14 ASSEMBLER OPERATION

The 68000 assembler described here is a full implementation with many features normally only found in expensive cross-assemblers running on minicomputer equipment. It is purpose designed for use with the Microdrive cartridges and standard QL serial-printer interfaces. Its specification includes:

1. full 2-pass assembly
2. output streaming to screen, printer or Microdrive
3. pseudo-operations (e.g., ORG, COND)
4. assembler directives (e.g., *HEADING)
5. simple expression parsing
6. long label names and local labels
7. alternative mnemonics, and
8. external library file inclusion.

Note that this chapter describes the facilities available within the assembler only. It does not attempt to discuss 68000 instructions.

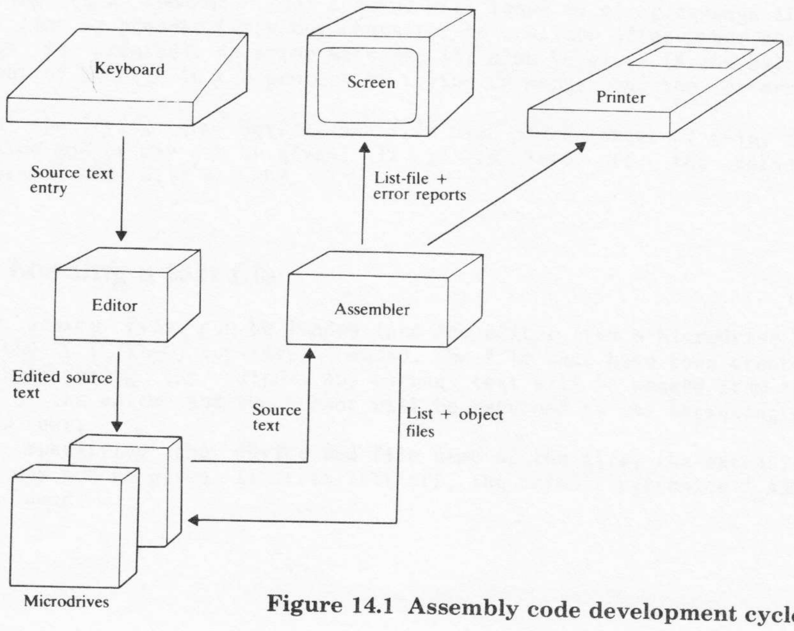

Figure 14.1 Assembly code development cycle

14.1 Assembler operation

The assembler lies at the heart of the assembly language system. It takes its input from a Microdrive file (or some other suitable mass storage medium), and can direct its output either to the screen, a printer, or the mass storage medium. Figure 14.1 illustrates the development cycle. The editor is used first in order to create the source program. This source is then fed to the assembler which creates its various output files. These output files, and in particular the object (binary) file, can then be manipulated in a number of ways. For example, the binary file may be left as it is and accessed by SuperBASIC's LBYTES command. Alternatively its contents could be loaded into memory and then re-saved in the form of an executable file for use with SuperBASIC's EXEC command.

The user manual, which comes with the assembler package, describes in detail the command options available for the assembler, and how the assembler interacts with the editor described in the previous chapter.

14.2 Assembler line syntax

The source input lines for the assembler are single statement lines. Given here is the general syntax of these lines, more detailed explanations being given later under the appropriate headings.

Assembler source input consists of a series of text lines of maximum length 80 characters, created by the editor described in the previous chapter. Each line is of the form:

```
LABEL: OPERATOR ARGUMENT ;COMMENT
```

Any of the four parts - label, operator, argument, or comment - may be omitted where this is appropriate. (Clearly a blank line would contain none of these, and a pure comment line would contain just the fourth element). Items are separated by one or more blanks (spaces or tab characters), the colon following a label, or the semi-colon preceding the comment.

LABELS

Each label name must start with a letter but thereafter may contain any combination of characters, underscores, or digits. No account is taken of case, everything of importance being converted into upper-case internally. Additionally a temporary label may be given (see Sec.14.4).

OPERATORS AND ARGUMENTS

Operators can be 68000 mnemonics (e.g., ADDX, ROR), assembler pseudo-operators (e.g., DEFB, COND), or an assembler directive (e.g., *INCLUDE). The format of the argument parameter will depend upon the operator that precedes it.

