Chapter Eight
Assembly Language II

In this chapter the BBC Micro's assembler is examined in a little more detail. Some of its less obvious features will be discussed and illustrated by a number of short projects. These projects are complete in themselves but they are also intended to form the basis for your own experiments.

Labels and P%

Almost the whole story behind assembly language labels is contained in the statement that labels are just BASIC variables that are used to hold an address. However, the wider meanings of this observation are worth spelling out. There are two ways to set a variable equal to an address. You can use it to *label* a position in a program by writing its name preceded by a full stop (as described in the last chapter) or you can assign a value to it just like any other variable. Assigning an address to a label can be used to make assembly language programs easier to read and write. For example, instead of writing JSR &FFF4 to call OSBYTE (see Chapter Three) you can write

```
10 OSBYTE%=&FFF4
20 [ JSR OSBYTE%
...
```

While on the subject of storing addresses in variables, it is worth pointing out that you can use either real or integer variables to hold addresses but, of course, integer variables will make the assembler work slightly faster and will save storage space in the variables area of memory.

It is a fundamental principle of BASIC that anywhere that you can use a variable or constant you can use an expression. For example, wherever you can write PRINT A, you can also write things like PRINT A*2+4. Unfortunately, not all versions of BASIC keep to this

rule in all situations. BBC BASIC sticks to it almost without exception. (You cannot use expressions in place of the constants in *FX commands but these are implemented by the MOS, not by BASIC). Labels are no exception to this general rule and so anywhere you can use a label you can use an expression. For example,

LDA #ASC("A")	Loads the A register with the ASCII code for the letter A.
STA DATA+2	Store the contents of the A register in the memory location whose address is the result of DATA+2

If you want to load a 16-bit value then it has to be split into two bytes, an MSB and a LSB. This can be done very easily using the MOD and DIV functions. For example:

LDX (DATA% MOD 256) loads X with the LSB

and

LDY (DATA% DIV 256) loads Y with the MSB

The ability to use expressions to work out addresses in the assembler is very useful. It should be remembered, however, that the expressions are evaluated when the assembly language is being assembled into machine code, and not when the machine code program is run.

The resident integer variable P% has already been discussed in the previous chapter in the context of setting the start address of the memory into which the machine code is loaded. However, P% can be used for much more than just this. The value stored in P% is continually changing during the assembly so as to always hold the address of the memory location that the next byte of machine code will be stored in. Thus, at the end of a successful assembly, P% points to the first memory location after the machine code program. You should now be able to see that the definition of a label using a full stop is the same as setting it equal to the value of P%. In other words:

```
100  [
110    LDA DATR
120 .LABEL STR DATA+1
130    JMP LOOP
```

is the same as

```
100 [
110  LDA DATA ]
120 LABEL=P%
130 [
140 STA DATA+1
150 JMP LOOP ]
```

Notice that it is necessary to leave the assembler to set the variable 'LABEL' equal to 'P%'. (This idea of leaving assembler in the middle of a program, using some BASIC and returning to the program is useful in itself and will be discussed later.) Not only can the value of P% be assigned to other variables, it can also be altered. For example, if you want to use a few memory locations in the middle of a program to store data then you can leave assembler, increase the value of P% by the number of bytes you require and then return to assembler. If you want to refer to the memory locations by name then all you have to do is set up labels equal to intermediate values of P%. For example, if you want to use three memory locations for data storage and call them DATA1, DATA2 and DATA3 all you have to do is:

```
    assembly language
 ]
  DATA1=P%
  DATA2=P%+1
  DATA3=P%+2
  P%=P%+3
 [
assembly language continued
```

The line numbers have been left out for clarity. You can use indirection operators to set initial values in memory locations used for data. You could add ?DATA1=33 to the last example if you wanted the memory locations whose address we stored in DATA1 to be initialised to 33.

Getting results to and from assembly language subroutines

The idea of passing values to assembly language subroutine using the resident integer variables was discussed in the previous chapter. The USR function caters for the need to return a few simple values but it can involve the use of messy expressions with AND and DIV to separate the different parts of the result. A cleaner, though non-standard

method, is to use the information presented in Chapter Two concerning the locations of the resident integer variables to store results into them directly. You can see an example of this method in Project 2.

