Chapter Seven

Using Subroutines, Macros and Look-up Tables

User subroutines

Mahine code programs which are called from, and intended to return to, BASIC via RTS are essentially subroutines. However, it is a common requirement for the machine code program itself to use subroutines, either user-designed or one of the many resident subroutines embedded within the operating system. Subroutines designed by the user are called by JSR followed by an operand, either an absolute machine address (not recommended) or a destination label. As in BASIC, machine code subroutines can be nested one within the other. Enthusiasm for high nesting levels should not be carried to excess or the stack could overflow. Each unreturned JSR uses up two stack locations, storing the two-byte return address in the Program Counter. No provision is made in the 6502 for saving the other registers. It is up to the programmer to make provisions for protecting valuable register data from corruption by the subroutine. Subroutines are best avoided altogether within loops which are time-critical. Each JSR squanders 6 clock cycles and RTS another six. It is far better to splice the code within the main program, even if it means writing the same segment of code several times.

Resident subroutines

Acorn strongly advise that programs affecting the input/output devices screen, keyboard, printer, etc.) do so via the appropriate resident subroutines. Circumvention, by writing your own, is not dangerous but the program may not work if a second processor is added. This need not be an effective deterrent. You may never consider owning the second processor. You may feel that the advantages of writing your own input/output routines outweigh other considerations. There are certain speed advantages to be gained by using direct screen addressing techniques. Some fascinating (often bizarre)

graphic displays cun be produced. Objects can be persuaded to move across the screen with far less, in fact almost imperceptible, flicker. However, it should be realised that writing your own code for screen animation is not going to be easy. The resident subroutines are excellent but we should bear in mind that design constraints arc inevitable during the development of general purpose software. In the case of the BBC system, additional constraints are imposed by the problem of maintaining compatibility with the Tube. It comes down in the end to a question of personal choice whether you regard or disregard Acorn's warning. The choice is simple: use the resident subroutines and feel safe, or bow to the spirit of adventure and experiment with your own. The designers of the BBC operating system have bent over backwards to provide free access to most of the ROM's internal anatomy. It would be hard to find a competitive machine which offers more scope for experimentation.

It would be pointless at this stage to plod wearily through the entire repertoire of operating system calls. The complete list appears in the User Guide but, for convenience, has been repeated in slightly different form in Appendix B.

The practical programs that appear throughout this chapter should be useful for insertion as subroutines in BASIC programs employing graphics and sound.

Vectors and indirection

One term which crops up when reading literature on operating system calls is *vector* so it is important to be certain of its meaning:

> A vector is a word (normally 2 bytes) in memory which contains the address of a particular routine.

Detailed specifications of routines might include something like:

OSRDCH: Calling address=&FFE0 : Indirected through &0210

The word 'indirected' can be read as 'internally redirected' and, in the above case, refers to the vector in address &0210. Thus, although we would call OSRDCH at &FFE0, the code at that address is not OSRDCH but simply information where OSRDCH can be found.

Why all this apparently needless treasure trail? The answer lies in flexibility. There are three possibilities for the machine code programmer:

(1) Intercept the standard operating system call by simply changing the vector (changing the address in &0210 in the above example). Calling OSRDCH now at address &FFE0 would call up a different routine, written by the user.

Program 7.1. Read text cursor position.

```
>LIST
   10 REM RERD TEXT CURSOR POSITION
   20 REM USING RN OSBYTE CALL
   30 MODE6
   40 DIM START 256
   50 OSBYTE=&FFF4
   60 XPOS=&70:YPOS=&71
   70 FOR PASS=0 TO 2 STEP 2
   80 P%=START
   90 [OPT PASS
  100 LDA #&86        \READ CURSOR POSITION
  110 JSR OSBYTE
  120 STX XPOS        \STORE X AND Y REG'S
  130 STY YPOS        \IN ZERO PAGE LOC'S
  140 RTS:]
  150 NEXT PASS
  160 REM RANDOMISE CURSOR POSITION
  170 VDU31,RND(24),RND(24)
  180 CALL START
  190 PRINT"XPOS=";?XPOS;" YPOS=";?YPOS;
```

(2) Intercept as before but using some preliminary code at the vectored address to modify the normal call. The original routine could then be re-entered.
(3) Operating system ROMs can be updated or modified without affecting the original call. All that needs to be changed are the contents of the vector.