COMMENTS

Any line may have a comment appended to aid source documentation. A comment must be preceded by a semi-colon (;). Anything after this comment delimiter will be ignored by the assembler.

THE 'END' PSEUDO-OPERATOR

Assembler source text can optionally be terminated with the END assembler pseudo-operator. If it is not used then the natural end-of-file will be taken as the end of the source text.

14.3 Symbols

Symbols, acting as constants for the duration of the assembly operation, can be defined either from within the source, or dynamically as boolean (true/false) constants at assembly time.

DEFINITION FROM SOURCE (EQU)

Alphanumeric symbols may be defined using the assembler pseudo-operator EQU (or simply an '=' sign):

For example:

```
LETA  EQU  $41          ;'A'
LETB  =    LETA+1       ;'B'
```

The argument following the EQU can be any valid simple expression (as defined later). If an attempt is made to redefine a symbol, an assembler 'M' (Multiple definition) error will ensue - during pass 1 only. If such an error occurs it would be sensible to halt assembly by pressing the ESC key as there may be many future errors, particularly if temporary labels are also being used (which will normally be the case). Upper and lower case are treated as being the same within symbol definitions:

For example:

```
LETC  EQU letb+1     ;'C'
letd  EQU letc+1     ;'D'
LETE  EQU LETD+1     ;'E'
```

Symbols are distinct only within the first eight alphanumeric characters and they must start with an alpha character (A..Z, a..z). If the latter rule is violated an 'L' (Label format) error will ensue.

For example:

```
DELAYforTimer1 = 64
Timer2Delay    = DELAYfor shl 2
```

DEFINITION AT ASSEMBLY TIME (QRY)

If a symbol is defined with the QRY pseudo-operator, the value may be given as either zero (false) by entering N at the keyboard, or as minus one (true) by entering Y. The prompt for the keyboard entry is given at assembly time (during pass 1), as defined by the QRY argument. For example:

```
        FLIST QRY Full listing required
```

will prompt with 'Full listing required?' and expect either a Y or an N as the response. The keyboard entry is immediate (no ENTER required) and the assembler will echo either Y or N as appropriate. Note that keying any letter other than Y will effect an N response. This facility is extremely useful when conditional assembly is being used as it allows the programmer to specify flag values at assembly time, and therefore the source does not have to be edited.

14.4 Labels

There are two types of label which can be used. Alphanumeric labels may be defined which will have a scope of the entire program. Temporary or local numeric labels may also be defined, which will have a scope limited to the area between the two standard labels in which they are defined.

STANDARD LABELS

A normal alphanumeric label is a special kind of symbol. It is declared by ending it with a colon (:), and it will be given the value of the location counter for the current statement. The label itself must obey the same rules as for symbols (i.e., must be alphanumeric, must start with an alpha character, and be significant in its first eight characters).

TEMPORARY (LOCAL) LABELS

Temporary or local variables have a number of important attributes. Each label takes up only one third of the symbol table space required for normal symbols. They do not appear in the symbol table and therefore the table will refer only to important locations, and they may be re-used within different scope blocks thereby greatly reducing the possibility of multi-defined labels.

A local label is defined by the label form '1%' to '255%' and may optionally be followed by a colon (:). A local label may only exist after a normal label has been declared, and its scope of existence is limited up to the next normal label:

```
nlabl:  moveq   #0,d0
        moveq   #delay,dl
1%:     cmp.b   d0,dl
        beq.s   2%
        addq.b  #1,d0
        bra     1%
2%:     rts
;
nlab2:  bra     1%          ;1% is undefined here
2%:     nop
nlab3:
```

During pass two a 'U' (Undeclared symbol) error will ensue if a local label does not exist within its defined scope.