For more complicated situations there is really no choice but to use the parameter-passing facility of the CALL statement. Following a CALL with parameters, information is stored starting at &600 as to the type and location of each parameter. This information can be used by assembly language programs to 'collect' data from or store results in BASIC variables. The details of this are straightforward and as there is an example given in the User Guide the subject will not be discussed further.

Using memory

Reserving memory for assembly language subroutines and data seems to be no problem on the BBC Micro. However, there are two instances where the usual DIM statement method of reserving memory is inadequate. The first case is when you want to 'hide' a machine code program in RAM so that other programs can be written without having to worry about the machine code while still having it available for use. An example of this problem will be found in Project 1 where a screen dump program is written. Ideally this program should be stored in an area of memory that isn't used by a BASIC program and one of the function keys set to produce the string CALL'address'|M where 'address' is the start of the screen dump program. This would allow the screen dump program to be loaded before any other BASIC program and be called into action by pressing the function key. There are several places where machine code can be 'tucked away' but none of them is particularly satisfactory. For example, you can lower the value of HIMEM and stored the machine code just below the screen memory. The trouble is that changing modes changes the value of HIMEM, possibly destroying the machine code. However, if you can ensure that a MODE statement will not be used, then altering HIMEM is a possibility. A much better way of hiding machine code is to alter PAGE but this is more difficult because the program that creates the machine code must be loaded and RUN AFTER the PAGE has been altered. For more details see Project 1.

The second situation where DIM is insufficient concerns the allocation of memory for data storage. Locations in page zero are

particularly important to the 6502. For example, the only way that you can specify a variable 16-bit address is by using:

```
LDY #0
LDA (ADDRESS),Y
```

where ADDRESS holds the address of the first of two page zero locations. The effect of this pair of instructions is to load the A register from the location whose address is stored in the two page zero memory locations. In this way pairs of memory locations in page zero can be used as ' 16-bipointers' to other memory locations. For this reason the BBC Micro sets aside page zero locations from &70 to &8F for user routines. Page zero locations are in short supply so use somewhere else unless you really *need* page zero.

Conditional assembly and macros

One of the great advantages of having a 6502 assembler as part of BASIC is that you can make use of all the BASIC statements in the translation of a program to machine code. For example, if you wanted to write a machine code subroutine that was to be used for a number of purposes and each purpose required a slightly different version, then you could use an IF statement to select which version was actually assembled. To illustrate this, consider the problem of setting the transmission speed of a communications program (see Project 2.) You could use something like:

```
IF FAST=1 THEN [ LDA 600:] ELSE [ LDA #300:]
```

where FAST is a normal BASIC variable that controls which of the two assembly language statements is translated into the machine code program. Notice that the IF is only carried out when the assembly language is being converted to machine code. It is not part of the resulting machine code program. (The colon before the] can be used instead of a carriage return to end an assembly language statement. In fact the colon can be used to put multiple statements on one line just as in BASIC!)

This selective assembly is a powerful tool and makes the BBC Micro' ssimple assembler look more like a complex *macro assembler*. As another example, suppose you needed to carry out the same

operation ROR A a number of times. Instead of writing it out each time, why not use:

```
FOR I=1 TO 4
[ ROR A
]
NEXT I
```

which will include four ROR A instructions in any assembly language program that it is part of. The best way to understand what each of these examples does is to type them in with some other assembly language and see what appears on the listing.

Finally, it is worth pointing out the BBC assembler coupled with BBC BASIC provides a full macro facility! A macro is like a subroutine except that it produces the necessary assembly language each time it is called. Once again, it is easier to understand this idea by means of an example. It is often necessary to add the contents of two memory locations together and store the answer in a third. The following ' macro' generates assembly language to do just this:

```
DEF PROCADD(N1,N2,ANS)
[ CLC
  LDA N1
  ADC N2
  STA ANS
]
ENDPROC
```

To add the two numbers in DATA1 and DATA2 and store the answer in DATA3 you would simply write:

```
PROCADD(DATA1,DATA2,DATA3)
```

This would generate the correct assembly language at the position at which it was used. You can use PROCADD as many times as you like in a program. A new copy of the assembly language will be produced each time. You can even use local labels within a macro simply by naming them in a LOCAL statement! The possibilities are endless!