Program 7.2 Typing practice

```
>LIST
   10 REM TYPING PRACTICE PROGRAM
   20 MODE6
   30 DIM START 256
   40 OSASCI=&FFE3:OSRDCH=&FFE0
   50 OSNEWL=&FFE7:OSWRCH=&FFEE
   60 FOR PASS=0 TO 3 STEP 3
   70 P%=START
   80 [OPT PASS
   90 LDA #12
  100 JSR OSWRCH \CLEAR SCREEN
  110 .BEGIN
  120 JSR OSRDCH \ACC=ASCII(KEY HIT)
  130 CMP #ASC("*") \COMP TO "*" ASCII
  140 BEQ FINISH \BR. FINISH IF =
  150 JSR OSASCI \SEND TO SCREEN
  160 BNE BEGIN \BR. BEGIN IF <>
  170 .FINISH
  180 LDA #7 \ACC=ASCII FOR BELL
  190 JSR OSWRCH \OUTPUT ACCUMULATOR
  200 JSR OSNEWL \EXIT WITH OF LINE
  210 RTS:]
  220 NEXT PASS
```

```
230 CALL START
```

Not all the system calls are indirected. Some of those that are include OSWRCH, OSRDCH, OSCLI, OSBYTE, OSWORD. Nearly all the vectors are situated in page 2 although there are a few which can extend into page &0D when using ROM paging Note carefully that page &0D is often referred to as the ' user subroutine area' .

Three examples, using the simpler routines are given, with outline explanation, in the three following programs. Program 7.1 reads the current text cursor position using OSBYTE. Program 7.2 is a simple typing practice program using OSASCI, OSRDCH, OSNEWL and OSWRCH To exit the program, enter *.

Using OSWORD

Program 7.3 is an example using the sound generators with envelope shaping via a pair of OSWORD calls. Lines 50 to 130 set up the respective parameter block data at &1E00 (an arbitrary address). The sound parameter block data is given in line 120 and the envelope parameter block data in line 130. The data given produce a laser ' zapping' sound, used etensively in many ' shoot out of the sky' types of game.

Program 7.3. Using the sound generator with envelope shaping.

```
>LIST
   10 REM USING THE SOUND GENERATORS
   20 REM WITH ENVELOPE SHAPING
   30 REM (LASER GUN TYPE NOISE)
   40
   50 P%=&1E00
   60 FOR item=1 TO 22
   70 READ D$
   80 D=EVAL(D$)
   90 ?P%=D:P%=P%+1
  100 NEXT item
  110
  120 DATA 1,0,1,0,200,0,3,0
  130 DATA 1,1,0,-4,0,0,50,0,42,&F0,&FE,&FE,126,94
  140
  150 OSWORD=&FFF1
  160 FOR PASS=0 TO 2 STEP 2
  170 P%=&1D00
  180
  190 [OPT PASS
  200 LDA #8          \CALL OSWORD WITH 8
  210 LDX #&08        \IN THE ACCUMULATOR
  220 LDY #&1E        \AND ENV PARA BLOCK
  230 JSR OSWORD      \ADDRESS IN X RND Y
  240
  250 LDA #7          \CALL OSWORD WITH 7
```

```
260 LDX #&00        \IN THE ACCUMULATOR
270 LDY #&1E        \AND SOUND PRRA BLOCK
280 JSR OSWORD      \ADDRESS IN X AND Y
290
300 RTS:]
310 NEXT PASS
320 CALL &1D00
```

Readers will no doubt be aware that this is the assembler equivalent of the SOUND and ENVELOPE statements used in BASIC. Notice that two bytes are used for each of the four SOUND parameters which are channel, envelope number, pitch and duration. The high-bytes are usually zero, except for the volume/envelope number parameter which can take a negative value, thus requiring &FF as the high-byte. On the other hand, the 14 envelope parameter block data items are all single-byte entities.

As a point of interest, alternative data for an explosion, gun shot and bonus signal are given below.

