LDA #255
1841            JMP osfile
1842 .teletext
1843            LDA #&7C
1844            STA paramblk+3
1845            STA paramblk+7
1846            STA paramblk+&0B
1847            LDA #255
1848            JMP osfile
1849 .osfile
```

```
1850          LDX #paramblk MOD 256
1851          LDY #paramblk DIV 256
1852          JMP &FFDD
1853 .filename
1854          EQUS"SSAVED"
1855          EQUB 13
1856 .paramblk
1857          EQUB filename MOD 256
1858          EQUB filename DIV 256
1859          EQUD&3000
1860          EQUD 0
1861          EQUD&3000
1862          EQUD&7FFF
1863 .message
1864          EQUB 7
1865          EQUS"Not a graphics Mode"
1866          EQUB 13
1867          EQUB 0
1868          ]
1869          NEXT
1870          ENDPROC
```

Program 5.2. PROCloadscreen - loads a saved graphics screen back into screen memory.

Printer Screen Dumper

If you own or have aspirations to own a printer then you will certainly wish to be able to dump the contents of screen to the printer at some time to obtain that all important hard copy, be it a graphics masterpiece or just a copy of a some neatly formatted data. Program 5.3 was designed specifically for use with 'bit-mapped' printers such as the Epson and Star ranges. The program is a stand-alone version and includes a short graphics program at the start which will be dumped correctly if you have a suitable printer attached.

```
 10 REM *** PRINTER SCREEN DUMPER ***
 20 REM *** EPSON FX and STAR ***
 30 MODE 5
 40 X=640 : Y=512
 50 increment=5
 60 FOR loop=1 TO 50
 70 MOVE X,Y
 80 DRAW X+increment,Y
 90 DRAW X+increment,Y+increment
100 DRAW X,Y+increment
110 DRAW X,Y
120 X=X-20 : Y=Y-20
130 increment=increment+40
```

```
 140 NEXT
 150 PROCscreen_dump(&70,&71,&72,&73,&7
 5,&76,&2E00)
 160 CALL screen_dump
 170 END
 180 :
1900 DEFPROCscreen_dump(xlo,xhi,ylo,yhi
,byte,bits,addr)
1901 FOR pass=0 TO 2 STEP 2
1902 P%=addr
1903 [ OPT pass
1904 .screen_dump
1905          LDA #2
1906          JSR &FFEE
1907          LDA #1
1908          JSR &FFEE
1909          LDA #27
1910          JSR &FFEE
1911          LDA #1
1912          JSR &FFEE
1913          LDA #65
1914          JSR &FFEE
1915          LDA #1
1916          JSR &FFEE
1917          LDA #8
1918          JSR &FFEE
1919          LDA #1
1920          JSR &FFEE
1921          LDA #10
1922          JSR &FFEE
1923          LDA# &FF
1924          STA ylo
1925          LDA# &3
1926          STA yhi
1927 .next_row
1928          LDA #&0
1929          STA xlo
1930          LDA# &0
1931          STA xhi
1932          JSR duel_density
1933          LDA# &1
1934          JSR &FFEE
1935          LDA# &D
1936          JSR &FFEE
1937          SEC
1938          LDA ylo
1939          SBC# 32
1940          STA ylo
1941          BCS check_finish
1942          DEC yhi
1943 .check_finish
1944          LDA yhi
```

```
1945               CMP# &FF
1946               BNE next_row
1947               LDA ylo
1948               CMP# &FF
1949               BNE next_row
1950               LDA #1
1951               JSR &FFEE
1952               LDA #12
1953               JSR &FFEE
1954               LDA #1
1955               JSR &FFEE
1956               LDA #27
1957               JSR &FFEE
1958               LDA #1
1959               JSR &FFEE
1960               LDA #64
1961               JSR &FFEE
1962               LDA #3
1963               JSR &FFEE
1964               RTS
1965 .duel_density
1966               LDA# &1
1967               JSR &FFEE
1968               LDA #27
1969               JSR &FFEE
1970               LDA #1
1971               JSR &FFEE
1972               LDA #76
1973               JSR &FFEE
1974               LDA #1
1975               JSR &FFEE
1977               JSR &FFEE
1978               LDA #1
1979               JSR &FFEE
1980               LDA #2
1981               JSR &FFEE
1982 .next_byte
1983               LDA #0
1984               STA bits
1985               LDA #128
1986               STA byte
1987 .read_pixel
1988               LDA #9
1989               LDX #xlo
1990               LDY #0
1991               JSR &FFF1
1992               LDA xlo+4
1993               AND #&FF
1994               BEQ step4
1995               LDA byte
1996               ORA bits
1997               STA bits
1998 .step4
```

```
1999            SEC
2000            LDA ylo
2001            SBC #4
2002            STA ylo
2003            BCS rotate
2004            DEC yhi
2005  .rotate
2006            CLC
2007            ROR byte
2008            BCC read_pixel
2009  .print_pattern
2010            LDA #1
2011            JSR &FFEE
2012            LDA bits
2013            JSR &FFEE
2014            CLC
2015            LDA ylo
2016            ADC #32
2017            STA ylo
2018            BCC over
2019            INC yhi
2020  .over
2021            CLC
2022            LDA xlo
2023            ADC #2
2024            STA xlo
2025            BCC leap_frog
2026            INC xhi
2027  .leap_frog
2028            LDA xhi
2029            CMP #5
2030            BNE do_again
2031            RTS
2032  .do_again
2033            JMP next_byte
2034            ]
2035            NEXT pass
2036            ENDPROC
```

Program 5.3. PROCscreen_dump - outputs the graphics screen to a
connected printer.

The machine code of the program assembles just below the memory
required by either of the 20K screen modes. It would be a good idea to
obtain a second source coding that will sit just below the MODE 4 and 5
memory, thus making the 'unused' screen memory available for use by
the program. A suitable value for 'addr' in this instance would be &5600.

The major part of any graphics-printer dump program is spent
preparing the pixel - in other words, converting it from its screen form
into a form that the printer can handle and translate into selecting which

of its eight dot-matrix pins it fires. (Yes, I know there are nine but we only use eight!) The steps required to perform this conversion process are summarised below:

(a) Read a pixel off the screen.
(b) Adjust the byte using suitable rotates.
(c) Check a counter to see if byte is complete.
(d) Adjust the value of Y and X as needed to allow for resolution changes.
(e) Send the byte to the printer in the form of a VDU1 command.

   Looking at the assembler program shows that the first section of code from line 1904 to 1926 is responsible for issuing a series of VDU1 codes to the printer using OSWRCH. In BASIC terms the following is performed:

   VDU 2, 1, 27, 1, 65, 1, 8, 1, 10

The VDU 2 is used to enable the printer while the intermediate codes set the line spacing to 8/72 inches. The final VDU 10 performs a line feed. Much of the code comprises these VDU 1 codes and they could be more efficiently incorporated into a look-up table if required. I have persevered with the long-winded method mainly for reasons of clarity. Lines 1922 to 1932 initialise the variables ylo, yhi and xlo, xhi. The pair ylo,yhi are loaded with &3FF which in decimal is 1023 and shows itself to be the maximum on-screen value of the Y axis. The xlo,xhi combination are set to zero. The 'dual_density' subroutine is responsible for putting the printer in graphics geaf and performs a BASIC VDU 1, 27, 1, 76, 1, 128, 2 selecting 640 dots per line in bit image mode.
   Before printing, the current screen pixel details must be read from the screen. This is readily performed with OSWORD and the accumulator holding 9 (lines 1987 to 1992). The parameter block requires five bytes set out as follows using the declared variables:

   xlo   - low byte X coordinate
   xhi   - high byte X coordinate
   ylo   - low byte Y coordinate
   yhi   - high byte Y coordinate
   xlo+4 - result after OSWORD call

As can be seen, we have neatly used the program variables to form the parameter block of the call, a saving in coding and space when it works!
   The byte to be sent to the printer is formed by rotating it through the carry flag position into the accumulator (lines 2003 to 2013) and

printing it via OSWRCH. The rest of the general housekeeping is performed in lines up to 2033 and the whole process repeated until the 'check_finish' (line 1943) routine indicates a completed picture. The final succession of VDU 1 calls issue a form feed, place the printer into its more standard printing mode and disenable it. Figure 5.3 shows a dump produced by the program on my own printer.

   One final point: always ensure that the graphics origin is set to its normal default position prior to calling the dump. This is best done by inserting a VDU 29,0;0; at the onset of the program. As it stands, the program looks at every screen coordinate; if any of these have been moved off the screen due to a redefined graphics origin then the pixel read routine will return -1 or &FF, which will cause awful black bars and lines to be printed as part of your dump in the offscreen areas.
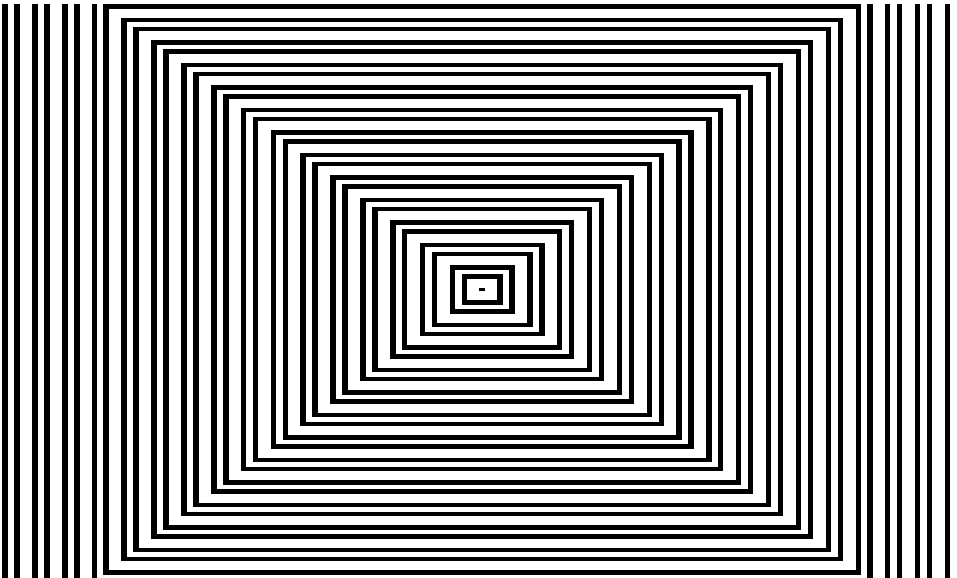

Fig. 5.3. Screen dump produced by Program 5.3.

Program fact sheets

Program 5.1

Procedure title	: PROCsavescreen
Variables required	: addr
Line numbers	: 1700 to 1770
Length	: 140 bytes
Zero page requirements	: none
Registers changed	: A, X, Y

Program 5.2

Procedure title	: PROCloadscreen
Variables required	: addr
Line numbers	: 1800 to 1870
Zero page requirements	: none
Registers changed	: A,X,Y

Program 5.3

Procedure title	: PROCscreen_dump
Variables required	: xlo, xhi, ylo, yhi, byte, bits, addr
Line numbers	: 1900 to 2036
Program length	: 260 bytes
Zero page requirements	: 7 bytes
Registers changed	: A,X,Y

Chapter Six
Softly, Softly

The Beeb allows the user to define characters using the VDU 23 command. This is followed by eight byte-sized numbers which represent the bit patterns of the eight bytes that form the character.

```
*** SOFT CHR CHARACTER DEFINITIONS ***
224:  32,165, 12,169,224,133,114,169,
225:  12,133,113,169,  0,133,112,133,
226: 115,208, 12,165,116,208, 12,165,
227: 115,208, 12,165,116,208,  8, 32,
228: 227, 12,208,240, 76,239, 12,165,
229: 114, 32, 89, 12,169, 58, 32,238,
230: 255,169, 32, 32,238,255,160,  0,
231: 117,112, 32, 89, 12,169, 44, 32,
232: 238,255,200,192,  8,208,241,  32,
233: 227, 12,169,  0,133,115,133,116,
234: 168,169, 13, 32,227,255, 76, 20,
235:  12,162,  0,134,117,201,100,144,
236:   8,233,100,232,134,117, 76, 93,
237:  12, 32,129, 12,162,  0,201, 10,
238: 144,  6,233, 10,232, 76,110, 12,
239:  32,129, 12, 24,105, 48, 76,238,
240: 255, 72,138,105, 48,201, 48,208,
241:   6,166,117,208,  2,169, 32, 32,
242: 238,255,104, 96,160,  7,177,112,
243:  24,101,115,133,115,144,  2,230,
244: 116,136, 16,242, 96,162,  0,189,
245: 180, 12, 48,  7, 32,238,255,232,
246:  76,167, 12, 96, 10, 13, 10, 13,
247:  42, 42, 42, 32, 83, 79, 70, 84,
248:  32, 67, 72, 82, 32, 67, 72, 65,
249:  82, 65, 67, 84, 69, 82, 32, 68,
250:  69, 70, 73, 78, 73, 84, 73, 79,
251:  78, 83, 32, 42, 42, 42, 10, 13,
252: 10, 13,255, 24,165,112,105,  8,
253: 133,112,230,114,240,  1, 96, 32,
254: 231,255,104,104, 96,  0,  0,  0,
```

Fig. 6.1. A typical output produced by PROCvduchrs.

Primarily these definable characters, 224 to 255, are used to create new characters whether they be fancy stylised alphanumeric characters or, more commonly, games characters. Program 6.1 provides a routine that will display the full definitions of any of these characters that have been defined. Figure 6.1 shows the output produced by the program.

User-definable characters are stored in the soft character definition area on page &C between &C00 to &CFF. Machine code programmers will know this area better as an assembly area for their code! As mentioned, eight bytes are associated with each character; thus, character VDU224 is allocated the eight bytes &C90 to &C07 inclusive; VDU 255 the bytes &C08 to &C0F, and on up to VDU 255 which is allocated the bytes &CF8 to &CFF. The first byte in each definition (the top-most one) is placed in the first byte of the corresponding memory location and so on - as Figure 6.2 illustrates.
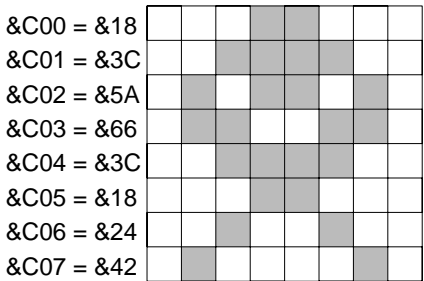
&C00 = &18
&C01 = &3C
&C02 = &5A
&C03 = &66
&C04 = &3C
&C05 = &18
&C06 = &24
&C07 = &42

Fig. 6.2. The byte definition storage of user-defined character 224.

As Figure 6.1 showed, the program does not print out the contents of every character - merely the characters that are or seem to be defined. This is quite simple to determine. On a power-up or reset, the MOS clears this area of memory with zero, so all the program needs to do is to add up the bytes corresponding to each VDU character. If the result is zero, no definition is present and the next character is sought. If, on the other hand, the result is non-zero then a definition is assumed and the contents printed. I say 'assumed' because it might not be a proper definition - it may, of course, be machine code! Also, the last 5 characters in the buffer, VDU 250 to VDU 255, seem to be susceptible to having garbage placed into them by the MOS.

Figure 6.3 flowcharts the program's operation. The definition test just discussed is performed by the 'test_for_definition' routine (lines 2135 to 2147 in Program 6.1). The restult of the summing is placed in Softly, Softly55 the 'addition' then a branch to 'print_definition' is executed (lines 2067 to 2070).
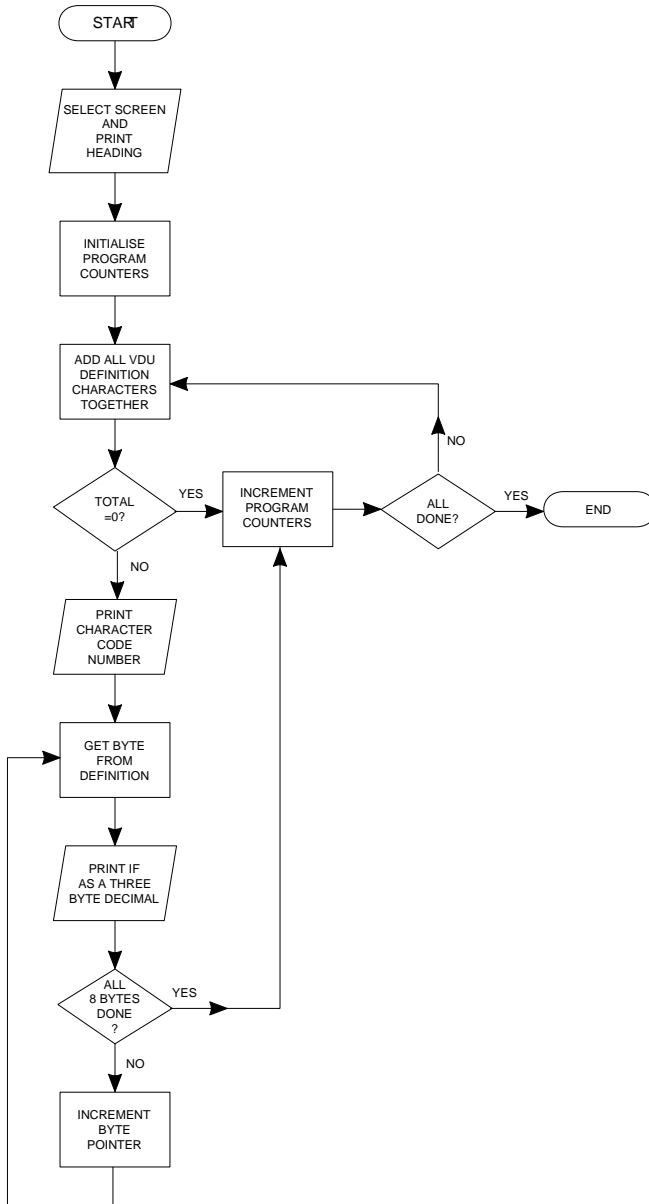
Fig. 6.3. The PROCvduchr flowchart.

The 'print_definition' loop (lines 2074 to 2097 in Program 6.1) begins by printing the VDU number of the current character followed by a colon. Each byte is then extracted in turn and printed to the screen in decimal form followed by a comma. After the last definition byte is printed a new line is printed and the next VDU character is sought. The 'update routine' (lines 2166 to 2173), as its name implies, increments all program counters and determines when every VDU character has been processed.

```
10 REM * SOFT CHR VDU'S VERSION V2 *
20 REM *(c) Bruce Smith/Acorn User *
30 PROCvdhchr(&70,&72,&73,&75,&4000)
40 *KEY0 CALL &4000|M
50 END
60 :
2050 DEF PROCvdhchr(soft_base,vdu_chara
cter,addition_bytes,flag,addr)
2051 FOR pass=0 TO 3 STEP3
2052 P%=&4000
2053 [ OPT pass
2054 .start
2055            JSR set_up_screen
2056            LDA #224
2057            STA vdu_character
2058            LDA #&C
2059            STA soft_base+1
2060            LDA #0
2061            STA soft_base
2062            STA addition_bytes
2063            STA addition_bytes+1
2064            TAY
2065 .main_loop
2066            JSR test_for_definition
2067            LDA addition_bytes
2068            BNE print_definition
2069            LDA addition_bytes+1
2070            BNE print_definition
2071            JSR update
2072            BNE main_loop
2073            JMP exit
2074 .print_definition
2075            LDA vdu_character
2076            JSR binary_decimal_print
2077            LDA #ASC":"
2078            JSR &FFEE
2079            LDA #ASC" "
2080            JSR &FFEE
2081            LDY #0
2082 .loop
2083            LDA (&70),Y
2084            JSR binary_decimal_print
```

```
2085              LDA #ASC","
2086              JSR &FFEE
2087              INY
2088              CPY #8
2089              BNE loop
2090              JSR update
2091              LDA#0
2092              STA addition_bytes
2093              STA addition_bytes+1
2094              TAY
2095              LDA #13
2096              JSR &FFE3
2097              JMP main_loop
2098 .binary_decimal_print
2099              LDX #0
2100              STX flag
2101 .hundreds
2102              CMP#100
2103              BCC no_hundreds
2104              SBC #100
2105              INX
2106              STX flag
2107              JMP hundreds
2108 .no_hundreds
2109              JSR print_decimal
2110              LDX #0
2111 .tens
2112              CMP #10
2113              BCC no_tens
2114              SBC #10
2115              INX
2116              JMP tens
2117 .no_tens
2118              JSR print_decimal
2119              CLC
2120              ADC #ASC"0"
2121              JMP &FFEE
2122 .print_decimal
2123              PHA
2124              TXA
2125              ADC #ASC"0"
2126              CMP #ASC"0"
2127              BNE no_zero
2128              LDX flag
2129              BNE no_zero
2130              LDA #32
2131 .no_zero
2132              JSR &FFEE
2133              PLA
2134              RTS
2135 .test_for_definition
2136              LDY#7
```

```
2137 .check_loop
2138          LDA (&70),Y
2139          CLC
2140          ADC addition_bytes
2141          STA addition_bytes
2142          BCC no_carry
2143          INC addition_bytes+1
2144 .no_carry
2145          DEY
2146          BPL check_loop
2147          RTS
2148 .set_up_screen
2149          LDX #0
2150 .next_character
2151          LDA table,X
2152          BMI done
2153          JSR &FFEE
2154          INX
2155          JMP next_character
2156 .done
2157          RTS
2158 .table
2159          EQUB 22
2160          EQUB 6
2161          EQUD &0D0A0D0A
2162          EQUS"*** SOFT CHR"
2163          EQUS" CHARACTER "
2164          EQUS"DEFINITIONS ***"
2165          EQUD &0D0A0D0A
2166          EQUB 255
2167 .update
2168          CLC
2169          LDA soft_base
2170          ADC#8
2171          STA soft_base
2172          INC vdu_character
2173          BEQ exit
2174          RTS
2175 .exit
2176          JSR &FFE7
2177          PLA
2178          PLA
2179          RTS
2180          ]
2181          NEXT pass
2182          ENDPROC
```

Program 6.1. PROCvduchrs - lists the soft character definitions.

The program incorporates a useful decimal printing routine between lines 2098 and 2134. This itself would be useful to have as a separate procedure. Character base conversion can seem difficult, but like most

things in life it is quite simple to do when you know how! As it stands, the routine will convert an eight-bit binary number held in the accumulator into a three-digit decimal ASCII number, or more correctly a string of three ASCII characters. Thus, if the accumulator held 1111000l (&F1) the ASCII string "241" would be printed.

To perform this, it is first necessary to calculate how many hundreds, tens and units there are in the byte. All that is required to do this is to subtract 100 or 10 from the byte and increment a hundreds or tens count each time the subtraction leaves a remainder. Using the byte &E1 mentioned above this would work as follows. First, the hundreds:

```
   241
  -100
  ─────
   141        hundreds count=1
   141
  -100
  ─────
    41        hundreds count=2
    41
  -100
  ─────
   -59        This result is negative
```

The final hundreds count is therefore 2. This can be converted into its ASCII code simply by adding ASC"0" and printing it.

Next, the tens count, and the value we use to start with is the remainder from the hundreds count.

```
   -10
  ─────
    31        tens count = 1
    31
   -10
  ─────
    21        tens count = 2
    21
   -10
  ─────
    11        tens count = 3
    11
   -10
  ─────
     1        tens count = 4
```

The final tens count is therefore 4, and adding ASCI"0" to this will derive the ASCII code for 4 which can be printed. Finally, the units count is left as the remainder, 1 in this case. Again, ASC"0" needs to be added to this to get the character's ASCII code so that it can be printed.

Program fact sheet

Program 6.1

Procedure title	: PROCvduchr
Variables Required	: soft_base, vdu_character, addition_bytes, flag, addr
Line numbers	: 2050 to 2181
Length	: 246 bytes
Zero page requirements	: 6 bytes
Registers changed	: A, X, Y

Chapter Seven
Global Variable Search and Replace

GREPL is the longest program in this book, a massive 582 bytes when assembled, but it is invaluable. Using it allows variable names within a program to be replaced throughout simply and easily. This eradicates the need to work through the program replacing them 'by hand', thus allowing new, more meaningful, names to be assigned or, if memory is tight, shorter names to be inserted.

Program description

Because of the long nature of the program, I have chosen to present the program details in a line-by-line block format which if used in conjunction with the flowchart of Figure 7.1 and the description of variable storage in Chapter 3 should make its understanding much easier.

Line 2195: Clear occurrence counter.
Lines 2196 to 2198: Print 'variable' prompt.
Lines 2199 to 2202: Input variable name to be replaced into buffer, pointed to by the Index registers and save the strings length in 'olen'.
Lines 2203 to 2205: Print 'Replace with' prompt.
Lines 2206 to 2208: Input new variable name and store it in buffer pointed to by the Index registers.
Lines 2210 to 2213: Calculate difference in variable name lengths and save result.
Lines 2214 to 2217: Read current setting of OSHWM.
Lines 2222 to 2221: Clear registers and get first byte from program.
Lines 2222 to 2227: If byte is ASCII return, check for the TOP marker &FF.
Lines 2228 to 2230: If TOP found perform OSNEWL and exit via 'report'.
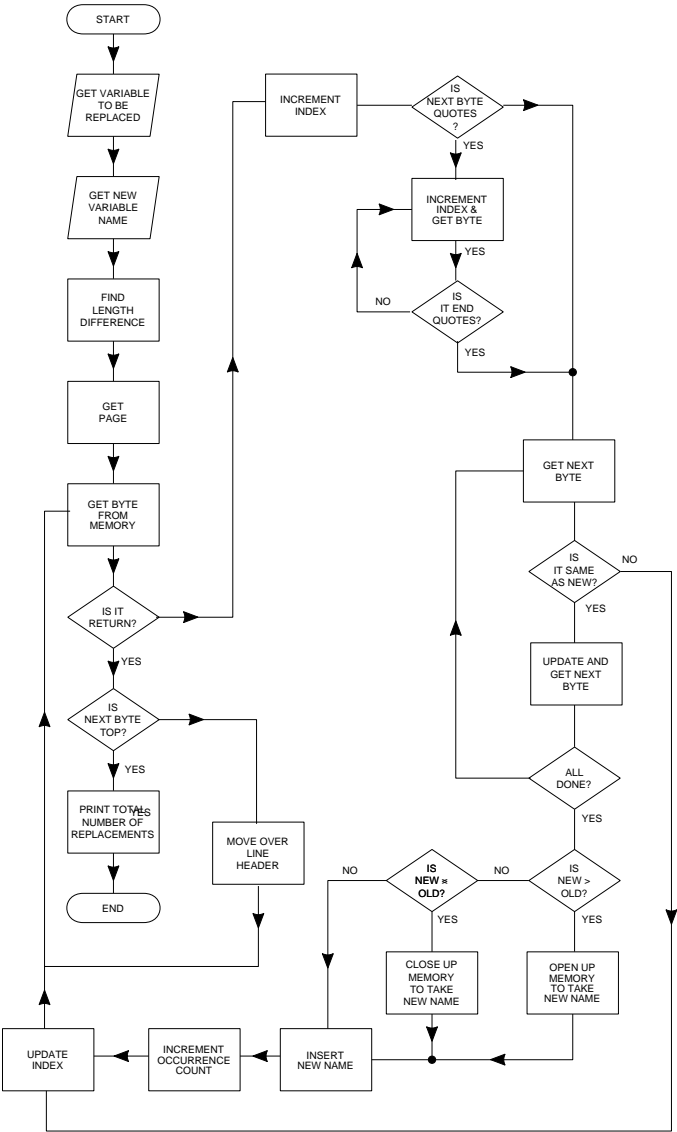
Fig. 7.1. Flowchart for PROCgrepl.

