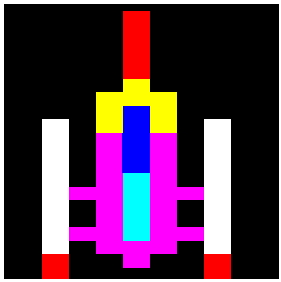
SECTION 3



12

UTILITIES FOR ASSEMBLER PROGRAMS

This chapter consists of a number of routines which are designed to be used in any assembler program. Typical calls are given for each, as are the values to be passed to the routines in the registers, and those values to be returned by them.

Note that these routines are not meant to be complete programs and cannot be run without additional code being added, e.g. assignment of values to any addresses being used and all the necessary assembler directives. If you wish to use any of these routines inside your own programs then the comments to the right of the assembler statements may be omitted. If the routines contain any BASIC commands then any comments to the right of these must be left out or preceded by REM statements.

12.1 Input/Output
Print BCD number -- printnumber
The 6502 microprocessor can perform arithmetic in two ways. These are binary (normal addition), and binary coded decimal (BCD). In the second method, each byte is split up into two nibbles, each of which can hold a decimal digit (0 to 9). Thus the largest number that can be held in one byte using this method is 99. The advantage of this method is that it

is easier to output a binary coded decimal number in decimal than it is to output a straight binary number in decimal.

The following routine takes a BCD number in the accumulator and prints it, with leading zero suppression, at the current cursor position.

On entry, A contains the number to be pnnted, Y contains the leading zero flag (0 for no suppression, else suppress zeros), and X contains the ASCII code of the character to be printed in place of leading zeros.

A typical call to print out a two-byte BCD number, with leading zeros being replaced by spaces, would be:

```
LDX #ASC" "      Set Leading zero character to space
LDY #&FF         Set Leading zero fLag
LDA highbyte     Get top byte of number
JSR printnumber Print it
LDA lowbyte      Get bottom byte of number
JSR printnumber Print it
     .
     .
```

On exit X has been preserved and A and Y will have been corrupted.

```
.print number
   PHA                  Save number
   OPT FNrotateacc (4) Get top digit (See section 8.2)
   JSR printit          Print it
   PLA                  Restore number
   AND #&0F             Get bottom digit
.printit               Fall through
   BNE validchar        If non-zero print
   TYA                  Check zero flag
   BNE leadingzero      If set must be leading zero
.validchar             Digit ok
   LDY #&00             Clear zero flag
   ORA #ASC"0"          Add in ASCII zero
   JMP oswrch           Print, and return
.Leadingzero
   TXA                  Get leading zero character
   JMP oswrch           Print and return
```

Keyboard scan -- inkey
This routine can be called to detect if a key is being held down at a particular instant; it uses INKEY of negative numbers.

On entry, X specifies the key to be tested. For details of the values for each key, see the table of INKEY negative numbers in Appendix A.

A typical call would be:

```
LDX #£firstkey
JSR inkey
BNE keypressed
```

On exit, the zero flag is set or cleared depending upon the key's position at the time of testing. This routine does NOT go via the keyboard buffer. A and Y will have been preserved.

```
.inkey
   PHA          Save A
   TYA          Save Y
   PHA
   LDY #&FF     Negative numbers
   LDA #&81     osbyte &81 is INKEY
   JSR osbyte   Do it
   PLA          Restore Y
   TAY
   PLA          Restore A
   CPX #&00     Adjust zero flag
   RTS          and return
```

Sound
Sound routines are always useful in games programs. The routine below works by using a table to hold all the sounds that it is to play, and, on entry, the number of the sound to be played is given. To set up the table, the following could be used:

```
FOR offset = 0 TO <number of sounds> * 8 STEP 2
READ sndbuff!offset
NEXT offset
DATA 1,-15,200,20
DATA 3,1,150,10
DATA etc.
```

Where 'sndbuff' is the table to be filled with sounds, on entry, A holds the number of the sound to be played. X and Y are irrelevant.