14.5 Expressions

The assembler will accept any non-prioritized simple expression consisting of:

1. symbols
2. normal/local labels
3. denary/hexadecimal numbers
4. single character strings
   (Up-arrow facility, see Sec.14.6, is neither required nor permitted)
5) the operators:

+	Unary plus / Add
-	Unary minus / Subtract
*	Unsigned 16-bit Multiply
/	Unsigned 16-bit Divide
SHR	Shift right ('n' places)
SHL	Shift left ('n' places)
OR	Logical OR
AND	Logical AND
NOT	One's complement

NUMBERS

Numeric values may be defined either in denary or in hexadecimal. If hexadecimal is being used the number must be preceded by an ampersand (&) or a dollar sign ($):

```
For example:    defb 12,45,&3A
                defw $E2,$3AB0
```

If the first digit following a $ or & hexadecimal delimiter is not a valid hexadecimal digit then an 'N' (Number format), or 'S' (Syntax), error will ensue.

SIMPLE EXPRESSIONS

A simple non-prioritized expression is defined in this case to mean any expression of the general form:

<+/-> <operand> (<operator> <operand>)

A unary minus or plus may precede the first operand. Further operator-operand pairs may be used if desired. Expression evaluation is strictly from left to right. The NOT operator is a special case in that only one operand may exist, and this operand must be a symbol or a normal label. An 'I' (Illegal expression) error will ensue if the assembler cannot pass the expression in its context. In most cases this will also be followed by an 'S' (Syntax) error. Some valid examples are:

```
true     =  -1
false    =  not true
days     =  5
;
prog:    moveq  #true and &FF,d0
         moveq  #name and 255,d2
         moveq  #name shr 8,d3
         moveq  #'A',d0
         moveq  #'z'+1,d0
;
         moveq  #''',d0           ;Up-arrow (see 14.6)
         moveq  #'^',d0           ;equivalents, ie:
         moveq  #'A'+$80,d0       ;short form is used.
;
         moveq  #name/256+1,d2
         moveq  #days*24,d3
;
1%:      defb   0                 ;Data store
;
         move.w store,a0
;
store:   defb   0,0
;
mask     = true shl 8 + 1
mask2    = mask or $2020
```

Expression values will take on an 8-bit, 16-bit, or 32-bit value depending upon the context of the expression. Assembler 'O' (Overflow) or 'R' (Range) errors will ensue if it seems that an assignment is out of context (e.g., if a 16-bit value is being used in an 8-bit context). Some assemblers will simply assign the least significant bytes in such cases, which greatly increases the amount of debugging time required when you find out that your program does not work as you intended. For the purposes of conditional assembly, the expression will be deemed true if the most significant bit of the result is set (e.g., -1), or false if this bit is unset (e.g., 0).

14.6 Data definition

Data may be defined by using the following assembler pseudo-operators:

DEFB	-	Define byte / char (8-bit)
DEFW	-	Define word (16-bit)
DEFL	-	Define long-word (32-bit)

Alternatively data storage space may be allocated (but not defined) by using the pseudo-operator:

DEFS	-	Define space (n bytes)

The four data pseudo-operators available enable any form of static data storage to be defined, and may be used in the following ways.

DEFB

This pseudo-operator is used to define byte values and character strings. A free integration of both types is permitted in any one definition line:

```
defb 13,'This is a message',13,0
defb 'ABCDEF'
defb 0,1,2,3,4,5,6,7,8,9
```

Each element of the definition line is separated from the next by a comma (,). If the first character of an element is a single quote, a string of characters is assumed to exist up to, but not including, the next single quote ('). In the context of string definitions the following is also applicable:

1. an up-arrow followed by a single quote will assemble as a single quote: defb '^''
2. an up-arrow followed by an up-arrow will assemble as a single up-arrow: defb '^^'
3. an up-arrow followed by any other character will force the most significant bit of that character to be set: defb '^A'

These special cases may exist anywhere with a string definition:

```
defb 'A^BC'
defb '^'up^''          ;'up' (with quotes)
defb 'A^^2'
```

DEFW AND DEFL

These pseudo-operators force numeric definitions to occupy 16-bits (in the case of DEFW) or 32-bits (in the case of DEFL) whether or not the actual value could reside in an 8-bit location.

```
        defw 34,$56
        defl 900,$4B330,2
```

Strings (as defined under DEFB) may not be defined using these pseudo-operators. Each element in the definition line must be separated from the next by a comma (,).