```
10 REM ***GLOBAL REPLACE - GREPL***
20 himem=HIMEM
30 himem=himem-&300
40 HIMEM=himem
50 PROCgrepl (&70,&72,&74,&76,&77,&78
,himem)
60 *KEY0 CALL HIMEM|M
70 END
80 :
2190 DEF PROCgrepl (current,last,link,o
len,nlen,result,himem)
2191 FOR pass=0 TO 3 STEP 3
2192 P%=HIMEM
2193 [OPT pass
2194 .Global_replace
2195          LDA #0 :STA number
2196          LDX #old_prompt MOD 256
2197          LDY #old_prompt DIV 256
2198          JSR print_string
2199          LDX #old_name_store MOD 256
2200          LDY #old_name_store DIV 256
2201          JSR input_string
2202          STA olen
2203          LDX #new_prompt MOD 256
2204          LDY #new_prompt DIV 256
2205          JSR print_string
2206          LDX #new_name_store MOD 256
2207          LDY #new_name_store DIV 256
2208          JSR input_string
2209 .do_again
2210          SEC
2211          STA nlen
2212          SBC olen
2213          STA result
2214          LDA #&83
2215          JSR &FFF4
2216          STX current
2217          STY current+1
2218 .main_loop
2219          LDX #0
2220          TXA :TAY
2221          LDA (current),Y
2222          CMP #13
2223          BNE not_return
2224          INY
2225          LDA (current),Y
2226          CMP #&FF
2227          BNE over
2228          JSR &FFE7
2229          LDA number
2230          JMP report
2231          RTS
```

```
2232 .over
2233            CLC
2234            LDA current
2235            ADC #3
2236            STA link
2237            LDA current+1
2238            ADC #0
2239            STA link+1
2240            LDY #4
2241            BNE update4
2242 .not_return
2243            CMP #&22
2244            BNE validity_test
2245 .end_quotes
2246            INY
2247            LDA (current),Y
2248            CMP #&22
2249            BEQ update3
2250            CMP #13
2251            BNE end_quotes
2252            BEQ update4
2253 .validity_test
2254            CMP #ASC"&"
2255            BEQ hexadecimal
2256            JSR check_variable
2257            BCC match_names
2258 .hexadecimal
2259            INY
2260            LDA (current),Y
2261            JMP validity_test
2262
2263 .match_names
2264            CPY olen
2265            BNE update2
2266            DEY
2267 .next_chr
2268            LDA (current),Y
2269            CMP old_name_store,Y
2270            BNE move_on
2271            DEY
2272            BPL next_chr
2273            BMI insert_new
2274 .move_on
2275            LDY olen
2276 .update2
2277            TYA
2278            BNE update4
2279 .update3
2280            INY
2281 .update4
2282            JSR memory_update
2283            DEY
2284            BNE update4
```

```
2285            BEQ main_loop
2286 .insert_new
2287            INC number
2288            LDA current
2289            STA last
2290            LDA current+1
2291            STA last+1
2292            LDY #0
2293            CLC
2294            LDA result
2295            ADC (link),Y
2296            CMP #238
2297            BCC leap_frog
2298            JMP bad_string
2299 .leap_frog
2300            LDX #2
2301            STA (link),Y
2302            LDA result
2303            BEQ overwrite
2304            BMI shuffle_down
2305 .back
2306            JMP memory_update
2307            LDA (last),Y
2308            CMP #&FF
2309            BNE back
2310            LDX #0
2311            LDY result
2312 .shuffle_up
2313            LDA (last,X)
2314            STA (last),Y
2315            LDA last
2316            BNE low_last
2317            DEC last+1
2318 .low_last
2319            DEC last
2320            LDA last
2321            CMP current
2322            BNE shuffle_up
2323            LDA last+1
2324            CMP current+1
2325            BNE shuffle_up
2326 .overwrite
2327            LDY #0
2328 .inset_loop
2329            LDA new_name_store,Y
2330            STA (current),Y
2331            INY :CPY nlen
2332            BNE inset_loop
2333            LDX #0
2334            BEQ update2
2335 .shuffle_down
2336            LDA result
```

```
2337               EOR #&FF
2338               TAY
2339               INY
2340 .next_down
2341               LDA (last),Y
2342               STA (last-2,X)
2343               JSR memory_update
2344               CMP #&FF
2345               BNE next_down
2346               BEQ overwrite
2347 .memory_update
2348               INC current,X
2349               BNE skip_high
2350               INC current+1,X
2351 .skip_high
2352               RTS
2353 .check_variable
2354               CMP #ASC"z"+1
2355               BCS less_than
2356               CMP #ASC"_"
2357               BCS greater_than
2358               CMP #ASC"Z"+1
2359               BCS less_than
2360               CMP #ASC"A"
2361               BCS greater_than
2362               CMP #ASC"$"
2363               BEQ greater_than
2364               CPY #0
2365               BEQ less_than
2366               CMP #ASC"9"+1
2367               BCS less_than
2368               CMP #ASC"0"
2369               BCS greater_than
2370               CMP #ASC"%"
2371               BEQ greater_than
2372 .less_than
2373               CLC
2374 .greater_than
2375               RTS
2376 .print_string
2377               STX current
2378               STY current+1
2379               LDY #0
2380 .print_string2
2381               LDA (current),Y
2382               BMI no_more
2383               JSR &FFE3
2384               INY
2385               BNE print_string2
2386 .no_more
2387               RTS
2388 .input_string
2389               STX last
```

```
2390              STY last+1
2391 .input_loop2
2392              LDY #0
2393 .get_character
2394              JSR &FFE0
2395              CMP #&1B
2396              BEQ escape
2397              CMP #13
2398              BEQ string_end
2399              CMP #&7F
2400              BEQ rub_out
2401              JSR check_variable
2402              BCC get_character
2403              STA (last),Y
2404              JSR &FFE3
2405 .input_loop3
2406              INY
2407              CPY #21
2408              BEQ too_big
2409              BNE get_character
2410 .string_end
2411              TYA
2412              BEQ get_character
2413              JSR &FFE7
2414              TYA
2415              RTS
2416 .rub_out
2417              DEY
2418              BMI input_loop3
2419              JSR &FFE3
2420              JMP get_character
2421 .too_big
2422              LDX #error1 MOD 256
2423              LDY #error1 DIV 256
2424              JSR print_string
2425              PLA :PLA :RTS
2426 .escape
2427              LDA #&7E
2428              JSR &FFF4
2429              PLA :PLA :RTS
2430 .report
2431              LDX #0
2432              SEC
2433 .decimal_loop
2434              SBC #10
2435              BMI no_jump
2436              INX
2437              JMP decimal_loop
2438 .no_jump
2439              DEX
2440              CLC
2441              ADC #58
```

```
2442              PHA
2443              TXA
2444              ADC #48
2445              CMP #ASC"0"
2446              BEQ no_print
2447              JSR &FFEE
2448 .no_print
2449              PLA
2450              JSR &FFEE
2451              LDX #done MOD 256
2452              LDY #done DIV 256
2453              JMP print_string
2454 .bad_string
2455              LDX #error2 MOD 256
2456              LDY #error2 DIV 256
2457              JSR print_string
2458              LDX #20
2459 .swap_pointers
2460              LDA old_name_store,X
2461              PHA
2462              LDA new_name_store,X
2463              STA old_name_store,X
2464              PLA
2465              STA new_name_store,X
2466              DEX
2467              BPL swap_pointers
2468              LDA olen
2469              PHA
2470              LDA nlen
2471              STA olen
2472              PLA :PLA :PLA
2473              RTS
2474 .old_name_store
2475              EQUS" "
2476 .new_name_store
2477              EQUS" "
2478 .old_prompt
2479              EQUB 13
2480              EQUS"Variable : "
2481              EQUB 255
2482 .new_prompt
2483              EQUB 13
2484              EQUS"Replace with :"
2485              EQUB 255
2486 .error1
2487              EQUB 13
2488              EQUS"Err1"
2489              EQUW &FF07
2490 .error2
2491              EQUB 13
2492              EQUS"Err2"
2493              EQUW &FF07
2494 .done
```

```
2495            EQUS" occurence(s) replaced"
2496            EQUW &FF0D
2497 .number    EQUB 0
2498            ] :NEXT pass
2499            ENDPROC
```

Program 7.1. PROCgrepl - a global search and replace facility

Lines 2233 to 2241: Otherwise move on past new line header bytes and force branch to 'update4' .

Lines 2243 to 2244: Test for quotes and branch if not there.

Lines 2245 to 2249: Locate the end pair of quotes.

Lines 2250 to 2252: If ASCII return found first, branch to 'update4' .

Lines 2254 to 2255: If a hexadecimal value is indicated, branch.

Lines 2256 to 2261: Check for a valid variable character.

Lines 2263 to 2272: Compare old variable name with the string pointed to in the program by 'current' . Exit on first unlike character.

Line 2273: If negative strings compared force a branch to 'insert_new'.

Lines 2274 to 2285: String not found so update all pointers and redo from 'main_loop' .

Lines 2286 to 2291: Increment occurrence pointer and update pointers.

Lines 2292 to 2298: Add new line length to the '}ink' byte. If link byte is greater than permissible value then perform 'bad_string' error. Else go to 'leap_frog' .

Lines 2299 to 2304: Calculate if space occupied by variable needs to be altered, if so, move distal portion of program up or down memory as required.

Lines 2305 to 2325: Open up the program at the variable name to make way for a longer variable name.

Lines 2326 to 2334: Write new variable name over the old variable name.

Lines 2335 to 2346: Close up variable space by desired amount to ensure that new shorter variable name fits correctly, then overwrite it.

Lines 2347 to 2352: Update current position in program vector.

Lines 2353 to 2375: Check that 'current' contents being investigated is a legal variable value.

Lines 2376 to 2387: Print the ASCII character string pointed to by the address held in the Index registers. Printing is terminated on encountering a negative byte, typically &FF.

Lines 2388 to 2415: Input an ASCII character string up to 20 characters long and store it in the buffer pointed to by the index registers.

Lines 2416 to 2420: Perform DELETE

Lines 2421 to 2425: Execute 'Too big' error.

Lines 2426 to 2429: Handle ESCAPE.
Lines 2430 to 2453: Print number of occurrences after first converting it
into an ASCII-based decimal number.
Lines 2454 to 2457: Print bad string error message.
Lines 2458 to 2473: Reset pointers to former values and exit to BASIC.
Lines 2474 to 2497: ASCII string storage area.

Using GREPL

Because of its large size, a hole must be created within the Beeb's
memory map to insert GREPL, because the normal page size areas are
not big enough. The program makes a niche by lowering HIMEM by
three pages and placing it above the new value, programming function
key 0 with the correct call address.

To use GREPL press f0 and answer to the prompts as they appear. The
new variable name may be up to 20 characters long; variables greater
than this are not accepted. Once the replace name is entered the program
goes about its business and the number of occurrences/ replacements are
indicated on completion.

Program fact sheet

Program 7.1

Procedure title	: PROCgrepl
Variables required	: current, last, link, olen, nlen, result, himem
Line numbers	: 2190 to 2499
Length	: 582 bytes
Zero page requirements	: 9 bytes
Registers changed	: A, X, Y

Chapter Eight
Time for Bed

Next to Saturday night's Match of the Day, the home computer must be the most frequent centrepiece of the friendly matrimonial dispute. Even four years on, my wife will often appear in the early hours of the morning to 'pull the plug out' of my latest sojourn into the land of ROM and RAM. It is certain that most hobbyists world-wide have suffered their mate's wrath in the small hours of the night at some time. It is difficult to explain to the non-committed that, once in front of the keyboard, time is meaningless.

This program was born at the specific request of my wife. It's a background clock that sits ticking its digits away at the top right-hand corner of the screen while the Beeb goes about its more important tasks, stopping once every second to create the tick or tock to push the second-hand a fraction further into the night!

The clock is based on the use of events or, more correctly, the redirection of events. The BBC Micro is built up around events, so much so that all the time it is switched on and being used it actually stops what it is doing every ten milliseconds to catch up on any outstanding house-keeping chores it needs to be. These chores take many guises and range from reading any pressed keys into the keyboard buffer to sampling some ofthe ADC channels. Due to the design of the BBC Micro it is possible to intercept these events as they take place and interpret them as we wish, and this concept forms the basis of the digital clock display.

There are several ways in which an event can be made to occur and these are listed in Figure 8.1. The one that we are particularly interested in is event 5 which occurs when the interval timer crosses zero. The interval timer is a 5-byte clock that is incremented one hundred times every second. When the timer is incremented so that it resets to zero, i.e. goes from &FFFFFFFFFF to &0, the event is initiated. When the event occurs, the operating system is directed through the event vector, EVNTV at &220, so that by redirecting this vector to our own event handler the appropriate action can be taken. The basic component in our clock is, of course, the second, so the interval timer must be made to time-out every second. Being an upcounting device, the interval timer must be loaded with -100 centiseconds. This write interval timer

operation is performed using an OSWORD 4 call. As with all OSWORD calls the index registers hold the address of the parameter block which contains the 5-byte value to be written. In Program 8.1, the parameter block is located at 'clock' lines 2606 to 2608.

Event	Cause
0	Output buffer empty
1	Input buffer full
2	Character for input buffer entering
3	ADC conversion finished
4	Vertical sync start
5	Interval timer crossing zero
6	ESCAPE detected
7	RS 423 error
8	Econet event detected
9	User event detected

Fig. 8.1. Details of operating system events

```
10 REM *** Continuous display clock *
**
20 REM *** redirects EVENTV vector *
**
30 *FX13,5
40 CLS
50 PRINTCHR$141;" The Mute Clock!"
60 PRINTCHR$141;" The Mute Clock!"'
''
70 INPUT"Hour :"H%
80 INPUT"Minute :"M%
90 INPUT"Seconds :"S%
100 PROCtime(H%,M%,S%,&A00)
110 PRINT''
120 PRINT"You have set the time for:"
130 PRINT" ";H%;":";M%;":";S%''
140 PRINT"Press key to start clock"
150 key=GET
160 CALL &A00
170 END
180 :
2500 DEF PROCtime(gethrs,getmins,getsec
s,addr)
2501 FOR pass=0 TO 2 STEP 2
2502 P%=addr
2503          [ OPT pass
2504          JMP setup
```

```
2505 .tick_clock
2506          PHP
2507          PHA
2508          TXA:PHA
2509          TYA
2510          PHA
2511          LDA#4
2512          LDY#clock DIV 256
2513          LDX#clock MOD 256
2514          JSR &FFF1
2515          INC seconds
2516          LDA seconds
2517          CMP#60
2518          BNE over
2519          LDA#0
2520          STA seconds
2521          INC minutes
2522          LDA minutes
2523          CMP#60
2524          BNE over
2525          LDA#0
2526          STA minutes
2527          INC hours
2528          LDA hours
2529          CMP#24
2530          BNE over
2531          LDA#0
2532          STA hours
2533 .over
2534          LDA hours
2535          LDY#72
2536          JSR display
2537          LDA#ASC":"
2538          STA HIMEM,Y
2539          INY
2540          LDA minutes
2541          JSR display
2542          LDA #ASC":"
2543          STA HIMEM,Y
2544          INY
2545          LDA seconds
2546          JSR display
2547          PLA
2548          TAY
2549          PLA
2550          TAX
2551          PLA
2552          PLP
2553          RTS
2554 .display
2555          LDX#0
2556          SEC
2557 .loop
```

```
2558            SBC#10
2559            BMI no_jump
2560            INX
2561            JMP loop
2562 .no_jump
2563            DEX
2564            CLC
2565            ADC#58
2566            PHA
2567            TXA
2568            ADC#48
2569            STA HIMEM,Y
2570            INY
2571            PLA
2572            STA HIMEM,Y
2573            INY
2574            RTS
2575
2576 .setup
2577            LDA #gethrs
2578            LDX #getmins
2579            LDY #getsecs
2580            STA hours
2581            STX minutes
2582            STY seconds
2583            LDA#22
2584            JSR &FFEE
2585            LDA#7
2586            JSR &FFEE
2587            LDA#28
2588            JSR &FFEE
2589            LDA#0
2590            JSR &FFEE
2591            LDA#24
2592            JSR &FFEE
2593            LDA#39
2594            JSR &FFEE
2595            LDA#2
2596            JSR &FFEE
2597            LDA#tick_clock MOD 256
2598            STA&220
2599            LDA#tick_clock DIV 256
2600            STA &221
2601            JSR tick_clock
2602            LDA#14
2603            LDX#5
2604            JSR &FFF4
2605            RTS
2606 .clock
2607            EQUD &FFFFFF9C
2608            EQUB &FF
2609 .hours EQUB 0
```

```
2610 .minutes EQUB 0
2611 .seconds EQUB 0
2612          ]
2613          NEXT
2614          ENDPROC
```

Program 8.1. PROCtime - a background digital clock

The initial program call to 'setup' (line 2504) does a number of things. First, it loads the hours, minutes and seconds values previously input into their respective counters (lines 2577 to 2582). A MODE 7 screen is then selected and a text window defined to ensure that the digital clock cannot be scrolled off the screen. Lines 2597 to 2600 reset the EVNTV vector to point to the 'tick_tock' routine at line 2505. The final lines (lines 2602 and 2603) perform an *FX14,5 which balances the previous *FX13,5 (line 30). These two calls disable and enable the interval timer crossing zero event.

The rest of the program's operation is straightforward. Each time the event occurs 'tick_tock' is entered and the interval timer reset to count a further second (lines 2511 to 2514). Note that on entry to the routine all processor registers are preserved. This is very important, otherwise the processor would probably crash when it returned to take up the task it was undertaking before the event occurred. Lines 25 15 to 2532 simply update the seconds, minutes and hours counters as required. The code between lines 2534 to 2546 stores the latest clock value at the top left-hand corner of the screen. The display subroutine (line 2554) called by the program performs a simple hex to decimal ASCII conversion by continually subtracting 10 from the value to be displayed.

Finally, the processor registers are restored (lines 2547 to 2552) before control is transferred back to the interrupted program.

Program fact sheet

Program 8.1

Procedure title	: PROCtime
Variables required	: gethrs, getmins, getsecs, addr
Line numbers	: 2500 to 2614
Zero page requirements	: none
Registers changed	: none

Chapter Nine
Error, Pack and Autorun

Error Lister (Program 9.1)

This utility can be of great help at the initial run-time debugging stage of a program. Normally, if an error occurs, one of the Beeb's terse error messages is issued and you are left to list the line referenced - often scratching your head wondering just where the problem is. The most infuriating part of debugging is when you make what amounts to a fundamental mistake to which you remain blind, no matter how many times you run and list the erroneous line. I speak from frequent experience!

One way around the error problem is to incorporate an error handling procedure in your program, starting the program off with a line such as:

   10 ON ERROR GOTO 5000

At line 5000, the error message and line can be printed out. The problem still remains that the erroneous line is not listed, nor is the source of the error listed.

Program 9.1 solves both these problems. After being set up and installed, errors occurring at run-time will be treated in the normal manner except that the line containing the error will be listed starting at the point of the error, thus highlighting the mistake. For example, the program line

   10 PRINT"HELLO" : STUPID ERROR : VDU 7

would normally result in the error:

   Mistake at line 10

at run-time. With the new error lister inserted, the response would be

STUPID ERROR : VDU 7
Mistake at line 10

```
  10 REM *** ERROR LISTER ***
  20 PROCerror (&C00)
  30 *KEY0 CALL &C00|M
  40 *KEY1 CALL &C24|M
  50 END
  60 :
2620 DEF PROCerror (addr)
2621 FOR pass=0 TO 3 STEP3
2622 brkv=?&202+(?&203*256)
2623 P%=addr
2624 [
2625 OPT pass
2626 .SETUP
2627          LDX #0
2628 .next_chr
2629          LDA message,X
2630          JSR &FFE3
2631          INX
2632          CMP #13
2633          BNE next_chr
2634          LDA &202
2635          STA address
2636          LDA &203
2637          STA address+1
2638          LDA #entry MOD 256
2639          STA &202
2640          LDA #entry DIV 256
2641          STA &203
2642          RTS
2643 .restore
2644          LDX #0
2645 .next_chr
2646          LDA message2,X
2647          JSR &FFE3
2648          INX
2649          CMP#13
2650          BNE next_chr
2651          LDA address
2652          STA &202
2653          LDA address+1
2654          STA &203
2655          RTS
2656 .entry
2657          BIT &FF
2658          BMI was_esc
2659          CLC
2660          LDA &1B
2661          ADC &39
2662          TAX
```

```
2663            LDY #0
2664            JSR &FFE7
2665 .next_error
2666            LDA (&19),Y
2667            CMP #13
2668            BEQ was_esc
2669            CMP #32
2670            BCC garbage
2671            CMP #&80
2672            BCS garbage
2673            JSR &FFEE
2674 .garbage
2675            INY
2676            DEX
2677            BNE next_error
2678 .was_esc
2679            JMP brkv
2680 .message
2681            EQUS" Error Lister On!"
2682            EQUB 7
2683            EQUB 13
2684 .message2
2685            EQUS" Error Lister Off!"
2686            EQUB 7
2687            EQUB 13
2688 .address
2689            EQUS" "
2690            ]
2691            NEXT
2692            ENDPROC
```

Program 9.1. PROCerror - lists the program line in which an error occured.

The section of line which created the mistake has been listed in addition to the normal error message.

The assembled program occupies just 141 bytes and is completely self-contained so that it can be tucked out of the way during debugging. As with other programs of this type in the Portfolio, there are two entry points - to switch the lister on (entry at line 2626) and off (entry at line 2643). The 'setup' section of code saves the normal contents of BRKV at &202 and revectors it to point to 'entry' at line 2656.

When the interpreter causes the program to abort via BRKV the new wedge coding is executed. It begins first at line 2657 by testing bit 7 of location &FF. If this bit is set then the abortion was due to the ESCAPE key being pressed and so the normal 'brkv' is jumped to. Assuming that ESC was not pressed, the length of the current expression being evaluated by the interpreter, and the one that caused the error to occur, is calculated. The bytes at &1B and &39 are summed (lines 2659 to 2662) and the result moved into the X register Location &1B contains the

current offset for the expression evaluation pointer while &39 contains the actual length of the expression.

The address of the current expression is held in the vector at &19 which is known as the expression evaluation base pointer, and each byte is in turn accessed and printed to the screen (line 2666). If a carriage return is encountered, the end of the line has been reached and the program jumps to the normal 'brkv' for the printing of tht error message (lines 2667 and 2668). The comparisons of lines 2669 and 2671 ensure that no garbage gets printed to the screen, should the program crash have caused any to have been poked into the program inadvertently.

A compact Pack (Program 9.2)

'Pack' is basically a simple program compacter that, when called. removes all traces of spaces and REM statements from it, leaving behind just the minimal program. There are two advantages in doing this. First, the program becomes smaller, and in the case of some programs much smaller. Second, by virtue of being smaller, they run and execute much faster; even a single space slows a program down a fraction, so a hundred spaces will slow a program down that much more! The saving in memory can make the difference between a program running in a high resolution mode and the dreaded 'Bad Mode' message being reported.

Pack searches through a program in the current text space and looks for spaces and REM statements and the messages that follow them. Of course, the program doesn't wipe the spaces and REMs out; rather, it just shifts the top end of the program down a byte or bytes to write over the offending space or REM.

Program 9.2 begins by placing the current value of PAGE into two zero page vectors (lines 2705 to 2710). These are used to keep position in the current program and point to the same, packed position, in the new program. A special byte is also cleared: this is the 'rem_flag' and is used to indicate if a REM statement is currently being processed. The heavy work of the program is performed by the subroutine, 'transfer' at lines 2784 to 2789. This moves a byte from its current position in the program undergoing packing to the Iinal version. Postindexed addressing is used throughout. After the subroutine call,the byte just moved is tested for TOP, by comparing it with &FF (line 2716) in which case the pack is complete. Note that at the start of each line an extra two transfers are required to move the line number down (lines 2718 and 2719).