Project 1 - a text screen dump program

The object of this project is to write a screen dump program. A general screen dump program would be capable of printing whatever was

displayed on the screen in any mode. This includes attempting to give a representation of the colours used as a grey scale! This is not an easy task and it is not possible to tackle the problem in general because the solution depends on which of the many graphics printers is available. To make the problem a little easier and as a starting point for more advanced programs we will only try to dump mode 7 text screens.

There are many ways of approaching the problem of writing a mode 7 screen dump program. At first thought it might seem like a good idea to use the screen memory map to retrieve the ASCII code stored in each screen location and send them to the printer in turn. However, this method becomes very difficult if the screen has scrolled because of the increased complexity of the memory map. Rather than have to live with a 'noscroll' restriction it is better to use the MOS subroutine OSBYTE called with A=135. This returns the ASCII code of the character under the text cursor position. Once we have decided to use this OSBYTE call, the other problems within the program are:

1. Sending characters to the printer only.
2. 'Scanning' the text cursor across the screen.
3. Sending carriage returns at the end of each line full of the screen.

Each of these problems is also solved with the help of an MOS call. The resulting program is:

```
 10 DIM MACH% 150
 20 OSBYTE%=&FFF4
 30 OSWRCH%=&FFEE
 40 OSASCI%=&FFE3
 50 XCORD%=MACH%
 60 YCORD%=MACH%+1
 70 CODE%=MACH%+2
 80 PROCASMB
 90 CLS
100 FOR I=1 TO 24
110 PRINT "TEST OUTPUT ",I
120 NEXT I
130 CALL CODE%
140 STOP

150 DEF PROCASMB
160 FOR PASS=0 TO 3 STEP 3
170 P%=CODE%
180 [OPT PASS

190 LDA #5
200 LDX #1
210 JSR OSBYTE%
220 LDX #0
```

```
230 STX XCORD%
240 STY YCORD%
250 .SCAN% LDA #31
260 JSR OSWRCH%
270 LDA XCORD%
280 JSR OSWRCH%
290 LDA YCORD%
300 JSR OSWRCH%
310 JSR DUMP
320 LDX XCORD%
330 INX
340 STX XCORD%
350 CPX #32
360 BNE SCAN%
370 LDX #0
380 STX XCORD%
390 LDX #&0D
400 JSR PRN
410 LDX YCORD%
420 INX
430 STX YCORD%
440 CPX #25
450 BNE SCAN%
460 RTS
470 .DUMP LDA #135
480 JSR OSBYTE%
490 JSR PRN
500 RTS

510 .PRN LDA #2
520 JSR OSWRCH%
530 LDA #21
540 JSR OSWRCH%
550 TXA
560 JSR OSWRCH%
570 LDA #6
580 JSR OSWRCH%
590 LDA #3
600 JSR OSWRCH%
610 RTS
620 ]

630 NEXT PASS
640 ENDPROC
```

The first few lines (10 to 70) set up the area of memory that the machine code will be stored in and the values of some labels that will be used later. Notice the way that two memory locations are reserved for data storage by lines 50,60 and 70 - the start of the machine code is stored is stored in CODE%. The main program calls PROCASMB to assemble the dump subroutine to machine code and then prints some test lines on the screen before calling it in line 130. The assembly language dump routine can be seen in line 190 to 620. The rest of PROCASMB implements two passes over the assembly language as described in Chapter Seven. The first part of the machine code 190 to

