4. The BBC BASIC assembler

There are two main ways of writing ARM assembly language programs. One is to use a dedicated assembler. Such a program takes a text file containing ARM assembly language instructions, assembles it, and produces another file containing the equivalent machine code. These two files are called the source files and object files respectively.

An alternative approach is to use the assembler built-in to BBC BASIC. The ability to mix assembler with BASIC is a very useful feature of the language, and one that is relatively straightforward to use. For this reason, and because of the widespread availability of BBC BASIC, we describe how to use its built-in assembler. The examples of the next two chapters are also in the format expected by the BASIC assembler.

4.1 First principles

Two special 'statements' are used to enter and exit from the assembler. The open square bracket character, **[**, marks the start of assembly language source. Whenever this character is encountered where BASIC expects to see a statement like **PRINT** or an assignment, BASIC stops executing the program and starts to assemble ARM instructions into machine code. The end of the source is marked by the close square bracket, **]**. If this is read where BASIC is expecting to see an instruction to be assembled, it leaves assembler mode and starts executing the (BASIC) program again.

To see the effect of entering and leaving the assembler, type in this short program:

```
10 PRINT "Outside the assembler"
20 [ ;In the assembler
30 ]
40 PRINT "Outside the assembler"
```

If you RUN this, you should see something like the following:

```
Outside the assembler
00000000                    ;In the assembler
```

Between the two lines produced by the **PRINT** statements is one which the assembler printed. Unless you tell it not to, the assembler prints an assembly listing. We shall describe this in detail, but for now suffice is to say that it consists of three parts: an address (the eight zeros above, which may be different when *you* run the program), the machine code instruction in hex, and the source instruction being assembled.

In our example above, no instructions were assembled, so no machine code was listed. The only line of source was a comment. The semi-colon, `;`, introduces a line of comment to the assembler. This acts much as a **REM** in BASIC, and causes the text after it to be ignored. The comment was indented somewhat in the assembly listing, as the assembler leaves space for any machine code which might be produced.

The address printed at the start of the line is called the location counter. This tells us two things: where the assembler is placing assembled machine code instructions in memory, and what the value of the program counter (PC) will be when that instruction is fetched for execution, when the program finally comes to be run. Because we are not assembling any instructions, no code will be written to memory, so it doesn't matter what the location counter's value is. Normally, though, when you assemble code, you should set the location counter to some address where the assembler may store the machine code.

It is easy to set the location counter, as it is stored in the system integer variable P%. If you enter the immediate statement:

```
PRINT ~P%
```

you will see the same figures printed as at the start of the middle line above (except for leading zeros).

Let us now assemble some actual code. To do this, we need to reserve some storage for the object code. This is easily done using the form of **DIM** which reserves a given number of bytes. Type in the following:

```
10 DIM org 40
20 P% = org
30 [ ;A simple ARM program
40 MOV R0,#32
50.LOOP
60 SWI 0
70 ADD R0,R0,#1
80 CMP R0,#126
90 BNE LOOP
```

```
110 ]
```

It may seem strange using the variable **org** in the **DIM**, then using it only once to set P%, but we shall see the reason for this later. Note that the **DIM** statement returns an address which is guaranteed to be aligned on a word boundary, so there is no need to 'force' it to be a multiple of four.

As in the previous example, the first line of the assembler is a comment. The next seven lines though are actual assembly language instructions. In fact, you may recognise the program as being very similar to the example given at the end of Chapter One. It is a simple loop, which prints the character set from ASCII 32 (space) to ASCII 126 (tilde or **~**).

The lines which are indented by a space are ARM mnemonics, followed by their operands. When the assembler encounters these, it will convert the instruction into the appropriate four-byte machine code, and then store this at the current location counter - P%. Then P% will be increased by four for the next instruction.

Line 50, which starts with a **.**, is a label. A label is a variable which marks the current place in the program. When it encounters a label, the assembler stores the current value of the location counter into the variable, so that this place in the program may be accessed in a later instruction. In this case the **BNE** at line 90 uses the label **LOOP** to branch back to the **SWI** instruction. Any numeric variable may be used as a label, but by convention floating point variables (without **%** sign) are used. Every label in a program should have a unique name. This isn't hard to achieve as BBC BASIC allows very long names.

If you RUN the program, another assembly listing will be produced, this time looking like:

```
000167C8                          ;A simple ARM program
000167C8  E3A00020                MOV  R0,#32
000167CC           .LOOP
000167CC  EF000000                SWI  0
000167D0  E2800001                ADD  R0,R0,#1
000167D4  E350007E                CMP  R0,#126
000167D8  1AFFFFFB                BNE  LOOP
000167DC  E1A0F00E                MOV  R15,R14
```

The first column is the location counter, and in the listing above, we can see that the first machine instruction was placed at address &167C8. Next is the hex representation of that instruction, i.e. the code which will actually be fetched and executed by the processor. You can verify that this is what was

```
PRINT ~!&167C8
```

(The address should be whatever was printed when you ran the program.) You will see **E3A00020**, as in the listing.