A typical call would be:

```
LDA #£firingsound
JSR sound
```

on exit, all registers will have been corrupted.

```
.sound
  ASL A               Multiply sound number by 8
  ASL A
  ASL A
  ADC #FNlo(sndbuff) add in address of sound table
  TAX                 Put low byte of addres in X
  LDY #Fnhi(sndbuff) Get hi byte in Y
  BCC nohibyte        If carry then X and Y correct
  INY                 Else increment hi byte
.nohibyte
  LDA #&07            Osword 7 is SOUND
  JMP osword          Do it and return
```

Print strings
ATOM style string -- atomstring
ATOM strings are defined as being groups of characters terminated by a RETURN character (&0D).

On entry, X and Y hold the start address in memory of the string to be printed (X holds low byte, and Y holds high byte). A is irrelevant.

A typical call would be:

```
LDX #FNlo(hiscorestring)
LDY #FNhi(hiscorestring)
JSR atomstring
```

On exit, all registers will have been corrupted.

```
.atomstring
  STX stringptr     Address of string to be printed
  STY stringptr + 1 is given in X and Y
  LDY #&00          Y is pointer along the string
.atomstringloop
  LDA (stringptr),Y Get next character from string
```

```
      JSR oswrch          Print it
      INY                 Increment pointer
      CMP #&0D            Is character return
      BNE atomstringloop  No? go back to start of loop
      RTS return
```

Microsoft style strings -- microsoftstring

Microsoft strings are defined as being groups of characters, preceded by a byte giving the length of the string. A microsoft string can be set up as follows:

```
.fredstring
  EQUB LEN(fred$)
  EQUS fred$
```

On entry, X and Y hold the start address of the string (X holds low byte, Y high byte). A is irrelevant.

A typical call might be:

```
LDX #FNlo(fredstring)
LDY #FNhi(fredstring)
JSR microsoftstring
```

On exit, all registers will have been corrupted.

```
.microsoftstring
   STX stringptr      Address of string to be printed
   STY stringptr + 1  is given in X and Y
   LDY #&00           Set pointer to length byte
   LDA (stringptr),Y  Get length of string
   STA len            Save length
.stringloop
   INY                Loop counter
   LDA (stringptr),Y  Get next character from string
   JSR oswrch         Print it
   CPY len            Printed all chars?
   BNE stringloop     No, go back to start of loop
   RTS                Else return
```

Centre a string -- centre

This routine will centre up microsoft strings on the current line of the cursor. The reason that only microsoft strings can be centred using this routine is that their length is far more accessible than the length

of an atom string, although the routine could be adapted for atom strings.

The BASIC equivalent of the routine given below is:

```
PRINT SPC((screenwidth - LEN(A$)) DIV 2);A$
```

On entry, X and Y point to the string to be centred, and A is irrelevant.

A typical call would be:

```
LDX #FNho(string)       Point to string
LDY UFNhi(string)
JSR centre              Centre it
```

On exit, all registers will have been corrupted.

```
.centre
  STX stringptr     Save Low byte of start address
  STY stringptr + 1 Save high byte of start address
  LDY #&00          Prepare to get Length byte
  LDA #screenwidth  Get screen width
  SEC
  SBC (stringptr),Y Subtract length
  LSR A             Divide by 2
  TAX               And transfer it to X
  LDA #ASC" "       Stand by to print X spaces
.centre loop
  JSR oswrch        Print a space
  DEX               Decrement counter
  BNE centreloop    and loop until counter is zero
  LDX stringptr     Restore string pointers
  LDY stringptr+1
  JMP microsoftstring And print the string
```

Move cursor to X, Y -- printtab
This is a very simple routine to move the text cursor to X, Y. It simulates BASIC' s ' PRINT TAB (X, Y)' .

On entry, X and Y hold the X and Y coordinates of the position that the text cursor is to be moved to, A is irrelevant.

