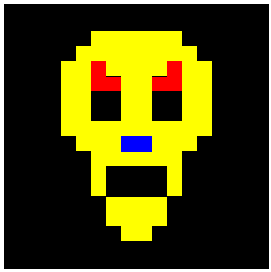

11

PROGRAM STRUCTURE

The aim of this chapter is to help you write a large assembler program. It gives several tips how to produce structured and readable code which can be debugged easily.

11.1 Where to start

There are two entirely different approaches which can be used when producing a structured program. The first is to work out what the program is going to do and what it is going to look like before you start writing any code. Then you can start by writing the main loop which, for a game, could look something like this:

```
.enter
   JSR initialise
.restart
   JSR setupscreen
.main
   JSR plotshapes
   JSR checkcollisions
   JSR keyboardscan
   JSR updatecoordinates
   JSR checkifdead
```

```
BNE main loop
DEC lives
BNE restart
RTS
```

Although, at this stage, the subroutines have not been defined, it is obvious from their names what they are meant to do. The next task is to write these subroutines, again breaking them down into simpler routines if this is possible.

Since the main loop is the top level in the overall structure, this method is known as 'top-down' approach. Its advantage is that you know what you are aiming at right from the start. You shouldn't need any 'fixes' in the subroutines to make up for the fact that when you wrote them you didn't make them apply to all cases eventually required.

The alternative method is to start by writing the individual routines and then to add the code which joins them together into the final program. The advantage of this method is that all routines can be tested individually before very much effort has gone into the program. Consider how much time would be wasted if you used the first method and completed the whole program except for one routine which proved impossible to write in such a way that it performed satisfactorily.

In practice most people use a mixture of the two methods, so that any demanding routines are written first; then the top-level is written, followed by the rest of the program.

11.2 Self-documenting code

Long, directly referential variable names are a good idea when writing a program as they make the program more intelligble, e.g. 'JSR updatecoordinates' is better than 'JSR label2'. These may make the source code longer, but the problem can be overcome by splitting up the code into separate sections and compiling these individually as described in chapter 10 (Large assembler programs). Note that the fewer JMP's there are, the easier it is to follow the flow of control. Hence, using macros rather than subroutines also helps to increase the clarity of the code.

11.3 Parameters

Parameter passing is also recommended in assembler, for precisely the same reasons as it is in BASIC, viz. It enables a single block of code to be used for more than one purpose. However, the method of achieving this is somewhat different. Parameters are usually passed in the three registers A, X and Y, or alternatively the X and Y registers are used to specify an address of an information block, and the A register then holds some other information. A third method is to pass parameters in specified locations, so enabling an arbitrary number of parameters to be passed. This method doesn't support nesting or recursion, however.

The CALL statement

This statement, which is used to transfer control from a BASIC program to a machine code program, can also pass parameters. When it is used, the bottom bytes of the BASIC variables A%, X% and Y% are transferred to the A, X and Y registers respectively. Also the lowest bit of C% is transferred to the carry flag.

Control is passed to the address given after the CALL statement and any parameters following this address are put into the parameter block starting at &600. The parameter block is of the following format:

&600	Number of parameters
&601	Parameter address (low byte)
&602	Parameter address (high byte)
&603	Parameter type
&604	Parameter address (low byte)

The parameter types are as follows:

Type 0	8-bit byte
Type 4	32-bit word
Type 5	40-bit floating point string
Type 128	ATOM string
Type 129	Microsoft string

In the case of a string parameter the address given points to a 'string information block', which contains the following:

Start address of the string
Number of bytes allocated

Current length of string

An example of a CALL statement is

```
CALL enter,fred,A$
```

This would cause the machine code from ' enterto be executed and the parameters ' fredand ' ASwould be described in the parameter block as follows:

```
 ------------------------------------------------
| 02 | C7 | 0E | 05 | DO | 0E | 81 | ?? | ?? | ?? |
 ------------------------------------------------
 0600 0601 0602 0603 0604 0605 0606 0607 0608 0609
```

The USR function
Another way of transferring control to a machine code routine is via the USR function. The differences between CALL and USR are that USR returns a result, a four-byte number consisting of the Status, Y, X and A registers (most significant byte to least significant byte), and takes only one parameter which is the address to which control is transferred.