In the third column is the label for that instruction, or spaces if there isn't one, followed by the rest of the line, as typed by you.

To see the fruits of you labours, type the command:

```
CALL org
```

The ASCII character set of the machine will be printed. The **CALL** statement takes the address of a machine code routine, which it then executes. The machine code is called as if a **BL** instruction had been used with the address given as the operand. Thus to return to BASIC, the return address in R14 is transferred to R15 (the PC) as in the example above.

We will have more to say about using **CALL** (and the associated function **USR**) later in this chapter.

4.2 Passes and assembly options

Frequently, a label is defined *after* the place (or places) in the program from which it is used. For example, a forward branch might have this form:

```
100 CMP R0,#10
110 BNE notTen
120 ; some instructions
130 ; ...
140.notTen
```

In this example, the label **notTen** has not been encountered when the instruction at line 110 is assembled. It is not defined until line 140. If this or a similar sequence of instructions were encountered while assembling a program using the method we have already described, a message of the type '**Unknown or missing variable**' would be produced.

We clearly need to be able to assemble programs with forward references, and the BBC BASIC assembler adopts the same approach to the problem as many others: it makes two scans, or passes, over the program. The first one is to enable all the labels to be defined, and the second pass assembles the code proper.

In the BASIC assembler we tell it to suppress 'errors' during the first pass by using the **OPT** directive. A directive (or pseudo-op as they are also called) is instructs the assembler to perform some action. It is used in the same place as a proper instruction.

The **`OPT`** directive is followed by a number, and is usually placed immediately after the **`[`**, but can in fact be used anywhere in the source code. The number is interpreted as a three-bit value. Bit zero controls the assembly listing; bit one controls whether '**`Unknown or missing variable`**' type errors are suppressed or not, and bit two controls something called offset assembly, which we shall come to later. The eight possible values may be summarised as below:

Value	Offset assembly	Errors given	Listing
0	No	No	No
1	No	No	Yes
2	No	Yes	No
3	No	Yes	Yes
4	Yes	No	No
5	Yes	No	Yes
6	Yes	Yes	No
7	Yes	Yes	Yes

If you don't use **`OPT`** at all, the default state is 3 - offset assembly is not used; errors are not suppressed, and a listing is given.

The most obvious way to obtain the two passes in BASIC is to enclose the whole of the source code in a **`FOR...NEXT`** loop which performs two iterations. Often, the code is enclosed in a procedure so that the **`FOR`** and **`NEXT`** of the loop can be close together. The desired assembly option can then be passed as a parameter to the procedure.

Below is an example following the outline described above. The program contains a forward reference to illustrate the use of two passes. The listing is produced in the second pass (as it always should be if there are two passes), so the values given to **`OPT`** are 0 and 3 respectively.

```
1000  REM Example of two-pass assembly
1010  DIM org 100
1030  FOR pass=0 TO 3 STEP 3
1040     PROCasm(pass,org)
1050  NEXT pass
1060  END
1070
2000  DEF PROCasm(pass,org)
2010  P%=org
2020  [ OPT pass
2030 .loop
```

```
2055  MOVEQ  R15, R14
2060  BCC    even
2070  SWI    256+ASC"O" ;Print O
2080  B      loop
2090 .even
2100  SWI    256+ASC"E" ;Print E
2110  B      loop
2120  ]
2130  ENDPROC
```

The most important thing to notice is that the P% variable is set at the start of each pass. If you only set it at the start of the first pass, by the end of the first pass it will contain the address of the end of the code, and be incorrect for the start of the second pass.

The program is fairly trivial. It reads characters from the keyboard and prints **E** if the character has an even ASCII code, or **O** if not. This repeats until an ASCII character 0 or 1 (**CTRL @** or **CTRL A**) is typed. The branch to **even** at line 2060 is a forward one; during the first pass the fact that **even** hasn't been defined yet is ignored, then during the second pass, its value - set in line 2090 - is used to address the following **SWI**.

In fact, because of the ARM's conditional instructions, this program could have been written without any forward branches, by making lines 2070 and 2080:

```
2070 SWICS   256+"O";Print O
2080 SWICC   256+"E";Print E
```

and deleting lines 2090 and 2100. It's always hard thinking of simple but useful ways of illustrating specific points.