Explosion data:
120 DATA 0,0,1,0,6,0,5,0
130 DATA 1,10,0,0,0,0,0,0,42,&F0,0,&FE,126,94

Gun shot data:
120 DATA 0,0,1,0,5,0,4,0
130 DATA 1,10,0,0,0,0,0,0,126,-16,0,-16,126,94

Bonus signal data:
120 DATA 1,0,1,0,200,0,50,0
130 DATA 1,1,0,20,0,0,10,0,0,0,0,-127,126,0

Perhaps the most useful to the machine code programmer is OSWRCH so it deserves more detailed treatment. Any examples given assume that symbolic operands used, such as OSWRCH itself, have been prior- assigned in BASIC. Such names are of mnemonic value only. They are not recognised by the operating system until equated to a specific machine address, in this case, &FFFE.

Using OSWRCH

OSWRCH writes the ASCII character code in the accumulator to ' the currently selected' output device.

The term ' currentlyselected' refers to either the screen, printer or RS423 interface. The default condition is to screen and printer only. Other combinations can be achieved by a prior call to OSBYTE (see later).

The way OSWRCH works is as follows:

(a) Calling address &FFEE (indirected via &020F).
(b) The A,X and Y registers have their contents preserved.
(c) The C,N,V and Z flags are undefined.

Example 1: LDA #72
JSR OSWRCH \Prints "H" on the screen
Example 2: LDA ASC("H")
JSR OSWRCH

Relating OSWRCH to VDU codes

However, OSWRCH is capable of much more than is suggested above, This is due to the cunning use of the 32 control codes which extend through the ASCII range 0 to 31. This code band was left vague when ASCII was launched way back in primeval times. It was felt that a degree of latitude was desirable at the bottom end to allow for individual hardware design, All the graphics facilities available in BASIC can be obtained in machine code by means of OSWRCH. Page 378 of the User Guide lists the VDU code summary. VDU statements can, amongst other things, control screen colour, define graphics windows and various x,y plotting operations. All these can be achieved in machine code by the use of OSWRCH. Columns 1 and 2 on Page 378 of the User Guide are the decimal and hex ASCII control codes. Column 3 relates to the CTRL keys, and column 4 is more or less useless. Column 5 - ' Bytesextra' - is particularly important for our purpose. All the codes have to place the appropriate ASCII code in the accumulator before using JSR OSWRCH. Some, however, require extra trips to OSWRCH, depending on the number of ' extrabytes' Codes which demand ' 0extra bytes are ' one-trip' excursions. Examples are:

```
LDA  #2          \Equivalent to VDU 2
JSR  OSWRCH      \Enable printer
LDA  #16         \Equivalent to VDU 16
JSR  OSWRCH      \Clear graphics area
LDA  #12         \Equivalent to VDU 12
JSR  OSWRCH      \Clear text area
```

VDU equivalents begin to be a little unruly when there are ' extrabytes. Each extra byte involves the setting of another number in the accumulator and, of course, another trip to OSWRCH. Examples are:

```
LDR #17  \VDU 17,2
JSR DBWRCH  \Define text colour 2
LDR #2
JSR OSWRCH

LDA #22 VDU 22,5
JSR OSWRCH  \Set Mode 5
LDA #5
JSR OSWRCH
LDA #25  \VDU 25,0,100;500;
JSR OSWRCH \PLOT K, x, y
LDA #0  \K=0 (move relative to
JSR OSWRCH \last point)
LDA #100  \x=100 (low byte)
JSR OSWRCH
LDA #0  \x=0 (high byte)
JSR OSWRCH
LDA #244  \y=244 (low byte)
JSR OSRDCH
LDA #1  \y=256 (high byte)
JSR OSWRCH
```

Users of BBC BASIC will be aware that VDU statements can be chained together. For example,

VDU 22,2 followed by VDU 24,0;0;1279:,767;
can be written more economically as
VDU 22,2,24,0;0;1279;767;

Although OSWRCH can indeed simulate any VDU statement, there is no point in denying that its use, particularly when chaining lengthy examples, is tedious. In short, it appears to be a weary and ponderous task. Consider for a moment the tedium of typing in scores of LDAs and JSR OSWRCHs involved in drawing a complex graphics screen. The end listing could eventually resemble a toilet roll. We need a routine where the computer does most of the work for us. There are two main methods of performing this task:

one is to use a BASIC procedure acting as a macro; the other is to create a look-up table of data bytes that can be accessed by indexed addressing. The chain of VDU parameters could then be parcelled up neatly within DATA statements and then assembled into equivalent source code.