```
  10 REM *** SPACE & REM REMOVER ***
  20 PROCpack (&70,&72,&C00)
  30 END
  40 :
2700 DEF PROCpack (current,new_position
,addr)
2701 FOR pass=0 TO 3 STEP3
2702 P%=addr
2703              [
2704              OPT pass
2705              LDA #0
2706              STA new_position
2707              STA current
2708              LDA &18
2709              STA new_position+1
2710              STA current+1
2711 .outer
2712              LDA #0
2713              STA rem_flag
2714              LDY #1
2715              JSR transfer
2716              CMP #&FF
2717              BEQ all_done
2718              JSR transfer
2719              JSR transfer
2720 .inner
2721              LDA (current),Y
2722              BIT rem_flag
2723              BPL flag_clear
2724              CMP #13
2725              BNE space
2726              JSR transfer
2727              BEQ end_of_line
2728 .flag_clear
2729              CMP #ASC" "
2730              BEQ space
2731              CMP #&F4
2732              BNE not_rem
2733              DEY
2734              LDA #&FF
2735              STA rem_flag
2736              BNE space
2737 .not_rem
2738              JSR transfer
2739              BEQ end_of_line
2740              CMP #&22
2741              BEQ inside_quote
2742              BNE inner
2743 .space
2744              INC current
2745              BNE inner
2746              INC current+1
2747              BNE inner
```

```
2748 .end_of_line
2749          DEY
2750          TYA
2751          PHA
2752          CPY #3
2753          BEQ clear
2754          LDY #3
2755          STA (new_position),Y
2756          CLC
2757          ADC new_position
2758          STA new_position
2759          BCC clear
2760          INC new_position+1
2761 .clear
2762          PLA
2763          CLC
2764          ADC current
2765          STA current
2766          BCC outer
2767          INC current+1
2768          BNE outer
2769 .inside_quote
2770          JSR transfer
2771          BEQ end_of_line
2772          CMP #&22
2773          BNE inside_quote
2774          BEQ inner
2775 .all_done
2776          LDA new_position
2777          CLC
2778          ADC #2
2779          STA &12
2780          LDA new_position+1
2781          ADC #0
2782          STA &13
2783          RTS
2784 .transfer
2785          LDA (current),Y
2786          STA (new_position),Y
2787          INY
2788          CMP #13
2789          RTS
2790 .rem_flag
2791          EQUS " "
2792          ]
2793          NEXT
2794          ENDPROC
```

Program 9.2. PROCpack - a space and REM remover.

The space and REM tests are performed in lines 2729 and 2731 respectively and the corresponding branch made accordingly. Ifa space is detected, the 'current' vector is incremented, no change is made to the 'new_position' vector and the space is not transferred. Thus, effectively the space gets lost as illustrated in Figure 9.1. The REM test looks for the token for REM which is &F4. If the token is found &FF is placed in the 'rem_flag' to indicate this - so that the program knows it is within a REM statement and is, in fact, 'deleting' items from the line rather than transferring them. A branch to 'space' (line 2736) increments the 'current' vector before a branch to 'inner' is forced.

Fig. 9.1. Overwriting bytes to compact a program.
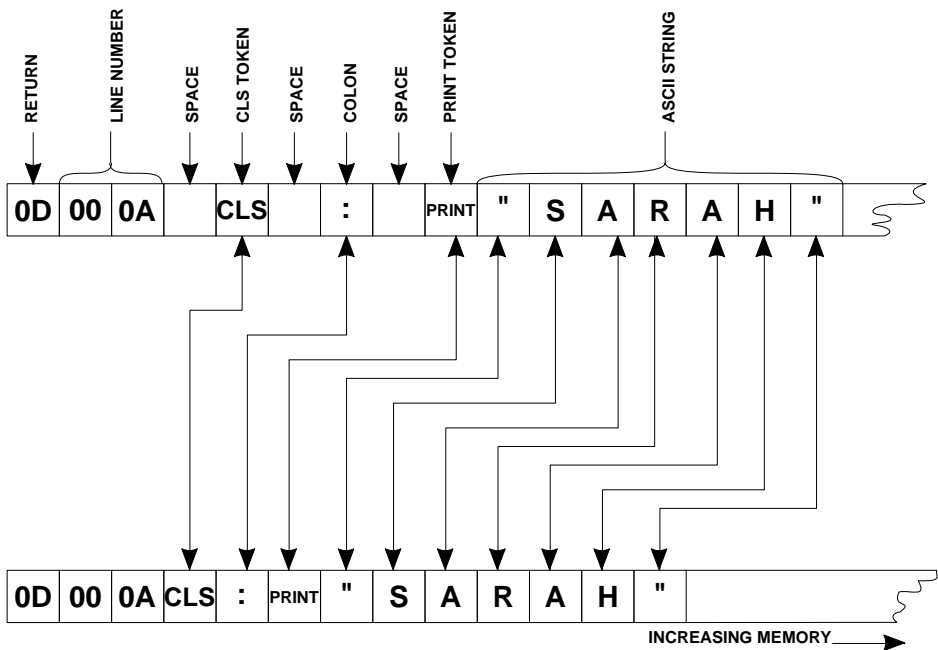
Figure 9.1. Overwriting bytes to compact a program

The relevant instruction here is in line 2722, where the 'rem_flag' is tested with BIT. If the flag is clear, the following branch is executed, otherwise the 'current' vector is incremented via 'space' .This entire process continues until the end of line return character is encountered (line 2724). The 'end_of_line' routine (line 2748) rapidly transfers the three-byte line header, as comparing this would be an utter waste of processor time.

Occasionally, spaces are required by programs. The most obvious occasion is within ASCII strings where they are used for formatting text. The 'inside_quote' coding ensures that any spaces occurring within the boundary of quotes are not removed. This section is entered via line 2741.

Autorun (Program 9.3)

This program is interesting in that once run you cannot stop it from automatically running the program at PAGE. No matter what combination of keys you try, be it ESCAPE, BREAK or even CTRL-BREAK the program runs! In fact, the only way to be rid of it is to turn off the power to the Beeb, so this makes it an easy way to protect your own programs from the hackers.

```
   0 REM!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!
  10 PROCautorun
  20 END
  30 :
2800 DEF PROCautorun
2801 *FX247,76
2802 *FX248,6
2803 A%=249
2804 Y%=0
2805 X%=PAGE DIV 256
2806 CALL &FFF4
2807 P%=PAGE+6
2808 [
2809           LDA #138
2810           LDX #0
2811           LDY #ASC("O")
2812           JSR &FFF4
2813           LDY #ASC("L")
2814           JSR &FFF4
2815           LDY #ASC("D")
2816           JSR &FFF4
2817           LDY #14
2818           DEY
```

```
2819              JSR &FFF4
2820              LDY #ASC("R")
2821              JSR &FFF4
2822              LDY #ASC("U")
2823              JSR &FFF4
2824              LDY #ASC("N")
2825              JSR &FFF4
2826              LDY #14
2827              DEY
2828              JSR &FFF4
2829              RTS
2830 ]
2831 ENDPROC
```

Program 9.3. PROCautorun - will automatically run a program no matter what!

The assembler is quite straightforward: an *FX 138 call is used to place the string "OLD<RETURN> RUN<RETURN>" into the keyboard buffer. It is not possible, however, to use this call to poke a return character, ASCII 13, into the buffer. To get round this, the Y register is loaded with 14 and then decremented (lines 2817 and 2826). The machine code is assembled in a rather strange place - in fact, it overwrites the 50 exclamation marks after the REM statement in line 0. As you can see, P% is set to PAGE+6 in line 2807. If you run the program then list it you will see that the !s are replaced by gobbledygook; this is just the interpreter trying to de-tokenise the machine code. These will have no effect when the program is run as they are away from the program, hiding behind the REM statement. The magic part of the program comes in lines 2801 to 2806. Here the BREAK intercept codes controlled by *FX247, *FX248 and *FX249 are rewritten to print the BRK handler to the code now stored at PAGE+6, so whenever any sort of BREAK is performed the interpreter comes here and OLDs and re-RUNs the program. Once the program has been run, only line 0 need remain; the others can be deleted as required.

Program fact sheets

Program 9.1

```
Procedure title           : PROCerror
Variables required        : addr
Line numbers              : 2620 to 2692
Length                    : 141 bytes
Zero page requirements    : none
Registers changed         : A, X, Y
```

Program 9.2

Procedure title	: PROCpack
Variables requircd	: addr
Line numbers	: 2700 to 2794
Length	: 149 bytes
Zero page requirements	: 4 bytes
Registers changed	: A, X, Y

Program 9.3

Procedure title	: PROCautorun
Variables required	: addr
Line numbers	: 2800 to 2831
Length	: 53 bytes (inside program)
Zero page requirements	: none
Registers changed	: A, X, Y

Chapter Ten
The Necessary Evil

Machine code programs of any length will often be required to manipulate numbers. Addition, subtraction, multiplication, division are all necessary evils in the computer world of data and figure manipulation. This chapter presents routines that should be versatile enough to cover most applications though often, by definition, they will be wasteful of memory and processor time. For example, rather than providing a procedure that will handle the multiplication of two eight-bit numbers a multi-byte multiplication procedure is provided. There is no reason, however, why you - the programmer - could not add a single-byte procedure to this Portfolio.

The programs provided in this chapter are as follows:

Program 10.1 : Multi-byte addition.
Program 10.2 : Multi-byte subtraction.
Program 10.3 : Multi-byte multiplication.
Program 10.4 : Multi-byte division.
Program 10.5 : Single-byte square root.
Program 10.6 : Double-byte square root.
Program 10.7 : Double-byte ASL.
Program 10.8 : Double-byte LSR.
Program 10.9 : Double-byte ROR.
Program 10.10 : Double-byte ROL.
Program 10.11 : Multi-byte ASL.

Multi-byte addition (Program 10.1)

Program 10.1 uses the post-indexed indirect address capabilities of the Beeb's 6502 to sum two sets of bytes stored anywhere in the Beeb's memory map, depositing the result over the first number. The start address of the two number sets is stored in the vectors 'first' and 'second'. These variables must therefore be assigned to addresses in zero page. A further variable, count, is required and this should contain the number of

bytes to be summed which is transferred into the X register to act as the bytes to add counter.

The program is simply an addition routine controlled by a loop counter. After seeding the index registers (lines 5005 to 5008) the carry flag is initially cleared. The 'first' byte is sourced and added to the 'second' byte with the result being stored at 'first' (lines 5009 to 5012). The index registers are adjusted and the loop reiterated until X becomes zero (lines 5013 to 5015).

Since the index registers are only capable of holding a maximum value of 255 the number of bytes to add together is limited to this value.

```
  10 REM *** MULTI-BYTE ADDITION ***
  20 PROCmulti_add(&70,&71,&73,&C00)
  30 @%=0
  40 ?&70=4
  50 !&71=&4000
  60 !&73=&4100
  70 !&4000=123456
  80 !&4100=123456
  90 PRINT'''>123456+123456=";
 100 CALL mbadd
 110 PRINT!&4000
 120 END
 130 :
5000 DEF PROCmulti_add(count,first,seco
nd,addr)
5001 FOR PASS=0 TO 3 STEP 3
5002 P%=addr
5003 [
5004 OPT PASS
5005 .mbadd
5006          LDX count
5007          LDY #0
5008          CLC
5009 .next_byte
5010          LDA (first),Y
5011          ADC (second),Y
5012          STA (first),Y
5013          INY
5014          DEX
5015          BNE next_byte
5016 RTS
5017 ]
5018 NEXT
5019 ENDPROC
```

Program 10.1. PROCmulti_add - adds two multi-byte numbers together

Multi-byte Subtraction (Program 10.2)

Program 10.2 operates identically to the last one. The only difference is that the SBC and the SEC instructions are substituted for their addition counterparts. It is important to remember that the 'second' value is subtracted from the 'first'.

```
  10 REM *** MULTI-BYTE SUBTRACTION ***
  20 PROCmulti_sub(&70,&71,&73,&C00)
  30 @%=0
  40 ?&70=4
  50 !&71=&4000
  60 !&73=&4100
  70 !&4000=123456
  80 !&4100=3456
  90 PRINT'''>123456-3456=";
 100 CALL mbsub
 110 PRINT!&4000
 120 END
 130 :
5030 DEF PROCmulti_sub(count,first,seco
nd,addr)
5031 FOR PASS=0 TO 3 STEP 3
5032 P%=addr
5033 [
5034 OPT PASS
5035 .mbsub
5036          LDX count
5037          LDY #0
5038          SEC
5039 .next_byte
5040          LDA (first),Y
5041          SBC (second),Y
5042          STA (first),Y
5043          INY
5044          DEX
5045          BNE next_byte
5046 RTS
5047 ]
5048 NEXT
5049 ENDPROC
```

Program 10.2. PROCmulti_sub - subtracts one multi-byte number from another.

Multi-byte Multiplication (Program 10.3)

Program 10.3 takes two multi-byte numbers (unsigned) stored low byte first, multiplies the 'first' by the 'second' and stores the result over the

'first'. In addition to requiring two vectored addresses, a 256- byte work buffer is required by the program. The TOTAL number of bytes to be multiplied together is expected in 'totlen' while the variable 'count' is used as a general loop counter by the program. The multiplication technique employed is a standard add-and-shift one. If the current bit being tested in the multiplier is a one, the multiplicand is added to the partial product, which is then rotated by one bit. If, on the other hand, the multiplier bit is 0 only the rotate is performed.

Because of the way 'mb_mult' is implemented, only the least significant bytes of the product are returned, i.e. the total number of bytes in the multiplier and multiplicand. The most significant bytes are always available in 'buffer' if required. Therefore, the user should check the 'buffer' for any overflow if it is suspected.

```
  10 REM *** MULTI-BYTE MULTIPLICATION
***
  20 PROCmulti_mult (&70,&72,&74,&75,&4
000,&4200)
  30 !&70=&3000
  40 !&72=&3100
  50 ?&74=4
  60 !&3000=1234
  70 !&3100=1234
  80 CALL mb_mult
  90 PRINT>Result of multiplication :";
 100 PRINT!&3000
 110 END
 120 :
5050          DEF PROCmulti_mult(first,second,to
tlen,count,buffer,addr)
5051          FOR pass=0 TO 3 STEP 3
5052          P%=addr
5053          [
5054          OPT pass
5055 .mb_mult
5056          LDA second
5057          SEC
5058          SBC #1
5059          STA second
5060          LDA second+1
5061          SBC #0
5062          STA second+1
5063          LDA first
5064          SEC
5065          SBC #1
5066          STA first
5067          LDA first+1
5068          SBC #0
5069          STA first+1
```

```
5070              LDA totlen
5071              BEQ finished
5072              STA count
5073              LDA #0
5074              ASL count
5075              ROL A
5076              ASL count
5077              ROL A
5078              ASL count
5079              ROL A
5080              STA count+1
5081              INC count
5082              BNE over
5083              INC count+1
5084 .over
5085              LDX totlen
5086              LDA #0
5087 .save_loop
5088              STA buffer-1,X
5089              DEX
5090              BNE save_loop
5091              CLC
5092 .loop
5093              LDX totlen
5094 .rotate_loop
5095              ROR buffer-1,X
5096              DEX
5097              BNE rotate_loop
5098              LDY totlen
5099 .rotate_save
5100              LDA (first),Y
5101              ROR A
5102              STA (first),Y
5103              DEY
5104              BNE rotate_save
5105              BCC no_add
5106              LDY #1
5107              LDX totlen
5108              CLC
5109 .add_loop
5110              LDA (second),Y
5111              ADC buffer-1,Y
5112              STA buffer-1,Y
5113              INY
5114              DEX
5115              BNE add_loop
5116 .no_add
5117              DEC count
5118              BNE loop
5119              LDX count+1
5120              BEQ finished
5121              DEX
5122              STX count+1
```

```
5123            JMP loop
5124 .finished
5125            RTS
5126            ]
5127            NEXT
5128            ENDPROC
```

Program 10.3. PROCmulti_mult - multiplies two multi-byte numbers

The program operates as follows:

Lines 5056 to 5062: Subtract 1 from address of 'second'.
Lines 5063 to 5069: Subtract 1 from address of 'first'.
Lines 5070 to 5071: If total length is zero then end.
Lines 5072: Set number of bytes to count.
Lines 5073 to 5079: Multiply count by eight.
Lines 5080 to 5083: Add one to value of count.
Lines 5084 to 5091: Save high product in buffer.
Lines 5092 to 5098: Shift carry bit into buffer and bit 0 of high product into the carry flag.
Lines 5099 to 5104: Rotate carry into most significant bit of 'first' and shift next bit of multiplier into the carry flag.
Line 5105: Carry clear so no addition required.
Lines 5106 to 5115: Carry flag is set so add *second' and high product together.
Lines 5116 to 5123: Decrement bit count and exit if zero, else repeat for the next bit.

The lines of BASIC show how the routine needs to be set up before calling it. In a larger assembler program these introductory peeks and pokes would be performed using assembler and the multiplication routine called as a subroutine from the main program. Lines 30 and 40 place the two addresses of data into the zero page vectors, while lines 60 and 70 place the values to be multiplied (both 1234) into these data buffers. Previously, in line 50, the total number of bytes to be combined, four (1234 can be held in two bytes), is poked into location &74 which corresponds to the variable 'totlen' in the procedure. After calling the subroutine (line 80) the final result is displayed. Check it on a calculator if you wish!

Multi-byte Division (Program 10.4)

Program 10.4 will divide two unsigned multi-byte number using a standard shift and subtract procedure whereby a 1 is placed in the quotient each time a subtraction is possible, and a 0 if not. The dividend is located at 'first' and the divisor at 'second'; during the division the quotient overwrites the dividend. Any remainder from the division is placed at the address given by 'hidiv_pointer'.

```
  10 REM *** MULTI-BYTE DIVISION ***
  20 PROCmult_div(&70,&72,&74,&75,&77,&
79,&3000,&3100,&4000)
  30 !&70=&3400
  40 !&72=&3500
  50 ?&74=3
  60 !&3400=10000
  70 !&3500=2
  80 CALL &4000
  90 PRINT >Result is :";
 100 PRINT!&3400
 110 PRINT>Remainder :";
 120 PRINT!&3000
 130 END
 140 :
5150 DEF PROCmult_div(first,second,totl
en,count,hidiv_pointer,pointer,buffer1,b
uffer2,addr)
5151          FOR pass=0 TO 3 STEP 3
5152          P%=addr
5153          [
5154          OPT pass
5155 .multi_div
5156          LDA totlen
5157          BNE begin
5158          JMP okay_out
5159 .begin
5160          STA count
5161          LDA #0
5162          ASL count
5163          ROL A
5164          ASL count
5165          ROL A
5166          ASL count
5167          ROL A
5168          STA count+1
5169          INC count
5170          BNE over
5171          INC count+1
5172 .over
```

```
5173            LDX totlen
5174            LDA #0
5175 .clear
5176            STA buffer1-1,X
5177            STA buffer2-1,X
5178            DEX
5179            BNE clear
5180            LDA #buffer1 MOD 256
5181            STA hidiv_pointer
5182            LDA #buffer1 DIV 256
5183            STA hidiv_pointer+1
5184            LDA #buffer2 MOD 256
5185            STA pointer
5186            LDA #buffer2 DIV 256
5187            STA pointer+1
5188            LDX totlen
5189            LDY #0
5190            TYA
5191 .check
5192            ORA (second),Y
5193            INY
5194            DEX
5195            BNE check
5196            CMP #0
5197            BNE divide
5198            JMP error
5199 .divide
5200            CLC
5201 .set_loop
5202            LDX totlen
5203            LDY #0
5204 .loop
5205            LDA (first),Y
5206            ROL A
5207            STA (first),Y
5208            INY
5209            DEX
5210            BNE loop
5211 .dec_count
5212            DEC count
5213            BNE set_shift
5214            LDX count+1
5215            BEQ okay_out
5216            DEX
5217            STX count+1
5218 .set_shift
5219            LDX totlen
5220            LDY #0
5221 .shift_loop
5222            LDA (hidiv_pointer),Y
5223            ROL A
5224            STA (hidiv_pointer),Y
```

```
5225              INY
5226              DEX
5227              BNE shift_loop
5228              LDY #0
5229              LDX totlen
5230              SEC
5231 .subtract
5232              LDA (hidiv_pointer),Y
5233              SBC (second),Y
5234              STA (pointer),Y
5235              INY
5236              DEX
5237              BNE subtract
5238              BCC set_loop
5239              LDY hidiv_pointer
5240              LDX hidiv_pointer+1
5241              LDA pointer
5242              STA hidiv_pointer
5243              LDA pointer+1
5244              STA hidiv_pointer+1
5245              STY pointer
5246              STX pointer+1
5247              JMP set_loop
5248 .okay_out
5249              CLC
5250              BCC finished
5251 .error
5252              SEC
5253 .finished
5254              RTS
5255              ]
5256 NEXT
5257 ENDPROC
```

Program 10.4. PROCmulti_div - divides one multi-byte number by another

The total number of bytes to be referenced in the division is placed in 'totlen' prior to the call. On exit from the routine, the carry flag bit is set if an error occurred during the division - otherwise it returns clear. The program operates as follows:

Lines 5156 to 5158: Get 'totlen' if zero then perform a no error finish.
Lines 5160 to 5166: Set count and then multiply by 8 to obtain total number of bits to do.
Lines 5167 to 5171: Add one to bit counter.
Lines 5173 to 5179: Initialise the high dividend result buffer to zero.
Lines 5180 to 5187: Point vectors to buffers.
Lines 5188 to 5198: Check that the divisor, held in 'second' is not zero!
Lines 5199 to 5200: Clear carry flag on entry into 'divide'.

Lines 5201 to 5210: Move the carry flag bit into the low dividend, 'first', to use as the next quotient bit. Then move the most significant bit of the low dividend into the carry flag bit.

Lines 5211 to 5217: Decrement ^count' by one and branch to 'okay_out' if all bits are done.

Lines 5218 to 5227: Transfer the carry flag bit into the least significant bit of the high dividend.

Lines 5228 to 5237: Subtract 'second' from high dividend and save result at 'pointer'.

Lines 5238 to 5246: If the carry flag bit is set then the trial subtraction worked! Therefore, set the quotient bit and switch pointers to replace remainder and dividend. If the carry flag is clear, the trial subtraction failed; therefore skip swap over and branch direct as next quotient bit is zero.

Line 5247: Do next bit.

Lines 5248 to 5250: Finished with no errors detected.

Lines 5251 to 5254: Finished with an error present.

Once again, the first few programs lines show how the procedure can be set up, using BASIC. The procedure is assembled passing workspace, vectors and buffer locations as parameters (line 20). The two vectors at 'first' and 'second' are poked with buffer addresses (lines 30 and 40), which are subsequently seeded with the dividend and divisor (lines 60 and 70). The dividend is 10000 and the divisor 2, which require a total of three bytes' storage, as indicated in line 50 which pokes the byte count into 'totlen'.

After calling the routine the result is extracted from the buffer at &3400 and the remainder from the buffer at &3000.

Finding the square root of a number in machine code might at first sight seem rather difficult. However, there is a quite straightforward solution. The method is simply this: 'the square root of an integer number is equal to the total number of successively higher odd integer numbers that can be subtracted from it'. Consider the number 36: first we subtract one from it, then three, then five and so on until we have no remainder. The total number of odd numbers subtracted is its square root! Thus,

```
36-1=35 : partial root = 1
35-3=32 : partial root = 2
32-5=27 : partial root = 3
27-7=20 : partial root = 4
20-9=11 : partial root = 5
11-11=0 : final square root = 6
```

If the final partial root does not yield a result of 0 then a remainder is available which can be 'floated' to provide the decimal portion ofthe root.

Program 10.5 provides a suitable assembler-based procedure to calculate the square root of any single byte number located at 'byte'. The Y register is used to keep a count of the partial root which is incremented each time round the 'loop'. The location at 'byte+1' is used to hold the odd number to be subtracted. The final root is deposited in 'byte'and any remainder at 'byte+1'.

```
  10 REM ***SINGLE BYTE SQUARE ROOT***
  20 PROConebyte_square(&70,&C00)
  30 ?&70=170
  40 CALL square
  50 PRINT>Square root =";?&70
  60 PRINT>remainder =";?&71
  70 END
  80 :
5270 DEF PROConebyte_square(byte,addr)
5271 FOR pass=0 TO 3 STEP 3
5272 P%=addr
5273 [
5274 OPT pass
5275 .square
5276           LDY #0
5277           LDA #1
5278           STA byte+1
5279           LDA byte
5280 .loop
5281           CMP byte+1
5282           BCC finished
5283           SBC byte+1
5284           INY
5285           INC byte+1
5286           INC byte+1
5287           JMP loop
5288 .finished
5289           STY byte
5290           STA byte+1
5291           RTS
5292 ]
5293 NEXT
5294 ENDPROC
```

Program 10.5. PROConebyte_square - calculates the square root of a single-byte number.

Program 10.5 is simple but effective. On entry to 'square' (line 5275) the Y register is initialised ready to take the partial root count; the

accumulator is loaded with the first odd number to be subtracted which is then written to the location 'byte+1' (lines 5276 to 5279). The subtract and count loop is embodied in lines 5280 to 5287. Line 5281 begins by comparing the contents from byte (the current remainder) with the next odd number, a clear carry flag denotes that the remainder is less than the next odd number and the program branches to 'finish'. A set carry and line 5283 subtracts the current odd number from the current remainder (line 5283) and the Y register is incremented. Before the loop is redone the two is added to the contents of 'byte+1' to move onto the next odd number (lines 5285 and 5286). Note that the program passes the immediate value through to the procedure for splitting into two bytes and storing in 'block' before the shift is performed. If the value is already held in 'block' then 'do-asl' can be called directly.

Program 10.6 is a double-byte version, Program 10.5 finding the square root of an unsigned 16-bit integer value. The two locations at 'byte' hold the integer value while 'temp' counts the double-byte odd number. The program operates virtually the same as its predecessor; but because a two-byte value is involved a subtraction rather than a compare must be performed initially. To ensure that the final subtraction will not erode any remainder, its possible low order byte is preserved in the X register. Looking further down the program listing (line 5322) it seems at first sight that the odd number counter is only being incremented by one. However, two is actually being added as the carry flag will be set at this point, if it is clear where the branch to 'finish' at line 5319 would have been performed.

```
  10 REM *** TWO BYTE SQUARE ROOT ***
  20 PROCtwobyte_square(&70,&72,&C00)
  30 !&70=1234
  40 CALL two_square
  50 PRINT>Square root =";?&70
  60 PRINT>remainder =";?&71
  70 END
  80 :
5300 DEF PROCtwobyte_square(byte,temp,a
ddr)
5301 FOR pass=0 TO 3 STEP 3
5302 P%=addr
5303 [
5304 OPT pass
5305 .two_square
5306            LDY #1
5307            STY temp
5308            DEY
5309            STY temp+1
5310 .loop
5311            SEC
```

```
5312            LDA byte
5313            TAX
5314            SBC temp
5315            STA byte
5316            LDA byte+1
5317            SBC temp+1
5318            STA byte+1
5319            BCC finished
5320            INY
5321            LDA temp
5322            ADC #1
5323            STA temp
5324            BCC loop
5325            INC temp+1
5326 .finished
5327            STY byte
5328            STX byte+1
5329            RTS
5330 ]
5331 NEXT
5332 ENDPROC
```

Program 10.6. PROCtwobyte_square - calculates the square root of a 16-bit value.

By the left!

The final five programs in this chapter deal with double-byte shifts and rotates. It may seem at first that these would be straightforward enough, but this is certainly not the case with the ASL and LSR combinations, as both of these introduce a 0 into bit 0 and bit 7 of the byte they are acting on respectively. Thus, a two-byte ASL will not yield the correct result if the sequence

    ASL byte
    ASL byte+1

is used.

To perform an overall ASL on two bytes, the initial ASL must be followed by a ROL. Program 10.7 illustrates the technique while Figure 10.1 shows what is happening. The ASL of line 5359 moves bit 7 of 'block+1' (the low byte in true 6502 back-to-frontness!) into the carry inserting a 0 into bit 0. Line 5360 then performs a ROL which moves bit 7 in the carry into bit 0 of 'block', shuffling the internal bits up one bit. The last bit, bit 7, falls out into the carry. The two-byte value has also been multiplied by two!