You should be aware of what actually happens when a forward reference is encountered in the first pass. Instead of giving an error, the assembler uses the current value of the location counter. In line 2060 above, for example, the value returned for the undefined variable is the address of the **BCC** instruction itself. If only the first pass was performed and you tried to call the code, an infinite loop would be caused by this branch to itself. It is important, therefore, that you always set bit 1 of the **OPT** value during the second pass, to enable errors to be reported, even if you don't bother about the listing.

4.3 Program style and readability

Any experienced assembler language programmer reading this chapter will probably be thinking by now, 'What a load of rubbish.' The examples presented so far are not, it must be admitted, models of clarity and elegance

This was deliberate; it gives us a chance to point out how they are deficient and how these deficiencies may be remedied.

As we learned in Chapter One, assembly language is about as far from the ideal as a normal human programmer is likely to venture. When writing an assembly language program, you should therefore give your reader as much help as possible in understanding what is going on. Bear in mind that the reader may well be you, so it's not just a question of helping others.

Comments are the first line of attack. As we have seen, comments may appear as the first (and only) thing on the line, or after the label, or after the label and instruction. A comment may not come between the label and instruction, as the instruction will be ignored, being taken as part of the comment. We have used **`;`** to introduce comments, as this is the same character that many other assemblers use. The BBC BASIC assembler also allows you to use **`\`** and **`REM`**. Whatever you use, you should be consistent.

A comment should be meaningful. The comments at the start of a routine should explain what it does, what the entry and exit conditions are, and what the side-effects are. Entry and exit conditions usually pertain to the contents of the ARM registers when the routine is called and when it returns, but may also include the state of specific memory locations at these points too.

The side-effects of a routine are similar to saying what it does, but are more concerned with actions which might affect the operation of other routines. A common side-effect is that of corrupting certain registers. Another example might be a routine which prints a character on the screen as its main action, but corrupts a pointer in RAM as a side-effect. By documenting these things in comments, you make it much easier to track down bugs, and to interface your routines to each other.

From the remarks above, it can be seen that the routines presented above are poorly served with comments, and this makes it more difficult to understand them, even though they are quite short.

Another important way of increasing readability is to use names instead of numbers. Examples where this clouded the issue in earlier programs are:

```
  80 CMP   R0,#126
2040 SWI   4   ;Read a char
```

Admittedly the second example has a comment which gives the reader some idea, but the first one is totally meaningless to many people. In the BBC power of the BASIC expression evaluator is available. The most obvious way to utilise this is to use variables as names for 'magic' numbers. For example, in the character program, we could have included an assignment

```
17 maxChar = ASC"~"
```

and made line 80:

```
80 CMP   R0,  #maxChar
```

In the first place we are told in line 17 what the last character to be printed is, and secondly the name **maxChar** gives a much better indication in line 80 of what we are comparing.

In the second example, the operand to **SWI** was a code number telling the operating system what routine was desired. This was the read character routine which goes by the name of **ReadC**. So the program can be made more readable by adding the line:

```
1005 ReadC = 4 : REM SWI to read a char into R0
```

and changing line 2040 to:

```
2040 SWI ReadC
```

which makes the comment redundant. The **SWI 256** is another instruction that can be improved. This particular **SWI** has an operand of the form 256+ch, where ch is the code of character to be printed. It is given the name **WriteI** (for write immediate) in Acorn documentation, so we could have an assignment like this:

```
1007 WriteI=&100 : REM SWI to write char in LSB of operand
```

and change the instructions to have the form:

```
2070 SWI WriteI+ASC"O"
```

The importance of giving names to things like **SWI** codes and immediate operands cannot be understated. A similarly useful tool is naming registers. So far we have used the denotations R0, R1 etc. to stand for registers. In addition to these standard names, the assembler accepts an expression which evaluates to a register number. An obvious application of this is to give names to important registers such as the SP by assigning the appropriate register number to a variable. For example, if we had these three assignments in the BASIC part of a program:

```
10 link = 14    : REM BL link register
```

then instructions could take the form:

```
1000 MOVCS   pc,link           ;Return if too big
1210 STM     (sp)!,{acc,link}  ;Save result and return addr
2300 LDM     (sp)!,{acc,pc}    ;Return and restore
```

There are two things to note about these examples. The first is that we use the name **pc** even though this wan't one of the variables we defined above. The BASIC assembler recognises it automatically and substitutes **R15**. The second is that **sp** has brackets around it. These are required because the **!** sign is used by BASIC to denote 'indirection'. To stop BASIC from getting confused you have to put brackets around any variable name used before a **!** in **STM** and **LDM** instructions. You don't need the brackets if a register name, such as **R13**, is used.