A typical call might be:

```
LDX #xcoord
LDY #ycoord
JSR printtab
```

On exit, X and Y will be preserved, and A will have been corrupted.

```
.printtab
  LDA #31         VDU 31 (move textcursor to X,Y)
  JSR oswrch
  TXA             Send X coordinate
  JSR oswrch
  TYA             Send coordinate
  JMP oswrch      Do it and return
```

Double height characters - double

This next routine will only work in Teletext mode, and is thus only suitable for the BBC microcomputer.

The BASIC equivalent of this routine is:

```
DEF PROCdouble(A$)
vpos% = VPOS : pos% = POS
FOR string% = 0 TO 1
PRINT TAB(pos%, vpos% + string%);CHR$&8D; A$;
                                          CHR$&8C;
NEXT string%
ENDPROC
```

On entry, X and Y point to the string that is to be printed in double height, A is irrelevant.

A typical call would be:

```
LDX #FNlo(string)
LDY# FNhi(string)
JSR double
```

On exit, all registers will have been corrupted.

```
.double
   STX stringptr       Save start address of string
   STY stringptr +1
   LDA #&86            Read text cursor position
   JSR osbyte
   STX pos             and store it
   STY vpos
   LDX #&02            Print string twice
```

```
  STX count
.double loop
  LDX pos              Move cursor to X,Y
  LDY vpos
  JSR printtab
  LDA #&8D             Teletext code for Double height
  JSR oswrch
  LDX stringptr        Restore string start address
  LDY stringptr + 1    Either 'microsoft' or 'atom'
  JSR string           To centre string JSR centre
  LDA #&8C             Teletext Normal height code
  JSR oswrch
  INC vpos             Move down a line
  DEC count            Done it twice yet
  BNE doubleloop       If not then do it again
  RTS                  Else return
```

Palette handling - ospalette
This routine performs VDU 19, Y, A, 0, 0, 0.

On entry, Y contains the logical colour to be defined, A contains the physical colour to change Y to. (See section 9.4 for details of palette handling.) X is irrelevant.

A typical call might be:

```
LDY #logicalcolour
LDA #physicalcolour
JSR ospalette
```

On exit, Y and X have been preserved, A has been set to zero.

```
.ospalette
   PHA                  Save physical colour
   LDA #19              VDU 19
   JSR oswrch
   TYA                  Get logical colour
   JSR oswrch           VDU it
   PLA                  Get physical colour
   JSR oswrch           VDU it
   LDA #&00             Pad out with zeroes
   JSR oswrch
   JSR oswrch
   JMP oswrch           and return
```

Another routine for those of you with Electrons or BBC micros with 1.0 or 1.2 Operating Systems is to change the palette with OSWORD 12. This has the advantage of being able to be called from an interrupt routine.

```
.ospalette
   STY paletteblock          Same format as VDU 19
   STA paletteblock
   LDX #FNIo(paletteblock) Note that all registers
   LDY #FNhi(paletteblock) are corrupted
   LDA #12
   JMP osword
```

This is called in the same manner as before. Note that the ' paletteblock'must contain zeros in the last three locations before the routine is called.

Wait for flyback - vsync

The next routine is a must for animation. It will wait for the electron beam inside the VDU (Visual Display Unit) to reach the top of the screen in BBC Microcomputers or the bottom of the screen in Acorn Electrons, and will then return. This is when all shapes should be updated to avoid flickering.

On entry, all registers are irrelevant.

A typical call would be:

```
JSR vsync
```

On exit, all registers will have been corrupted.

```
.vsync
  LDA #19              Osbyte 19 is wait for vsync
  JMP osbyte           (vertical sync roni sat ion)
```

or

```
.vsync        This routine is only for Issue 0.10
LDA #&02      Operating Systems on the BBC micro
STA viaier    This is at &FE4E
.vloop
BIT viaifr    viaifr stands for Versatile
BEQ vloop   Interface Adapter Interrupt Flag
LDA #&82      Register, which is at &FE4D
STA viaier
RTS
```

12.2 Analogue to digital routines
Analogue to digital value -- adval
This routine reads any A to D channel.

On entry, X holds channel to be read. Y and A are irrelevant.