210 sets up a parallel printer as described in the User Guide using the OSBYTE equivalent of *FX5,1. Lines 220 and 240 initialise the data locations XCORD% and YCORD% to zero to start the scan of every location on the screen. XCORD% is used to hold the x co-ordinate and YCORD% is used to hold the y co-ordinate of the text cursor. Lines 250 to 300 use the VDU drivers via OSWRCH to set the cursor to the position stored in XCORD% and YCORD%. Line 300 calls a machine code subroutine DUMP which sends the character under the cursor to the printer (details given later). The rest of the program, from 320 to 460, is concerned with moving the cursor over the screen by updating XCORD% and YCORD%. After each character is dumped to the printer, XCORD% has one added to it (lines 320 to 340) and it is then compared with 32 to see if we have reached the end of a screen line. If we have, then XCORD% is set to zero (lines 370 and 380), a carriage return is sent to the printer to start a new line (lines 390 and 400) and YCORD% has one added to it to take the cursor down one line (lines 410 to 430). At this point YCORD% is compared to 25 (line 440). If it is equal to 25 then all the lines have been dumped and the subroutine ends (line 500). The only things left to explain are the two subroutines DUMP and PRN. DUMP uses the OSBYTE call discussed earlier to get the ASCII code of the character under the cursor into the X register (lines 470 to 480). It then calls subroutine PRN which sends the character in the X register to the printer only. Subroutine PRN starts by turning the printer on with the equivalent of a VDU 2 command (lines 510 and 520). It then disables the VDU drivers so that the character only goes to the printer using the equivalent of VDU 21 (lines 530 and 550). The character is then sent to the printer using OSWRCH (line 550 and 560). The remainder of the subroutine enables the VDU drivers (lines 570 and 580) and then disables the printer (lines 590 and 600).

There is one last part to this project. The dump subroutine is only really useful if it can be in memory at the same time as any BASIC program - not just the one that produces it. This can be done by first saving the whole dump on tape. Then type NEW and type PRINT ~PAGE and write down the answer. Then alter PAGE by typing PAGE=PAGE+&100. This moves the start of any BASIC program &100 memory locations higher up and leaves space for the machine code. Now load the dump program from tape and alter line 10 to read MACH%= ' thœold value of PAGE' (which you wrote down earlier). Also delete the test section of the main program, lines 90-130. RUN

the program and then delete it using NEW. You can now load and run any other BASIC program you like and obtain a screen dump program by typing CALL ' the old value of PAGE+2' .You can even program a function key to produce the same command and hence dump the screen with a single key stroke!

This program can be easily adapted to dump text from any mode screen by simply changing the values to which XCORD% and YCORD% are compared to detect the end of line and the end of screen respectively. You can even go on to expand the program to dump graphics by using the OSWORD call that returns the value of a single point on the screen but this is more complicated!!

Project 2 - a VDU program

The next project is likely to be of interest only to people with access to another computer because it turns the BBC micro into a VDU. The principle behind this is straightforward and is based on the description of the RS423 interface given in Chapter One. This project is a good example of how BASIC and assembly language can be used together. The program is split into a number of small subroutines. INIT% initialises the ACIA and the serial controller. CHAROUT% sends a character stored in the X register to the ACIA transmit register. CHAREADY checks to see if there is a character in the receive register. It returns its result by storing the A register in the first byte of the resident integer variable A% at &0404. CHARGET% retrieves the character in the receive register. It, too, returns its result through A%. Using these three subroutines and a little BASIC, the entire VDU program can be written:

```
 10 DIM INIT% 20
 20 DIM CHAROUT% 20
 30 DIM CHAREADY% 20
 40 DIM CHARGET% 20
 50 SERCON%=&FE10
 60 ACIACON%=&FE08
 70 ACIASTAT%=&FE08
 80 ACIATRAN%=&FE09
 90 ACIAREC%=&FE09
100 GOSUB 130
110 CALL INIT%
120 GOSUB 490
130 FOR S=0 TO 3 STEP 3
140 P%=INIT%
150 [OPT S
```

```
160  LDA #&13
170  STA ACIACON%
180  LDA #&56
190  STA ACIACON%
200  LDA #&49
210  STA SERCON%
220  RTS
230 ]
240 P%=CHAROUT%
250 [OPT S
260 .OUT1 LDA ACIASTAT%
270  AND #&0A
280  BEQ OUT1
290  STX ACIATRAN%
300  RTS
310 ]
320 P%=CHAREADY%
330 [OPT S
340  LDA ACIACON%
350  AND #&01
360  STA &404
370  RTS
380 ]
390 P%=CHARGET%
400 [OPT S
410  LDA ACIAREC%
420  AND #&7F
430  STA &0404
440  RTS
450 ]
460 NEXT S
470 RETURN
480 REM VDU LOOP
490 X%=INKEY(0)
500 IF X%=-1 THEN GOTO 520
510 CALL CHAROUT%
520 CALL CHAREADY%
530 IF A%=0 THEN GOTO 490
540 CALL CHARGET%
550 PRINT CHR$(A%);
560 GOTO 4900
```