In the examples in this book, we use the names **pc**, **link** and **sp** as shown above, and we also use names for other registers where possible. There is no confusion between register numbers and immediate operands, of course, because the latter are preceded by a **#** sign. So:

```
4310 MOV  acc,sp
```

copies the contents of register **sp** (i.e. R13) to register **acc**, whereas:

```
4310 MOV  acc,#sp
```

would load the immediate value 13 into the **acc** register.

The final point about style concerns layout. The assembler doesn't impose too many constraints on the way in which you lay out the program. As we have already mentioned, labels must come immediately after the line number, and comments must appear as the last (and possibly only) thing on the line. However, it is advisable to line up the columns where possible. This makes it easier to follows comments if nothing else. Comments describing individual routines or the whole program can be placed at the start of the line. This also applies to comments for instructions which wouldn't fit on the line easily (e.g. after an **STR** with a long register list).

Line spacing can also be important. It is a good idea to leave blank lines between routines to separate them.

A typical format would be: ten spaces for the label, 25 for the instruction, and the rest of the line for the comment. For example:

```
1230          MOV  count, #15      ;Init the counter
1240.loop     SWI  writeI+ASC"*"   ;Print an asterisk
1250          SUB  count, #1       ;Next count
```

Another minor point of style in the listing above is the spacing of operands. It is easier to separate operands if there is at least one space character between them.

By way of a simple contrast, we list the very first program of this chapter, trying to practise that which we have just been preaching.

```
 10 DIM org 40
 20 P% = org
 30 outReg = 0 : REM Register use by WriteC. Preserved
 40 link =   14 : REM BL link register
 50 pc   =   15 : REM ARM program counter
 60 WriteC = 0 : REM SWI code to print char in R0
 70 firstChar = ASC" "
 80 lastChar = ASC"~"
 90 [
100 ;A simple ARM program. This prints the ASCII
110 ;character set from 'firstChar' to 'lastChar'
120 MOV outReg, #firstChar       ;Init the output char
130 .loop
140 SWI WriteC                   ;Display the char.
150 ADD outReg, outReg, #1       ;Increment the character
160 CMP outReg, #lastChar        ;Finished?
170 BNE loop                     ;No so loop
180 MOV pc, link                 ;else return
190 ]
```

Because the line numbers are not particularly relevant to the assembly language, listings in subsequent chapters don't use them.

Finally, a note on cases. The assembler is totally case-insensitive. Mnemonics and 'R's in R0 etc. may be in upper or lower case or any combination thereof. Labels, however, must obey the rules of the rest of BASIC, which state that variable names are case dependent. Thus **`LOOP`** is a different label from **`loop`**, and **`Loop`** is again different. The convention used in this book is upper case for mnemonics and R0-type designations, and a mixture of upper and lower case (e.g. **`strPtr`**) for labels and other names. As usual, the method you adopt is up to you, consistency being the important thing.

4.4 Offset assembly

The facility of offset assembly was mentioned above. It is enabled by bit 2 of the **`OPT`** expression. When offset assembly is used, the code is still assembled to execute at the address in the P% variable. However, it is not stored there. Instead, it is placed at the address held in the O% variable.

Motivation for offset assembly can be explained as follows. Suppose you are writing a program which will load and execute at address &8000. Now, using **DIM** to obtain code space, you find you can only obtain addresses from about &8700 up. You could set P% explicitly to &8000, but as this is where the BASIC interpreter's workspace lies, you will be in dire danger of overwriting it. This invariably leads to unpleasantness.

The solution to this dilemma is to still assign P% to &8000, as if it is going to execute there, but use offset assembly to store the machine code at some conveniently **DIM**med location. Here is the outline of a program using offset assembly, without line numbers.

```
org = &1000    : REM Where we want the code to run
DIM code 500   : REM Where the code is assembled to
code$= "  "+STR$~code+"  "
org$  = "  "+STR$~org+"  "

FOR pass=4 TO 6 STEP 2
    P%=org
    O%=code
    [   OPT pass
    ;The lines of the program come here
    ;...
    ;...
    ]
NEXT pass
OSCLI "SAVE PROG "+code$+STR$~O%+org$+org$
```

Both P% and O% are set at the start of each pass. Because bit 2 of the pass looping value is set, offset assembly will take place and the assembled code will actually be stored from the address in **code**. Once assembled, the code is saved with the appropriate load and execution addresses, and could be activated using a ***** command.

When using offset assembly, you shouldn't use **CALL** to test the machine code, as it will be assembled to execute at an address different from the one in which it is stored.

Having illustrated the use of offset assembly, we will now say that you should rarely have to use it. As the next chapter tries to show, it is quite easy to write ARM programs which operate correctly at any load address. Such programs are called position independent, and there are a few simple rules you have to obey in order to give your programs this desirable property.