A typical call might be:

```
LDX #1                  Get value of channel 1
JSR adval
```

On exit, the value is returned in Y and X (Low byte and high byte respectively).

```
.adval
LDY #0                  Get ADVAL (X)
LDA #&80
JMP osbyte
```

Joystick handler --joystick
This routine reads either of the two joysticks connected to the A to D converter. Note that the sensitivity of the reading is dependant upon the value of the constant ' joyrange'(0 - insensitive to 127 - very sensitive). The variables ' xcoord'and ' ycoord' are user variables.

On entry, X holds the number of the joystick to be read (1 or 3). A and Y are irrelevant.

A typical call might be:

```
LDX #1                  Get readings of first joystick
JSR joystick
```

On exit, all registers will have been corrupted.

```
.joystick
   STX temp             Preserve joystick number
   JSR adval            Get horizontal reading
   LDX temp             Restore joystick number
   CPY #joyrange        Is reading within Limit ?
   BCS tryleft          See if within other limit
.right
   INC xcoord           Go right
.tryleft
```

```
   CPY #256-£joyrange  Is reading within limit ?
   BCC getotherpot     No, try vertical component
.left
   DEC xcoord          Go left
.getotherpot           Get vertical component
   INX                 Get adval (joystick + 1)
   STX temp            Preserve as before
   JSR adval           Get reading
   LDX temp            Restore joystick number
   CPY #joystick       All this is as above
.down                  except that y coordinate
   DEC ycoord          is being adjusted
.tryup
   CPY #256- joyrange
   BCC tryfire
.up
   INC ycoord
.tryfire               Get fire button
   TXA                 Halve X (for fire button
   LSR A               mask)
   STA temp            (ADVAL (0) AND X DIV 2)
   LDX #0              Get ADVAL (0)
   JSR adval
   TXA                 X holds fire button status
   AND temp            AND with mask
   BEQ exit            Not held down, then exit
.fire
   JSR firebullet      Else do something exit
.exit
   RTS                 Return
```

Oscilloscope

The program displays the four A to D channels as four different colours (colours 1 to 4). This program will only work on the BBC Microcomputer Model B, as it uses MODE 2, the Analogue to Digital converter, and Hardware scroll. The program also only reads the top eight bits of each channel, as the graphics vertical resolution is only 256. Sampling of the channels takes places every 1/50 of a second.