11.4 Size of routines
All routines, whether in BASIC or assembler, should be as small as possible, so that they can be seen easily in their entirety. This is another real help when debugging. Debugging is often overlooked when estimating the time needed to write a program, and yet it is probably true to say that at least 50% of the time taken to write a program is taken up with debugging.

11.5 Conditional assembly as an aid to debugging
Conditional assembly can be used to insert extra instructions which print out intermediate values during debugging: these statments can be removed when the program is finally assembled. To do this a logical variable (' flag'in the following example) is given the value FALSE during debugging and TRUE otherwise. In the following example, if ' flagis FALSE a routine to print the value of the accumulator in hexadecimal notation is assembled, and calls to this routine is inserted at two relevant points in the test program.

```
 10 REM print hex digits
 20 DIM code 100
 30 oswrch = &FFEE
 40 FOR pass = 0 TO 3 STEP 3
 50 P% = code
 60[OPT pass
 70.enter CLC : ADC #&40 : ]
 80 IF flag = FALSE [OPT pass : JSR debug : ]
 90[OPT pass
100   BEQ exit : SBC #&10 : ]
110 IF flag = FALSE [OPT pass : JSR debug : ]
120 [OPT pass
130.exit RTS : ]
140 IF flag THEN 360
150[OPT pass
160 # print hex digits
170.print
180   AND #&0F       Get bottom four bits
190   CMP #&0A       if less then 10 then miss
200   BCC P% + 4     the next instruction
210   ADC #&06       Add 7 (6 + carry)
220   ADC #ASC"0 "   ADD ADC (0)
230   JMP oswrch     Write the character
240 # print A in hex
250.debug
260   PHA
270   PHA
280   LSR A          Exchange top four bits
290   LSR A          for bottom four bits
300   LSR A
310   LSR A
320   JSR print      Print out first hex character
330   PLA
340   JSR print      Print second hex character
350   PLA            Restore original value of A
360   RTS            Return
370]
38ONEXT pass
```

The program works by finding out whether or not the four bits corresponding to each hex digit represent

a number less than ten. If it is less than ten then the value ASC ("0") is added and the character, which will be a number 0... 9 will be printed. If it is greater than or equal to ten then a number equivalent to ASC ("A-10") is added so that ten will be printed as A, eleven as B etc.

For debugging purposes this program is assembled by typing

```
15 flag = FALSE
RUN
```

The program can then be executed for various values of A% by typing

```
A% = &12 : CALL enter
```

The final version of the program is assembled, without the debugging aids, by typing

```
15 flag = TRUE
RUN
```

11.6 Lower case variable names

As a convention, lower case is used for variable names. Most people consider this to make the code more readable. It also means that there is no chance of using variable names that conflict with BASIC' s keywords e.g. ' print'can be used as a variable name, even though PRINT cannot).

11.7 Constants

Constants are used to give names to numbers which will be used several times throughout a program. They help to make the code easier to understand, e.g.

```
LDX #initial-lives
```

explains far better what is happening than

```
LDX #3
```

Using constants has another advantage - if a value needs to be changed and constants have been used, only the definition of the constant would need to be altered, rather than every occurrence of that value.

Some languages support constants and variables

as totally different data types, and make it impossible to change the value of the constant. BASIC does not treat variables and constants differently (except for it' sown constants, e.g. PI) and so it is up to the programmer to make sure that any constants defined retain their value. One convention used for this is to prefix all constants with a pound (£) sign as a reminder.

11.8 Lookup tables

Lookup tables are useful when converting one value to another. As an example consider the following:

```
LDY index
LDA table, Y
```

In this example the value of A is dependent upon the value of Y. The example consists of a table of values, starting at ' table' .

Note that the above assembler is directly equivalent to the BASIC ' A=table?index' .

The values in ' table'could be the values of a palette, for example

```
.table
        EQUB &3 # Colour 0 is yellow
        EQUB &1 # Colour 1 is red (default)
        EQUB &6 # Colour 2 is cyan
        etc.
```

Thus lookup tables should be used wherever it is necessary to produce a value from another value, but only when there is no simple realationship between the two.

11.9 Use of absolute addresses

A mark of a good assembler program is that it will contain no absolute addresses in the assembler source code. Thus, if a data table starts at location &30F6, a constant should be set up initially to have the value &30F6, and the constant used in the assembler code, not the number &30F6. This follows on from the above points, and also makes the code easier to read and alter.