```
  10 REM*** DOUBLE BYTE ASL ***
  20 PROCtwo_byte_asl(1,&70,&C00)
  30 CALLset_asl
  40 FOR loop=1 TO 15
  50 PRINT?&70*256+?&71
  60 CALLdo_asl
  70 NEXT loop
  80 END
  90 :
5350 DEF PROCtwo_byte_asl(num,block,add
r)
5351 P%=addr
5352 [
5353 .set_asl
5354          LDA #num MOD 256
5355          STA block
5356          LDA #num DIV 256
5357          STA block+1
5358 .do_asl
5359          ASL block+1
5360          ROL block
5361          RTS
5362 ]
5363 ENDPROC
```

Program 10.7. PROCtwo_byte_asl - arithmetical shift left on a 16-bit value.
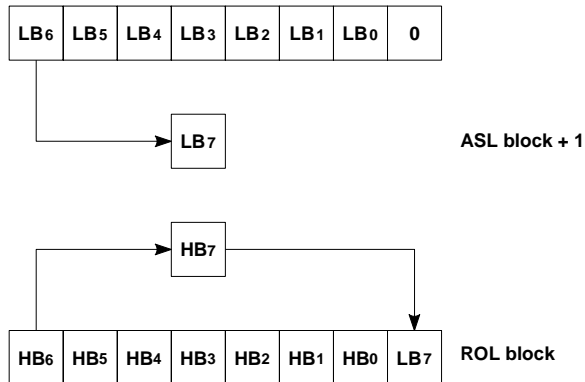
Fig. 10.1. Implementing a 16-bit shift register using an ASL/ROL

Program 10.8 works in the opposite direction performing an overall LSR using an LSR and ROR in conjunction. As the shift works in the opposite direction the bytes are referenced in the opposite order to an ASL. The shift is performed on the low byte in 'block' and the rotate on the high byte in 'block+1'. The total effect is to halve the two-byte number.

```
10 REM*** DOUBLE BYTE LSR ***
20 PROCtwo_byte_lsr(65535,&70,&C00)
30 CALLset_lsr
40 FOR loop=1 TO 15
50 PRINT?&70*256+?&71
60 CALLdo_lsr
70 NEXT loop
80 END
90 :
5370 DEF PROCtwo_byte_lsr(num,block,add
r)
5371 P%=addr
5372 [
5373 .set_lsr
5374          LDA #num DIV 256
5375          STA block
5376          LDA #num MOD 256
5377          STA block+1
5378 .do_lsr
5379          LSR block
5380          ROR block+1
5381          RTS
5382 ]
5383 ENDPROC
```

Program 10.8. PROCtwo_byte_lsr - logical shift right on a 16-bit value.

Programs 10.9 and 10.10 perform double-byte RORs and ROLs respectively. The only difference to note here is the order in which the bytes in 'block' are rotated. In RORing a two-byte number, the low byte is rotated first. With a ROL it is the high byte that is manipulated first.

```
10 REM*** DOUBLE BYTE ROR ***
20 PROCtwo_byte_ror(32768,&70,&C00)
30 CALLset_ror
40 FOR loop=1 TO 15
50 PRINT?&70*256+?&71
60 CALLdo_ror
70 NEXT loop
80 END
90 :
5390 DEF PROCtwo_byte_ror(num,block,add
```

```
r)
5391 P%=addr
5392 [
5393 .set_ror
5394          LDA #num MOD 256
5395          STA block
5396          LDA #num DIV 256
5397          STA block+1
5398 .do_ror
5399          ROR block
5400          ROR block+1
5401          RTS
5402 ]
5403 ENDPROC
```

Program 10.9. PROCtwo_byte_ror - a double-byte rotate right.

```
  10 REM*** DOUBLE BYTE ROL ***
  20 PROCtwo_byte_rol(1,&70,&C00)
  30 CALLset_rol
  40 FOR loop=1 TO 15
  50 PRINT?&70*256+?&71
  60 CALLdo_rol
  70 NEXT loop
  80 END
  90 :
5410 DEF PROCtwo_byte_rol(num,block,add
r)
5411 P%=addr
5412 [
5413 .set_rol
5414          LDA #num MOD 256
5415          STA block
5416          LDA #num DIV 256
5417          STA block+1
5418 .do_rol
5419          ROL block
5420          ROL block+1
5421          RTS
5422 ]
5423 ENDPROC
```

Program 10.10. PROCtwo_byte_rol - a double-byte rotate left.

Finally, Program 10.11 shows how a multi-byte shift, left in this instance, can be implemented on an unsigned value between 2 and 255 bytes in length. The start of the bytes to be shifted are located in 'start' while the number of them is found in 'bytes'. The program commences by placing the 'bytes' count into the Y register and performing the initial

ASL on 'start' (lines 5434 to 5436). The X register is loaded with one and after decrementing tlne Y register the 'next' loop is entered (lines 5436 to 5438). From here on, indexed addressing is used to facilitate the ROL on the remaining bytes. The first handful of lines in the program point out the sort of interesting, and perhaps useful (!) applications the program can be used for. The test of line 50 is printed onto the MODE 6 screen before the memory is used to hold the test is shifted left twice (lines 60 and 70). The net effect is to provide 3D text!

```
  10 REM *** MULTI LEFT SHIFT ***
  20 REM *** GIVES 3D CHARACTERS ***
  30 PROCmulti_left (&6000,64,&C00)
  40 MODE 6
  50 PRINT> HELLO THERE!!"
  60 CALL&C00
  70 CALL&C00
  80 END
90 :
5430 DEF PROCmulti_left (start,bytes,ad
dr)
5431 FOR PASS=0 TO 3 STEP3
5432 P%=addr
5433 [ :OPT PASS
5434            LDY bytes
5435            ASL start
5436            LDX #1
5437            DEY
5438 .next
5439            ROL start,X
5440            INX
5441            DEY
5442            BNE next
5443            RTS
5444 ]
5445 NEXT
5446 ENDPROC
```

Program 10.11. PROCmulti_left - performs an arithmetic shift left on a multi-byte number.

Program fact sheets

Program 10.1

Procedure title	: PROCmulti_add
Variables required	: count, first, second, addr
Line numbers	: 5000 to 5019
Length	: 16 bytes

Zero pagc requircments: : five bytes
Registcrs changed   : A, X, Y

Program 10.2

Procedure title   : PROCmulti_sub
Variables required   : count, first, second, addr
Line numbers   : 5030 to 5049
Length   : 16 bvtes
Zero page requiremcnts   : 5 bytes
Registers changed   : A, X, Y

Program 10.3

Procedure title   : PROCmulti_mult
Variables required   : first, sccond, totlen, count buffer, addr
Line numbers   : 5050 to 5128
Length   : 114 bytes
Zero page requirements: : 6 bytes
Registers changed   : A, X, Y

Program 10.4

Procedure title   : PROCmulti_div
Variables required   : first. second, totlen, count, hidiv_pointer,
                               pointer, 1 buffer2, addr
Line numbers   : 5150 to 5257
Length   : 54 bytes
Zero page requirements   : 11 bytes
Registers changed   : A, X, Y

Program 10.5

Procedure title   : PROConebyte_square
Variables required   : byte, addr
Line numbers   : 5270 to 5294
Length   : 27 bytes
Zero page requirements   : 2 bytes
Registers changed   : A, X, Y

Program 10.6

Procedure title   : PROCtwobyte_square

```
Variables required      : byte, temp, addr
Line numbers            : 5300 to 5332
Length                  : 39 bytes
Zero page requirements  : 4 bytes
Registers changed       : A, X, Y
```

Program 10.7

```
Procedure title         : PROCtwo_byte_asl
Variables required      : num, block, addr
Line numbers            : 5350 to 5363
Length                  : 13 bytes
Zero page requirements  : 2 bytes
Registers changed       : A
```

Program 10.8

```
Procedure title         : PROCtwo_byte_lsr
Variables required      : num, block, addr
Line numbers            : 5370 to 5383
Length                  : 13 bytes
Zero page requirements  : 2 bytes
Registers changed       : A
```

Program 10.9

```
Procedure title         : PROCtwo_byte_ror
Variables required      : num, block, addr
```
106 The BBC Micro Machine Code Portfolio
```
Line numbers            : 5390 to 5403
Length                  : 13 bytes
Zero page requirements  : 2 bytes
Registers changed       : A
```

Program 10.10

```
Procedure title         : PROCtwo_byte_rol
Variables required      : num, block, addr
Line numbers            : 5410 to 5423
Length                  : 13 bytes
Zero page requirements  : 2 bytes
Registers changed       : A
```

Program 10.11

Procedure title	: PROCmulti_left
Variables required	: start, bytes, addr
Line numbers	: 5430 to 5446
Length	: 16 bytes
Zero page requirements	: 1 byte
Registers changed	: A, X, Y

Chapter Eleven
Vision On

This chapter is devoted entirely to exploring the graphics capabilities of the Beeb from machine code. Many of the procedures are based on the VDU driver routine OSWRCH and all the graphics commands available from BASIC are implemented here plus a few more! These extras include two new screen modes which give scaled down versions of MODE 2 and MODE 5, plus a routine utilising the *640 table in the BASIC interpreter to convert an X, Y coordinate pair into the corresponding screen address.

I use many of these routines as part of a simple graphics compiler (SGC) which uses simple INPUT commands to call the appropriate PROC to compile the necessary machine code but on to the routines.

Just mode about you

Program 11.1 performs a mode change in machine code. This is done by sending the VDU value 22 to the driver followed by the mode number which should be passed into the procedure via 'action'. The assembled code is very short - just 11 bytes including the RTS.

```
  10 REM *** DO MODE ***
  20 CLS
  30 INPUT"Which MODE ?"M%
  40 PROCmode (M%,&C00)
  50 CALL mode
  60 PRINT"This is MODE ";M%
  70 END
  80 :
6000 DEF PROCmode (action,addr)
6001 P%=addr
6002 [
6003 .mode
6004           LDA #22
6005           JSR &FFEE
6006           LDA #action
6007           JSR &FFEE
6008           RTS
```

```
6009 ]
6010 ENDPROC
```

Program 11.1. PROCmode - performs a MODE change.

Program 11.2 provides a new screen mode. As it is made out of the MODE 2 screen I have christened it MODE 2A. This new mode still has all the sixteen colours of a normal MODE 2 available but only requires half the memory, 10K, for displaying them. The screen itself is composed of 25 rows of 20 characters. The program is given in its long-winded form so that I can try to explain its operation better! Obviously, it would be more economical in terms of memory to implement the final version with the VDU codes in a look-up table using an indexing routine to pull them out one by one and send them to OSWRCH.

```
  10 REM *** NEW MODE 2A SCREEN ***
  20 PROCmode2A (&A00)
  30 CALL &A00
  40 END
  50 :
6100 DEF PROCmode2A (addr)
6101 FOR PASS=0 TO 3 STEP3
6102 P%=addr
6103 [ :OPT PASS
6104          LDA #22
6105          JSR &FFEE
6106          LDA #2
6107          JSR &FFEE
6108          LDA #23
6109          JSR &FFEE
6110          LDA #0
6111          JSR &FFEE
6112          LDA #6
6113          JSR &FFEE
6114          LDA #25
6115          JSR &FFEE
6116          JSR SIX
6117          LDA #23
6118          JSR &FFEE
6119          LDA #0
6120          JSR &FFEE
6121          LDA #7
6122          JSR &FFEE
6123          LDA #30
6124          JSR &FFEE
6125          JSR SIX
6126          LDA #23
6127          JSR &FFEE
6128          LDA #0
6129          JSR &FFEE
```

```
6130              LDA #12
6131              JSR &FFEE
6132              LDA #8
6133              JSR &FFEE
6134              JSR SIX
6135              LDA #23
6136              JSR &FFEE
6137              LDA #0
6138              JSR &FFEE
6139              LDA #14
6140              JSR &FFEE
6141              LDA #8
6142              JSR &FFEE
6143              JSR SIX
6144              LDA #5
6145              STA &FE40
6146              LDA #56
6147              STA &302
6148              LDA #24
6149              STA &309
6150              LDA #40
6151              STA &D9
6152              STA &34B
6153              STA &34E
6154              STA &351
6155              STA &354
6156              LDA #&40
6157              STA &7
6158              LDA #0
6159              STA &6
6160              RTS
6161 .SIX
6162              LDA #0
6163              LDX #6
6164 .AGAIN
6165              JSR &FFEE
6166              DEX
6167              BNE AGAIN
6168              RTS
6169 ]
6170 NEXT
6171 ENDPROC
```

Program 11.2. PROCmode2A - implements a scaled down version of MODE
2.

It might be easier to understand exactly what is going on if the
assembler is broken down into its BASIC equivalent which, incidentally,
will also produce the desired effect.

Line 6104 to 6107 : MODE 2
Line 6108 to 6116 : VDU 23;6,25;0;0;0;
Line 6117 to 6125 : VDU 23;7,30;0;0;0;
Line 6126 to 6134 : VDU 23;12,8;0;0;0;
Line 6135 to 6143 : VDU 23;14,8;0;0;0;
Line 6144 to 6145 : ?&FE40=5
Line 6146 to 6147 : ?&302=56
Line 6148 to 6149 : ?&309=24
Line 6150 to 6151 : ?&D9=40
Line 6152 to 6155 : ?&34B=40
                       : ?&34E=40
                       : ?&351=40
                       : ?&354=40
Lines 6156 to 6159 : HIMEM=&4000

The VDU statements (lines 6108 to 6143) reprogram several registers of the 6845 cathode ray tube controller (CRTC) which is responsible for organising the screen memory. The BASIC equivalents show that the first and second parameter bytes are used in programming the CRTC. The first determines the CRTC register and the second the value to be written into it. Taking the four VDU23 statements in turn they perform the following tasks:

(a) Program number of lines
(b) Set position of vertical sync in number of row times
(c) Set top of screen address
(d) Set cursor position

The remaining pokes write to the VDU variables directly which, strictly speaking, is rather naughty! The poke to &FE40 is writing to the system VIA scroll-controlling register, while the subsequent two pokes define the bottom row, in pixels, of the graphics window and the bottom row of the text window. &D9 holds the high byte of the current address of the top scan line of a character (HIMEM is being set to &4000, thus the &40); &34B high byte of the top cursor location; &34E top+1 address of user memory; &351 the high byte address of the top left-hand corner of the screen; and finally &354 the high byte of the screen memory size.

Because the screen is not an official mode it is organised rather crookedly. For example, the pixel coordinates for the Y axis do not run from 0 to 1023 as one might expect but from 225 to 1023. Also, the screen itself tends to sit in the middle of the TV rather than using it all. To counteract the Y axis distortion, the graphics origin could be reset to 0,225 using VDU 29, thus:

VDU 29,0;225;
MOVE 0,0

This will reduce the maximum on-screen Y graphics coordinate to 798 but the range 0 to 798 is easier to use than 225 to 1023. Figure 11.1 provides a suitable map of MODE 2A.
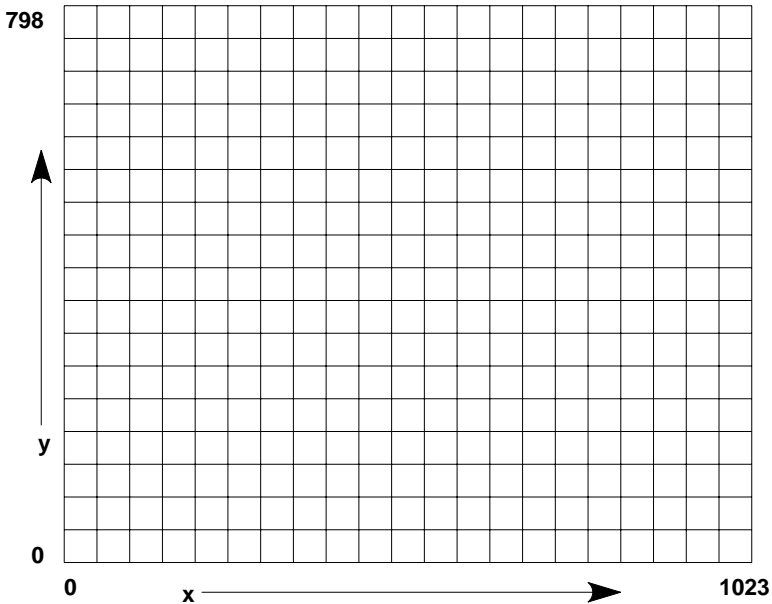

Fig. 11.1. The MODE 2A screen map.

Program 11.3 works along similar lines in that it pokes various VDU variables to set up a new graphics mode screen from MODE 5. However, rather than reprogramming the CRTC, it writes to the Video ULA using an OSBYTE call (lines 6027 to 6030). This writes, in fact, to the Video Control Register whose layout is given in Figure 11.2. The byte written is 224 or &E0 in hex, thus causing a large cursor two bytes in width to be displayed.

```
10 REM *** NEW MODE 5A ***
20 PROCmode5A (&A00)
30 DRAW1000,100
40 CALL &A00
50 MOVE 100,100
```

```
  60 DRAW 1000,100
  70 DRAW 1000,1000
  80 DRAW 100,1000
  90 DRAW 100,100
 100 END
 110 :
6020 DEF PROCmode5A (addr)
6021 P%=addr
6022 [
6023            LDA #22
6024            JSR &FFEE
6025            LDA #5
6026            JSR &FFEE
6027            LDA #154
6028            LDX #224
6029            JSR &FFF4
6030            LDA #15
6031            STA &360
6032            LDA #1
6033            STA &361
6034            LDA #32
6035            STA &34F
6036            LDA #&55
6037            STA &363
6038            LDA #&AA
6039            STA &362
6040            LDA #9
6041            STA &30A
6042            LDA #20
6043            JSR &FFEE
6044            LDA #&54
6045            STA &07
6046            LDA #0
6047            STA &6
6048            JMP &FFEE
6049 ]
6050 ENDPROC
```

Program 11.3. PROCmode5A - implements a scaled down version of MODE
5.

7	6	5	4	3	2	1	0
CURSOR SIZE BIT	CURSOR WIDTH BITS		6845 CLOCK BIT	CHARACTERS PER LINE BITS		TELETEXT SELECT BIT	FLASHING COLOUR BIT
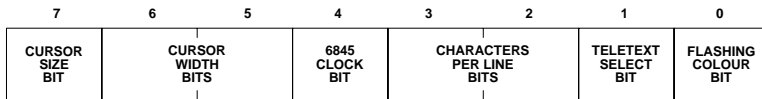
Fig. 11.2. The Video Control Register

This mode requires just 10K of RAM but also allows 16 colours like MODE 2 and MODE 2A! The mode allows 16 rows of 10 characters and HIMEM is set to &5400. The program description follows.

Lines 6023 to 6026: Select MODE 5.
Lines 6027 to 6029: Write to video ULA cursor control bits.
Lines 6030 to 6031: All 16 colours available.
Lines 6032 to 6033: Two 4-bit pixels per byte.
Lines 6034 to 6035: 32 bytes used per character.
Lines 6036 to 6039: Set colour details.
Lines 6040 to 6041: 10 characters on each line (0 to 9).
Lines 6042 to 6043: Do VDU 20 and rest default colours.
Lines 6044 to 6048: Set HIMEM = &5400.

Moving on

The three drawing-orientated processes, MOVE, DRAW and PLOT, can be performed using a VDU25 sequence, once again passing bytes through OSWRCH. After issuing the VDU25 sequence, OSWRCH expects five more bytes to be passed through to it. The first of these determines exactly what function is to be performed, while the remaining four bytes provide the double-byte values of first the X and then the Y coordinates, low bytes first.

Program 11.4 lists a suitable MOVE procedure. The MOVE code is 4 (line 6186) while the X, Y coordinates are passed for immediate addressing through the variables 'xpos' and 'ypos' .The demo uses the procedure to move the graphics cursor to the centre of the screen at 640,512 before plotting a point there (lines 20 to 50).

```
10 REM *** DO MACHINE CODE MOVE ***
20 PROCmove(640,512,&A00)
30 MODE 5
40 CALL move
50 DRAW 640,512
60 END
70 :
6180 DEF PROCmove(xpos,ypos,addr)
6181 P%=addr
6182 [
6183 .move
6184          LDA #25
6185          JSR &FFEE
6186          LDA #4
6187          JSR &FFEE
6188          LDA #xpos MOD 256
6189          JSR &FFEE
6190          LDA #xpos DIV 256
6191          JSR &FFEE
6192          LDA #ypos MOD 256
6193          JSR &FFEE
6194          LDA #ypos DIV 256
6195          JSR &FFEE
6196          RTS
6197 ]
6198 ENDPROC
```

Program 11.4. PROCmove - performs MOVE.

Program 11.5 uses the driver code 6 (line 6206) to execute the machine code equivalent of a DRAW. The positions passed into the procedure are taken to be the coordinates to draw to. The demo program draws a line diagonally across the MODE 4 screen from 0,0 to 1000,1000. Once again, immediate addressing is used in the program to obtain the X, Y coordinates which must therefore be passed into the procedure at assembly time.

```
  10 REM *** DO MACHINE CODE DRAW LINE
***
  20 PROCdraw(1000,1000,&C00)
  30 MODE 4
  40 MOVE 0,0
  50 CALL &C00
  60 END
  70 :
6200 DEF PROCdraw(xcord,ycord,addr)
6201 P%=addr
6202 [
6203 .draw_line
6204 LDA #25
```

```
6205 JSR &FFEE
6206 LDA #6
6207 JSR &FFEE
6208 LDA #xcord MOD 256
6209 JSR &FFEE
6210 LDA #xcord DIV 256
6211 JSR &FFEE
6212 LDA #ycord MOD 256
6213 JSR &FFEE
6214 LDA #ycord DIV 256
6215 JSR &FFEE
6216 RTS
6217 ]
6218 ENDPROC
```

Program 11.5. PROCdraw - performs DRAW.

```
10 REM *** DO MACHINE CODE PLOT ***
20 PROCplot(85,1000,1000,&C00)
30 MODE 4
40 MOVE 0,0
50 MOVE 1000,0
60 CALL plot
70 END
80 :
6220 DEF PROCplot(code,xcord,ycord,addr
)
6221 P%=addr
6222 [ OPT 2
6223 .plot
6224          LDA #25
6225          JSR &FFEE
6226          LDA #code
6227          JSR &FFEE
6228          LDA #xcord MOD 256
6229          JSR &FFEE
6230          LDA #xcord DIV 256
6231          JSR &FFEE
6232          LDA #ycord MOD 256
6233          JSR &FFEE
6234          LDA #ycord DIV 256
6235          JSR &FFEE
6236          RTS
6237 ]
6238 ENDPROC
```

Program 11.6. PROCplot - performs PLOT.

A PLOT is performed using the driver code which is equivalent to the

plot function required. Program 11.6 shows how the PLOT code is passed into the procedure through the variable 'code' .The demo uses code 85 to draw and fill a triangle in a MODE 4 screen. As you may now realise, the previous two programs were, in fact, simply using the plot codes for move and draw.

Paint-box

The use of colour is usually desirable for graphics and both COLOUR and GCOL can be readily performed. Program 11.7 can be used to redefine the text colour used by PRINT. It uses the VDU17 command with a second byte in the range 0 to 15 being passed to OSWRCH to define the colour. The number associated with each physical colour is detailed in Figure 11.3 and the chosen value should be passed to the procedure in the 'print_colour' variable. The demo sets up printing in red on a MODE 2 screen.

Number	Colour
0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White
8	Flashing black-white
9	Flashing red-cyan
10	Flashing green-magenta
11	Flashing yellow-blue
12	Flashing blue-yellow
13	Flashing magenta-green
14	Flashing cyan-red
15	Flashing white-black

Fig 11.3. The physical colours.

```
10 REM *** DO PRINT COLOUR ***
20 PROCcolour (1,&C00)
30 MODE 2
40 CALL colour
50 END
```

```
  60 :
6250 DEF PROCcolour(print_colour,addr)
6251 P%=addr
6252 [
6253 .colour
6254          LDA #17
6255          JSR &FFEE
6256          LDA #print_colour
6257          JSR &FFEE
6258          RTS
6259 ]
6260 ENDPROC
```

Program 11.7. PROCcolour - performs COLOUR.

Program 11.8 shows how the background colour can be redefined using
VDU17 again. Essentially the program is the same as its predecessor. To
stipulate a background colour, however, the most significant bit of the
colour byte must be set. In everyday terms, this simply means adding
128 to the colour value. After passing the background colour to the VDU
driver (lines 6276 to 6277) the screen must be cleared. This is facilitated
simply by printing the equivalent of a VDU12 (lines 6278 to 6279). The
demo program initialises a red MODE 2 screen.

```
  10 REM *** DO BACKGROUND COLOUR ***
  20 REM *** SET RED BACKGROUND ***
  30 PROCbackgrnd (129,&C00)
  40 MODE 2
  50 CALL backgrnd
  60 END
  70 :
6270 DEF PROCbackgrnd (back_col,addr)
6271 P%=addr
6272 [
6273 .backgrnd
6274 LDA #17
6275 JSR &FFEE
6276 LDA #back_col
6277 JSR &FFEE
6278 LDA #12
6279 JSR &FFEE
6280 RTS
6281 ]
6282 ENDPROC
```

Program 11.8. PROCbackgrnd - changes the mode background colour.

Performing GCOL is almost as easy, however. The GCOL statement requires two parameters. After issuing VDU18 first, the byte depicting the action required (i.e. AND, OR, EOR) should be passed to OSWRCH followed by the colour. These bytes are shown in Program 11.9 as 'action' and 'colour' and the associated demo program (lines 10 to 70) set up a flashing black and white diagonal line across the MODE 2 screen.

The graphics screen can be cleared from BASIC using the command CLG. In machine code this is simplicity it selfand only requires the vdu driver to print the code 16 through OSWRCH. Program 11.10 demonstrates this.