DEFS

If an area of memory is to be allocated to some use, but the initial values within this area do not need to be specified (e.g., heap storage space), this pseudo-operator may be used. The single argument which must follow this operator will specify the number of bytes to reserve.

14.7 Origin setting

The memory address where the assembled code is to start is defined by the ORG pseudo-operation:

```
            ORG $2A000
```

More than one ORG statement may exist within a program although it would be unwise to define an origin which was lower in memory than the current assembly address. Previously declared labels or symbols may be used within an expression as an argument to ORG. For example, it would be possible to force an ensuing piece of code to reside at a clean page boundary:

```
        current:
        ;
                    ORG current+256 and $FFFFFF00
        ;
        ncode:
```

It is common practice when writing executable code programs and extensions to SuperBASIC, to omit the ORG statement altogether. Assembly will then be based at address zero.

WARNING: Labels and symbols used in ORG expressions **must** be pre-defined. If this is not the case, different origins will exist during pass 1 and pass 2. In such cases the code will fail to assemble properly.

14.8 Conditional assembly

Individual blocks of code may be conditionally assembled using the COND, ELSE, and ENDC pseudo-operators. The operator COND expects an expression as an argument. If the most significant bit of the result is set, the value is deemed true and the following code will be assembled.

Conditional assembly (or non-assembly) of code will continue up until the next ELSE or ENDC operator. If an ELSE operator is found, the condition for assembly is reversed, and the appropriate assembly continued up until the next ENDC operator. The particular level of conditional assembly is terminated on reaching the corresponding ENDC operator.

Conditional assembly may be nested. If pass 1 is completed, but nesting levels for conditional assembly have not been completely matched, a fatal 'Assembler error' will ensue and assembly will cease (i.e., pass 2 will not be entered). A 'C' error will ensue if an ELSE or an ENDC operator is encountered before a corresponding COND operator. Examples of this nesting are as follows:

```
          yes_please     = -1
          no_thank_you   = not yes_please

     1.   cond yes_please
            subx   d2,d0          ;assembled
          else
            subx   d0,d2          ;not-assembled
          endc

     2.   addx   d1,d2              ;level 0
          cond no_thank_you
            addx   d2,d3            ;level 1a
            cond true
              addx   d3,d4          ;level 2a
            else
              subx   d4,d3          ;level 2b
            endc
          else                      ;level 1b
            subx   d3,d2
          endc
          nop                       ;Back to level 0
```

Note that the QRY form of defining symbol values as true or false (described in Sec.14.3), is an extremely useful mechanism for conditional assembly, for example, in cases where slightly different code needs to be generated depending on whether or not the code is to run in ROM. The actual source code need never be changed - it would simply be a matter of entering the appropriate responses at assembly time.

14.9 Directives

The assembler supports a number of assembly directives, invoked by using an asterisk (*) as the first non-blank character in a statement line. The following are supported:

1. *Eject
2. *Heading <string>
3. *List <on/off>
4. *Number <on/off>
5. *Include <filespec>

All of these may be abbreviated to just their first character (for example, *E is the same as *EJECT).

*EJECT AND *HEADING

*Eject causes a form-feed to occur in the list file, and the page number to be increased by one. Any heading, which had previously been defined, remains.

*Heading allows a heading message to be defined which will be used to document page headings in the list file. A form-feed will also occur automatically (as with *E). The maximum length of a heading is 35 characters. Headings longer than this will be truncated.

If one of these two directives is not given before a form-feed is due on a list file (in order to skip over pages in perforated listing paper), then the assembler will force a page throw as and when necessary (normally after 56 lines of assembly listing).

*LIST

*List is used to turn the listing on and off. If the word ON follows the directive then the listing will be turned on. If the word OFF follows the directive then the listing will be turned off. Note that the directive *L ON will have no effect if the list-file device, specified in the original command line, was coded as null (Z). The directive is particularly useful for conditionally listing parts of a large source file. The symbol table is always produced if the list-file is active and therefore one way of getting just a symbol table as the list output is to (conditionally) set the list directive off at the beginning of the source:

```
FLST QRY Full listing required
;
    cond not FLST
*L off
    endc
;
 <Symbol table produced anyway!>
```

*NUMBER

*Number has the same syntax requirements as *List. The directive enables the generation and printing of line numbers within the list file to be switched on and off. The normal state is for line numbers to be given.