The macro approach

First, the macro method will be described. The routine is given in Program 7.4 and is universal whichever BASIC ROM happens to be installed. However, a simpler version for those with a BASIC II ROM is given in Program 7.5. This modification is due to the introduction of the EQUS pseudo op-code.

To be useful, the macro should have the following qualities:

(1) It should utilise decimal or hex data.
(2) It should handle single- or double-byte data automatically.
(3) It should assemble singie-byte labelled locations.
(4) It should handle positive and negative data elements whether single- or double-byte in length.

Referring to Program 7.4, an example list of VDU chains is put into DATA statements in lines 470 and 480. The data sets the text and graphics windows and constructs a yellow square on a blue backcloth in MODE 2. Where a two-byte VDU entity is required by the operating system (that is, a number followed by a ',' rather than a ',') the DATA element must be followed by a '@' so that the routine can differentiate between the two. Notice that negative decimal two-byte data can be used which is especially useful in relative plotting. Armed with this routine, a graphics screen can be planned out in BASIC and quickly changed to an assembly language version. The example also shows how labelled locations can be incorporated. The macro can be used once or many times during an assembly program by coming out into BASIC and typing PROCvdu(N) where N is the sequential number of DATA elements you require to incorporate at that particular time. It is essential, however, to restore the DATA pointer at the start of each pass of the assembler (line 290).

Program 7.4. Macro assembly of VDU purameters.

```
>LIST
   10 REM CONDITIONAL ASSEMBLY PROGRRM
   20 REM FOR CHAINING VDU PRRRMETERS
   30 GOTO240
   40
   50 DEFPROCvdu(N)
   60 LOCAL D,D$,B,byte,item,lbyte
```

```
   70 FOR item=1 TO N
   80 READ D$
   90 IF RIGHT$(D$,1)="@" THEN B=2 ELSE
B=1
  100 D=EVAL(D$)
  110 IF ASC(D$)>64 THEN [OPT PASS:LDA D
:JSR OSWRCH:]:GOTO150
  120 IF D<0 THEN D=(ABS(D) EOR &FFFF)+1
  130 byte=D MOD 256:PROCform
  140 IF B=2 THEN byte=D DIV 256:PROCfor
m
  150 NEXT item
  160 ENDPROC
  170
  180 DEFPROCform
  190 IF byte<>lbyte THEN [OPT PASS:LDA
#byte:]
  200 [OPT PASS:JSR OSWRCH:]
  210 lbyte=byte
  220 ENDPROC
  230
  240 OSWRCH=&FFEE
  250 BCOL=&70:SQCOL=&71
  260 DIM START 256
  270 FOR PASS=0 TO 3 STEP 3
  280 P%=START
  290 RESTORE
  310 [OPT PASS
  320 LDA #3          \SET SQUARE CULOUR
  330 STA SQCOL
  340 LDA #&84        \SET BACKGROUND COLOUR
  350 STA BCOL
  360 ]
  370 PROCvdu(39)
  380 [OPT PASS
  390 \ ANY SOURCE CODE
  400 RTS:]
  420 NEXT PASS
  430 CALL START
  450 REM DATA IS A LIST OF VDU CHAINS
  460 REM YELLOW SQUARE/BLUE BACKGROUND
  470 DATA 22,2,28,0,3,19,1,24,0@,0@,127
9@,767@,18,0,BCOL,16
  480 DATA 18,0,SQCOL,25,4,500@,500@,25,
1,200@,0@,25,81,0@,-200@,25,1,-200@,0@,2
5,81,0@,200@
```

Program 7.4 takes advantage of the concept of conditional assembly. In this case, a conditional test is made to see if the next byte of data is the same as the byte preceding it. If this is so then a LDA #byte will not be required (since the data byte will already be in the accumulator).

The operation of the program is as follows: The routine reads in each DATA element and tests for the ' @‰character termination, setting the variable B to the number of bytes as appropriate. Line 100 uses EVAL rather than VAL to evaluate the data string (D$) and places the result into the variable D.

This fine is necessary when the following situations occur in D$.

(1) A '&' precedes a number, indicating hexadecimal.
(2) A labelled location is encountered, in which case the assigned location address is put into the variable D, assuming, of course, it has been previously defined.
(3) The temporary '@' character is ignored (it has outgrown its usefulness).

