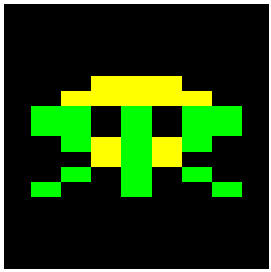


10

LARGE ASSEMBLER PROGRAMS

When writing substantial assembler programs it soon becomes evident that even 32K of memory is insufficient to hold both the assembler source text and the object code produced. This difficulty is heightened still further if the graphics modes are used. The problem can be overcome by breaking the source text into smaller modules or files and this chapter looks at how to set these up and how to use them.

10.1 Source files and the master compiler' program

A source module is a program which acts like a subroutine but has assembly code inside it. The master 'compiler' program reads in each source text module and assembles it. This program is shown below. (Note that anything enclosed within < > (angled brackets) is not to be typed in, but is to be replaced with the value for your application.)

```
 0 REM Compile program
10 origin = <start of area for machine-code>
20 file$ = "ABC"
30 PROCrun("I",<page for source files>)
40 PROCrun("M",<page for macro file> (optional))
```

```
 50 FOR pass = 0 TO 2 STEP 2
 60 P% = origin
 70 FOR files = 1 TO LENfile$
 80 PROCrun(MID$(file$,files,1),<page for
    source files>)
 90 NEXT files
100 NEXT pass
110 PRINT '"Object code from &'';~or igin ''to &;~P%
120 END
130
140 DEFPROC run (name$,start)
150 PRINT name$;
160 OSCLI "LOAD SOURCE"+name$+" "+STR$~start
170 PAGE = start
180 GOSUB 0
190 ENDPROC
```

The files are assumed to be called SOURCEA, SOURCEB, ... i.e. the word ' SOURCE'followed by a single letter. The string ' file$'holds the letters which identify them and hence in this example it contains ' ABC' Since single letters are used, the length of ' file$'gives the number of source programs. Note that in this example ' I'and ' M'should not be used for naming source files since they have their own special uses.

The program starts by reading in and assembling SOURCEI which is the initialisation file to be described later in this chapter. Then the macro file (if one exists) will be treated similarly; again this will be explained later. The main loop takes each of the source files in turn, loads it into the area you have defined as reserved for the source files, and then assembles it. For the first source file the object code starts at the value of the variable ' origin'and P% is used by the assembler as a pointer to the next free byte. Hence this allows subsequent source files to be assembled so that their code follows on directly after that produced by the previous one.

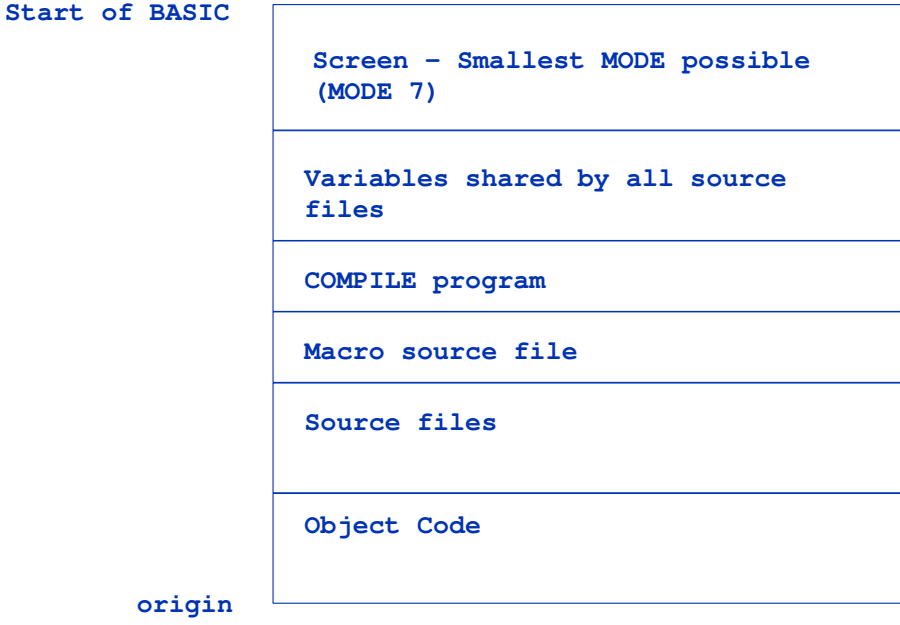
A typical source file would be of the following format:

```
  0 REM SOURCEX
 10
```

```
 20[OPT pass

 30
 .     (Assembler text)
 .
 .
 .
900
910] : RETURN
```

A typical memory map might look like this:



Note that if you are using a 6502 second processor then the screen mode selected will not make any difference.

This method has been designed for use with disc based systems, but can also be used on cassettes if the tape is rewound between the two passes. To remind you of this you should place a ' Rewind Tape' message, together with a ' dummy=GETstatement (to wait for a key to be pressed as an indication that the tape is in the correct position), between the two NEXT statements.

10.2 Saving source files

One routine which is useful when using discs and the above method is a PROCsave routine;

```
  DEF PROCsave OSCLI("SAVE <filename>+ STR$~PAGE +
" "+STR$~TOP)
  ENDPROC
```

Note: this routine will not work on BASIC I. See chapter 9 (BASIC 1, BASIC II and Electron BASIC) for a description of OSCLI and the equivalent BASIC I routine.

The routine should be inserted at the ends of all the source files, and called whenever you wish to save the source file that is in memory at the time.