Some of the ideas in this program can be adapted to other programs. Note the modular construction, with each module as small as possible, so that it could be debugged easily during development.

```
 10 REM Oscilloscope V1
 20 DIM code 512              Set aside area for code
 30 PROCassemble              Assemble code
 40 MODE 2                    Set up screen mode
 50 CALL oscilloscope         Call machine code
 60 END
 70
 80 DEF PROCassemble
 90 osbyte = &FFF4            Set up variables
100 oswrch = &FFEE           constants
110 top = &80                Zero page allocation
120 screen = top + 2
130 temp = screen + 2
140 DIM colour 3,sidebuffer 255    Define vectors
150 !colour = &030C0F30            Fill colour vector
160 FOR pass = 0 TO 2 STEP 2
170 P% = code
180[OPT pass
190.osciIloscope             Entry point
200   JSR clearsidebuffer
210   JSR readchannels
220   JSR vsync
230   JSR scrollscreen
240   JSR writeside
250   JMP oscilloscope
260
270.clearsidebuffer
280   LDA #0                 Set buffer to 0
290   TAY                    Buffer is a page long
300.clearloop
310   STA sidebuffer,Y
320   DEY
330   BNE clearloop
340   RTS
350
360.readchannels
370   LDX #4                 Get four chanels
380.loop JSR adval
390   LDA colour-1, X        Get channel coour
400   STA sidebuffer, Y      Put reading in buffer
410   DEX                    Get next channel
420   BNE Loop
430   RTS
440
450. vsync
```

```
460   LDA # 19              See section 12.1 for
470   JMP osbyte            OS 0.10 vsync routine
480
490.scrollscreen
500   LDA top               Top = 16 bit address
510   STA temp
520   LDA top+1             Store top DIV 8 in the
530   LSR A                 6845 CRTC chip. See
540   ROR temp              section 9.3 for
550   LSR A                 details of screen
560   ROR temp              scrolling
570   LSR A
580   ROR temp
590   LDX #12               Register 12 & 13
600   JSR os6845            hold start addres of
610   LDX # 13              memory to be displayed
620   LDA temp
630   JSR os6845
640 # Now increment top
650   CLC                   Adjust variable that
660   LDA top               holds start address
670   ADC #8
680   STA top
690   LDA top + 1
700   ADC #0
710   BPL validaddress
720   SEC                   Allow for wraparound at
730   SBC #&50              &3000 / &8000 barrier
740.validaddress
750   STA top+1
760   RTS
770
780.writeside               Dump buffer to screen
790   LDX #&FF              Start at top of buffer
800   LDY #0                Set offset to zero
810   LDA top               Add &270 to top
820   CLC                   to right-hand side of
830   ADC #&70              the screen
840   STA screen
850   LDA top+1
860   ADC #2
870   BPL validaddress2
880   SEC                   Again allow for
90    SBC #&50              wraparound
900.validaddress2
```

```
 910   STA screen+1
 920.outerloop
 930   LDA sidebuffer,X       Get next byte from buffer
 940   STA (screen),Y         Store to screen
 950   DEX                    Adjust buffer pointer
 960   INY                    Every 8 bytes down,
 970   CPY # 8                the program must add in
 980   BNE outerloop          &280 to the address ,in
 990   LDY #0                 order to move to the next
1000   LDA screen            line
1010   CLC
1020   ADC #&80
1030   STA screen
1040   LDA screen + 1
1050   ADC # 2
1060   BPL validaddress3
1070   SEC
1080   SBC #&50
1090.validaddress3
1100   STA screen + 1
1110   CPX #&FF              Buffer all transferred to
1120   BNE outerloop         screen ?
1130   RTS
1140
1150.adval
1160   TXA                   Preserve channel pointer
1170   PHA
1180   LDY # 0               Get adval(X)
1190   LDA #&80
1200   JSR osbyte
1210   PLA                   Restore channel pointer
1220   TAX
1230   RTS
1240
1250.os6845
1260   PHA                   Perform
1270   LDA #23               VDU23; X, A, 0; 0; 0;
1280   JSR oswrch            To put A into 6845
1290   LDA #0                register X
1300   JSR oswrch
1310   TXA
1320   JSR oswrch
1330   PLA
1340   JSR oswrch
1350   LDX #6
```

```
1360   LDA #0
1370.pad
1380   JSR oswrch
1390   DEX
1400   BNE pad
1410   RTS
1420]
143ONEXT pass
1440 ENDPROC
```

12.3 Numerical routines

MOD -- mod

This routine can be useful when rounding numbers down, or when the remainder of a value is wanted, and the AND instruction cannot be used (e.g. value MOD 3). Note that this method is not one to be recommended for anything other than single byte operations, as the method of repeated subtraction would then be too slow. Also, there is no error checking procedure, so setting Y to 0 would cause the routine to loop forever.

On entry, A holds dividend, and Y holds the divisor. X is irrelevant.

A typical call might be:

```
LDA #dividend
LDY #divisor
JSR mod
```

On exit, A holds the remainder, X and Y are preserved.

```
.mod              Performs A = A MOD Y
   STY temp       Store divisor in temporary location
   SEC Set carry ( for subtraction )
.modloop
   SBC temp      Repeatedly subtract Y from A
   BCS modloop   until A becomes less than zero
   ADC temp      Add divisor (Note carry clear)
   RTS           and return
```

Random number generator -- rnd

A routine which is always useful for games programs is a random-number generator. The following routine generates a pseudo-random number in A.