Line 110 tests whether D$ contains numeric or alpha data. If the data is alpha - that is, a labelled location - then the accumulator is loaded with the assigned address since, in this case, we do not want immediate addressing.

Line 120 checks if D is negative so that a two-byte two's complement form can be generated. Line 130 forms the low-byte and line 140 forms the high-byte if required. PROCform handles conditional assembly using immediate addressing.

With the upgraded BASIC II ROM installed, the above program can still be used but a more convenient version is given in Program 7.5. The difference is that we need not leave the assembler to use the macro. The pseudo op-code EQUS can place a string, where positioned, within a program. If we use the function FNvdu(N) which returns a null string(we do not want a string returned) then all the goings on of the macro will be peformed without the complication of leaving the assembler!

This type of modification can be employed in similar programs where procedures are encountered on breaking out of the assembler. For the sake of standardisation, the method is not used in further examples since large numbers of BBC Micro's have BASIC I installed.

Program 7.5. Macro assembly of VDU parameters (BASIC II onwards).

```
>LIST
   10 REM CONDITIONAL ASSEMBLY PROGRRM
   20 REM FOR CHAINING VDU PRRRMETERS
   30 GOTO240
   40
   50 DEF FNvdu(N)
   60 LOCAL D,D$,B,byte,item,lbyte
   70 FOR item=1 TO N
   80 READ D$
   90 IF RIGHT$(D$,1)="@" THEN B=2 ELSE
B=1
  100 D=EVAL(D$)
  110 IF ASC(D$)>64 THEN [OPT PASS:LDA D
:JSR OSWRCH:]:GOTO150
  120 IF D<0 THEN D=(ABS(D) EOR &FFFF)+1
  130 byte=D MOD 256:PROCform
  140 IF B=2 THEN byte=D DIV 256:PROCfor
m
  150 NEXT item
  160 =""
  170
```

```
  180 DEFPROCform
  190 IF byte<>lbyte THEN [OPT PASS:LDA#
byte:]
  200 [OPT PASS:JSR OSWRCH:]
  210 lbyte=byte
  220 ENDPROC
  230
  240 OSWRCH=&FFEE
  250 BCOL=&70:SQCOL=&71
  260 DIM START 256
  270 FOR PASS=0 TO 3 STEP 3
  280 P%=START
  290 RESTORE
  300
  310 [OPT PASS
  320 LDA #3 \SET SQUARE CULOUR
  330 STA SQCOL
  340 LDA #&84 \SET BACKGROUND COLOUR
  350 STA BCOL
  360 EQUS FNvdu(39)
  370 RTS:]
  380
  390 NEXT PASS
  400 CALL START
  410
  420 REM DATA IS A LIST OF VDU CHAINS
  430 REM YELLOW SQUARE/BLUE BACKGROUND
  440 DATA 22,2,28,0,3,19,1,24,0@,0@,127
9@,767@,18,0,BCOL,16
  450 DATA 18,0,SQCOL,25,4,500@,500@,25,
1,200@,0@,25,81,0@,-200@,25,1,-200@,0@,2
5,81,0@,200@
```

The look-up table approach

Another approach to this problem is by using a look-up table. Program 7.6 shows the essential differences. For a start, conditional assembly is not used as in the previous example and the use of labelled locations has been dropped. The DATA elements in the BASIC routine are split up into the relevant single bytes and stored as a look-up table from the location labelled ' data'onwards. The short piece of code at lines 230 to 290 can be placed anywhere within the source program to access this table sequentially. The term ' look-up' is used since any element can be looked up by setting the index register to the required offset from the table base address. It is conventional to place data tables such as this at the end of a program. The previous example has the edge on speed but this method is more econormcal in the use of memory locations

Program 7.7 is an example of the type of moving graphics available by using the macro approach. The program sets up a graphics screen in mode 2 and produces the ubiquitous bouncing ball. by now the standard

apprenticeship exercise in the use of animated graphics. The macro is used a line 310 to set up the screen and again at line 980 to form the subroutine BALL.