Thus to SAVE the program all that is needed is to type ' PROCsave' The reason this is so useful is that when editing large numbers of source files which all look alike, it is very easy to overwrite an existing file by typing the wrong name.

An alternative to this, which works on all versions of BASIC, is to type, in immediate mode, ' SAVE $(PAGE+6)' This looks at the first line of the program to find the filename. So, each program should start with

```
  0 REM <filename>
100 <program>
200 ....
```

Default soft keys

Another useful idea is to employ the machine' ssoft keys. This is best done by having a default soft keys program, which can be loaded at the beginning of the session.

```
 10 REM Default Soft Keys
 20
 30 *KEY 0 |LLIST|N|M
 40 *KEY 1 RUN |M
 50 *KEY 2 LOAD"SOURCE
 60 *KEY 3 CALLenter||M
 70 *KEY 5 PPOCsave|M
 80 *KEY 6 PROCfind("
```

```
 90 *KEY 7 MODE7|MPAGE=&6000|ML0AD
    "COMPILE"|M
100 *KEY 9 |L*CAT|M
```

Note that line 80 has a reference to PROCfind. This procedure is to be used from immediate mode to find all occurrences of strings in the current source module. (PROCfind is defined in section 12.6.) This procedure should be at the end of every source file.

10.3 Macro source files

If source files are to be used then calling macros can be a problem. To understand why, some knowledge of how BASIC works is required. When BASIC first comes across a reference to a function or procedure its search for the function or procedure definition starts from PAGE. Once it has found the definition it stores the address of the start of the function or procedure in memory, so that the next time it finds a reference to that particular function or procedure it doesn' thave to waste time searching through the whole program again.

In the simplest case, when just one source file references a given macro, the macro can be added to the end of that source file, and the file treated normally. Consider, however, what would happen if two source files both had a reference to a macro called ' FNfred'and this macro was put at the end of each of them. Since they would almost certainly be different sizes the definition of ' FNfred'would, in each case, start at different addresses. Moreover, when the first source file was assembled the address of the macro in this file would be stored for use by all later references to the macro. Thus, when the second source file tried to use the macro, no searching through would occur. Instead the assembler would jump straight to the address which was stored by the first file and be unlikely to find ' DEFFNfred' starting there.

To avoid this problem a macro file is set up containing all the macros referenced in any of the assembler source files. This is present in the memory all the time, though in a different area of memory to the other files. Each of the macros must be called before the source files are assembled, however, so that the addresses where their definitions may be found

are available to the compiler. Otherwise the first time that the compiler comes across a reference to a macro, e.g. FNfred, it will start searching for 'DEFFNfred', starting at PAGE, look through to the end of the source file, not find the definition, conclude that the macro doesn't exist and report 'No such FN/PROC'. So, set PAGE to the bottom of the macro file and call each macro once.

Thus a typical macro source file looks like this:

```
 0 REM Macro file
10
20 pass = -1 : REM Dummy compilation
30 A% = Fnmacro1(0,0) + FNmacro2(0,0,0,0)..
40 RETURN
50
60 DEF FNmacro(temp, no)
70 IF pass <0 THEN = TRUE
80[OPT pass
90 ...
```

Line 20 sets 'pass' to a value that the assembler will not use, so that the first time the macro is referenced it will not generate any code. Note that every macro in the file must have a test to see if it is being referenced for the first time (as at line 70).

Since the macro file needs to be resident in memory during the compilation a space will have to be assigned for it. This is usually between the top of the normal source files and the bottom of the COMPILE program.

10.4 Initialisation file

The initial file mentioned in COMPILE (line 30) is the file that sets up all variables to be used in the later source files. This is normally in the form:

```
 0 REM SOURCEI
10
20 REM variables
30
-------------------------------------------------------
This is where all the variables that will be accessed
by all the source files are defined. Note that all the
```

```
variables are defined to be resident in memory one
after the other. Thus to move the block of variables
around in memory, all you have to do is to simply
change the value of 'P%' in line 40.
-----------------------------------------------------
 40 P% = <start of memory to put variables>
 50[OPT2 \ Report any errors, but don't list
 60.<first variable> EQUB 0 \ Reserves one byte
 70.<second variable> EQUW 0 \ Reserves two bytes
 80.<third variable> EQUS STRING$(20,CHR$(0))
 90 etc.
  .
200]
-----------------------------------------------------
This section sets up any constants that may be used in
the program. The use of constants in any program is
very important as is explained in chapter 12 (Program
structure).
-----------------------------------------------------
210
220 REM Constants
230
240 limit = 45
250 numberofshapes = 12
260 oswrch = &FFEE
270 etc.
    .
    .
-----------------------------------------------------
This section reserves memory for the data tables.
-----------------------------------------------------

300
310 REM Main Memory Allocations
320
330 P%=<start of memory allocated for tables>
340[OPT2
350.jim
360 OPT FNspace(80)
370.fred
380 OPT FNspace(300)          See section 12.5
390.lenqth                    for 'FNspace'
```

```
400 OPT FNspace(100)
410 etc
    .
    .
-------------------------------------------------------------
This section fills the data tables.
-------------------------------------------------------------
600
610 REM fill data tables
620
630 FOR offset = 0 TO numberofshapes
640 READ offset?length, offset?width, offset?type
650 NEXT offset
660
670 DATA 20,12,3 ,36,24,1 ,etc.
680 etc
    .
    .
999 RETURN
```