The machine subroutines that make up this program have already been described briefly and should cause no trouble. INIT% sets the baud rate to 1200 but this can easily be changed using the information given in Chapter One. However, notice the alternative way of writing a set of assembly language subroutines using a separate area for each. The main program starts at 490 and forms and infinite loop that can only be broken by pressing ESCAPE or BREAK. Line 490 reads the keyboard to see if a key is being pressed. If one is, then the ASCII code in X% is sent to CHAROUT%. If no key is pressed, CHAREADY% is called to see if there is a character in the receive register. If there is, it is retrieved by CHARGET% (line 540) and printed by line 550.

This program is really only the basis for a full VDU program. It

should give the user the option of selecting any baud rate, parity etc., in an extended initialisation section. However, the program given here does illustrate how assembly language and BASIC and be used together and how the resident integer variables can be used to transfer information.

Project 3 - a moving graphics program

This book would not be complete without a moving graphics program and Project 3 is just that. It forms the basis for a 'squash'-type program. A square graphics character CHR$(224) is bounced around the screen. Because it is written in assembly language this program gives you an idea of just how fast graphics can be on the BBC Micro.

```
 10 MODE 4
 20 DIM MACH% 300
 30 VDU 23,224,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
 40 PROCASMB
 50 PROCPLAY
 60 STOP

 70 DEF PROCASMB
 80 OSWRCH%=&FFEE
 90 OSBYTE%=&FFF4
100 XCORD%=MACH%
110 YCORD%=MACH%+1
120 XVEL%=MACH%+2
130 YVEL%=MACH%+3
140 CODE%=MACH%+4
150 ?XVEL%=1
160 ?YVEL%=-1
170 FOR PASS=0 TO 3 STEP 3
180 P%=CODE%
190 [OPT PASS

200 STX XCORD%
210 STY YCORD%
220 .LOOP JSR SHOW%
230 JSR MOV%
240 JSR BOUNCE%
250 JSR DELAY%
260 JMP LOOP

270 .SHOW% LDA #8
280 JSR OSWRCH%
290 LDA #32
300 JSR OSWRCH%
310 LDA #31
320 JSR OSWRCH%
330 LDA XCORD%
340 JSR OSWRCH%
350 LDA YCORD%
```

```
360 JSR OSWRCH%
370 LDA #224
380 JSR OSWRCH%
390 RTS

400 .MOV% LDA XCORD%
410 CLC
420 ADC XVEL%
430 STA XCORD%
440 LDA YCORD%
450 CLC
460 ADC YVEL%
470 STA YCORD%
480 RTS

490 .BOUNCE% LDA XCORD%
500 CMP #1
510 BEQ FLIPX%
520 CMP #39
530 BNE YBON%
540 .FLIPX% LDA #0
550 SEC
560 SBC XVEL%
570 STA XVEL%
580 .YBON%  LDA YCORD%
590 CMP #1
600 BEQ FLIPY%
610 CMP #31
620 BNE FIN%
630 .FLIPY% LDA #0
640 SEC
650 SBC YVEL%
660 STA YVEL%
670 .FIN% RTS

680 .DELAY% LDA #19
690 JSR OSBYTE%
700 RTS

710 ]
720 NEXT PASS
730 ENDPROC

740 DEF PROCPLAY
750 CLS
760 VDU 23,1,0;0;0;0;
770 X%=RND(15)
780 Y%=RND(5)+10
790 CALL CODE%
800 ENDPROC
```

The main program calls PROCASMB to produce the machine code used by PROCPLAY to bounce the ball around the screen. There are four assembly language subroutines within PROCASMB - MOVE% moves the ball, BOUNCE% makes sure that the ball doesn't go off the screen, SHOW% updates the ball's position on the screen and DELAY% slows things down so that the ball can be seen!