Overleaf is a version of the outline listing above, assuming that the program between the **[** and **]** statements is position independent.

```
org = &1000    : REM Where we want the code to run
DIM code 500   : REM Where the code is assembled to
code$= " "+STR$~code+" "
org$ = " "+STR$~org+" "

FOR pass=0 TO 2 STEP 2
    P%=code
    [   OPT pass
    ;The lines of the program come here
    ;...
    ;...
    ]
NEXT pass
OSCLI "SAVE PROG "+code$+STR$~P%+org$+org$
```

The differences are that O% doesn't get a mention, and the values for the **OPT** directive are 0 and 2 respectively. The value of **org** is used only to set the load and execution addresses in the **SAVE** command.

In the light of the above, you may be wondering why offset assembly is allowed at all. It is really a carry-over from early versions of BBC BASIC, running on the BBC Micro. These incorporated assemblers for the old-fashioned 6502 processor that the BBC machine used. It is virtually impossible to write position independent code for the 6502, so offset assembly was very useful, especially for those writing 'paged ROM' software.

There is still one valid use for offset assembly, and that is in defining areas of memory for data storage. We shall describe this in the next section.

4.5 Defining space

Most programs need some associated data storage area. This might be used for constant data, such as tables of commands, or for holding items such as character strings (filenames etc.) or arrays of numbers which won't fit in the ARM's registers. There are four assembler directives which enable you to reserve space in the program for data. They are called:

EQUB - Reserve a byte
EQUW - Reserve a word (two bytes)
EQUD - Reserve a double word (four bytes)
EQUS - Reserve a string (0 to 255 bytes)

The reference to a 'word' being two bytes may be confusing in the light of our earlier claims that the ARM has a 32-bit or four-byte word length. Like offset assembly, the notation has its origins in the murky history of BBC BASIC. It's probably best to forget what the letters stand for, and just

Each of the directives is followed by an expression, an integer in the first three cases, and a string in the last. The assembler embeds the appropriate number of bytes into the machine code, and increments P% (and if necessary O%) by the right amount.

Here is an example of using EQUB. Suppose we are writing a VDU driver where control characters (in the range 0..31) may be followed by 1 or more 'parameter' bytes. We would require a table which, for each code, gave us the parameter count. This could be set-up using **EQUB** as below:

```
.parmCount
  ; This table holds the number of parameters bytes
  ; for the control codes in the range 0..31
 EQUB 0          ;NUL.   Zero parameters
 EQUB 1          ;To Printer.   One parm.
 EQUB 0          ;Printer on.   Zero parms.
 EQUB 0          ;Printer off.   Zero parms
 ....        ....
 ....        ....
 EQUB 4          ;Graf origin.   Four bytes
 EQUB 0          ;Home.   No parameters
 EQUB 2          ;Tab(x,y).   Two parms.
  ; End of parmCount table
  ;
```

Each of the **EQUB**s embeds the count byte given as the operand into the machine code. Then P% is moved on by one. In the case of **EQUW**, the two LSBs of the argument are stored (lowest byte first) and P% incremented by two. For **EQUD**, all four bytes of the integer operand are stored in standard ARM order (lowest byte to highest byte) and P% is incremented by 4.

There are a couple of points to note. We are storing data in the object code, not instructions. Therefore you have to ensure that the program will not try to 'execute' the bytes embedded by **EQUB** etc. For example, if there was code immediately before the label **parmCount** above, there should be no way for the code to 'drop through' into the data area. The result of executing data is unpredictable, but invariably nasty. For this reason it is wise to separate 'code' and 'data' sections of the program as much as possible.

Secondly, in the example above we embedded 32 bytes, which is a multiple of four. So assuming that P% started off on a word boundary, it would end up on one after the list of **EQUB**s. This means that if we started to assemble instructions after the table, there would be no problem. Suppose, however, that we actually embedded 33 bytes. After this, P% would no longer lie on a word boundary, and if we wanted to start to assemble instructions after the table, some corrective action would have to be taken. The **ALIGN** directive is used to perform this.

You can use **ALIGN** anywhere that an instruction can be used. Its effect is to force P% and O% to move on to the next word boundary. If they already are word-aligned, they are not altered. Otherwise, they are incremented by 1, 2 or 3 as appropriate, so that the new values are multiples of four.