Program 7.6 Using data tables for chaining VDU parameters.

```
>LIST
   10 REM USING DATA TABLES FOR
   20 REM CHAINING VDU PARAMETERS
   30 GOTO160
   40
   50 DEFPROCdatatable(N)
   60 FOR item=1 TO N
   70 READ D$
   80 IF RIGHT$(D$,1) ="@" THEN B=2 ELSE
 B=1
   90 D=EVAL(D$)
  100 IF D<0 THEN D=(ABS(D) EOR &FFFF)+1
  110 ?P%=D MOD 256:P%=P%+1
  120 IF B=2 THEN ?P%=D DIV 256:P%=P%+1
  130 NEXT item
  140 ENDPROC
  150
  160 OSWRCH=&FFEE
  170 DIM START 500
  180 FOR PASS=0 TO 3 STEP 3
  190 P%=START
  200 RESTORE
  210
  220 [OPT PASS
  230 LDY #0          \LOOP OUTPUTS
  240 .LOOP           \DATA ITEMS STORED
  250 LDA data,Y      \FROM data ONWARDS
  260 JSR OSWRCH      \BY PROCdatatable
  270 INY             \AND CAN BE REPLACED
  280 CPY #53         \ANYWHERE WITHIN
  290 BNE LOOP        \SOURCE CODE PROG.
  300 BEQ FINISH
  310 .data
  320
  320 ]PROCdatatable(39)
  330 [OPT PASS
  340 .FINISH
  350 RTS:]
  360
  370 NEXT PASS
  380 CALL START
  390
  400 REM DATA IS A LIST OF VDU CHAINS
  410 REM YELLOW SQUARE/BLUE BACKGROUND
  420 DATA 22,2,28,0,3,19,1,24,0@,0@,127
9@,767@,18,0,132,16
  430 DATA 18,0,3,25,4,500@,500@,25,1,20
0@,0@,25,81,0@,-200@,25,1,-200@,0@,25,81
,0@,200@
```

Any Moving Object (MOB) must have an associated velocity. In simple cases this is little more than an increment added to the object' sprevious position so as to calculate its next position on the screen. An acceleration, incidentally, can be mimicked (position-wise) by adding a further steadily inreasing increment as new positions are calculated. When the object reaches a boundary, the velocity (increment) must reverse sign if it is to remain on the screen. This is straightforward to program in BASIC. In machine code, however, the object' sposition on the screen is a two-byte number. You could be forgiven for thinking that the increment need only he a single-byte number because of its normally small value. However, this r, not so. Numbers differing in byte length cannot be added if they are of mixed sign, therefore we must also have a two-byte increment.