```
  10 REM *** DO MACHINE CODE GCOL ***
  20 REM *** FLASHING B/W LINE ***
  30 PROCgcol (0,8,&C00)
  40 MODE 2
  50 CALL gcol
  60 MOVE 0,0:DRAW 1000,1000
  70 END
  80 :
6285 DEF PROCgcol (action,colour,addr)
6286 P%=addr
6287 [
6288 .gcol
6289 LDA #18
6290 JSR &FFEE
6291 LDA #action
6292 JSR &FFEE
6293 LDA #colour
6294 JSR &FFEE
6295 RTS
6296 ]
6297 ENDPROC
```

Program 11.9. PROCgcol - performs GCOL.

```
  10 REM**CLEAR GRAPHICS SCREEN -CLG**
  20 PROCclg (&C00)
  30 MODE 2
  40 COLOUR129
  50 CLS
  60 PRINT"PRESS A KEY TO CLEAR SCREEN"
  70 A=GET
  80 CALL &C00
  90 END
 100 :
6300 DEF PROCclg (addr)
6301 P%=addr
```

```
6302 [
6303 .clear_graphics
6304 LDA #16
6305 JSR &FFEE
6306 RTS
6307 ]
6308 ENDPROC
```

Program 11.10. PROCclg - performs CLG.

Programming the palette is done as in BASIC using VDU 19 in the form:

VDU 19, log, phy, 0,0,0

where 'log' and 'phy' refer to the logical and physical colours respectively. Program 11.11 shows how this is translated into assembler. After the 19 is printed (lines 6314 and 6315) the logical and physical colour codes are passed to OSWRCH (lines 6316 to 6319) followed by the three padding zeros (lines 6320 to 6323) reserved for future expansion, whatever that is! Once again, the values passed into the procedure for 'log' and 'phy' are interpreted as immediate values by the assembler.

The lines 10 to 110 show how the procedure is used in this case to re-set the current screen background logical colour to each physical colour in turn.

```
  10 REM *** DO VDU 19 ***
  20 REM** GO FRU ALL COLOURS **
  30 MODE 2
  40 FOR loop=1 TO 15
  50 PROCchange_palette (0,loop,&C00)
  60 CALL chgpalette
  70 FOR N=0 TO 999:NEXT
  80 NEXT
  90 PROCchange_palette (0,0,&C00)
 100 CALL chgpalette
 110 END
 120 :
6310 DEF PROCchange_palette (log,phy,ad
dr)
6311 P%=addr
6312 [ OPT 2
6313 .chgpalette
6314 LDA #19
6315 JSR &FFEE
6316 LDA #log
6317 JSR &FFEE
```

```
6318 LDA #phy
6319 JSR &FFEE
6320 LDA #0
6321 JSR &FFEE
6322 JSR &FFEE
6323 JSR &FFEE
6324 RTS
6325 ]
6326 ENDPROC
```

Program 11.11. PROCchange_palette - reprograms the palette using OSWORD.

Read it write!

Occasionally it is useful to be able to know the last two sets of coordinates visited by the graphics cursor, so Acorn have implemented an OSWORD call to enable this feat. The call code is 13 and as with all OSWORD calls an address held with the index registers points to a parameter block where in this case OSWORD deposits the required information, Figure 11.4 details the information contained in the block after the call and Program 11.12 the technique. Lines 10 to 140 demonstrate the call by first moving the graphics cursor to a new positions on the MODE 4 screen before calling the procedure and reading its eight-byte result from the parameter block which in this instant is in zero page.

> XY+0 : previous X coordinate LSB
> XY+1 : previous X coordinate MSB
> XY+2 : previous Y coordinate LSB
> XY+3 : previous Y coordinate MSB
> XY+4 : current X coordinate LSB
> XY+5 : current X coordinate MSB
> XY+6 : current Y coordinate LSB
> XY+7 : current Y coordinate MSB

Fig. 11.4. OSWORD 13 parameter block for reading last two graphics coordinates.

```
10 REM *** READ LAST 2 GRAPHICS ***
20 REM *** CURSOR POSITIONS ***
30 MODE 4
40 MOVE 200,200
```

```
  50 DRAW 500,500
  60 MOVE 700,700
  70 DRAW 900,900
  80 CLS
  90 PROCgcursor(&70,&C00)
 100 CALL &C00
 110 FOR loop=&70 TO &77
 120 PRINT~loop;" ";?loop
 130 NEXT loop
 140 END
 150 :
6330 DEF PROCgcursor(block,addr%)
6331 FOR pass=0 TO 3 STEP3
6332 P%=addr%
6333 [ OPT pass
6334          LDA #13
6335          LDX #block MOD 256
6336          LDY #block DIV 256
6337          JSR &FFF1
6338          RTS
6339 ]
6340 NEXT
6341 ENDPROC
```

Program 11.12. PROCgcursor - uses OSWORD to read the last two
graphics coordinates.

The condition of any pixel on the screen can also be read using
OSWORD with the accumulator holding 9 - in effect, mimicking
BASIC's POINT command (see Program 11.13). Before calling the
operating system routine, the obligatory parameter block (detailed in
Figure 11.5) must have some relevant details placed into it, namely the
X,Y coordinates of the byte to be tested. Each coordinate uses two bytes
ofthe parameter block and these are derived in lines 6345 to 6361 ofthe
procedure. The procedure again assumes that the actual coordinates, and
not an address containing them, are passed through the variable X and Y.
Each byte is then stored in the relevant parameter block location. After
seeding the parameter block address into the index registers (lines 6362
to 6364) thc OSWORD call is performed leaving the logical colour of
the craordinate in the fifth block of the parameter block - or &FF if the
print was off of the screen.

```
  10 REM READ PIXEL VALUES
  20 PROCpixel(&70,100,100,&C00)
  30 MODE 2
  40 CALL&C00
  50 PRINT~block?4
```

```
  60 END
  70 :
6350 DEF PROCpixel(block,X,Y,addr)
6351 FOR pass=0 TO 3 STEP3
6352 P%=addr
6353 [ OPT pass
6354 LDA #X MOD 256
6355 STA block
6356 LDA #X DIV 256
6357 STA block+1
6358 LDA #Y MOD 256
6359 STA block+2
6360 LDA #Y DIV 256
6361 STA block+3
6362 LDX #block MOD 256
6363 LDY #block DIV 256
6364 LDA #9
6365 JSR &FFF1
6366 RTS
6367 ]
6368 NEXT
6369 ENDPROC
6332 P%=addr%
6333 [ OPT pass
6334 LDA #13
6335 LDX #block MOD 256
6336 LDY #block DIV 256
6337 JSR &FFF1
6338 RTS
6339 ]
6340 NEXT
6341 ENDPROC
```

Program 11.13. PROCpixel - reads the state of a screen pixel.

XY+0 : X coordinate LSB
XY+1 : X coordinate MSB
XY+2 : Y coordinate LSB
XY+3 : Y coordinate MSB
XY+4 : Logical colour of point, &FF if point off screen.

Fig. 11.5. OSWORD 9 parameter block to perform POINT.

```
10 REM *** READ COLOUR PALETTE ***
20 MODE 4
30 VDU19,1,3,0,0,0
```

```
  40 PROCreadpalette (&70,&C00,1)
  50 CALL &C00
  60 PRINT"Logical colour :";?&70
  70 PRINT"Physical colour :";?&71
  80 END
  90 :
6380 DEF PROCreadpalette (block, addr%,
L%)
6381 FOR pass=0 TO 3 STEP 3
6382 P%=addr%
6383 [ OPT pass
6384          LDA #L%
6385          STA block
6386          LDA #11
6387          LDX #block MOD 256
6388          LDY #block DIV 256
6389          JSR &FFF1
6390          RTS
6391 ]
6392 NEXT pass
6393 ENDPROC
```

Program 11.14. PROCreadpalette - reads the physical colour associated with a logical colour.

The colour palette can itself be read using an OSWORD 11 as shown in Program 11.14. The logical colour to be read should be placed into the five-byte parameter block. After the call, the physical colour currently assigned to the logical colour is in the second byte of the parameter block. The remaining three parameter block bytes contain zero - yes, for future expansion! The BASIC demo uses the call to read the physical colour assigned to logical colour 1 on the MODE 4 screen, this having been defined prior to the call in line 30 as 3.

Program 11.15 performs the operation in the reverse direction by writing to the palette using OSWORD 12. The parameter block is identical to that in a read operation except that the physical colour to be written must also be placed into the parameter block. The procedure passes both logical and physical colours to the assembler through the variables L% and PY%. The demo resets the MODE 4 logical colour 0, the background colour, to physical colour yellow, thereby performing an instant change in background colour.

```
  10 REM *** WRITE TO PALETTE ***
  20 MODE 4
  30 PROCwritepalette (&70,0,3,&C00)
  40 CALL &C00
  50 END
  60 :
6400 DEF PROCwritepalette (block,L%,PY%
```

```
,addr%)
6401 P%=addr%
6402 [
6403 LDA # L%
6404 STA block
6405 LDA # PY%
6406 STA block+1
6407 LDA #0
6408 STA block+2
6409 STA block+3
6410 STA block+4
6411 LDA #12
6412 LDX #block MOD 256
6413 LDY #block DIV 256
6414 JSR &FFF1
6415 RTS
6416 ]
6417 ENDPROC
```

Program 11.15. PROCwritepalette - performs VDU 19.

Co-ordinating screen addresses

The final routine in this chapter, Program 11.16, utilises the BASIC interpreter's *640 table at &C357 to convert an XY coordinate position on the screen (MODES 0, 1 and 2 only) into an absolute memory address. The table is a 32 byte by 2 byte affair which, unusually, is presented high byte first.

```
  10 REM ** CONVERT X,Y TO ADDRESS **
  20 PROCxyaddr(&80, &72, &71,&900)
  30 REPEAT
  40 INPUT"What is the X axis value - 0
to 79 "?&71
  50 INPUT''"What is the Y axis value -
0 to 255 "?&72
  60 CALLCODE
  70 PRINT~?&80+?&81*256
  80 UNTIL0
  90 :
6500 DEFPROCxyaddr(vector, yaxis, xaxis
, addr)
6501 FORI%=0TO2 STEP2
6502 P%=addr
6503 [ OPTI%
6504 .CODE
6505          LDA#0
6506          STA vector
```

```
6507             STA vector+2
6508             LDA#&30
6509             STA vector+1
6510             LDA yaxis
6511             AND#7
6512             STA yaxis +1
6513             EOR yaxis
6514             LSRA
6515             LSRA
6516             TAY
6517             INY
6518             LDA&C375,Y
6519             CLC
6520             ADC vector
6521             ADC yaxis +1
6522             STA vector
6523             DEY
6524             LDA&C375,Y
6525             ADC vector+1
6526             STA vector+1
6527             LDA xaxis
6528             LDX#3
6529 .LOOP
6530             ASLA
6531             ROL vector+2
6532             DEX
6533             BNE LOOP
6534             ADC vector
6535             STA vector
6536             LDA#0
6537             ADC vector+2
6538             ADC vector+1
6539             STA vector+1
6540             LDY#0
6541             LDA#&FF
6542             STA(vector),Y
6543             RTS
6544 ]
6545 NEXT
6546 ENDPROC
```

Program 11.16. PROCxy_addr - converts an X,Y coordinate into a screen address.

The program begins by clearing a few bytes of memory (lines 6505 to 6509) and setting vector to the start screen address. The MOD 8 value of the 'yaxis' is then calculated along with the DIV 8 value (lines 6510 to 6513). The actual value to be calculated is. in fact, Y DIV 8 *640. However, since the table values are two-byte the DIV is restricted to 4 (lines 6514 to 6516). The accumulator is transferred into the Y register

to get the index into the table, and is subsequently incremented to get the second, low byte (lines 6516 to 6518). The low byte is added to give Y axis MOD 8 (lines 6519 to 6522) and after extracting the high byte from the table this is added to give Y axis DIV 8 *640 (lines 6523 to 6527). Finally, the X axis value is multiplied by 8 and any bits falling off are caught in 'vector+3' (lines 6528 to 6533). This is then added to the low byte of the screen address to give the final address (lines 6534 to 6539). By way of demonstration, &F is then poked into screen memory at this point: to see this the program will need to be run in MODE 2 (lines 6540 to 6542).

Program fact sheets

Program 11.1

Procedure title	: PROCmode
Variables required	: action, addr
Line numbers	: 6000 to 6010
Length	: 11 bytes
Zero page requirements	: none
Registers changed	: A

Program 11.2

Procedure title	: PROCmode2A
Variables required	: addr
Line numbers	: 6100 to 6170
Length	: 153 bytes
Zero page requirements	: none
Registers changed	: A, X

Program 11.3

Procedure title	: PROCmode5A
Variables required	: addr
Line numbers	: 6020 to 6049
Length	: 58 bytes
Zero page requirements	: none
Registers changed	: A

Program 11.4

Procedure title	: PROCmove
Variables required	: xpos, ypos, addr
Line numbers	: 6180 to 6198
Length	: 31 bytes
Zero page requirements	: none
Registers changed	: A

Program 11.5

Procedure title	: PROCdraw
Variables required	: xcord, ycord, addr
Line numbers	: 6200 to 6218
Length	: 31 bytes
Zero page requirements	: none
Registers changed	: A

Program 11.6

Procedure title	: PROCplot
Variables required	: code, xcord, ycord, addr
Line numbers	: 6220 to 6238
Length	: 31 bytes
Zero page requirements	: none
Registers changed	: A

Program 11.7

Procedure title	: PROCcolour
Variables required	: print_colour, addr
Line numbers	: 6250 to 6260
Length	: 11 bytes
Zero page requirements	: none
Registers changed	: A

Program 11.8

Procedure title	: PROCbackgrnd
Variables required	: back_col, addr
Line numbers	: 6270 to 6282
Length	: 16 bytes
Zero page requirements	: none
Registers changed	: A

Program 11.9

Procedure title	: PROCgcol
Variables required	: action, colour, addr
Line numbers	: 6285 to 6297
Length	: 16 bytes
Zero page requirements	: none
Registers changed	: A

Program 11.10

Procedure title	: PROCclg
Variables required	: addr
Line numbers	: 6300 to 6308
Length	: 6 bytes
Zero page requirements	: norie
Registers changed	: A

Program 11.11

Procedure title	: PROCchange_palette
Variables required	: log, phy, addr
Line numbers	: 6310 to 6326
Length	: 27 bvtes
Zero page requirements	: none
Registers changed	: A

Program 11.12

Procedure title	: PROCgcursor
Variables required	: block, addr
Line numbers	: 6330 to 6341
Length	: 10 bytes
Zero pagc requirements	: none
Registers changed	: A, X, Y

Program 11.13

Procedure title	: PROCpixel
Variables required	: block, X, Y, addr
Line numbers	: 6350 to 6369
Length	: 26 bytcs

Zero page requirements : none
Registers changed : A, X, Y

Program 11.14

Procedure title : PROCreadpalette
Variables required : block, addr, L%
Line numbers : 6380 to 6393
Length : 14 bytes
Zero pagc requirements : none
Registers changed : A, X, Y

Program 11.15

Procedure title : PROCwritepalette
Variables required : block, L%, PY%, addr
Line numbers : 6400 to 6417
Length : 26 bytes
Zero page requirements : none
Registers changed : A, X, Y
Vision On129

Program 11.16

Procedure title : PROCxy_addr
Variables required : vector, yaxis, xaxis, addr
Line numbers : 6500 to 6545
Length : 69 bytes
Zero page requirements : 6 bytes
Registers changed : A, X, Y

Vision On **131**

Chapter Twelve
Assembling Data and Lists

Most programs written by most advanced BASIC programmers require the manipulation of data at some stage. Everyday life revolves around manipulating data and lists correctly. A telephone directory or an address book are samples of ordered lists (though a look at my address book with its loose and sellotaped pages would make you think otherwise!) whereby each entry is in alphabetical order. Searching through the pages for a particular address or phone number is quite simple. Imagine the problems if these entries were unordered.

Performing searches, adding and deleting items from lists and sorting in machine code is not as easy as its BASIC counterparts. The procedures in this chapter cover each of these aspects and should provide you with the basis for most of the data handling you require.

The programs provided in this chapter are:

Program 12.1: Byte search.
Program 12.2: Add a byte to an ordered list.
Program 12.3: Delete a byte from an ordered list.
Program 12.4: Find minimum and maximum values in an unordered list.
Program 12.5: Delete a byte from an unordered list.
Program 12.6: Access a byte in a one-dimensional byte array.
Program 12.7: Access a byte in a two-dimensional byte array.
Program 12.8: Access a word from a one-dimensional word array.
Program 12.9: Four-byte signed integer sort.
Program 12.10: Form new list from an old list of every nth element.
Program 12.11: Perform quicksort on a fbur-byte integer array.

Byte search

Program 12.1 provides a single-byte binary search algorithm through an ordered list. Just to clarify, an ordered list is a list in which its element

are arranged in an ascending order. For example,

    1,2,3,4,5,6. . .

would be an example of an ordered list, whereas

    4,9,2,6,8,12,34,2,1,0. . .

is an example of an unordered list.

   Because the list is ordered, it is not necessary for the machine code to search through the entire list. What the binary search technique does is to divide the list into half, calculate which half the search byte is in and divide this section in half again. This process continues until the search byte is located by zeroing in on it.

```
  10 REM *** SINGLE BYTE BINARY SEARCH
***
  20 PROCbin_search (&70,&71,&73,&74,&A
00)
  30 FOR loop=0 TO 150
  40 loop?&4001=loop
  50 NEXT loop
  60 ?&4000=150
  70 !&71=&4000
  80 ?&70=75
 100 CALL bin_search
 110 RESULT%=?&73
 120 IF RESULT%=0 PRINT"NOT FOUND" : EN
D
 130 PRINT"BYTE LOCATED AT +";RESULT%
 140 end
 150 :
7000 DEF PROCbin_search (byte,list,pos,
temp,addr)
7001 FOR PASS=0 TO 3 STEP 3
7002 P%=addr
7003 [
7004 OPT PASS
7005 .bin_search
7006          LDY #0
7007          LDA (list),Y
7008          STA pos
7009          STA temp
7010          INY
7011 .next_byte
7012          LSR pos
7013          BNE not_finished
7014          BCS over
7015          RTS
7016 .not_finished
```

```
7017              BCC over
7018              INC pos
7019 .over
7020              LDA (list),Y
7021              CMP byte
7022              BEQ byte_found
7023              BCS sub_inc
7024              TYA
7025              ADC pos
7026              CMP temp
7027              BEQ equal
7028              BCS next_byte
7029 .equal
7030              TAY
7031              JMP next_byte
7032 .sub_inc
7033              TYA
7034              SBC pos
7035              BEQ next_byte
7036              BCS set
7037              BMI next_byte
7038 .set
7039              TAY
7040              JMP next_byte
7041 .byte_found
7042              STY pos
7043              RTS
7044              ]
7045              NEXT
7046              ENDPROC
```

Program 12.1. PROCbin_search - performs a binary search on an ordered list.

The program searches the list looking for the 8-bit value held in 'byte'. The list is addressed indirectly so the vector 'list' is used to hold its address, &4000 in the demo. Note that the very first byte of the list is not, in fact, an element but the length of the list itself. The list proper therefore starts at (list)+1. The variable 'pos' is used to return the position of the element in the list; if this byte contains 0 it means that the element was not found. Remember that a value of 1 would be returned if the element was the very first in the list.

The binary search begins by obtaining the length of the list from the length of list element (lines 7005 to 7009). The search proper is then begun by executing a logical shift right on the list length byte in 'pos' (line 7012), thus dividing it by two. A result of zero indicates that the list does not contain the element being searched for and the RTS of line 7015 returns back to the calling routine leaving 'pos' holding zero. If the carry flag is set, control continues from 'over' (line 7019). The INC

instruction of line 7018 is used to round any odd numbers up to an even one should the division have left an odd value in 'pos'.

The byte comparison is nothing unusual. If the byte is found, the branch to 'byte_found' is performed (line 7022) where the Y register's contents art; placed in 'pos' and an RTS performed (lines 7041 to 7043). If the byte is not located then the program needs to determine which half of the section in which it is located contains the byte so that the program can halve that section. Assuming that the byte is larger than the element tested, the branch to 'sub_inc' is performed (line 7023). Here the current 'pos' is subtracted from the Y register, now transferred into the accumulator (lines 7032 to 7034) resulting in the lower portion of the list half being searched for the 'byte'. If, on the other hand, the byte is less than the element tested the branch does not take place and the 'pos' is added to the Y register so that the search continues in the upper section of the list half (lines 7024 to 7028).

The demo section of the program (lines 30 to 130) shows how the data needs to be set up before calling the subroutine. The procedural call assembles the routine at &A00 using five locations in zero page for variable storage, though only 'list' need be there. The FOR. . .NEXT loop then pokes an ordered list into memory from &4000 placing the number of elements in the list, 150, into the first byte (lines 30 to 60), before placing the address ofthe list in 'list' (line 70). The byte to be searched for - in this case 75 - is then poked into location &70 as this corresponds to 'byte'. After running the program, the result returned is

BYTE LOCATED AT +76

which is correct because 75+1=76.

An ordered addition

Figure 12.1 flowcharts very simply the steps required in adding an element to a list of ordered elements. It would be easy to look through each item in the list in turn, starting with the first element, moving onto the next and so forth until a number less than the byte and greater 134 The BBC Micro Machine Code Portfolio than the byte to be added is found. A space can be made for the byte by moving the distal portion of the list up memory by a byte, and the byte is inserted. This is not particularly efficient especially as we now have a binary search subroutine to hand! Program 12.2 combines Program 12.1, further illustrating the use of procedures to assemble segments of code, while the procedure PROCordered_add uses 'bin_search' as a subroutine call

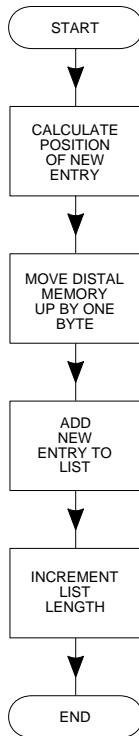(line 7155) to locate the desired position of the new element to be added.

Fig. 12.1. Flowchart for PROCordered_add.

```
  10 REM *** ADDITION OF AN ELEMENT TO
***
  20 REM *** AN ORDERED LIST
***
  30 PROCbin_search (&70,&71,&73,&74,&3
000)
  40 addr=P%
  50 PROCordered_add(&70,&71,&73,&74,ad
dr)
  60 FOR loop=0 TO 254 STEP2
  70 ?(&4000+(loop/2))=loop
```

```
  80 NEXT
  90 ?&4000=127
 100 !&71=&4000
 110 ?&70=221
 120 FOR N=&4000 TO (&4000+128)
 130 PRINT ~N;" ";?N
 140 NEXT
 150 PRINT"Press a key to execute "
 160 A=GET
 170 CALL add_element
 180 FOR N=&4000 TO (&4000+128)
 190 PRINT ~N;" ";?N
 200 NEXT
 210 END
 220 :
7100 DEF PROCbin_search (byte,list,pos,
temp,addr)
7101 FOR PASS=0 TO 3 STEP 3
7102 P%=addr
7103 [
7104 OPT PASS
7105 .bin_search
7106          LDY #0
7107          LDA (list),Y
7108          STA pos
7109          STA temp
7110          INY
7111 .next_byte
7112          LSR pos
7113          BNE not_finished
7114          BCS over
7115          RTS
7116 .not_finished
7117          BCC over
7118          INC pos
7119 .over
7120          LDA (list),Y
7121          CMP byte
7122          BEQ byte_found
7123          BCS sub_inc
7124          TYA
7125          ADC pos
7126          CMP temp
7127          BEQ equal
7128          BCS next_byte
7129 .equal
7130          TAY
7131          JMP next_byte
7132 .sub_inc
7133          TYA
7134          SBC pos
7135          BEQ next_byte
7136          BCS set
```

```
7137           BMI next_byte
7138 .set
7139           TAY
7140           JMP next_byte
7141 .byte_found
7142           STY pos
7143           RTS
7144 ]
7145 NEXT
7146 X=addr
7147 ENDPROC
7148 :
7149 DEF PROCordered_add(byte,list,pos,
temp,addr)
7150           FOR PASS=0 TO 3 STEP 3
7151           P%=addr
7152           [
7153           OPT PASS
7154 .add_element
7155           JSR bin_search
7156           LDA pos
7157           BNE present
7158           STY pos
7159           SEC
7160           LDA temp
7161           SBC pos
7162           TAX
7163           LDA byte
7164           CMP (list),Y
7165           BCS greater
7166           INY : INX
7167           JMP get_index
7168 .greater
7169           INC pos
7170           CPX #0
7171           BEQ enter_element
7172 .get_index
7173           LDY temp
7174 .next_element
7175           LDA (list),Y
7176           INY
7177           STA (list),Y
7178           DEY
7179           DEY
7180           DEX
7181           BNE next_element
7182 .enter_element
7183           LDA byte
7184           LDY pos
7185           STA (list),Y
7186           INC temp
7187           LDA temp
```

```
7188              LDY #0
7189              STA (list),Y
7190 .present
7191              RTS
7192 ]
7193 NEXT
7194 ENDPROC
```

Program 12.2. PROCordered_add - adds a value to an ordered list.

The PROCordered_add procedure (lines 7149 to 7194) uses the same variables/locations as the binary search procedure. Before it is called, the machine code assembled by the former expects to find the element to be added to the ordered list in 'byte' ,the location ofthe list in 'list' and the first byte ofthe list stating the number of elements in the list.

After the 'bin_search' subroutine call, the accumulator' scontents are loaded with 'pos'. Remember, if the list did not contain the byte being searched out, this will be zero. Ifthe byte is non-zero, the list already contains the element and need not be added - therefore the branch to 'present' (line 7157) is performed and the program completes. If the byte is zero the byte is not present, and the Y register holds the position of the last element to be examined before the search was exited. As it happens, this also corresponds to the position in the list where the binary search routine expected to find it! The Y register is therefore saved in 'pos' (line 7158) and the position whcrc the byte to be added is to one side of this element.

Before the routine locates exactly where the byte is to be added it must first calculate how many elements must be moved up a byte to make space for the new addition. This is really quite simple as it just requires 'pos' to be subtracted from 'temp' (lines 7159 to 7161), where 'temp' is used to hold the list's length. The result is then transferred across into the X register (line 7162) to act as a loop counter when the move takes place.

Calculating the exact position of the byte in the list is facilitated with a simple comparison with the element pointed to by the Y register index (line 7164). If this sets the carry flag, the position is immediately following the element pointed to by the index register Therefore the branch to 'greater' is performed (line 7165) where 'pos' is incremented. The compare X with zero instructions (lines 7170 to 7171) test to see whether the entry position will be outside the list in which case no space needs to be made for it. If the comparison clears the carry flag then after incrementing the X register (line 7166) a jump is performed.

Moving the upper section of the list is straightforward. Starting with the highest element, each byte is read, the Y register incremented and the byte stored (lines 7172 to 7177). Subtracting two from the Y register

restores the index at the next byte, while the X register acting as counter is decremented to signify one less element to move (lines 7178 to 7181). Ensuring that the move is performed in the reverse order, down memory, is important so as not to overwrite other elements in the list before they are also transferred!

Finally, the 'enter_element' routine pokes the new element into its correct position and the number of elements in the list is updated by adding one to it (lines 7182 to 7189).

The BASIC demo provides details on the ordered add routine's use. The two sections of assembler are assembled (lines 30 to 50) passing the value of the program counter through 'addr' to ensure that the two subroutines occupy successive bytes in memory. The FOR...NEXT loop then pokes an ordered list into memory from &4000 which consists of only even numbers (lines 60 to 80). Lines 90 and 100 set up the number of elements in the list and the 'list' vector itself. The value to be inserted into the list. 221 - an odd number - is then poked into 'byte'.

The entire contents of the list are then printed out to ensure that only even numbers in steps of two are present (lines120 to160). After the 'add _element' call the list is reprinted and the new element can be seen at the top of the screen (lines 170 to 210).

An ordered delete

Deleting an element from an ordered list is performed simply by using a slightly modified Program 12.2. All that is required is first to find the position of the element in the list using the 'bin_search' routine and then move all the elements distal to the byte down memory by one, thereby overwriting the byte to be deleted. Program 12.3 lists the delete program in its entirety while the BASIC demo is similar to the one previously described.