The string directive, **EQUS**, enables us to embed a sequence of bytes represented by a BBC BASIC string into memory. For example, suppose we had a table of command names to embed in a program. Each name is terminated by having the top bit of its last letter set. We could set-up such a table like this:

```
.commands
 ; Table of commands.  Each command has the top bit
 ; of its last char.  set.  The table is terminated
 ; by a zero byte.
         EQUS  "ACCES"+CHR$(ASC"S"+&80)
         EQUS  "BACKU"+CHR$(ASC"P"+&80)
         EQUS  "COMPAC"+CHR$(ASC"T"+&80)
 ....
 ....
         EQUS  "TITL"+CHR$(ASC"E"+&80)
         EQUS  "WIP"+CHR$(ASC"E"+&80)
 ;
         EQUB  0
         ALIGN
 ; End of command table
```

Note the use of the **ALIGN** directive at the end of the table.

The examples presented so far have shown initialised memory being set-up, that is memory which contains read-only data for use by the program. The need to reserve space for initialised memory also arises. For example, say a program uses an operating system routine to load a file. This routine might require a parameter block in memory, which contains all of the data necessary to perform the load. To reserve space for the parameter block, a sequence of **EQUD**s could be used:

```
.fileBlock
 ; This is the parameter block for the file load operation
 ;
.namePtr     EQUD  0    ; Pointer to the filename
.loadAddr    EQUD  0    ; The load address of the file
.execAddr    EQUD  0    ; The execution address
.attributes  EQUD  0    ; The file's attributes
 ;
```

```
;
.doLoad
 ; This routine sets up the file block and loads the file
 ; It assumes the name is on the stack
;
        STR   sp, namePtr      ; Save the name pointer
        MOV   R0, #&1000       ; Get 4K for the file
        SWI   getMemory        ; Returns pointer in R1
        STR   R1, loadAddr     ; Save the address
 ; Get the address of the parameter block in R0
        ADR   R0, fileBlock
        MOV   R1, #loadCode    ; Command code for SWI
        SWI   file             ; Do the load
...
```

The **EQUD**s to set up the parameter block do not have meaningful operands. This is because they are only being used to reserve memory which will be later filled in by the program. Each **EQUD** has a label so that the address of the four-byte area reserved can be accessed by name.

Following the set-up directives is some code which would use such a parameter block. It uses some hypothetical **SWI** calls to access operating system routines, but you can assume that similar facilities will be available on any ARM system you are likely to use. Some of the instructions may not be very clear at this stage, but you might like to come back and re-examine the example having read the next two chapters.

The **ADR** instruction isn't an ARM mnemonic, but is a directive used to load a register with an address. The first operand is the register in which the address is to be stored. The second operand is the address to be loaded. The directive is converted into one of these instructions:

```
ADD  reg,pc,#addr-(P%+8)  or
SUB  reg,pc,#(P%+8)-addr
```

The first form is used if the **addr** is after the directive in the program, and the second form is used if **addr** appears earlier, as in the example above. The idea of using the directive is to obtain the required address in a position-independent manner, i.e. in a way which will work no matter where the program is executing. We say more about position-independence in Chapter Five.

To reserve space for items larger than double words, you can use **EQUS**. The operand should be a string whose length is the required number of bytes. The BASIC **STRING$** function provides a good way of obtaining such a string. The example below reserves two 32-byte areas:

```
path1      EQUS  STRING$(32,CHR$0)   ;32 bytes for filename1
```

By making the second operand of **STRING$ CHR$0**, the bytes embedded will by ASCII 0. As this is an illegal filename character, it can be used to guard against the program omitting to initialise the strings before using them.

In each of the examples above, the directives were used to reserve storage inside the machine code of the assembly language program. Another use is in defining labels as offsets from the start of some work area. For example, we may use register R10 as a pointer to a block of memory where we store important, frequently accessed data (but which doesn't fit in the registers). The **LDR** group of instructions would be used to access the data, using immediate offsets. Suppose the structure of the data is thus:

Offset	Item	Bytes
00	Program pointer	4
04	Top of program	4
08	Start of variables	4
12	End of variables	4
16	Character count	4
20	ON ERROR pointer	4
24	String length	1
25	List option	1
26	OPT value	1
27	....	1

The data items are the sort of thing that a BASIC interpreter would store in its data area. Notice that all of the word-length items are stored in one group so that they remain word aligned. One way of assigning the offsets in the BBC BASIC assembler would be with assignments:

```
page = 0
top = page+4
lomem = top+4
vartop = lomem+4
count = vartop+4
errptr = count+4
strlen = errptr+4
listo = strlen+1
opt = listo+1
....= opt+1
```

This method has a couple of drawbacks. First, to find out how many bytes have been allocated to a given item, you have to look at two lines and perform a subtraction. Similarly, to insert a new item, two lines have to be amended. Even worse, the actual values assigned to the labels can't be discovered without printing them individually - they don't appear in the