Program 7.7. The ubiquitous bouncing ball.

```
>LIST
   10 REM BOUNCING BALL EXAMPLE
   20 GOTO 230
   30
   40 DEFPROCvdu(N)
   50 LOCAL D,D$,B,byte,item,lbyte
   60 FOR item=1 TO N
   70 READ D$
   80 IF RIGHT$(D$,1)="@" THEN B=2 ELSE
B=1
   90 D=EVAL(D$)
  100 IF ASC(D$)>64 THEN [OPT PAS:LDA D:
JSR OSWRCH:]:GOTO 140
  110 IF D<0 THEN D=(ABS(D) EOR &FFFF)+1
  120 byte=D MOD 256:PROCform
  130 IF B=2 THEN byte=D DIV 256:PROCfor
m
  140 NEXT item
  150 ENDPROC
  160
  170 DEFPROCform
  180 IF byte<>lbyte THEN [OPT PASS:LDA
#byte:]
  190 [OPT PASS:JSR OSWRCH:]
  200 lbyte=byte
  210 ENDPROC
  220
  230 OSWRCH=&FFEE:OSBYTE=&FFF4
  240 X=&70:Y=&72:XINC=&74:YINC=&76
  250 BCOL=&78
  260 DIM START 1000
  270 FOR PASS=0 TO 3 STEP 3
  280 P%=START
  290 RESTORE
  300 REM SET UP SCREEN
  310 PROCvdu(16)
  320 DATA 22,2,28,0,3,19,1,24,0@,0@,127
```

```
9@,767@,18,0,132,16
  330
  340 [OPT PASS
  350 LDA #0 \INITIALISE X,Y,XINC
  360 STA X \AND YINC
  370 STA Y
  380 STA XINC+1
  390 STA YINC+1
  400 STA X+1
  410 STA Y+1
  420 LDA #8
  430 STA XINC
  440 STA YINC
  450
  460 .LOOP
  470 LDA #&13
  480 JSR OSBYTE
  490 LDA #4
  500 STA BCOL
  510 JSR BALL
  520 LDA X
  530 CLC
  540 ADC XINC
  550 STA X
  560 LDA X+1
  570 ADC XINC+1
  580 STA X+1
  590 LDA Y \ADD YINC TO Y
  600 CLC \(2 BYTES)
  610 ADC YINC
  620 STA Y
  630 LDA Y+1
  640 ADC YINC+1
  650 STA Y+1
  660 LDA #7 \DELETE BALL WITH
  670 STA BCOL \BACKGROUND COLOUR
  680 JSR BALL
  690 LDA Y+1
  700 CMP #3
  710 BCC YONSCR
  720 LDA YINC \FIND 2's COMPLIMENT
  730 EOR #&FF \OF YINC (2 BYTES)
  740 STA YINC
  750 LDA YINC+1
  760 EOR #&FF
  770 STA YINC+1
  780 INC YINC
  790 BNE YONSCR
  800 INC YINC+1
  810 .YONSCR
  820 LDA X+1 \CHECK X ON SCREEN
  830 CMP #5
  840 BCC XONSCR
  850 LDA XINC \FIND 2's COMPLIMENT
  860 EOR #&FF \OF XINC (2 BYTES)
  870 STA XINC
  880 LDA XINC+1
  890 EOR #&FF
```

```
  900 STA XINC+1
  910 INC XINC
  920 BNE XONSCR
  930 INC XINC+1
  940 .XONSCR
  950 JMP LOOP
  960
  970 .BALL
  980 ]PROCvdu(13)
  990 DATR 18,0,BCOL,25,4,X,X+1,Y,Y+1,25
,1,0@,-8@
 1000 [RTS:]
 1010
 1020 NEXT PASS
 1030 CALL START
 1040 END
```

The program is liberally remarked but one area worthy of further comment is where the on-screen position checks are made. Referring to the case of the Y screen boundary checks in lines 690 to 710, the CMP instruction sets the carry flag if M<=A and clears it if M>A. In this case, the CMP #3 instruction in line 700 will set the carry flag if the accumulator mntents Y+1>=3. This will occur when the accumulator contents are between 3 and 255 inclusive (&3 and &FF). Consequently, the carry flag wfll be clear when the high byte of the screen position Y+1 takes a value of 0,1, or 2 which are legitimate screen positions. The end result of the conditional branch in line 710 is that XINC (2 bytes) is only reversed in sign (two' scomplement) when the carry flag is set. Therefore, only one comparison is required to test for both the top and bottom screen boundaries. The screen boundary tests in the other dimension are conducted in a similar fashion.

Summary

1. High levels of subroutines nesting can overflow the stack.
2. Each unreturned JSR uses two stack locations.
3. Each JSR uses 6 clock cycles and RTS another 6, so a JSR within a loop can squander time.
4. Resident subroutines, within the ROM operating system, are plentiful and easy to use in machine code programs.
5. Where the screen display is involved, there are speed advantages to be gained by using direct screen addressing. However, the program may not work through the Tube with the second processor. If you never buy one, this won' t matter anyway.
6. A vector is a two-byte word in memory which is the address of a routine.

7. Some resident subroutines have vectored addresses. Changing the contents of the vector allows interception to a different routine.

8. Most vectored addresses are in page 2.

9. Graphic facilities are handled by OSWRCH at address &FFEE. indirected via &020E.

10. The sound generators and envelope shaping are handled by OSWORD.

11. Input data is handled by OSRDCH at address &FFE0, indirected via &0210.

12. Although the assembler does not offer macro facilities, they can be simulated by temporary transfer from assembly code to a BASIC procedure.

13. If the new BASIC II ROM is installed, the EQUS structure can be used within the assembler for simulating macros and other functions

Self test

7.1 Use the macro procedure to draw a large red square on a yellow background in MODE 2.

7.2 Use the look-up table method to draw a yellow triangle on a blue background.

7.3 Program the sound generator to play a scale in F# major.

7.4 Add a keyboard controllable bat routine to intercept the ball in Program 7.7.

7.5 Employ the user-definable character method for plotting the ball in Program 7.7.