```
  10 REM *** DELETE AN ENTRY FROM WITHI
N ***
  20 REM *** AN ORDERED LIST
***
  30 PROCbin_search (&70,&71,&73,&74,&3
000)
  40 addr=P%
  50 PROCordered_del(&70,&71,&73,&74,ad
dr)
  60 FOR N%=0 TO 200
  70 ?(&4001+N%)=N%
  80 NEXT
```

```
  90 ?&70=179
 100 ?&4000=200
 110 !&71=&4000
 120 CALL del_element
 130 FOR N%=0 TO 200
 140 PRINT~(N%+&4001);" ";?(N%+&4001)
 150 NEXT
 160 END
 170 :
7200 DEF PROCbin_search (byte,list,pos,
temp,addr)
7201 FOR PASS=0 TO 3 STEP 3
7202 P%=addr
7203 [
7204 OPT PASS
7205 .bin_search
7206            LDY #0
7207            LDA (list),Y
7208            STA pos
7209            STA temp
7210            INY
7211 .next_byte
7212            LSR pos
7213            BNE not_finished
7214            BCS over
7215            RTS
7216 .not_finished
7217            BCC over
7218            INC pos
7219 .over
7220            LDA (list),Y
7221            CMP byte
7222            BEQ byte_found
7223            BCS sub_inc
7224            TYA
7225            ADC pos
7226            CMP temp
7227            BEQ equal
7228            BCS next_byte
7229 .equal
7230            TAY
7231            JMP next_byte
7232 .sub_inc
7233            TYA
7234            SBC pos
7235            BEQ next_byte
7236            BCS set
7237            BMI next_byte
7238 .set
7239            TAY
7240            JMP next_byte
7241 .byte_found
7242            STY pos
```

```
7243           RTS
7244 ]
7245 NEXT
7246 X=addr
7247 ENDPROC
7248 :
7249 DEF PROCordered_del(byte,list,pos,
temp,addr)
7250 FOR PASS=0 TO 3 STEP 3
7251 P%=addr
7252 [
7253 OPT PASS
7254 .del_element
7255           JSR bin_search
7256           LDA pos
7257           BEQ all_done
7258           INY
7259 .next_element
7260           LDA (list),Y
7261           DEY
7262           STA (list),Y
7263           INY
7264           INY
7265           CPY temp
7266           BCC next_element
7267           BEQ next_element
7268           LDA temp
7269           SBC #1
7270           LDY #0
7271           STA (list),Y
7272 .all_done
7273           RTS
7274           ]
7275           NEXT
7276           ENDPROC
```

Program 12.3. PROCordered_del - deletes a value from an ordered list.

A maximum minimum

The ability to be able to locate the maximum and minimum values in a list is important for example, in processing data from the ADVAL channels to determine a range of results. Program 12.4 performs this task on an unordered list; in an ordered list these would be the last and first bytes respectively in the list!

```
  10 REM *** FIND MINIMUM AND MAXIMUM *
**
```

```
  20 REM *** VALUES IN AN UNORDERED LIS
T ***
  30 PROCmax_min_list(&70,&71,&72,&C00)
  40 !&72=&4000
  50 FOR loop=1 TO 100
  60 loop?&4000=RND(255)
  70 NEXT
  80 ?&4000=100
  90 CALL minmax
 100 PRINT "Minimum value was :";?&70
 110 PRINT "Maximum value was :";?&71
 120 END
 130 :
7300 DEF PROCmax_min_list(min,max,list,
addr)
7301 FOR PASS=0 TO 3 STEP 3
7302 P%=addr
7303 [ OPT PASS
7304 .minmax
7305           LDY #0
7306           LDA (list),Y
7307           TAX
7308           INY
7309           LDA (list),Y
7310           STA min
7311           STA max
7312 .next_byte
7313           DEX
7314           BEQ all_done
7315           INY
7316           LDA (list),Y
7317           CMP min
7318           BCS test_max
7319           STA min
7320 .test_max
7321           CMP max
7322           BCC next_byte
7323           BEQ next_byte
7324           STA max
7325           JMP next_byte
7326 .all_done
7327           RTS
7328 ]
7329 NEXT
7330 ENDPROC
```

Program 12.4. PROCmax_min_list - finds the maximum and minimum values
in an unordered list.

   The routine finds these values by taking the first element from the list
and then using this initially as the maximum and minimum values. Then
each of the remaining elements in the list are compared in turn. If an

element is found to be larger than the present maximum value it becomes the new maximum value. Similarly, if an element is located that is smaller than the current minimum value it takes the current minimum value's place. When the last element in the list has been sampled, the maximum and minimum values have been located.

The 'minmax' routine performs this get and compare procedure. The address of the unordered list is held within the vector 'list' while the variables 'min' and 'max' are used as stores for the two extremes. As with the previous list operations, the first byte in the list holds its length. The subroutine begins by accessing this length byte and moving it across into the X register to act as a counter after which the first element is read and placed in the two variable stores (lines 7304 to 7311).

The main loop of the program is entered at line 7312. The X register counter is decremented, and if zero all the elements have been shifted through so the program exits (lines 7313 to 7314). The indexing register is incremented and the next element in the list sought (lines 7315 to 7316). The element in the accumulator is then tested against that in 'min'. If this clears the carry flag a smaller element is indicated so this is stored as the new minimum value. If the carry is set by the comparison a larger value than 'min' is indicated so a branch to 'test_max' is performed (lines 7318 to 7320). Here, a comparison against 'max' takes place. If the byte in the accumulator is found to be greater, it is stored at 'max' ,otherwise the next element in the list is sought out (lines 7321 to 7325).

The BASIC primer pokes 100 random single-byte values into a list starting at &4000 (lines 50 to 70) and the maximum and minimum values are ascertained by the 'minmax' routine.

An un-ordered delete

Program 12.5 shows how a byte can be deleted from an unordered list. Because the list is unordered, the binary search technique employed in the ordered lists cannot be used; instead, starting at the front of the list, each byte must be compared in turn. Once the byte is located, all that is required is to move all the distal elements remaining down through memory by a single byte, thus overwriting the deleted byte.

```
  10 REM *** DELETE ITEM FROM UNORDERED
LIST ***
  20 PROCunordered_del(&70,&71,&C00)
  30 ?&70=255
  40 !&71=&4000
  50 FOR N=1 TO 100
  60 ?(&4000+N)=N
```

```
  70 ?&4000=100
  80 NEXT
  90 ?&4050=255
 100 FOR N=&4000 TO &4064
 110 PRINT~N;" ";?N
 120 NEXT
 130 A=GET
 140 CALL &C00
 150 FOR N=&4000 TO &4064
 160 PRINT~N;" ";?N
 170 NEXT
 180 END
 190 :
7400 DEF PROCunordered_del(byte,list,ad
dr)
7401 FOR PASS=0 TO 3 STEP3
7402 P%=addr
7403 [ OPT PASS
7404           LDY #0
7405           LDA (list),Y
7406           TAX
7407           LDA byte
7408 .next
7409           INY
7410           CMP (list),Y
7411           BEQ delete
7412           DEX
7413           BNE next
7414           RTS
7415 .delete
7416           DEX
7417           BEQ updat
7418           INY
7419           LDA (list),Y
7420           DEY
7421           STA (list),Y
7422           INY
7423           JMP delete
7424 .updat
7425           LDA (list,X)
7426           SBC #1
7427           SYA (list,X)
7428           RTS
7429 ]
7430 NEXT
7431 ENDPROC
```

Program 12.5. PROCunordered_del - deletes a byte from an unordered list.

As with the other list processing programs, the address of the list is held in the vector 'list' while the first element in the list itself is its length. The byte to be deleted is placed in 'byte' prior to the call. Note that only the

first occurrence of the 'byte' is deleted, not all occurrences.

The BASIC program sets up an unordered list (well, it's actually ordered but who cares!) and then pokes the value 1155 into location &4050 (now its unordered!). The list is displayed both prior to and after the machine code call to show that the e]ement has indeed been deleted from the list (lines 30 to 170).

First bytes

In many respects, a one-dimensional byte array can be thought of simply as a list either ordered or unordered. The purpose of Program 12.6 is to calculate the absolute address of an element in the byte array by summing the array's base address and index, and then to extract that byte from the array.

The assembler requires two variables to be passed into it; 'subscript' is the index into the array while 'array' is its base address. Because 'subscript' is a single-byte value the array may only be a maximum of 256 bytes in length. The addition of 'array' and 'subscript' is performed in lines 7506 to 7511, then using indirect addressing the element is extracted and stored in 'subscript' (lines 7513 to 7515).

The BASIC tester sets up an ordered byte array at &4000 then extracts the 100th element (lines 20 to 90). Note that, unlike a list, the byte array does not need its length to be placed in the first byte of the array, which therefore starts from the address given by 'array' rather than this address plus one.

```
  10 REM **GET BYTE FROM BYTE ARRAY**
  20 PROCbyte_array (&70,&71,&C00)
  30 FOR array=0 TO 255
  40 ?(array+&4000)=array
  50 NEXT
  60 !&71=&4000
  70 ?&70=100
  80 CALL byte_array
  90 PRINT"Element in array was:";?&70
 100 END
 110 :
7500 DEF PROCbyte_array (subscript,arra
y,addr)
7501 FOR pass=0 TO 3 STEP 3
7502 P%=addr
7503 [
7504 OPT pass
7505 .byte_array
7506           LDA subscript
7507           CLC
```

```
7508              ADC array
7509              STA array
7510              BCC over
7511              INC array+1
7512 .over
7513              LDY #0
7514              LDA (array),Y
7515              STA subscript
7516              RTS
7517 ]
7518 NEXT
7519 ENDPROC
```

Program 12.6. PROCbyte_array - extracts a value from a one-dimensional byte array.

Accessing a byte from a two-dimensional byte array is a little less straightforward as there are two subscripts to take into consideration. These two subscripts are the row length and the column length. The resultant program is listed as Program 12.7 and basically it works by multiplying the row subscript by the row size and then adding the column subscript to its result. This result is then added to the base address of the array to give an absolute address.

```
  10 REM *** ACCESSING A TWO DIMENSIONA
L ***
  20 REM *** BYTE ARRAY ANYWHERE IN RAM
***
  30 PROCtwodim_byte(&70,&72,&74,&76,&7
8,&3000)
  40 FOR count=1 TO 32
  50 ?(count+&4000)=count
  60 NEXT
  70 !&70=2
  80 !&72=4
  90 !&74=8
 100 !&78=&4000
 110 CALL twodimbyte
 120 @%=0
 130 PRINT'''
 140 PRINT"Address of element in array:
&";
 150 PRINT~!&78 AND &FFFF
 160 PRINT'''"Byte located here :";
 170 PRINT?&70
 180 END
 190 :
7530 DEF PROCtwodim_byte (subscript1,su
bscript2,sub_size,temp,array,addr)
```

```
7531 FOR PASS=0 TO 3 STEP 3
7532 P%=addr
7533 [
7534 OPT PASS
7535 .twodimbyte
7536           LDA #0
7537           STA temp
7538           STA temp+1
7539           LDX #17
7540           CLC
7541 .multiply
7542           ROR temp+1
7543           ROR temp
7544           ROR subscript1+1
7545           ROR subscript1
7546           BCC no_add
7547           CLC
7548           LDA sub_size
7549           ADC temp
7550           STA temp
7551           LDA sub_size+1
7552           ADC temp+1
7553           STA temp+1
7554 .no_add
7555           DEX
7556           BNE multiply
7557           LDA subscript1
7558           CLC
7559           ADC subscript2
7560           STA subscript1
7561           LDA subscript1+1
7562           ADC subscript2+1
7563           STA subscript1+1
7564           LDA array
7565           CLC
7566           ADC subscript1
7567           STA array
7568           LDA array+1
7569           ADC subscript1+1
7570           STA array+1
7571           LDY #1
7572           LDA (array),Y
7573           STA subscript1
7574           RTS
7575 ]
7576 NEXT
7577 ENDPROC
```

Program 12.7. PROCtwodim_byte - extracts a value from a two-dimensional byte array.

The program commences by clearing two bytes at 'temp' which will act as a partial product during the multiplication procedure which is based on a standard shift and add type of affair. The X register is then initialised as a counter, to count out the shifts required during the multiplication (lines 7536 to 7539). The 'multiply' routine (lines 7541 to 7556) then performs the row subscript) (row length multiplication. When this is completed, the second subscript, the column, is added to the product of the multiplication (lines 7558 to 7563) and then to the base address of the array itself (lines 7565 to 7570). Finally, the array element is loaded into the accumulator and placed at 'subscript1'.

The BASIC program sets up the two-dimensional byte array using concurrent numbers. Of course, the array is stored physically in memory as a continuous list but is implemented as depicted in Figure 12.2, consisting of 4 rows of 8 columns. The position of any point in the array is given by (row,column), therefore the byte at (2,5) would be 22. The two-byte subscripts are poked into their two-byte locations at 'subscript1' and 'subscript2' (lines 70 and 80). Line 90 then places the row length into the two-byte variable 'sub_size'. After calling the machine code, the absolute address of the element is extracted from 'array'and the byte that is located there is printed (lines 110 to 170).

COLUMN

		0	1	2	3	4	5	6	7
ROW	**0**	1	2	3	4	5	6	7	8
	1	9	10	11	12	13	14	15	16
	2	17	18	19	20	21	22	23	24
	3	25	26	27	28	29	30	31	32

Fig. 12.2. Construction of a two-dimensional byte array.

Word arrays

Program 12.8 takes the one-dimensional byte array program a step further and implements the extraction of a two-byte word from a one-dimensional word array. As each word entry is two bytes long, all that the program needs to do to calculate the address of a particular element is to multiply the subscript by two and then add this to the base address of the array.

```
  10 REM *** GET WORD FROM SINGLE WORD
ARRAY ***
  20 PROCword_array(&70,&72,&C00)
  30 FOR array=0 TO 255
  40 ?(array+&4000)=array
  50 NEXT
  60 !&70=100
  70 !&72=&4000
  80 CALL word_array
  90 PRINT"Word element is :";!&70 AND
&FFFF
 100 END
 110 :
7600 DEF PROCword_array (subscript,arra
y,addr)
7601 FOR pass=0 TO 3 STEP 3
7602 P%=addr
7603 [
7604 OPT pass
7605 .word_array
7606           LDA subscript
7607           ASL A
7608           STA subscript
7609           LDA subscript+1
7610           ROL A
7611           STA subscript+1
7612           CLC
7613           LDA array
7614           ADC subscript
7615           STA array
7616           LDA array+1
7617           ADC subscript+1
7618           STA array+1
7619           LDY #0
7620           LDA (array),Y
7621           STA subscript
7622           INY
7623           LDA (array),Y
7624           STA subscript+1
7625           RTS
7626 ]
7627 NEXT
7628 ENDPROC
```

Program 12.8. PROCword_array - extracts a value from a one-dimensional
word array.

To perform the multiplication on the two-byte subscript, a two-byte shift
left is performed using the ASL/ROL combination (lines 7606 to 7611).
The double subscript is then added to 'array' (lines 7612 to 7618) and the

two-byte word extracted and placed in 'subscript' (lines 7619 to 7624).

Integer sort

Handling single-byte sorts is relatively easy but arrays of multi-byte numbers are more complex to handle. Program 12.9 provides an algorithm to sort a set of four-byte integer numbers stored consecutively in memory. Note that it assumes signed values.

```
  10 REM *** 4 BYTE SIGNED INTEGER SORT
***
  20 PROCsort32 (&70,&72,&74,&76,&C00)
  30 INPUT"How many numbers to sort ?"c
ount
  40 ?&76=count-1
  50 buffer=&4000
  60 ?&74=0:?&75=&40
  70 FOR random=0 TO count-1
  80 !(buffer+4*random)=RND
  90 NEXT random
 100 CALL &C00
 110 FOR look=0 TO count-1
 120 PRINT !(buffer+4*look)
 130 NEXT
 140 END
 150 :
7700 DEF PROCsort32 (one,two,vector,cou
nt,addr )
7701 FOR pass=0 TO 3 STEP 3
7702 P%=addr
7703 [
7704 OPT pass
7705 .entry
7706           LDA vector
7707           STA two
7708           LDA vector+1
7709           STA two+1
7710           LDA #0
7711           STA loop
7712 .once_more
7713           LDY #0
7714           LDA two+1
7715           STA one+1
7716           LDA two
7717           STA one
7718           CLC
7719           ADC #4
7720           STA two
7721           BCC no_inc
```

```
7722            INC two+1
7723 .no_inc
7724            LDX #4
7725            SEC
7726 .subtract
7727            LDA (two),Y
7728            SBC (one),Y
7729            INY
7730            DEX
7731            BNE subtract
7732            BVC vclear
7733            EOR #&80
7734 .vclear
7735            EOR #0
7736            BPL no_swap
7737            DEY
7738 .swap
7739            LDA (one),Y
7740            STA store
7741            LDA (two),Y
7742            STA (one),Y
7743            LDA store
7744            STA (two),Y
7745            DEY
7746            BPL swap
7747 .no_swap
7748            INC loop
7749            LDA loop
7750            CMP count
7751            BNE once_more
7752            DEC count
7753            BNE entry
7754            RTS
7755 .loop
7756            EQUS" "
7757 .store
7758            EQUS" "
7759 ]
7760 NEXT
7761 ENDPROC
```

Program 12.9. PROCsort32 - sorts a list of four-byte values.

The procedure passes four variables used by the assembler. The variable 'vector' is, as its name implies, a zero page vector that the machine code expects to contain the start address of the array. On entry to the code at 'entry' this address is passed into the two working vectors 'one' and 'two' (how's that for originality!), lines 7705 to 7711. The sort routine is, in fact, a 'bubble sort' procedure. This works by working through the numbers in the array and comparing sets of two contiguous

numbers. If the lower number is greater than the second number then they are swapped over. This process continues through all the numbers in the array until there are none left. The net effect is that numbers seem to bubble up through the array, thus the terminology. The disadvantage of a bubble sort is that it is very slow, although this is not so noticeable in machine code. However, it is not nearly as fast as the quicksort described later.

Let's get back to the program description. After seeding both vectors, the address in 'two' ,which points to the second of the two integers, is placed in 'one' (lines 7713 to 7717) and the vector 'two' is incremented by 4 to give it the address of the next vector in the array (lines 7718 to 7722). The X register is used as a byte counter so is initialised to 4 (line 7724) whereupon the integer 'one' is subtracted from the integer at 'two'. If on completion of the subtraction the overflow flag is set, it is necessary to reverse the sign of the most significant bit of the result now in the accumulator. This is performed in line 7733, and the EOR #&80 will set the negative flag to the value of the most significant bit of the accumulator. This process is particularly important as the negative flag is used to determine whether a swap is needed or not. If the flag is clear, no swap is indicated and a branch to 'no_swap' performed. The swap takes place, therefore, if the negative flag is set and is carried out by lines 7738 to 7746.

The final bytes of code (lines 7747 to 7753) test to see if the bubble sort has been completed, i.e. when a pass through it results in no swaps being performed after which control is passed back to BASIC. The BASIC demo routine at the start of the program pokes a random array of four-byte integers into a 'buffer' from &4000. After the sorting routine has been completed the array is displayed to show each four-byte integer in ascending order.

New lists for old

Program 12.10 uses an assembler routine to form a new list from an old one. What actually happens is that it extracts every nth item and places this in a list buffer elsewhere to form the list. This has many applications. For example, a list of four-byte integer numbers could be sorted into sub-lists of correspondingly significant bytes grouping all the most significant bytes together and so forth, which is useful if look-up tables are being used by another section of the program. The program is straightforward and the assembler expects three variables. The first 'source' is the address of the main list, while 'new' is the address of the new list. Note that these are not implemented as vectored addresses but

this could be used if desired. Finally, 'step' is the increment size determining the elements to be extracted.

After initialising the index registers, the main program loop is entered at 'next_byte'. The first byte of the source list is accessed and saved in the new list (lines 7777 to 7778). The 'new' list indexing register is then incremented and the 'source' list indexing register incremented by the value of 'step'. As the first byte of the source list holds its length, this byte is compared to the new index value to see if the end of the list has been passed. If not, the loop branches to repeat again, otherwise the contents of X are saved at 'new' to provide its length details.

```
  10 REM *** FORM NEW LIST FROM OLD ***
  20 PROCnew_list (&4000,&4200,5,&C00)
  30 FOR N=0 TO 100
  40 ?(&4001+N)=N
  50 NEXT
  60 ?&4000=100
  70 CALL &C00
  80 C=?&4200
  90 FOR N=0 TO C-1
 100 PRINT?(&4201+N)
 110 NEXT
 120 END
 130 :
7770 DEF PROCnew_list (source,new,step,
addr)
7771 FOR pass=0 TO 3 STEP 3
7772 P%=addr
7773 [ OPT pass
7774          LDY #0
7775          LDX #0
7776 .next_byte
7777          LDA source+1,Y
7778          STA new+1,X
7779          INX
7780          TYA
7781          CLC
7782          ADC #step
7783          TAY
7784          CMP source
7785          BCC next_byte
7786          STX new
7787          RTS
7788 ]
7789 NEXT
7790 ENDPROC
```

Program 12.10. PROCnew_list - creates a sublist from a main list.

A fast quicksort!

The final program in this chapter, Program 12.11, provides a very fast four-byte integer sorting routine based on the 'quicksort' algorithm, also known in some areas of Highbury as a fastsort!

   The quicksort procedure is less well known than the more illustrious bubble sort so a brief description of its working is probably useful. Consider the set of ten simple integers shown arranged randomly in Figure 12.3(a). One of these numbers is selected and is called the key. In the figure, the key is 46 and is shaded to make its position clear. Working from right to left, the key is in turn compared with each byte until a smaller byte is encountered. First time through, the first smaller value encountered is the third one in, 24. The key then swaps positions with this byte as shown in Figure 12.3(b). Next, the search process is repeated but this time, working from left to right, the first byte encountered that is larger than the key is swapped with the key 70 in this instance (Figure 12.3(c)). The process repeats again until no more swaps are possible, as Figure 12.3(d) shows.

(a) | 46 | 35 | 30 | 70 | 39 | 75 | 48 | 24 | 63 | 50 |

(b) | 24 | 35 | 30 | 70 | 39 | 75 | 48 | 46 | 63 | 50 |

(c) | 24 | 35 | 30 | 46 | 39 | 75 | 48 | 70 | 63 | 50 |

(d) | 24 | 35 | 30 | 39 | 46 | 75 | 48 | 70 | 63 | 50 |
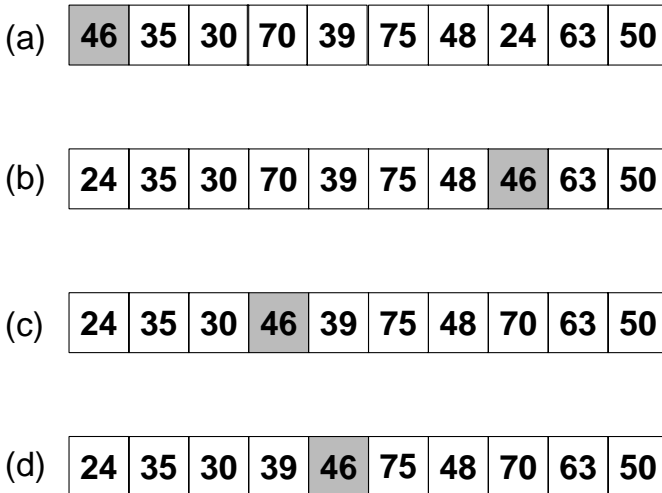
Fig. 12.3. (a) Assigning the key in a quickshort. (b) The array after the first quicksort pass. (c) The array after the second quicksort pass. (d) The array after the third quicksort pass.

   Looking at Figure 12.3(d) carefully shows that it is divided into two halves. All the numbers on the left of the key are smaller than the key itself while those on the right are larger. In other words, the key has now found its final position in the list. The two sections of the list can now be sorted independently using new keys, and then the sections these provide and so on until the quicksort is completed.

   Because the number of elements to be sorted reduces each time through the quicksort the time taken for it to complete its task is substantially quicker than the bubble sort method described earlier which processes the whole array each time through. In fact, to sort 1000 four-byte integers using the bubble sort would take around 50 seconds, compared with under two seconds for the quicksort. A BASIC version of the quicksort is only slightly slower than the assembler bubble sort!