The seed for the random number is three bytes long. This seed can be initialised before using the

routine, in order to generate a fixed sequence of numbers. Note that the seed should never contain zero in all three bytes as then the routine will continually give zero.

On entry, all registers are irrelevant.

A typical call would be:

```
JSR rnd
```

On exit, A holds the next pseudo-random number. X and Y are preserved.

```
.rnd
   LDA seed              Get low byte of shift register
   AND #&48
   ADC #&38
   ASL A
   ASL A
   ROL seed + 2
   ROL seed + 1
   ROL seed
   LDA seed
   RTS
```

12.4 Miscellaneous

Cyclic redundancy check - CRC

Cyclic redundancy checks can be useful for error detection when comparing blocks of data.

Using the program below you can give any block of memory a ' unique'two-byte signature. Thus you can check that two copies of a program are identical, by seeing if they have the same signature. This method is very secure, as it is very unlikely that two different blocks of memory would give the same signature.

```
  0 REM CRC calculator
 10
 20 signature = &70
 30 addr = signature + 2
 40 endaddr = addr + 2
 50 DIM code 200
 60 FOR pass = 0 TO 2 STEP 2
 70 P%=code
```

```
 80 [OPTpass
 90.crc
100   LDA #0            Iinitialise signature
110   STA signature
120   STA signature+1
130.mainloop
140   JSR crcbyte       Get crc for each byte
150   INC addr          16 bit increment
160   BNE nohibyte
170   INC addr+1
180.nohibyte
190   LDA addr          If at last address
200   CMP endaddr       then end,
210   BNE mainloop      Else do another byte
220   LDA addr+1
230   CMP endaddr+1
240   BNE mainloop
250   RTS
260
270.crcbyte
280   LDY #0
290   LDA (addr),Y      Get byte
300   LDX #8            8 bits in a byte
310.loop
320   LSR A             Do crc
330   ROL signature
340   ROL signature+1
350   BCC nextbit
360   PHA
370   LDA signature
380   EOR #&2D
390   STA signature
400   PLA
410.nextbit
420   DEX get new bit in byte
430   BNE loop
440   RTS
450]
460 NEXT pass
470 INPUT "start address &"start$
480 !addr = EVAL("&"+start$)
490 INPUT"Length &"length$
500 !endaddr = EVAL("&"+length$+"+&"+start$)
510 CALL crc
520 PRINT "Signature is &";~!signature AND &FFFF
```

12.5 General purpose macros
Get low byte -- FNlo
DEF FNhi (value) value AND &FF
An example call is

```
LDA #FNlo(table)
```

Get hi byte --FNhi
DEF FNhi (value) (value AND &FF00) DIV &100
An example call is

```
LDY #FNhi(string)
```

Reserve space -- FNspace

```
DEF FNspace(amount)
P% = P% + amount
0% = 0% + amount This line is only relevant on BASIC
                II or electron
= pass
```

An example call is

```
.table
   OPT FNspace(500)
```

16 bit addition -- FNadc
Provides 16 bit addition.

A typical call might be:

```
OPT FNadc(&3000, &2000, &2004)
```

This would add the contents of &2000 (low byte) and &2001 (high byte) to the contents of &3000 (low byte) and &3001 (high byte) and store the result in locations &2004 and &2005.

```
 DEF FNadc(operand1, operand2, result)
[OPT pass
   LDA operand1
   CLC
   ADC operand2
   STA result
   LDA operand1+1
   ADC operand2+1
   STA result+1
```

```
]
= pass
```