PROCPLAY turns off the cursor (line 760) and sets the starting position for the ball at random (lines 770 and 780) before calling the machine code (line 790). The first thing the machine code does is to use X and Y to initialise XCORD% and YCORD% to the starting position. The main loop within the program calls SHOW%, MOV%, BOUNCE% and DELAY% repeatedly (lines 220 to 260). SHOW% is straightforward and uses the equivalent of VDU 31 to move the cursor to the new position (lines 290 to 360). It then prints the ball character using OSWRCH (lines 370 to 390). The only difficult part of the subroutine occurs at the start, where lines 270 to 300 move the cursor back one place and print a blank (ASCII code 32) to remove the old ball' s position. The MOV% subroutine updates the X and Y coordinates in XCORD% and YCORD% by adding the contents of XVEL% and YVEL% respectively. Notice how these data bytes are initialised in lines 150 to 160. BOUNCE% is the most complicated subroutine, but all it does is to check to see if the new co-ordinates are on the edge of the screen or not. If they are it reverses one of the velocities contained in XVEL% or YVEL%. Reversing a velocity is done by subtracting it from zero (lines 540 to 570 and 630 to 660). DELAY% is a very simple subroutine and merely carries out the equivalent of an *FX 19. This halts the program until the start of the NEXT TV frame. This means that the ball can only move every fiftieth of a second. If you want to see how fast the ball can really move just remove line 250!

You should be able to add a bat to the ball bouncing routine without too much trouble and have one of the fastest squash games ever!

Project 4 - a pulse generator

The final project is more hardware-oriented than the previous three. The BBC Micro contains so many timers that an obvious technical application is to use it as a general purpose pulse generator. Project 4 is a simple pulse generator that can produce a square wave out of PB7. A second purpose behind this project is to illustrate some of the information given about the timers in Chapter Six.

The program consists of three parts - an assembly language subroutine that sets the registers in VIA-B to the appropriate values, a BASIC procedure PROCPULSE that uses the machine code to

program the VIA, and a main program. The main program first assembles the assembly language by calling PROCASMB and then reads in PL% and calls PROCPULSE to set the period and start the train of square waves coming out of PB7.

```
 10 DIM CODE% 100
 20 PROCASMB
 30 INPUT PL%
 40 PROCPULSE(PL%)
 50 GOTO 30
 60 STOP

 70 DEF PROCASMB
 80 AUXC%=&FE6B
 90 T1CLL%=&FE64
100 T1CH%=&FE65
110 IRQC%=&FE6E
120 P%=CODE%
130 FOR PASS=0 TO 3 STEP 3
140 [OPT PASS

150 LDA #&40
160 STA IRQC%
170 LDA AUXC%
180 ORA #&C0
190 STA AUXC%
200 STX T1CLL%
210 STY T1CH%
220 RTS
230 ]

240 NEXT PASS
250 ENDPROC

260 DEF PROCPULSE(PERIOD%)
270 X%=PERIOD% MOD 256
280 Y%=PERIOD% DIV 256
290 CALL CODE%
300 ENDPROC
```

Notice once again the different way of forming the machine code by calling PROCASMB which contains the definitions of all the labels used by the machine code. It is a good idea, however, to leave the DIM statement that reserves memory at the start of the program so that it is easy to find. The machine code subroutine first disables any interrupts the timer 1 might try to produce (lines 150 to 160). It then sets up timer 1 in the free running mode with PB7 enabled by storing the correct control bits in the auxiliary control register (line 170 to 190). Finally, it stores the X register into the low order latch and the Y register into the high order counter/latch which starts the counter off. The procedure PROCPULSE(PERIOD%) simply divides the value in PERIOD% into

a low byte and a high byte and stores them in X% and Y% respectively. When you run the program, the value that you give to PL% will determine the width of each pulse in the square wave in microseconds. Once a pulse train is started it may be altered by entering a different value of PL%. You might be surprised, however, to discover that the pulses carry on after the program has been stopped by pressing ESCAPE!

Conclusion

This second chapter on the BBC Micro's assembly language builds on the information presented in Chapter Seven. It contains four projects which between them cover many aspects of machine code programming. The BBC Micro's facility for combining BASIC and assembly language is an attractive feature which is explored via two of these practical examples. A useful philosophy for writing software on the BBC Micro is to use BASIC wherever possible and resort to assembly language when necessary to speed things up or to add commands to the system.