To overcome these drawbacks, we use the assembler's offset assembly ability. The method is illustrated below:

```
 DIM org 1000 : REM Enough for code and data declarations
 FOR pass=0 TO 2 STEP 2
   PROCdeclarations(org,pass)
   PROCasm(org,pass)
 NEXT
 END

 DEF PROCdeclarations(org,pass)
 P%=0 : O%=org
 [          OPT pass + 4
.page       EQUD 0
.top        EQUD 0
.lomem      EQUD 0
.vartop     EQUD 0
.count      EQUD 0
.errptr     EQUD 0
.strlen     EQUB 0
.listo      EQUB 0
.opt        EQUB 0
...         EQUB 0
 ]
 ENDPROC

 DEF PROCasm(org,pass)
 P%=org
 [          OPT pass
 ....
```

The procedure **`PROCdeclarations`** contains the **`EQU`** directives which define the space for each item. By adding 4 to the **`pass`** value, offset assembly is used. Setting P% to zero means that the first label, **`page`**, will have the value zero; **`top`** will be set to 4, and so on. O% is set to **`org`** originally so that the bytes assembled by the directives will be stored in the same place as the program code. As **`PROCasm`** is called after **`PROCdeclarations`**, the code will overwrite the zero bytes.

When this technique is used, the addresses of the labels appear in the assembly listing (using **`FOR pass=0 TO 3 STEP 3`**); it is easier to see how many bytes each item occupies (by seeing which directive is used), and adding a new item is simply a matter of inserting the appropriate line.

4.6 Macros and conditional assembly

Sometimes, a sequence of instructions occurs frequently throughout a data, a two byte one and two single byte items. To set up the table, we could have a list of 32 instruction sequences, each of the form:

```
B       address
EQUW    data1
EQUB    data2
EQUB    data3
```

Typing in these four lines 32 times might prove a little laborious, and possibly error-prone.

Many dedicated assemblers provide a facility called *macros* to make life a little easier in situations like the one described above. A macro can be viewed as a template which is defined just once for the sequence. It is then 'invoked' using a single-line instruction to generate all of the instructions. The BBC BASIC assembler implements macros in a novel way, using the user-defined function keyword, **FN**.

If the assembler encounters an **FN** where it is expecting to find a mnemonic, the function is called as usual, but any result it returns is disregarded. Once you are in **FN**, you have the whole power of BASIC available. This includes the assembler, so you can start to assemble code within the **FN**. This will be added to the machine code which was already assembled in the main program.

To clarify the situation, we will implement a macro to make a table entry of the type described above.

```
10000  DEF  FNtable(address,data1,data2,data3)
10010 [          OPT          pass
10020            B            address
10030            EQUW         data1
10040            EQUB         data2
10050            EQUB         data3
10060 ]
10070 =""
```

As you can see, the 'body' of the macro is very similar to the list of instructions given earlier. The macro is defined in a separate part of the program, outside of the main assembly language section (hence the very high line number). It assumes that the looping variable used to control the passes of the assembler is called **pass**, but this could have been given as an extra parameter.

To call the macro, an **FNtable** call is used, one for each entry in the table. A typical sequence might look like this:

```
1240      FNtable(list,1,2,3)
1250      FNtable(delete,2,3,4)
1260      FNtable(renumber,1,1,2)
1270 ....
```

When the **FN** is reached, the assembler calls the function using the parameters supplied. The effect of this is to assemble the instructions and then return. The return value **""** is disregarded. The net effect of all this is to assemble four lines of code with only one instruction.

Another use of macros is to 'invent' new instructions. For example, instead of always having to remember which type of stack you are using (full, empty, ascending or descending), you could define a macro which pushes a given range of registers. By always using the macro, you also avoid the pitfuls of forgetting the **!** to denote write-back.

The routine below is a macro to push a range of registers onto a full, descending stack, using R13.

```
10000  DEF  FNpush(first,last)
10010 [          OPT      pass
10020            STMFD    R13!,{first-last}
10030  ]
10040  =""
```

Another advantage of using a macro to implement a common operation like push is that it can be changed just the once to affect all of the places where it is used. Suppose, for example, that you have to change your program to use an empty stack, so that it is compatible with someone else's program. By simply changing line 10020 above, every push operation will use the new type of stack.

There are situations where you might want to be able to assemble different versions of a program, depending on some condition. This is where conditional assembly comes in.

Using conditional assembly is a bit like **IF…** type operations in BASIC, and that is exactly what we use to implement it in the BBC BASIC assembler. The listing overleaf shows how the macro **FNpush** could be written to cope with either a full or empty stack.