16 bit subtraction -- FNsbc
Provides 16 bit subtraction

A typical call would be

```
OPT FNsbc(&3000, &2000, &2004)
```

This would subtract the contents of &2000 (low byte) and &2000 (high byte) from the contents of &3000 (low byte) and &3001 (high byte) and store the result in locations &2004 and &2005.

```
 DEF FNsbc(operand1, operand2,result)
[OPT pass
   LDA operand1
   SEC
   SBC operand2
   STA result
   LDA operand1+1
   SBC operand2+1
   STA result+1
]
= pass
```

Debugging macro -- FNdebug
This can be inserted anywhere in the sources code to provide an indication of which path the processor has taken through the program. When executed, the routine will make a ' Bleep'sound and wait for a key to be pressed before continuing. All registers are preserved.

A typical call would be

```
OPT FNdebug
```

```
 DEF FNdebug
[OPT pass
  PHP              Save all registers
  PHA
  TYA
  PHA
```

```
  TXA
  PHA
  LDA #7            Make 'Bleep' sound
  JSR oswrch
  LDX #1            Flush keyboard buffer
  LDA #15
  JSR osbyte
  JSR osrdch        Wait for a key
  PLA               Restore all registers
  TAX
  PLA
  TAY
  PLA
  PLP
]
 = pass
```

16 bit rotation -- FNshift
This provides a 16-bit shift instruction.

A typical call might be

```
OPT FNshift(&2034, TRUE, 2)
```

which would shift locations &2034 and &2035 right twice.

```
 DEF FNshift(addr, right, number)
 LOCAL shift
FOR shift 1 TO number
  IF right [OPT pass :LSR addr +1 :ROR addr :]
      ELSE [OPT pass ASL addr: ROL addr + 1: ]
NEXT shift
 = pass
```

12.6 BASIC routines for use with assembler
Double height -- PROCdouble

The next two routines can be used to produce double height characters in MODEs 0,1,2,4 and 5. MODEs 3 and 6 (and 7 on the Acorn Electron) have gaps between lines which make it impossible to do double height. To centre the String, type ' PRINT TAB ((screenwidth DIV 2) - LEN(A$) DIV 2,VPOS); after ' LOCAII%' Note that ' blockis a global array which should be DIMensioned at the start of the program, using, for example, DIM block 9. Note also that ' char'

is the character to be defined, in this case it is always 224. The routine currently prints out the characters as it redefines them, although it is possible to suppress this.

```
DEFPROCdouble (A$)
LOCAL I%
  FOR I% = 1 TO LEN(A$)
PROCchar(ASC(MID$(A$,I%,1)),224)
NEXT I%
ENDPROC

DEFPROCchar (C%,char)
LOCAL A%,X%,Y%,J%,I%
?block = C%
A% = 10
X% = FNlo(block)
Y% = FNhi(block)
CALL osword           osword is at &FFF1
FOR J% = 0 TO 1
  VDU 23, char
  FOR I% = 2 TO 9
    VDU block?(J% * 4 + I% DIV 2)
  NEXT I%
VDU char,10,8
NEXT J%
VDU 11,11,9
ENDPROC
```

Find string in program -- FROCfind

This next routine will find all occurrences of a specified string in the BASIC program, and print out the line numbers in which the string occurs. In its present form, the routine will not find BASIC keywords (FOR, REPEAT, PROC, etc). To allow for this it will be necessary to store the string as a line of BASIC, which could then be used as the target string in the search. So type PROCfind ($(PACE + 4)). This will find the string specified in the first line of the program, which should be line 0 to avoid the routine searching for the wrong string. The first line of the routine can now be changed to ' DEFPROCfind' as the parameter is now in line 0. Note that the first IF statement is only needed in BASIC I.

```
DEFPROCfind (A$)
```

```
LOCAL Z%,A%
Z% = PAGE
REPEAT A% = Z% + 4
  IF LEN ($A%) >= LEN (A$) IF INSTR($A%,A$)
     PRINT Z%?1 * 256 + Z%?2: Z% = Z% + Z%?3
UNTIL Z%?1 > &7F
ENDPROC
```