```
  10 REM *** FOUR BYTE INTEGER FASTSORT
***
  20 PROCquick(&3900,&70,&72,&74,&76,&7
8,&7A,&80)
  30 !&80=&5000:!(&80+2)=20
  40 FOR N=0 TO 20 STEP4
  50 !(&5000+N)=RND:NEXT
  60 CALL fastsort
  70 FOR N=0TO 20 STEP4
  80 PRINT!(&5000+N):NEXT
  90 END
 100 :
7800 DEF PROCquick(softstk,left,right,c
urrent_left,current_right,stack,middle,d
ata)
7801 FOR pass=0 TO 3 STEP 3
7802 P%=&4000
7803 [ OPT pass
7804 .fastsort
7805          LDA #softstk MOD 256
7806          STA stack
7807          LDA #softstk DIV 256
7808          STA stack+1
7809          LDA data+2
7810          STA left
7811          LDA data+3
7812          STA left+1
7813          LDY #1
7814 .setup
7815          LDA (left),Y
7816          STA current_left,Y
7817          DEY
7818          BPL setup
7819          LDY #2
7820 .shift_two
```

```
7821              ASL current_left
7822              ROL current_left+1
7823              DEY
7824              BNE shift_two
7825              LDA data
7826              CLC
7827              ADC current_left
7828              STA right
7829              LDA data+1
7830              ADC current_left+1
7831              STA right+1
7832              LDA data
7833              SEC
7834              SBC #4
7835              STA left
7836              LDA data+1
7837              SBC #0
7838              STA left+1
7839 .save_value
7840              LDA left
7841              CLC
7842              ADC #4
7843              STA current_left
7844              LDA left+1
7845              ADC #0
7846              STA current_left+1
7847              LDA right
7848              STA current_right
7849              SEC
7850              SBC current_left
7851              BNE over
7852              LDA right+1
7853              SBC current_left+1
7854              BNE over
7855              JMP pull
7856 .over
7857              LDA right+1
7858              STA current_right+1
7859              JSR swap
7860              LDY #3
7861 .back
7862              LDA (current_left),Y
7863              STA key,Y
7864              DEY
7865              BPL back
7866 .adjust_right
7867              LDA current_right
7868              SEC
7869              SBC #4
7870              STA current_right
7871              BCS compare_hiright
7872              DEC current_right+1
7873 .compare_hiright
```

```
7874            LDA current_left+1
7875            CMP current_right+1
7876            BCC not_right
7877            LDA current_left
7878            CMP current_right
7879            BEQ equal
7880 .not_right
7881            LDX #4
7882            LDY #0
7883            SEC
7884 .compare_keyright
7885            LDA (current_right),Y
7886            SBC key,Y
7887            INY
7888            DEX
7889            BNE compare_keyright
7890            BVC no_mask
7891            EOR #&80
7892 .no_mask
7893            AND #&FF
7894            BPL adjust_right
7895            DEY
7896 .exchange
7897            LDA (current_right),Y
7898            STA (current_left),Y
7899            DEY
7900            BPL exchange
7901 .adjust_left
7902            LDA current_left
7903            CLC
7904            ADC #4
7905            STA current_left
7906            BCC compare_lefthigh
7907            INC current_left+1
7908 .compare_lefthigh
7909            LDA current_left+1
7910            CMP current_right+1
7911            BCC not_left
7912            LDA current_left
7913            CMP current_right
7914            BEQ equal
7915 .not_left
7916            LDX #4
7917            LDY #0
7918            SEC
7919 .compare_keyleft
7920            LDA key,Y
7921            SBC (current_left),Y
7922            INY
7923            DEX
7924            BNE compare_keyleft
7925            BVC no_mask_again
```

```
7926              EOR #&80
7927 .no_mask_again
7928              AND #&FF
7929              BPL adjust_left
7930              DEY
7931 .exchange_over
7932              LDA (current_left),Y
7933              STA (current_right),Y
7934              DEY
7935              BPL exhange_over
7936              BMI adjust_right
7937 .equal
7938              LDY #3
7939 .exc_loop
7940              LDA key,Y
7941              STA (current_left),Y
7942              DEY
7943              BPL exc_loop
7944              LDA current_left
7945              SEC
7946              SBC left
7947              STA word
7948              LDA current_left+1
7949              SBC left+1
7950              STA word+1
7951              LDA right
7952              SEC
7953              SBC current_left
7954              STA temp
7955              LDA right+1
7956              SBC current_left+1
7957              STA temp+1
7958              LDA word
7959              SEC
7960              SBC temp
7961              LDA word+1
7962              SBC temp+1
7963              BCC save_hi
7964 .save_lo
7965              LDY #0
7966              LDA left
7967              STA (stack),Y
7968              INY
7969              LDA left+1
7970              STA (stack),Y
7971              INY
7972              LDA current_left
7973              STA (stack),Y
7974              STA left
7975              INY
7976              LDA current_left+1
7977              STA (stack),Y
7978              STA left+1
```

```
7979            JMP update
7980 .save_hi
7981            LDY #2
7982            LDA right
7983            STA (stack),Y
7984            INY
7985            LDA right+1
7986            STA (stack),Y
7987            INY
7988            LDA current_left
7989            STA (stack),Y
7990            STA right
7991            INY
7992            LDA current_left+1
7993            STA (stack),Y
7994            STA right+1
7995 .update
7996            CLC
7997            LDA stack
7998            ADC #4
7999            STA stack
8000            BCC update1
8001            INC stack+1
8002 .update1
8003            JMP save_value
8004 .pull
8005            LDA stack
8006            SEC
8007            SBC #softstk MOD 256
8008            BNE pull1
8009            LDA stack+1
8010            SBC #softstk DIV 256
8011            BNE pull1
8012            RTS
8013 .pull1
8014            LDA stack
8015            SEC
8016            SBC #4
8017            STA stack
8018            BCS pull2
8019            DEC stack+1
8020 .pull2
8021            LDY #3
8022 .pull3
8023            LDA (stack),Y
8024            STA left,Y
8025            DEY
8026            BPL pull3
8027            JMP save_value
8028 .swap
8029            LDA current_right
8030            SEC
```

```
8031              SBC current_left
8032              AND #&F8
8033              STA middle
8034              LDA current_right+1
8035              SBC current_left+1
8036              STA middle+1
8037              LSR middle+1
8038              ROR middle
8039              LDA current_left
8040              CLC
8041              ADC middle
8042              STA middle
8043              LDA current_left+1
8044              ADC middle+1
8045              STA middle+1
8046              LDY #3
8047 .swap_loop
8048              LDA (current_left),Y
8049              STA word
8050              LDA (middle),Y
8051              STA (current_left),Y
8052              LDA word
8053              STA (middle),Y
8054              DEY
8055              BPL swap_loop
8056              RTS
8057 .key  EQUD 0
8058 .word EQUW 0
8059 .temp EQUW 0
8060 ] NEXT
8061 ENDPROC
```

Program 12.11. PROCquick - implements a four-byte quickshort routine

The main areas operation of Program 12.11 are as follows:

Lines 7805 to 7808: Set up vector software stack which will be used to hold pointers and data for future reference by the routine.
Lines 7809 to 7813: Save base address of the integer array in 'left'.
Lines 7814 to 7818: Seed integer into the 'current_left'position.
Lines 7819 to 7855: Seed next integer into 'current_right' position and test to see if they are equal. If both are equal, then pull pointers and data from stack and move onto next subsection.
Lines 7856 to 7860: If items are not equal perform the swap.
Lines 7861 to 7865: Then save key for future reference.
Lines 7866 to 7900: Compare key with integers to the right of it and perform the swap if greater.
Lines 7901 to 7936: Compare key with integers to the left of it and

perform swap if less than.

Lines 7937 to 8043: Place key back in 'current_left'.

Lines 7944 to 7963: Now save pointers of unsorted sections of integer array on software stack. Concentrate on smallest section first.

Lines 7964 to 8003: Routine to push pointers and data items onto software stack.

Lines 8004 to 8027: Routine to pull all pointers and data items off software stack.

Lines 8028 to 8055: Subroutine to perform the key integer swap.

Two sets of two variables are used by the assembler routine. The boundaries of the current section half being sorted are held in 'left' and 'right'. The current left- and right-hand numbers being tested with the key are held in 'current_left'and 'current_right' respectively. The BASIC primer sets up an array of random integer values at &5000. After calling the quickshort, the sorted array is displayed (lines 20 to 90).

Program fact sheets

Program 12.1

Procedure title	: PROCbin_scarch
Variables required	: bvte, list. pos, temp, addr
Line numbers	: 7000 to 7046
Length	: 57 bytes
Zero page requirements	: 5 bytes
Registers changed	: A, X, Y

Program 12.2

Procedure title	: PROCordered_add (also requires PROCbin_search)
Variables required	: byte, list, pos, temp, addr
Line numbers	: 7100 to 7194
Length	: 115 bytes
Zero page requirements	: 5 bytes
Registers changed	: A, X, Y

Program 12.3

Procedure title	: PROCordered_del (also requires PROCbin_search)
Variables required	: byte, list, pos, temp, addr

Line numbers   : 7200 to 7276
Length   : 87 bytes
Zero page requirements : 5 bytes
Registers changed   : A, X, Y

Program 12.4

Procedure title   : PROCmax_min_list
Variables required   : min, max, list, addr
Line numbers   : 7300 to 7330
Length   : 36 bytes
Zero page requirements : 4 bytes
Registers changed   : A, X, Y

Program 12.5

Procedure title   : PROCunordered_del
Variables required   : byte, list, addr
Linc numbers   : 7400 to 7431
Length   : 36 bytes
Zero page requirements : 3 bytes
Registers changed   : A, X, Y

Program 12.6

Procedure title   : PROCbyte_array
Variables required   : subscript, array, addr
Line numbers   : 7500 to 7519
Length   : 18 bytes
Zero page requirements : 3 bytes
Registers changed   : A, Y

Program 12.7

Procedure title   : PROCtwodim_byte
Variables required   : subscriptl, subscript2, sub_size, temp,
                           array, addr
Line numbers   : 7530 to 7577
Length   : 68 bytes
Zero page requirements : I0 bytes
Registers changed   : A, X, Y

Program 12.8

Procedure title	: PROCword_array
Variables required	: subscript, array, addr
Line numbers	: 7600 to 7628
Length	: 34 bvtes
Zero page requirements	: 3 byies
Registers changed	: A, X, Y

Program 12.9

Procedure title	: PROCsort32
Variables required	: one, two, vector, count, addr
Line numbers	: 7700 to 7761
Length	: 86 bytes
Zero page requirements	: 8 bytes
Registers changed	: A, X, Y

Program 12.10

Procedure title	: PROCnew_list
Variables required	: source, new, step, addr
Line numbers	: 7770 to 7790
Length	: 25 bytes
Zero page requirements	: 1 byte
Registers changed	: A, X, Y

Program 12.11

Procedure title	: PROCquick
Variables requircd	: softstk, left, right, current_left, current_right, stack, middle, data
Line numbers	: 7800 to 8061
Length	: 437 bytes
Zero page requirements	: 14 bytes
Registers changed	: A, X, Y

Chapter Thirteen
Communication

An important part of good software is good communication between the program and the user. Excellent software is very often degraded because the writer has not made any attempt to make the program user-friendly by presenting instructions neatly onto the screen and using sensible keys for inputting information. This chapter provides some short but important procedures that will enable the programmer both to neaten up screen presentation and have the program input data with minimal fuss. Most programs are based around the excellent set of operating system commands, thus reducing the amount of coding.

The routines included are:

Program 13.1 : Perform a CLS.
Program 13.2 : Perform VPOS and POS.
Program 13.3 : Perform SPC.
Program 13.4 : Perform STRING$.
Program 13.5 : Perform TAB (X).
Program 13.6 : Perform TAB (X,Y).
Program 13.7 : Perform GET.
Program 13.8 : Perform INKEY.
Program 13.9 : Super quick key test.
Program 13.10 : Read line.
Program 13.11 : Read a VDU definition.
Program 13.12 : Hexadecimal to ASCII.
Program 13.13 : Packed BCD to ASCII.
Program 13.14 : ASCIl to packed BCD.

Onto the screen

The first six programs presented here deal with formatting text on the screen. Program 13.1 performs a simple CLS to clear the screen, by printing a control code 12 through OSWRCH.

```
  10 REM*** CLEAR SCREEN -CLS ***
  20 PROCcls (&C00)
  30 PRINT"PRESS A KEY TO CLEAR SCREEN"
  40 A=GET
  50 CALL &C00
  60 END
  70 :
8000 DEF PROCcls (addr)
8001 P%=addr
8002 [
8003 .clear_screen
8004           LDA #12
8005           JSR &FFEE
8006           RTS
8007 ]
8008 ENDPROC
```

Program 13.1. PROCcls - performs CLS

Knowing where the cursor is at a particular time is another useful function to perform. In BASIC, the POS and VPOS functions return the horizontal (X axis) and vertical (Y axis) components of the cursor. These two functions have a direct equivalent within the MOS, an OSBYTE &86 call. Program 13.2 shows that the X and Y coordinates of the cursor are returned in the respective index registers which can be saved for evaluation.

```
  10 REM *** DO POS & VPOS ***
  20 PROCcursor (&70,71,&C00)
  30 CLS
  40 PRINTTAB(10,10);
  50 CALLcursor
  60 PRINT'"Cursor positions were :"
  70 PRINT"POS = ";?&70
  80 PRINT"VPOS= ";?&71
  90 END
 100 :
8010 DEF PROCcursor (pos,vpos,addr)
8011 P%=&C00
8012 [ OPT 2
8013 .cursor
8014           LDA #&86
8015           JSR &FFF4
8016           STX pos
8017           STY vpos
8018           RTS
```

```
8019 ]
8020 ENDPROC
```

Program 13.2. PROCcursor - returns the X,Y coordinates of the text cursor.

Program 13.3 imitates the BASIC command SPC, to print a specified number of spaces from the current cursor position. The number of spaces to be printed should be passed into the procedure via the variable 'spc'. At assembly time, this variable is treated as an immediate value being loaded directly into the X register to act as a simple loop control. Prior to entering 'loop', the accumulator is loaded with 32, the ASCIl code of a space, and the required number is printed.

```
  10 REM *** DO MACHINE CODE SPC ***
  20 CLS
  30 INPUT"How many spaces ?"spc
  40 PROCspace (spc,&C00)
  50 CALL space
  60 END
  70 :
8030 DEF PROCspace (spc,addr)
8031 P%=addr
8032 [ OPT 2
8033 .space
8034          LDX #spc
8035          LDA #32
8036 .loop
8037          JSR &FFEE
8038          DEX
8039          BNE loop
8040          RTS
8041 ]
8042 ENDPROC
```

Program 13.3. PROCspace - performs SPC.

STRING$ is a BASIC command that allows a specified number of the same string to be printed. This can result in a great saving of memory space. For example, it is much neater to print 40 asterisks using

   PRINT STRING$ (40,"*")

rather than enclosing 40 asterisks inside quotes or from a machine code point of view in an ASCII data table. Program 13.4 emulates this

command by printing a string located at 'buffer', 'num' number of times.

```
  10 REM *** DO STRING$ ***
  20 CLS
  30 INPUT"Enter your string :"$&C50
  40 INPUT"How many times :"num
  50 PROCstring (&C50,num,&C00)
  60 CALL string
  70 END
  80 :
8050 DEF PROCstring (buffer,num,addr)
8051 FOR pass=0 TO 2 STEP 2
8052 P%=addr
8053 [ OPT pass
8054 .string
8055           LDY #num
8056 .count
8057           LDX #0
8058 .next_chr
8059           LDA buffer,X
8060           JSR &FFE3
8061           INX
8062           CMP #13
8063           BNE next_chr
8064           DEY
8065           BNE count
8066           RTS
8067 ]
8068 NEXT
8069 ENDPROC
```

Program 13.4. PROCstring - performs STRING$

The program commences by loading the Y register with the 'num' count (line 8055) and then setting the X register to zero (line 8057), which is to be used as an absolute index into 'buffer'. The get and display loop is embodied in lines 8058 to 8063. Each character is extracted from 'buffer' and printed in turn until a carriage return has been printed. The Y register is decremented and the 'count' loop repeated until it reaches zero. Note that in the code the carriage return at the end of the string (put there by BASIC's INPUT statement - line 30) is printed, so that each string occupies a new line rather than being printed continuously across the screen. The CMP #13 test could be performed earlier so that the program exists before issuing a return if so required.

Positioning text on the screen is performed using the TAB function. There are two bytes of TAB, namely TAB(X) and TAB(X,Y). The simplest way to perform a TAB(X) is to print the move cursor right

code, ASCII 9, the required number of times. Program 13.5 details the assembler, the required value of X passing into the procedure through 'xpos'. Once again, a simple loop is used printing the code 9 through OSWRCH until the X register has been decremented to zero. It is worth bearing in mind that this routine is in fact different from the one given in Program 13.3 as it has no effect on any text the cursor passes over. Only the cursor is moved and no spaces are output as in the former program.

```
  10 REM *** DO TAB(X)***
  20 CLS
  30 INPUT"Number of tabs :"xpos
  40 PROCtabx(xpos,&C00)
  50 CLS
  60 CALL tabx
  70 PRINT"*"'
  80 PRINT"* marks the TAB position"
  90 END
 100 :
8080 DEF PROCtabx (xpos,addr)
8081 P%=addr
8082 [ OPT 2
8083 .tabx
8084           LDA #9
8085           LDX #xpos
8086 .xtab
8087           JSR &FFEE
8088           DEX
8089           BNE xtab
8090           RTS
8091 ]
8092 ENDPROC
```

Program 13.5. PROCtabx - performs TAB(X)

TAB(X,Y) is performed through OSWRCH using the driver code 31 followed by first the X and then the Y coordinate to tab to. Program 13.6 uses immediate addressing to pass the two TAB parameters into the assembler via 'xpos'and 'ypos'.

```
  10 REM *** DO TAB(X,Y) ***
  20 CLS
  30 INPUT"Tab X position :"xpos
  40 INPUT"Tab Y position :"ypos
  50 PROCtabxy (xpos,ypos,&C00)
  60 CLS
```

```
  70 CALL tabxy
  80 PRINT"*"'
  90 PRINT"* marks the TAB position"
 100 END
 110 :
8100 DEF PROCtabxy (xpos,ypos,addr)
8101 P%=addr
8102 [ OPT 2
8103 .tabxy
8104          LDA #31
8105          JSR &FFEE
8106          LDA #xpos
8107          JSR &FFEE
8108          LDA #ypos
8109          JSR &FFEE
8110          RTS
8111 ]
8112 ENDPROC
```

Program 13.6. PROCtabxy - performs TAB(X,Y).

The key to detection

I doubt if there are many programs that do not require some sort of interaction from the user at the keyboard, whether it be a simple 'press key to continue' affair or more complex data input. Whatever it is, the need to perform the task efficiently and correctly is important. A simple GET type keyboard read can be performed by calling OSRDCH at &FFE9 directly (see Program 13.7). This call waits for a key to be pressed and returns with its ASCII value in the accumulator. However, when using this call it is important to test to see if the ESCAPE key was the key pressed. This can be performed simply by comparing the accumulator contents with &1B (line 8126). If the key was pressed then it must be acknowledged with an OSBYTE &7E call (lines 8128 to 8129) otherwise the MOS will hang up or do crazy things!

```
  10 REM *** TEST FOR KEY **
  20 CLS
  30 PRINT"Press key to test for";
  40 chr$=GET$
  50 chr=ASC(chr$)
  60 PROCgetkey(chr,&C00)
  70 PRINT''"Press ";chr$;" to end";
  80 CALL getstring
  90 PRINT'''"Finished!"
 100 END
```

```
 110 :
8120 DEF PROCgetkey (chr,addr)
8121 FOR pass=0 TO 2 STEP 2
8122 P%=addr
8123 [ OPT pass
8124 .getstring
8125          JSR &FFE0
8126          CMP #&1B
8127          BNE no_escape
8128          LDA #&7E
8129          JSR &FFF4
8130 .no_escape
8131          CMP #chr
8132          BNE getstring
8133          RTS
8134 ]
8135 NEXT
8136 ENDPROC
```

Program 13.7. PROCgetkey - performs GET.

Using OSBYTE &81, an INKEY timed input can be performed. The index registers are used to hold the wait period which is specified in centiseconds. Program 13 .8 shows how it is set up in the procedure PROCinkey. Prior to the actual OSBYTE call an *FX15,1 is performed to flush all input buffers (lines 8145 to 8148). The wait period is passed into the assembler via the variable 'wait', the high and low bytes are loaded into the respective registers usingthe MOD and DIV operators (lines 8149 to 8152). As with the previous procedure, the ESCAPE key should be tested for and acknowledged with the appropriate call if need be (lines 8153 to 8156). Note that in this instance the escape code is returned in the Y register. If a key is detected in the allotted time period it is returned from OSBYTE in the X register and both the Y register and carry flag are clear. If no character is detected within the time period, Y returns containing &FF and the carry is set.

```
  10 REM *** DO MACHINE CODE INKEY ***
  20 PROCinkey (1000,&70,&C00)
  30 CLS
  40 PRINT"Press key within time limit
"
  50 CALL inkey
  60 PRINT"Key pressed was :";CHR$(?&70
)
  70 END
  80 :
8140 DEF PROCinkey (wait,result,addr)
```

```
8141 FOR pass=0 TO 2 STEP 2
8142 P%=addr
8143 [ OPT pass
8144 .inkey
8145          LDA #15
8146          LDX #1
8147          LDY #0
8148          JSR &FFF4
8149          LDA #&81
8150          LDX #wait MOD 256
8151          LDY #wait DIV 256
8152          JSR &FFF4
8153          CPY #&1B
8154          BNE no_escape
8155          LDA #&7E
8156          JSR &FFF4
8157 .no_escape
8158          STX result
8159          RTS
8160 ]
8161 NEXT
8162 ENDPROC
```

Program 13.8. PROCinkey - performs INKEY.

OSBYTE &81 can also be used to perform a single keyboard scan if it is called with the Y register holding &FF and the X register the negative INKEY value.

The demonstration program performs an INKEY(1000) equivalent, which basically causes the MOS to look at the keyboard for 10 seconds or until a key is pressed.

Both of the above two routines suffer from one drawback. They are slow! Well, in machine code terms they are. Consider that the fastest MOS-based routine, using OSBYTE &81 with X holding the negative inkey value will take at least 300 cycles and anything up to 1200 cycles. Program 13.9 does the whole scan in a mere 12 cycles and what's more it can test for two keys being pressed at once! The code assembled by PROCkeytest looks at locations &EC and &ED. If a key is pressed at any time then the MOS places zero into &ED and &EC contains the key's number. The actual numbers stored by the MOS are internal key numbers +128. For almost all purposes the internal key numbers are the negative INKEY numbers made positive and then decremented by one. For example, C is equal to INKEY(-83) so its internal number is calculated as ABS(-83)-1 or 83-1=82. Testing for C using 'keytest' therefore requires a test for 82+128=210.

If two keys are pressed almost simultaneously then &ED contains the number of the first key pressed and &EC the second key. If no keys are detected then both these bytes are zero.

```
  10 REM *** QUICK TEST FOR KEY ***
  20 PROCkeytest (&900)
  30 VDU15
  40 CALL start
  50 END
  60 :
8300 DEFPROCkeytest (addr)
8301 FOR pass=0 TO 2 STEP 2
8302 P%=addr
8303 [
8304 OPT pass
8305 .start
8306           LDA&ED
8307           BEQ check_EC
8308           JSR valid_key
8309           BEQ check_EC
8310           JSR&FFEE
8311 .check_EC
8312           LDA&EC
8313           BEQ start
8314           JSR valid_key
8315           BEQ start
8316           JSR&FFEE
8317           JMP start
8318 .valid_key
8319           CMP#&F0
8320           BNE next1
8321           PLA
8322           PLA
8323           LDA#15
8324           JSR&FFF4
8325           RTS
8326 .next1
8327           CMP#&E1
8328           BNE next2
8329           LDA#&5A
8330           RTS
8331 .next2
8332           CMP#&C2
8333           BNE next3
8334           LDA#&58
8335           RTS
8336 .next3
8337           CMP#&D2
8338           BNE next4
8339           LDA#&43
8340           RTS
8341 .next4
8342           CMP#&E3
8343           BNE next5
```

```
8344              LDA#&56
8345              RTS
8346 .next5
8347              LDA#0
8348              RTS
8349 ]
8350 NEXT
8351 ENDPROC
```

Program 13.9. PROCkeytest - a 12-cycle key test.

As it stands, the routine has been set up to look for Z, X, C, V and ESCAPE. The hex begins by peeking &ED. If this is zero then &EC can be tested straightaway so the branch to 'check_EC' is performed (lines 8306 to 8307). The subroutine 'valid_key' tests for each of the above keys. The first tested is ESCAPE, code &F0 (line 8319). If it is detected then the RTS address is pulled from the stack and the MOS entered with 15 in the accumulator to handle the ESCAPE. The following bytes then look for each key in turn. The codes for each are:

&E1 = Z
&C2 = X
&D2 = C
&E3 = V

If any of these compare, the letter is printed out. The 'check_EC' routine works exactly the same.

If you run the program and then press either the Z, X, C, or V keys then you'll see just how fast this key test routine is. The screen half fills with letters before you can lift your finger off the key! Finally, I should point out that this method is not condoned by Acorn as it is MOS-dependent. It will work on OS 1.0 and OS 1.2 but I haven't tried it on OS 0.1 so it may not work if you are using this version of MOS. The GET single key type routines could be employed to read in a string of characters, placing each one into a defined buffer until a set number of characters is reached or a return character detected. This does involve some extra coding, though, as a loop and buffer would need to be implemented. A neater way is to use the MOS line input routine OSWORD &00. This is a very useful call as it allows a number of parameters to be specified regarding the characters being input, Figure 13.1 details the parameter block. The first two bytes contain the address ofthe input buffer, the second byte the maximum number of characters to be placed in the buffer, while the last two bytes determine the maximum and minimum acceptable ASCII characters that will be accepted and

placed in the buffer!

This OSWORD call does have one major disadvantage, though. Although only characters in the specified ASCII range will be placed into the buffer, any other characters presses will be echoed to the screen even though they are not deposited in the buffer. Program 13.10 provides a suitable procedure. The parameter block address for the call is defined in 'block' while B% determines the input line buffer location; L% the number of characters acceptable, i.e. the buffer's maximum length; and 'max', 'min' the acceptable character range.

XY+0	: LSB of input buffer address
XY+1	: MSB of input buffer address
XY+2	: Maximum number of characters
XY+3	: Minimum ASCII value of character acceptable
XY+4	: Maximum ASCII value of character acceptable

Fig. 13.1. OSWORD &00 parameter block

```
  10 REM ** READ LINE FROM INPUT **
  20 PROCinputline(&70,&C00,10,ASC"A",A
SC"Z",&4000)
  30 CALL &4000
  40 PRINT$&C00
  50 END
8170 DEF PROCinputline(block,B%,L%,max,
min,addr)
8171 FOR pass=0 TO 3 STEP3
8172 P%=addr
8173 [ OPT pass
8174          LDA #B% MOD 256
8175          STA block
8176          LDA #B% DIV 256
8177          STA block+1
8178          LDA #L%
8179          STA block+2
8180          LDA #max
8181          STA block+3
8182          LDA #min
8183          STA block+4
8184          LDA #0
8185          LDX #block MOD 256
8186          LDY #block DIV 256
8187          JSR &FFF1
8188          RTS
8189 ]
```

```
8190 NEXT
8191 ENDPROC
```

Program 13.10. PROCinputline - input a line of text

Program 13.11 will read the bit map definition of any ASCII or defined
VDU character. OSWORD &9A performs the task and all that is
required prior to the read is to define a nine-byte parameter block, 'block'
in the program. The ASCII code of the character to be read should be
located in the first byte ofthe parameter block and on return the
following eight bytes contain the character's definition starting with the
top row of the character.