The variable **fullStack** is a global which is defined at the start of the program to **TRUE** if we are assembling for a full stack and **FALSE** if we want an empty one.

```
10000  DEF  FNpush(first,last)
10010  IF  fullStack  THEN
10020  [        OPT    pass
10030           STMFD  R13!,{first,last}
10040  ]
10050  ELSE
10060  [        OPT    pass
10070           STMED  R13!,{first,last}
10080  ]
10090  ENDIF
10100  =""
```

Most features of dedicated assemblers can be implemented using a combination of BBC BASIC and the BASIC assembler. For example, some assemblers have a **WHILE** loop directive which can be used to assemble repeated instructions which would be tedious (and maybe error prone) to type. The BASIC **WHILE** (or **FOR** or **REPEAT**) loops can be used to similar effect. The macro below implements a 'fill memory' function. It takes a count and a byte value, and stores the given number of bytes in the object code.

```
10000  DEF  FNfill(byte,count)
10010  IF  count=0  THEN  =""
10020  LOCAL  i
10030  FOR  i=1  TO  count
10040  [        OPT    pass
10050           EQUB   byte
10060  ]
10070  NEXT  i
10080  [        OPT    pass
10090           ALIGN
10100  ]
10110  =""
```

Although we could have quite easily 'poked' the bytes into the code without using **EQUB**, the method we use guarantees that the effect of **FNfill** shows up on the assembly listing. It is important to keep what is printed in the listing consistent with what is actually going on, otherwise it becomes difficult to debug programs from the listing.

4.7 Calling machine code

Many of the example programs in the next two chapters are accompanied by short test routines. These are called from BASIC after the code has been assembled. To help you understand these programs, we describe the **CALL** statement and the **USR** function in this section.

The format of the **CALL** statement is:

```
CALL  <address>,[<parameters>]
```

The **<address>** is the location of the machine code that you want to execute. The optional **<parameters>** are BASIC variables, separated by commas.

When the called routine starts to execute, it finds its registers set-up by BASIC. We won't describe the full extent of what BASIC provides for the machine code program, but instead point out the most useful features. First, R0 to R7 are set-up from the values of eight of the system integer variables (A% etc). So, R0 holds the contents of A%, R1 is initialised from B%, and so on, up to R7 which has been loaded from H%.

R9 points to the list of parameters, and R10 contains the number of parameters that followed the **<address>**. This may be zero, as the parameters are optional. The data to which R9 points is a list of pairs of words. There is one word-pair for each parameter.

The first word contains the address of the variable, i.e. where its value is stored in memory. The second word holds the type of the variable. From this, the meaning of the data addressed by the first word can be determined. The main types are as follows:

<table>
<tr><th>Type</th><th>Meaning</th></tr>
<tr><td>0</td><td>Single byte, e.g. ?addr</td></tr>
<tr><td>4</td><td>Integer, e.g. fred%, !fred, fred%(10)</td></tr>
<tr><td>5</td><td>Real, e.g. fred, |fred, fred(10)</td></tr>
<tr><td>128</td><td>String, e.g. fred$, fred$(10)</td></tr>
<tr><td>129</td><td>String, e.g. $fred</td></tr>
</table>

The type-128 string value consists of five bytes. The first four are the address of the contents of the string (i.e. the characters), and the fifth byte is the string's length. A type-129 string is a sequence of bytes terminated by a carriage-return character, and the address word points to the first character of the string. Note that the data values can have any alignment, so to access them you must use a general-alignment routine of the type presented in Chapter Six.

It is important to realise that the word-pairs describing parameters are in reverse order. This means that the word pointed to by R9 is the address of the *last* parameter's value. The next word is the type of the last parameter, and the one after that is the address of the penultimate parameter, and so

The final registers of importance are R13, which is BASIC's stack pointer, and R14, the link register. BASIC's (full, descending) stack may be used by the called routine, as long as the stack pointer ends up with the same value it had on entry to the routine. R14 contains the return address, as if the routine had been called with a **BL** instruction. This enables the routine to return to BASIC with an instruction such as:

```
MOV     pc,R14
```

The **USR** function is used in a similar way to **CALL**. There are two important differences. First, it cannot take parameters, so a typical use would be:

```
PRINT  ~USR  address
```

Second, it returns a result. This result is the contents of R0 on return from the machine code. There are several examples of **USR** in the next two chapters.

4.8 Summary

In this chapter we have looked at one of the assemblers available to the ARM programmer. Because it is part of ARM BASIC, it is straightforward to use. Although the assembler itself is not overflowing with features, the fact that assembler and BASIC can be freely combined has many advantages. It is relatively easy to implement many of the features of 'professional' assemblers, such as macros and conditional assembly.

Finally we looked briefly at the **CALL** and **USR** keywords in BASIC, in preparation for their use in later chapters.