*INCLUDE

*Include requires a full file specification as its argument. The specified file will be included in the source input stream at that point in the assembly. This feature enables a suite of library sources to be kept on a Microdrive cartridge and included in a program as and when required.

Only one level of inclusion is allowed and a file will fail to be included if its *I directive is within an already included file. In such cases an 'F' (File inclusion) error will ensue and assembly will continue at the next line in the current source file.

If a file cannot be opened because, for example, the file specification is incomplete or wrong, an error message will be given and assembly will stop. Note that the file specification must be the same as that which would be given to access a Microdrive under SuperBASIC. There are no restrictions on extensions, as is the case within command line specifications.

It is normal practice with large source documents to have one (short) main module which *Includes all other external modules that are required.

14.10 Alternative mnemonics

A set of alternative mnemonics exist within the assembler to aid the programmer both in terms of style and readability. First is the mnemonic for 'exclusive-or' operations. There are two widely used mnemonics for this instruction and both are supported:

Standard	Alternative
EOR	XOR

Second, there is the common confusion, especially with processors which cater for signed and unsigned arithmetic, as to the true interpretation of the 'carry-clear' and 'carry-set' conditional statements. As such the assembler provides the following:

Standard	Alternative
BCC, BCS	BHS, BLO
DBCC, DBCS	DBHS, DBLO
SCC, SCS	SHS, SLO

The mnemonic part 'HS' stands for 'higher or same', and 'LO' stands for 'lower'. They differ from the 'greater or equal' (GE) and 'less than' (LT) mnemonics in that they refer to conditions set after an unsigned operation.

14.11 Error messages

The assembler performs many checks while running and a number of errors and list-file error codes will occur if the source is illegal in some way. The error codes and messages which exist are as follows:

N>	Number format error. A hexadecimal number is illegal.
L>	Label format error. The format of a normal or local label is incorrect.
S>	Syntax error. A catch-all message for lines which contain some form of illegal syntax.
M>	Multiple definition. An attempt is being made to redefine a label or symbol during pass 1.
I>	Illegal expression. The arithmetic or logical expression is illegal within the context given.
U>	Undeclared identifier. During pass 2 a symbol or label is being referenced which was not defined during pass 1.
O>	Overflow / Branch out of range error. A 16-bit value is being assigned to an 8-bit location, or a relative branch is out of range.
C>	Conditional assembly error. An ELSE or ENDC operator was found before a corresponding COND.
F>	File inclusion error. More than one level of file inclusion is being attempted.
R>	Range error. An out-of-limits range is being specified within a particular instruction.

GENERAL ERROR MESSAGES

A few other errors may occur, usually fatal in effect. If a file cannot be opened or a Microdrive cartridge error occurs, an appropriate message is displayed and assembly will cease. If bad conditional assembly exists in pass 1, an error message is displayed and pass 2 is not entered. In all these fatal cases the error message will indicate the nature of the fault.

14.12 Word boundary alignment (ALIGN)

The 68000 processor will always require a word or long-word of data to begin on a word boundary (i.e., an even memory address). This implies that any instruction opcode must also be on a word boundary. When the assembler DEFB or DEFS pseudo-operators are used, the location counter could point to an odd address at the end of the definition line. If a 68000 instruction, DEFW line, or DEFL line immediately follows the definition, the resultant object code will not execute as expected. The 68000 will enter an error type exception process when an attempt is made to access any instruction or word of data at an odd address.

To stop you from having to count byte definitions, in order to make sure there are an even number of bytes defined (and getting it wrong!), the assembler pseudo-operator ALIGN is provided. This operator should follow any byte definition line that must, because of what follows, leave the location counter at an even address. For example:

```
        :
        dat1:       defb 6,'FREEIT'
                    align
        dat2:       defw first,last,max,min
        :
```

If the location counter is incremented internally, to produce alignment, the byte skipped over will be set to zero by the assembler.