```
  10 REM **READ VDU CHR DEFINITION**
  20 CLS
  30 PRINT"PRESS A KEY TO DISPLAY ITS D
EFINITION";
  40 chr$=GET$
  50 chr=ASC(chr$)
  60 PROCread_chr(&70,chr,&4000)
  70 CALL &4000
  80 CLS
  90 PRINT"THE DEFINITION OF ";chr$;" I
S :"
 100 FOR N=&71 TO &78:PRINT?N:NEXT
 110 END
8200 DEF PROCread_chr(block,chr,addr%)
8201 FOR pass=0 TO 3 STEP 3
8202 P%=addr%
8203 [ OPT pass
8204          LDA #chr
8205          STA block
8206          LDA #10
8207          LDX #block MOD 256
8208          LDY #block DIV 256
8209          JSR &FFF1
8210          RTS
8211 ]
8212 NEXT
8213 ENDPROC
```

Program 13.11. PROCread_chr - reads the eight-byte definition of any
character.

A simple change

The final three programs provide some simple conversion routines.

Program 13.12 will convert the hexadecimal value at 'byte' and print it to the screen as two ASCII hex bytes. Thus if 'byte' held &FF then the letters F and F will be printed. The program works as follows. After loading the byte for conversion into the accumulator, the logical AND ensures that only the lowest four bits are set (lines 8233 to 8234). After setting the decimal flag, the addition of &90 to binary values 0 to 9 will result in a value of &90 to &99 with the carry flag clear. A further addition of &40 will convert these characters to values in the range &30 to &39 with the carry set. Remember that decimal addition is in operation so that adding 1 to &99 will give a result of &00 rather than &9A.

```
  10 REM *** SIMPLE HEX TO ASCII ***
  20 PROChex_asc(&70,&71,&72,&C00)
  30 ?&70=255
  40 CALL &C00
  50 PRINT''
  60 PRINTCHR$(?&71);CHR$(?&72)
  70 END
  80 :
8230 DEF PROChex_asc(byte,high,low,addr
)
8231 P%=addr
8232 [
8233           LDA byte
8234           AND #15
8235           SED
8236           CLC
8237           ADC #&90
8238           ADC #&40
8239           CLD
8240           STA low
8241           LDA byte
8242           LSR A
8243           LSR A
8244           LSR A
8245           LSR A
8246           SED
8247           CLC
8248           ADC #&90
8249           ADC #&40
8250           CLD
8251           STA high
8252           RTS
8253 ]
8254 ENDPROC
```

Program 13.12. PROChex_asc - convert a hexadecimal number into two ASCII values.

Adding &90 to the binary values &A to &F results in a byte in the range &00 to &05 with the carry set. A further addition of &40 (plus the set carry) converts this to values &41 to &46, the ASCIl codes for the letters A to F.

The byte is then saved and the next four bits treated likewise after shifting them into the lower nibble position with four logical shifts. Both ASCII codes are stored at 'high' and 'low' for future reference. If you wish not to save the two results but to print them directly then remember that the high nibble must be treated first and then the low nibble as the digits are printed, most significant first, on the screen.

Program 13.13 converts a packed BCD digit into its component ASCII codes. To perform the conversion the byte must first be transformed into its two component nibbles, which should be placed in the low nibble position before having bit 5 forced to 1 using ORA #&30. The high nibble is treated first. After loading the packed byte from 'bcd' it is pushed onto the hardware stack for future reference. The high nibble is then shifted into the low nibble position (lines 8263 to 8268). Bit 5 is then forced and the ASCII character code placed in 'high' - it could at this point be printed using JSR &FFEE instead. The stack is pulled and the high nibble masked out with an AND (lines 8271 to 8272). Bit 5 is forced and the result placed in 'low' (lines 8273 to 8274).

```
  10 REM *** PACKED BCD TO ASCII ***
  20 PROCbcd_ascii(&70,&71,&72,&C00)
  30 ?&70=&12
  40 CALL &C00
  45 PRINT''
  50 PRINT CHR$(?&72); CHR$(?&71)
  60 END
  70 :
8260 DEF PROCbcd_ascii(bcd,low,high,add
r)
8261 P%=addr
8262 [
8263           LDA bcd
8264           PHA
8265           LSR A
8266           LSR A
8267           LSR A
8268           LSR A
8269           ORA #&30
8270           STA high
8271           PLA
8272           AND #15
8273           ORA #&30
8274           STA low
```

```
8275            RTS
8276 ]
8277 ENDPROC
```

Program 13.13. PROCbcd_ascii - converts a packed BCD byte into two
ASCII values.

Performing the reverse conversion, two ASCII digits to packed BCD,
involves reversing the procedure as depicted in Program 13.14. The high
digit is extracted from 'high' shifted right so that bit 5 is lost and only the
binary bits remain. This byte is pushed onto the hardware stack. The
'low' digit is loaded into the accumulator, the low nibble preserved, and
the stack pointer copied into the X register (lines 8289 to 8291). The two
digits have now been stripped of bit 5, and all that is now required is to
merge them together. This is done by logically ORing the two together
(lines 8292). The X register is incremented and copied back into the
stack pointer thus 'popping' the byte from the stack. Finally the result is
placed at 'bcd'.

```
  10 REM *** ASCII TO PACKED BCD ***
  20 PROCasc_bcd(&70,&71,&72,&C00)
  30 ?&70=ASC("6")
  40 ?&71=ASC("9")
  50 CALL &C00
  60 PRINT~?&72
  70 END
  80 :
8280 DEF PROCasc_bcd(high,low,bcd,addr)
8281 P%=addr
8282 [
8283            LDA high
8284            ASL A
8285            ASL A
8286            ASL A
8287            ASL A
8288            PHA
8289            LDA low
8290            AND #15
8291            TSX
8292            ORA &101,X
8293            INX
8294            TXS
8295            STA bcd
8296            RTS
8297 ]
8298 ENDPROC
```

Program 13.14. PROCasc_bcd - converts two ACII values into a packed

BCD byte.

Program fact sheets

Program 13.1

Procedure title	: PROCcls
Variables required	: addr
Line numbers	: 8000 to 8008
Length	: 6 bytes
Zero page requirements	: none
Registers changed	*: A*

Program 13.2

Procedure title	: PROCcursor
Variables required	: pos, vpos, addr
Line numbers	: 8010 to 8020
Length	: 10 bytes
Zero page requirements	: 2 bytes
Registers changed	: A, X, Y

Program 13.3

Procedure title	: PROCspace
Variables required	: spc, addr
Line numbers	: 8030 to 8042
Length	: 11 bytes
Zero page requirements	: none
Registers changed	: A, X

Program 13.4

Procedure title	: PROCstring
Variables required	: buffer, num, addr
Line numbers	: 8050 to 8069
Length	: 19 bytes
Zero page requirements	: none
Registers changed	: A, X, Y

Program 13.5

Procedure title	: PROCtabx

Variables required   : xpos, addr
Line numbers   : 8080 to 8092
Length   : 17 bytes
Zero page requirements : none
Registers changed   : A, X

Program 13.6

Procedure title   : PROCtabxy
Variables required   : xpos, ypos, addr
Line numbers   : 8100 to 8112
Length   : 16 bytes
Zero page requirements : none
Registers changed   : A

Program 13.7

Procedure title   : PROCgetkey
Variables required   : chr, addr
Line numbers   : 8120 to 8136
Length   : 17 bytes
Zero page requirements : none
Registers changed   : A

Program 13.8

Procedure title   : PROCinkey
Variables required   : wait, result, addr
Line numbers   : 8150 to 8162
Length   : 31 bytes
Zero page requirements : 1 byte
Registers changed   : A, X, Y

Program 13.9

Procedure title   : PROCkeytest
Variables required   : addr
Line numbers   : 8300 to 8351
Length   : 69 bytes
Zero page requirements : none
Registers changed   : A

Program 13.10

Procedure title   : PROCinputline
Variables required   : block, B%, L%, max, min, addr
Line numbers   : 8170 to 8191
Length   : 26 bytes
Zero page requirements : 5 bytes
Registers changed   : A, X, Y

Program 13.11

Procedure title   : PROCread_chr
Variables required   : block, chr, addr
Line numbers   : 8200 to 8213
Length   : 14 bytes
Zero page requirements : 9 bytes
Registers changed   : A, X, Y

Program 13.12

Procedure title   : PROChex_asc
Variables required   : byte, high, low, addr
Line numbers   : 8230 to 8254
Length   : 29 bytes
Zero page requirements : 3 bytes
Registers changed   : A

Program 13.13

Procedure title   : PROCbcd_ascii
Variables required   : bcd, low, high, addr
Line numbers   : 8260 to 8277
Length   : 19 bytes
Zero page requirements : 3 bytes
Registers changed   : A

Program 13.14

Procedure title   : PROCasc_bcd
Variables required   : high, low, bcd, addr
Line numbers   : 8280 to 8289
Length   : 20 bytes
Zero page requirements : 3 bytes
Registers changed   : A, X

Chapter Fourteen
Odd One Out

This final chapter in the Portfolio draws together eight programs that offer a variety of functions. The programs are:

Program 14.1 : Find highest IRQ.
Program 14.2 : Timer 1 delay.
Program 14.3 : Timer 2 delay.
Program 14.4 : One second delay.
Program 14.5 : Save all processor registers.
Program 14.6 : Restore all processor registers.
Program 14.7 : Two-byte incrementing counter.
Program 14.8 : Two-byte decrementing counter.

Interrupt polling

Program 14.1 is not a complete routine as it stands as it expects extra code, written by the user, to be tagged onto it. Basically it is an interrupt polling sequence for the User VIA, capable of identifying the highest priority interrupt request on any one ofthe seven lines capable of having an IRQ.

In order to know which interrupt servicing routine to call, the source of the IRQ must be determined. To find this out, bit 7 ofthe Interrupt Flag Register (IFR7) must be tested. If this bit is set then an IRQ has been issued. The IFR is read using an OSBYTE &96 call to read Sheila. As the IFR occupies location &6D in Sheila this value is placed in the X register. After the call, the Y register contains the byte just read, which is transferred into the accumulator. If bit 7 is clear the branch to 'next_device'will be executed (lines 9006 to 9010). What is required now is to read the Interrupt Enable Register (IER) at Sheila &6E, as a set bit in this register will give the identity of the IRQ. This is performed by lines 9012 to 9015. For an interrupt to have occurred, the corresponding bits in the IFR and IER must have been set; to determine the actual bit these two bytes are logically ANDed together to preserve the set bit (line

9015 to 9016). Now all that is required is to shift the byte to condition the negative flag. Setting the negative flag will determine that the bit just shifted was the lineassociated bit that caused the IRQ and thus the BMI for that test will proceed.

```
  10 REM *** FIND HIGHEST IRQ ***
  20 REM ***needs extra user coding***
  30 REM **to run sequence correctly**
  40 :
9000 DEF PROChighestIRQ (temp,addr)
9001 FOR pass=0 TO 3 STEP 3
9002 P%=addr
9003 [
9004 OPT pass
9005 .findIRQ
9006          LDA #&96
9007          LDX #&6D
9008          JSR &FFF4
9009          TYA
9010          BPL next_device
9011          PHA
9012          LDA #&96
9013          LDX #&6E
9014          JSR &FFF4
9015          STY temp
9016          PLA
9017          AND temp
9018          ASL A
9019          BMI timer1
9020          ASL A
9021          BMI timer2
9022          ASL A
9023          BMI cb1
9024          ASL A
9025          BMI cb2
9026          ASL A
9027          BMI shift_reg
9028          ASL A
9029          BMI ca1
9030          ASL A
9031          BMI ca2
9032          JMP error
9033 .timer1
9034          JMP T1service
9035 .timer2
9036          JMP T2service
9037 .cb1
9038          JMP cb1service
9039 .cb2
9040          JMP cb2service
9041 .shift_reg
```

```
9042            JMP srservice
9043 .ca1
9044            JMP ca1service
9045 .ca2
9046            JMP ca2service
9047 .error
9048            \ error service routine here
9049 .next_device
9050            \ more polling here
9051 ]
9052 NEXT
9053 ENDPROC
```

Program 14.1. PROChighestIRQ - locates the highest priority interrupt.

In Program 14.1 I have used a left to right priority system, so that bit 6 has a greater priority than bit 5. Therefore an interrupt on Tl interrupt enable has a greater priority than an interrupt on T2 and so forth. You can arrange your own priorities to suit, though the test procedure might not be a simple shift and branch sequence, and more complex coding might be required.

A timed delay

Both the timers in the User VIA can be used to produce delays. Program 14.2 uses Timer 1 to provide one-millisecond delay using it in its one-shot mode of operation. To place T1 into this mode, zero must be written to the Auxillary Control Register at Sheila &6B. Next the delay must be loaded into the latches. One millisecond corresponds to 1000 cycles, however, and it should be remembered that T1 has an operating overhead of 1.5 cycles, so the actual value loaded into the latches must be the actual value minus 2. Thus, 998 is to be deposited into the T1 latches. Lines 9070 to 9075 perform this using the hex equivalent &3E8; writing to the msb latch starts the timer running. On timing out, the Tl bit in the IFR will be set. To test for this, &40, is loaded into the accumulator and the IFR tested using the BIT operation. This small loop (lines 9077 to 9079) continues until the BEQ fails, denoting time out. The T1 flag is then cleared by reading the latch (line 9080).

```
  10 REM *** MILLISECOND DELAY USING T1
***
  20 PROCtimerone_delay (&C00)
  30 CALL millisec1
  40 END
```

```
  50 :
9060 DEF PROCtimerone_delay (addr)
9061 FOR pass=0 TO 3 STEP 3
9062 P%=addr
9063 [
9064 OPT pass
9065 .millisec1
9066            LDA #&97
9067            LDX #&6B
9068            LDY #0
9069            JSR &FFF4
9070            LDX #&64
9071            LDY #&E8
9072            JSR &FFF4
9073            LDX #&65
9074            LDY #3
9075            JSR &FFF4
9076            LDA #&40
9077 .loop
9078            BIT &FE6D
9079            BEQ loop
9080            LDA &FE64
9081            RTS
9082 ]
9083 NEXT
9084 ENDPROC
```

Program 14.2. PROCtimerone_delay - a one millisecond delay using Timer 1.

Program 14.3 performs a similar delay using Timer 2. Only the addresses in the program and the bit mask change.

```
  10 REM *** MILLISECOND DELAY USING T2
***
  20 PROCtimertwo_delay (&C00)
  30 CALL millisec
  40 END
  50 :
9100 DEF PROCtimertwo_delay (addr)
9101 FOR pass=0 TO 3 STEP 3
9102 P%=addr
9103 [
9104 OPT pass
9105 .millisec
9106            LDA #&97
9107            LDX #&6B
9108            LDY #0
9109            JSR &FFF4
9110            LDX #&68
9111            LDY #&E8
9112            JSR &FFF4
```

```
9113             LDX #&69
9114             LDY #3
9115             JSR &FFF4
9116             LDA #&20
9117 .loop
9118             BIT &FE6D
9119             BEQ loop
9120             LDA &FE6B
9121             RTS
9122 ]
9123 NEXT
9124 ENDPROC
```

Program 14.3. PROCtimertwo_delay - a one millisecond delay using Timer 2.

The timers are fine for providing very short delay loops but for substantial delays they are not really suitable. Program 14.4 will provide a 1-second delay. It does this by just executing a series of timed loops. As the clock on the Beeb operates at 2MHz, the delay loop need only count out 2000000 cycles to create the delay.


```
  10 REM *** 1.0 SECOND DELAY ***
  20 PROCdelay (&C00)
  30 INPUT"How many seconds delay ?: "w
ait
  40 TIME=0
  50 FOR loop=1 TO wait
  60 CALL &C00
  70 NEXT
  80 time%=TIME
  90 time%=(time%/100)
 100 PRINT"Time taken was :";time%;
 110 PRINT" second(s)"
 120 END
 130 :
9130 DEF PROCdelay (addr)
9131 FOR PASS=0 TO 3 STEP 3
9132 P%=addr
9133 [ OPT PASS
9134             PHP
9135             PHA
9136             TXA
9137             PHA
9138             TYA
9139             PHA
9140             LDY outer
9141 .loop1
9142             TYA
```

```
9143            PHA
9144            LDX inner
9145 .loop2
9146            LDY #5
9147 .loop3
9148            DEY
9149            BNE loop3
9150            DEX
9151            BNE loop2
9152            LDY #2
9153 .loop4
9154            DEY
9155            BNE loop4
9156            PLA
9157            TAY
9158            DEY
9159            BNE loop1
9160            LDY fine
9161 .loop5
9162            DEY
9163            BNE loop5
9164            PLA
9165            TAY
9166            PLA
9167            TAX
9168            PLA
9169            PLP
9170            RTS
9171 .outer
9172            EQUB 251
9173 .inner
9174            EQUB 0
9175 .fine
9176            EQUB 197
9177 ]
9178 NEXT
9179 ENDPROC
```

Program 14.4. PROCdelay - a one second delay loop.

The main loop counter is provided by 'outer' while the finer inner loop counter is 'inner'. The program commences by pushing all processor registers onto the stack thus preserving their status on return. This process takes 16 cycles (lines 9134 to 9139). The Y register is then loaded with 'outer' (4 cycles line 9140) and 'loopl' entered. This major outer loop has a smaller loop within it between lines 9145 and 9151 which takes a total of 256*31-1 cycles or 7935 cycles to execute. As the outer 'loop1' is controlled by the contents of the Y register, 251, the main loop between lines 9141 and 9159 takes 251*7964-1 or 1998963 cycles

to run. In both cases, the -1 is needed because the final branch does not take place and therefore only uses 2 cycles and not the 3 allowed in the calculation. The final 'fine' loop plus restoring the registers add a further 30+984 (the loop in lines 9162 to 9163) cycles to the overall delay. The total delay provided by the loop is therefore calculated as 1998963 + 16 + 4 + 30 + 984 = 1999997 cycles. This is obviously three cycles short of the desired delay, which doesn't really bear thinking about!

The BASIC demo asks how long a delay you would like. Because there is an overhead in the BASIC operations, don't be too surprised if you enter 20 in response to the prompt and get an answer of 23 seconds back. The extra three seconds were created by the BASIC interpreter working through the program! Now a question. Add the following line to the program -

    25 CLS

and run it. Why does it seem to work twice as quick? Don't expect the answer; I'm still trying to fathom it out for myself!

A push me pull you

The last program showed that a subroutine call need not destroy the contents of the processor registers if their contents are important. With the exception of two programs in this book all the assembler procedures alter the contents of at least the accumulator and many of the index registers as well. Programs 14.5 and 14.6 provide suitable procedures to save and then restore all processor registers. The main point to note here is that the assembler is not implemented as a subroutine; in other words it does not end with an RTS. This means that the code must be placed at the point it is needed, with the register changing hex following straight after. It would be possible to implement it as a subroutine if required though1would not recommend it as it would need some jiggery pokery with the stack to put the RTS address after the pushed register values.

```
  10 REM*SAVE ALL PROCESSOR REGISTERS*
  20 REM*DOES NOT HAVE RTS MUST BE *
  30 REM*USED DIRECTLY IN CODE. *
  40 :
9200 DEF PROCpush_all(addr)
9201 P%=addr
9202 [ OPT 2
9203 .pushall
```

```
9204            PHP
9205            PHA
9206            TXA
9207            PHA
9208            TYA
9209            PHA
9210 ]
9211 ENDPROC
```

Program 14.5. PROCpush_all - save all processor registers on hardware stack.

```
  10 REM*RESTORE ALL PROCESSOR REGISTER
S*
  20 REM*DOES NOT HAVE RTS MUST BE *
  30 REM*USED DIRECTLY IN CODE. *
  40 :
9220 DEF PROCpull_all(addr)
9221 P%=addr
9222 [ OPT 2
9223 .pullall
9224 PLA
9225 TAY
9226 PLA
9227 TAX
9228 PLA
9229 PLP
9230 ]
9231 ENDPROC
```

Program 14.6. PROCpull_all - restore all processor registers off of the hardware stack.

Count on it

And so to the last two of the 75 programs in this book, which1do hope you have found informative and useful. The two programs implement double-byte counters, useful for loop control with a count greater than 255 or for updating a two-byte memory address.

Program 14.7 is an incrementing counter. The procedure first loads the start value of the counter, 'num' into the two bytes from 'block' (lines 9244 to 9248). The incrementing starts at 'plus_one' (line 9249) where 'block' has one added to it (line 9250). Any carry is detected by the zero flag and if set, one is added to 'block+1' else a return via 'no_inc' is performed.

```
   10 REM *** TWO BYTE COUNTER ***
   20 PROCinc_count (5000,&70,&C00)
   25 CALL seed_value
   30 REPEAT
   40 PRINT?&71*256+?&70
   50 CALL plus_one
   60 UNTIL ?&71=0
   70 END
   80 :
 9240 DEF PROCinc_count (num,block,addr)
 9241 FOR pass=0 TO 3 STEP 3
 9242 P%=addr
 9243 [ OPT pass
 9244 .seed_value
 9245           LDA #num MOD 256
 9246           STA block
 9247           LDA #num DIV 256
 9248           STA block+1
 9249 .plus_one
 9250           INC block
 9251           BNE no_inc
 9252           INC block+1
 9253 .no_inc
 9254           RTS
 9255 ]
 9256 NEXT
 9257 ENDPROC
```

Program 14.7. PROCinc_count - performs a double-byte increment.

```
   10 REM *** DOUBLE BYTE DECREMENT ***
   20 PROCdec_count (5000,&70,&C00)
   25 CALL seed_value
   30 REPEAT
   40 PRINT?&71*256+?&70
   50 CALL minus_one
   60 UNTIL ?&71=0
   70 END
   80 :
 9260 DEF PROCdec_count (num,block,addr)
 9261 FOR pass=0 TO 3 STEP 3
 9262 P%=addr
 9263 [ OPT pass
 9264 .seed_value
 9265           LDA #num MOD256
 9266           STA block
 9267           LDA #num DIV256
 9268           STA block+1
 9269 .minus_one
 9270           LDA block
```

```
9271              BNE no_dec
9272              DEC block+1
9273 .no_dec
9274              DEC block
9275              RTS
9276 ]
9277 NEXT
9278 ENDPROC
```

Program 14.8. PROCdec_count - performs a double-byte decrement.

The application program in lines 20 to 60 shows that once a count value has been seeded only 'plus_one' should be called to increment the block.

Decrcmenting a two-byte counter is a little less straightfbrward (Program 14.8). As before, the count start value is first secded through 'num' (lines 9264 to 9268). The decrement process begins at 'minus_one' by first loading the low byte of the count at 'block' into thc accumulator (line 9270). If this byte should be zero then thc decrement must take the high page byte into consideration and decrement this (line 9272). Finally, the low byte is decremented (line 9274).

As with the previous program, after the initial set up it is 'no_dcc' that must be called to perform the decrement as the BASIC demo illustrates.

Program fact sheets

Program 14.1

Procedure title	: PROChighestIRQ
Variables required	: temp, addr
Line numbers	: 9000 to >9050
Length	: variable
Zero page requirements	: none
Registers changed	: A, X, Y

Program 14.2

Procedure title	: PROCtimerone_delav
Variables required	: addr
Line numbers	: 9060 to 9084
Length	: 34 bytes
Zero page requirements	: none
Registers changed	: A, X, Y

Program 14.3

Procedure title	: PROCtimertwo_delay
Variables required	: addr
Line numbers	: 9100 to 9124
Length	: 34 bytes
Zero page requirements	: none
Registers changed	: A, X, Y

Program 14.4

Procedure title	: PROCdelay
Variables required	: addr
Line numbers	: 9130 to 9179
Length	: 48 bytes
Zero page requirements	: none
Registers changed	: none

Program 14.5

Procedure title	: PROCpush_all
Variables required	: addr
Line numbers	: 9200 to 9211
Length	: 6 bytes
Zero page requirements	: none
Registers changed	: none

Program 14.6

Procedure title	: PROCpull_all
Variables required	: addr
Line numbers	: 9220 to 9231
Length	: 6 bytes
Zero page requirements	: none
Registcrs changed	: A, X, Y
Odd One Out193	

Program 14.7

Procedure title	: PROCinc_count
Variables required	: num, block, addr
Line numbers	: 9240 to 9257
Length	: 15 bytes

        Zero page requirements : 2 bytes
        Registers changed   : A

Program 14.8

        Procedure title   : PROCdec_count
        Variables required   : num, block, addr
        Line numbers   : 9260 to 9278
        Length   : 17 bytes
        Zero page requirements : 2 bytes
        Registers changed   : A

Index